

2550 Intro to cybersecurity

L9: Passwords

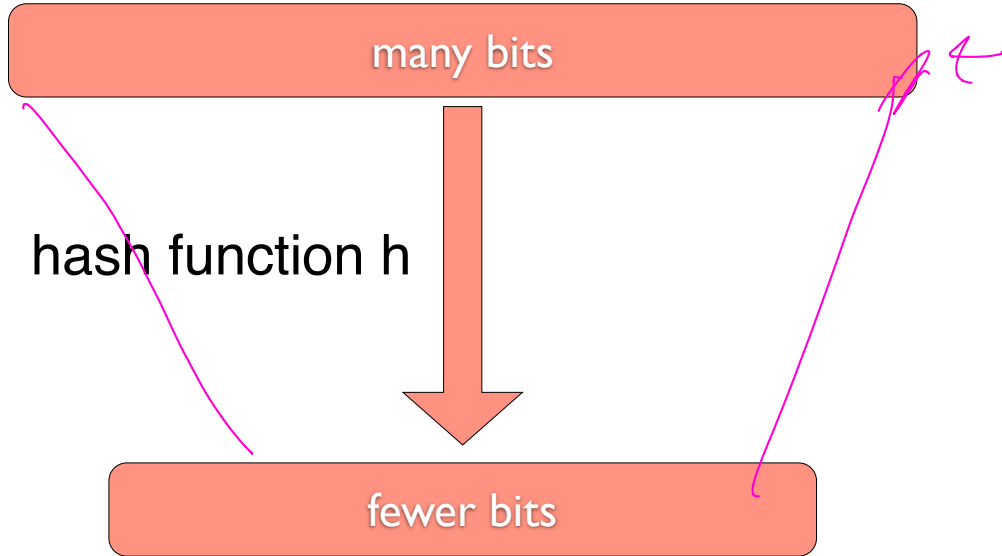
abhi shelat

What is this H() function?

hash function.

$$\text{Sign}_{sk}(m) = \left(\underbrace{0 \dots r \dots ff \dots F \ 0, \ K_{10} \cdot \underline{\underline{H(m)}}}_{\substack{\text{pad the message} \\ \text{hash} \\ \text{message.}}} \right)^{sk}$$

goal of a hash function



a hash function is a function

$$h : \{0, 1\}^d \rightarrow \{0, 1\}^r$$

such that h is easy to evaluate
and $r < d$

b.s

smaller

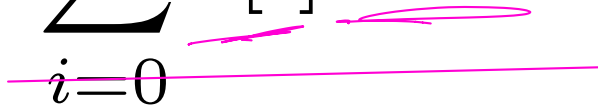
collisions should be rare

```
public class test
{
    public static void main(String[] args)
    {
        System.out.println(args[0].hashCode());
    }
}
```

```
abhi$ java test HHHHHHHHHHHHHHHHHHHHHGGGDD
-1644493785
```

```
abhi$ java test "hello world"
1794106052
```

java hash function

$$h(s) = \sum_{i=0}^n s[i] 31^{n-i}$$


java hash function

$$h(s) = \sum_{i=0}^n s[i] 31^{n-i}$$

it is thus easy to find a pair s_1, s_2
such that $h(s_1) = h(s_2)$

```
public class test
{
    public static void main(String[] args)
    {
        System.out.println(args[0].hashCode());
    }
}
```

```
abhi$ java test HHHHHHHHHHHHHHHHHHHHHHHGGGDD  
-1644493785
```



```
public class test
{
    public static void main(String[] args)
    {
        System.out.println(args[0].hashCode());
    }
}
```

abhi\$ java test HHHHHHHHHHHHHHHHHHHHHGGGDD
-1644493785

abhi\$ java test HHHHHHHHHHHHHHHHHHHHHGGGCC
-1644493785



```
public class test
{
    public static void main(String[] args)
    {
        System.out.println(args[0].hashCode());
    }
}
```

```
abhi$ java test HHHHHHHHHHHHHHHHHHHHHGGGDD
-1644493785
```

```
abhi$ java test HHHHHHHHHHHHHHHHHHHHHGGGCc
-1644493785
```

$$'D' - 'c' + 31('D' - 'C') = 0$$

Collision resistant hash function

CRHF.

dlog

in addition to being easy to compute,
it should be “hard” for a p.p.t. adversary
to find a hash collision.

md4 1990

md5 1992

sha1 1994

sha256 2005

Sha3 2015

md4	1990	128 bit
md5	1992	<u>128 bit</u>
sha1	<u>1994</u>	160 bit
sha256	2005	256 bit
Sha3	2015	

$$2^{128} \sim 2^{64}$$

too weak
birthday paradox.

md4 1990 128 bit 1995

md5 1992 128 bit 1998 - 2004

sha1 1994 160 bit 2005*

sha256 2005 256 bit

Sha3 2015

abhi18:neu abhi\$ shasum -a 256

Noble patricians, patrons of my right,
Defend the justice of my cause with arms.

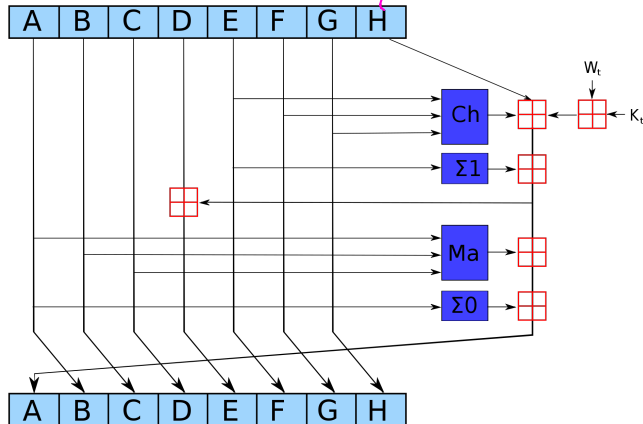
0c3c007b97cf8b75cfbd717804414a6a79b2defb4400eca9ea764a531a9ff193 -

Sha256

Pre-process the input

Break input into chunks

For each "chunk", repeat this 64 times:



Most cryptographers consider SHA256 to be indistinguishable from a “Random oracle”, i.e., a random function on arbitrary length messages.

Recap: ① Perfect Security (Encryption)

Shannon Security

② One-time pad construction

③ Security flaws (key too long)

④ Computational hardness, indistinguishability

⑤ PRGs, Blum-Micali, AES,

⑥ PRFs

⑦ Symmetric Key Encryption, MACs.

⑧ public key crypt: IND-CPA encryption, El Gamal, RSA

⑨ Digital Signatures & Hash Functions-

Passwords

Main problem:

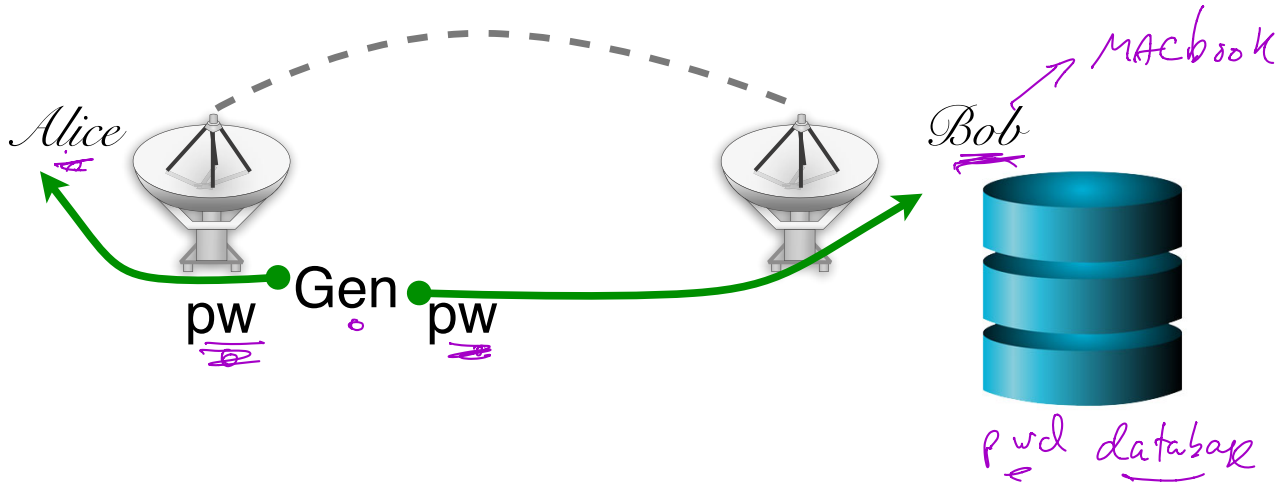
*Alice wants to
authenticate herself to Bob.*



Passwords

pw must be "human" usable.

