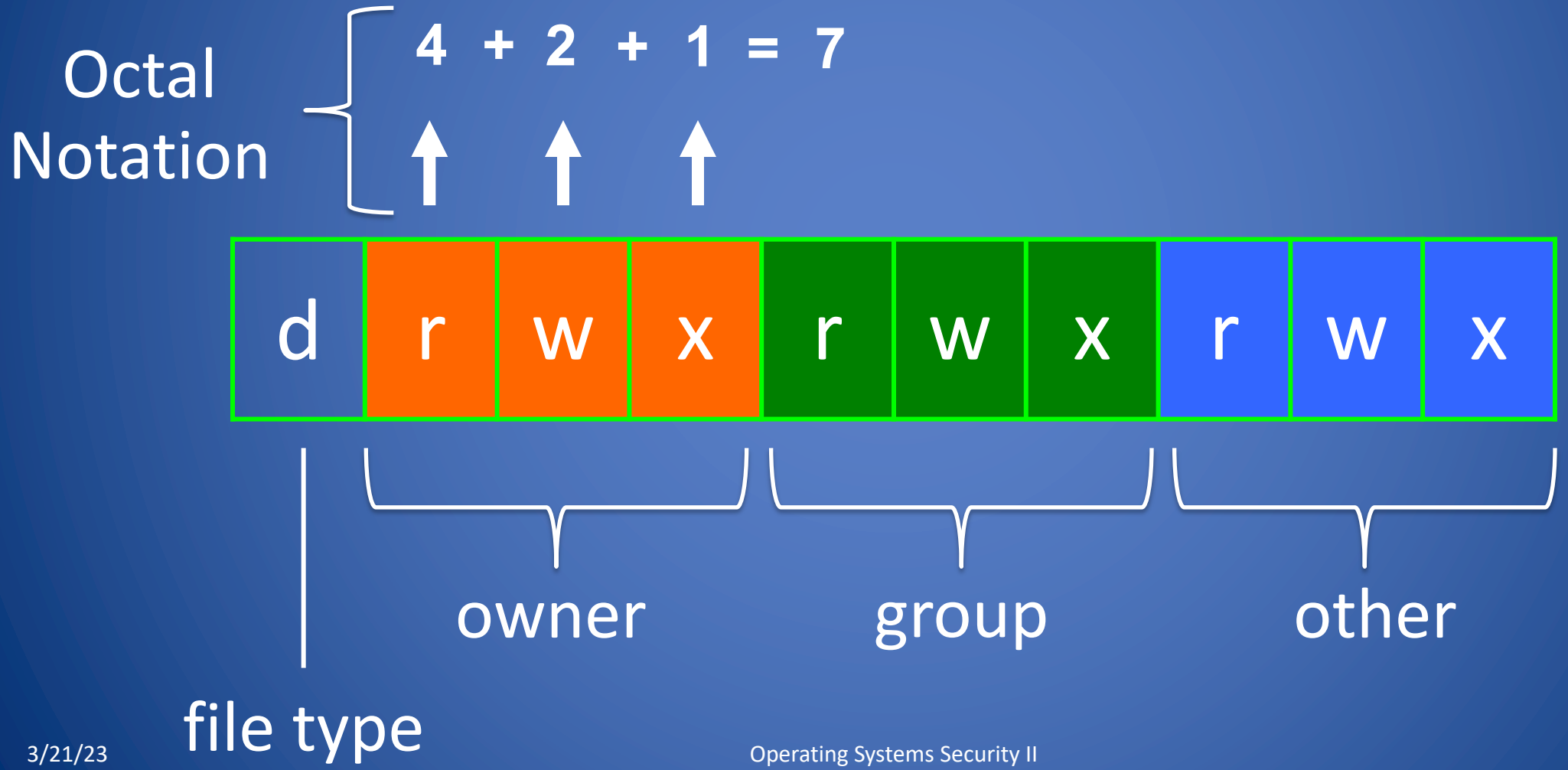
Operating Systems Security III

CS 1660: Introduction to Computer Systems Security

Unix File Types RWX and octal notation



Octal Notation (recap)

Another way to specify permissions

- Digits from left (most significant) to right(least significant):

  [special bits][user bits][group bits][other bits]

- Special bit digit =

  (4 if setuid) + (2 if setgid) + (1 if sticky)

- All other digits =

  (4 if readable) + (2 if writable) + (1 if executable)

Permissions Examples (Regular Files)

-rw-r—r--	read/write for owner, read-only for everyone else
-rw-r-----	read/write for owner, read-only for group, forbidden to others
-rwx------	read/write/execute for owner, forbidden to everyone else
-r--r--r--	read-only to everyone, including owner
-rwxrwxrwx	read/write/execute to everyone

Permissions for Directories

- Permissions bits interpreted differently for directories
- *Read* bit allows listing names of files in directory, but not their properties like size and permissions
- *Write* bit allows creating and deleting files within the directory
- *Execute* bit allows entering the directory and getting properties of files in the directory
- Lines for directories in `ls -l` output begin with `d`, as below:

```
jk@sphere:~/test$ ls -l
Total 4
drwxr-xr-x  2 jk ugrad 4096 2005-10-13 07:37 dir1
-rw-r--r--  1 jk ugrad    0 2005-10-13 07:18 file1
```

Permissions Examples (Directories)

drwxr-xr-x	all can enter and list the directory, only owner can add/delete files
drwxrwx---	full access to owner and group, forbidden to others
drwx--x---	full access to owner, group can access known filenames in directory, forbidden to others
-rwxrwxrwx	full access to everyone

The /tmp Directory

- In Unix systems, directory /tmp is
  - Read/write for any user
  - Wiped on reboot (or lives entirely in memory)

Convenience

  - Place for temporary files used by applications
  - Files in /tmp are not subject to the user's space quota

What could go wrong?

Special Permission Bits

Three other important permission bits

- Set-user-ID (“suid” or “setuid”) bit
- Set-group-ID (“sgid” or “setgid”) bit
- Sticky bit

setuid bit: Set-user-ID

- On executable files, causes the program to run as file owner regardless of who runs it

- How to view: shown as s instead of x

  -rwsr-xr-x: setuid, executable by all

  -rwxr-xr-x: executable by all, but not setuid

Setuid Programs

- Unix processes have two user IDs:
  - real user ID (UID): user launching the process
  - effective user ID (EUID): user whose privileges are granted to the process
- If a user A executes a setuid file owned by B, then the effective user ID of the process is B and not A

Setuid Programs

- System call setuid(uid) allows a process to change its effective user ID to uid
- Some programs that access system resources are owned by root and have the setuid bit set (setuid programs)
  - e.g., passwd and su
- Setuid generally ignored on shell scripts—why?

setgid bit: Set-group-ID (recap)

- On executable files: causes the program to run with the file's group, regardless of whether the user who runs it is in that group
- On directories, causes files created within the directory to have the same group as the directory

```
Examples
        -rwxr-sr-x: setgid file, executable by all
        drwxrwsr-x: setgid directory; files within will have group of directory
```

Symbolic Link (recap)

- In Unix, a symbolic link (aka symlink) is a file that points to (stores the path of) another file
- A process accessing a symbolic link is transparently redirected to accessing the destination of the symbolic link
- Symbolic links can be chained, but not to form a cycle

- ln -s really_long_directory/even_longer_file_name myfile

Code as Data

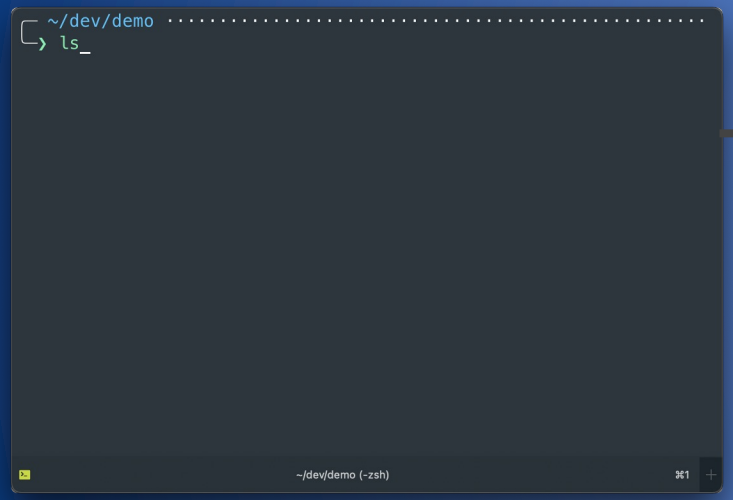
Theme: What will our code do?

