O/S & Access Control

Aggelos Kiayias

One system Many users

- Objects that require protection
  - memory
  - I/O devices (disks, printers)
  - programs and processes
  - networks
  - stored data in general

Separation Layers

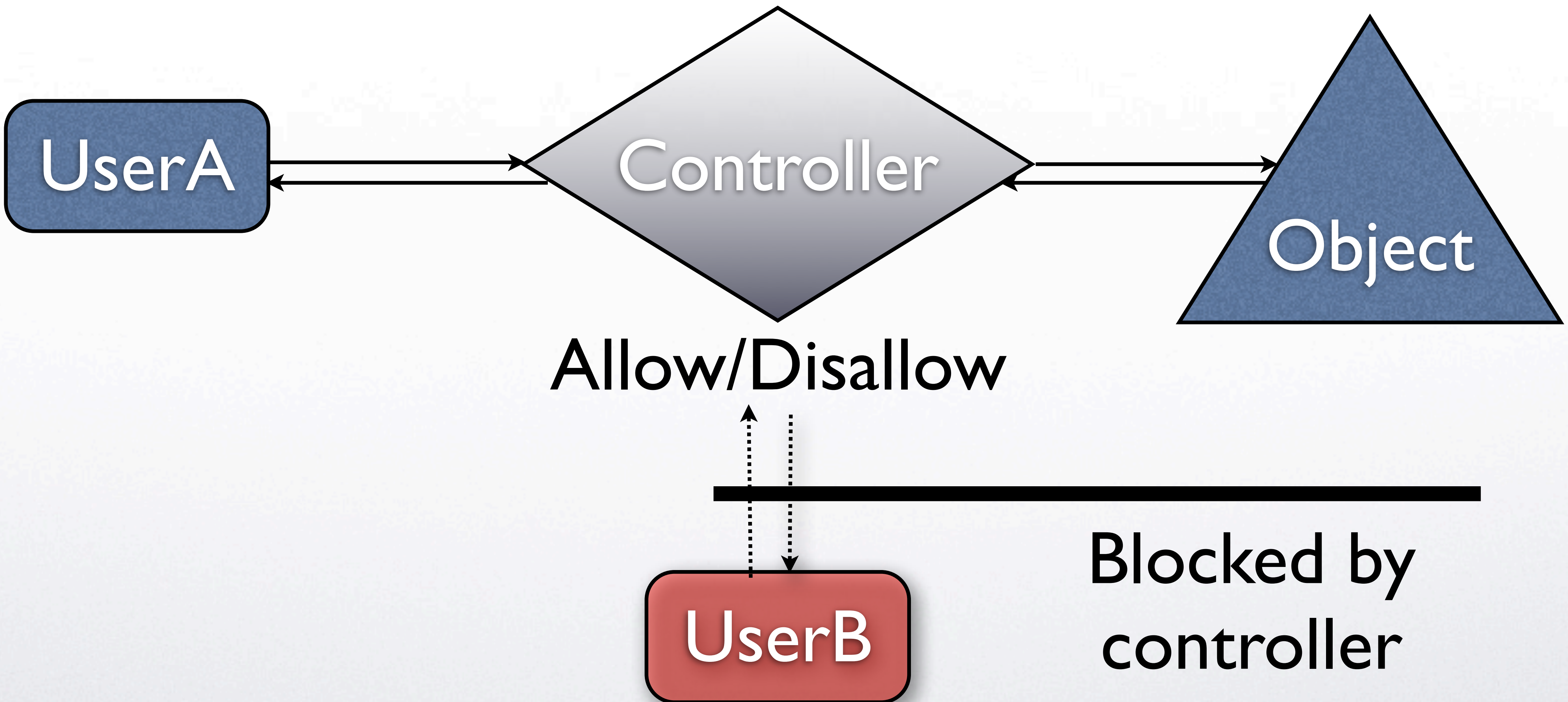
- keeping one user's objects separate from others.
  - *Physically* - Separate devices
  - *Logically* - e.g. Access Control Lists
  - *Cryptographically* - Encryption

Sharing is Inevitable

- Users need to coexist on a single system:
  - Two extremes:
    Monolithic: all users share a single account

    Isolation: every resource is assigned to strictly one user
- In real world systems, a blend of the above is typically used

Logical Access Control



Focus: file system

- A paradigm for access control.
- All objects can be thought as files (*NIX).