Main problem:





abhi



Enter Password

Touch ID or Enter Password



Sleep



Restart



Shut Down

Northeastern University Information Technology Services

Welcome to NUwave-guest

Log in to Northeastern's unsecured wireless network NUwave-guest using the username and password you received via text message.

Need to register? [Click here.](#)

One Day Conference Login [Click here.](#)

Have a myNEU login? You must log into NUwave - the secure wireless network.

NUwave-guest Login

Username:

Password:



LTE



Touch ID or Enter Passcode



1

2

3

ABC

DEF

4

5

6

GHI

JKL

MNO

7

8

9

PQRS

TUV

WXYZ

0

Emergency

Cancel

Authentication

- **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 →

Natural authenticators

- (1) pwd : "something you know"
 - (2) "something you have"
 - (3) Biometrics.
property of you that is "unique"
and hard to produce.
- } same.

Operating
Systems

R. Stockton Gaines
Editor

Password Security: A Case History

Robert Morris and Ken Thompson
Bell Laboratories

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

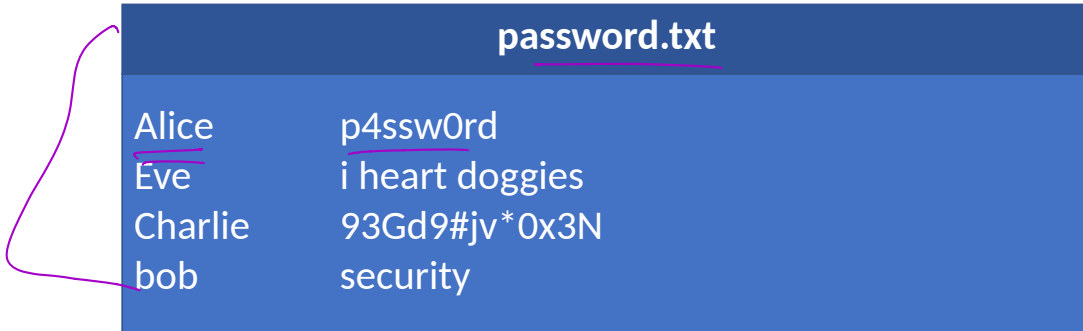
Key Words and Phrases: operating systems, passwords, computer security

CR Categories: 2.41, 4.35

"The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons."

Checking Passwords

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



password.txt	
Alice	p4ssw0rd
Eve	i heart doggies
Charlie	93Gd9#jv*0x3N
bob	security

Attacks against the Password Model

Mallory

① Online brute force attacks

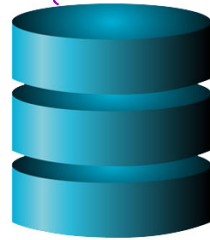
counter measures:

(a) 3 guesses, lockout

(b) pwd hygiene

② Adv steals this file 

Bob



{username: pwd}

password.txt

Alice	p4ssw0rd
Eve	i heart doggies
Charlie	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?

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

RockYou Hack: From Bad To Worse



Nik Cubrilovic




@nikcub / 2:42 am EST • December 15, 2009

Comment



Earlier today news spread that social application site RockYou had suffered a data breached that

resulted in the exposure of over 32 Million user accounts. To compound the severity of the security breach, it was found that **RockYou**  are storing all user account data in plain text in their database, exposing all that information to attackers. RockYou have yet to inform users of the breach, and their blog is eerily silent – but the details of the security breach are going from bad to worse.

Data UserAccount [32603388]



=====

1|jennaplanerunner@hotmail.com|mek*****|myspace|0|bebo.com
2|phdlance@gmail.com|mek*****|myspace|1|
3|jennaplanerunner@gmail.com|mek*****|myspace|0|
5|teamsmackage@gmail.com|pro*****|myspace|1|
6|ayul@email.com|kha*****|myspace|1|tagged.com
7|guera_n_negro@yahoo.com|emi*****|myspace|0|
8|beyootifulgirl@aol.com|hol*****|myspace|1|
9|keh2oo8@yahoo.com|cai*****|myspace|1|
10|mawabiru@yahoo.com|pur*****|myspace|1|
11|jodygold@gmail.com|att*****|myspace|1|
12|aryan_dedboy@yahoo.com|iri*****|myspace|0|
13|moe_joe_25@yahoo.com|725*****|myspace|1|
14|xxxnothingbutme@aol.com|1th*****|myspace|0|
15|meandcj069@yahoo.com|too*****|myspace|0|
16|stacey_chim@hotmail.com|cxn*****|myspace|1|
17|barne1en@cmich.edu|ilo*****|myspace|1|
18|reo154@hotmail.com|ecu*****|myspace|1|
19|natapappaslie@yahoo.com|tor*****|myspace|0|
20|ypiogirl@aol.com|tob*****|myspace|1|
21|brittanyleigh864@hotmail.com|bet*****|myspace|1|myspace.com
22|topenga68@aol.com|che*****|myspace|0|
23|marie603412@yahoo.com|cat*****|myspace|0|
24|mellowchick41@aol.com|chu*****|myspace|0|

Operating
Systems

R. Stockton Gaines
Editor

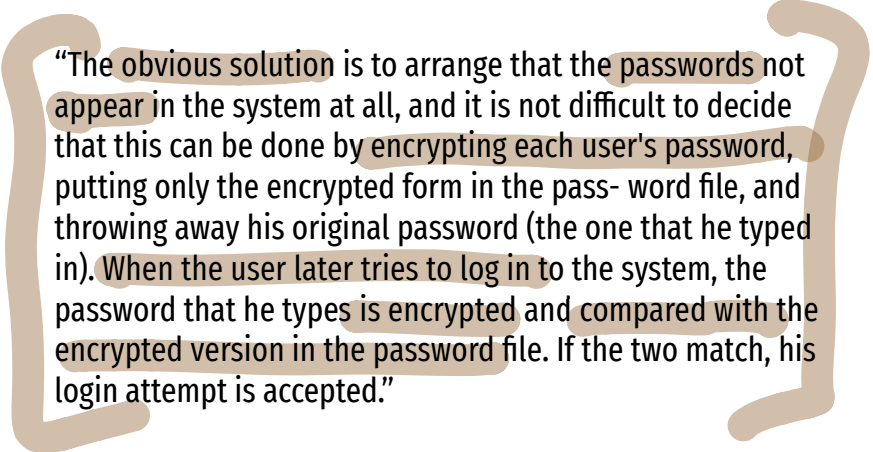
Password Security: A Case History

Robert Morris and Ken Thompson
Bell Laboratories

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

Key Words and Phrases: operating systems, passwords, computer security

CR Categories: 2.41, 4.35



“The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted.”