- It is ***hard*** to reason about what our code will do
  - There are some fundamental limits (e.g. halt problem)

- System components consume data and treat it like code:
  User input is *unpredictable*

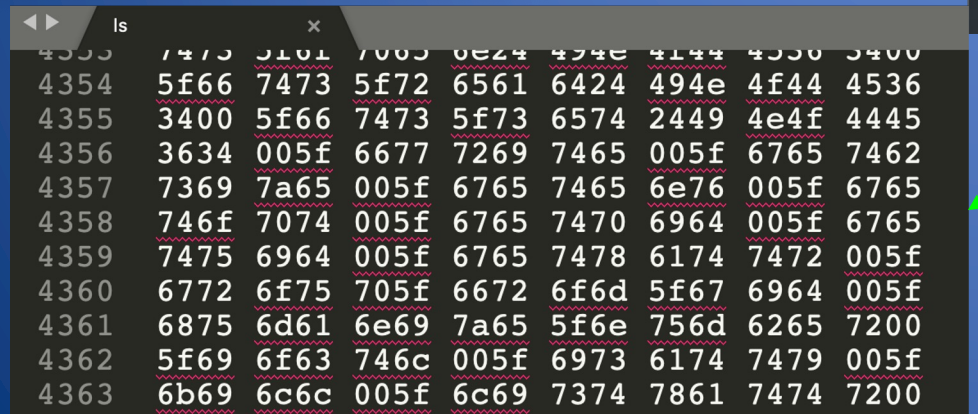
Code as Data

Example: Bash Shell -- What happens when we run “ls”?




```
$PATH=/Users/wds/.shell/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/GPAC.app/Contents/MacOS/:/Library/TeX/texbin:
```

```
[*          dash*       expr*       ln*         pwd*        sync*
bash*       date*       hostname*   ls*         rm*         tcsh*
cat*        dd*         kill*       mkdir*      rmdir*      test*
chmod*      df*         ksh*        mv*         sh*         unlink*
cp*         echo*       launchctl*  pax*        sleep*      wait4path*
csh*        ed*         link*       ps*         stty*       zsh*
```



Code as Data: Unexpected Results

```
alice@handin-cs162-wschor:~$ echo $HOME
/home/alice
alice@handin-cs162-wschor:~$ _
```

```
alice@handin-cs162-wschor:~$ ls $HOME
CS162_README  demo
alice@handin-cs162-wschor:~$ _
```

```
alice@handin-cs162-wschor:~$ HOME=-R
alice@handin-cs162-wschor:/home/alice$ _
```

Code as Data: Unexpected Results

```
alice@handin-cs162-wschor:/home/alice$ ls $HOME
.:
CS162_README  demo

./demo:
file1  file2  file3
alice@handin-cs162-wschor:/home/alice$ _
```

So setuid/setgid is dangerous...

setuid/setgid is dangerous...

In modern times: only for programs that really need it

- System programs that changing passwords/users, legacy programs
  - Don't do this yourself!
- Very very bad idea for shell scripts

What else can we do?

When do we need this?

In the shell: su, sudo

- Run as another user (if you have permissions)

```
user@shell:~$ su -c "command" other user
```

- Run commands as root (or another user) based on system config file (/etc/sudoers)
  - Can restrict to specific commands, environment, ...

```
user@shell:~$ sudo whoami
root
```

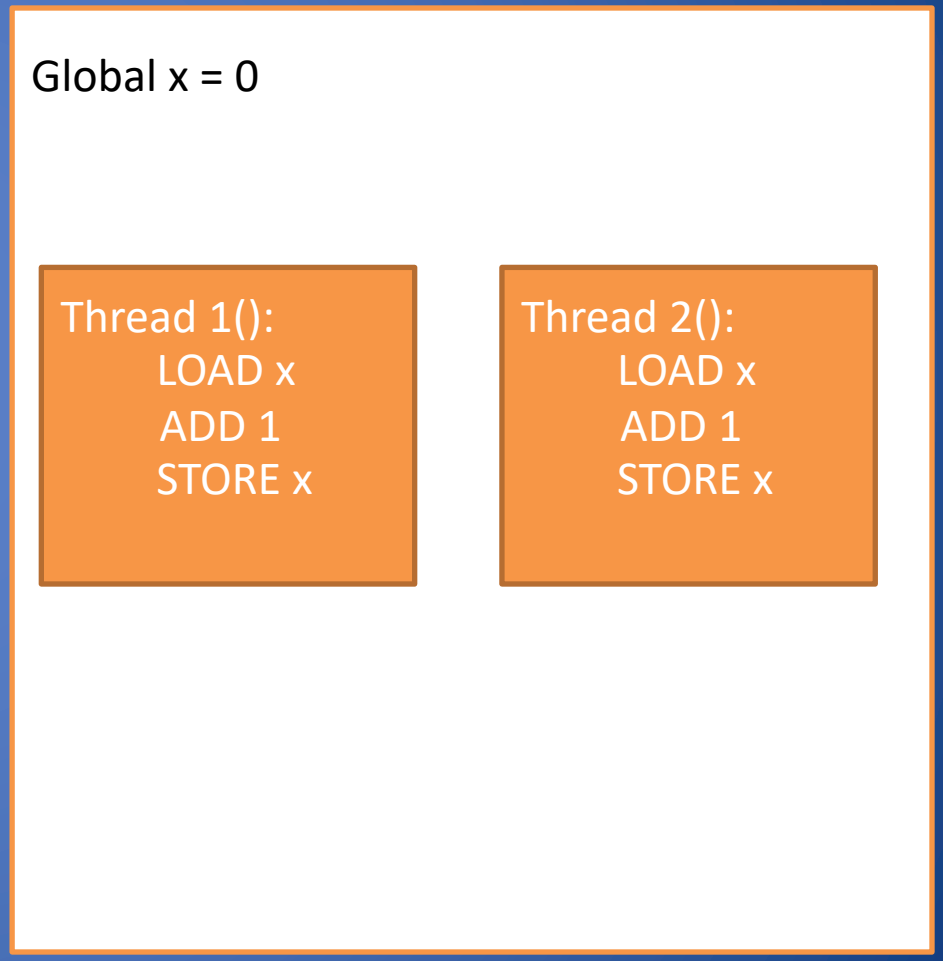
```
/etc/sudoers:
%wheel ALL=(ALL) NOPASSWD: ALL

. . .
```

What ELSE could we do?

Race Condition

Race Condition

- A race condition occurs when two threads want to access the same memory
- Run Thread 1() and Thread 2()
  - Outcome is 1 or 2



Race Condition

```
1.  if (!access("/tmp/X", W_OK)) {
        /* the real user ID has access right */
2.      f = open("/tmp/X", O_WRITE);
3.      write_to_file(f);
        }
    else {
        /* the real user ID does not have
        access right */
4.      fprintf(stderr, "Permission denied\n");
        }
```

- Fragment of setuid program that writes into file /tmp/X on behalf of a user who created it

- access verifies permission of real user ID
  - Transparently follows symlinks
- open verifies permission of effective user ID
  - Transparently follows symlinks
- What can go wrong?

Source: Kevin Du, Race Condition Vulnerability, Lecture Notes

TOCTOU Vulnerability

```
1.  if (!access("/tmp/X", W_OK)) {
        /* the real user ID has access right */
2.      f = open("/tmp/X", O_WRITE);
3.      write_to_file(f);
        }
    else {
        /* the real user ID does not have
        access right */
4.      fprintf(stderr, "Permission denied\n");
        }
```

- What can go wrong?
  - In between (1) and (2), user could replace /tmp/X with symlink to /etc/passwd
  - Not easy to accomplish (timing)
- Example of time of check to time of use (TOCTOU) vulnerability

Attempt to Fix the Race Condition

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.    int f = open("/tmp/X", O_RDWR);
4.    fstat(f, &statAfter);
5.    if (statAfter.st_ino == statBefore.st_ino) {
         /* the I-node is still the same */
6.     write_to_file(f);
       }
7.    else perror("Race Condition Attacks!");
       }
8. else fprintf(stderr, "Permission denied\n");
   }
```

Source: Kevin Du, Race Condition Vulnerability, Lecture Notes

- lstat and fstat access file descriptor for a path, which includes unique file ID (st_ino)
  - lstat does not traverse symlink
  - fstat accesses descriptor of open file, after symlink traversed by open
- Step (5) compares IDs of
  - file checked in (1) and
  - file opened in (3)
- Check-use-check_again approach
  - Defeats swapping in symlink between access and open
- Fails also if /tmp/X is a symlink when (2) is executed

Does the Fix Work?