Access Control List

- Each object has an ACL.

Directory

File1	o
File2	o
File3	o

pointer

File1 ACL

User A	Read
User B	Read and Write
Other	no access

Windows NTFS (5+)

- ACL is stored with every file.
- Contains users and groups and corresponding permissions for each.
- Folder permissions:
  Read, Write, List, Read & Execute, Modify, Full Control

In Unix/Linux

- Processes make requests to access resources.
- Each process is associated with a **uid**.
- Each file has an ACL that contains a triple of
  rwx rwx rwx
  user group other
- The ACL contains both *user* and *group* info.
- **x** is *execute* for <u>files</u> and *access* for <u>dirs</u>.

Processes

- Objects are accessed by processes.
- How is a process assigned a **uid**?
  - It is created by a parent process and inherits the **uid** of the parent process.
  - But how does a user access a system?

Login Process (console)

- Prompt process: *invites user* (cf., getty running as root)
- and *challenges user to authenticate*.
  - if login is unsuccessful, restart the prompt
  - if successful an interface process is spawned that inherits the **uid** and **gid** of the authenticated user.

Temp Acquired Permission: *suid bit*

- How is it possible to allow a certain **uid** to peep into a higher access level via an executable?
- When an executable has the *suid bit* set, an executed file inherits the **uid** of its *owner* rather than the **uid** of the *caller*. E.g.,

-rwsr-xr-x 1 root wheel 32680 2013–10–11 12:13 passwd

Separation

- in a multi-user environment
  - Access-control as described so far offers a logical separation; is this foolproof?
  - What would a cryptographic separation offer?

User Authentication

- Can be based on:
  - Something the user *knows*. (e.g. Password)
  - Something the user *has*. (e.g. Physical Token)
  - Something the user *is*. (e.g. Fingerprint)
- Two Factor Authentication (or Multi Factor)

Password-based Auth

- Authentication based on what a user *knows*.
- O/S must keep a database of username/password pairs.
- Where to store it?
- What to store?