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 (hopefully)
 - Example: 2^{21} tries for md5

Hashed Password Example



User: Charlie

hashed_password.txt

charlie	2a9d119df47ff993b662a8ef36f9ea20
greta	23eb06699da16a3ee5003e5f4636e79f
alice	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

Hashed Password Example


User: Charlie

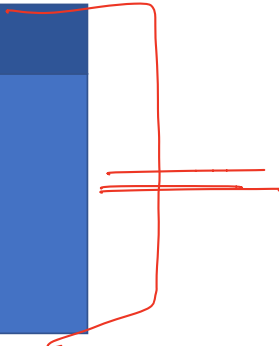


MD5('p4ssw0rd') =

2a9d119df47ff993b662a8ef36f9ea20



<u>hashed_password.txt</u>	
<u>charlie</u>	<u>2a9d119df47ff993b662a8ef36f9ea20</u>
greta	23eb06699da16a3ee5003e5f4636e79f
alice	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9



Hashed Password Example



User: Charlie



MD5('p4ssw0rd') =
2a9d119df47ff993b662a8ef36f9ea20

hashed_password.txt

charlie	2a9d119df47ff993b662a8ef36f9ea20
greta	23eb06699da16a3ee5003e5f4636e79f
alice	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

Hashed Password Example



User: Charlie

MD5('p4ssw0rd') =

2a9d119df47ff993b662a8ef36f9ea20



MD5('2a9d119df47ff993b662a8ef36f9ea20')

= b35596ed3f0d5134739292faa04f7ca3

hashed_password.txt

charlie	2a9d119df47ff993b662a8ef36f9ea20
greta	23eb06699da16a3ee5003e5f4636e79f
alice	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

Offline

brute

force

attack

Hashed Password Example



User: Charlie

MD5('p4ssw0rd') =

2a9d119df47ff993b662a8ef36f9ea20



MD5('2a9d119df47ff993b662a8ef36f9ea20')

= b35596ed3f0d5134739292faa04f7ca3



hashed_password.txt

charlie	2a9d119df47ff993b662a8ef36f9ea20
greta	23eb06699da16a3ee5003e5f4636e79f
alice	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9

Attacking Password Hashes

- Recall: cryptographic hashes are collision resistant
 - Locating A' such that $\text{hash}(A) = \text{hash}(A')$ takes a long time (hopefully)
- Are hashed password secure from cracking?

Attacking Password Hashes

- Recall: cryptographic hashes are collision resistant
 - Locating A' such that $\text{hash}(A) = \text{hash}(A')$ takes a long time (hopefully)
- 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**

offline
brute
force
attacks

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time,

- 15 were a single ASCII character;
- 72 were strings of two ASCII characters;
- 464 were strings of three ASCII characters;
- 477 were strings of four alphanumerics;
- 706 were five letters, all upper-case or all lower-case;
- 605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831 or 86 percent of this sample of passwords fell into one of these classes.

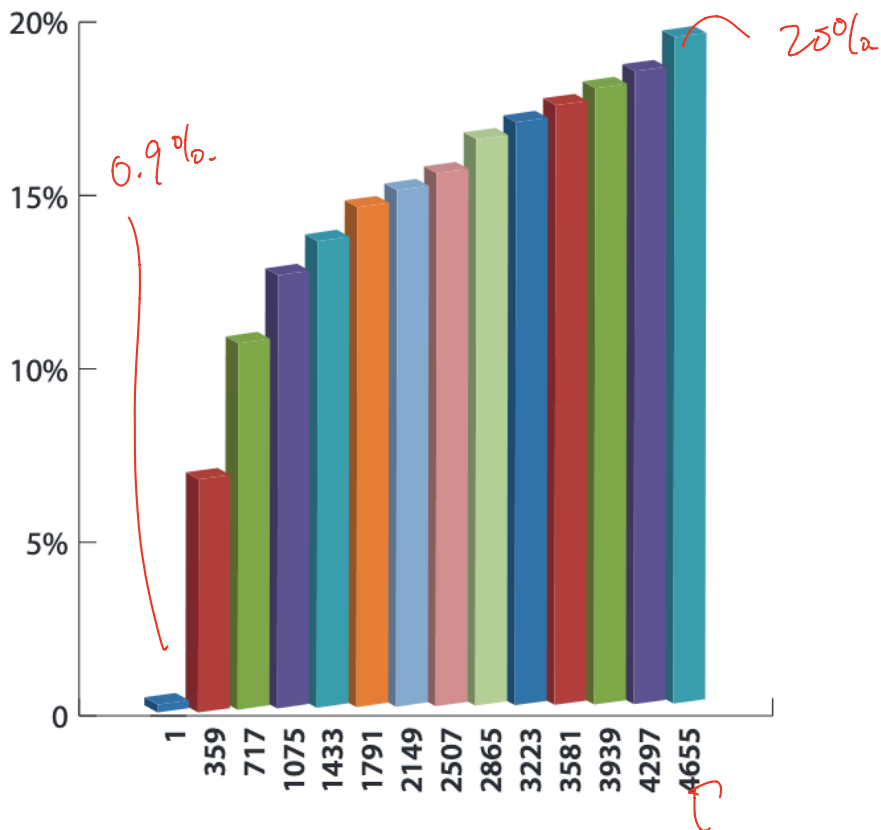
1979

From Rockyou breach

Rank	Password	Number of Users with Password (Absolute)
1	123456	290731
2	12345	79078
3	123456789	76790
4	Password	61958
5	iloveyou	51622
6	princess	35231
7	rockyou	22588
8	1234567	21726
9	12345678	20553
10	abc123	17542

Password Popularity—Top 20

Rank	Password	Number of Users with Password (Absolute)
11	Nicole	17168
12	Daniel	16409
13	babygirl	16094
14	monkey	15294
15	Jessica	15162
16	Lovely	14950
17	michael	14898
18	Ashley	14329
19	654321	13984
20	Qwerty	13856

