CS 419: Computer Security

Recitation: week of 2020-09-14

Project 1 Discussion

TA: Shuo Zhang
Paul Krzyzanowski

Background: OS access control

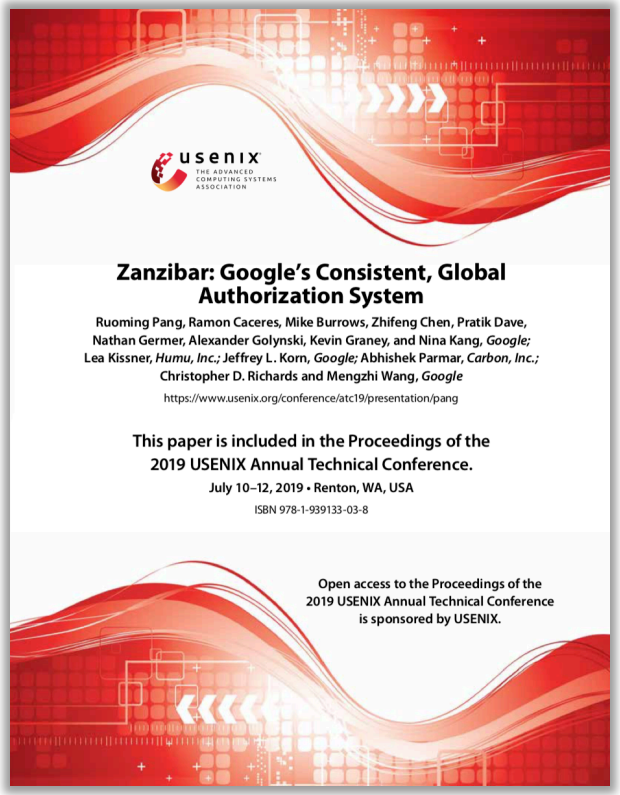
- Operating systems have traditionally been responsible for enforcing access permissions
- Example: access control lists
- Operating system checks whether a user (subject) is allowed to access a file (object) when the user's process opens a file

Background: OS-based Access Control Problems

- **But this does not work for network services**
  - You may have an account on eBay but you don't have an account on any of eBay's computer systems
  - How does the service know what you should be allowed to do?
  - How do you log in?
- **It also does not work for application-aware operations**
  - An operating system controls read/write/append/execute permissions for files but does not know the meaning of the data
  - The operating system doesn't know application-specific actions
    - Are you allowed to delete an employee from a database?
    - Do you have rights to stream video content from a video server?

Background: What can be done?

- **Applications end up managing their own authentication & access control**
  - Authentication for users that access services
  - Access control to decide who is allowed what access to an object
  - This risks introducing extra vulnerabilities: programmers can make implementation mistakes
- **In 2019, Google introduced Zanzibar**
  - A consistent authorization system for Google services world-wide
  - Applications can now use a single system to manage permissions



usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Zanzibar: Google's Consistent, Global Authorization System

Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, and Nina Kang, *Google;* Lea Kissner, *Humu, Inc.;* Jeffrey L. Korn, *Google;* Abhishek Parmar, *Carbon, Inc.;* Christopher D. Richards and Mengzhi Wang, *Google*

https://www.usenix.org/conference/atc19/presentation/pang

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

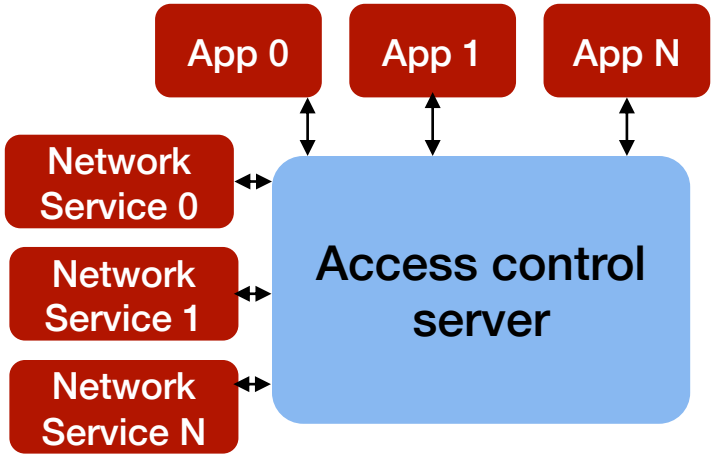
ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Project goals

The goal of this project is to build an authentication & access control service that any program can use so programmers will not have to write their own for each new application

- **Ideally, this would be a network service that could be shared among anyone who wants to use it**
- **For simplicity, we will write the service as one program**
  - It can easily be wrapped network serviceas a



What the program does

- **Authentication**
  - Manage users and passwords
  - Check if a password matches a user account

- **Access control**
  - Manage access permissions
    - Arbitrary operations on objects
      - They don't have to be read/write/execute
      - They can be anything the service needs: view, destroy, invert, hide, …
    - Similar to type enforcement model
      - Each user can be assigned one or more labels – called a domain ⇒ user groups
      - Each object can be assigned one or more labels – called a type ⇒ object groups
      - Access control matrix of domains and types