```
1. lstat("/tmp/X", &statBefore);
2. if (!access("/tmp/X", O_RDWR)) {
3.    int f = open("/tmp/X", O_RDWR);
4.    fstat(f, &statAfter);
5.    if (statAfter.st_ino == statBefore.st_ino) {
         /* the I-node is still the same */
6.     write_to_file(f);
       }
7.    else perror("Race Condition Attacks!");
       }
8. else fprintf(stderr, "Permission denied\n");
   }
```

- New attack
  - Before (1) /tmp/X is a hard link to /etc/passwd
  - Between (1) and (2) swap in hard link to user-owned file
  - Between (2) and (3) swap in again hard link to /etc/passwd
- This passes the ID check in (5) and allows the user to write to /etc/passwd

Negative Result

- Assumptions
  - Setuid program
  - Path-based permission check for real user ID via syscall access(path, permission) that returns 0 or -1
  - No atomic check-and-open file syscall
- Theorem
  - Program is vulnerable to TOCTOU race condition
- Proof
  - Attacker can always swap good file before access and bad file after access
  - lstat/fstat do not help since they are path-based as well
- Reference
  - Drew Dean, Alan J. Hu: Fixing Races for Fun and Profit: How to Use access (2). USENIX Security Symposium, 2004.

Mitigating and Eliminating Race Conditions

- Hardness amplification
  - Force the adversary to win a large number of races instead of just one or two in order to exploit the vulnerability
  - Reduces the probability of success
  - Complex to accomplish correctly
  - Reference
    - Dan Tsafrir, Tomer Hertz, David Wagner, Dilma Da Silva: Portably Solving File TOCTTOU Races with Hardness Amplification. USENIX File and Storage Technologies, 2008
- Temporary privilege downgrade
  - Within same process
    - Drop to real user ID privileges via setuid(real_userid)
    - Open file
    - Restore root privileges
  - With child process
    - Fork child process with real user ID privileges to open file
  - Approach not portable across Unix variants

https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf

Historical setuid Unix Vulnerabilities: lpr

- Command lpr
  - running as root setuid
  - copied file to print, or symbolic link to it, to spool file named with 3-digit job number (e.g., print954.spool) in /tmp
  - Did not check if file already existed
  - Random sequence was predictable and repeated after 1,000 times
- How can we exploit this?

- Attack
  - A dangerous combination: setuid, /tmp, symlinks, …
  - Create new password file newpasswd
  - Print a very large file
  - lpr –s /etc/passwd
  - Print a small file 999 times
  - lpr newpasswd
  - The password file is overwritten with newpasswd

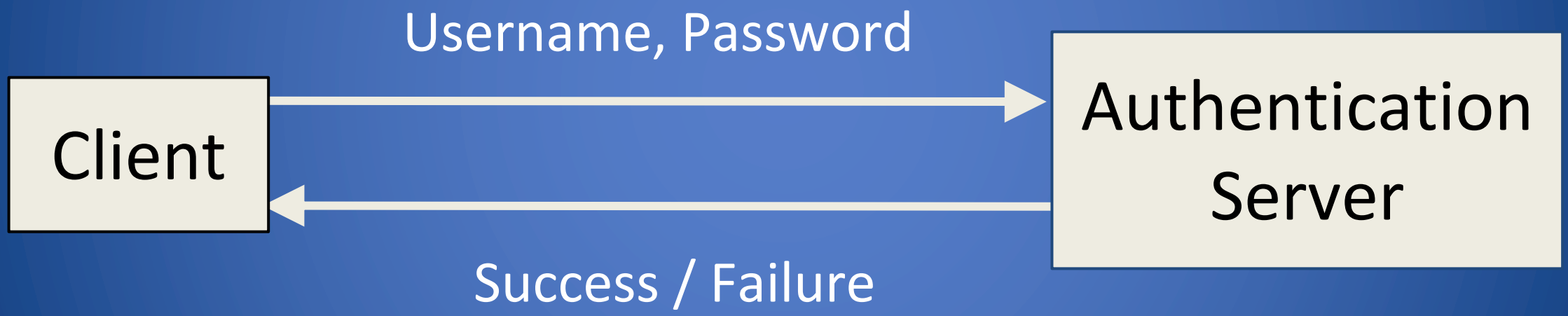
https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Race_Condition.pdf

Passwords

CS 1660: Introduction to Computer Systems Security

Password Authentication

Password Authentication

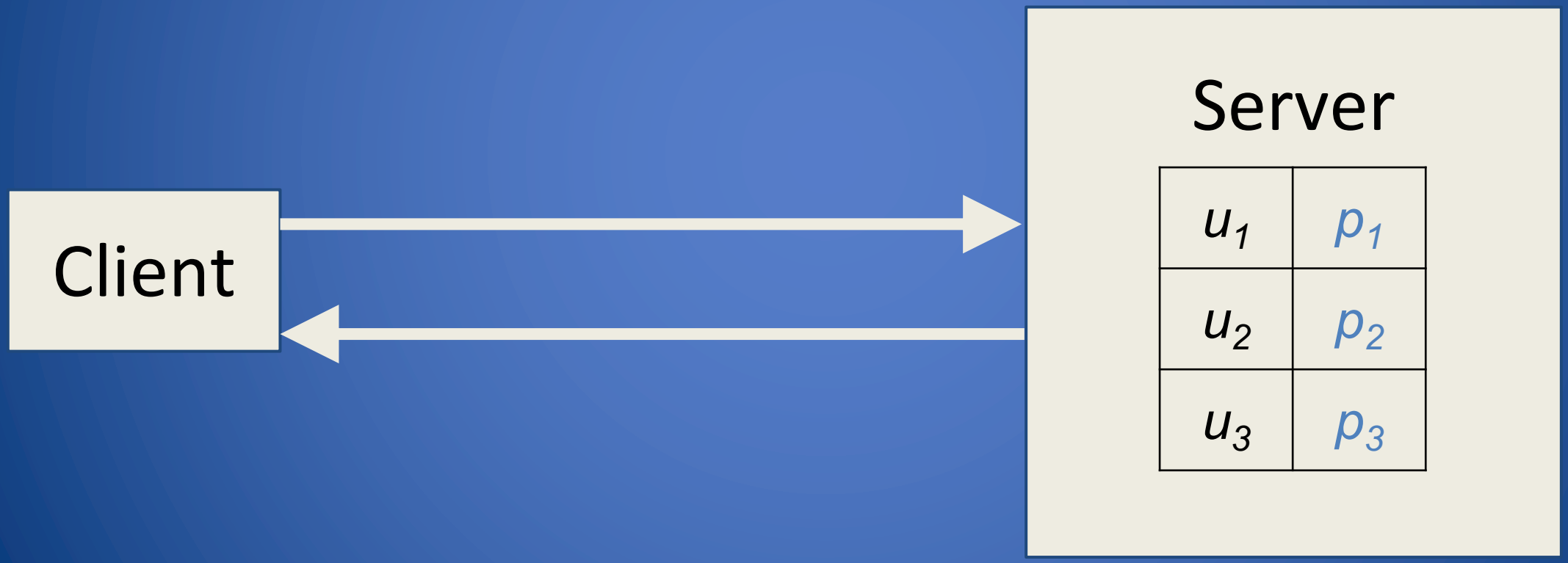


Storing Passwords

How Should the Server Store Passwords?

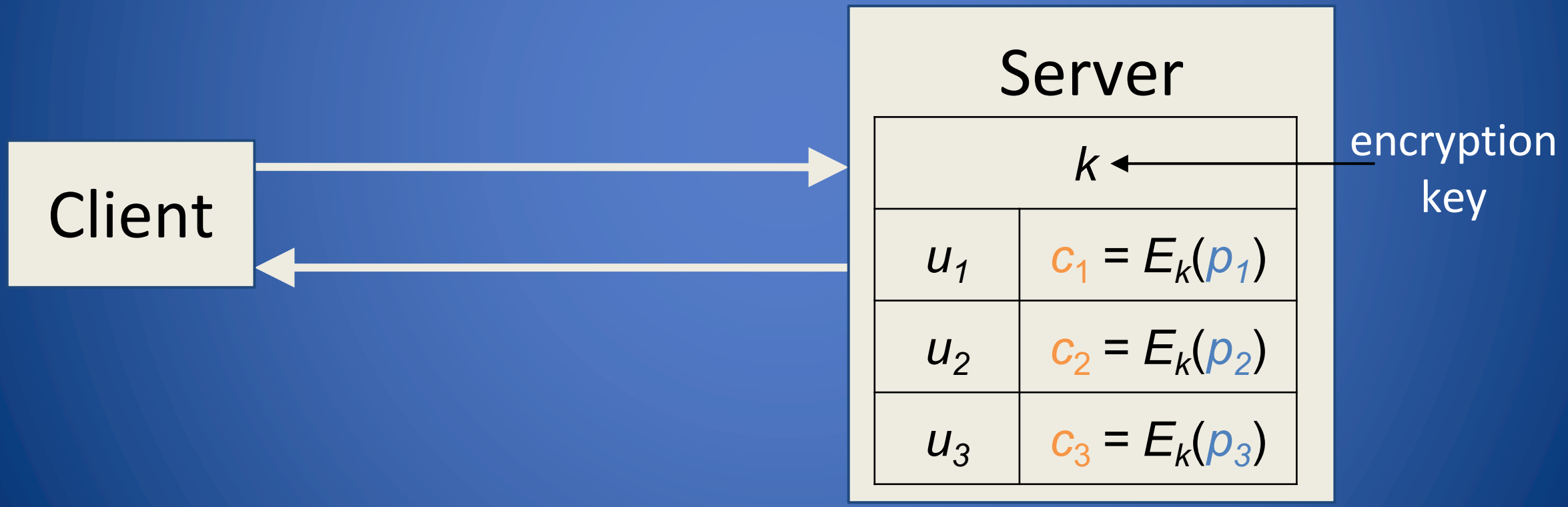
- Our goal is to defend from attacks that exfiltrate the password database stored by the server
  - Most common password-related attack on server
- We don't consider other password attacks on the server
  - Eavesdropping passwords submitted by users
  - Modifying the password authentication code

Attempt #1 - Plaintext



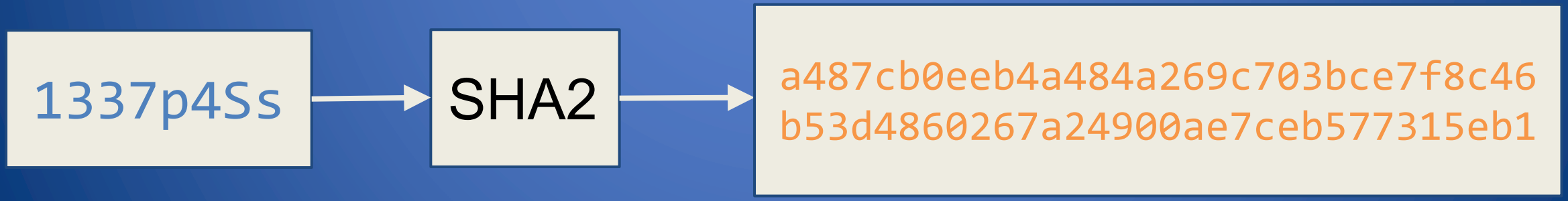
Attempt #2 - Encryption

- Store encrypted passwords
- Decrypt and compare on login



Attempt #3 - Hashing

Example:

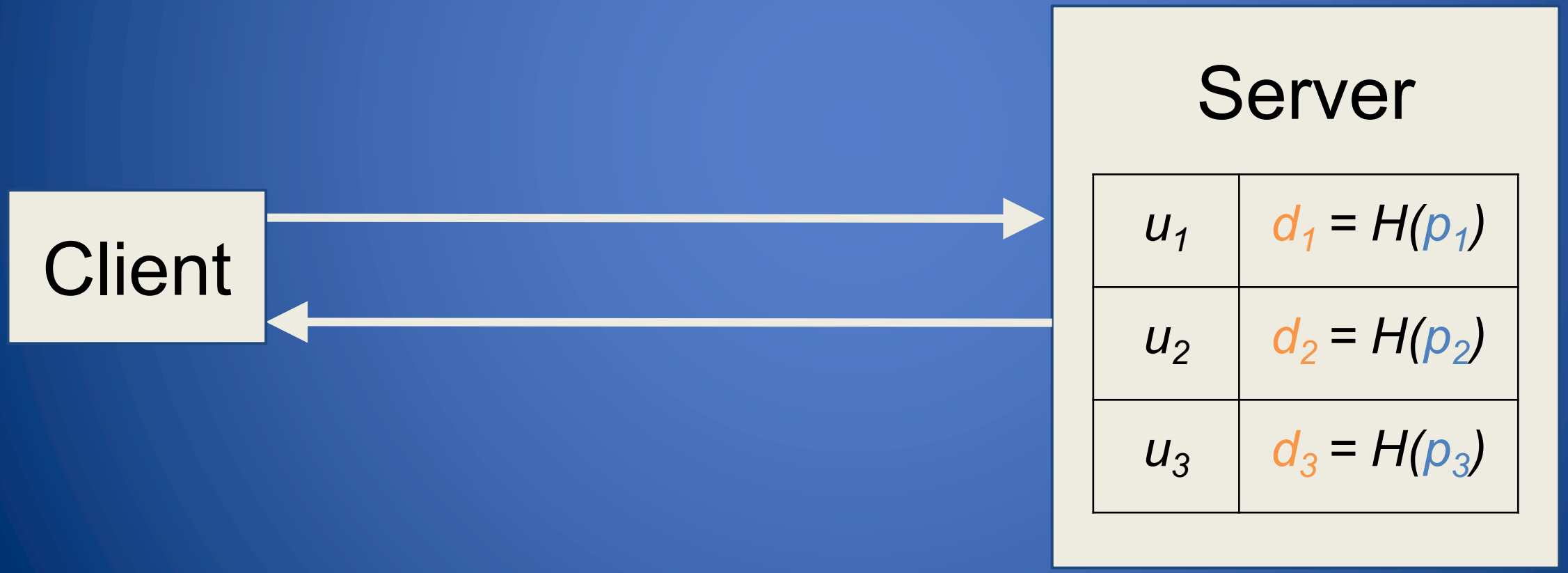


Attempt #3 - Hashing

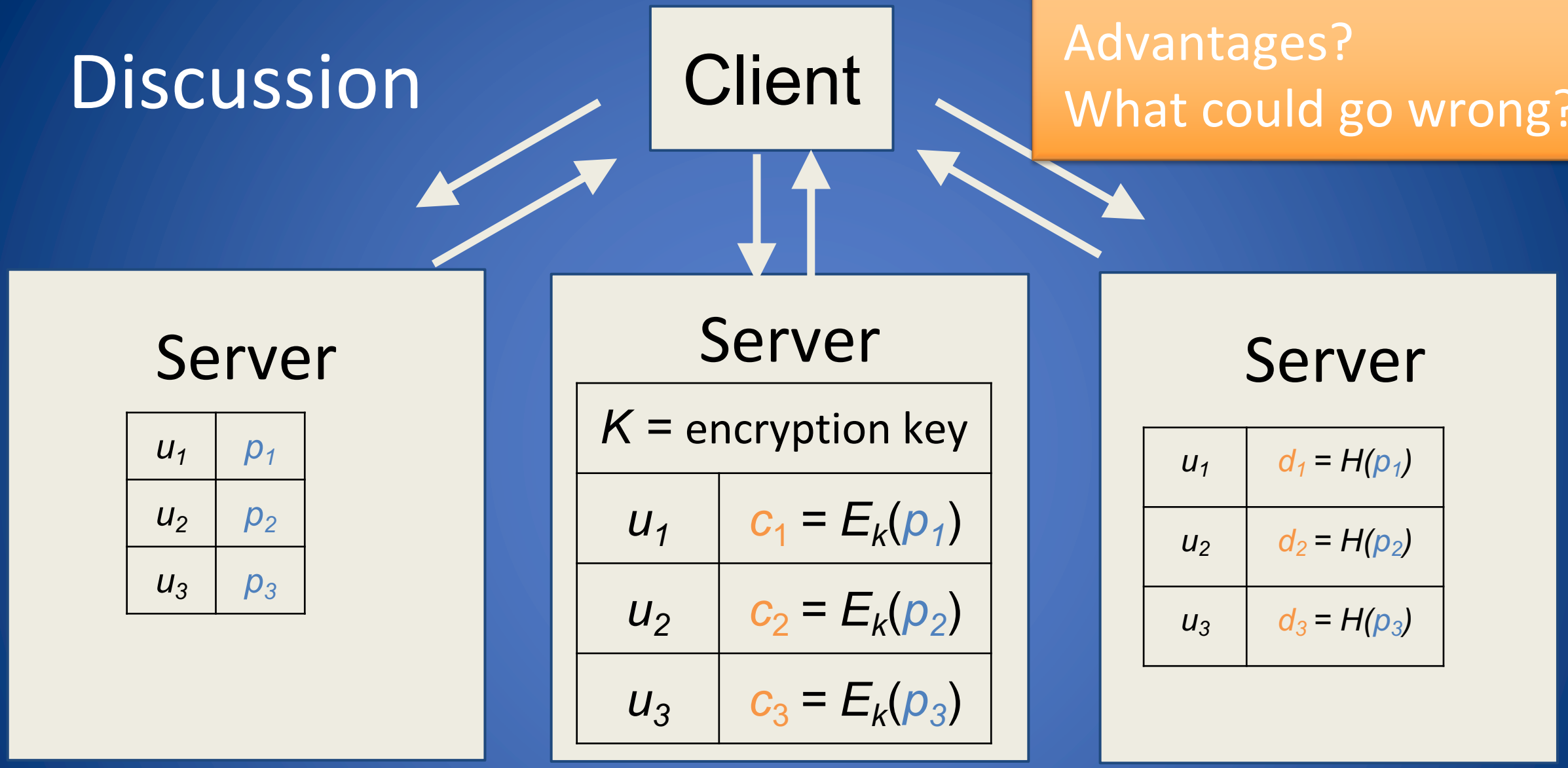
- Recall cryptographic hashing:
  - Variable length input, fixed length "random" output
  - One-way
    - Given hash x, hard to find p such that $H(p) = x$
  - Weak collision resistance
    - Given input p, hard to find q such that $H(p) = H(q)$
  - Strong collision resistance
    - Hard to find distinct p, q such that $H(p) = H(q)$

Attempt #3 - Hashing

- Hash the password, store the hash
- Hash the user-supplied password and compare



Discussion


Client
Advantages?
What could go wrong?
Server
u_1 p_1
u_2 p_2
u_3 p_3
#1 - Plaintext
Server
K = encryption key
u_1 $c_1 = E_k(p_1)$
u_2 $c_2 = E_k(p_2)$