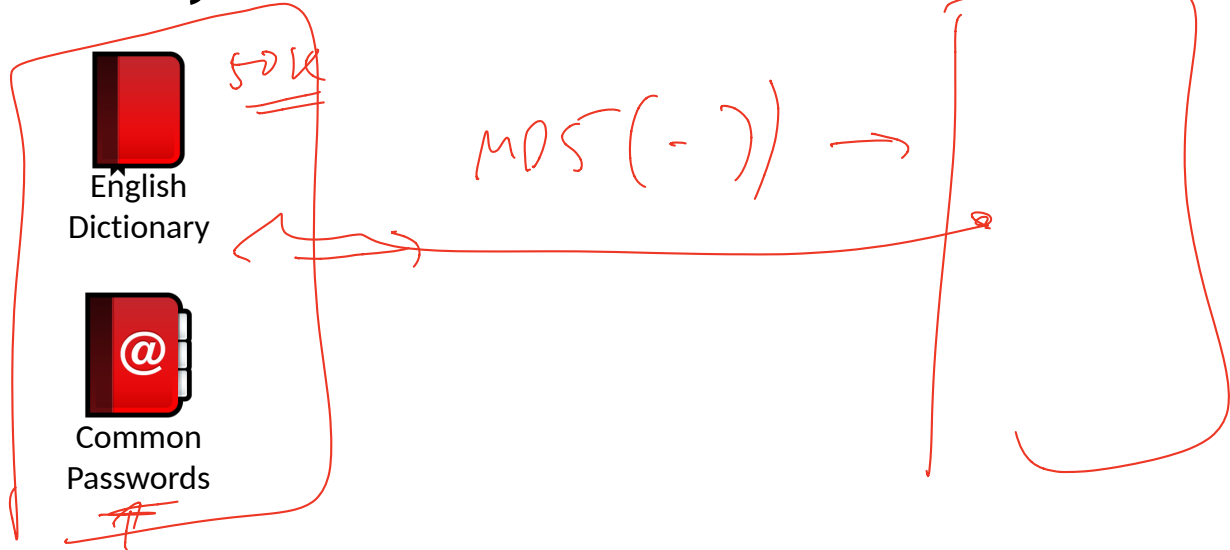


Accumulated Percent of Dictionary Attack Success

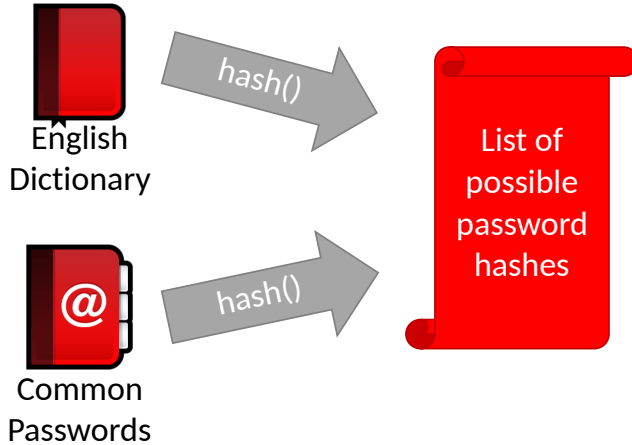
Most Common Passwords

Rank	2013	2014
1	123456	123456
2	password	password
3	12345678	12345
4	qwerty	12345678
5	abc123	qwerty
6	123456789	123456789
7	111111	1234
8	1234567	baseball
9	iloveyou	dragon
10	adobe123	football

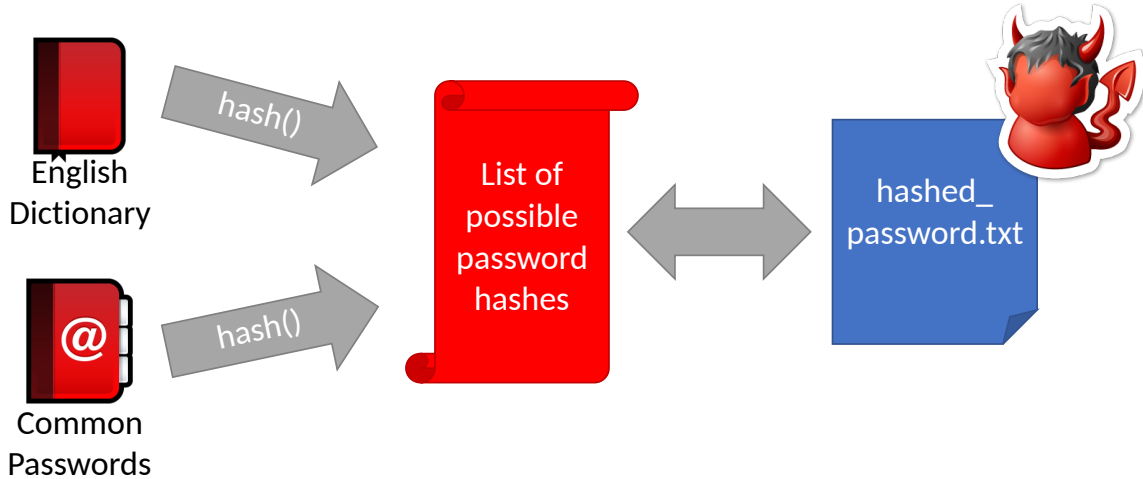
Dictionary Attacks



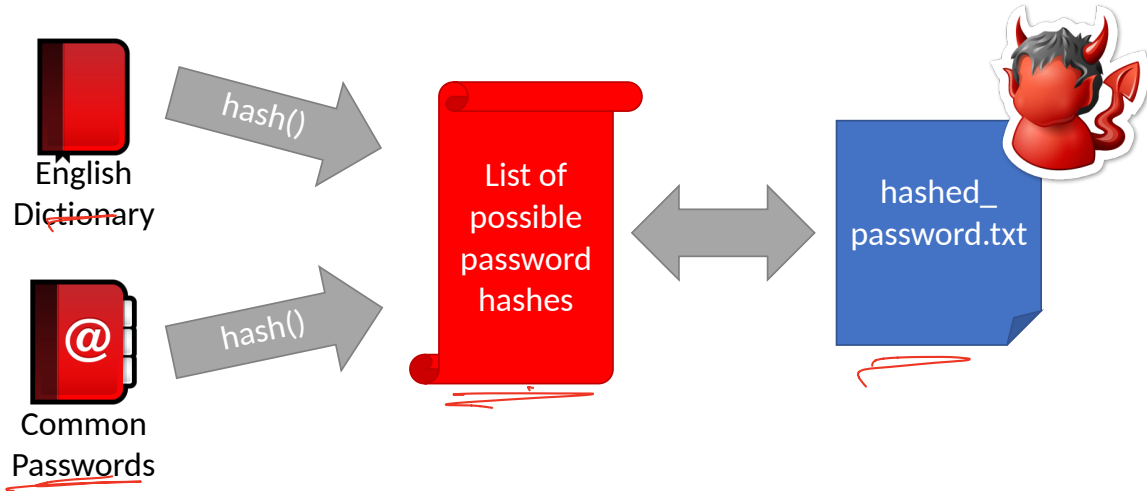
Dictionary Attacks



Dictionary Attacks



Dictionary Attacks



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

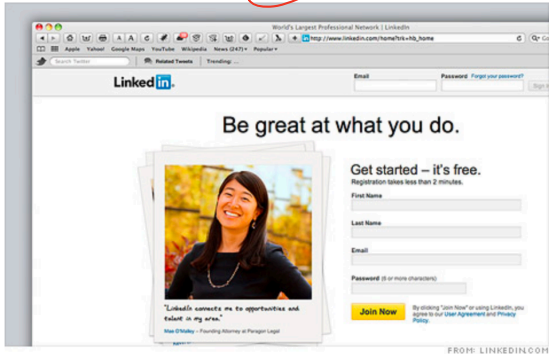
Pwd breaches

CNN Money

THE CYBERCRIME ECONOMY

More than 6 million LinkedIn passwords stolen

By David Goldman @CNNMoneyTech June 7, 2012: 9:34 AM ET



Researchers say a stash of what appear to be LinkedIn passwords were protected by a weak security scheme.

NEW YORK (CNMMoney) -- Russian hackers released a giant list of passwords this week, and on Wednesday security researchers identified their likely source: business social networking site LinkedIn.

2012: 6.5 million hashes leaked onto Internet 90% cracked in 2 weeks

2016: 177.5 million more hashes leaked 98% cracked in 1 week



MDS(→)

2012 LinkedIn Breach had 117 Million Emails and Passwords Stolen, Not 6.5M

May 18, 2016



Long time users of LinkedIn users may very well need to change their passwords once more



Related Posts

- Web Skimming Attack on Blue
- Bear Affects School Admin

by Paul Ducklin



3 DES

One month ago today, we wrote about Adobe's [giant data breach](#).

As far as anyone knew, including Adobe, it affected about 3,000,000 customer records, which made it sound pretty bad right from the start.



But worse was to come, as recent updates to the story bumped the number of affected customers to a [whopping 38,000,000](#).

We took Adobe to task for a lack of clarity in its breach notification.