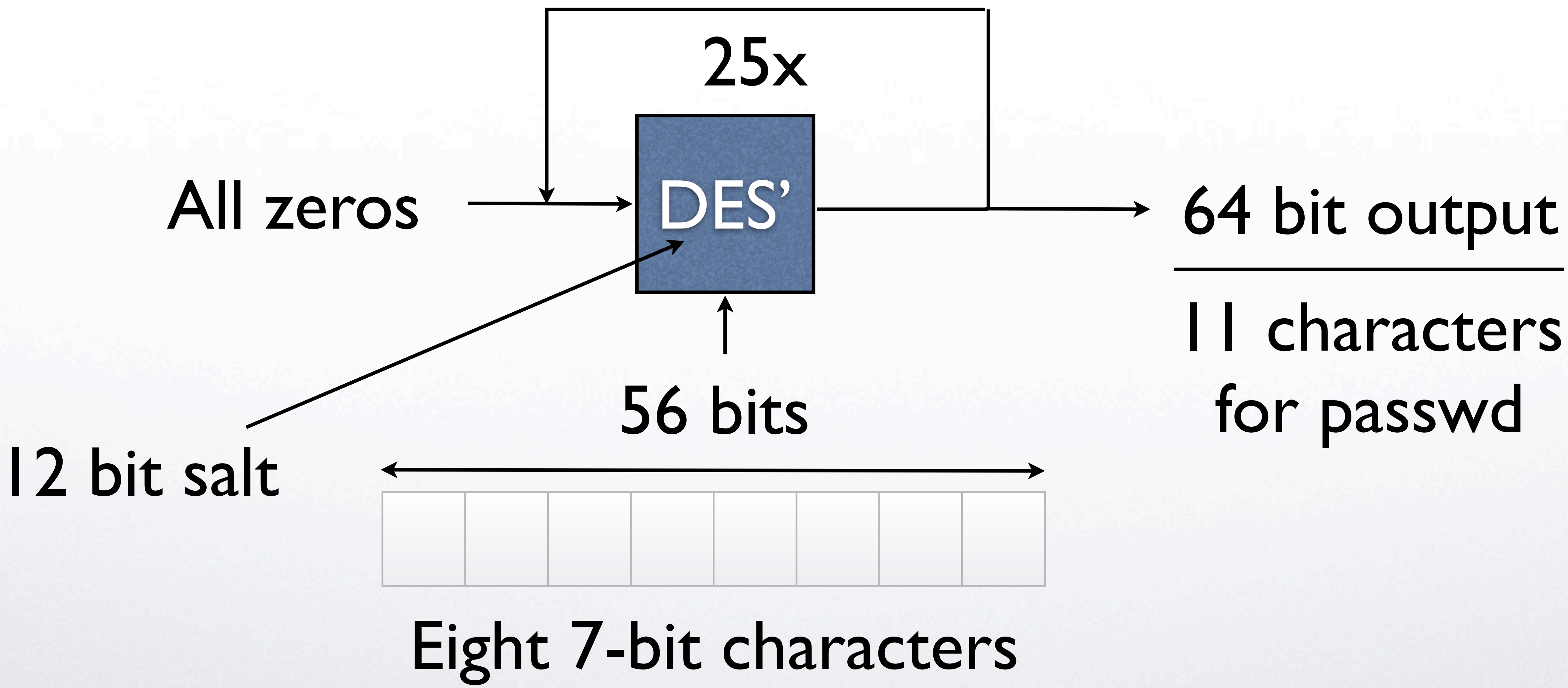
The /etc/passwd file in UNIX

- FORMAT
  Name:Password: UserID:PrincipleGroup:Gecos: HomeDirectory:Shell
- guest:AvCSyg9e75YZM:200:0::/home/guest:/usr/bin/sh
- One line for each user.
- The file is publicly readable.
  - In current deployments the passwords are *shadowed* in another location (e.g., /etc/shadow) -- this file is not publicly readable.

Storing passwords

- Should passwords be stored in the clear?
  - No, use a one-way transformation.
  - Can be based on a hash function.

The old crypt() function



Examples

- crypt(”password”,”Ee”) = EeAJqAJ0sluG.

  salt

- crypt(”password”,”4!”) = 4!wpbYhg6W8qM
- crypt(”password is what some people choose but I chose a passphrase!”,”4!”) =4!wpbYhg6W8qM

  this results in a collision since the DES’ based crypt function only used the first 8 characters

The glibc2 extension

- If salt starts with 1 followed by at most 8 characters, terminated by $; then it is not using the DES based algorithm.
- MD5 based algorithm with 22 char output from [a-zA-Z0-9./].
- entire password is now significant.

Examples

- crypt(”password”,”1GoodSalt”) =

  1GoodSalt$czxN1PirYBY5pqE1Q98el.

- crypt(”password is what people choose but I chose a passphrase”,”1GoodSalt”) = 1GoodSalt$Obp/S5k35O0r1ymT0v9t./

- currently $2y$=Blowfish,5=SHA-256, 6=SHA-512

  - *test on linux:* perl -e 'print(crypt("password","\$1\$GoodSalt")."\n");'

In Windows?

