

Lecture: #07
Attacks on Passwords



Dr. Hesham Abou Elfetouh
Cybersecurity Department
Faculty of Informatics
Midocean University





This chapter covers the following topics and concepts: **Attacks on Passwords**

Password Selection Criteria

One-Time Passwords

**Challenge-response
Authentication**

Attacks on Passwords



In Previous Lecture you learned:

- ❖ User authentication
 - Password authentication
 - Access control
 - Probable Passwords
 - Passwords Likely for a User



When you complete this chapter, you will be able to:

- Describe the Password Selection Criteria
- Explain the One-Time Passwords
- Describe the Advantages of salt
- Explain the Attacks on Passwords



- Use characters other than just AZ.
 - If passwords are chosen from the letters AZ, there are only 26 possibilities for each character.
 - Adding digits expands the number of possibilities to 36.
 - Using both uppercase and lowercase letters plus digits expands the number of possible characters to 62.
 - It takes about 100 hours to test all 6-letter words chosen from letters of one case only, but it takes about 2 years to test all 6-symbol passwords from upper- and lowercase letters and digits.



- Choose long passwords

- Choosing longer passwords makes it less likely that a password will be uncovered.
- Remember that a brute force penetration can stop as soon as the password is found.
- Some penetrators will try the easy cases known words and short passwords and move on to another target if those attacks fail.

- Avoid actual names or words

- There are 266 or about 300 million 6-letter "words", but there are only about 150,000 words in a good dictionary.



- Choose an unlikely password

- To remember the password easily, you want one that has special meaning to you
- you don't want someone else to be able to guess this special meaning.
- The first letters of words from a song, a few letters from different words of a private phrase, or a memorable football team are examples of reasonable passwords.
- Password-cracking tools also test replacements of 0 (zero) for o or O (letter "oh") and 1 (one) for l (letter "ell") or \$ for S (letter "ess").



- **Change the password regularly**
 - Even if there is no reason to suspect that the password has been compromised, change is advised.
 - **Don't write it down**
- **Don't tell anyone else**
 - The easiest attack is social engineering, in which the attacker contacts the system's administrator or a user to elicit the password in some way.
 - Under no circumstances should you ever give out your private password.



User education

users can be told the importance of using hard to guess passwords and can be provided with guidelines for selecting strong passwords

Computer generated passwords

users have trouble remembering them

Reactive password checking

system periodically runs its own password cracker to find guessable passwords

Proactive password checking

user is allowed to select their own password, however the system checks to see if the password is allowable, and if not, rejects it

goal is to eliminate guessable passwords while allowing the user to select a password that is memorable



- A **one-time password** is one that changes every time it is used.
- Instead of assigning a **static phrase** to a user, the system assigns a **static mathematical function**.
- The system provides an argument to the function, and the user computes and returns the function value.



➤ $f(x) = x + 1$.

- The system prompts with a value for x , and the user enters the value $x + 1$.
- The kinds of mathematical functions used are limited only by the ability of the user to compute the response quickly and easily.
- Other similar possibilities are:
 - $f(x) = 3x^2 - 9x + 2$.
 - $f(x) = d * h$, where d is the date and h is the hour of the current time.



➤ $r(x) = r(x)$

- The receiver uses the argument as the seed for a random number generator.
- The user replies with the value of the first random number generated.
- A variant of this scheme uses x as a **number of random numbers** to generate.
- The receiver generates x random numbers and sends the X^{th} of these to the host.



➤ $f(a_1a_2a_3a_4a_5a_6) = a_3a_1a_1a_4$

- The system provides a character string, which the user must transform in some predetermined manner.

➤ $f(E(x)) = E(D(E(x)) + 1)$

- In this function, the computer sends an encrypted value, $E(x)$.
- The user must decrypt the value, perform some mathematical function, and encrypt the result to return it to the system.



Goal: Bob wants Alice to “prove” her identity to him

Protocol ap1.0: Alice says “I am Alice”



Failure scenario??



Goal: Bob wants Alice to “prove” her identity to him

Protocol ap1.0: Alice says “I am Alice”