u_3 $c_3 = E_k(p_3)$
#2 - Encryption
Server
u_1 $d_1 = H(p_1)$
u_2 $d_2 = H(p_2)$
u_3 $d_3 = H(p_3)$
#3 - Hashing

Attempt #1 - Plaintext

- Advantages
  - Easier to manage
  - Less computational needs
- What could go wrong?
  - Ifdatabase is stolen, so are passwords!
  - Admins have access to passwords.
- Ex. Reddit (2006), Twitter (2018)

Attempt #2 - Encryption

- Advantages
  - If encrypted passwords are stolen, they can't be decrypted
  - Only administrators with key can decrypt
- What could go wrong?
  - If the encrypted passwords are stolen, what is to keep the key from also being stolen?
  - Anyone with the key (admins) can view passwords
- Ex. Adobe (2013)

Attempt #3 - Hashing

- Registration
  - Hash password, store hash
- Login
  - Hash user-supplied password, compare with stored hash
- What advantages does this scheme have?
  - If database is stolen, hashes need to be cracked
    - Correct
  - Cracking must be done brute-force for every password
    - Is this accurate?

Attempt #3 - Hashing

- What could go wrong?
  - Identical passwords produce identical hashes
  - Once you've cracked a given hash, you can trivially crack it every time you see the same hash again
  - Humans pick bad passwords
    - Frequency analysis
    - Precompute massive tables for popular hash functions
    - Common passwords are very common!
    - Even a small table cracks most passwords
    - Updated version

Clicker Question 1

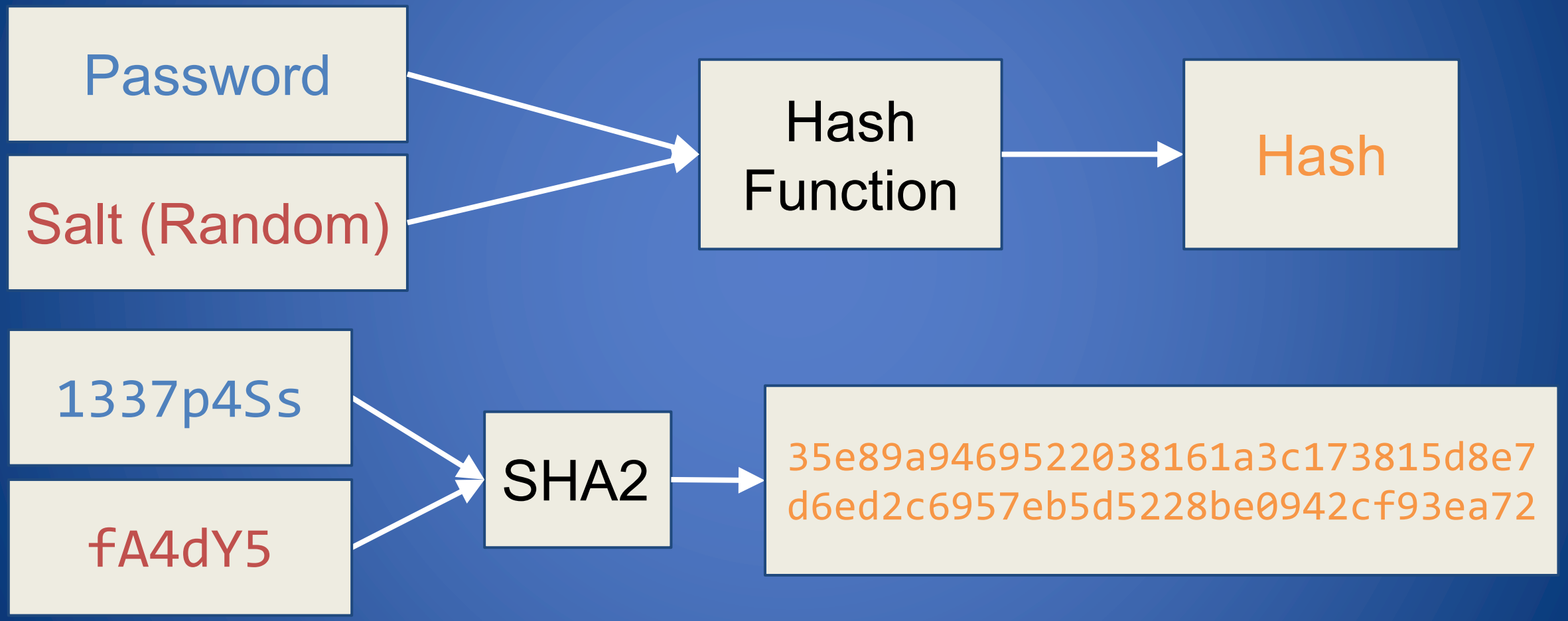
Mallory steals a database of encrypted passwords (but cannot steal the key). Could she recover the plaintext passwords?

A. Yes, all of them
B. Yes, a fraction of them
C. No, since the database is encrypted
D. No, since it is computationally infeasible

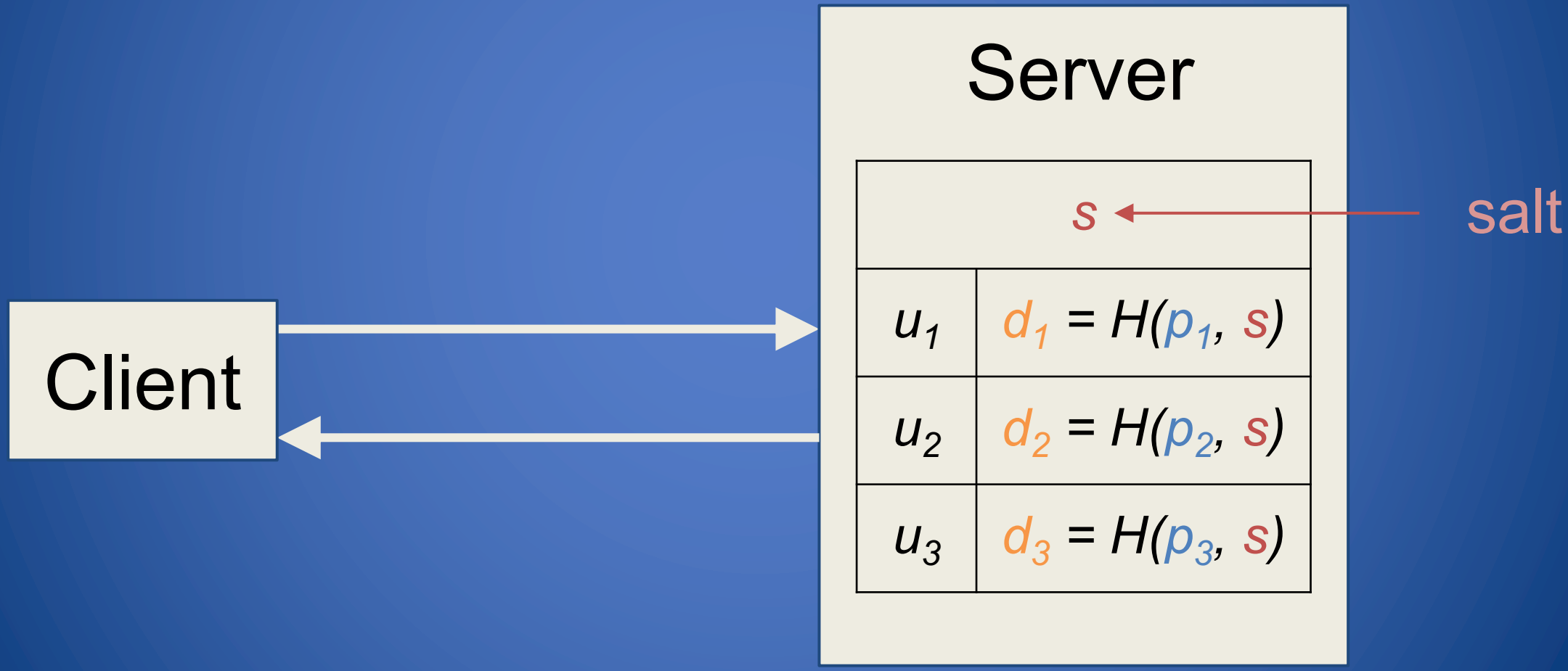
Clicker Question 1 - Answer

- Answer: B
  - Identical passwords produce identical ciphertexts
  - If you know one password, you know all passwords same ciphertext
  - Humans pick bad passwords and hints
    - Frequency analysis (0.5% of users use `password`)
    - Password hints (e.g., numbers 123456)
  - Unique passwords with good hints are safe

Attempt #4 - Salting



Attempt #4 - Salting



Attempt #4 - Salting

- Store hash of salted password
- Hash the password and salt, then compare
- Advantages
  - In order to precompute, need password and salt
  - Since salts are random, guessing salt is useless
  - Even if salt is known, computation must be redone for every site

Clicker Question 2

Using hashing and salting to store passwords, the server successfully defends over frequency analysis attacks.

A. True.
B. False.
C. Not enough information.

Clicker Question 2 - Answer

Using hashing and salting to store passwords, the server successfully defends over frequency analysis attacks.

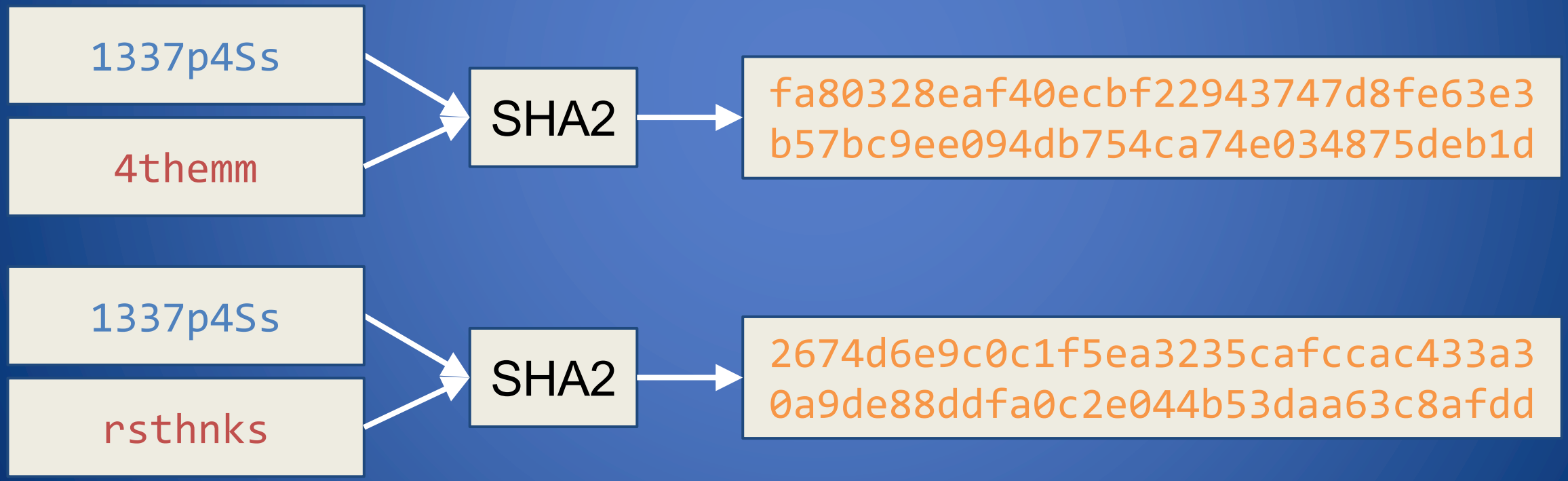
A. True.
B. False.
C. Not enough information.

Attempt #4 - Salting

- What could go wrong?
  - Identical passwords and identical salts produce identical hashes
  - Humans pick bad passwords --> frequency analysis
    - If you crack one password, you crack all identical ones
    - For big sites, precomputation is worth it

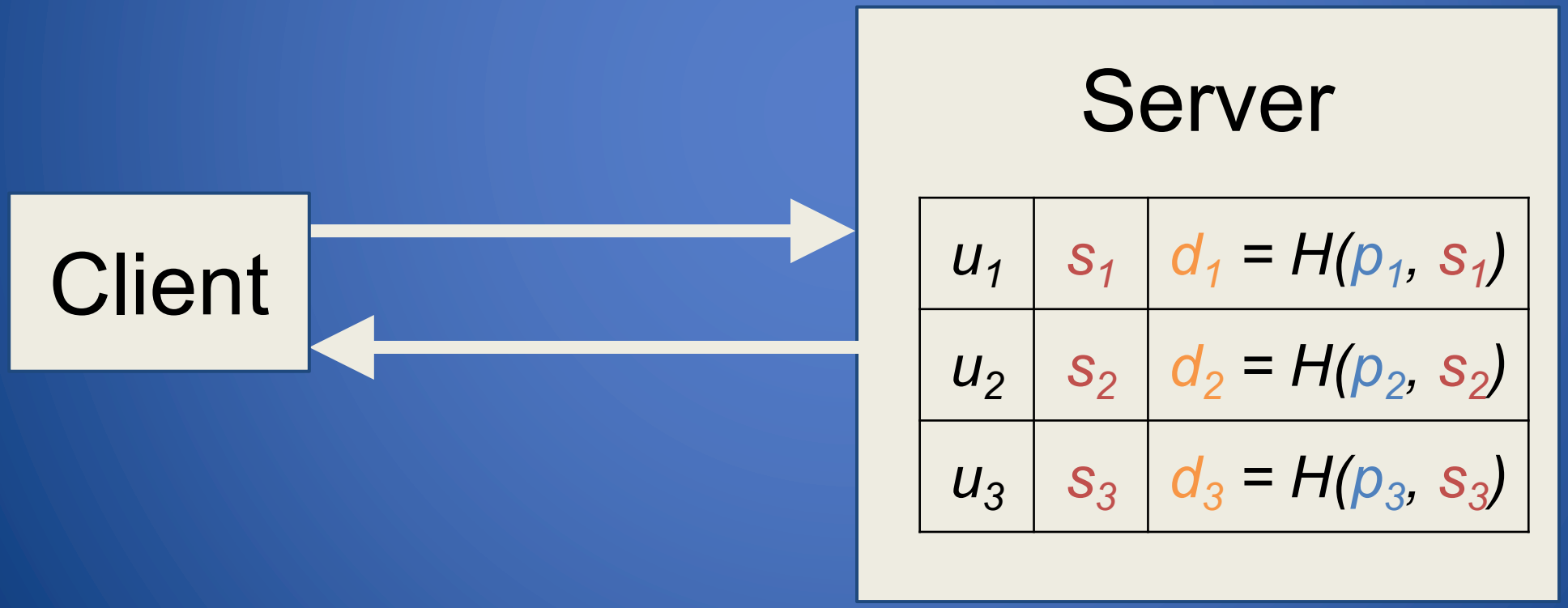
Attempt #5 - Salting

- Hashing same password with different salt will produce different hashes



Attempt #5 - Per-User Salting

- Generate a salt, hash the password, store salt and hash
- Hash the given password with the user's salt and compare



Attempt #5 - Per-User Salting

- Generate a salt, hash the password, store the hash
- Hash the given password with the user's salt and compare
- Advantages
  - Since every user has different salt, identical passwords will not have identical hashes
    - No frequency analysis
    - No using known passwords to crack other passwords
    - No precomputation, hence much harder to crack

Password Cracking

Standard Authentication in OS

butch

Beaver

Password file

...
butch 4566178...

find

hash and compare

The user inserts username and password into the login window

The system looks up the username compares the hash of the password with the stored hash

Storing Passwords

Beaver

cryptographic hash function

butch

Password file

