# Part IV
# Access Control

Confidentiality and integrity are often enforced using a form of authorization known as *access control*, which involves the following assumptions.

- Predefined *operations* are the sole means by which principals can learn or update information.
- A reference monitor is consulted whenever one of these predefined operations is invoked; the operation is allowed to proceed only if the invoker holds the required *privileges*.

Confidentiality then is achieved by restricting whether a given principal is authorized to execute operations that reveal information; integrity is achieved by restricting execution of operations that perform updates.

An *access control policy* specifies which of the operations associated with any given *object* (including operations to change the policy) each given principal[1] is authorized to have performed. Principals are entities to which execution can be attributed—users, processes, threads, or even procedure activations. Objects are entities on which operations are defined—storage abstractions, such as memory or files (with read, write, and execute operations), and code abstractions, such as modules or services (with operations to initiate or suspend execution). Distinct privileges are typically associated with distinct operations, and sometimes there are distinct privileges for each different operation on each different object. So, for example, there might be a specific privilege $\texttt{read}_{\texttt{Obj}}$ that a principal must hold to perform operation `read` on object `Obj`.

The Principle of Least Privilege is best served by having fine-grained principals, objects, and operations; the Principle of Failsafe Defaults favors defining an access control policy by enumerating privileges rather than prohibitions. That, however, is only a small part of the picture, and there is much ground to cover in our discussions of access control policy.

[1] The term *subject* has also been used to designate entities that make requests to execute operations. Access control policies then specify which operations a subject is authorized to perform on each object.

# Chapter 7

# Discretionary Access Control

## 7.1 The DAC Model

In a *discretionary access control* (DAC) policy, the initial assignment and subsequent propagation of all privileges associated with an object are controlled by the *owner* of that object and/or other principals whose authority can be traced back to the owner. DAC policies are what commercial operating systems typically enforce. Here, the principals are users; the objects include files, I/O devices, and other passive system abstractions.

The assignment of privileges by a DAC policy can be depicted using a table that has a row for each principal and a column for each object. The table entry for a principal $P$ and an object $O$ lists privileges corresponding to those operations on $O$ that are authorized when invoked by execution being attributed to $P$. Figure 7.1, for example, gives a table that assigns privileges for users `fbs`, `mmb` and `jhk` to perform operations on files `c1.tex`, `c2.tex`, and `invtry.xls`. Only execution attributed to `fbs` can read (`r`) or write (`w`) `c1.tex` and `c2.tex`; only execution attributed to `mmb` can write `invtry.xls`; execution attributed to any of the three users can read `invtry.xls`.

| | object | | |
|---|---|---|---|
| principal | `c1.tex` | `c2.tex` | `invtry.xls` |
| `fbs` | `r`, `w` | `r`, `w` | `r` |
| `mmb` | | | `r`, `w` |
| `jhk` | | | `r` |

Figure 7.1: Example DAC Policy

The table that Figure 7.1 depicts is called an *access matrix*.[1] However, the term "matrix" here is misleading, because the row and column order has no significance; in a matrix, it would. An access matrix really just specifies an unordered set *Auth* of triples, where $\langle P, O, op\rangle \in Auth$ holds if and only if principal $P$ holds privilege $op$ for object $O$. We call *Auth* an *authorization relation*.

Any DAC policy can be circumvented if principals are permitted to make arbitrary changes to *Auth*. Yet as execution of a system proceeds, changes to *Auth* will inevitably be needed. New objects must be accommodated, object reuse requires changing the set of principals that are authorized to access an object, and trust relationships between principals evolve in response to events both inside and outside of the computing system. To characterize permitted changes to *Auth*, a DAC policy includes *commands*. Each command specifies a parameter list, a Boolean *precondition*, and an *action*. If the command is invoked and the precondition holds, then the action is executed; if the precondition does not hold, then the command fails. Evaluation of the precondition and execution of the action is assumed to be indivisible.[2] The precondition and action are permitted to name constants, *Auth*, and the formal parameters.

As an example, here is a command that might be found on a system where principals are system users.

*addPriv*$(U, U', O, op)$: **command**
  **pre**: $invoker(U) \ \wedge \ \langle U, O, \texttt{owner}\rangle \in Auth \ \wedge \ op \neq \texttt{owner}$
  **action**: $Auth := Auth \cup \{\langle U', O, op\rangle\}$

The precondition that guards *addPriv* is

$$invoker(U) \ \wedge \ \langle U, O, \texttt{owner}\rangle \in Auth \ \wedge \ op \neq \texttt{owner}$$

where predicate $invoker(U)$ is satisfied if and only if *addPriv* is invoked by execution attributed to user $U$. So, the precondition implies that the invoker of *addPriv* has the `owner` privilege for an object $O$ and that $op$ is not the `owner` privilege. The action specified by *addPriv* is an assignment statement

$$Auth := Auth \cup \{\langle U', O, op\rangle\}$$

that adds to *Auth* a triple authorizing operation $op$ on object $O$ by user $U'$. Thus, *addPriv* is consistent with the defining characteristic for a DAC policy—the owner of object $O$ is the principal that grants privileges for operations on $O$.

Separation of Privilege suggests that it is better to have a distinct privilege[3] say $op^*$ for granting a privilege $op$ than to have a single generic privilege, like `owner`, that allows any privilege to be granted. So a better command than *addPriv* would be:

[1] Some prefer the term a *protection matrix*.

[2] In practice, checking a precondition and executing the code for the action is likely to involve multiple atomic actions. It suffices that the effect nevertheless appears indivisible with respect to execution of other commands.

[3] Privilege $op*$ is sometimes called a *copy flag* for $op$.

*grantPriv*$(U, U', O, op)$: **command**
**pre**: *invoker*$(U) \wedge \langle U, O, op^* \rangle \in Auth$
**action**: $Auth := Auth \cup \{\langle U', O, op \rangle\}$

### 7.1.1 Finer-Grained Principals: Protection Domains

The Principle of Least Privilege suggests that the set of operations a principal should be authorized to execute depends on the task to be performed. Users are thus too coarse-grained to serve as the basis for aggregating privileges in DAC policies. This leads to employing *protection domains* as the set of principals. Each thread of control is associated with a protection domain, each protection domain is associated with a different set of privileges, and we allow transitions from one protection domain to another as execution of the thread proceeds. Different sets of privileges can now be associated with a thread as it progresses from one task to the next, and the Principle of Least Privilege is now easily instantiated.

For efficient implementation, protection-domain transitions must be associated with events that a run-time environment can detect cheaply. For instance, protection-domain transitions that coincide with certain kinds of control transfer (e.g., invoking a program) are typically inexpensive for a run-time environment to support, as are those that coincide with certain state changes (e.g., changing from user mode to supervisor mode). But few processors could efficiently support protection-domain transitions being triggered by branches to some specific instruction or by stores to an arbitrary memory location.

When an operating system supports protection domains, certain system calls cause protection-domain transitions. System calls for invoking a program or changing from user mode to supervisor mode are obvious candidates. Some operating systems provide an explicit domain-change system call rather than implicitly linking protection-domain transitions to other functionality; the application programmer or a compiler's code generator is then required to decide when to invoke this domain-change system call.

Since distinct tasks are typically implemented by distinct pieces of code, the Principle of Least Privilege could be well served if we associate different protection domains with different code segments. We might, for example, contemplate having a protection domain $U \triangleright pgm$ for each code segment *pgm* executing on behalf of a user $U$. Here, *pgm* could be an entire program, a method, a procedure, or a block of statements; it might be executed by a process started by user $U$ or by a process started by some other user in response to a request from $U$. Ideally, protection domain $U \triangleright pgm$ would hold only the minimum privileges needed for *pgm* to execute for $U$.

Figure 7.2 reformulates the access matrix of Figure 7.1 in terms of principals that are protection domains associated with code segments corresponding to entire programs: a shell (`sh`), a text editor (`edit`), and a spreadsheet application (`excel`). Notice, `c1.tex` and `c2.tex` now can be written by user `fbs` only while executing `edit`, and `invtry.xls` can be accessed only by executing `excel` (with `mmb` still the only user who can perform write operations to that object).

| | object | | |
|---|---|---|---|
| domain | c1.tex | c2.tex | invtry.xls |
| fbs▷sh | | | |
| fbs▷edit | r, w | r, w | |
| fbs▷excel | | | r |
| mmb▷sh | | | |
| mmb▷edit | | | |
| mmb▷excel | | | r, w |
| jhk▷sh | | | |
| jhk▷edit | | | |
| jhk▷excel | | | r |

Figure 7.2: Example DAC Policy for Domains

A given protection domain might or might not be appropriate for execution that performs a given task and, therefore, according to the Principle of Least Privilege, transitions ought to be authorized only between certain pairs of protection domains. For example, we would expect that execution in a shell should be allowed to start either a text editor or a spreadsheet application, but execution in a spreadsheet application should not be allowed to start a shell. We can specify such restrictions by defining an `enter` privilege for each protection domain and by including protection domains in the set of objects that can be named by *Auth*. A protection domain $D$ must possess the `enter` privilege for a protection domain $D'$—that is, $\langle D, D', \texttt{enter} \rangle \in Auth$ must hold—for execution in $D'$ to be started by execution in $D$.[4] Figure 7.3 incorporates such constraints, using `e` to denote the `enter` privilege.

### 7.1.2 Amplification and Attenuation

The sets of privileges before and after a protection-domain transition are likely to be related.

> **Attenuation of Privilege.** Suppose execution in a protection domain $D$ initiates a subtask, which is executed in protection domain $D'$. Then $D'$, having a more circumscribed scope, should not have all of the privileges $D$ has. We use the term *attenuation of privilege* for a transition into a protection domain that eliminates privileges. □

> **Amplification of Privilege.** Suppose execution in a protection domain $D'$ implements an operation on some object $O$, as a service to execution in a protection domain $D$. Then $D'$ should grant privileges for $O$ that $D$

[4] An alternative to having protection domains be objects is having code segments (independent of user) be objects. With this alternative, we specify a protection domain from which a user $U$ is allowed to next start execution of code segment *pgm* by granting that protection domain an `execute` privilege for object *pgm*.

| | object | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| domain | c1.tex | c2.tex | invtry.xls | fbs▷sh | fbs▷edit | fbs▷excel | mmb▷sh | mmb▷edit | mmb▷excel | jhk▷sh | jhk▷edit | jhk▷excel |
| fbs▷sh | | | | | e | e | | | | | | |
| fbs▷edit | r, w | r, w | | | | | | | | | | |
| fbs▷excel | | | r | | | | | | | | | |
| mmb▷sh | | | | | | | | e | e | | | |
| mmb▷edit | | | | | | | | | | | | |
| mmb▷excel | | | r, w | | | | | | | | | |
| jhk▷sh | | | | | | | | | | | e | e |
| jhk▷edit | | | | | | | | | | | | |
| jhk▷excel | | | r | | | | | | | | | |

Figure 7.3: Example DAC Policy with Domain Entry

does not. We use the term *amplification of privilege* for a transition into a protection domain that adds privileges. □

Notice that attenuation of privilege and amplification of privilege both have a role to play in supporting the Principle of Least Privilege. Attenuation of privilege, for instance, underlies *restricted delegation*. Here, one principal $P$ requests that a task be performed by another principal $P'$, granting $P'$ only the subset of privileges $P'$ requires for that task. Amplification of privilege is crucial for supporting *data abstraction*, where users of an object are deliberately kept ignorant of how that object is implemented.

**The Confused Deputy.** Protection-domain transitions bring the risk of a *confused deputy* attack. Here, a client issues requests that cause some server to abuse privileges it holds but the client does not hold.

To make this concrete, consider a server to handle requests that name a client's file. The server computes a function $F$ on that file, writes that value to the file, and records billing information in the server's file `charges.txt`.

*Server*: **operation**( *f* : **file** )
  $S_1$: *buffer* := *FileSys.Read*( *f* )
  $S_2$: *results* := *F*( *buffer* )
  $S_3$: *chrgs* := *calcBill*( *results* )
  $S_4$: *FileSys.Write*( *f*, *results* )
  $S_5$: *FileSys.Write*( `charges.txt`, *chrgs* )
  **end** *Server*

Further, suppose the server but no client holds a `write` privilege for `charges.txt`,

and that when processing a each request from a client $C$, a domain change occurs so the server is granted `read` and `write` privileges $C$ holds for file named in $f$.

We might expect that since client $C$ lacks `write` privileges for the server's file `charges.txt`, then $C$ cannot cause the contents of `charges.txt` to become corrupted. But that expectation is naive. By naming `charges.txt` as the argument in its request, $C$ would cause the server to execute with $f$ = `charges.txt`. Execution of $S_4$ would then corrupt `charges.txt`, since the server does hold a `write` privilege for that file. What happened? The server functioned as a deputy to the client, and the deputy became "confused" by the client's request. Specifically, the deputy was fooled into writing *results* to a file (`charges.txt`) even though the client did not hold write privileges for that file.

One obvious defense is for the server to check that the client holds appropriate privileges for the file named in $f$. This defense, however, would require programmers to include checks in every program that might invoke operations on objects whose names were passed in arguments. Many programmers would regard adding those checks as onerous and not bother.

A more elegant defense is to combine naming and authorization. Instead of names for objects (like files), we might require programs to use unforgeable *bundles* comprising the name for an object along with privileges for that object. Bundles would provide the sole means by which programs name, hence access, objects. In the example above, the client request would convey a bundle for a file (incorporating a `read` and `write` privileges). The server would use this client-supplied bundle for reading and updating that file. The server would also have a bundle for `charges.txt` (incorporating a `write` privilege). Since the client does not have a `write` privilege for `charges.txt`, a client-supplied bundle for `charges.txt` could not incorporate a `write` privilege. So a request from the client, where a `charges.txt` bundle was provided as the file name for $f$, would cause write $S_4$ to fail. The confused deputy is no longer duped into writing the wrong content into `charges.txt`.

A final approach to preventing confused deputy attacks could be to associate different sets of privileges with the different statements in a piece of code. For example, instead of granting a `write` privilege for `charges.txt` to all statements in *Server*, we might instead only grant it to $S_5$ (and not to $S_4$). Now, execution of $S_4$ with $f$ = `charges.txt` will fail, so the confused deputy attack no longer works. The primary problem with this defense is the complexity that comes from having to define and manage the larger numbers of such fine-grained protection domains.

### 7.1.3 *Undecidability of Privilege Propagation

A central concern when designing the commands for changing authorization relation *Auth* is having assurance that certain privileges cannot be granted to particular principals. We can formalize this concern as a predicate.

**Privilege Propagation.** $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ is *true* if

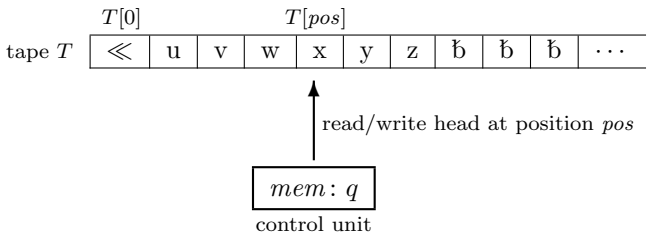



Figure 7.4: A Turing Machine

and only if principal $P$ eventually can be granted privilege *op* to object $O$ by starting from authorization relation *Auth* and allowing principals not in *Prins* to execute commands from set $\mathcal{C}$. □

Typically, *Prins* would be the set of principals that are both trusted and authorized to grant $\langle P, O, op\rangle$ and, therefore, if $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ holds, it then indicates unauthorized propagation of privilege.

To determine whether $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ holds, we might write a program that computes the value of $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ by generating all authorization relations that principals not in *Prins* executing sequences of commands from $\mathcal{C}$ could derive from *Auth*. However, if commands create new principals or new objects then there might be an infinite number of infinite-length command sequences to try. Termination is not guaranteed for a program undertaking such an enumeration. So that approach is not guaranteed to work.

What other approaches might we use? None! We prove below that any program that computes the value of $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ could be used to solve the halting problem for Turing machines. The latter is well known to be an *undecidable problem*—a decision problem for which no algorithm can exist that terminates with a correct answer for every input. By reducing Privilege Propagation to an undecidable problem, we establish there cannot exist a program that always correctly evaluates $CanGrant(Prins, \mathcal{C}, Auth, \langle P, O, op\rangle)$ for any possible arguments.

**Review of Turing Machines and Undecidability.** A *Turing machine* is an abstract computing device. It has an infinite tape, a read/write head, and a control unit with finite memory. Figure 7.4 depicts these components.

The *tape* comprises an infinite sequence $T[0]$, $T[1]$, ... of *tape squares*. Each tape square is capable of storing some symbol from a finite set $\Gamma \cup \{\ll, ƀ\}$. Symbol $\ll$ is stored in $T[0]$ to indicate that this is the first tape square; ƀ is stored in any tape square that has never been written.

The *read/write head* is always positioned at tape square $T[pos]$, which we

refer to as the *current* tape square. The *control unit* uses the read/write head to sense and/or change the symbol stored in the current tape square. In addition, the control unit can reposition the read/write head, moving it one tape square left or right.

The control unit has a memory *mem* capable of storing one symbol from finite set $Q$ of *control states*. We avoid confusion between tape and memory symbols by assuming that $Q$ and $\Gamma \cup \{\ll, \text{ƀ}\}$ are disjoint. The control unit performs execution steps, as specified by a transition function $\delta$. By defining

$$\delta(q, \gamma) = \langle q', \gamma', v \rangle$$

where $v \in \{-1, 1\}$, we specify that the following *execution step* occurs when $mem = q$ and $T[pos] = \gamma$:

$$T[pos] := \gamma';\ mem := q';\ pos := pos + v$$

That is, $\gamma'$ becomes the symbol stored by current tape square $T[pos]$, $q'$ becomes the new control state, and the read/write head position moves left ($v = -1$) or moves right ($v = 1$).

We impose two requirements on what execution steps are possible by the Turing machines that we will consider.

- An execution step $\delta(q, \gamma)$ is defined for each possible symbol $\gamma \in \Gamma \cup \{\ll, \text{ƀ}\}$ and control state $q \in Q - \{q_F\}$, where $q_F$ is called the *halt state* for the Turing machine.

- No execution step replaces the $\ll$ symbol stored at $T[0]$ nor moves the read/write head off the left end of the tape: for each $q \in Q - \{q_F\}$ there exists some $q' \in Q$ such that $\delta(q, \ll) = \langle q', \ll, 1 \rangle$.

A Turing machine is said to *halt* when no execution step is defined, so the first of these requirements implies that execution of a Turing machine halts if and only if $q_F$ is stored into the control state.

One way to represent transition function $\delta$ for each possible $q \in Q - \{q_F\}$ and $\gamma \in \Gamma \cup \{\ll, \text{ƀ}\}$ is with a *transition table* that has $|Q - \{q_F\}|$ rows and $|\Gamma \cup \{\ll, \text{ƀ}\}|$ columns; the cell in the row for $q$ and the column for $\gamma$ contains the value of $\delta(q, \gamma)$. This representation has finite size because $Q$ and $\Gamma$ are, by definition, finite sets. So, the transition table could be stored in a finite number of tape squares on a Turing machine's tape.