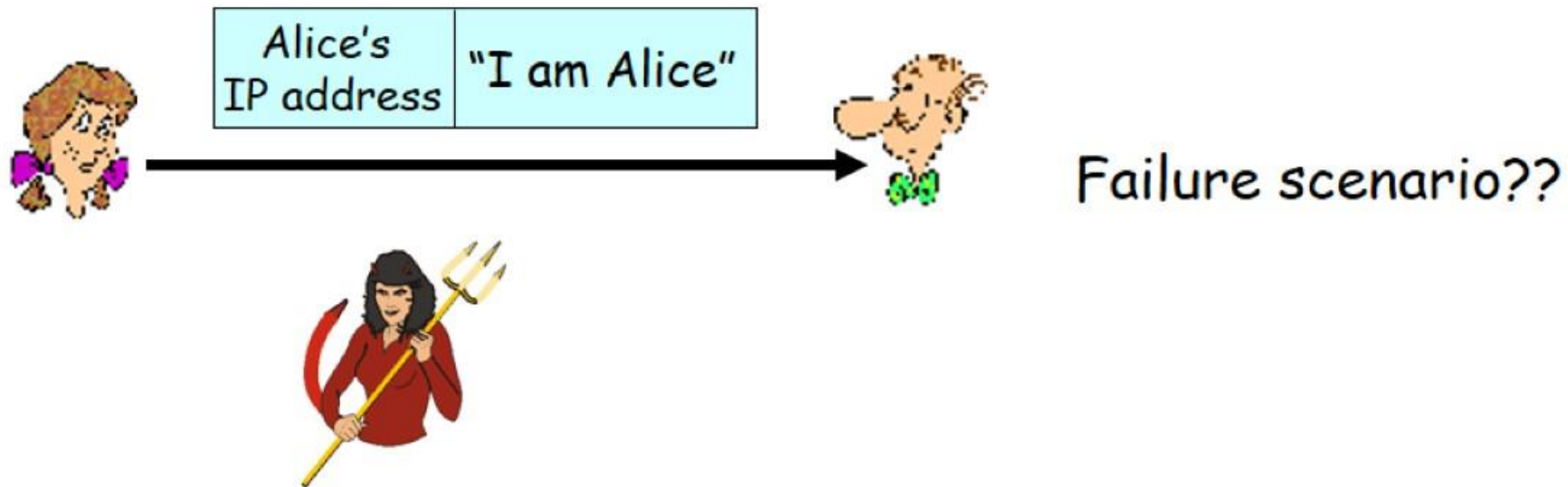


“I am Alice”

in a network,
Bob can not “see”
Alice, so Trudy simply
declares
herself to be Alice



Protocol ap2.0: Alice says "I am Alice" in an IP packet containing her source IP address





Protocol ap2.0: Alice says "I am Alice" in an IP packet containing her source IP address



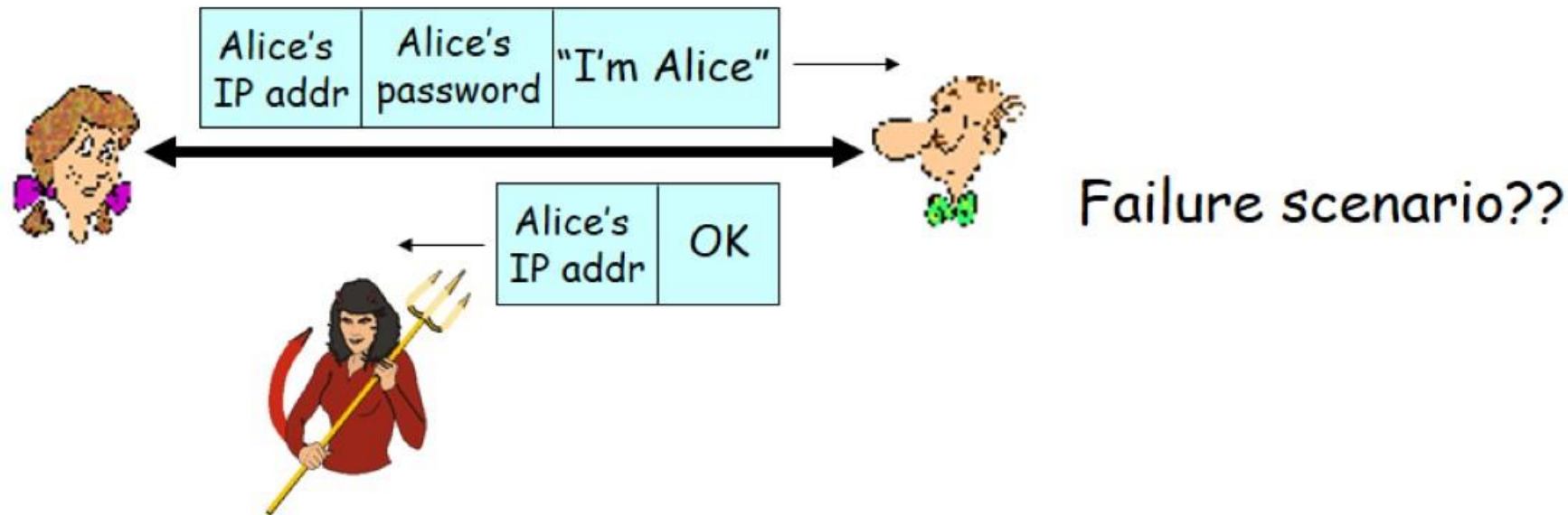
Alice's
IP address

"I am Alice"