- Security Accounts Management Database (SAM) stored in the registry (hive).
- It stores hashed copies of user passwords.
- The database itself is encrypted with a locally stored system key.
- It is possible to store this key elsewhere.
    - Attack against NT4.0, 2000 if SAM was deleted one gets a free login.
    - **check:** ***Offline NT Password and Registry editor*** **or** ***(NTPASSWD live linux CD)***

Online Dictionary Attack

- Given a dictionary of possible passwords.
  - you have a way to test whether a guessed password is correct.
  - e.g., you have access to console login, or you have the password hash and the salt.
- *Salting is not intended to protect against this attack*.

check: John the Ripper (JTR) password cracker http://www.openwall.com/john/

Codebook Dictionaries

- Produce a "codebook dictionary"
  - Apply one-way transformation to each candidate pwd.
- Sort according to transformation output

linear in dictionary size

Offline!

- Given the password hash, binary search through the codebook dictionary.
  - *the online cost is **low**! but salting can really make a difference against **this** attack.*

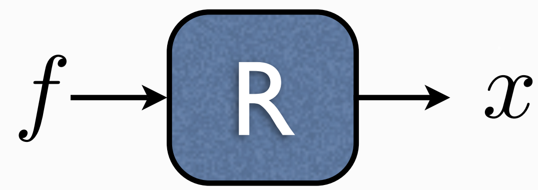
logarithmic *in dictionary size*

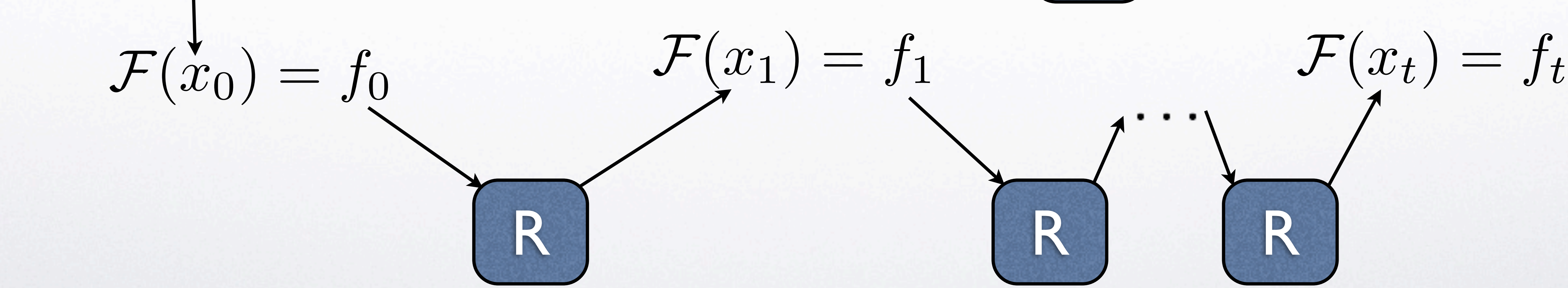
Online!

Time-Memory Tradeoff

- Can we do something between the previous approaches? Build *rainbow* tables to invert any function $\mathcal{F}$ (e.g., crypt) with output value f (the obfuscated password)

Consider the "magic" function R:





Storage reduction: $(\underline{x_0}, f_0), (x_1, f_1), \ldots, (x_t, \underline{f_t})$