```
...
butch:a4566178ab4f56a3f90968e210b95bca
sundance:2d0f1a836407eb204c0a9186687b713e
...
```

Most common scenario

- The hacker is able to get usernames and password hashes

Location of password file

- Windows (32 bit): C:\WINDOWS\system32\config\SAM
- Linux: /etc/passwd and /etc/shadow
- Mac OS X: /var/db/dslocal/nodes/Default/users/<username>.plist

Password Cracking Methods

- Brute force
  - Try all passwords in a given space
  - Eventually succeeds given enough time and CPU power
- Dictionary
  - Precompute hashes of a set of likely passwords
  - Store (hash, password) pairs sorted by hash
  - Fast look up for password given the hash
  - Requires large storage and preprocessing time
- Rainbow table
  - Partial dictionary of hashes
  - More storage, shorter cracking time

Cracking passwords with Hashcat

What is Hashcat?

Hashcat is the self-proclaimed free fastest password recovery tool.

Benchmarks:

```
./hashcat -b
```

Brute Force

- Try all passwords in a given space
  - Parallelizable
- Eventually succeeds given enough time and computing power
  - Best done with GPUs and specialized hardware (FPGAs, or Asic)
- Large computational effort for each password cracked

Brute Force Cracking

- The attacker has 60 days to crack a password by exhaustive search
- How many hash computations per second are needed?
  - 5 characters: 1,415
  - 6 characters: 133,076
  - 7 characters: 12,509,214
  - 8 characters: 1,175,866,008
  - 9 characters: 110,531,404,750

Dictionary Attack

- Precompute hashes of a set of likely passwords
  - Parallelizable
- Store (hash, password) pairs sorted by hash
- Fast look up for password given the hash
- Requires large storage and preprocessing time

Setup: Dictionary Attack

STEP 1: Make a plaintext password file of bad passwords (called `wordlist`):