Trudy can create
a packet
"spoofing"
Alice's address

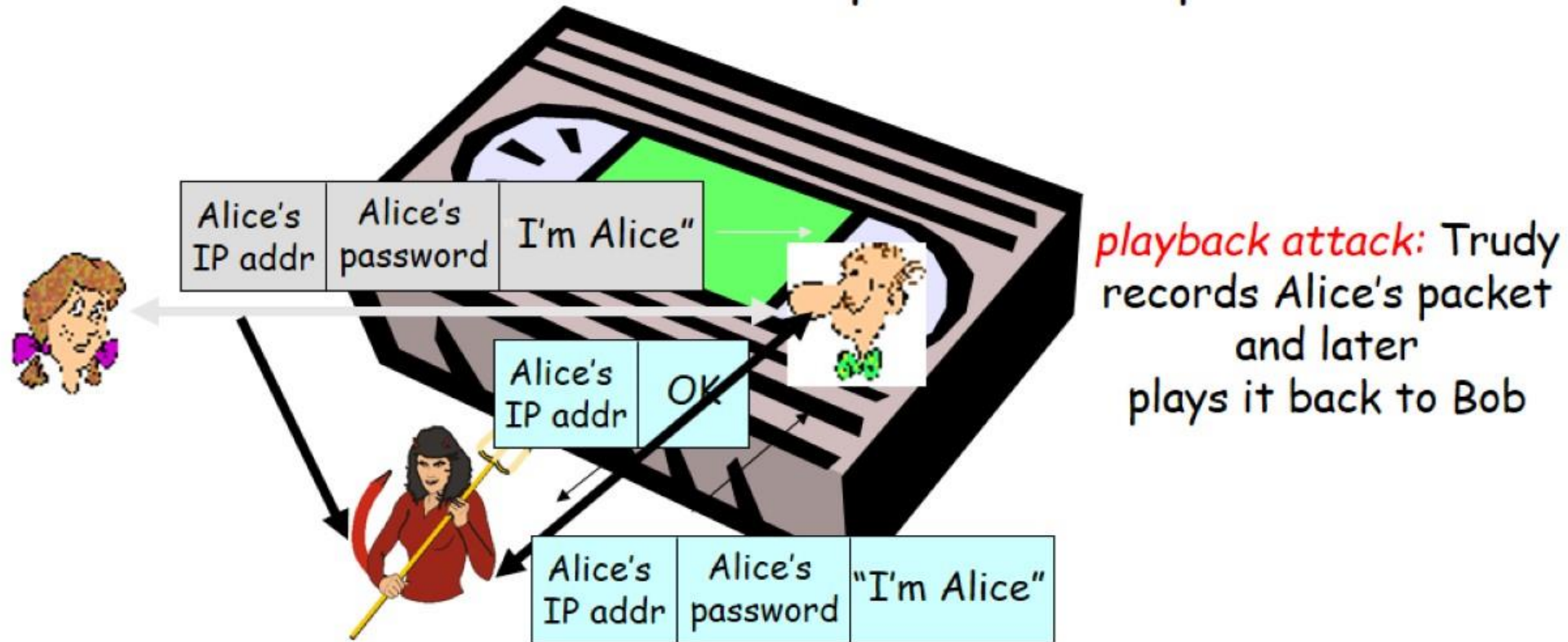


Protocol ap3.0: Alice says "I am Alice" and sends her secret password to "prove" it.



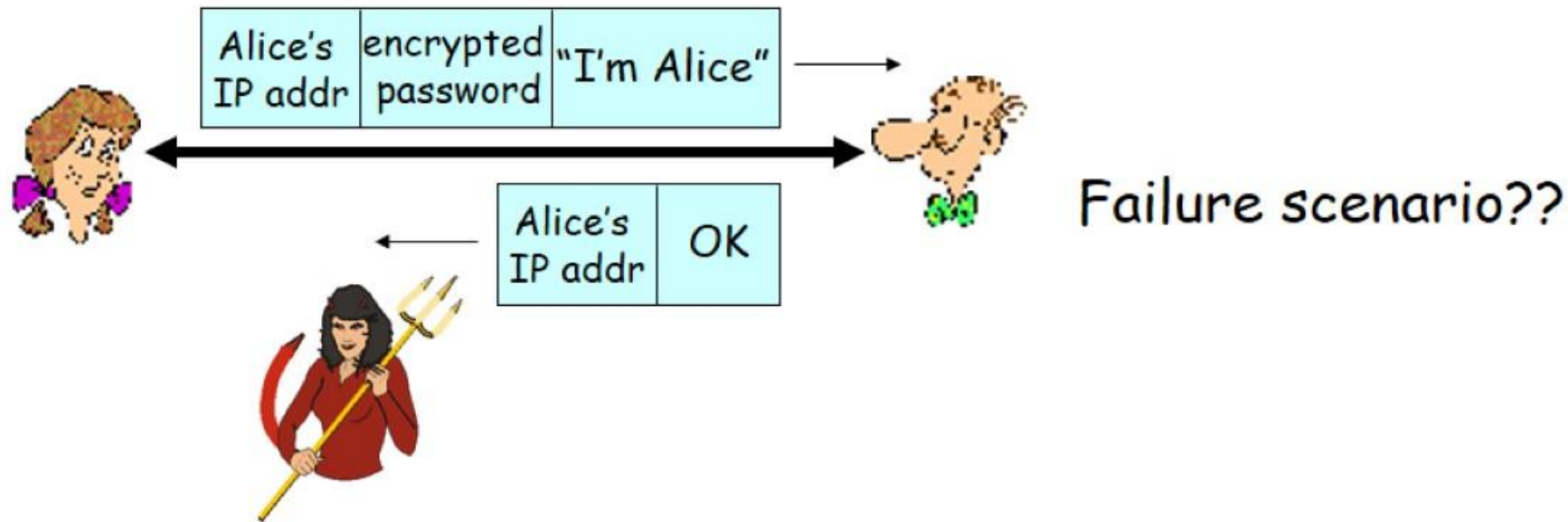


Protocol ap3.0: Alice says "I am Alice" and sends her secret password to "prove" it.