Time-Memory Tradeoff

Create the rainbow table by sorting according to end of chains

offline

online

Given $f = f[0]$ calculate $f[1], \ldots, f[t]$ applying $\boxed{\text{R}}$

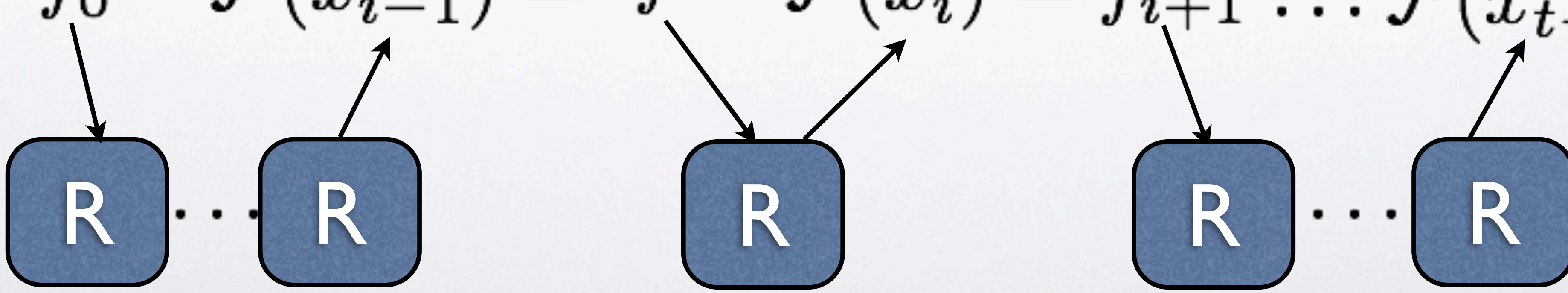
Perform binary search in the codebook dictionary
for each of $f = f[0]$, $f[1], \ldots, f[t]$
every chain hit gives a candidate password

Tradeoff: Dictionary size has been reduced by size ~ t
searching time has been multiplied by ~ t

Time-Memory Tradeoff

- How to recover the password after you hit end of chain?
- Start from the beginning of chain.

$$\mathcal{F}(\overset{x_0}{\downarrow}{x_0}) = f_0 \cdots \mathcal{F}(x_{i-1}) \overset{?}{=} f \quad \mathcal{F}(x_i) = f_{i+1} \ldots \mathcal{F}(x_{t-1}) = f_t$$



Time-Memory Tradeoff

- Tight tradeoff is contingent on a good choice of R
- Too few/short chains may not cover the full dictionary.
- Too many/long chains will overlap and waste space/time.

Time-Memory Tradeoff

- Even possible to model $\boxed{R}$ to produce "human" passwords, i.e., consider those chains for which it holds that *x* follows a certain distribution

Time-Memory Tradeoff

- Rainbow tables
  - are a very powerful technique if applied against unsalted hashes - can break any **strong** human memorizable password.
- Implementations: *Ophcrack, RainbowCrack.*
- ***Random*** Windows NT Lan Manager passwords can be broken in 13 seconds with 1.4 GB tables. [*Oeschlin* CRYPTO '03]

Choosing a Dictionary

- Without salting one is totally vulnerable (even with **random** *but of human-memorizable length* passwords).

How does salting help?

- No Salting : you want to invert $\mathcal{F}(\cdot)$
- Salting : you want to invert a member of $\{\mathcal{F}_s(\cdot) \mid s\}$

It is possible to build rainbow tables that simultaneously walk along passwords and members of $\{\mathcal{F}_s(\cdot) \mid s\}$

Complexity : is multiplied by a factor equal to the family size

Language Entropy

http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf

Language Entropy

- Shannon : *entropy* of English language is 1.5 bits per character for 8-char long words {a...z} *versus* lg(26)=4.7 bits theoretical max for purely random selection.

http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf

Language Entropy

- Shannon : *entropy* of English language is 1.5 bits per character for 8-char long words {a...z} *versus* lg(26)=4.7 bits theoretical max for purely random selection.
- How much entropy do Human memorizable passwords have?

http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf

Language Entropy

- Shannon : *entropy* of English language is 1.5 bits per character for 8-char long words {a...z} *versus* lg(26)=4.7 bits theoretical max for purely random selection.

- How much entropy do Human memorizable passwords have?

- NIST: Using an allowed 94 character alphabet.

http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf

Language Entropy

- Shannon : *entropy* of English language is 1.5 bits per character for 8-char long words {a...z} *versus* lg(26)=4.7 bits theoretical max for purely random selection.

- How much entropy do Human memorizable passwords have?