OUR COMPLAINT

One of our complaints was that Adobe said that it had lost *encrypted* passwords, when we thought the company ought to have said that it had lost

```

4464 [User ID] yahoo.com|-g2B6PhWEH36 [Password hint] try: qwerty123 --
4465 |--|xxxxx@jcom.home.ne.jp|-Eh5tLomK+N+82csoVwU9bw==|-|????| --
4466 |--|xx@hotmail.com|-ahw2b2BELzGRTWYvQgn+kw==|-|quiero a...|--
4467 |--|xxx@yahoo.com|-leMTCMPEPcjioxG6CatHBw==|-| --
4468 |--| [Username] [Username] e.com|-2GthVrmsFRZioxG6CatHBw==|-| --
4469 |--|xxxxxyahoo.com|- [4LSlo772tH4] [Password data (base64)] --
4470 |--|xxx@hotmail.com|- [xwz2SizR8uioxG6CatHBw==|-| --
4471 |--|xxxx@yahoo.com [Email address] xG6CatHBw==|-|myspace|--
4471 |--|xxx@hotmail.com|-kby1918wDrrioxG6CatHBw==|-|regular|--

```

Adobe password data	Password hint
110edf2294fb8bf4	-> numbers 123456
110edf2294fb8bf4	-> ==123456
110edf2294fb8bf4	-> c'est "123456"
8fda7e1f0b56593f e2a311ba09ab4707	-> numbers
8fda7e1f0b56593f e2a311ba09ab4707	-> 1-8
8fda7e1f0b56593f e2a311ba09ab4707	-> 8digit
2fca9b003de39778 e2a311ba09ab4707	-> the password is password
2fca9b003de39778 e2a311ba09ab4707	-> password
2fca9b003de39778 e2a311ba09ab4707	-> rhymes with assword
e5d8efed9088db0b	-> q w e r t y
e5d8efed9088db0b	-> ytrewq tagurpidi
e5d8efed9088db0b	-> 6 long qwert
ecba98cca55eabc2	-> sixxone
ecba98cca55eabc2	-> 1*6
ecba98cca55eabc2	-> sixones

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

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

Example Salted Hashes

hashed_password.txt	
cbw	2a9d119df47ff993b662a8ef36f9ea20
sandi	23eb06699da16a3ee5003e5f4636e79f
amislove	98bd0ebb3c3ec3fbe21269a8d840127c
bob	e91e6348157868de9dd8b25c81aebfb9



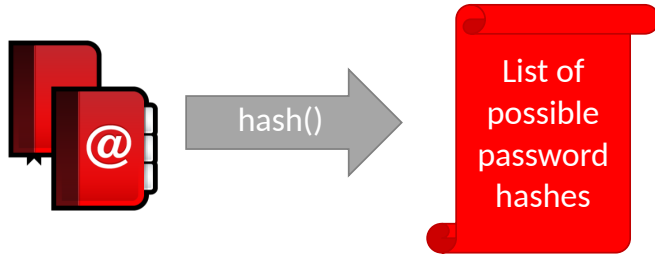
Stolen!!



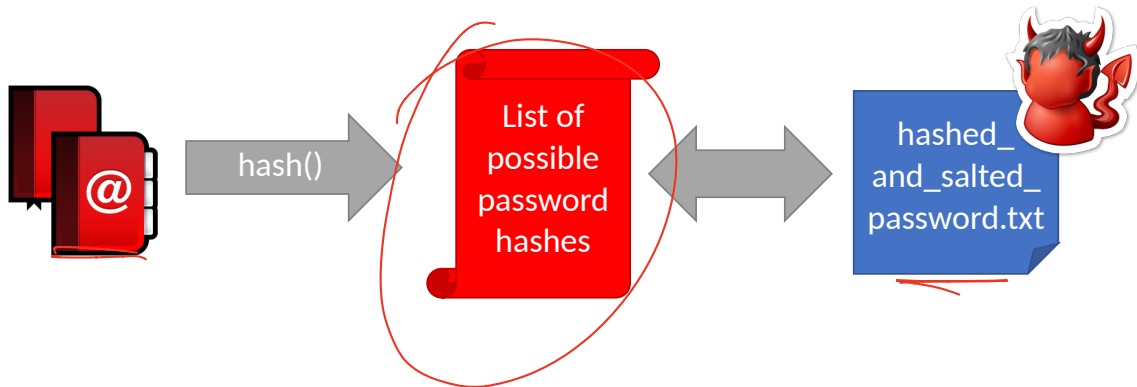
hashed_and_salted_password.txt		
cbw	a8	af19c842f0c781ad726de7aba439b033
sandi	0X	67710c2c2797441efb8501f063d42fb6
amislove	hz	9d03e1f28d39ab373c59c7bb338d0095
bob	K@	479a6d9e59707af4bb2c618fed89c245

$h(\text{salt} \parallel \text{pwd})$

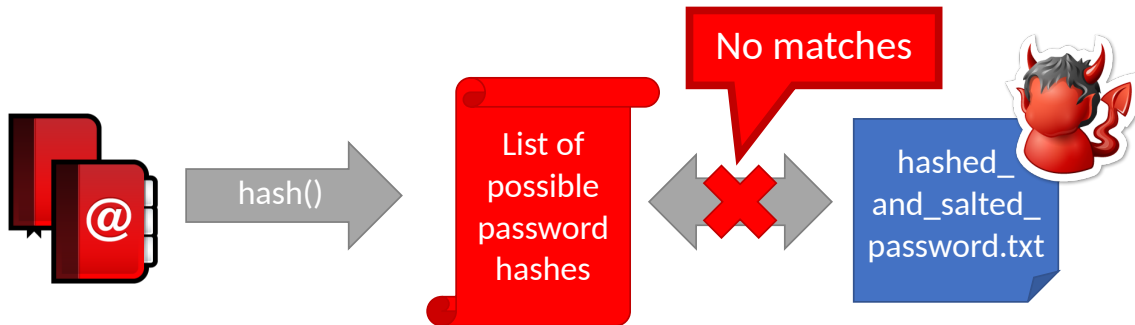
Attacking Salted Passwords



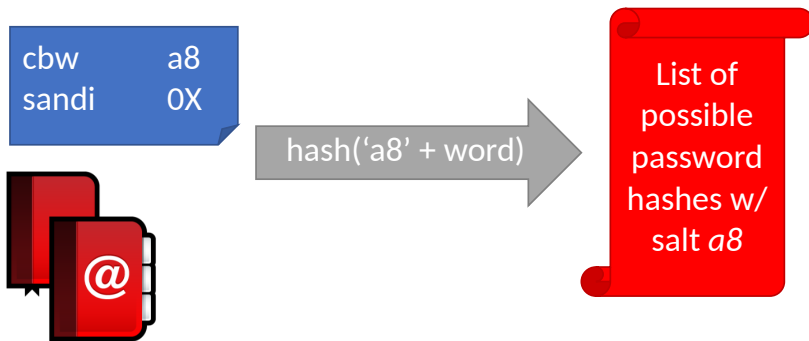
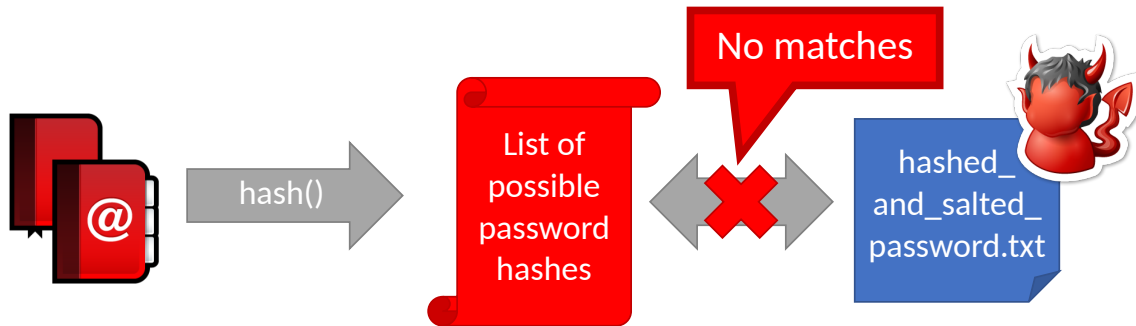
Attacking Salted Passwords