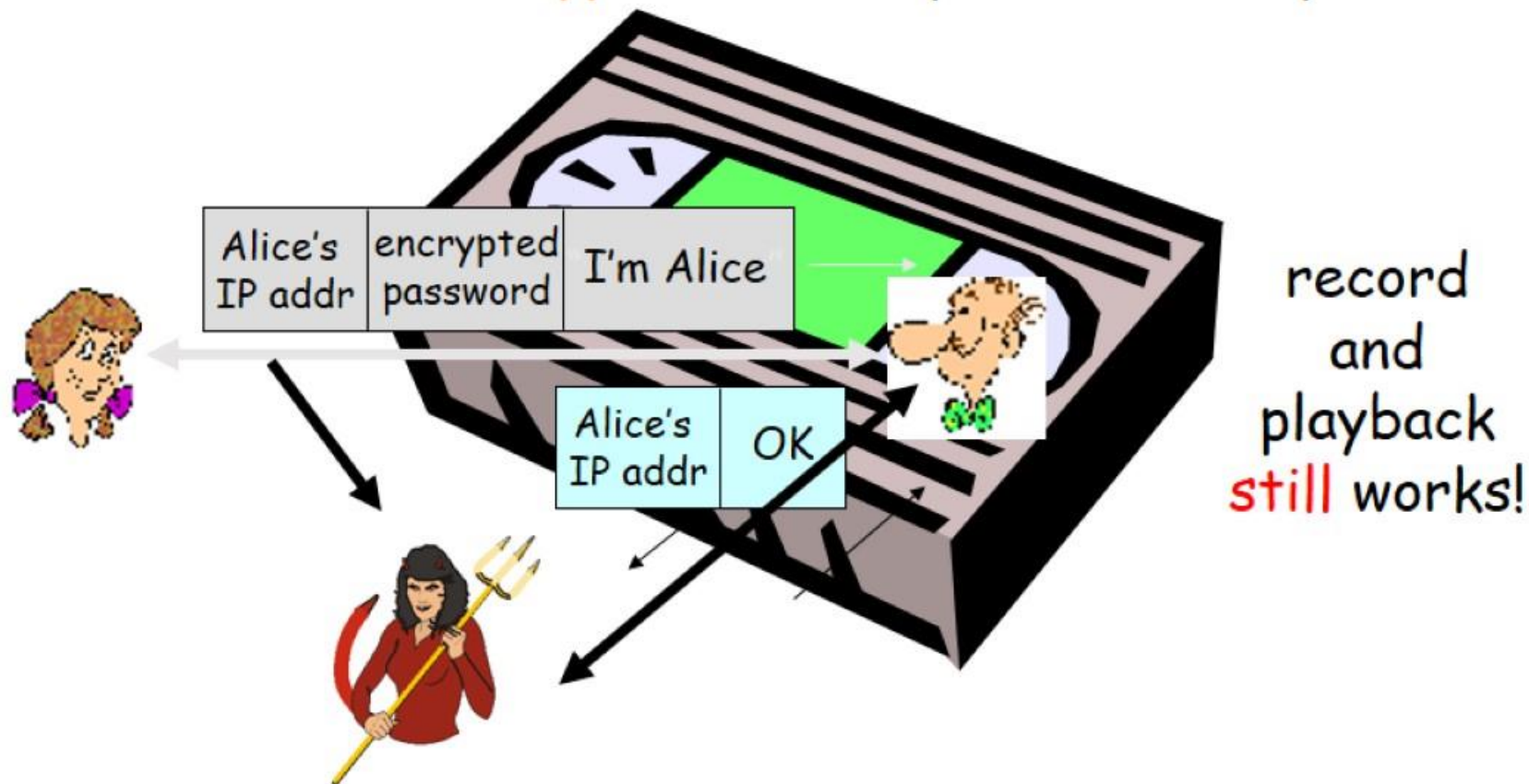


Protocol ap3.1: Alice says "I am Alice" and sends her *encrypted* secret password to "prove" it.





Protocol ap3.1: Alice says "I am Alice" and sends her *encrypted* secret password to "prove" it.