A Turing machine *configuration* is characterized by a triple $\langle T, pos, mem \rangle$, where $T$ is a tape, $0 \leq pos$, and $mem \in Q$. The configuration is considered *initial* if $pos = 0$ and $mem = q_0$ hold; it is considered *terminal* if $mem = q_F$ holds. Execution steps of a Turing machine $M$ induce a relation $\longrightarrow$ on configurations;

$$\langle T, pos, mem \rangle \longrightarrow \langle T', pos', mem' \rangle$$

holds if and only if an execution step starting from configuration $\langle T, pos, mem \rangle$ produces configuration $\langle T', pos', mem' \rangle$.[5] An execution in which $M$ halts is

[5] Formally, $\langle T, pos, mem \rangle \longrightarrow \langle T', pos', mem' \rangle$ holds if and only if $\delta(mem, T[pos]) = \langle mem', \gamma', v \rangle$, $pos' = pos + v$, $T'[pos] = \gamma'$, and $T'[i] = T[i]$ for all $i$ where $i \neq pos$ holds.

described by a finite sequence of configurations

$$\langle T_0, pos_0, mem_0\rangle \longrightarrow \langle T_1, pos_1, mem_1\rangle \longrightarrow \cdots \longrightarrow \langle T_i, pos_n, mem_n\rangle$$

where configuration $\langle T_0, pos_0, mem_0\rangle$ is initial and configuration $\langle T_i, pos_n, mem_n\rangle$ is terminal. An execution that does not halt is described by an infinite sequence of configurations

$$\langle T_0, pos_0, mem_0\rangle \longrightarrow \langle T_1, pos_1, mem_1\rangle \longrightarrow \cdots \longrightarrow \langle T_i, pos_i, mem_i\rangle \longrightarrow \cdots$$

where configuration $\langle T_0, pos_0, mem_0\rangle$ is initial and no subsequent configuration is terminal.

Besides performing execution steps, Turing machines read input and produce output. A finite-length input *inp* is conveyed by storing *inp* on the Turing machine's tape prior to execution; the output of the execution is the tape's contents when (and if) execution of the Turing machine halts. We write $M(inp) = out$ to denote that execution of Turing machine $M$ on input *inp* halts and produces output *out*, and we write $M(inp) = \perp$ if $M$ does not halt on input *inp*.

*Halting Problem Undecidability.* Because transition functions have finite-length representations, the description for a Turing machine can itself be the input to a Turing machine. So it is sensible to speak about a Turing machine $M$ that produces as its output the result of analyzing some Turing machine $M'$ whose description is provided to $M$ as an input. The *halting problem* is concerned with constructing a Turing machine $M_{HP}$ that satisfies the following specification.

$$M_{HP}(M, inp)\colon \begin{cases} 0 & \text{if } M(inp) = \perp \\ 1 & \text{if } M(inp) \neq \perp \end{cases} \tag{7.1}$$

Thus, $M_{HP}(M, inp) = 1$ if and only if Turing machine $M$ halts on input *inp*.

A Turing machine $M_{HP}$ that satisfies specification (7.1) cannot exist. We prove this by showing that the existence of $M_{HP}$ would lead to a contradiction. Assume $M_{HP}$ exists. We use $M_{HP}$ to construct a Turing machine $M$ that satisfies the following specification:

$$M(inp)\colon \begin{cases} 0 & \text{if } M_{HP}(M, inp) = 0 \\ \perp & \text{if } M_{HP}(M, inp) = 1 \end{cases} \tag{7.2}$$

That is, $M$ invokes $M_{HP}$ and either (i) terminates with output a 0 or (ii) loops forever. So there are two cases to consider.

- *Case 1:* $M(inp) = 0$. From (7.2), we conclude $M_{HP}(M, inp) = 0$ holds. According to specification (7.1) for $M_{HP}$, this means that $M(inp)$ does not halt. But this leads to a contradiction, because we assumed for this case that $M(inp) = 0$, which implies execution of $M(inp)$ did halt.
- *Case 2:* $M(inp) = \perp$. From (7.2), we conclude $M_{HP}(M, inp) = 1$ holds. According to specification (7.1) for $M_{HP}$, this means that $M(inp)$ halts. But this leads to a contradiction, because we assumed for this case that $M(inp)$ does not halt.

| | object | | | | | | |
|---|---|---|---|---|---|---|---|
| principal | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
| $P_0$ | $\ll$ | nxt | | | | | |
| $P_1$ | | u | nxt | | | | |
| $P_2$ | | | v | nxt | | | |
| $P_3$ | | | | w, q | nxt | | |
| $P_4$ | | | | | x | nxt | |
| $P_5$ | | | | | | y | nxt |
| $P_6$ | | | | | | | z, end |

Figure 7.5: Representation of a Turing Machine Configuration by *Auth*

Since both cases lead to contradictions, we conclude that the existence of $M_{HP}$ leads to a contradiction—no Turing machine can solve the halting problem.

**Privilege Propagation and Undecidability.** We now prove that determining whether $\mathit{CanGrant}(\mathit{Prins}, \mathcal{C}, \mathit{Auth}, \langle P, O, op\rangle)$ holds is an undecidable problem. We do this by showing that a program *CG* could be used to solve the halting problem if invoking $\mathit{CG}(\mathcal{C})$ for a set $\mathcal{C}$ of commands determines whether there exists a finite sequence of commands from $\mathcal{C}$ that eventually causes $\langle P, O, op\rangle \in \mathit{Auth}$ to hold.

The heart of the proof is a construction for simulating any given Turing machine $M$. We represent $M$'s configuration by using an authorization relation *Auth* where the set of objects equals the set of principals. And we simulate $M$'s execution steps by using a set $\mathcal{C}$ of commands, where every execution of $M$ is simulated by a sequence of commands. Moreover, the commands in $\mathcal{C}$ are defined in such a way that $\langle P_0, P_0, q_F\rangle \in \mathit{Auth}$ holds if and only if $M$ halts. Consequently, a program *CG* that determines whether $\mathit{CanGrant}(\emptyset, \mathcal{C}, \mathit{Auth}, \langle P_0, P_0, q_F\rangle)$ holds would constitute a solution to the halting problem.

Figure 7.5 depicts an authorization relation *Auth* that represents a Turing machine configuration $\langle T, 3, \mathrm{q}\rangle$ for a tape $T$ storing: $\ll$ u v w x y z ƀ ƀ ƀ ... . The representation is based on the following.

- $\langle P_i, P_i, x_i\rangle \in \mathit{Auth}$ signifies that symbol $x_i \in \Gamma \cup \{\ll, \text{ƀ}\}$ is stored by tape square $T[i]$.
- $\langle P_i, P_i, q\rangle \in \mathit{Auth}$ signifies that $\mathit{mem} = q$ and $\mathit{pos} = i$ hold, where $q \in Q$ and $0 \leq i$.
- The linear order of tape squares is encoded by privileges: nxt, end, $\ll$:
  - $\langle P, P, \ll\rangle \in \mathit{Auth}$ implies that principal $P$ is used to represent $T[0]$.
  - $\langle P, P, \mathrm{end}\rangle \in \mathit{Auth}$ implies that principal $P$ is used to represent the last non-blank tape square of $T$.

$C_{\mathrm{R}}(p, p', q, q', \gamma, \gamma')$: **command**
**pre**: $\delta(q, \gamma) = \langle q', \gamma', 1\rangle$
$\quad \wedge \langle p, p, q\rangle \in Auth \ \wedge \ \langle p, p, \gamma\rangle \in Auth \ \wedge \ \langle p, p', \mathrm{nxt}\rangle \in Auth$
**action**: $Auth := Auth - \{\langle p, p, q\rangle, \langle p, p, \gamma\rangle\}$
$\quad Auth := Auth \cup \{\langle p, p, \gamma'\rangle, \langle p', p', q'\rangle\}$

$C_{\text{R-end}}(p, q, q', \gamma, \gamma')$: **command**
**pre**: $\delta(q, \gamma) = \langle q', \gamma', 1\rangle$
$\quad \wedge \langle p, p, q\rangle \in Auth \ \wedge \ \langle p, p, \gamma\rangle \in Auth \ \wedge \ \langle p, p, \mathrm{end}\rangle \in Auth$
**action**: $Auth := Auth - \{\langle p, p, q\rangle, \langle p, p, \gamma\rangle, \langle p, p, \mathrm{end}\rangle\}$
$\quad p' := \mathrm{newPrincipal}()$
$\quad Auth := Auth \cup \{\langle p, p, \gamma'\rangle, \langle p', p', q'\rangle, \langle p', p', \mathrm{end}\rangle, \langle p, p', \mathrm{nxt}\rangle\}$

$C_{\mathrm{L}}(p, p', q, q', \gamma, \gamma')$: **command**
**pre**: $\delta(q, \gamma) = \langle q', \gamma', -1\rangle$
$\quad \wedge \langle p', p', q\rangle \in Auth \ \wedge \ \langle p', p', \gamma\rangle \in Auth \ \wedge \ \langle p, p', \mathrm{nxt}\rangle \in Auth$
**action**: $Auth := Auth - \{\langle p', p', q\rangle, \langle p', p', \gamma\rangle\}$
$\quad Auth := Auth \cup \{\langle p', p', \gamma'\rangle, \langle p, p, q'\rangle\}$

$C_{\mathrm{HALT}}(p)$: **command**
**pre**: $\langle p, p, q_F\rangle \in Auth$
**action**: $Auth := Auth - \{\langle p, p, q_F\rangle\}$
$\quad Auth := Auth \cup \{\langle P_0, P_0, q_F\rangle\}$

Figure 7.6: Commands $\mathcal{C}_{TM}$ to Simulate a Turing Machine

– $\langle P, P', \mathrm{nxt}\rangle \in Auth$ implies that the tape square represented using principal $P'$ immediately follows the tape square represented using principal $P$.

The need for $\langle P, P', \mathrm{nxt}\rangle$ privileges at first might seem puzzling. It arises because tape squares for a Turing machine are linearly ordered, we are associating each tape square with a distinct principal, but we have not assumed any ordering on principals and $Auth$ does not induce one.[6] To create the ordering on tape squares, we define an ordering relation $\prec$ on principals. Let $P_{pos}$ be the principal that corresponds to tape square $T[pos]$. We desire that $P_i \prec P_{i+1}$ hold for $0 \leq i$, and we encode this relation by having $\langle P, P', \mathrm{nxt}\rangle \in Auth$ hold if and only if $P \prec P'$ does.

The set $\mathcal{C}_{TM}$ of commands to simulate execution steps for a Turing machine are based on that Turing machine's transition function $\delta$. The precondition of each command describes a Turing machine configuration; the action updates $Auth$ in accordance with the changes to the configuration prescribed by $\delta$. Fig-

[6] You might hope that the integer subscripts used to construct names for principals would suffice to define an ordering on the principals. But we assumed only that different principals have different names; integer subscripts are merely a convenient notation for that.

ure 7.6 gives the commands. The commands are constructed in a way that, at any time, there is at most one command for which argument values exist to satisfy its precondition. Invoking that command simulates the next execution step of the Turing machine being simulated.

For example, the Turing machine execution step specified by $\delta(q, \gamma) = \langle q'\gamma', 1\rangle$ is simulated by commands $C_{\text{R}}$ and $C_{\text{R-end}}$ in Figure 7.6.

- $C_{\text{R}}$ handles the case where the read/write head position is not at the right-most tape square that is not ƀ; this case is distinguished because the current tape square corresponds to some principal $P$ for which there does exist another principal $P'$ where $\langle P, P'\text{nxt}\rangle \in Auth$ holds.

- $C_{\text{R-end}}$ handles the case where the read/write head position is at the right-most tape square that is not ƀ; this case is distinguished by having the tape square correspond to a principal $P$ where $\langle P, P, \text{end}\rangle \in Auth$ holds.

And an execution step specified by $\delta(q, \gamma) = \langle q', \gamma', -1\rangle$ is simulated by $C_{\text{L}}$ in Figure 7.6.

$C_{\text{HALT}}$ of Figure 7.6 is included in $\mathcal{C}_{TM}$ so that $\langle P_0, P_0, q_F\rangle$ is granted if ever privilege $q_F$ is granted to any principal. This command does not simulate a Turing machine's execution step, but now $CanGrant(\emptyset, \mathcal{C}_{TM}, Auth, \langle P_0, P_0, q_F\rangle)$ holds if and only if the Turing machine being simulated halts. Thus, a program that evaluates $CanGrant(\emptyset, \mathcal{C}_{TM}, Auth, \langle P_0, P_0, q_F\rangle)$ would constitute a solution to the halting problem. The halting problem is undecidable, so we have proved what we set out to show—that no single program can exist to evaluate $CanGrant(\emptyset, \mathcal{C}, Auth, \langle P_0, P_0, q_F\rangle)$ for an arbitrary value of $\mathcal{C}$.

This undecidability result does not imply that a program cannot exist to compute the value of $CanGrant(\emptyset, \mathcal{C}, Auth,\ TupleP, O, op)$ for one or another specific command set $\mathcal{C}$. Such programs do exist, and they have even been implemented for command sets $\mathcal{C}$ that model DAC policies enforced by actual systems. The undecidability result nevertheless is important, because it gives insight into the difficulty of analyzing an important class of authorization mechanisms.

### 7.1.4 Implementation of DAC

At the heart of any implementation of DAC will be a scheme for representing authorization relation $Auth$. That scheme must support certain functionality:

- computing whether $\langle P, O, op\rangle \in Auth$ holds and, therefore, a principal $P$ is authorized to perform some operation $op$ on a given object $O$,

- changing $Auth$ in accordance with commands the DAC policy defines, and

- associating a protection domain with each thread of control and performing transitions between protection domains as execution proceeds.

In addition, support for two kinds of *review* are also often desired: (i) listing, for a given principal, the privileges it holds for each object, and (ii) listing, for a given object, the principals and the privileges each holds for that object.

The naive scheme for representing *Auth* is to employ a 2-dimensional array-like data structure resembling an access matrix. But such an array is likely to be sparse, because a typical principal will hold privileges for only a small fraction of all objects in a system. Implementors thus favor data structures that store only the non-empty cells of the access matrix. There are two general approaches. An *access control list* encodes the non-empty cells associated with a column (object); a list of *capabilities* encode the non-empty cells associated with a row (principal).

Access control lists and capabilities can, in theory, express the same policies. In practice, they differ in important ways. One such difference is the cost of performing revocation and review. With access control lists, revocation of access to an object $O$ by some principal is straightforward—that principal is simply deleted from a list associated with $O$. And it is relatively cheap to enumerate the principals and their privileges for a given object. But listing for all objects what privileges are granted to some specified principal requires scanning the access control lists for all of those objects, and that is likely to be expensive.

With capabilities, the cost of performing revocation or review depends on implementation details. When the capabilities held by a principal are easy to find—for example, they are stored in an easily located list associated with that principal—then determining what privileges a given principal holds is cheap. But if capabilities are instead scattered throughout the memory and files accessible to principals, then determining what privileges are held by a given principal could be infeasible. Revocation of access to an object $O$ by some principal $P$ requires finding all copies of certain capabilities directly or indirectly accessible to $P$. That cost is implementation dependent but rarely insignificant (as will become clear in §7.3).

Many find access control lists attractive because the implementation—a reference monitor and a list—is localized, which enables a separation of concerns. That analysis ignores the cost and complexity of managing the many small protection domains that must be defined explicitly for instantiating the Principle of Least Privilege when access control lists are in use. When authorization is enforced with capabilities, that complexity is eliminated because protection domains are not defined explicitly. But, when capabilities are being used, a potentially large set of components must now be analyzed to understand what accesses are, or could become, possible as execution proceeds. This same decentralization, however, is what makes capabilities so appealing for controlling access to user-defined objects. And, as the confused deputy attack in §7.1.2 showed, considerable benefits accrue from the combined naming and authorization that capabilities implement.

## 7.2 Access Control Lists

The *access control list* for an object $O$ is a list

$$\langle P_1, Privs_1\rangle\ \langle P_2, Privs_2\rangle\ \ldots\ \langle P_n, Privs_n\rangle \tag{7.3}$$

of *ACL-entries*. Each ACL-entry $\langle P_i, Privs_i\rangle$ is a pair, where $P_i$ names a principal, $Privs_i$ is a non-empty set of privileges, and $op \in Privs_i$ holds if and only if $\langle P_i, O, op\rangle \in Auth$ holds. For example, the access control list for `invtry.xls` in Figure 7.1 is

$$\langle \texttt{fbs}, \{\texttt{r}\}\rangle\ \langle \texttt{mmb}, \{\texttt{r}, \texttt{w}\}\rangle\ \langle \texttt{jhk}, \{\texttt{r}\}\rangle.$$

### 7.2.1 Access Control List Representations

Long access control lists are difficult for people to understand and, therefore, problematic to update; they are also expensive for enforcement mechanisms to scan when authorizing access requests. Therefore, representations have been proposed for shortening the number of ACL-entries in an access control list and/or for making important but complicated kinds of updates easier to perform.

**Groups of Principals.** Particularly in corporate and institutional settings, users might be granted privileges by virtue of membership in a group. Students who enroll in a class, for example, should be given access to that semester's class notes and assignments simply because they are members of the class.

The minimalist approach is to list group members individually on the various access control lists. However, if membership in a group confers privileges for many objects, then adding or deleting a member requires updating many access control lists. That can be error-prone. Moreover, updating an individual access control list can be subtle. Suppose, for example, user $U$ is in some group where membership grants privilege *op* for object $O$. If $U$ is being dropped from the group then you might be tempted to delete *op* from the ACL-entry naming $U$ in the access control list for $O$. But this ignores the possibility that $U$ might also be a member of some other group that also grants *op* for $O$ to its members, in which case $U$ should not lose the *op* privilege for $O$.

We avoid these difficulties by allowing names for groups of principals to appear in access control lists.

**Groups in Access Control Lists.**

- A *group declaration* associates a *group name* with a set of principals. The set is specified either by enumerating its elements or by giving a predicate that all principals in the set must satisfy.[7]

[7] An enumeration should be short enough so that the absence or presence of specific principals is unlikely to be overlooked; a predicate for characterizing a set should be transparent enough so that it defines all of the intended principals and no others.

– An ACL-entry $\langle G, Privs \rangle$, where $G$ is a group name and *Privs* is a set of privileges, grants all privileges in *Privs* to all principals $P$ that are members of $G$. □

Groups in ACL-entries introduce indirection that eliminates the need to update multiple access control lists when group membership changes—only the group declaration needs to be changed. Moreover, given ACL-entry $\langle G, Privs \rangle$ on the access control list for an object $O$, deleting $P$ from $G$ revokes $P$'s privileges to $O$ only if $P$ does not appear elsewhere on that access control list (directly or through membership in some other group).

**Permission and Prohibition.** That a given principal does not hold a specified privilege is sometimes what's important. Yet in order to conclude that $P$ does not hold *op* for an object $O$, we would have to enumerate and check all principals granted *op* by the access control list for $O$. This process—especially when groups are present—could be time-consuming. So some systems allow a *prohibition* $\overline{op}$ to appear in an ACL-entry. As the term suggests, ACL-entry $\langle P, \{\overline{op}\} \rangle$ specifies that execution of operation *op* by $P$ is prohibited.

The introduction of prohibitions does raise a question about the meaning of an access control list that contains both a privilege *op* and a conflicting prohibition $\overline{op}$ for the same principal. Different systems resolve this conflict in different ways, but it is not uncommon to base the resolution on the relative order of the conflicting ACL-entries. A system, for example, might give precedence to the first one in the list.