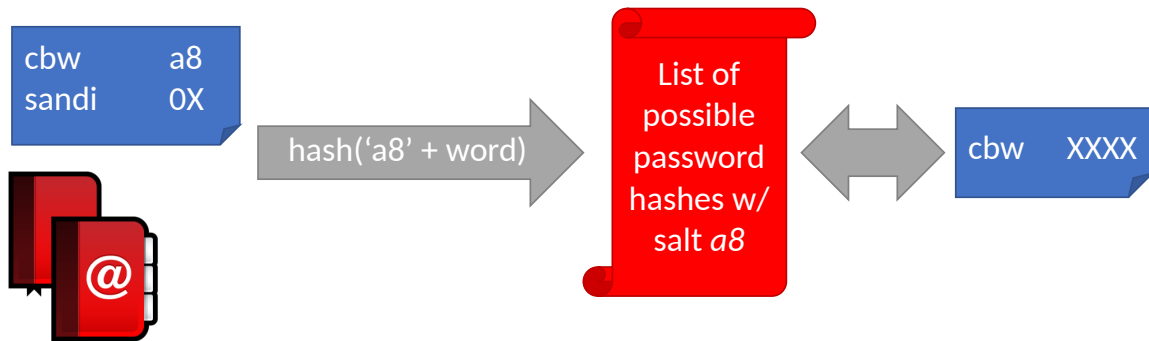
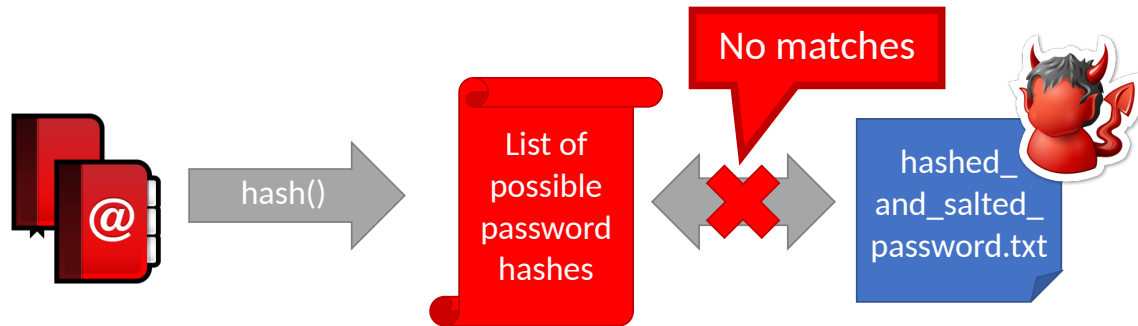
Attacking Salted Passwords



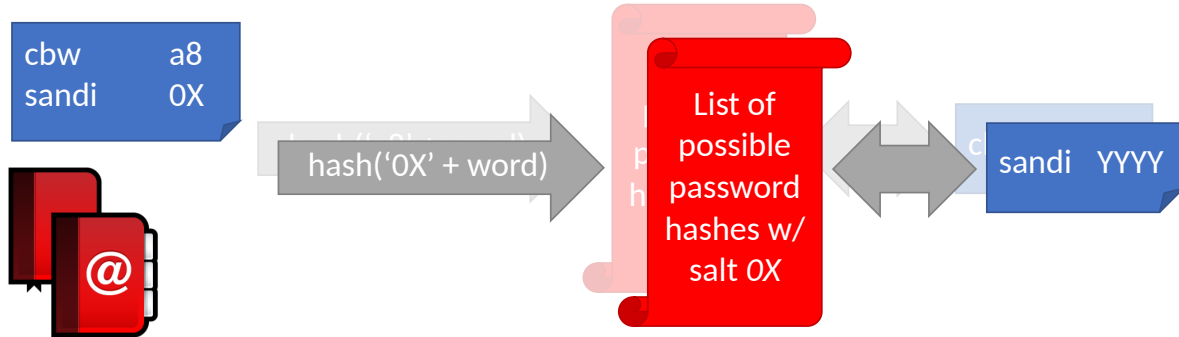
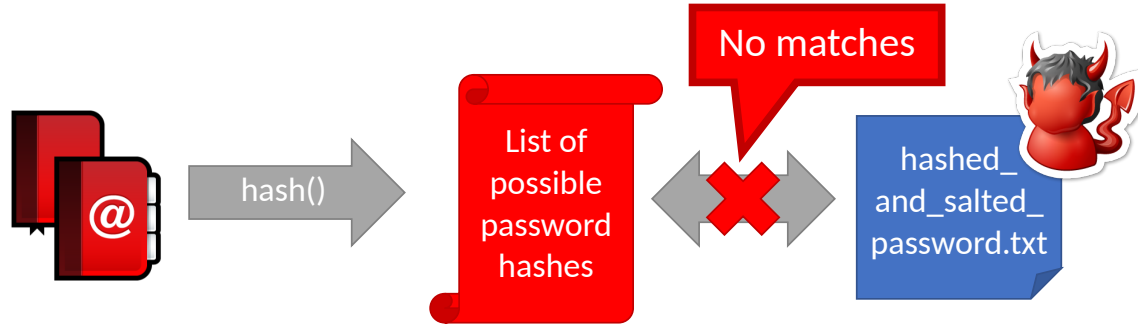
Attacking Salted Passwords



Attacking Salted Passwords



Attacking Salted Passwords



Breaking Hashed Passwords

- **Stored passwords should always be salted**
 - Forces the attacker to brute-force each password individually

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

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

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 $\rightarrow (26+10)^6 = 2\text{B}$ 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