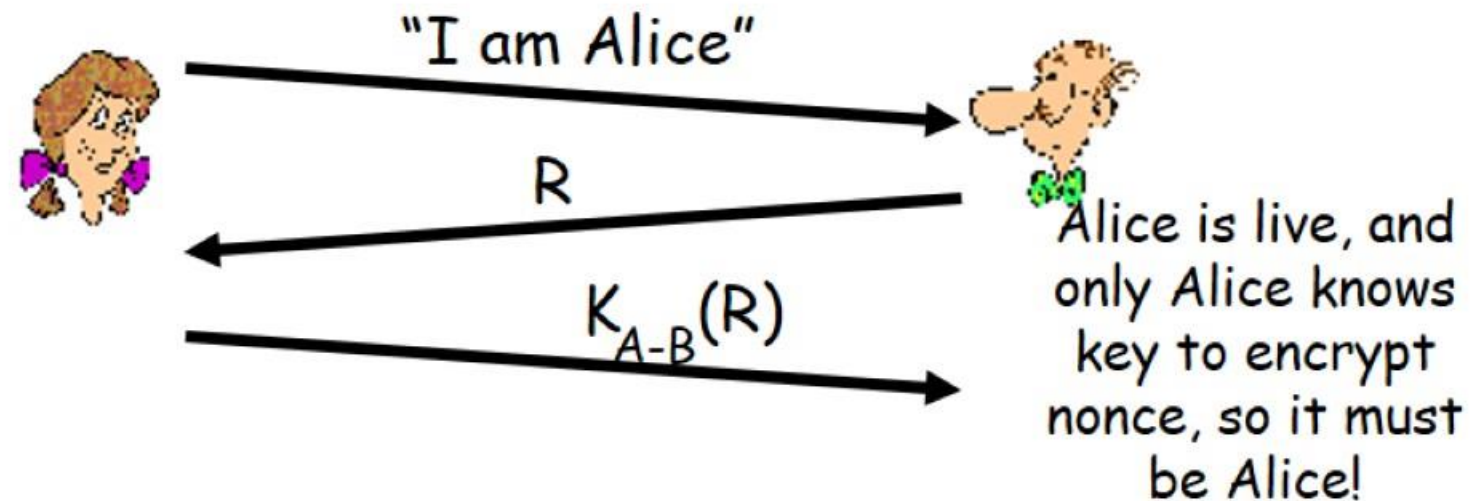




Goal: avoid playback attack

Nonce: number (R) used only *once -in-a-lifetime*

ap4.0: to prove Alice "live", Bob sends Alice **nonce**, R. Alice must return R, encrypted with shared secret key

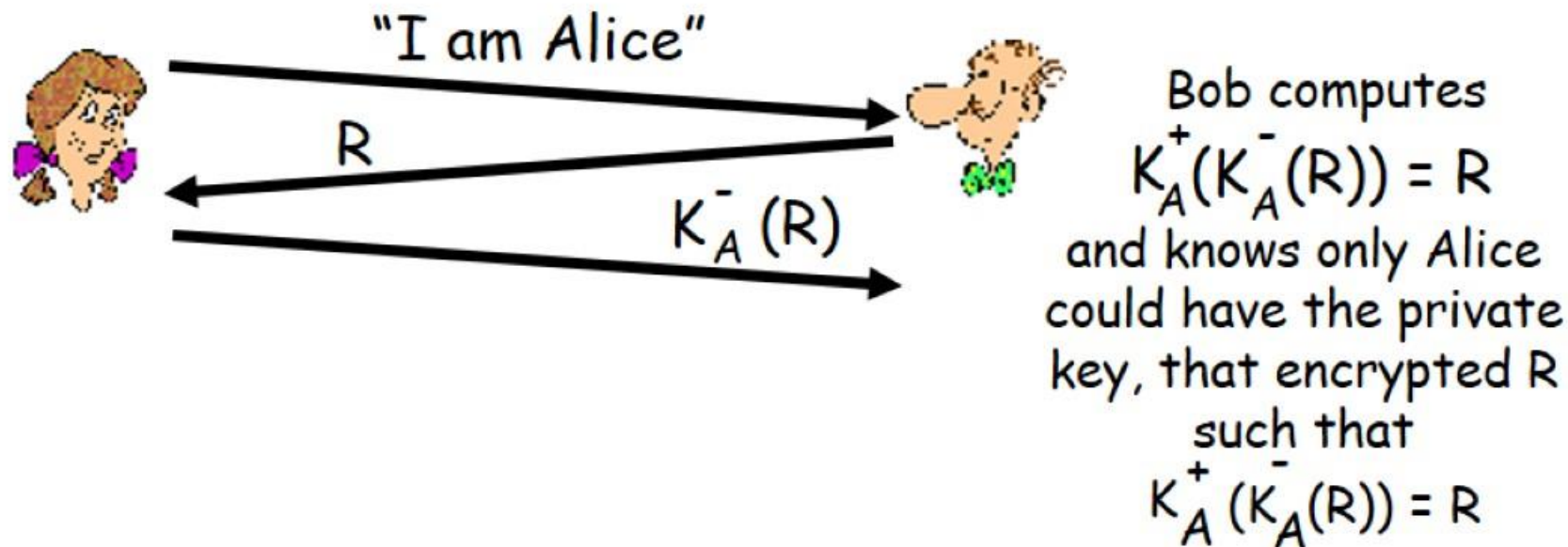




ap4.0 doesn't protect against server database reading

- can we authenticate using public key techniques?