### 7.2.2 Pragmatics

Designers and implementors of access control mechanisms are primarily concerned with three things: flexibility, understandability, and run-time cost. Without sufficient flexibility, we might not be able to specify the security policy we desire. But support for flexibility usually brings complexity and cost. We should eschew complexity, because it introduces the risk that people will be unable or disinclined to write or understand policies; it also tends to undermine our assurance in an access control mechanism's implementation. And higher run-time costs are problematic if this causes access-control policies that involve less checking to be favored over those that enforce what really is needed.

**Principals.** For any given system, the choice of what can be principal is constrained in practice by the availability of efficient means for attributing (authenticating) accesses, since the name of a principal making a request is what's needed for checking an access control list.

Operating systems typically have mechanisms for authenticating users and processes. Some language run-time environments do even better and attribute execution of each statement to the current chain of nested procedure invocations.[8] Such a setting would, for instance, allow a distinct protection domain

[8] One such a scheme was used in Figure 7.3, another is discussed in §7.5.

to be associated with the execution by some program $pgm_3$ invoked by a call within $pgm_2$, itself invoked by a call from $pgm_1$, running on behalf of user $U$ versus the protection domain just before $pgm_3$ is invoked or just after it returns.

Independent of what constitutes a principal, care should be exercised in recycling principal names. Otherwise, some future incarnation of a principal name could inadvertently receive privileges held by a past incarnation. One solution is simply not to reuse principal names, but this (i) requires saving enough state to ensure no future principal name duplicates a past name and (ii) constrains the choice of principal names, potentially making policies harder to understand. The more widely-adopted solution is, as part of deleting a principal from the system, to delete that principal's name from all access control lists.

**Objects.** The set of objects also constrains what policies can be expressed using access control lists. Each object must be associated with some reference monitor. The reference monitor must intercept every access to the object, and that requirement restricts possible choices for objects. In addition, each access control list must be stored in a way that its integrity is protected. Two solutions here are common: (i) store the access control list with the object, so that updates to the access control list are checked by the reference monitor; (ii) store the access control list with the reference monitor that reads it, so that the mechanism protecting the integrity of the reference monitor also protects the integrity of the access control list.

Operating system abstractions are particularly well suited to serve as the objects when access control lists are in use. First, system calls are then the only way to access an object; a reference monitor is embedded in the operating system routine that handles a system call. Second, operating system abstractions typically either are large enough (e.g., files) to accommodate storing their own access control lists or are relatively few in number (e.g., locks or ports) so that the operating system's memory can be used to store the access control lists.

**ACL-entry Representations.** Many advocate terse representations for ACL-entries, starting from a debatable premise that checking shorter access control lists is faster.[9] One approach is to employ patterns and wildcard symbols for specifying names of principals or privileges, so that a single ACL-entry can replace many; another approach is to replace a set of ACL-entries that grants privileges with a set of ACL-entries imposing prohibitions on the compliment if the latter is shorter. Terse representations are often harder for humans to understand, though, so there can be a trade-off: human understandability versus computation time for enforcement. And cheap enforcement of a poorly understood policy is not a desirable outcome.

[9]The premise is debatable because a terse representation might require additional computation or lookups per ACL-entry. And the cost of the additional computation per ACL-entry might well exceed the savings of processing fewer ACL-entries.

## 7.3 Capabilities

Abstractly, a *capability* is a pair $\langle O, Privs \rangle$, where $O$ is an object and $Privs$ is a set of privileges. Any principal that *holds* capability $\langle O, Privs \rangle$ is granted privileges $Privs$ for operations on $O$. So the authorizations specified by $Auth$ are enforced provided two properties remain satisfied during execution.

- Invocation by principal $P$ of operation $op$ on an object $O$ requires that $P$ hold a capability $\langle O, Privs \rangle$ with $op \in Privs$.
- Capabilities cannot be counterfeited or corrupted.

The first property suggests that capabilities could provide the sole means by which principals identify and access objects, supplanting ordinary names and addresses. This is called *capability-based addressing*. It was suggested in §7.1.2 for solving the confused deputy problem.

The second property stipulates that we prevent unauthorized creation of new capabilities and prevent unauthorized changes to existing capabilities. A variety of schemes have been developed for implementing this *capability authenticity*; the classics are outlined below (in §7.3.1 though §7.3.4). In all of these schemes, changes to $Auth$ are supported by providing routines that enable an authorized principal $P$ to

- create a new object and, in so doing, receive a capability for that object,
- transfer to other principals one or more capabilities $P$ holds, with attenuation and/or amplification of privilege applied where specified, and
- revoke capabilities that derive from capabilities $P$ holds.

So privileges are controlled by the owner or by principals whose authority can be traced to the owner, because all capabilities for an object $O$ are derived from the one received initially by the principal that created (owns) $O$. This makes the policies DAC.

**Object Names.** Unless the object name $O$ in a capability $\langle O, Privs \rangle$ always identifies the same unique object—independent of what principal is exercising $\langle O, Privs \rangle$ or of when that capability is being exercised—then transferring capabilities or even storing them could have unintended consequences. For example, if an object name $O$ designates an object $Obj$ but is later recycled to designate object $Obj'$, then by holding $\langle O, Privs \rangle$ long enough, a principal eventually gets privileges to access $Obj'$ (even though access to $Obj$ was what had been authorized).[10] And if an object name $O$ designates object $Obj$ in one principal's address space but $Obj'$ in another's, then transferring $\langle O, Privs \rangle$ unwittingly gives the recipient privileges to $Obj'$ (even though a capability for $Obj$ was being transferred).

[10]This problem can be avoided if the run-time environment's object-deletion routine also eliminates or revokes all capabilities naming the object. But, as will become clear, some implementations of capabilities are not amenable to the bookkeeping necessary for this.

One approach to naming objects is to employ a single, global virtual address space. An object that occupies *len* bytes and starts at a virtual address $v$ is given name $\langle v, len \rangle$. Virtual address spaces found on today's processors are large enough (64 bits) for distinct objects each to be assigned distinct virtual addresses from now until (almost) eternity. Object names (virtual addresses) thus never need to be recycled. The object's length is being incorporated into its name to ensure that distinct but overlapping objects nevertheless have different names. For example, the byte and a doubleword that start at the same virtual address $v$ are different objects and they now receive different names: $\langle v, 1 \rangle$ versus $\langle v, 8 \rangle$.

Address-translation hardware, however, is not the only way to implement a mapping from names in capabilities to addresses where the corresponding objects are stored. It suffices for the run-time environment to maintain a *directory* that maps object names to memory addresses. Principals are expected to invoke a system routine (passing an object name and an operation name) to perform each operation on an object; this system routine uses the directory to determine where the object is stored and then performs the named operation on the object found at that location. Notice, this scheme allows objects to be relocated dynamically, because updating only a single directory entry—rather than the name field in every capability for that object—suffices. Also, either a real or virtual address can be stored in the directory entry for an object.

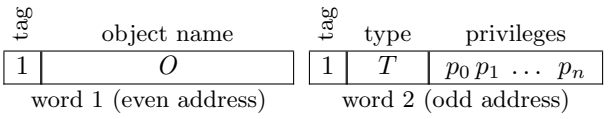
**Capability Archives.** Often, a main memory representation is used for capabilities held by executing principals, and an *archive* is kept in secondary storage to save capabilities for objects (e.g., files) that persist when principals holding those capabilities are not executing. Capabilities for accessing the archive are held by a special, trusted principal. This trusted principal executes as long as the system is running (it might be part of the operating system); it retrieves and loads appropriate capabilities into main memory whenever a principal requires them. The retrieval keys are often human-readable names, so the archive resembles a file system directory.

### 7.3.1 Capabilities in Tagged Memory

Hardware support for tagged memory is rarely found in commodity computers. Nevertheless, this approach to capability authenticity is worth understanding because it is both elegant and illuminating.

In a computer having tagged memory, each register and each word of memory is assumed to store a *tag* in addition to storing ordinary data. We assume 1-bit tags in the hypothetical scheme outlined here. Capabilities are stored in words having tags that equal 1; all other data is stored in words having tags that equal 0. For example, assume 64-bit memory words and that object names can be 63-bit virtual memory addresses. The hardware might then define a capability $\langle O, Privs \rangle$ to be any two consecutive words that start on an even address, where

both words have tags that equal 1:




Here, *Privs* comprises a *type* $T$ for object $O$ and a *privileges bit string* $p_0\, p_1 \ldots p_n$. The type defines how $p_0\, p_1 \ldots p_n$ is interpreted. For some types, each bit $p_i$ specifies whether the capability grants its holder a corresponding privilege $priv_i$ for $O$; for other types (e.g., memory segments), pre-defined substrings of $p_0\, p_1 \ldots p_n$ specify other properties of $O$ (e.g., the segment length) needed for enforcing an access control policy.

Tag bits alone are not sufficient to ensure that capabilities cannot be counterfeited or corrupted, though. The processor's instruction set must be defined with capability authenticity in mind. Typically, this entails enforcing restrictions on updates to words whose tags equal 1 and on changes to tags. For example, a user-mode instruction[11]

`cap_copy` @*src*, @*dest* (7.4)

for copying two consecutive words of memory from source address @*src* to destination address @*dest* would cause a trap unless (i) the source and destination both start on even addresses, (ii) the tags on both words of the source equal 1, and (iii) the principal executing (**??**) has read access to the source and write access to the destination.

User-mode `invoke` and `return` instructions for invoking operations on objects would also have restrictions. If *cap* is a capability for object $O$ and *op* is an integer, then execution of

`invoke` *op*, @*cap*

might work as follows. A trap occurs if *cap* is not a capability, $0 \leq op \leq n$ does not hold, or privilege bit $p_{op}$ in *cap* equals 0. Otherwise, execution of the `invoke` (i) loads integer *op* into some well known register (say) `r1`, (ii) synthesizes a capability *retCap* of type `return` that records in its object name field the address of the instruction following the `invoke`, (iii) pushes @*retCap* onto the run-time stack, and (iv) executes instructions starting at address $O$.

When an `invoke` instruction is executed, the code starting at $O$ is presumed to be a `case` statement which, based on the contents of `r1`, transfers control to operation number *op* of $O$. Control is later transferred back to the invoker by popping @*retCap* off the run-time stack and executing

`return` @*retCap*

which loads the program counter with the contents in the object name field of *retCap* (the previously stored address of the instruction immediately after the

[11] We write @$w$ to denote the address of $w$.

`invoke` in the caller) and also, to prevent reuse, sets the tags in *retCap* to 0. Executing a `return` causes a trap if @*retCap* cannot be read or if *retCap* is not a capability that has type `return`.

Ordinary instructions executed in supervisor mode can suffice for most other capability manipulations, provided executing those instructions in user mode causes a trap if the instruction attempts to set a tag to 1 or change the contents of a word having a tag equal 1. System routines executed in supervisor mode would likely be provided for the following functionality.

- New objects and their capabilities are created by calling a system routine that instantiates the object, generates a corresponding capability *cap*, stores *cap* in the caller's address space, and returns @*cap* to the caller.
- Capabilities can be propagated from one principal to another that do not share an address space (so `cap_copy` would not work) by calling operating system routines to send and receive capabilities. The operating system presumably has access to every principal's address space and can execute the needed `cap_copy` instruction.
- The functionality of the `cap_copy` instruction might be extended to perform attenuation and amplification, where desired, by having the source and destination principals invoke system routines.
  - Attenuation is supported by a system routine that takes as inputs (i) a set *Rmv* of privileges to remove and (ii) the address of a capability having some set *Privs* of privileges; it returns the address of another capability for the same object but with $Privs - Rmv$ as its set of privileges.
  - Amplification is supported by a system routine that takes as inputs the addresses for two capabilities: one capability names an object $O$ with some type $T$ and a set $Privs_O$ of privileges, and the second capability gives type $T$ as its name (and `type` as its type) and a set $Privs_A$ of privileges; it returns the address of a new capability for object $O$ with type $T$ but having $Privs_O \cup Privs_A$ as its privileges.

Tagged-memory capabilities can even be used to ensure that appropriate privileges are held for each and every memory access that a principal makes. Figure 7.7 suggests a format for such a capability to a memory segment.[12] It particularizes the type and the privileges bit string in the capability format given above, as follows. Type 0...0 signifies that the capability is for a memory segment; object name $O$ gives the starting address of the memory segment; prefix $p_0$, $p_1$, and $p_2$ (labeled $R$, $W$, and $X$) of the privileges bit string specifies privileges for operations read, write, and execute; and suffix $p_3\, p_4 \ldots p_n$ specifies the segment length.

Such memory segment capabilities might be integrated into a processor's memory access logic, as follows.

[12] A *memory segment* is a contiguous region of an address space; it is defined by a starting address and a length.

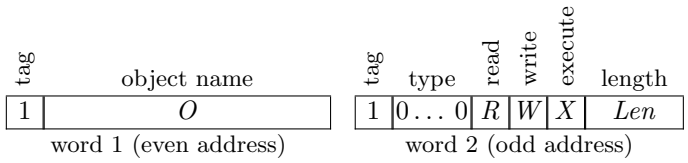



Figure 7.7: Example Format of Capability for a Memory Segment

- A set of *segment capability registers* store capabilities for memory segments.[13] And a memory access to some address $\alpha$ is allowed to proceed only if (i) $\alpha$ is the address of a word in some memory segment whose capability currently resides in a segment capability register and (ii) the requested operation (read, write, or execute) is one for which the corresponding privileges bit is set in that capability. An *access-fault* trap occurs, otherwise.

- The processor provides a supervisor-mode instruction

  `load_scr` *scr*, @*cap*

  for loading a segment capability register *scr* with the memory segment capability stored at address @*cap*; this instruction causes a trap if executed in user mode or when *cap* is not a capability having type 0...0.

The operating system then provides routines that use these hardware facilities and allow execution to *map* and *unmap* memory segments. The set of mapped memory segments at any given time defines a protection domain by establishing what memory can be addressed, hence what set of capabilities the executing principal holds. In some systems, the set of mapped memory segments for a given principal will be partitioned: memory accessible to every principal, memory accessible only to this principal throughout its execution, and memory accessible only while some operation on a given object is being executed.

An operating system might support having more memory segments being mapped at a given time than there are segment capability registers. To accomplish this, system software multiplexes the segment capability registers in much the same way that a small set of page frames is multiplexed to create a much larger virtual memory. Specifically, the operating system maintains a set *MappedSegs* of the capabilities for memory segments that are mapped. Whenever an access fault trap occurs, the corresponding trap-handler checks whether *MappedSegs* contains a capability *seg_cap* (say) for the memory segment encompassing the address that caused the access-fault. If *MappedSegs* does, then the trap handler replaces the contents of some segment capability register with *seg_cap* and retries the access; otherwise, the memory access attempt is deemed to violate the security policy.

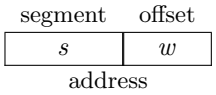
[13] In some architectures, these register might contain a capability for a segment that itself contains capabilities for segments. This additional level of indirection allows a small number of segment capability registers to support addressing a significantly larger number of segments.

### 7.3.2 Capabilities in Protected Address-Spaces

Modern processor hardware invariably enforces some form of memory protection, if only to protect the operating system by isolating its code and data from user programs. Typically, memory is partitioned into one or more regions and access restrictions are enforced on each region. Although coarse-grained in comparison to tagged memory, this simple form of memory protection does suffice for implementing capability authenticity.

We segregate capabilities and store them together in memory regions that cannot be written by user mode execution. Operating system routines, which execute in supervisor mode, are granted read/write-access to these memory regions. And functionality that requires creating or modifying capabilities is implemented by the operating system (rather than by special-purpose instructions, as for tagged memory). So there would be system routines for instantiating a new object (and its corresponding capability), copying capabilities, sending and receiving capabilities between principals, and the invocation and return from operations on objects (with attenuation and amplification).

**Capabilities Stored in Virtual Memory Segments.** One scheme for implementing this protected address-space approach uses segmented virtual memory. A *virtual address* here is a bit string; some predefined, fixed-length prefix of that bit string is interpreted as an integer $s$ that identifies a segment, and the remaining suffix specifies an integer offset for a word $w$ in the segment:




A *segment table*, which comprises a set of *segment descriptors*, is used during execution to translate virtual addresses into real addresses. See Figure 7.8. Each segment descriptor gives the name, length, and starting address for a segment, as well as *access bits* ($R$, $W$, and $X$) that indicate whether words in the segment can be read, written, and/or executed. The segment table thus defines an address space and access restrictions on the contents of that address space.

The operating system associates a segment table with an executing process by loading the (real) address of that segment table into the processor's *segment table register*, which is considered part of the processor context.[14] We can thus arrange for execution by the operating system and for execution by each process to use different segment tables and, therefore, to have different (virtual) address spaces and/or different access restrictions being enforced.

The use of segments to store capabilities is now straightforward. The access bits in segment descriptors allow us to implement one or more virtual address

[14]The *processor context* comprises the general-purpose registers, the program counter, and any other processor state that must be saved and restored when an operating system time-multiplexes the processor over a collection of tasks.

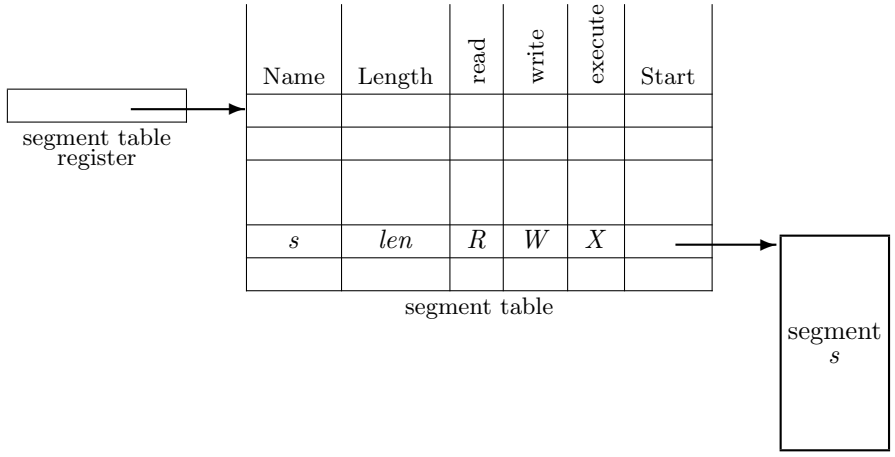



Figure 7.8: Addressing with a Segment Table

spaces where writes are prohibited. By requiring all user-mode execution to use such segment descriptors for accessing segments that contain capabilities, the operating system prevents user-mode execution from counterfeiting or corrupting capabilities. Principals are implemented as user-mode processes, with the set of capabilities held by a principal defined to be those capabilities stored in designated segments (which user-mode execution can read but not write).

The size and format of capabilities being implemented here is defined by software. And the choice of what segments are designated for storing capabilities is unconstrained. Often a convention is adopted (e.g., only segments named with low-numbers store capabilities), but alternatively the operating system could itself store the names of all segments dedicated to storing capabilities. Segment descriptors on some architectures contain bits unused by the address-translation hardware, and a run-time environment might employ these bits to indicate whether a segment stores capabilities versus ordinary data.