- NIST: Using an allowed 94 character alphabet.
  - 8 chars: only 18 bits! vs. 52 bits for random

http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63v6_3_3.pdf

Kerberos

The Kerberos Approach

Users and services share private keys with Kerberos

Kerberos is password-based

k_{user}

k_{service}

User

Service

SETUP

Kerberos, II

K_{session}

Random key selected on the fly

User: I want *service*

$\{\text{service}, K_{\text{session}}\}_{K_{\text{user}}}$

$\boxed{\{\text{user}, \text{service}, \text{time}, \text{life}, K_{\text{session}}\}_{K_{\text{service}}}}$

ticket

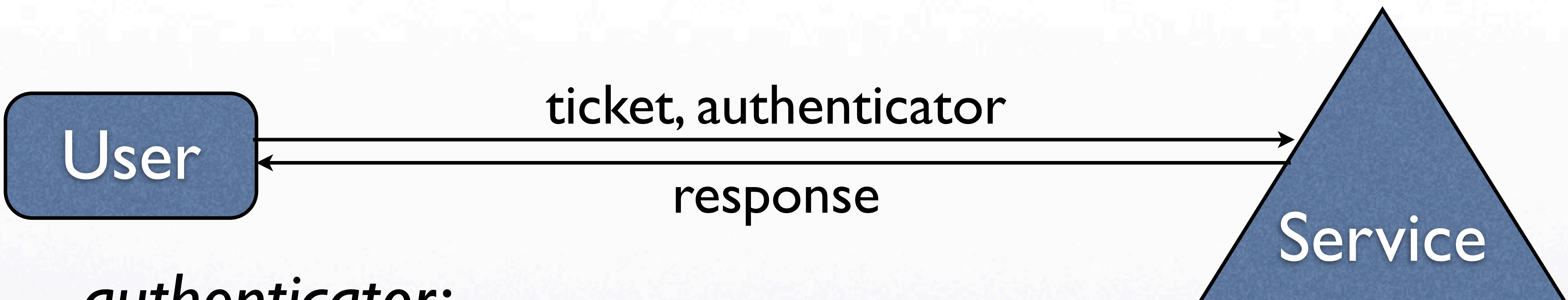
User

Can be used multiple times before expiration

Kerberos, III

ticket:

$$\{\text{user, service, time, life}, K_{\text{session}}\}_{K_{\text{service}}}$$



authenticator:

$$\{\text{user, time}\}_{K_{\text{session}}}$$

Kerberos, IV

- Above description too stressful for Kerberos.
- Easing Kerberos task:
  - Kerberos will recognize only one service, the *Ticket Granting Service*.
  - Instead of giving tickets for every service it will give tickets only for using the TGS.

Kerberos, V

TGS-ticket
can be reused

User: I want *TGS*

$\{\text{user}, K_{\text{TGSsession}}\}_{K_{\text{user}}}$

$\boxed{\{\text{user}, \text{TGS}, \text{time}, \text{life}, K_{\text{TGSsession}}\}_{K_{\text{TGS}}}}$