ap5.0: use nonce, public key cryptography





Can block offline guessing attacks by denying access to encrypted passwords

make available
only to
privileged users

**shadow
password file**

- a separate file from the user IDs where the hashed passwords are kept

vulnerabilities

weakness in
the OS that
allows access
to the file

accident with
permissions
making it
readable

users with
same
password on
other
systems

access from
backup
media

sniff
passwords in
network
traffic





- Bad idea to store passwords in a file
- But need a way to verify passwords
- Cryptographic solution: **hash** the password
 - Store $y = h(\text{password})$
 - Can verify entered password by hashing
 - If Trudy obtains “password file”, she does not obtain passwords
- But Trudy can try a *forward search*
 - Guess x and check whether $y = h(x)$



- Hash password with **salt**
- Choose random salt **s** and compute
$$y = h(\text{password}, s)$$
and store (s,y) in the password file
- Note: The salt s is not secret
- Easy to verify salted password
- But Trudy must re-compute dictionary hashes for each user
 - Lots more work for Trudy!



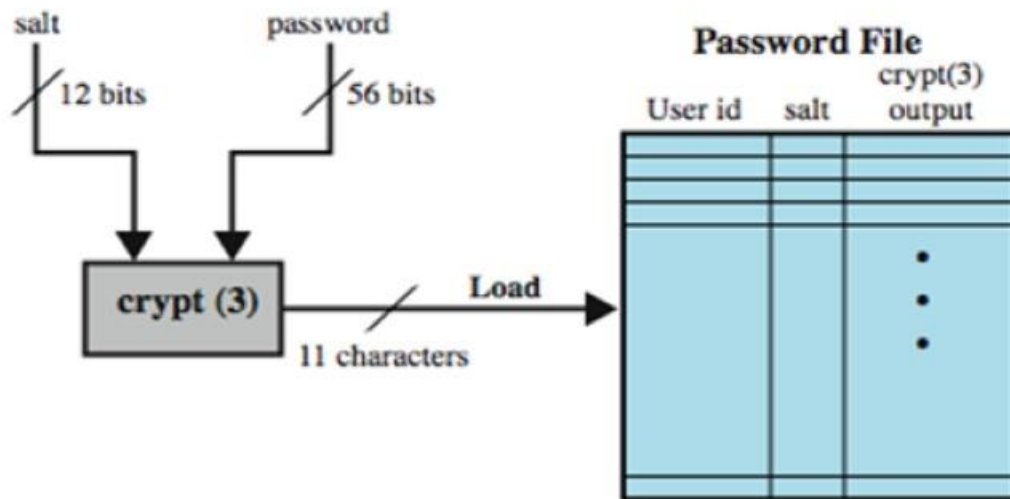
Without salt

- Same hash functions on all machines
 - Compute hash of all common strings once
 - Compare hash file with all known password files

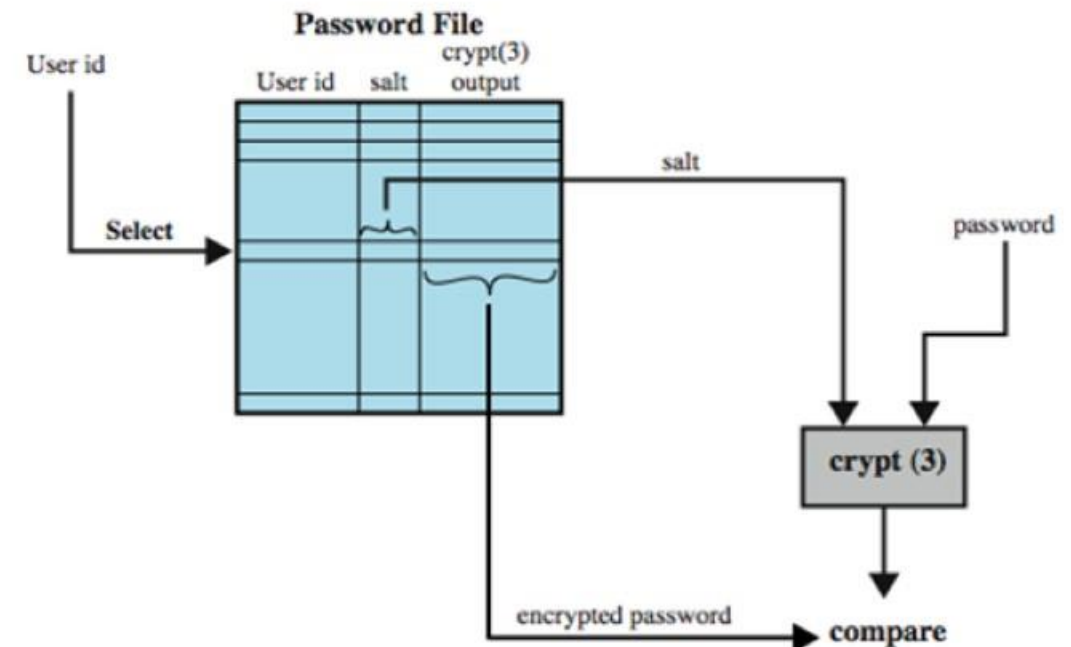
With salt

- One password hashed 2^{12} different ways
 - Precompute hash file?
 - Need much larger file to cover all common strings
 - Dictionary attack on known password file
 - For each salt found in file, try all common strings