The use of memory segments to store capabilities does not preclude the use of capabilities to control access to memory. As above for the tagged-memory implementation of capabilities, the operating system would provide routines to map and unmap a memory segment. The map and unmap routines for a segment $s$ would check that the invoker holds a capability $cap_s$ for $s$, where $cap_s$ specifies appropriate read, write and/or execute access privileges. If the invoker does hold such a capability, then map (unmap) modifies the invoker's segment table and adds (deletes) the segment descriptor for $s$. The segment table thus simulates the set of segment capability registers, which explains why the information found in a segment descriptor is so similar to what is found in a memory segment capability.

When capability authenticity is implemented by memory segments, a segment table specifies which capabilities the executing process holds and which

other data it can access (because those data segments have been mapped). So a segment table defines a protection domain, and protection-domain transitions require changing that segment table.

For example, protection-domain transitions typically accompany operation invocations. An operating system routine to invoke an operation might expect to be passed two arguments:[15] the name *op* of the operation and the address @*cap* of a capability for an object on which *op* is to be performed. Execution then proceeds as follows.

1. *Authorization.* Validate that (i) the segment named by @*cap* is dedicated to storing capabilities (so *cap* is a capability), (ii) the caller can read *cap* (and thus holds that capability), and (iii) *cap* grants permission for *op*.

2. *Segment Table Construction.* If the checks in step 1 are satisfied then build a segment table containing segments for capabilities and state that should be accessible when performing *op* on the object named by *cap*.

3. *Control Transfer.* In software, orchestrate the designated transfers of control to and from *op* in *obj*: (i) construct a capability *retCap* of type `return` for the return address, (ii) push @*retCap* onto the run-time stack, (iii) load the segment table register with the address of the new segment table, and (iv) transfer control to the code for *op* in *obj*.

Run-time environment support for other functionality involving capabilities would be built along similar lines.

**Capabilities Stored in Kernel Memory.** Even if segmented virtual memory is not supported by a processor, we should expect to find a single memory region that can be accessed by supervisor-mode but not by user-mode execution.[16] To implement capabilities using such *kernel memory*:

- All capabilities are stored in the protected memory region.

- Each principal is implemented by a user-mode process.

Capabilities are then typically organized in tables, historically called *c-lists* (for "capability list"). Each entry in a c-list stores a single capability and is identified by its index into the table. Some schemes have a separate c-list for each process; in others, c-lists are shared by processes.

Operating system routines then provide the sole means for creating, examining, and manipulating capabilities and the c-lists that store them. For example, send and receive routines might be implemented to pass capabilities from one process to another, whether or not those processes share a c-list. Specifically, a process $P$ could invoke send, giving a destination process $P'$ and identifying

[15] These arguments give the same information that the `invoke` instruction expects to find in registers for the tagged-memory implementation of capabilities discussed in §7.3.1.

[16] Without such hardware support, the integrity of an operating system's local storage is easily compromised by user programs.

some capabilities that would then be buffered at $P'$ for receipt; by invoking receive, $P'$ would cause any buffered capabilities to be moved to a specified c-list and would obtain the indices where they are stored.[17]

The operating system would also provide routines for creating and managing c-lists. The routine for creating a new process might have an argument that specifies whether the new process should share access to the caller's c-list or get a new one (and, for a new c-list, what subset of the caller's capabilities and privileges are included). Call/return operations would not only transfer control but might cause a new c-list to be constructed temporarily for use by the invoked code. This new c-list could be populated with capabilities associated with the invoked code as well as attenuated and/or amplified versions of specified capabilities in the caller's c-lists.

### 7.3.3 Cryptographically-Protected Capabilities

The protections required for capability authenticity are well matched to the security properties that digital signatures provide. Consequently, digital signatures provide an alternative to hardware-implemented tags or protected memory. The costs—both in compute time and space—can be significant, but digital signatures are the only practical way to implement capabilities in some settings.

Our starting point is a digital signature scheme comprising algorithms to generate and to validate *signed* bit strings, where the following properties hold.

- *Unforgeability.* For any bit string $b$, only those principals that know private key $k$ can generate $k$-signed bit string $\mathcal{S}_k(b)$.
- *Tamper Resistance.* Principals that do not know $k$ find it infeasible to modify $\mathcal{S}_k(b)$ and produce a different $k$-signed bit string $\mathcal{S}_k(b')$.
- *Validity Checking.* Any principal that knows the public key $K$ corresponding to a private key $k$ can validate whether a sequence of bits is a $k$-signed bit string.

We then implement capabilities as signed bit strings, because the Unforgeability and Tamper Resistance properties imply capability authenticity[18] and the Validity Checking property gives a way for system components to ascertain whether a bit string represents an authentic capability.

Specifically, for *cap* a bit string that gives the name, type, and privileges for an object $O$, the $k_O$-signed bit string $\mathcal{S}_{k_O}(cap)$ serves as a capability that

[17] A system call to examine the capability stored for each index would be provided so $P'$ can determine what the new capabilities authorize.

[18] More precisely, this implementation of capability authenticity requires a digital signature scheme that is secure against selective forgery under a known message attack. Security against a known message attack accounts for the possibility that attackers might have access to some authentic $k$-signed bit strings (i.e., capabilities) but would not be able to get an arbitrary bit string signed (because any reasonable security policy the system enforces should preclude generating arbitrary capabilities on demand). Selective forgery is the right concern here, because we want to prevent an attacker from generating capabilities that grant specific privileges for specific objects.

grants its holders the specified privileges for $O$. Implicit, here, are the following assumptions about key distribution.

(i) Private key $k_O$ is known only by component(s) authorized to generate capabilities for $O$.

(ii) Corresponding public key $K_O$ is available to any principal needing to check the authenticity of a capability represented as $\mathcal{S}_{k_O}(cap)$.

Assumption (i) holds if the operating system enforces memory isolation between user-mode processes; and assumption (ii) can be discharged by broadly disseminating digital certificates signed by some well-known trusted authority. Notice that code to generate capabilities or to check capability authenticity need not execute in supervisor mode—user mode works just fine for performing these cryptographic calculations.

In some systems, only one component, such as the operating system, is authorized to create capabilities. So a single public key validates the authenticity of all capabilities. In other systems, a well known mapping defines which public key validates capabilities for each given object; the mapping typically uses some characteristic(s) of an object, such as its name or type, to select that public key. For instance, capabilities for all objects of each given type $T$ might be validated by a single well known public key $K_T$.

Capabilities implemented as signed bit strings are easy to transfer between processes on the same computer and even between different computers. It is just a matter of copying the bits (assuming infrastructure is in place to disseminate public keys for checking capability authenticity). However, performing amplification and attenuation for a capability $\mathcal{S}_k(cap)$ being transferred is another matter. The Tamper Resistance property implies that a component with knowledge of private key $k$ must be involved—either to generate a new capability with modified privileges or to modify the privileges in $\mathcal{S}_k(cap)$ directly. But sharing private keys is risky, and cryptographic computations are costly. Yet absent support for amplification and attenuation, the Principle of Least Privilege now becomes harder to support.

Three costs are noteworthy when a digital signature scheme is used to support capability authenticity: the amount of space required to store a signed bit string, the amount of time required to generate one, and the amount of time required for validity checking. To facilitate a comparison with the use of hardware-implemented tagged memory or protected address-spaces, suppose that the name, type, and privileges conveyed by a capability can together be represented in 64 bits.

For a digital signature scheme being deployed in 2010, NIST recommends 2048-bit RSA with SHA-256. The cost estimates that follow are derived from available implementations of those algorithms on commodity hardware, although the conclusions hold for other digital signature algorithms as well.

- To create a signed bit string $b$, a tag is appended to $b$. The length of this tag depends on the RSA key size and not on how long $b$ is; for 2048-bit

keys, that tag will be approximately 2048 bits. Thus, our implementation of capabilities as signed bit strings entails a substantial space overhead—a 2048 bit tag is required in order to protect 64 bits of content.

- The execution time required to create or check the validity of a tag for a 64 bit string $b$ is dominated by the RSA key size. Creation of $\mathcal{S}_k(b)$ using a 2048-bit RSA key takes many orders of magnitude longer than the time required for a kernel call; validity checking takes somewhat less time, but execution time still is orders of magnitude longer than the time required for a kernel call.

Needless to say, cryptographic protection is a relatively expensive way to implement capability authenticity.

Cryptographically-protected capabilities, however, can be attractive in distributed systems. Consider the alternatives and what they cost for that setting. If hardware-implemented tags or protected memory regions are used to implement capabilities, then transmitting a capability from one computer to another requires communication between the operating systems at those computers. The integrity and authenticity of that communication must be ensured. Digital signatures are the usual defense here. So signature generation and validity checking is necessary for transmitting a capability between computers—the costs of cryptographic computation that we were trying to avoid would be incurred, anyway.

Also note that cryptographic protection allows the authenticity of a capability to be checked locally, in user mode. Such a local check is considerably cheaper than querying the operating system on a remote computer that generated a capability. In fact, transferring a capability even between two principals executing on the same computer does not require the operating system to serve as an intermediary, which can have various advantages.

**Extension to Support Restricted Delegation.** Viewed abstractly, a principal $P$ that holds a capability $\langle O, Privs \rangle$ performs a restricted delegation to some other principal $P'$ by generating and forwarding to $P'$ a new capability $\langle O, Privs' \rangle$, where $Privs \supset Privs'$ holds. With the cryptographic implementation of capabilities described above, a principal holding a private $k_O$ must participate in every restricted delegation of capabilities for object $O$. In a distributed system, all holders of $k_O$ might be located on distant hosts; communication delays are now problematic. An extension we discuss here, overcomes the problem.

Instead of a single private key $k_O$, we admit an open-ended set of private keys. Some key from this set is included in each capability and, as before, capabilities are represented using signed bit strings.

> **Authenticity for Capabilities with Delegation.** A $k_i$-signed bit string does not represents an authentic capability $\langle O, Privs \rangle$ unless
>
> (i) $k_i = k_O$ holds, or

(ii) $k_i$ is the private key included in the representation of some authentic capability $\langle O, Privs' \rangle$, where $Privs' \supset Privs$ holds. □

Clause (i) will be satisfied by any authentic capability for $O$ that was generated along with the creation of object $O$. Clause (ii) concerns capabilities produced by restricted delegations. It defines authenticity for a capability $C$, recursively, in terms of checks involving all capabilities in the chain of restricted delegations that led to the creation of $C$.

Knowledge of the public key $K_i$ that corresponds to a private key $k_i$ suffices for checking whether a bit string is $k_i$-signed. Thus, to discharge clause (ii) for validating authenticity of a capability $C$, a principal requires access to the public keys and sets of privileges associated with all of the capabilities in the chain of delegations that produced $C$. By including that information in the representation of $C$, it is available when needed.

So a capability $\langle O, Privs_N \rangle$ is represented using signed bit string

$$\mathcal{S}_{k_N}(O,\ Pdg_O(Privs_N, K_{N+1}),\ k_{N+1}) \tag{7.5}$$

where $Pdg_O(Privs_N, K_{N+1})$ gives the *pedigree* for the capability by listing a sequence $d_0\, d_1\, \ldots\, d_N$ of *delegation certificates*

$$d_i = \mathcal{S}_{k_i}(O, Privs_i, K_{i+1}),$$

each characterizing a restricted delegation in the chain that led to the creation of the capability represented by (7.5).

If $K_0$ is chosen to be well known public key for object $O$ (so $k_0 = k_O$ and $K_0 = K_O$ hold) then $Pdg_O(Privs_N, K_{N+1})$ contains exactly the information needed to check the authenticity of capability (7.5). And we check authenticity of a capability by using its pedigree, as follows.

**Authenticity from Pedigree.** A capability represented by (7.5) is authentic if and only if the following tests are satisfied.

(i) Use well known public key $K_O$ for object $O$ to check that delegation certificate $d_0$ is a $k_O$-signed bit string.

(ii) For $0 \leq i < N$, use public key $K_{i+1}$ in $d_i$ to check that delegation certificate $d_{i+1}$ is a $k_{i+1}$-signed bit string.

(iii) Use public key $K_N$ to check that (7.5) is a $k_N$-signed bit string.

(iv) For $0 \leq i < N$, use $Privs_i$ from $d_i$ and use $Privs_{i+1}$ from $d_{i+1}$ to check that $Privs_i \supset Privs_{i+1}$ holds. □

The idea behind representation (7.5) is simple, even if Authenticity from Pedigree looks complicated with all those subscripts. $N$ in (7.5) is the number of restricted delegations that were performed to produce the capability being represented. And each delegation certificate $\mathcal{S}_{k_i}(O, Privs_i, K_{i+1})$ characterizes authentic capabilities $C'$ produced through restricted delegation from an authentic capability that itself is a $k_i$-signed bit string—such a $C'$ would be represented by a signed bit string that can be validated using public key $K_{i+1}$ (i.e.,

1. Choose a new private key $k_{N+2}$ with corresponding public key $K_{N+2}$
2. Construct pedigree $Pdg_O(Privs_{N+1}, K_{N+2})$ for the new capability:
   (a) Use $k_{N+1}$ from (7.5) to construct $\mathcal{S}_{k_{N+1}}(O, Privs_{N+1}, K_{N+2})$.
   (b) Append $\mathcal{S}_{k_{N+1}}(O, Privs_{N+1}, K_{N+2})$ to $Pdg_O(Privs_N, K_{N+1})$ found in (7.5), thereby obtaining $Pdg_O(Privs_{N+1}, K_{N+2})$.
3. Use $k_{N+1}$ found in (7.5) to construct $k_{N+1}$-signed bit string

$$\mathcal{S}_{k_{N+1}}(O,\ Pdg_O(Privs_{N+1}, K_{N+2}),\ k_{N+2})$$

that serves as the representation of authentic capability for $\langle O, Privs_{N+1}\rangle$.

Figure 7.9: Generation of a Restricted Delegation $\langle O, Privs_{N+1}\rangle$

a $k_{i+1}$-signed bit string) and must grant a subset of the privileges in $Privs_i$. Figure 7.9 gives steps for generating the representation for an authentic capability $\langle O, Privs_{N+1}\rangle$ produced by performing a restricted delegation from an authentic capability $\langle O, Privs_N\rangle$ represented using (7.5). So, for example, if a capability $\langle \texttt{foo}, \{\texttt{r}, \texttt{w}, \texttt{x}\}\rangle$ is generated with the creation of file `foo` and represented according to (7.5) by

$$\mathcal{S}_{k_{\texttt{foo}}}(\texttt{foo},\ \ \mathcal{S}_{k_{\texttt{foo}}}(\{\texttt{r}, \texttt{w}, \texttt{x}\},\ K_{rwx}),\ \ k_{rwx}) \tag{7.6}$$

then the procedure in Figure 7.9 would yield representations $repC^{r}_{\texttt{foo}}$ for capability $\langle \texttt{foo}, \{\texttt{r}\}\rangle$ and capability $repC^{rw}_{\texttt{foo}}$ for $\langle \texttt{foo}, \{\texttt{r}, \texttt{w}\}\rangle$ obtained by delegations on (7.6), as follows.

$$\begin{aligned} repC^{r}_{\texttt{foo}} &= \mathcal{S}_{k_{rwx}}(\texttt{foo},\ \ \mathcal{S}_{k_{\texttt{foo}}}(\{\texttt{r}, \texttt{w}, \texttt{x}\}, K_{rwx})\ \mathcal{S}_{k_{rwx}}(\{\texttt{r}\}, K_r),\ \ k_r) \\ repC^{rw}_{\texttt{foo}} &= \mathcal{S}_{k_{rwx}}(\texttt{foo},\ \ \mathcal{S}_{k_{\texttt{foo}}}(\{\texttt{r}, \texttt{w}, \texttt{x}\}, K_{rwx})\ \mathcal{S}_{k_{rwx}}(\{\texttt{r}, \texttt{w}\}, K_{rw}),\ \ k_{rw}) \end{aligned}$$

### 7.3.4 Capabilities Protected by Type Safety

Programs written in type-safe programming languages associate a *type* with each variable, and their execution is restricted to ensure that only values with a suitable type are stored in variables. Since support for capabilities also involves enforcing restrictions on values, a natural question is whether the restrictions type safety introduces can be used to implement the restrictions capabilities require. We answer in the affirmative here by defining types for capabilities, where (i) possession of a suitable capability is necessary for executing operations on objects, and (ii) capability authenticity is enforced.

**Type Safe Execution.** For our purposes, it suffices that a type $T$ defines (i) a set $vals_T$ containing values that includes the special constant $\perp$ indicating uninitialized, and (ii) a set $ops_T$ of *operations* defined on values in $vals_T$. The following restrictions are then enforced for *type-safe execution*:

**Type-Safe Assignment Restriction.** Throughout execution, variables declared to have type $T$ only store elements of $vals_T$. □

**Type-Safe Invocation Restriction.** Throughout execution, only operations in $ops_T$ are invoked for values in $vals_T$. □

For any types $T$ and $T'$, relation $T \preceq T'$ is defined to hold if and only if $vals_T \supseteq vals_{T'}$ and $ops_T \subseteq ops_{T'}$ both hold. Notice that $T \preceq T'$ characterizes when the type-safe execution restrictions above are not violated by storing values of type $T'$ in variables declared to have type $T$.

A static check of the program text can establish that execution of a given assignment statement will comply with the Type-Safe Assignment Restriction, as follows. Assignment statement $v := Expr$ evaluates $Expr$ and stores the resulting value in variable $v$. Letting $type(x)$ denote the type of a variable or expression $x$, the following condition implies that $Expr \in vals_{type(v)}$ holds, which is what Type-Safe Assignment Restriction requires for executions of $v := Expr$.

**Type-Safe Assignment.** Execution of $v := Expr$ cannot violate Type-Safe Assignment Restriction if $type(v) \preceq type(Expr)$ holds. □

This condition is statically checkable if the declaration for $v$ gives $type(v)$ and if declarations for the variables and operators in $Expr$ suffice to infer a type for the value $Expr$ produces.

Next, consider an *invocation statement*

$$\textbf{call } obj.m(Expr_1, \ldots, Expr_i, \ldots, Expr_N) \tag{7.7}$$

Here, $obj$ is a program variable that designates some object and $m$ names an operation. The definition for $type(obj)$ will give declarations for all operations supported on instances of $type(obj)$, where a declaration for an operation $m$ might have the following form.

$$m\text{: } \textbf{operation}(p_1\text{:}T_1,\ \ldots,\ p_i\text{:}T_i,\ \ldots,\ p_N\text{:}T_N)\ \ body_m\ \ \textbf{end}$$

This declares each formal parameter $p_i$ to have a type $T_i$ and associates program statement $body_m$ with the operation. Execution of invocation statement (7.7) assigns the value of each argument $Expr_i$ to the corresponding formal parameter $p_i$ and then executes $body_m$.

To ensure type-safe execution for invocation statement (7.7), we must be concerned both with the Type-Safe Assignment Restriction and with the Type-Safe Invocation Restriction. Assume that checking has established $body_m$ exhibits type-safe execution when started in a state where $p_i = Expr_i$ holds for $1 \leq i \leq N$. Type-Safe Assignment Restriction for (7.7) is then implied it the Type-Safe Assignment Restriction holds for $p_i := Expr_i$ where $1 \leq i \leq N$ which, according to Type-Safe Assignment, requires that $T_i \preceq type(Expr_i)$ holds for $1 \leq i \leq N$. And Type-Safe Invocation Restriction requires that $obj \neq \perp$ and m $\in ops_{type(obj)}$ hold. Three conditions thus characterize type-safe invocation statements.