User

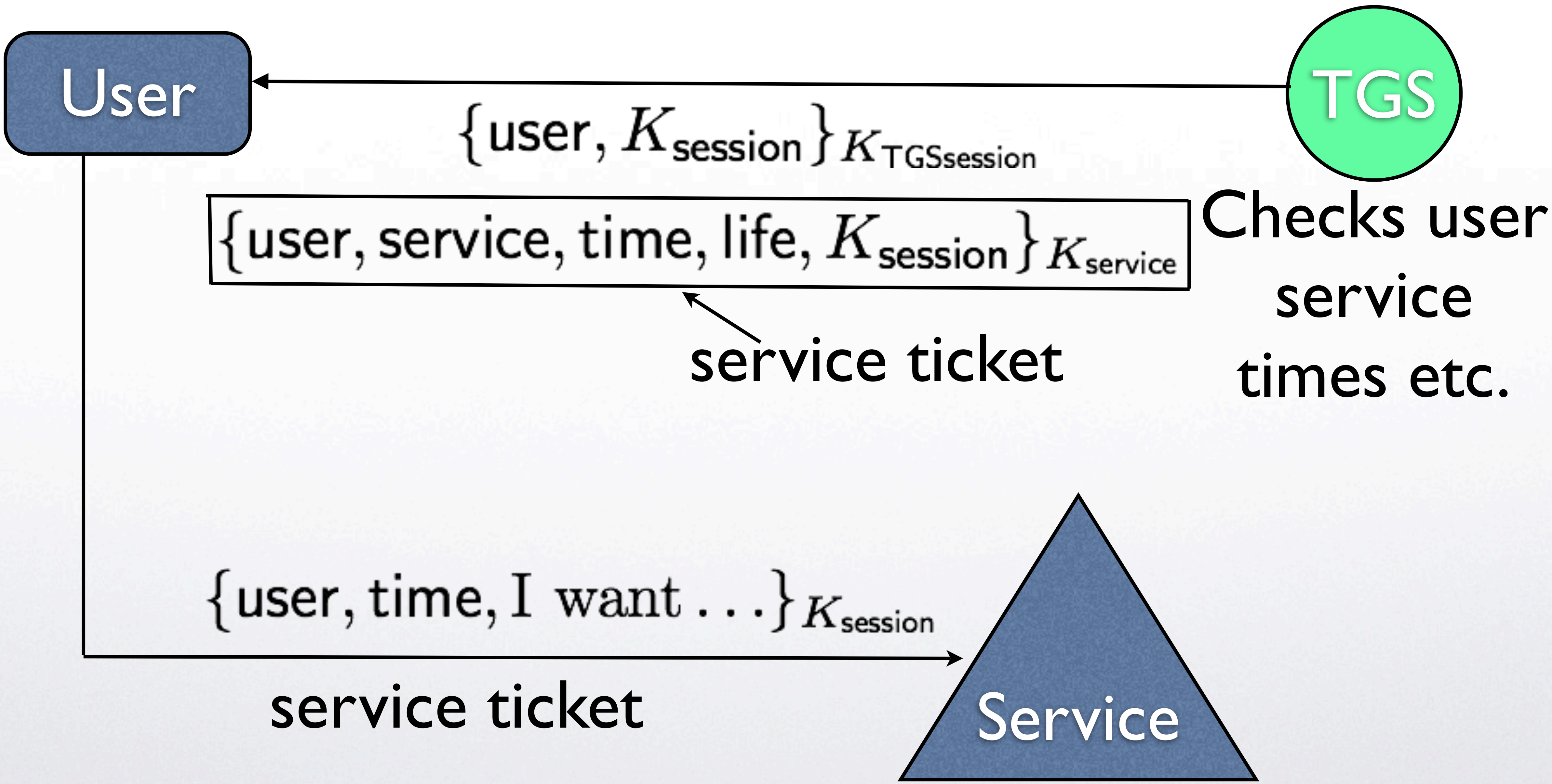
TGS

$\boxed{\{\text{user}, \text{time}, \text{ I want service}\}_{K_{\text{TGSsession}}}}$

and

$\boxed{\text{TGS-ticket}}$

Kerberos, VI



User
TGS
$\{\text{user}, K_{\text{session}}\}_{K_{\text{TGSsession}}}$
$\boxed{\{\text{user, service, time, life}, K_{\text{session}}\}_{K_{\text{service}}}}$
service ticket
Checks user service times etc.
$\{\text{user, time, I want} \ldots\}_{K_{\text{session}}}$
service ticket
Service

Kerberos VII

- Kerberos server knows all user keys and the TGS key. It handles user authentication.
- Ticket Granting Server knows service keys. It handles user requests to access services.
- Kerberos does not need to know about system services. TGS does not need to worry about authenticating users.

Kerberos VIII

- Where do keys come from?
  - user keys are derived from human passwords.
  - service keys are random and stored locally. assumed to be stored securely.

Kerberos IX

- Kerberos advantages:
  - Human passwords are never communicated. Only on the fly usage by local “login” challenge.
  - Mutual authentication between users and services.
- Kerberos disadvantages:
  - monolithic

Kerberos X

- Windows (all the way since 2000) uses Kerberos for authentication services.
- Possible to install for Linux, Unix.
- Mac-OS X has built-in Kerberos support.