- prevents duplicate passwords from being visible in the password file.
- greatly increases the difficulty of offline dictionary attacks. For a salt of length b bits, *the number of possible passwords is increased by a factor of 2^b .*
- becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them.



(a) Loading a new password



(b) Verifying a password



- Some ways you might be able to determine a user's password, in decreasing order of difficulty.
 - Try all possible passwords.
 - Try frequently used passwords.
 - Try passwords likely for the user.
 - Search for the system list of passwords.
 - Ask the user.



12 steps an attacker might try in order to determine a password

1. No password
2. The same as the user ID
3. Is, or is derived from, the user's name
4. Common word list (for example, "password," "secret," "private") plus common names and patterns (for example, "asdfg," "aaaaaa")
5. Short college dictionary
6. Brute force, full character set
7. Common non-English language dictionaries
8. Short college dictionary with capitalizations (PaSsWoRd) and substitutions (0 for O, and so forth)
9. Complete English with capitalizations and substitutions
10. Common non-English dictionaries with capitalization and substitutions
11. Brute force, lowercase alphabetic characters
12. Complete English word list



- The attacker tries all possible passwords, usually in some automated fashion.
- If passwords are words consisting of the 26 characters AZ and can be of any length from 1 to 8 characters:
 - There are 26^1 passwords of 1 character, 26^2 passwords of 2 characters, and 26^8 passwords of 8 characters.
 - The system as a whole has $26^1 + 26^2 + \dots + 26^8 = 26^9 - 1$ almost equals $5 * 10^{12}$ or five million possible passwords.



- If we want to use a computer to create and try each password at a rate of checking one password per millisecond, it would take on the order of 150 years to test all.
- But if we can speed up the search to one password per microsecond, the work factor drops to about two months.



- Searching for a single particular password does not necessarily require all passwords to be tried.
- An intruder needs to try only until the correct password is identified.
- People tend to pick simple passwords.



- A common attack path for Trudy would be:
- outsider $\longrightarrow$ normal user $\longrightarrow$ administrator
- One weak password on a system –or one weak password on an entire network- could be enough for the first stage of the attack to succeed.



➤ Attacker could...

- Target one particular account
- Target any account on system
- Target any account on any system
- Attempt denial of service (DoS) attack

➤ Common attack path

- Outsider → normal user → administrator
- May only require **one** weak password!



❖ Typical password dictionary

- 1,000,000 entries of common passwords
 - people's names, common pet names, and ordinary words.
- Suppose you generate and analyze 10 guesses per second
 - This may be reasonable for a web site; offline is *much* faster
- Dictionary attack in at most 100,000 seconds = 28 hours, or 14 hours on average

❖ If passwords were random

- Assume six-character password
 - Upper- and lowercase letters, digits, 32 punctuation characters
 - 689,869,781,056 password combinations.
 - Exhaustive search requires 1,093 years on average



- Trudy pre-computes $h(x)$ for all x in a **dictionary** of common passwords
- Suppose Trudy gets access to password file containing hashed passwords
 - She only needs to compare hashes to her pre-computed dictionary
 - After one-time work, attack is trivial
- Can we prevent this attack? Or at least make attacker's job more difficult?



- **Dictionary attacks**

- develop a large dictionary of possible passwords and try each against the password file
- each password must be hashed using each **salt value** and then compared to stored hash values

- **Rainbow table attacks**

- pre-compute tables of hash values for all **salts**
- a mammoth table of hash values
- can be countered by using a sufficiently large salt value and a sufficiently large hash length



- Password crackers exploit the fact that people choose easily guessable passwords
 - Shorter password lengths are also easier to crack
- John the Ripper
 - Open-source password cracker first developed in 1996
 - Uses a combination of brute-force and dictionary techniques



Observed Password Lengths

Length	Number	Fraction of Total
1	55	.004
2	87	.006
3	212	.02
4	449	.03
5	1260	.09
6	3035	.22
7	2917	.21
8	5772	.42
Total	13787	1.0

Purdue University study on 54 systems and 7000 users





- Passwords Cracked from a Sample Set of 13,797 Accounts Compiled 3M dictionary words and their variations.
- The more words that need to be tested for a match, the lower the cost/benefit ratio.