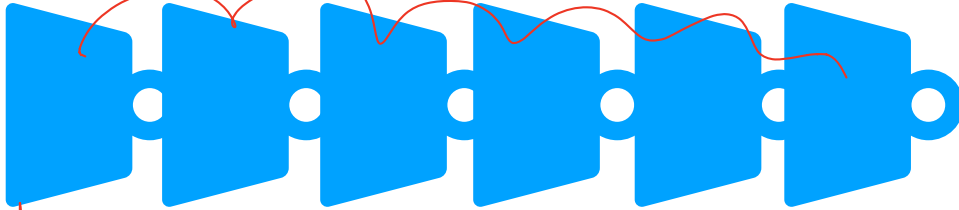
Slow hash movement



slow hash design

Iterated hash function {x times}

Pw
Salt



Hashed pwd

bcrypt Example

- Python example; install the *bcrypt* package

```
[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

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

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

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 doesn't 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

Database



Honeyserver



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	User	Index
cbw	aB	y4DvF7	fl	bH	cbw	2
					sandi	3

Honeywords Example



cbw

Database



Honeyserver



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	User	Index
cbw	aB	y4DvF7	fl	bH	cbw	2
					sandi	3

Honeywords Example



cbw

SHA512("fl" | "p4ssW0rd") → bHDJ8I

Database



Honeyserver



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	User	Index
cbw	aB	y4DvF7	fl	bHcbw	cbw	2
					sandi	3

Honeywords Example



cbw

SHA512("fl" | "p4ssW0rd") → bHDJ8I

Database



User	Salt 1	H(PW 1)	Salt 2	H(User)	Index
cbw	aB	y4DvF7	fl	bHcbw	2
				sandi	3

Honeyserver



Honeywords Example



cbw



SHA512("fl" | "p4ssW0rd") → bHDJ8I

Database



Honeyserver



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	User	Index
cbw	aB	y4DvF7	fl	bHcbw	cbw	2
					sandi	3

Honeywords Example



cbw



Cracked Passwords

User	PW 1	PW 2
cbw	123456	p4
sandi	puppies	ilov
amislove	coff33	3s

SHA512("fl" | "p4ssW0rd") → bHDJ8I

Database



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	Index
cbw	aB	y4DvF7	fl	bHcbw	2
				sandi	3

Honeyserver



Honeywords Example



cbw



Cracked Passwords

User	PW 1	PW 2
cbw	123456	p4
sandi	puppies	ilov
amislove	coff33	3s

SHA512("fl" | "p4ssW0rd") → bHDJ8I

Database



User	Salt 1	H(PW 1)	Salt 2	H(PW 2)	Index
cbw	aB	y4DyF7	fl	bH-cbw	2
				sandi	3

Honeyserver



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**
 - 4. Implement honeywords to detect breaches**
- These rules apply to any system that needs to authenticate users
 - Operating systems, websites, etc.

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: its possible to crack password reset

Cracking Password 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

Cracking Password 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?

Cracking Password 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

Cracking Password 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

Password Cracking

Password Theory

Hash Chains

Rainbow Tables

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



I wanna login to those user accounts!

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



I wanna login to those user accounts!

Cracked Passwords

User	Password
cbw	p4ssW0rd

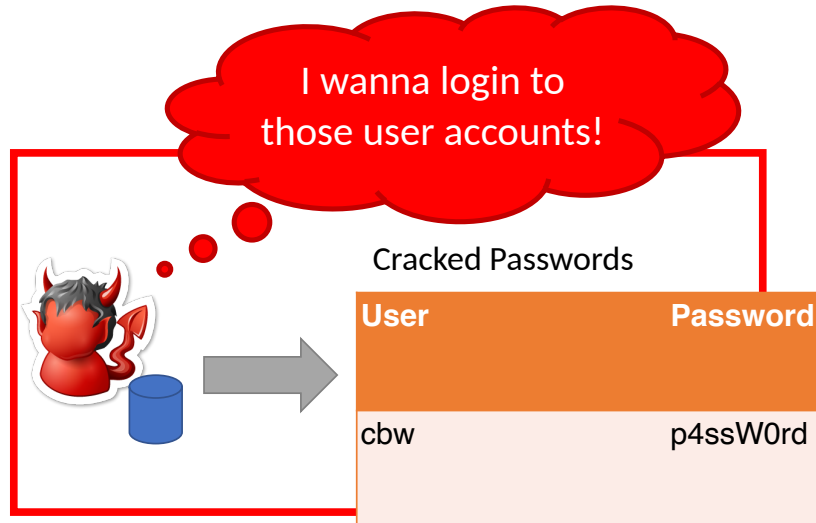
Attacker Goals and Threat Model

- 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



Password Quality

$$S = \log_2 N^L \rightarrow L = \frac{S}{\log_2 N}$$

- How do we measure password quality? **Entropy**