**Type-Safe Invocation.** An invocation statement

**call** $obj.m(Expr_1,\ Expr_2,\ \ldots,\ Expr_N)$

for an operation

$m$: **operation**$(p_1{:}T_1,\ \ldots,\ p_i{:}T_i,\ \ldots,\ p_N{:}T_N)$ $body_m$ **end**

never violates the Type-Safe Assignment Restriction or the Type-Safe Invocation Restriction provided the following hold:

- $obj \neq \perp$
- $m \in ops_{type(obj)}$
- $type(p_i) \preceq type(Expr_i)$ for $1 \leq i \leq N$. □

Condition $obj \neq \perp$ must be checked at run-time if analyzing the program text cannot guarantee that it always holds prior to reaching the invocation statement; the other two conditions can be discharged statically by using the type declarations present in the program text.

**Capability Types.** We now can explore how capabilities might be implemented using types. For a type $T$ whose values $vals_T$ are objects and whose operations $ops_T$ include $m_1$, $m_2$, ..., $m_N$ (but perhaps others too), the *capability type*

$$\mathbf{cap}(T)\{m_1, m_2, \ldots, m_N\}$$

is a type whose set of values is the values of type $T$ and whose set of authorized operations is the subset of operations $m_1, m_2, \ldots, m_N$ also supported by $T$:

$$\begin{aligned} vals_{\mathbf{cap}(T)\{m_1,m_2,\ldots,m_N\}} &= vals_T \\ ops_{\mathbf{cap}(T)\{m_1,m_2,\ldots,m_N\}} &= ops_T \cap \{m_1, m_2, \ldots, m_N\} \end{aligned}$$

Substitution into the definition of $\preceq$, we get that

$$\mathbf{cap}(T)\{m_1, m_2, \ldots, m_P\} \ \preceq\ \mathbf{cap}(T')\{m_1{}', m_2{}', \ldots, m_Q{}'\}$$

holds if and only if $T \preceq T'$ and $\{m_1, m_2, \ldots, m_P\} \subseteq \{m_1{}', m_2{}', \ldots, m_Q{}'\}$ hold. Therefore, if $C \preceq C'$ holds for capability types $C$ and $C'$ then a Type-Safe Invocation for operation $m$ of an object designated by variable *obj* having type $C$ will exhibit type-safe execution even if an object having type $C'$ is stored in *obj*.

To make these definitions concrete and understand their implications, consider a type *dbase* that supports three operations: $read(x, val)$, $update(x, val)$, and $reset()$. We declare two variables to store capabilities for objects of type *dbase*:

**var** $cap1$ : $\mathbf{cap}(dbase)\{read, update\}$
$cap2$ : $\mathbf{cap}(dbase)\{read\}$

Assume that *cap1* designates object *db1* and *cap2* designates object *db2*.

Invocation statement **call** *cap1.update*(...), when its argument values have suitable types, satisfies Type-Safe Invocation because *cap1* $\neq \perp$ holds (by assumption) and *cap1* is declared to have a capability type that includes operation *update* (since $ops_{type(cap1)} = \{read, update\}$). So a principal holding a capability that includes an *update* privilege for *db1*—therefore, the principal should be authorized to execute *update*—is allowed to perform that operation by exercising that capability. An access that should be permitted is permitted.

What about an access that should be denied? A capability that does not include a privilege for *update* should not enable invocation of that operation. For example, *cap2* does not include a privilege for *update*. So an attempt to use *cap2* for such an invocation had better violate type-safety. And, indeed **call** *cap2.update*(...) does not satisfy Type-Safe Invocation, since *update* $\in$ $ops_{type(cap2)}$ does not hold.

**Generation of New Capabilities.** Expressions whose evaluation produce values having type capability will be called *capability expressions*. A careful definition for capability expressions enables type-safe execution to prevent forgery of capabilities. At a minimum, the class of capability expressions must include (i) expressions that manufacture a capability whenever a new object is created and (ii) expressions that materialize a capability already held by the principal evaluating the expression.

We embrace that minimum. To satisfy (i), we introduce capability expression **new**($T$). When **new**($T$) is executed, the run-time environment creates a new object having type $T$ and returns a capability authorizing all operations for that new object. And, to satisfy (ii), we define all variables declared with type capability to be capability expressions, thereby allowing existing capabilities to be retrieved for copying (perhaps between principals) and to be used for invocations.

> **Capability-Valued Expressions.** An expression *Expr* is defined to have type **cap**($T$)$\{m_1, m_2, \ldots, m_N\}$ if
>
> - *Expr* is the invocation of built-in function **new**($T$) and $m_1$, $m_2$, ... , $m_N$ are operations that objects of type $T$ support.
> - *Expr* is either a variable or function that was declared to have type **cap**($T$)$\{m_1, m_2, \ldots, m_N\}$. □

Returning to the type *dbase* example, consider two additional variables.

**var** *cap3* : **cap**(*dbase*){*read*, *update*, *reset*}
*cap4* : **cap**(*dbase*){*write*}

Thus, by definition, assignment statements

*cap1* := **new**(*dbase*), *cap2* := **new**(*dbase*), *cap3* := **new**(*dbase*)

all satisfy Type-Safe Assignment. And in each case, Type-Safe Assignment ensures that the variable being assigned ends up storing an object with support for those operations the capability authorizes. Assignment statement $cap4 := \mathbf{new}(dbase)$, however, does not satisfy Type-Safe Assignment: $ops_{cap4} \subseteq ops_{\mathbf{new}(dbase)}$ does not hold. Since execution of $cap4 := \mathbf{new}(dbase)$ would forge a *write* privilege, having $cap4 := \mathbf{new}(dbase)$ violate Type-Safe Assignment is exactly what we desire.

Capability-Valued Expressions also allows a variable with capability type to appear as the right-hand side of an assignment statement. Such assignment statements not only can be used to transfer capabilities between principals but they can implement attenuation of privilege. For example, consider $cap2 := cap1$, which satisfies Type-Safe Assignment because $type(cap2) \preceq type(cap1)$ holds. This assignment stores into *cap2* a capability for *db1* that authorizes fewer operations than *cap1* does; the assignment statement implements attenuation of privilege.

Type-Safe Assignment does not allow amplification of privilege, though. Assignment statement $cap1 := cap2$ does not satisfy Type-Safe Assignment because $type(cap2) \preceq type(cap1)$ does not hold. This means that program fragment

$$cap1 := cap2; \quad \textbf{call } cap1.update(\ldots) \tag{7.8}$$

is not type safe, which is exactly what we desire—allowing (7.8) to execute would enable a principal holding a capability (*cap2*) for *db2* that authorizes only *read* operations to invoke an *update* operation on *db2* (since **call** $cap1.update(\ldots)$ is type safe).

A form of amplification is intrinsic in the usual lexical scoping rule for variables declared in an object. This scoping rule stipulates that variables declared in an object may be named within that object's operations but not outside. For example, Figure 7.10 defines a type *dbase*. It declares a single variable *dbCntnt*, which stores the state of a *dbase* instance; the scoping rule allows *dbCntnt* to be named within the body of operations *read*, *update*, and *reset* but not elsewhere. A form of amplification thus occurs during execution of the operations, because the body of an operation can directly access the object's variables. Moreover, if variables with capability types are declared within an object, then only by executing an operation can these capabilities be exercised. So, a principal executing an operation has amplified privileges relative to what it had when executing outside the operation.

### 7.3.5 Revocation of Capabilities

Revoking a principal's authorization can be problematic when *Auth* is implemented using capabilities. To delete $\langle P, O, op\rangle$ from *Auth*, we must find and delete or invalidate all copies of capability $\langle O, Privs\rangle$ that $P$ is holding and where $op \in Privs$ is satisfied. Moreover, the rationale for revoking $P$'s authorization to $O$ might well apply to other principals that received copies of

```
type dbase = object
                var dbCntnt : map
                read: operation(x : field, var val : field)
                        val := dbCntnt[x]
                        end read
                update: operation(x : field, val : field)
                          dbCntnt[x] := val
                          end update
                reset: operation()
                        dbCntnt := ∅
                        end reset
             end dbase
```

Figure 7.10: Definition of type dbase

$\langle O, Privs \rangle$ directly or indirectly from $P$. With *transitive revocation*, we would be required to delete or invalidate those copies of $\langle O, Privs \rangle$ too.

**Brute-Force Approaches.** Brute-force approaches to find and delete copies of a capability are perhaps the most obvious approach to implementing revocation. However, this is feasible only if the capability of interest can be found by scanning a relatively small amount of storage. Capabilities that employ hardware-implemented tagged memory or cryptographic protection can be stored anyplace in a principal's address space; brute-force searching is infeasible here.

Brute-force approaches are feasible when capabilities are implemented by strong-typing declarations or by protected address-spaces in virtual memory segments or kernel memory, because then all capabilities are stored in a small number of easily identified memory locations. Some systems[19] employ brute-force approaches to revocation by adopting a hybrid approach to authorization. Here, capabilities and access control lists are both used. Access to persistent objects, such as files on disk, is controlled by access control lists; access to resources that do not outlive their creators is controlled by capabilities. So capabilities can be stored in kernel memory, where brute-force approaches to revocation are feasible.

**Intermediaries.** Another method for ensuring that capabilities concerning a given object can easily be found for deletion is to define an *intermediary* $\widehat{O}$ for each given object $O$. Capabilities that any principal holds for a given object $O$ are stored only in $\widehat{O}$. Therefore, invoking operations at intermediary $\widehat{O}$ is the only way that a client $P$ can exercise a capability for $O$.

The implementation of $\widehat{O}$ might (i) store a c-list of capabilities concerning $O$ for each principal or (ii) store a single copy of each capability $\langle O, Privs \rangle$ plus

[19] UNIX is an example of such a system. See §7.4 for details.

a list $L_{\langle O,Privs\rangle}$ of principals holding $\langle O, Privs\rangle$. In either case, a capability $\langle O, Privs\rangle$ that $P$ holds is now easily revoked, because it is stored in a well known location (i.e., in $\widehat{O}$). An intermediary is basically a reference monitor, and the information it stores is equivalent to an access control list. The approach then is just a capability-based implementation of access control lists.

**Revocation Tags.** An alternative to finding and deleting revoked capabilities is simply to block access attempts that use them. This can be implemented by ensuring that some reference monitor is consulted whenever a capability is used to perform an access. Include a *revocation tag* in each capability; a capability is now a triple $\langle O, Privs, revTag\rangle$. And, for each object $O$, the reference monitor maintains a set $RevTags_O$ containing revocation tags for revoked capabilities for $O$.

- *Revocation.* A capability $\langle O, Privs, revTag\rangle$ is revoked by adding revocation tag $revTag$ to $RevTags_O$. This operation is authorized only if the *revocation* privilege is present in $Privs$.
- *Validity Checking.* An access attempted through $\langle O, Privs, revTag\rangle$ is denied by the reference monitor if $revTag \in RevTags_O$ holds, because then $\langle O, Privs, revTag\rangle$ had already been revoked.

Capability authenticity is presumed to prevent principals from changing the revocation tag in a capability; the operating system is presumed to protect the integrity of $RevTags_O$ and to provide a system call for adding elements to $RevTags_O$ (but preventing other changes to $RevTags_O$).[20]

Capabilities that incorporate revocation tags can be used to support *selective revocation*. Here, we seek means to revoke capabilities that some subset of principals hold for a given object. One solution is to define a *capability-facsimile generation* operation **facGen**$(\cdot,\cdot)$, with associated privilege (say) $fg$. Provided $fg \in Privs$ holds, execution of

$$cap := \textbf{facGen}(\langle O, Privs, revTag\rangle,\ Privs')$$

generates and stores in $cap$ a capability $\langle O, Privs \cap Privs', revTag'\rangle$, thereby assigning to $cap$ a capability for $O$ that has a fresh revocation tag and (possibly) attenuated privileges.

Principals whose authorization for an object $O$ might have to be revoked all together can now be given capabilities having the same revocation tag. Moreover, if some principal $P$ passes a capability $\langle O, Privs, revTag\rangle$ to another principal and that capability is forwarded further, then all those copies of

[20] In the implementation just sketched, $RevTags_O$ grows without bound. This could be problematic when memory is finite. An element $revTag$ can be removed from $RevTags_O$ once all capabilities containing that revocation tag have been deleted. It would be possible to ascertain that those capabilities have all been deleted, for example, when a principal that cannot share its capabilities with other principals is terminated and all of its storage is reclaimed. Also, $RevTags_O$ can be deleted when object $O$ is deleted, so for short-lived objects the storage required by $RevTags_O$ is unlikely to be a problem.

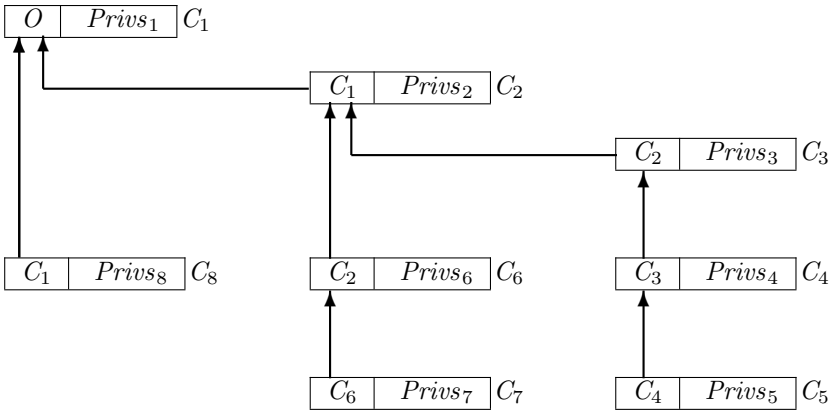



Figure 7.11: Chain of Capabilities

$\langle O, Privs, revTag \rangle$ are revoked when *revTag* is added to $RevTags_O$. Revocation tags thus support only a limited form of selective revocation: Prior to disseminating capabilities to principals, we must anticipate what sets of capabilities should together be revoked (because all capabilities in each such a set must share a revocation tag), and these sets must be non-intersecting (because each capability contains only one revocation tag).

**Capability Chains.** Indirection is the basis for our final approach to implementing revocation of capabilities. The idea is simple. We permit the object named in a capability to be another capability. Now, chains of capabilities can be constructed that lead to the capability for a given object $O$. And an access to $O$ using a capability $C$ is permitted if and only if (i) $C$ starts a chain of capabilities that all satisfy capability authenticity, (ii) the chain ends with a capability for $O$, and (iii) each capability in the chain contains privileges that authorize the requested access to $O$.[21]

Figure 7.11 gives an example. Each of the capabilities there (including $C_1$) starts a chain that ends with a capability for object $O$. So a principal $P$ is authorized to invoke an operation *op* on object $O$ if $P$ holds one of the capabilities $C_i$ depicted in Figure 7.11, and $op \in Privs_k$ holds for every capability $C_k$ in the chain from $C_i$ to $C_1$ (the capability for $O$).

[21] Condition (iii) is just one of several sensible schemes for deciding whether an operation should be permitted. An alternative would be to require that the last capability in the chain authorize the operation and all other capabilities traversed in the chain grant an *indirect* privilege. We might in addition require that the principal making the access hold all capabilities in the chain.

A chain of capabilities is severed if all copies of some capability $C$ that appears in a chain are deleted. After the chain is severed, access attempts that traversed $C$ in order to reach some capability $\langle O, Privs \rangle$ no longer succeed. So any principal that is authorized to delete all copies of $C$ is authorized to perform a selective revocation. For instance, with the chains in Figure 7.11, deleting all copies of $C_2$ revokes access to $O$ from capabilities $C_3$, $C_4$, $C_5$, $C_6$ or $C_7$; access to $O$ from $C_1$ or $C_8$ is unaffected, though.

The authorization for a principal $P$ to delete a capability $C$ might derive from $P$ being authorized to update the memory that contains $C$. Or it might derive from $P$ holding a capability $\langle C, \{..., delCap, ...\} \rangle$ for $C$, where the *delCap* privilege is required by the operation that deletes capabilities.

Capability chains support richer forms of selective revocation than revocation tags do. Because chains can overlap, deletion of (all copies of) a single capability could revoke a union of sets of capabilities, where each set could have each been revoked by itself. This is illustrated in Figure 7.11. Deleting all copies of $C_3$ also revokes $C_4$ and $C_5$; deleting all copies of $C_6$ also revokes $C_7$; and deleting $C_2$ revokes the union of those sets plus $C_3$ and $C_6$. Thus, in contrast to revocation tags, indirection allows non-disjoint sets of capabilities to be revoked.

## 7.4 Case Study: DAC in Unix

The access control policy implemented by UNIX[22] is concerned with authorizing requests that processes make to perform operations on files. However, file names are used in UNIX to name most other system resources, too. All operations on files and other system resources are implemented by operating system code. So authorization can be enforced by a reference monitor located in the operating system.

- A unique *user id* identifies a user, and a unique *group id* identifies a group of users. Each process executes with an *effective user id* and an *effective group id* that together specify the protection domain for that process.[23]

- Each file $F$ has an associated access control list, a user id $owner_F$ that is the file's *owner*, and a group id $group_F$ that is the file's *group*. This information is stored in the i-node for the file, along with other meta-data.

Only the owner of a file is permitted to change the access control list for that file, so UNIX implements DAC.[24]

[22] The mechanisms featured in this section should be common to most versions of UNIX and also will be found in many other of today's commercial operating systems.

[23] Newer versions of UNIX allow concurrent membership in multiple groups.

[24] In contemporary versions of UNIX, only a system administrator (user id `root`) is permitted to change a file's owner. This restriction prevents an unscrupulous user from changing the owner of a file as a means to (i) circumvent a resource accounting scheme that charges a file's owner for space occupied by that file or (ii) store stolen information in a file that seemingly implicates another user in the theft.

Access control lists in UNIX sacrifice expressiveness in favor of succinctness. The access control list for a file $F$ defines three sets of privileges: the owner's privileges $Privs_F.owner$, the group's privileges $Privs_F.group$, and others' privileges $Privs_F.other$. A process having *euid* as its effective user id and *egid* as its effective group id is authorized to perform an operation requiring a privilege $p$ provided the following holds.

$$\begin{array}{ll} & (p \in Privs_F.owner \ \wedge \ euid = owner_F) \\ \vee & (p \in Privs_F.group \ \wedge \ euid \neq owner_F \ \wedge \ egid = group_F) \\ \vee & (p \in Privs_F.other \ \wedge \ euid \neq owner_F \ \wedge \ egid \neq group_F) \end{array} \tag{7.9}$$

Authorization check (7.9) was doubtless selected by the UNIX designers because it can be efficiently evaluated by executing a nested **if-then-else** statement. The semantics that (7.9) defines is somewhat unintuitive, though. For instance, a process executing with effective group id *egid* might not be able to exercise a privilege $p$ on a file whose access control list authorizes $p$ to group *egid*. This is illustrated by a file $F$ where

$$Privs_F.owner = \{\mathtt{r}\}, \qquad Privs_F.group = \{\mathtt{r}, \mathtt{w}\}, \qquad Privs_F.other = \emptyset$$

hold. You would expect that a process for which $egid = group_F$ holds should be permitted to perform an operation requiring privilege `w`, since $\mathtt{w} \in Privs_F.group$ holds. However, if $euid = owner_F$ holds then such a request would be denied according to (7.9), because $\mathtt{w} \notin Privs_F.owner$ implies the first disjunct of (7.9) is *false* and $euid = owner_F$ implies the second and third disjuncts are *false* too.