Type of Password	Search Size	Number of Matches	Percentage of Passwords Matched	Cost/Benefit Ratio ^a
User/account name	130	368	2.7%	2.830
Character sequences	866	22	0.2%	0.025
Numbers	427	9	0.1%	0.021
Chinese	392	56	0.4%	0.143
Place names	628	82	0.6%	0.131
Common names	2239	548	4.0%	0.245
Female names	4280	161	1.2%	0.038
Male names	2866	140	1.0%	0.049
Uncommon names	4955	130	0.9%	0.026
Myths and legends	1246	66	0.5%	0.053
Shakespearean	473	11	0.1%	0.023
Sports terms	238	32	0.2%	0.134
Science fiction	691	59	0.4%	0.085
Movies and actors	99	12	0.1%	0.121
Cartoons	92	9	0.1%	0.098
Famous people	290	55	0.4%	0.190
Phrases and patterns	933	253	1.8%	0.271
Surnames	33	9	0.1%	0.273
Biology	58	1	0.0%	0.017
System dictionary	19683	1027	7.4%	0.052
Machine names	9018	132	1.0%	0.015
Mnemonics	14	2	0.0%	0.143
King James bible	7525	83	0.6%	0.011
Miscellaneous words	3212	54	0.4%	0.017
Yiddish words	56	0	0.0%	0.000
Asteroids	2407	19	0.1%	0.007
TOTAL	62727	3340	24.2%	0.053



- Study in 2013
- 25,000 students at university with complex password policy

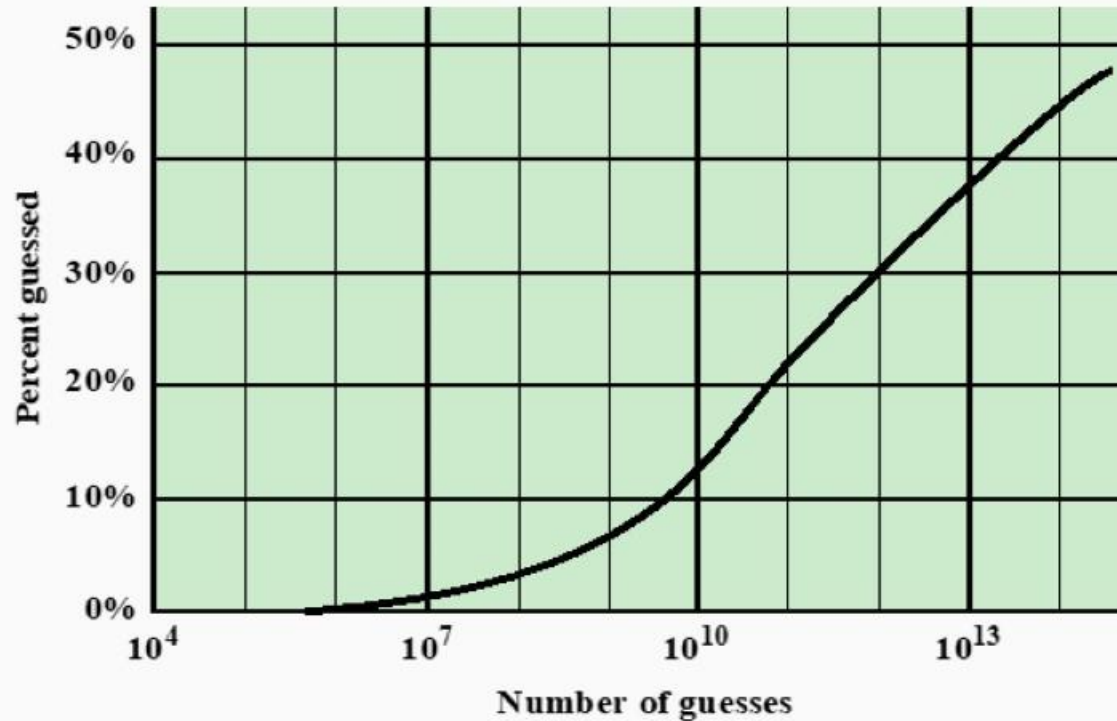


Figure 3.3 The Percentage of Passwords Guessed After a Given Number of Guesses



- Much stronger hash/salt schemes available for Unix
- Recommended hash function is based on MD5
 - salt of up to 48-bits
 - password length is unlimited
 - produces 128-bit hash
 - uses an inner loop with 1000 iterations to achieve slowdown
- OpenBSD (Open source UNIX O.S) uses Blowfish block cipher based hash algorithm called Bcrypt
 - most secure version of Unix hash/salt scheme
 - uses 128-bit salt to create 192-bit hash value



❖ Complex password policy

- Forcing users to pick stronger passwords

❖ However, password-cracking techniques have also improved

- The processing capacity available for password cracking has increased dramatically
- The use of sophisticated algorithms to generate potential passwords
- Studying examples and structures of actual passwords in use



- **Rule enforcement**
 - Specific rules that passwords must adhere to
- **Password cracker**
 - Compile a large dictionary of passwords not to use
- **Bloom filter**
 - Used to build a table based on dictionary using hashes
 - Check desired password against this table



This chapter explained:

- Password Selection Criteria
- One-Time Passwords
- Challenge-response Authentication
- Attacks on Passwords



Thank You