 - 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 ...

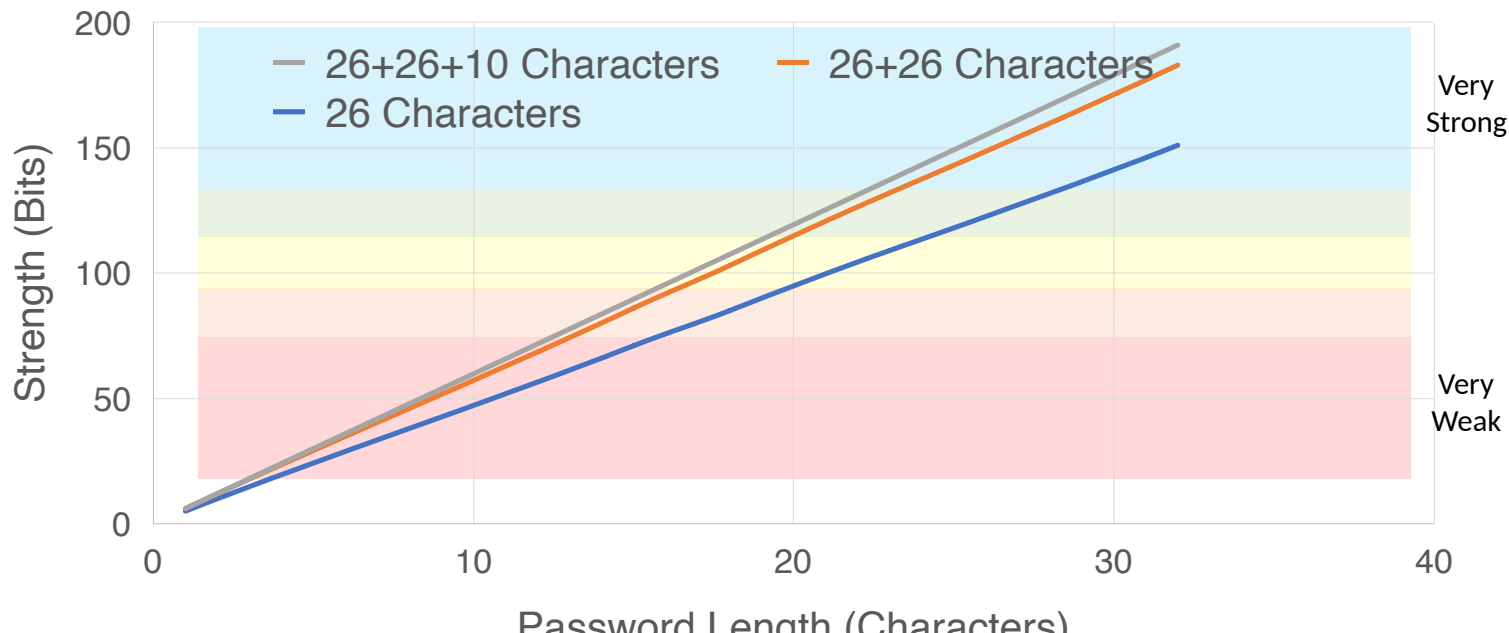
Password Quality

$$S = \log_2 N^L \rightarrow L = \frac{S}{\log_2 N}$$

- How do we measure password quality? **Entropy**
 - 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?

The Strength of Random Passwords

$$S = L * \log_2 N$$

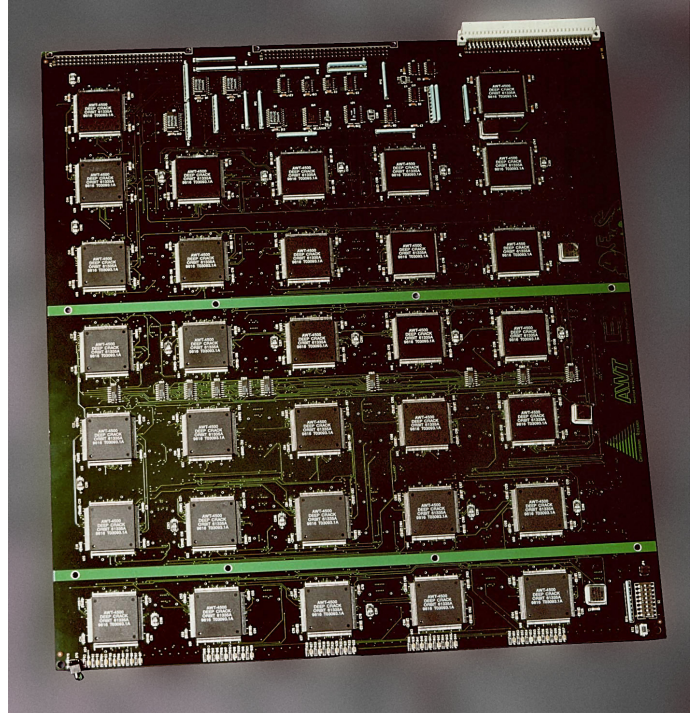


Basic Password Cracking

- 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 → password1 → etc.
- Many password cracking tools exist (e.g. John the Ripper, hashcat)

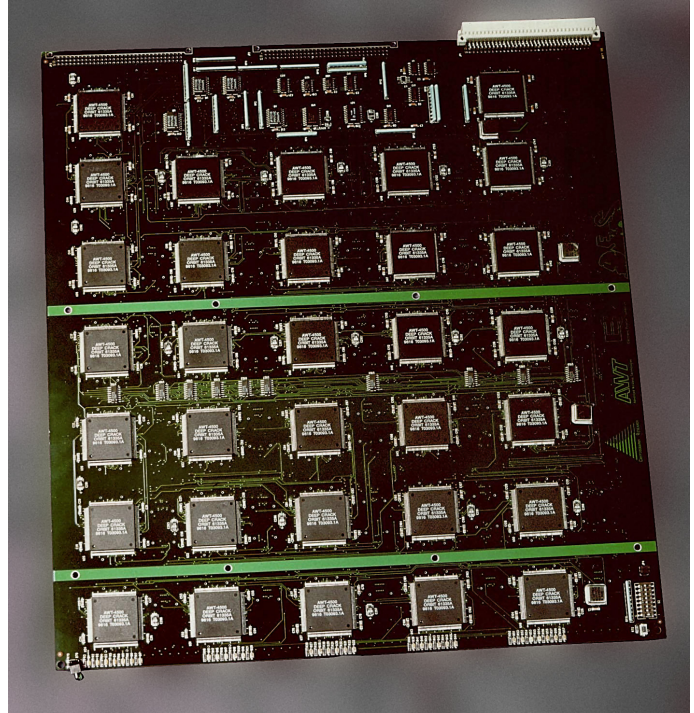
“Deep Crack”: The EFF DES Cracker

- 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



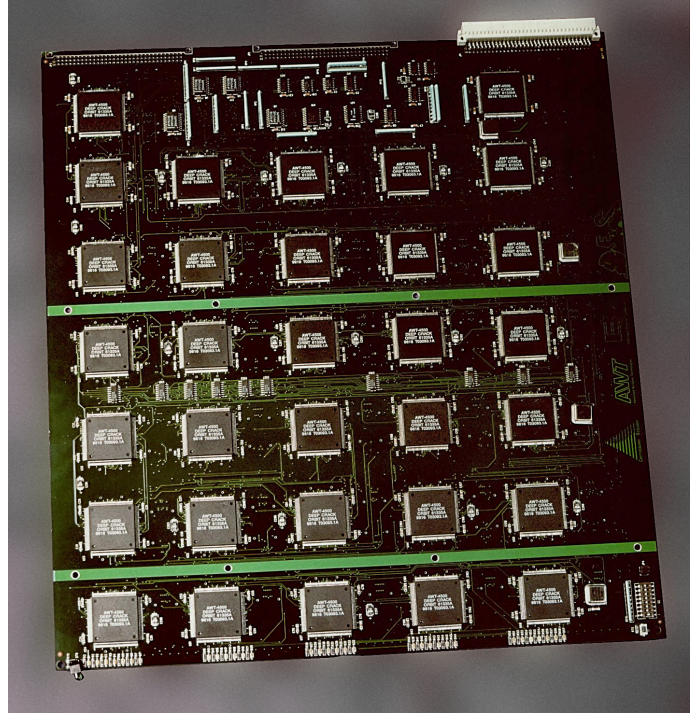
“Deep Crack”: The EFF DES Cracker

- 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?



“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

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 $H(p) = h$

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

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

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
- 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

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') = h''$
abcde	\\WPNP_	vlsfqp	_QOZLR
passw	VZDGEF	gfnxsk	ZLGEKV
12345	SM-QK\9	sawtzg	RHKP_D

$K = 3$

Uncompressed Hash Chain Example

Only these two columns get stored on disk

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') = h''$
abcde	\\WPNP_	vlsfqp	_QOZLR
passw	VZDGEF	gfnxsk	ZLGEKV
12345	SM-QK\9	sawtzg	RHKP_D

$K = 3$

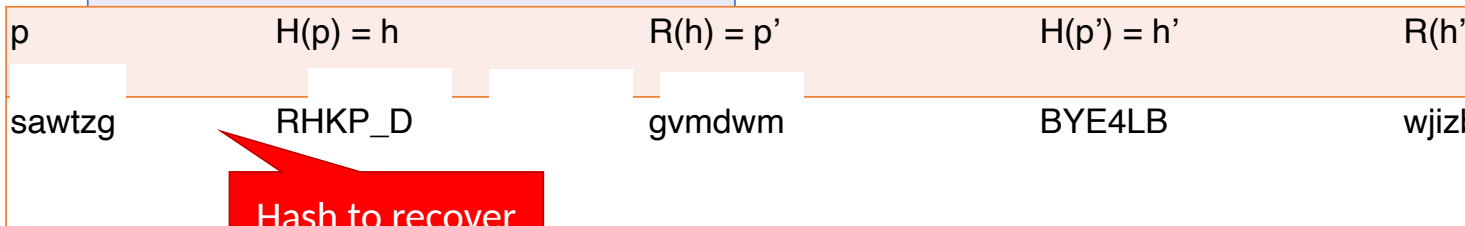
Hash Chain Example

p'	p^*
abcde	cjldar
$K = 3$	
passw	zvxscs
12345	wjizbn

p	$H(p) = h$	$R(h) = p'$	$H(p') = h'$	$R(h')$
sawtzg	RHKP_D	gvmdwm	BYE4LB	wjizbn

Hash Chain Example

p'	p^*
abcde	cjldar
$K = 3$	
passw	zvxscs
12345	wjizbn



Hash Chain Example

p'	p^*
abcde	cjldar
$K = 3$	
passw	zvxscs
12345	wjizbn

p	$H(p) = h$	$R(h) = p'$	$H(p') = h'$	$R(h')$
sawtzg	RHKP_D	gvmdwm	BYE4LB	wjizbn

Desired password

Hash to recover

Hash Chain Example

p'	p^*
abcde	cjldar
passw	vxscs
12345	wjizbn

$K = 3$

p	$H(p) = h$	$R(h) = p'$	$H(p') = h'$	$R(h')$
sawtzg	RHKP_D	gvmawm	BYE4LB	wjizbn

Desired password

Hash to recover

Hash Chain Example

p'	p^*
abcde	cjldar
passw	[red box] vxscs

$K = 3$

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') =$
12345	SM-QK9	sawtzg	RHKP_D
sawtzg	[red box]		

Desired password

Hash to recover

Hash Chain Example

p'	p^*
abcde	cjldar
passw	<input type="text"/>

$K = 3$

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') =$
	<input type="text"/>		

p	12345	SM-QK9	sawtzg	RHKP_D
sawtzg		<input type="text"/>		

Desired password

Hash to recover

Hash Chain Example

p'	p^*
abcde	cjldar
passw	[redacted]

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

$K = 3$

p'	$H(p') = h'$	$R(h') = p''$	$H(p'') =$
12345	SM-QK9	sawtzg	RHKP_D
sawtzg	[redacted]		

p
sawtzg

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)

Final Thoughts on 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**
 - Precomputed tables can be bought or downloaded for free