UNIX employs three identifiers—`r`, `w`, and `x`—for designating file access privileges. A small number of bits thus suffices to represent the privilege sets that appear on UNIX access control lists, a design decision made when memory and disk space were dear. Which operations are authorized by identifiers `r`, `w`, and `x` depends on the type of file. UNIX distinguishes between *ordinary* files, *directories*, and *special* files. Ordinary files and directories are storage abstractions; special files provide a uniform way for authorizing access to I/O devices and system abstractions, such as ports. Figure 7.12 summarizes the intended effects of `r`, `w`, and `x` for ordinary files and for directories; the interpretation of `r`, `w`, and `x` for special files is idiosyncratic to the object.

From Figure 7.12, we conclude that the appearance of `x` on the access control list for an ordinary file authorizes execution of an `exec` system call naming the file. For ordinary files that store executables, specifying `x` but not `r` protects a proprietary executable, since it allows clients to execute the code but not steal it. A file that has `r` but not `x` forces an executable to be copied before it can be executed—the executable is now stored in a file having the client's user id as owner. For a directory, `x` is interpreted as allowing the `stat` system call. By restricting execution of `stat`, UNIX offers a way to block file accesses that include a specified directory on paths used to identify files.

**Domain Change in Unix.** Change the effective user id and/or the effective group id of process and, according to (7.9), the domain of that process changes.

| Privilege | File Type | |
|---|---|---|
| | Ordinary | Directory |
| `r` | can read file contents | can read file names stored in the directory but not other information |
| `w` | can change file contents or truncate file | can change directory contents, allowing file creation, deletion, renaming. |
| `x` | can execute file | can traverse directory to access files or subdirectories; can read information in the i-node for the file. |

Figure 7.12: Interpretations for UNIX Privileges

UNIX supports such domain change by providing `suid` and `sgid` privileges. Each extends the meaning of `x` for ordinary files that store executables.[25] When a file $F$ having `suid` executes, the effective user id is changed to $owner_F$; analogously, `sgid` causes the effective group id to be changed to $group_F$. So domain-change is coupled with the `exec` system call.

In early versions of UNIX, `suid` was used primarily by programs that implemented system services. Accesses by programs executing as `root` receive special treatment from the UNIX operating system—they are not subject to restrictions specified in access control lists. So programs implementing system services were executed with effective user id `root`. All manner of design sins could be overcome (but also committed) when system services are granted unfettered access to everything. This architecture, however, also meant that attackers could access any file simply by discovering an exploitable bug in a system service.

Subsequent versions of UNIX addressed this weakness by better embracing the Principle of Least Privilege.

- Fewer system services execute with effective user id `root`. Instead, distinct system services are assigned different user id's, all files associated with a given service are owned by that service-specific user id, and programs implementing the service execute `suid` to that user id.

- New system calls allow a process to change its effective user id without invoking `exec`. So domain change is no longer wed to `exec`, and fine-grained domains can be associated with regions of code within an executable. In addition, some UNIX versions add a single *saved* user id for each process and provide system calls that allow an effective user id to be stored there. This functionality enables a domain change that returns to what the effective user id was before. It also refines the definition of a protection

[25] Different versions of UNIX assign different semantics to `suid` and `sgid` for directories. The details are not important for our discussion.

domain to comprise a pair containing the effective user id and the saved user id.

**Lower-overhead Authorization Checks.** Because its designers were concerned about run-time overhead, UNIX systems only implement an approximation to the access control scheme just explained. Authorization is partitioned into a potentially expensive check, which is done infrequently, and cheaper checks, which are performed for each file access. The expensive check is moved into an additional system call. This `open` system call for a file must be executed prior to attempting `read` or `write` system calls on that file.

The constraint that `open` be executed first is enforced because `read` and `write` require a *file handle* argument (rather than a file name), and invoking `open` is the only way to obtain a file handle for a given file. The `open` system call takes as arguments (i) a full path that names a file and (ii) a bit mask specifying accesses (`r` and/or `w`). Provided (7.9) allows traversal of each directory named in (i) and also allows access to the file as specified in bit mask argument (ii) then execution of `open` returns a file handle that, thereafter, can be used for the specified accesses to this file. So `open` requires information stored in the i-node for a file as well as information from all directories traversed to reach that file. The cost for executing `open` thus could be high, even if all of the needed information is cached in main memory. But the cost for authorization of `read` and `write` is always low, since these operations need only check a small amount of system state stored for a given file handle.

The usual UNIX implementation of a file handle as an offset into a per-process *file-descriptor* table is analogous to naming capabilities by using indices into per-process c-lists. Execution of `open` returns the index; `read` and `write` perform operations on a file only when provided with an index for a file descriptor authorizing that operation on the file. Moreover a typical UNIX system also supports a `close` system call, which deletes a file descriptor and a `fork` system call, which creates a new process that executes with a copy of the invoker's file-descriptor table. Some UNIX systems even support inter-process communications with `send` and `recv` system calls that not only allow opaque data to be transmitted from one process to another but allow entries to be copied from a sender's file-descriptor table to the receiver's.

The hybrid of access control lists and capability-like authorization (file handles denoting file descriptors) just outlined is not a panacea. Its latency for revocations can be unbounded, because the access control list is not rechecked each time `read` and `write` execute (since only `open` does that check). So a process that successfully executes an `open` naming a file $F$ can continue executing `read` and `write` operations for $F$ long after the authorizing privileges have been deleted from the access control list for $F$ or from the access control lists for directories traversed to reach $F$. That exposure was unlikely to be long when hardware failures and software bugs meant that processes rarely ran uninterrupted for very long. Those days are long gone, however. Improvements in hardware speed and in software reliability bring significantly longer intervals of

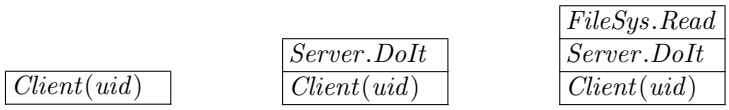
exposure, making it harder for designers of contemporary systems to rationalize reductions in authorization overhead by risking unbounded revocation times.

## 7.5 Case Study: Stack Inspection

By permanently associating domains of protection with programs, modules, or objects, we run the risk of confused deputy attacks. We can mitigate this risk by having privileges granted during execution of a *code unit* depend, at least in part, on execution history. So we are instantiating the Principle of Least Privilege, using knowledge about the past to establish what task is being performed and granting privileges accordingly.

An approximation to this history-based approach for authorization is provided by by *stack inspection*. Here, authorization decisions are based on aspects of the execution history recorded in a run-time stack for each process. A new frame is pushed onto this stack whenever execution of a code unit is invoked; that frame is popped when the code unit returns to its invoker. Thus, the run-time stack records all code units in which a statement $S$ either is actively being executed or is suspended awaiting termination of a code unit that $S$ invoked.

As an illustration, consider the client-server system of Figure 7.13. If we monitor the run-time stack during execution, we find a single frame just before $C_1$ in *Client* starts executing as user *uid*, two frames when $S_1$ in *Server* starts executing ($C_1$ is suspended), and three frames while $F_2$ in the *FileSys.Read* operation invoked by $S_1$ is executing ($C_1$ and $S_1$ are suspended):




$C_1$ $\quad$ $C_1{\uparrow}S_1$ $\quad$ $C_1{\uparrow}S_1{\uparrow}F_2$

Here, we are using infix $\uparrow$ operator[26] in labels for snapshots—a label $S{\uparrow}S'{\uparrow}S''$ indicates that statement $S$ is suspended having invoked a code unit containing statement $S'$, $S'$ is suspended having invoked a code unit containing statement $S''$, and $S''$ is executing.

In authorization policies defined using stack inspection, a set of privileges is derived from the run-time stack. Run-time stacks thus define the domains of protection for this authorization regime. And the set of privileges authorized by a given run-time stack is derived from sets of privileges associated with the frames comprising that run-time stack—each frame gets privileges from a declaration for the code unit whose invocation the frame records. Figure 7.14 declares file access privileges for code units *Client*, *Server*, and *FileSys*. The declarations imply that stack frames for execution of *Client* under user id *uid* are

[26] By convention, we depict a stack by drawing its base towards the bottom of the page, and the stack grows in the upward direction. The $\uparrow$ in $S{\uparrow}S'$ conveys the direction of stack growth when $S$ invokes $S'$.

associated with `read` and `write` privileges to files in directory */fsys/Users/uid* but have no privileges for */fsys/Server/acntFile* or any other files; both `read` and `write` privileges for file */fsys/Server/acntFile* are associated with stack frames for execution of *Server*.

The set of privileges enforced for the protection domain defined by a run-time stack is the intersection of the privilege sets associated with the frames comprising that stack. Any larger set would admit execution by a code unit but with the privileges declared for that code unit not being enforced, which seems undesirable. Notice that the intersection-based definition causes the confused deputy attack being attempted by $C_2$ in *Client* to fail in statement $S_4$ for lack of a `write` privilege to */fsys/Server/acntFile*. In particular, the privileges declared for code unit *Client*(*uid*) do not include a `write` privilege for */fsys/Server/acntFile*, and therefore any intersection of privilege sets that involves a stack frame for *Client*(*uid*) will not either. Thus the *CheckPrivilege* at $F_3$ fails whenever *FileSys.Write* is invoked by $S_4$ from $C_2$.

```
Client: process(uid)
          C1: Server.DoIt( "/fsys/Users/uid/dataFile" )
          C2: Server.DoIt( "/fsys/Server/acntFile" )
        end Client

Server: service
          DoIt: operation( f : file)
                  S1: buffer := FileSys.Read( f )
                  S2: results := F( buffer )
                  S3: chrgs := calcBill( results )
                  S4: FileSys.Write( f, results )
                  privileged do
                    S5:  FileSys.Write( "/fsys/Server/acntFile", chrgs )
                    end
                  end DoIt
        end Server

FileSys: service
           Read: operation( f : file) : string
                   F1: CheckPrivilege( ⟨f, read⟩ )
                   F2: fetch and return contents of file f
                   end Read
           Write: operation( f : file; v : string)
                   F3: CheckPrivilege( ⟨f, write⟩ )
                   F4: extend file f with v
                   end Write
         end FileSys
```

Figure 7.13: Client/Server using Stack Inspection

| Code Unit | Objects | Privileges |
|---|---|---|
| *Client*(*uid*) | */fsys/Users/uid/∗* | `read`, `write` |
| *Server* | */fsys/Users/∗* | `read`, `write` |
| | */fsys/Server/acntFile* | `read`, `write` |
| *FileSys* | */fsys/∗* | `read`, `write` |

Figure 7.14: Privileges Declared for Client/Server Code Units

The intersection-based definition for protection domains implies that a protection domain transition accompanies execution that causes a stack frame to be pushed or popped. Since invocation pushes a new frame onto the run-time stack, invocations cause protection domain transitions. By definition, the new protection domain's privileges intersects an additional set of privileges. Furthermore, because $P \supseteq (P \cap Q)$ holds for all sets $P$ and $Q$, we conclude that the privilege set associated with the new protection domain cannot be a superset of the set before. The domain of protection entered by executing an invocation thus cannot have added privileges.

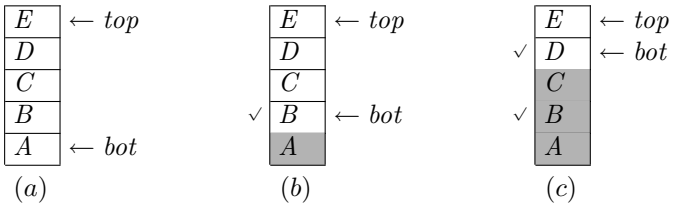
Yet there are situations where added privileges are necessary. For instance, successful execution of *FileSys.Write* invoked by $S_5$ in *Server.DoIt* requires a `write` privilege for file */fsys/Server/acntFile*, despite the absence of that privilege from the intersection of privilege sets associated with frames on the run-time stack. Execution of $S_4$, however, had better not be granted that same privilege or else the confused deputy attack at $C_2$ will succeed. A notation to specify amplification of privilege is needed.

We should be willing to grant additional privileges to specified statements whose executions can be trusted not to abuse those privileges. To achieve this effect, we allow a stack frame temporarily to be marked *privileged* during execution, and we redefine the set of privileges associated with a run-time stack. The new definition omits from the intersection all stack frames that appear below one that is marked privileged. So the set *DomPrivs*(*rts*) of privileges associated with the protection domain for a run-time stack *rts*[1..*top*] is formally defined as follows

$$DomPrivs(rts) = \bigcap_{bot \le j \le top} FPrivs(rts[j]) \tag{7.10}$$

where *FPrivs*($f$) is the set of privileges associated with a stack frame $f$, and *bot* identifies the top-most stack frame marked privileged (with $bot = 1$ if *rts* contains no frames marked privileged).

We illustrate the definition of *DomPrivs*(*rts*) by depicting three snapshots of a run-time stack, where ✓ indicates stack frames that are marked privileged. In each, *DomPrivs*(·) is the intersection of the associated privilege sets for only the non-shaded stack frames.



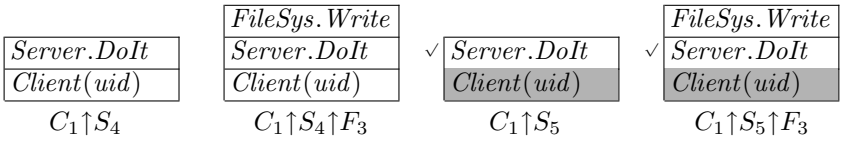

Notice that while $E$ is executing, $DomPrivs(\cdot)$ for snapshot (b) can contain privileges not declared for $A$; for snapshot (c), it can contain privileges not declared for $A$, $B$, or $C$.

We have thus far ignored how programmers specify when a stack frame becomes marked privileged. One approach is for "marked privileged" to be among the attributes that can be declared for a code unit. If a code unit is invoked having that attribute declared, then the frame pushed onto the run-time stack would be marked privileged. Only small blocks of statements typically need to exercise amplified privileges, so packaging these blocks as separate code units can be problematic. First, invocation is an expensive form of control transfer. Second, parameter-passing now must be employed to share state with the blocks of statements being granted amplified privileges.

Systems that support stack inspection avoid these problems by adopting a finer-grained approach. They introduce a syntax

$S$: **privileged do** $T$ **end**

for specifying a block $T$ of statements that should be executed with amplified privileges. Execution of $S$ causes the frame currently at the top of the run-time stack to be marked privileged just before $T$ is executed; when $T$ terminates, the frame is unmarked if it was unmarked when $S$ started execution.[27] In Figure 7.13, $S_5$ of *Server* appears in a **privileged do** but $S_4$ does not. The following snapshots of the run-time stack are possible.




The invocation of *FileSys.Write* by $S_4$ (snapshot $C_1{\uparrow}S_4{\uparrow}F_3$) thus does not execute with a `write` privilege for file */fsys/Server/acntFile*, because that privilege is not declared for code unit *Client*(*uid*). But the invocation by $S_5$ (snapshot $C_1{\uparrow}S_5{\uparrow}F_3$) does, since privileges associated with the stack frame for *Client*(*uid*) are no longer part of the intersection that defines the set of privileges associated with the run-time stack. Consequently, the confused deputy attack at $C_2$ is foiled.

[27] Although this operational semantics supports arbitrary nesting of **privileged do** statements, no additional privilege amplification results from such nesting.

### 7.5.1 Stack Inspection Implementation Details

The obvious implementation of stack inspection is simply to use a stack for storing the privilege sets associated with each code unit that is executing or suspended:

**var** *pds* : **array**[0 .. ] **of record**
*fPrivs* : **set of privileges initial**($\emptyset$)
*marked* : **boolean initial**(*false*)
**end record**
*pdsTop* : **integer initial**(0)

Code for the various operations involving this stack appears in Figure 7.15. Invocation of a code unit *CU* causes associated set *FPrivs*(*CU*) of privileges to be pushed onto *pds*; return causes that stack to be popped. Entry/exit from a **privileged do** marks/unmarks the frame appearing at the top of the stack. And the code for *CheckPrivilege*(*p*) uses a loop[28] that, starting from the top frame of *pds*, checks for privilege *p* in *pds*[·].*fPrivs* sets, thereby computing whether *p* is a member of *DomPrivs*(*pds*) according to definition (7.10).

**Using Stack Compression.** In the above implementation, all of the operations except *CheckPrivilege* have small and fixed run-time costs. The run-time cost of *CheckPrivilege*, however, is linear in the height of the *pds* stack. This linear cost can become problematic if the number of *CheckPrivilege* operations is anticipated to be large or if *CheckPrivilege* invocations appear in code units whose invocations are deeply nested.

A fixed-cost implementation of *CheckPrivilege* is possible, though. We again employ a stack of privilege sets

**var** *cpds* : **stack of set of privileges**

but now we arrange for the each stack frame to store the set of all privileges associated with a protection domain—not just those privileges associated with a single code unit. In particular, we ensure

$$cpds.\mathbf{top}() = DomPrivs(rts) \tag{7.11}$$

[28] We have that $1 \leq pdsTop$ holds whenever *CheckPrivilege* is invoked, because invocations of *CheckPrivilege* appear only inside code units. So the following is an invariant for the loop in *CheckPrivilege*.

$$1 \leq i \leq pdsTop \quad \wedge \quad (\forall j : i < j \leq pdsTop : \neg pds[j].marked)$$

$$\wedge \quad inInter \;=\; p \in \left( \bigcap_{i \leq j \leq pdsTop} pds[j].fPrivs \right)$$

When the loop terminates, $pds[i]marked \vee 1 = i$ holds, which implies that $pds[i]$ must be marked privileged and/or be the frame at the bottom of the stack. Moreover, since the loop invariant holds when the loop terminates, we conclude that *inInter* equals *true* if and only if $p \in DomPrivs(rts)$ holds.

```
Invoke: operation( CU : code_unit )
          pdsTop := pdsTop + 1
          pds[pdsTop].fPrivs := FPrivs(CU)
          pds[pdsTop].marked := false
          end Invoke

Return: operation
          pdsTop := pdsTop − 1
          end Return

DoPrivEnter: operation
               pds[pdsTop].marked := true
               end DoPrivEnter

DoPrivExit: operation
              pds[pdsTop].marked := false
              end DoPrivExit

CheckPrivilege: operation( p : privilege )
                  var i : integer
                      inInter : boolean
                  inInter := (p ∈ pds[pdsTop].fPrivs)
                  i := pdsTop
                  while ¬pds[i]marked ∧ 1 ≠ i do
                          i := i − 1
                          inInter := inInter ∧ (p ∈ pds[i].fPrivs)
                          end
                  if ¬inInter then throw protection_exception
                  end CheckPrivilege
```

Figure 7.15: An Implementation of Stack Inspection

holds throughout execution, where *cpds*.**top**() evaluates to the contents of the frame appearing at the top of stack *cpds*. Figure 7.16 compares a snapshot of stack *pds* and the equivalent snapshot of stack *cpds* for a hypothetical partial execution $A{\uparrow}B{\uparrow}C{\uparrow}D{\uparrow}E$, where the statement suspended in code unit $B$ and the one executing in $E$ are within **privileged do** statements. Notice how a single frame of *cpds* may store the intersection of privileges stored by a sequence of frames in *pds*, thereby "compressing" privilege sets into a form ideally suited for the stack inspection operations.

