1 Implementing ACLs

For this problem, assume each object has a single ACL, and ACL entries are of the form (Subject, Permission, GRANT or DENY). Subject may be a user or, if the system supports groups, a group. Permission specifies an action (e.g. read or write). Each entry specifies whether the permission is GRANTED or DENIED.

Recall that in some access control list systems, access is granted if any entry in the ACL GRANTS permission, some deny access if any entry in the ACL DENYs permission, and some GRANT or DENY based on the first rule that applies to the current request.

This question asks you to contemplate different data structures for implementing ACLs in these different settings. Recall that we need to perform two basic operations on ACLs:

- Given a Subject and a Permission, determine whether the ACL GRANTS or DENIES that Permission to that Subject.
- Add and delete entries. If ACLs are order-dependent, then a position must be specified when adding the new entry.

If a system only has users (but not groups), what kind of data structure (e.g. tree, hash table, linked list, something else) would you use to represent order-independent ACLs? Order dependent?

If the system supports users and groups, what kind of data structure would you use to represent order-independent ACLs? Order dependent?

Do you conclude that order-dependent or order-independent ACLs are more efficient, or are they the same?
1.1 Solution

For order-independent or order-dependent user-only ACLs, a hash table should work.

For order-independent ACLs with users and groups, a hash table can also work.

For order-dependent ACLs with groups, we assume each process has a user-id, and groups are simply sets of users. Groups can be changed at any time, so we can’t pre-compute anything based on the configuration of the groups at the time the ACL is created, unless we want to potentially rebuild the ACLs on the system whenever a group is modified.

Since groups can change at any time, we have to check every group in the ACL in order to see if it currently contains the process’ user-ID. But we can still use a hash table for all the user-entries in the ACL. To keep the order correct, each entry in the hash table can be of the form \((u, r, a, g)\), where \(u\) is the user, \(r\) is the access right, \(a\) is ALLOW or DENY, and \(g\) is the first group that follows this user entry in the ACL, or \(\bot\) if there is no subsequent group entry. ACL evaluation would then go as follows:

\[
\text{lookup}(\text{user}, \text{action}, \text{acl})
\]

\[
\text{terminal\_group} = \text{nil}
\]

\[
\text{answer} = \text{deny}
\]

\[
\text{if acl.user\_rules(user, action) exists}
\]

\[
(\text{answer}, \text{terminal\_group}) = \text{acl.user\_rules(user, action)}
\]

\[
\text{for (g, r, a) in acl.groups}
\]

\[
\text{if user is in g and r == action}
\]

\[
\text{return a}
\]

\[
\text{if g == terminal\_group}
\]

\[
\text{break}
\]

\[
\text{return answer}
\]

2 Hardware Capabilities

Imagine a CPU that uses capabilities to govern access to memory. A capability \((p, l)\) grant access to the memory starting at \(p\) and extending for \(l\) bytes.

The hardware designers attempt to build this system by taking a standard load/store CPU and making the following changes:

- They add new capability registers \(\%c_1, \ldots, \%c_8\). Each capability register holds a capability \((p, l)\).
- They modify \text{load} so that you must use a capability register to specify the memory.
address, i.e.

\[ \text{load} \ %\text{dest}, \ %c_i, \ %r_j \]

This loads the word at the offset specified in general-purpose register \( %r_j \) in the region specified by capability \( %c_i \) into the register specified by \( %\text{dest} \). Note that \( %\text{dest} \) may be a general-purpose or capability register. If \( %r_j \) exceeds the length of the region specified in \( %c_i \), then the instruction generates a segfault.

- \textbf{store} is modified analogously.

Analyzing and completing the basic system:

- Give the interface for \textbf{store}, similar to how \textbf{load} is specified above.
- The system, as currently described, is flawed. Show how to obtain a capability to any address on the system. (Hint: forge a capability).
- Describe how to fix the issue you exploited above. You may modify the CPU, RAM, etc. Simpler modifications are better.

Now consider trying to support object-oriented programming on such a system. We want objects to have public methods, and we want to ensure that other code can only jump to the first instruction of a public method. Furthermore, we want to enable objects to have private data, which can only be accessed while executing code from that object (or code that it calls and explicitly passes a capability).

- How can you enforce public interfaces in hardware? (Hint: make the program counter a capability register, extend capabilities to specify read, write, and/or execute permissions)
- How can you set it up so that class methods can access private data from objects of that class, but no other part of the program can? (Hint: treat any capability in the program counter as implicitly having “read” permission)

Finally, let’s build some run-time tools

- Describe an instruction for creating a sub-capability from a capability.
- Show how to use this instruction to implement malloc.
- What about free?
- What about garbage collection?
2.1 Solution

- “store \%c_i, \%r_j, \%val” stores the contents of register \%val to the memory pointed to by \%c_i, \%r_j, assuming \%r_j is in bounds. The val register can be a general-purpose register or a capability register.

- We can load anything we want into a capability register by storing arbitrary data (from a general-purpose register) to memory and then loading it into a capability register.

- Tag each word of memory to indicate whether it is a capability or not. Attempts to load non-capability words into capability registers will generate an exception.

- Extend capabilities to have read, write, execute bits. Make the PC a capability (and an offset). The call instruction takes a capability register and jumps to offset zero from that capability, assuming the capability has the execute bit set. Thus the vtable for an object would have to have a capability for each public function of the object.

- Ugh, this gets complicated. Maybe one of you guys came up with a better solution. But here’s something that seems to work.

  Allow loads relative to the PC capability. Treat the PC capability as if it always has the read bit set.

Each class has a region of memory that holds its methods. The first function in the region is a dispatch function that examines its first argument to determine which interface function to call.

Each instance of the class has two memory regions. One region holds all the state data of the instance. The other holds the call gate that is used to invoke methods on the object. The call gate has two capabilities, one is a read-write capability for the private data region, and the other is an execute capability for the class methods. When a user invokes the call gate, it uses pc-relative loads to load those capabilities into c7 and c8, and then jumps to the class dispatch method. The dispatch method uses the value in general register 0 to invoke the real interface function, which can access the instance’s private data through c8.

Arguments to calls are passed in registers. Registers may contain pointers to memory regions with larger data items being passed between arguments. Each object should have its own stack frame, which it can allocate upon entry. One of the arguments to a function call is a capability for a return-gate. The return gate reloads all the state of the caller and resumes its execution.

- “attenuate \%c_i, \%c_j, \%r_k, \%r_l, \%r_n” copies c_j to c_i, but advances its start by r_k and reduces its length by r_l and masks its permission bits by r_n.
• malloc starts with a capability for the entire “malloc arena”. Whenever a user allocates memory, it uses attenuate to create a capability for the region being allocated. (It stores its metadata elsewhere. How it does so is up to the malloc implementation.)

• free is a problem, since a user could free some memory but hold onto a pointer to it. Then if it gets reallocated, the original user could corrupt the new user’s data. I can’t see an easy way to fix this. One idea would be to create a new doubly-indirected type of capability, i.e. a capability $a$ that points to a block of memory with a single capability $b$ in it. All the holder of $a$ can do is use it to dereference $b$. $b$ can be any kind of capability, but its permissions get masked by the permissions on $a$. When malloc hands out memory, it creates a $b$ and returns $a$. When a user calls free, malloc zeros out the $b$.

Not a great solution.

• Add a “supervisor” mode to the CPU. Every so often, an interrupt jumps to a specified “interrupt handler” in supervisor mode. In supervisor mode, all capabilities are readable. The supervisor runs a mark-and-sweep garbage collector and returns any un-reachable memory to malloc.