```
bernardo12345
letmein
zaq1zaq1
```

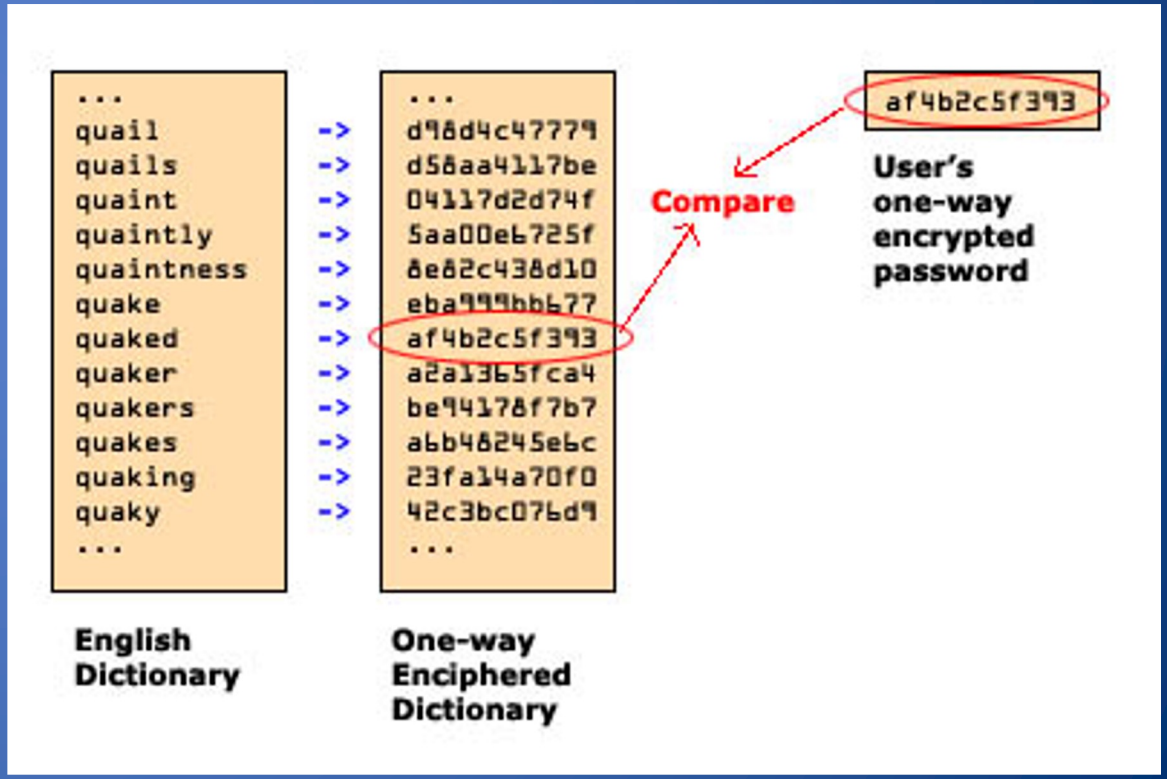
STEP 2: Generate MD5 hashes:

```
for i in $(cat wordlist); do
      echo -n "$i" | md5sum | tr -d " *-"; done > hashes
```

STEP 3: Get a dictionary file. (We're using rockyou.txt which lists most common passwords from the RockYou hack a couple years back…)

Dictionary Attack

```
./hashcat -m 0 -a 0 hashes rockyou.txt
```



Cracking: http://web.mit.edu/dheera/www/blur_dictattack.jpg

Hashtypes: -m [NUM]

0 : MD5 [8743b52063cd84097a65d1633f5c74f5]

100 : SHA1 [b89eaac7e61417341b710b727768294d0e6a277b]

1400 : SHA256

Hash-Mode 1800 (sha512crypt 6, SHA512 (Unix)

[6Lw5wXCCssJp3Ei8S$413/7AdrNLD.T/waBp61ItYXa0eUSzQQp3/iM.oiuTCecZAR79GBom2yJlTgC.5Q5p5DHInYY/9AXjRIQ5r6K1]

https://hashcat.net/wiki/doku.php?id=example_hashes

Time-memory Trade-Off

1980 - Martin Hellman

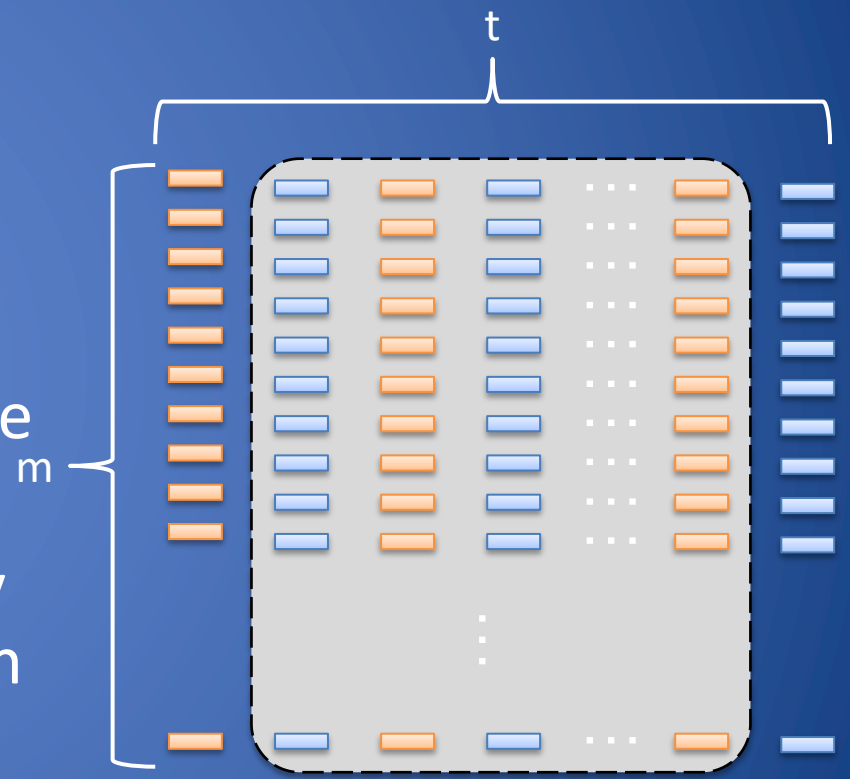
- In this kind of trade-off, you reduce the time you need to crack a password by using a large amount of memory
  - **Benefits**: It would seem more efficient to do the brute-forcing once, store the result, and then use this stored result to accelerate the cracking on any machine.
  - **Flaws**: this kind of database takes tens of memory's terabytes.

Password Cracking Tradeoff

Rainbow Table

2003 - Philippe Oechslin

- In this kind of trade-off, you reduce greatly the amount of memory and increase slightly the time of cracking.
  - A procedure for reduction of a hash database to a much smaller table.
  - Calculate the hash of all passwords, but only store a very small fraction of them in such an order that.
  - project-rainbowcrack.com/table.htm



Attack Modes: -a [NUM]

- 0: Straight (Dictionary attack)
  1: Combination (concatenating words from multiple wordlists)
  2: Toggle-Case (toggling case of characters. You can use rules for this)
  3: Brute-force (trying all characters from given charsets, per position)
  4: Permutation (dictionary generates permutations of itselt)
  5: Table-Lookup (Disabled in later hashcat implementations)
  8: Prince (Intelligent guessing implemented by hashcat)
- https://hashcat.net/wiki/doku.php?id=hashcat

Mask Attack (Brute-Force)

Built-in charsets:

?l = abcdefghijklmnopqrstuvwxyz
?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d = 0123456789
?s = !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
?a = ?l?u?d?s
?b = 0x00 - 0xff

Run (try all lowercase passwords of length 5):

```
./hashcat -a 3 hashes ?l?l?l?l?l
```

Source: https://www.4armed.com/blog/perform-mask-attack-hashcat/

Rule Based Attack

If you have a list of passwords:
password
mysecret
qwerty

What kind of rules are there?

- You can create a rule file with these rules:

  sa@

Rule 1) Substitute 'a' for '@'
p@ssword

$1 $2 $3

Rule 2) Add 123 to end:
password123
mysecret123
qwerty123

$c

Rule 3) Capitalize first word:
Password
Mysecret
Qwerty

Source: https://www.4armed.com/blog/hashcat-rule-based-attack/

Rule Based Attack

Run:

```
./hashcat -m 0 hashes rockyou.txt -r ruledemo --debug-mode=1 --debug-file=matched.rule
```

Running `cat matched.rule` will show you all the rules that matched so you can build better rules.

There also lots of built-in rules provided in the default hashcat installation

Intelligent Guessing Methods

- Try the top N most common passwords
  - Check out several lists of passwords on Daniel Miessler's github
- Dictionary of words, names, places, notable dates
- Combinations of the above
- Replace and intersperse digits, symbols
- Syntax model
  - e.g., two dictionary words with some letters replaced by numbers: `elitenoob, e1iten00b, 31it3n00b, …`
- Markov chain model or a trained neural network

Intelligent Guessing

- A 10-character randomly selected password would take years of CPU time to crack
- For any scheme that involves guessing, the time to crack is reduced by guessing intelligently
- Key insight: not all passwords are equally likely
- Idea: try most likely passwords first

eHarmony Hack

In 2012, 1.5 Million passwords were stolen from eHarmony and published online. As a CS166 student you can now hack like the pros and recover some eHarmony passwords using hashcat.

eHarmony uses MD5: so try `-m 0`

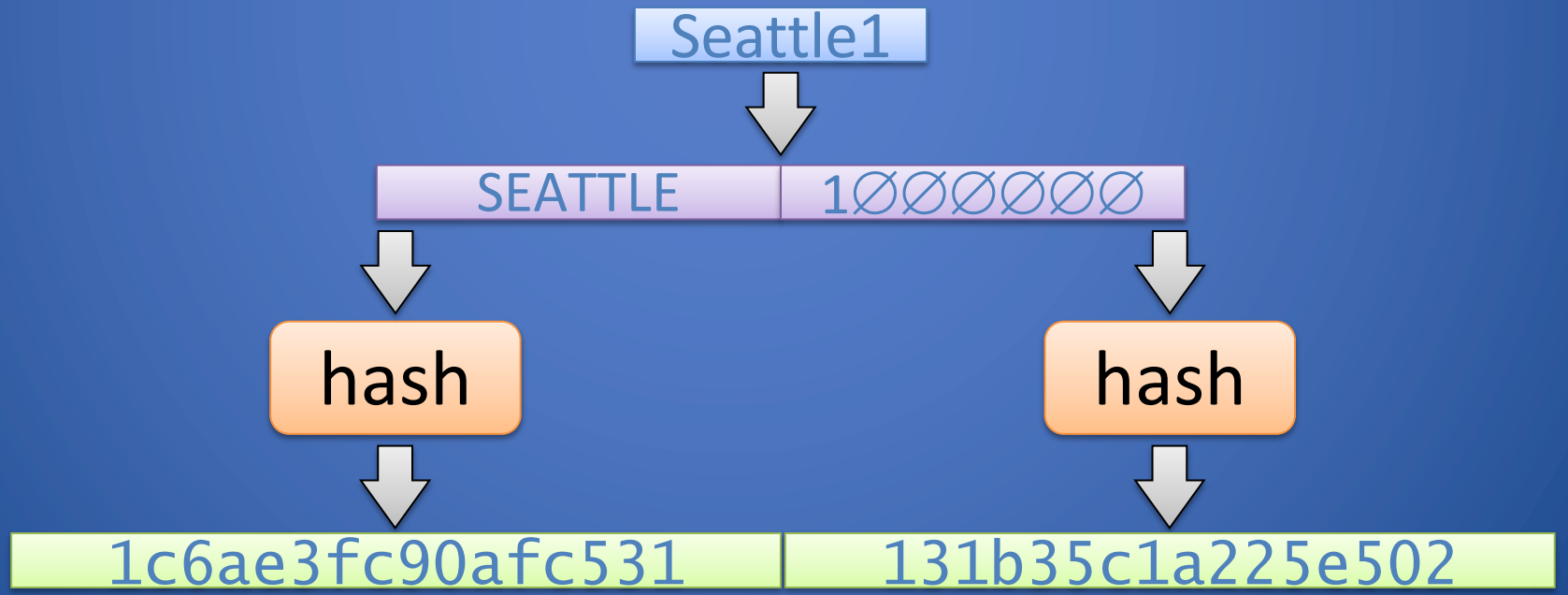
Warning before running command: some passwords in here are offensive

Source: http://www.adeptus-mechanicus.com/codex/hcateasy/hcateasy.php http://www.nydailynews.com/life-style/eharmony-passwords-hacked-1-5-million-users-dating-site-data-compromised-article-1.1091568

Password algorithm issue

Windows XP Password Hashing

- LAN Manager Hash
  - Convert password to uppercase, truncated to length 14
  - Split into two 7-charcter halves
  - Compute 64-bit hash of each half



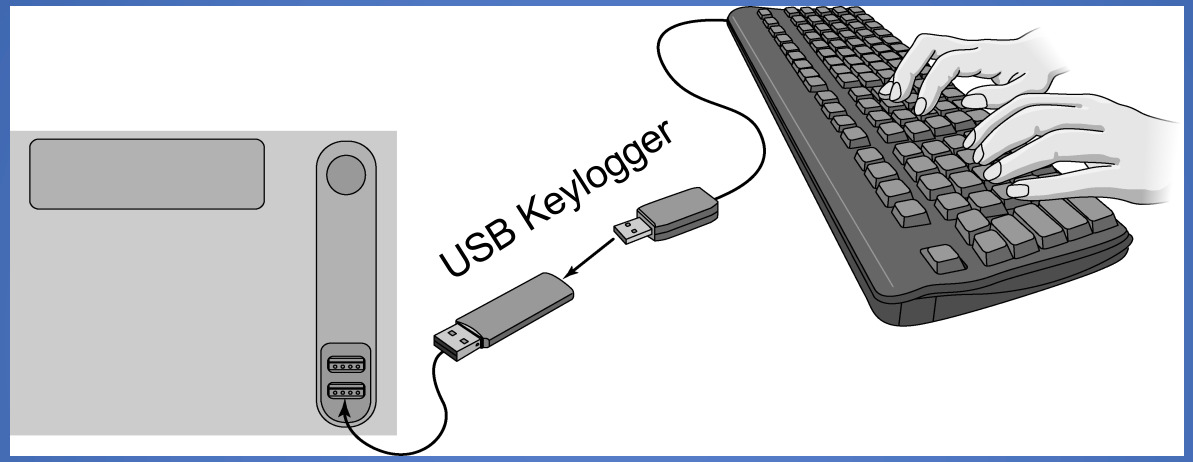
LAN Manager Hash Weaknesses

- Small password space
  - Equivalent to two uppercase passwords of 7 characters
  - About 6.7 trillion possible passwords
- Attack performed with rainbow table
  - 1.4GB storage
  - 14 seconds recovery time
  - 99.9% success rate

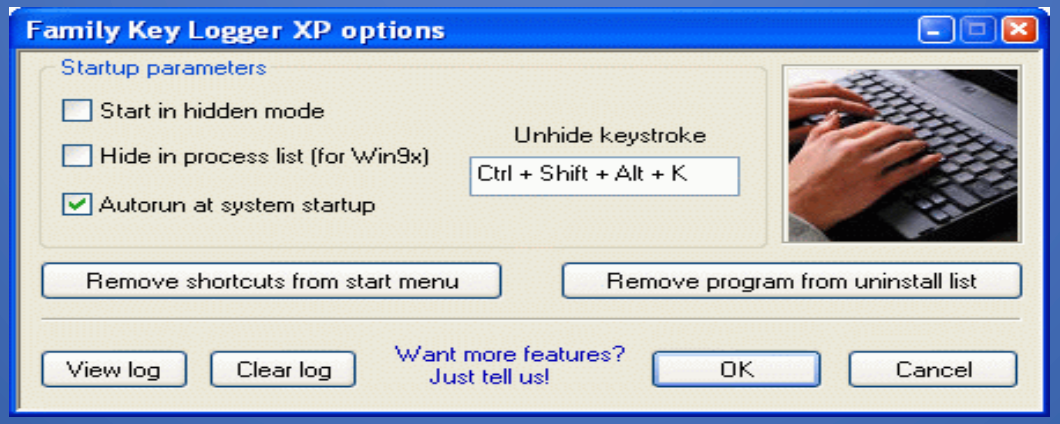
If Cracking does not Work

Other attacks…

Keyloggers

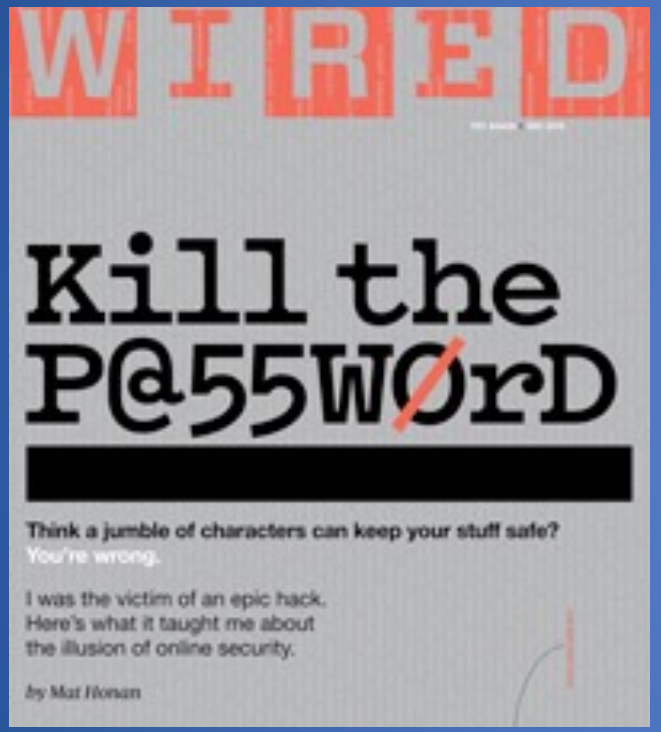


Hardware



Software

Password are still important..



"Hackers destroyed my entire digital life in the span of an hour" 12 - 2012
Mat Honan Wired senior writer

- http://y2u.be/Srh_TV_J144

What We Have Learned

- Password authentication
  - Principles and attack vectors
- Password storage methods
  - Use salted hashes
- Password cracking
  - Brute-force
  - Dictionary precomputation
  - Intelligent guessing
- Demos
- Ethical and legal issues
  - Compelled password disclosure