A *CheckPrivilege* implementation having fixed run-time cost now is trivial. *CheckPrivilege*($p$) simply checks whether $p \in$ *cpds*.**top**() is *true*. Code for that operation and for the others appears in Figure 7.17. This code exploits the insight that an invocation or entry operation causing transition from one protection domain $PD$ to a new one $PD'$ is ultimately followed by an operation causing the transition from $PD'$ back to $PD$. If the transition to new protec-

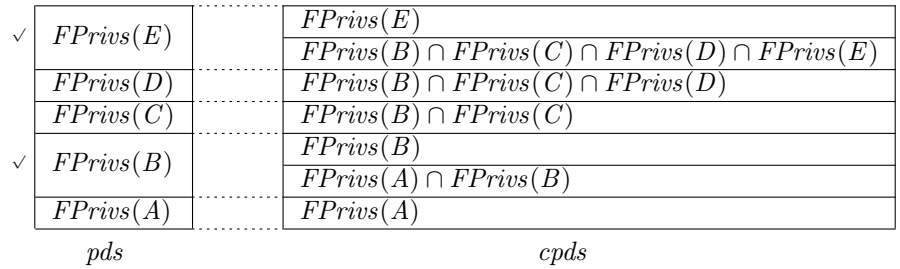



Figure 7.16: Stack Compression Example

tion domain $PD$ is being implemented by $cpds$.**push**($PD$) and definition (7.11) holds, then the transition back can then be implemented by $cpds$.**pop**(). Moreover, each operation requires only a fixed number of stack operations, thereby having fixed run-time cost.

A more-detailed explanation of the code for each of the operations in Figure 7.17 follows.

- If a code unit $CU$ is invoked when the protection domain $PD$ is stored by the top-most stack frame of $cpds$ then there will be a transition to protection domain $FPrivs(CU) \cap PD$. By pushing $FPrivs(CU) \cap PD$ onto $cpds$, (7.11) will continue to hold—the set of privileges associated with the

```
Invoke: operation( CU : code_unit )
          var pd : set of privileges
          pd := cpds.top()
          cpds.push(FPrivs(CU) ∩ pd)
          end Invoke

Return: operation
          cpds.pop()
          end Return

DoPrivEnter: operation
               cpds.push(FPrivs(currCU))
               end DoPrivEnter

DoPrivExit: operation
              cpds.pop()
              end DoPrivExit

CheckPrivilege: operation( p : privilege )
                  if p ∉ cpds.top() then throw protection_exception
                  end CheckPrivilege
```

Figure 7.17: Stack Compression Implementation

current protection domain is now stored in *cpds*.**top**(). The subsequent return from *CU* causes a transition to the protection domain that was in force prior to the invocation; the pop to *cpds* thus does not invalidate (7.11).

- If a **privileged do** in a code unit *CU* commences when the protection domain *PD* is stored by the top-most stack frame of *cpds* then, according to (7.10), the protection domain will transition to *FPrivs*(*CU*). If we assume *currCU* is the currently executing code unit, then by pushing *FPrivs*(*currCU*) onto *cpds*, (7.11) will again hold—we restore to *cpds*.**top**() the set of privileges associated with the current protection domain. Exit from that **privileged do** causes a transition to the protection domain that was in force prior to the push of *FPrivs*(*currCU*); a pop on *cpds* thus suffices to re-establish the truth of (7.11).

### 7.5.2 Privileges for Code Units

Stack inspection was originally developed for defending against attacks conveyed in web pages or other content downloaded from the Internet. For that setting, the identity of the author or of another other trusted principal that attests to authenticity of some code seemed a reasonable basis for predicting the trustworthiness of that code. Code believed to be trustworthy is granted privileges for accessing system resources; other code is granted virtually no privileges and, therefore, its ability to inflict damage is limited.

Two schemes were then employed for associating privileges with code units:

- *Associate sets of privileges with code libraries.* Where a code unit is loaded from thus determines its privileges.
- *Associate sets of privileges with digital signatures.* When multiple signatures on a code unit are allowed, then the code unit is granted the union of the privilege sets associated with all of the signing keys.

Code installed in system libraries or digitally-signed by well known software providers is now easily distinguished from code obtained from potentially disreputable sources. And that was the original goal. We might, however, also have as our goal to support the Principle of Least Privilege at the granularity of code units. That, unfortunately, is not easily achieved if what set of privileges is granted to a code unit is determined by its library or digital signature(s). This is because we would likely want distinct code units each to be granted different sets of privileges; the number of separate code libraries and/or signing keys would become unwieldy when the number of code units is large.

## Exercises for Chapter 7

**7.1** DAC policies are defined in §7.1 with a model that comprises an authorization relation *Auth* and a set $\mathcal{C}$ of commands. As with any model, certain things

are easy to express but other things can be awkward or impossible to express. Illustrate this incompleteness by describing access control policies that would be awkward or impossible to express in this model.

**7.2** In some systems, the same object might have different names. We see this in many guises: (i) when shared memory segments are mapped to different virtual addresses in different processes, (ii) when aliasing occurs in a high-level language program through parameter passing or the use of pointers, (iii) when either a symbolic or a hard "link" is created for accessing the same file from different directories, and (iv) when a device in a network is named by its network address and that device is "multi-homed".

(a) What, if any, opportunities could this functionality provide to attackers if $O$ in $\langle P, O, op\rangle$ could be one of many names that principals use for the object.

(b) Explain how objects that might have multiple names could be accommodated by extending the model §7.1 comprising an authorization relation $Auth$ and set $\mathcal{C}$ of commands.

**7.3** Consider an authorization relation $Auth$ and a set $\mathcal{C}$ of commands that together specify an access control policy. Must this necessarily specify a DAC policy? If so, explain why. If not, give conditions on $Auth$ and/or $\mathcal{C}$ that ensure the policy will be DAC.

**7.4** What efficiently checked constraints on the initial value of $Auth$ and any set $\mathcal{C}$ of commands would ensure that each of the following access control policies are being enforced? If such constraints cannot exist, then explain why; if they do exist, indicate the extent to which the constraints rule out things that they shouldn't.

(a) At most one principal has privileges for accessing each given object.

(b) Any principal granted privilege `w` for an object is also granted privilege `r`.

(c) Principal $P_0$ is never granted privilege `r` for object $Obj_0$.

(d) No principal is ever granted a privilege that would subsequently allow principal $P_0$ to be granted a privilege `r` for object $Obj_0$.

**7.5** Consider a graph $G = \langle N, E, L\rangle$ comprising a finite set $N$ of nodes, a finite set $E$ of edges (each edge, a pair of nodes), and a function $L$ that assigns a label $L(e)$ to each edge $e$. $G$ can be represented by a diagram where (i) dots correspond to nodes and (ii) for each edge $e = \langle n, n'\rangle$ in $E$, an arrow labeled $L(e)$ is drawn from the dot representing $n$ to the dot representing $n'$.

(a) Describe how the information in an authorization relation $Auth$ can be represented by such a graph.

(b) Give necessary and sufficient conditions on such a graph for it to define an authorization relation *Auth*.

(c) The transitive closure $G^*$ of graph $G$ is defined to be the graph $\langle N, E^*, L\rangle$ where $E^*$ is the smallest set of edges such that

- $E \subseteq E^*$ holds, and
- if edges $\langle n, n'\rangle$ and $\langle n', n''\rangle$ are in $E^*$ and have the same label then so is an edge $\langle n, n''\rangle$ with that label.

What kinds of insights become apparent from the diagram of transitive closure $G^*$ for a graph $G$ that corresponds to a given authorization relation *Auth*.

**7.6** For each of the following, (i) devise a sensible authorization policy, (ii) model it by using an authorization relation *Auth* and a set $\mathcal{C}$ of commands, and (iii) explain whether the authorization policy is DAC.

(a) Every user $U$ of a file system has a separate directory $D_U$ which, for each file that it lists, associates either a read (`r`) or read/write (`rw`) privilege as well as a list of all users authorized to `link` that file. $D_U$ is updated by the system whenever (i) $U$ invokes a system call to `create` or `delete` a file or (ii) $U$ invokes a system call to `link` or `unlink` to a file in another user's directory. So $D_U$ contains an entry for every file that $U$ has created (but not yet deleted) or linked (but not yet unlinked). Execution of system calls to `read`, `write`, `create`, `delete`, `link`, and `unlink` is restricted in the expected way.

(b) An e-cash system is implemented using objects—each called a *reserve note*—to represent transferable sums of money. The system includes operations to create a reserve note (presumably because some goods of equivalent value have been produced), delete a reserve note (in exchange for a good with equivalent value), and transfer a reserve note from one user to another.

(c) The users of a course-management system are students, graders, and professors. The objects it manages include assignment descriptions, student-submitted solutions, answer keys, and grades. Operations are supported so that a student may submit a solution, read the answer key, and/or look-up the grade; a grader may read the answer key, read and annotate a student solution, and/or assign a grade (but cannot change that grade, thereafter); a professor may post an assignment description, post an answer key, and/or review a student solution for which a grade has already been assigned and then post an updated grade.

**7.7** Consider a collection of fine-grained objects $Obj_1$, $Obj_2$, ..., $Obj_n$. A set $Privs_i$ of privileges is associated with accesses to object $Obj_i$, and an access control policy is specified in terms of an authorization relation *Auth* and set $\mathcal{C}$ of

commands. Given is a system that (only) supports access control for relatively coarse-grained objects, $Obj'_1$, $Obj'_2$, ... $Obj'_m$ where $m < n$ holds. Suppose each coarse-grained object groups a set of fine-grained objects. Describe an authorization relation $Auth'$, set $\mathcal{C}'$ of commands, and sets $Privs'_i$ of privileges for each $Obj'_i$ to ensure that the authorization requirements imposed by the original fine-grained access control restrictions will still be enforced

**7.8*** Consider the following candidates for a restriction that might be imposed on a set $\mathcal{C}$ of commands. For each candidate, (i) explain whether the restriction makes privilege propagation into a decidable problem and (ii) support your claim with an undecidability argument or with a decision procedure.

(a) No principals or objects are created or deleted.

(b) No principals or objects are created, but principals and objects may be deleted.

(c) No principals are created but objects can be created; principals and objects may be deleted.

(d) The action for each command contains a single assignment statement that adds or deletes a single triple.

**7.9*** Consider a fixed set $\{P_1, P_2, \ldots, P_n\}$ of principals, which also serve as the sole objects in the system. Suppose there are only two kinds of privileges. Privilege `pp` (for propagation permitted) does not propagate beyond its initial assignment in *Auth*; privilege `t` (for token) can propagate according to the following commands.

*grantToken*$(P, P')$: **command**
**pre**: *invoker*$(P) \wedge \langle P, P', \texttt{pp}\rangle \in Auth \wedge \langle P, P, \texttt{t}\rangle \in Auth$
**action**: $Auth := Auth \cup \{\langle P', P', \texttt{t}\rangle\}$

*takeToken*$(P, P')$: **command**
**pre**: *invoker*$(P') \wedge \langle P, P', \texttt{pp}\rangle \in Auth \wedge \langle P, P, \texttt{t}\rangle \in Auth$
**action**: $Auth := Auth \cup \{\langle P', P', \texttt{t}\rangle\}$

There are no other commands.

(a) Is the privilege propagation problem for `t` decidable for this system? If it is, then give an efficient algorithm for deciding whether some sequence of *grantToken* and *takeToken* commands can cause $\langle P, P, \texttt{t}\rangle \in Auth$ to hold for any given initial value of *Auth* and principal $P$; if it is not decidable, then give a proof.

(b) If additional privileges may not be introduced, then are there commands that can be added to the system and have the answer to (a) change? If so, then describe those command(s); if not, then explain why.

(c) If additional privileges may be introduced, then are there commands that can be added to the system and have the answer to (a) change? If so, then describe those privileges and command(s); if not, then explain why.

**7.10** An access control list

$$\langle P_1, Privs_1\rangle\ \langle P_2, Privs_2\rangle\ \ldots\ \langle P_L, Privs_L\rangle$$

is defined to have length $L$ provided $i \neq j$ implies $P_i \neq P_j$. Consider the possible access control lists for an object that appears in a system with $n$ principals, where there are $m$ different kinds of privileges.

(a) If the system does not include support for groups, then what is the longest possible access control list?

(b) Suppose $n > m$ holds and the system includes support for groups comprising subsets of the original $n$ principals. Then (i) what is the longest access control list possible and (ii) is it ever necessary to construct that list if our concern is with access authorization but not with review of privileges or with changes to group compositions or to the privileges granted to each principal?

(c) Suppose $n < m$ holds and the system includes support for groups comprising subsets of the original $n$ principals. Then (i) what is the longest access control list possible and (ii) is it ever necessary to construct that list if our only concern is with access authorization but not with review of privileges or with changes to group compositions or to the privileges granted to each principal?

**7.11** A *regexp access control list* is a sequence of pairs

$$\langle RE_1, Privs_1\rangle\ \langle RE_2, Privs_2\rangle\ \ldots\ \langle RE_n, Privs_n\rangle$$

where (i) each regular expression $RE_i$ characterizes a set $\mathcal{L}(RE_i)$ of principals[29] and (ii) $Privs_i$ is the non-empty set of privileges being granted to principals in $\mathcal{L}(RE_i)$. What are the advantages and disadvantages of this scheme over access control list syntax (7.3) described in §7.2.

**7.12** Instead of storing ACL-entries $\langle P_i, Privs_i\rangle$ in a list, we might employ a data structure that supports having principal names be retrieval keys. What are the advantages and disadvantages of the following candidates.

(a) Hash table.

(b) Binary search tree.

[29] Recall, regular expressions provide a terse way to characterize sets of finite sequences. For example, regular expression "$(a + b)^*$" denotes finite sequences that contains 0 or more $a$'s and $b$'s interleaved in any order, whereas regular expression "$(fb + eg)s$" denotes set $\{fbs, egs\}$.

**7.13** Capabilities could be described as an authorization mechanism that is based on "something you have". How might we analogously describe the following mechanisms for controlling access to confidential information?

(a) Access control lists.

(b) Encryption.

**7.14** Figure 7.7 depicts a 2-word format for a memory segment capability. Segments are typically multiples of the system's page size, but ordinary variables are typically much smaller than that. So if access control to individual variables is desired then the Principle of Separation of Privilege would lead to partially-filled pages, hence wasted memory. Discuss the advantages and disadvantages of supporting a capability format for much small-sized regions of memory—single bytes, words, or double-words.

**7.15** The capability format discussed in §7.3.1 spans 2 words starting at an even address. Memory usage can be improved by (i) allowing a capability to start at any address rather than only at an even address, and (ii) adopting a variable-length format for capabilities.

(a) Explain why embracing (i) and (ii) might lead to better memory usage.

(b) What disadvantages come with embracing (i) and (ii).

(c) Is it sensible to adopt (i) without (ii)? Explain.

(d) Is it sensible to adopt (ii) without (i)? Explain.

**7.16** Many operating system kernels provide a separate memory segment to each user-mode process. Only the associated process is permitted read, write, and/or execute access to that segment. Suppose such a kernel does not support capabilities but does support message-passing for communication between processes. Moreover, suppose the receiver of a message always learns the sender's identity in addition to learning a message body. Describe how to implement support for capabilities to user-space objects by using a single user-mode process (but not modifying the kernel).

**7.17** We are given a $k$-signed bit string $\mathcal{S}_k(cap)$ that represents a capability $cap$. According to the properties of a digital signature scheme, knowledge of private key $k$ is needed for computing representation $\mathcal{S}_k(cap')$ for a capability that amplifies or attenuates privileges conveyed by $\mathcal{S}_k(cap)$. We might, however, contemplate representing $cap$ using a set, where each member is a $k$-signed bit string that conveys only one of the privileges that $\mathcal{S}_k(cap)$ did. Discuss the functionality and practicality of this new approach.

**7.18** A large, sparse space for object names can be used in place of digital signatures for implementing capability authenticity. We represent a capability for $Obj$ by using a pair $\langle Nme(Obj),\ Privs\rangle$, where $Nme(Obj)$ is a random 128

bit string and *Privs* is the set of privileges conferred by the capability. The function $Nme(\cdot)$, if it exists at all, is kept secret. What functionality expected for capabilities does this alternative support and where (if at all) does it fall short?

**7.19** Suppose that all operations on files are implemented by a file service *FS*. *FS* returns a capability $cap_F$ when a file $F$ is created. That capability must be provided to *FS* with any subsequent requests for operations involving $F$. Can shared key cryptography be used to implement capability authenticity for these capabilities even if these capabilities can be forwarded from one client to another but no other services use them for authorization? If so, outline a scheme that uses the minimum number of keys.

**7.20** Define *cap′* and *cap* to be *split capabilities* for a privilege $p$ if:

(i) Together, *cap* and *cap′* authorize privilege $p$.

(ii) Neither *cap* nor *cap′* alone suffices to authorize privilege $p$.

Explain how to support this functionality in systems providing cryptographically-protected capabilities.

**7.21** A `dlg` privilege might be defined in connection with capabilities, as follows. The holder of a capability $\langle O, Privs \rangle$ is permitted to delegate that capability to some other principal only if `dlg` $\in$ *Privs* holds. Explain how to support this functionality by modifying Authenticity from Pedigree (page 124) for restricted delegation of cryptographically-protected capabilities.

**7.22** Suppose that an invocation statement can appear as (part of) the expression in the right-hand side of an assignment statement, as illustrated in

$$x := op(v_1, ...v_i, ...v_n).$$

Thus, the syntax of operation declarations is extended to specify a type $T_{ret}$ for the value when operation *op* is performed.

$op$: **operation**$(p_1{:}T_1, \ \ldots, \ p_i{:}T_i, \ \ldots, \ p_N{:}T_N) : T_{ret}$
    $body_{op}$
    **end**

(a) Describe any extensions necessary to Type-Safe Assignment (page 126) for supporting this new functionality.

(b) Describe any extensions necessary to Type-Safe Invocation (page 127) for supporting this new functionality.

**7.23** A claim is made in §7.3.5 that capability chains support richer forms of selective revocation than revocation tags.

(a) Give a capability chain that permits selective revocation and argue why revocation tags cannot be used to achieve that same effect.

(b) Explain how a capability chain can be constructed to achieve the same effect as any given sets of revocation tags.

**7.24** With an eye toward supporting additional forms of selective revocation, we adopt the following generalization of revocation tags.

(i) Revocation tags are $n$-tuples $\langle v_1, v_2, \ldots, v_n \rangle$ of integers. These $n$-tuples are ordered by relation $\preceq$, where

$$\langle v_1, v_2, \ldots, v_n \rangle \preceq \langle w_1, w_2, \ldots, w_n \rangle$$

holds if and only if $v_i \leq w_i$ for $1 \leq i \leq n$.

(ii) Access attempted through a capability $\langle O, \mathit{Privs}, \mathit{revTag} \rangle$ is denied if $\mathit{revTag} \preceq \tau$ holds for some $\tau \in \mathit{RevTags}_O$.

Compare the expressive power of this scheme for selective revocation against what can be achieved using ordinary revocation tags or using capability chains.