Types

Domains	Employee_data	Videos	Audio
Admin	read, write	read	read
Editor	read	download, write, delete	listen, delete
User		stream	stream

Authentication

- **Create (user, password) data**
  - The command
    ```
    portal AddUser alice monkey
    ```
  - Will add a user named `alice` with a password `monkey`
    ```
    Success
    ```
- **Authenticate a password**
  - Check if the password belongs to bob:
    ```
    $ portal Authenticate alice monkey
    Success
    $ portal Authenticate alice password
    Error: bad password
    ```

See the assignment writeup for details on error conditions

Implementation advice

- **Since the program exists after each command, you will need to store the data in a file (or multiple files)**
- **There are many, many ways to do this**
  - Multiple individual files in a directory structure
    - Example: `accounts/alice` will store the password for `alice`
  - A single file with a user & password on each line (e.g., a comma-separated list):
    - Example: The file accounts can contain

```
alice,monkey
bob,password
```

  - Store one or more data structures in a serial format
    - Python pickle serialization, Python PyYAML, Java Serializable, Google Protobufs
  - Keep it simple.
    - You can use an external library (e.g., to parse CSV data) but make sure it's included in your submission or it is available on the iLab systems
    - If any tools need to be compiled or build, include the build in a makefile or some make script

Domains

- **A domain is a grouping of subjects (users)**
  - A user may belong to multiple domains
  - You can think of a domain as an attribute of a user
- **Domains make access control management easier because the administrator does not have to think of individual users, just categories of users**
- **Two commands:**

```
$ portal SetDomain alice admin
Success
```

  - Adds `alice` to the domain `admin`

```
$ portal DomainInfo admin
alice
```

  - Shows all the users in that domain – may be none
  - If there is no such domain then there are no users in it so list nothing

Domains – Implementation advice

- **You need to store a list of 0 or more domains**
- **Each domain will contain 1 or more users**
- **As before, any storage structure works**
- **One really simple implementation is to create a file per domain**
  - Example: the file `domains/admin` can contain a list of users, one per line

Types

- **A type is a grouping of objects (e.g., files)**
  - An object may have multiple types associated with it
  - As with domains, you can think of a type as an attribute of an object
- **Like domains, types make access control management easier because the administrator does not have to think of individual objects, just categories**
- **Two commands:**

```
$ portal SetType mulan.mp4 videos
Success
```

  - Adds `mulan.mp4` to the domain `videos`

```
$ portal TypeInfo admin
mulan.mp4
```

  - Shows all the objects in the given type – there may be none
  - If there is no such type then there are no objects in it so list nothing

Types – Implementation advice

- **The implementation of types is *exactly* the same as domains!**

Access

- **The core of the service is:**
  - Add a permission for a domain to an object
  - Deciding whether a user has specific permissions for an object
- **Access control is (logically) an access control matrix**
- **Example**

```
$ portal AddAccess download Editor Videos
Success
```

  - Adds `download` permission for the set of objects in `Videos` to the domain `Editor`

Domains	Types: Employee_data	Types: Videos	Types: Audio
Admin	read, write	read	read
Editor	read	download, write, delete	listen, delete
User		stream	stream

Access checking

- **Example**

```
$ portal CanAccess download alice mulan.mp4
Success
```

  - Tests if the operation “download” is permitted for user “alice” on the object “mulan.mp4”
  - This means there must exist some domain that alice is a member of that has a “download” operation for some type that mulan.mp4 is a member of

Types

Domains	Employee_data	Videos	Audio
Admin	read, write	read	read
Editor	read	download, write, delete	listen, delete
User		stream	stream

Access checking

- **We can check if a user has access on an object by iterating over domains and types:**

```
for d in domains(user)
    for t in types(object)
        if access[d][t] contains operation
            return true
return false
```

Types

Domains	Employee_data	Videos	Audio
Admin	read, write	read	read
Editor	read	download, write, delete	listen, delete
User		stream	stream

Access implementation

- You will need to think about how to store the access matrix
- A two-dimensional structure is simple if you store the matrix as a serialized object (e.g., JSON, Java serialization, etc.)
- If you want to use files, you can create a directory hierarchy where each
  `domain_name/type_name`
  file contains a list of allowable operations for that (*domain*, *type*)

Incremental development

- **This program can be developed incrementally!**
- **Start with user authentication: get that to work**
- **Then implement domains (adding, listing)**
- **Test thoroughly!**
  - You should be able to handle hundreds of users and domains
  - Test for null users and null domains
- **Copy your domains code to handle types**
- **Only then … add access (AddAccess, CanAccess)**

Partial credit

- **You will get no credit if nothing works … even if you wrote a lot of code**
  - That's why incremental development & testing is important!
- **Credit**
  - Authentication: 25%
  - Domains & Types: 25%
  - Access management: 50%
- **You will lose points for**
  - Not conforming to the interface
    - We do not have time to figure out how to run your program!
  - Not providing clear instructions on how to compile and run your program
    - We will not have time to figure this out – test by having a friend follow your instructions
  - Not working in a different environment (e.g., using full pathnames)
  - Not handling errors (including checking # of parameters)

The End