1 Overview

In the last lecture we examined Authentication techniques and their vulnerabilities. In this lecture we consider instead access control. In specific we will explore Access Control Matrices and their implementations in the form for Access Control Lists and Capabilities.

2 Access Control

In any multi-user system, there needs to be framework for managing different users’ privilege levels. There needs to be a way to assign differing read, write, and execute abilities for a file for each user. In addition, this system must respect the Principle of Attenuation of Privilege. The Principle of Attenuation of Privilege says that a subject cannot grant permissions they don’t already have themselves. This is necessary as without it, a subject could grant themselves any privileges they desired, negating the entire purpose of having the privileges.

The traditional theoretical model for such an access control system is the Access Control Matrix.

2.1 Access Control Matrix

An Access Control Matrix (ACM) is a two dimensional matrix with a row for each subject $s_i$ and a column for each object $o_j$. Each cell of this matrix stores the permissions the given subject $s_i$ has for an object $o_j$. This matrix thus defines permissions for all users for all objects. Despite its comprehensiveness though, ACMs are fairly rarely ever literally implemented. A literally implemented ACM has the disadvantages that it:

- rapidly grows in size as creating a new object means adding $i$ new cells, one for each subject, and adding a new user means adding $j$ new cells,

- the matrix would be extremely sparse as most users use and access only a small subset of the system’s objects.

<table>
<thead>
<tr>
<th>subject/object</th>
<th>$o_1$</th>
<th>$o_2$</th>
<th>$o_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$r_{1,1}$</td>
<td>$r_{2,1}$</td>
<td>$r_{3,1}$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$r_{1,2}$</td>
<td>$r_{2,2}$</td>
<td>$r_{3,2}$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$r_{1,3}$</td>
<td>$r_{2,3}$</td>
<td>$r_{3,3}$</td>
</tr>
</tbody>
</table>

Given the impracticality of a literal implementation of an ACM, we instead use minimized more efficient representations. The primary two options are Access Control Lists and Capabilities.
3 Access Control Lists

An Access Control List is a list of permissions $r_i$ that each subject $s_i$ has for a given object $o$. It is essentially the column for a single object from an ACM. We can define a function describing an ACL:

- A given an object $o$ has a list of $i$ tuples $(s_i, r_i)$
- $acl(o) = \{(s_i, r_i)\}$

In a system using ACLs for its access control, each object will have an associated ACL defining its permissions.

3.1 Considerations

Though ACLs are an effective solution, they are not free of issues, notably the question of access to the ACL objects themselves.

**Question:** Who can modify an ACL? Do ACLs apply to privileged users?

In Unix, two users have the power to modify the ACL of an object. The owner of an object has top priority and has the ability to modify their object’s ACL so as to restrict or grant other users’ access to it. In addition, Root has the power to modify ACLs so as to control access to system objects.

**Question:** Is it possible for permissions to conflict within an ACL?

In Windows, unlike in Unix, access can not only be granted, but can also be restricted. The fact that an object has both a white and a black list means that these must kept in agreement. Otherwise, a subject might be both allowed and restricted from accessing an object.

4 Capabilities

Alternately, we can store the ACM row by row. Each row is a capability: $cap(s_i) = \{(o_i, r_i)\}$, where $s$ is the subject, $o$ is the object, and $r$ is the right.

4.1 Capabilities vs. ACLs

Unix permissions/ACLs are stored and owned by the system, so they have inherent tamper protection. However, capabilities are stored with the user, so they can be tampered with. As a result, capabilities require tamper protection so malicious users cannot simply create their own capability.

4.2 Forgery Protection

We can try to restrict users from escalating their privileges by only allowing them to add permissions in directories they own. However, this is not versatile enough, as users may want to create directories in public folders like /tmp.
4.2.1 Capabilities as a Collections of Bits

The capability object is represented somewhere in memory as a collection of bits. In many cases, we can create new objects from collections of bits we have selected or modified, removing any security we had from the capabilities representation. We can create objects from bits easily with:

- c++ (casting)
- python (pickling), in addition to nonexistant object protection

So, we need a way to prevent users from constructing their own capability objects.

4.2.2 Preventing Users From Crafting Capabilities

We discuss three methods for preventing capability forging.

**Language-based Mechanisms:** Some languages, such as java, have private constructors and object protection mechanisms built into the JVM. These prevent arbitrary users from using our own tools to create their own capabilities.

**Protected Memory:** We explore using memory protection to prevent capability tampering by exploring file descriptors, which serve as capabilities in unix.

```plaintext
int fd;
fd = open(...);
read(fd);
```

File descriptors are used to index into a table of open files maintained by the kernel on a per-process basis. While we can arbitrarily modify the descriptor given to read() in the preceding example, we can’t access any files not already on the table, as it is maintained in protected kernel memory.

**Cryptography:** We discuss a case study, where we would like to make links so users of a photo upload site can securely share photos without friends needing to create an account, beginning by brainstorming potential solutions.

5 Cryptography Case Study: Sending a Photo

5.1 Failed ideas

1. Just have the username and the image name in the link.
   *We can see other users pictures by guessing common filenames.*

2. Hash user/filename.
   *For simple strings like photo filenames, hashes can be rainbowed back. It’s also hard for the server to reverse if it doesn’t have all the hashes stored (it would have to try all filenames).*
3. Hash the content of the image: https://xxx.com/h(data).
   We would have to store a ton of hashes, and identical images from different users are a problem as they will hash the same.

4. Add user info to the hash: https://xxx.com/h(data, username, password).
   We don’t neccesarily have all of that information present, and there may be a directory heierarchy our link has to reference past just the filename.

5. Include the file path and encrypt: https://xxx.com/E_k(/path/to/file).
   The server doesn’t know the user when it gets the request.

   A solution is to introduce the username again: https://xxx.com/E_k(/path/to/file)/user

   This works, but now the user must maintain a shared key with the server. Upon consideration, we realize that the key doesn’t have to come from the user; however, there does need to be a different key for each user to prevent a replay attack where the username is changed to try to access the same file from a different user. We now have several other considerations.

5.2 Considerations

Problem: The system now has to decrypt $E_k(/path/to/file)$ for every request, which can be an issue for huge systems.
Solution: We can just send the plaintext as well to reduce lookup time, and use the cyphertext to verify.

Problem: The link has to include the entire encrypted path, so it will likely end up long and messy
Solution: We can send a MAC or digitally sign the link. As we want to avoid sharing a private key with the user, we can use CVC-MAC created serverside, which only requires a public key.

Final Form: https://xxx.com/path_to_file/MAC_k(path_to_file)

This solution has several advantages over the previously discussed schemes:

1. Users can not see any other pictures.
2. File heierarchies are permitted.
3. Low load on the server - no lookup table to attempt to reverse hashes.
4. CVC-MAC is much shorter than $E_k(/path/to/file)$, as we do not have to recover the path - cleaner links presented to the user.