**7.25** Consider a system that supports public-key cryptography and also provides a scheme to revoke private keys. Due to this key-revocation scheme, any and all attempts to use a public key $K$ will fail after corresponding private key $k$ has been revoked.

(a) Discuss how the key-revocation scheme can be used to support revocation of capabilities $\langle O, \mathit{Privs} \rangle$ implemented as ordinary $k$-signed bit strings $\mathcal{S}_{k_O}(O, \mathit{Privs})$. Assume that $K_O$ is a well known public key for checking authenticity of capabilities for object $O$.

(b) Compare the flexibility of your proposal in part (a) with what could be achieved using revocation tags and with what could be achieved with capability chains.

(c) Discuss how the key-revocation scheme can be used to support revocation of capabilities $\langle O, \mathit{Privs} \rangle$ implemented as signed bit strings according to (7.5).

(d) Compare the flexibility of your proposal in part (c) with what could be achieved using revocation tags and with what could be achieved with capability chains.

**7.26** Construct an authorization relation *Auth* that models the semantics of access control lists in UNIX. Explain how your construction supports the protection domain changes possible in UNIX. For simplicity, assume that each user is a member of exactly one group (as was the case for the very first version of UNIX).

**7.27** A process having *euid* as its effective user id and *egid* as its effective group id is authorized to perform an operation requiring a privilege $p$ provided (7.9)

holds. What are the consequences of replacing (7.9) with the following.

$$\begin{aligned} & (p \in Privs_F.owner \ \wedge \ euid = owner_F) \\ \vee \ & (p \in Privs_F.group \ \wedge \ egid = group_F) \\ \vee \ & (p \in Privs_F.other) \end{aligned}$$

**7.28** Give examples of situations where the following combinations of UNIX privileges could be particularly useful.

(a) an ordinary file with `w` but not `r`.

(b) a directory with `x` but not `r` or `w`.

**7.29** The text (page 136) draws a parallel between UNIX file descriptors and capabilities. What are the similarities and differences?

**7.30** Consider a UNIX file `/dir/foo` that is the only file in a directory `/dir`. Some user has read access to the file. Enumerate settings of the protection bits that would block all accesses by users who are not the owner of `/dir` or `/dir/foo`.

**7.31** Various schemes for associating privileges with code units are mentioned in §7.5.2. Consider the following approach:

> Each code unit *CU* includes a preamble that explicitly enumerates the privileges $FPrivs(CU)$ to be associated with that code unit.

This scheme seemingly gives programmers latitude to stage attacks—the ill-intentioned programmer of *CU* incorporates a preamble that is too liberal in what privileges it associates with *CU*.

(a) Discuss whether this fear is well founded by comparing the risks for this scheme with the risks present when privileges for each code unit are stored in a single centrally-controlled database.

(b) If the scheme is in use, what steps can an honest developer take when writing a code unit *CU*, so that no code unit invoked by *CU* is allowed to perform certain pre-specified actions.

**7.32** The information depicted in Figure 7.14 is a set of triples, each specifying a code unit, an object, and a set of privileges. Despite any resemblance, the set of such triples mean something different than an authorization relation. Explain the differences and discuss what would be involved in constructing an authorization relation *Auth* from a table like that of Figure 7.14

**7.33** $NDP(rts)$ below is an alternative definition for the set of privileges asso-

ciated with the protection domain for a run-time stack *rts*.

$$NDP(rts)\colon \begin{cases} FPrivs(\mathbf{top}(rts)) & \text{if } \mathbf{empty}(\mathbf{pop}(rts)) \\ NDP(\mathbf{pop}(rts)) \cap FPrivs(\mathbf{top}(rts)) & \text{if } \neg\mathbf{empty}(\mathbf{pop}(rts)) \\ & \quad \wedge \neg IsPriv(\mathbf{top}(rts)) \\ NDP(\mathbf{pop}(rts)) \cup FPrivs(\mathbf{top}(rts)) & \text{if } \neg\mathbf{empty}(\mathbf{pop}(rts)) \\ & \quad \wedge IsPriv(\mathbf{top}(rts)) \end{cases}$$

$NDP(\cdot)$ and $DomPrivs(\cdot)$ differ in how stack frames marked privileged are handled.

(a) Can $NDP(rts) = DomPrivs(rts)$ hold for some snapshot of some run-time stack *rts*? If so, give example code units, associated declarations of privileges, and indicate the call sequence for the snapshot. If not, explain why.

(b) Can $NDP(rts) \subset DomPrivs(rts)$ hold for some snapshot of some run-time stack *rts*? If so, give example code units, associated declarations of privileges, and indicate the call sequence for the snapshot. If not, explain why.

(c) Can $NDP(rts) \supset DomPrivs(rts)$ hold for some snapshot of some run-time stack *rts*) in a system of code units? If so, sketch the code units and associated declarations.

(d) Discuss why $DomPrivs(rts)$ is a more sensible definition for protection domains than $NDP(rts)$ is.

**7.34** Using definition (7.10) of $DomPrivs(rts)$, prove that (as asserted in footnote 27) no additional privilege amplification is ever achieved by nesting **privileged do** statements.

**7.35** Consider a social network, comprising *individuals* linked to each other according to a relation *BFF*. In particular, access to postings is based on *BFF*, as follows: If $\langle I, I' \rangle \in BFF$ holds then individual $I$ is authorized to read postings by individual $I'$. Goals that might constrain *BFF* as it applies to a given individual $I$ could include

(i) restrict which individuals can read postings by $I$, and

(ii) restrict which individuals' postings $I$ can read.

(a) Give a plausible situation where goal (i) is useful.

(b) Give a plausible situation where goal (ii) is useful.

(c) To what extent do (i) and/or (ii) become more difficult to preserve if the *BFF* relation is symmetric and/or it is transitive.

## Notes and Reading

The term "discretionary access control" (DAC) was popularized by the Trusted Computer System Evaluation Criteria (TCSEC) [11], later known as the Orange Book from the color of its cover. TCSEC characterized security features and levels of assurance for computing systems operated by the U.S. Department of Defense. However, as early as 1974 in Walter et al. [45, page 12], authorization policies where "The user gives access rights at his own discretion ..." had been identified as a distinct and interesting class. The DAC premise—that the owner of an object $O$ is the authority about which principals could have access to $O$—considerably predates work in computer security. It instantiates a defining characteristics of private property, with origins in political and legal philosophy of Blackston, Bastiat, and others starting in 1760. So it was a natural choice in the 1960's for builders of early time-shared computer systems, who saw an analogy between digital content and physical objects.

In parallel with the development of time-sharing, Computer Science was maturing into a full-fledged academic discipline. That required establishing agreement about the content for a set of courses. Several curriculum-design task forces were supported by the COSINE Committee of the National Academy of Engineering, including one [12] for an undergraduate elective course on operating system principles. Lampson was a member, and he suggested that access control be presented in terms of a mathematical function. Lampson's *access function* mapped two inputs—a (protection) domain[30] [*sic*] and an object—to a set of privileges. The task force hesitated, fearing that functions were too mathematical for the intended audience. So Lampson switched to using a 2-dimensional matrix [31].[31] This now-familiar access matrix preserved his pedagogical insight that access control lists and capabilities were different realizations of the same mathematical abstraction.

Lampson's presentation [31] gives a few rules for changing an access matrix to support of privilege propagation; Graham and Denning [21] goes considerably further. That paper gives a set of basic operations for manipulating an access control matrix and uses these to form rules that are structured like the commands of §7.1. The way was now paved for the Harrison, Ruzzo and Ullman [23] proofs that privilege propagation is undecidable in general but decidable in so-called mono-operational protection systems (see exercise 7.8(d)). These computability results, despite having limited practical significance, alerted the research community to potential benefits of having a theoretical foundation for access control. Other researchers then explored alternative models (e.g., [34]), hoping to characterize real access control mechanisms yet have privilege propagation be decidable.

From the outset, however, research in computer security was driven largely by implementation efforts. The utility of a proposed principle or mechanism

[30] The term domain had been previously introduced by Lampson [30] as an alternative to the more verbose "spheres of protection" used earlier by Dennis and van Horn [13].

[31] Access functions were not completely abandoned in the prose by Lampson [31, page 439]: " ... and an *access matrix* or access function which we will call $A$."

would be evaluated by designing and building a new time-sharing system. For example, Lampson's description [30] of the (never completed) BCC Model I time-sharing system makes the case for small protection domains. And the Hydra operating system kernel [46] showed that Amplification of Privilege was crucial for capabilities to objects having user-defined types.[32] Prior to the work on Hydra, Attenuation of Privilege was the widely-accepted basis for privilege propagation across protection-domain transitions (with amplification the rare exception for highly-trustworthy components).

Access control lists and capabilities both were developed at about the same time at MIT. The term "access control list" is first used in a December 1965 paper [10] by Daley and Neumann describing a Multics file system. However, the Compatible[33] Time Sharing System (CTSS) which was developed at MIT before Multics, did have a feature that foreshadowed access control lists. Daley, also developer of the CTSS file system, had employed a single system-wide file to specify when each user could link to files owned by others; CTSS PERMIT and REVOKE commands [7, §AH.3.05] updated that file. The Multics file system replaced that single CTSS system file with per-file data structures, obtaining access control lists.

Dennis and van Horn [13] coined[34] the term "capability" and outlined an operating system supervisor to support this approach to access control; an implementation of that supervisor was running on a Digital Equipment Corporation PDP-1 mini-computer by October 1967 [1]. Fabry [18] later argued for using capabilities as the sole means to address and control access to all system objects: memory, processes, files, devices, etc. And over a decade later, Hardy [22] described the confused deputy problem, strengthening that case.

Various research groups have explored the utility, limitations, as well as software and hardware-assisted implementations of capabilities. See Levy [32] for an authoritative account of systems prior to 1984 that supported capabilities. A summary of that material would not do it justice; the discussion here about capabilities is limited to summarizing origins of material covered in §7.3.

Hardware-implemented tags had been in use since 1961 in the Burroughs B5000 system [3]. The IBM System/38 [2][32, chptr 8], general purpose computer, not only implemented capabilities in hardware but was commercially quite successful. System/38 was intended as the successor to the then-popular IBM System 360 architecture. But IBM came to realize that existing System 360 customers might be lost to competitors if forced to rehost software on a new architecture. That led IBM to develop System 370 as a System 360 successor,

[32] The term amplification is not actually used by Hydra researchers until Jones and Wulf [27].

[33] "Compatible" because CTSS allowed programs compiled for its predecessor—Fortran Monitor System (FMS)—to be executed unchanged.

[34] Iliffe [25] points out that the Rice University computer [26] codeword mechanism and the Burroughs B-5000 [3] descriptor elements both embody the concept of a capability and preceded Dennis and van Horn [13] by at least 5 years. Dennis and van Horn [13], in fact, does acknowledge the B5000 as an inspiration. But Dennis and van Horn [13] was the first to introduce the term capability and to discuss a role for capabilities in managing processes and other operating system abstractions. So Dennis and van Horn [13] is considered the defining paper for capabilities.

adding virtual memory and multi-processor support to the System 360 architecture. System/38, however, was quite successful for hosting small and mid-range businesses that were first computerizing their operations and, thus, not already invested in System 360.

The c-lists of §7.3.2 were originally suggested by Dennis and van Horn [13], and the approach is still widely used today for implementing capabilities in operating systems. Plessey 250 [17][32, chptr 4], introduced the idea of storing capabilities in a separate segment, the alternative to c-lists we describe in §7.3.2. Plessey 250 was also the first operational capability system to be sold commercially. It was intended for telephone-switching applications, where reliability is crucial—the Plessey 250 designers believed that a strong protection mechanism would prevent bugs in one process from crashing another.

Encryption was suggested in Chaum and Fabry [4] as a means of implementing capability authenticity. An operating system developed for the Octopus network at Lawrence Livermore National Laboratory [14] appears to be the first use of digital signatures for capability authenticity. Developers of the Amoeba distributed operating system [39] subsequently explored other cryptographic approaches, concentrating on lighter-weight one-way functions and hardware support; that work inspired exercise 7.18, although using a large sparse name space is suggested by Chaum and Fabry [4]. van Renesse [37], one of Amoeba's designers, suggested the approach to restricted delegation in §7.3.3.

Jones and Liskov [29] in spring 1976 shows how type safety could implement capability authenticity; the scheme outlined in §7.3.4 is essentially that work. It combines two separate threads of research from the preceding decade: abstract data types from programming languages and capabilities from operating systems.

- The programming languages thread starts with Simula 67 [9], which supported a type `class`. This new type enabled programmers to define abstract data types and thus facilitated an object-oriented approach to structuring programs and systems.[35] Abstract data types subsequently became the focus of much research in languages and methodology. Among the most visible efforts was one lead by Liskov, starting in 1973, to develop the CLU programming language [35].

- Operating systems researchers at CMU in the early 1970's were developing the Hydra kernel [46] to explore program structures that exhibited a clear separation between policy and mechanism. And objects seemed like an ideal structuring mechanism for that purpose.[36] Jones had participated in the design of Hydra, and her 1973 Ph.D. dissertation [28] focused on developing Hydra capabilities for enforcing access control in object-oriented

[35] A predecessor language, Simula I, intended for programming discrete-event simulations had also been developed by Dahl and Nygarrd; it was running on UNIVAC 1107 machines by December 1964 [24]. Probably more influential was chapter 3 by Dahl and Hoare in a collection [8] of three monographs. That entire volume is still worth reading.

[36] An object's policy was the expectations its clients could have about effects of invoking the object's operations; its mechanism was how those operations were implemented.

systems.

Jones and Liskov together, then, were well positioned to incorporate capabilities into an object-oriented programming language. Their success further reinforced the view that Hydra was demonstrating and that others [33] had embraced—that capabilities were ideal for object-oriented systems.

Almost a decade elapsed after capabilities were first proposed before alternatives emerged to brute-force approaches for revocation. Intermediaries were used for revocation in the first version of Hydra [27] but, by 1975, Hydra [6] employed a scheme derived from Redell's Ph.D. dissertation [36]; that scheme was not unlike capability chains. The revocation tags approach described in §7.3.5 is a special case of conditional capabilities from Ekandadham's Ph.D. dissertation [15], work that became widely available [16] too late to have real impact. Gligor [19] is a good source about the rationale and requirements for capability revocation and review in centralized systems.

The designers of today's widely-used commercial operating systems have largely eschewed capabilities. A natural question is: Why? Is it inertia from design decisions made for early time-sharing systems or do compelling reasons still exist for avoiding capabilities? Early time-sharing systems were designed to serve the needs of their users, given the characteristics of available hardware. Security was rarely a high priority for those users, few programmed in object-oriented languages, and capabilities seemed expensive to implement. So there are could be many reasons why few commercially-successful early time-shared systems were capability-based.

But commercial operating systems today are deployed in a rather different setting—security is higher priority for users and operators, object-oriented languages are widely used, and implementation of capabilities is practical on commodity hardware. Inertia nevertheless seems to have prevailed, which suggests industry believes that building next-generation operating systems based on capabilities would be too expensive, too risky, or unlikely to be embraced by the market. That last rationale seems the most likely. First, companies that have large investments in software and expertise for one architecture are understandably hesitant about transition costs that accompany a new one. Second, inherent in access control lists is a centralized point of control for each resource, whereas capabilities can be propagated in ways that a resource owner has neither control nor knowledge. Limitations in control and visibility are uncomfortable for people—especially management and especially in networked settings.

We selected UNIX for a case study not only because UNIX is among today's widely-used operating systems but because it introduced ideas—for access control and elsewhere—that many have adopted. UNIX development started as a rogue project at AT&T Bell Laboratories on a discarded PDP-7 computer in the summer of 1969. Management had just decided to withdraw from a collaboration with GE, IBM, and Project MAC to develop the Multics time-sharing system at MIT—intended to support 1000 users, Multics was far behind schedule and could handle only 3 users.[37] A quirky development path (see, for example,

[37]Originally designed to support only a single user, the first version of UNIX was

the history [40]) ultimately led to a version of UNIX in 1970 that supported text processing applications on a PDP-11. Widespread interest and the first external adoptions of the system followed from a public presentation at the Fourth ACM Symposium on Operating Systems (in 1973); the classic paper [38] by Richie and Thomson is based on that presentation. See Chen et al. [5] for an in-depth discussion of `suid` and proposed safer semantics for supporting domain change in UNIX.

Stack inspection is the best known (and perhaps only) general-purpose DAC authorization mechanism where access decisions reference execution history. The idea was developed during summer 1996 at Netscape by Wallach [42], working as an intern in collaboration with Roskind and Tenneti; it was then documented in a paper [43] Wallach coauthored with his thesis adviser and Roskind.

Sun Microsystems had released the HotJava browser in May 1995, introducing the possibility for local execution of downloaded content (dubbed applets). Applets would be written in the then little-known Java programming language, which Sun had developed for use in consumer electronics. A binary authorization policy was initially adopted—browsers granted no privileges to applets but granted full privileges to code downloaded from the local disk (where installed browser extensions were stored). That policy was implemented in Netscape's browser by counting stack frames to predict whether the currently executing routine would be an applet. This implementation was ad hoc, and stack depth measures had to be rechecked whenever browser software was re-engineered.

A binary authorization policy was too restrictive for applets to implement interesting functionality. So the next step was an authorization policy for granting additional privileges to applets that were digitally signed by trusted sources. Netscape's stack inspection was developed to implement that by generalizing from stack frame counting. That same scheme was ultimately incorporated by Sun into JDK 1.2 (later called Java 2), the new, more-secure version of Java [20].[38] Various implementation of stack inspection have since been explored [44, 41], and stack inspection is supported in Microsoft's C# as well as other contemporary programming languages intended for the Internet.

# Bibliography

[1] William B.Ackerman and William W. Plummer. An implementation of a multiprocessing computer system. In *Proceedings of the First ACM symposium on Operating System Principles*, SOSP '67, pages 5.1–5.10, New York, NY, USA, 1967. ACM.

named UNICS (Un-multiplexed Information and Computing Service), an allusion to Multics (Multiplexed Information and Computing Service).

[38] Sun's aspirations for Java to be a general programming language led to some small differences between Sun's definition of stack inspection and Netscape's. Neither company nor the growth of the World Wide Web would have been well served by having both alternatives in the market. Unable to resolve the dispute themselves, their managements invited IBM to perform binding arbitration. IBM sided with Sun, and that was adopted by all [20].

[2] Viktors Berstis. Security and protection of data in the IBM System/38. In *Proceedings of the 7th annual symposium on Computer Architecture*, ISCA '80, pages 245–252, New York, NY, USA, 1980. ACM.

[3] Burroughs Corporation. *The Descriptor—A Definition of the B5000 Information Processing System*, 1961. Michigan.

[4] D. L. Chaum and R. S. Fabry. Implementing capability-based protection using encryption. Technical Report UCB/ERL M78/46, University of California at Berkeley, July 1978.

[5] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association.

[6] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth ACM symposium on Operating Systems Principles*, SOSP '75, pages 141–160, New York, NY, USA, 1975. ACM.

[7] P. A. Crisman, (editor). *The Compatible Time-Sharing Sstem. A Programmers Guide.* The M.I.T. Computation Center, 1965. http://www.bitsavers.org/pdf/mit/ctss/CTSS_ProgrammersGuide_Dec69.pdf.

[8] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming.* Academic Press Ltd., London, UK, UK, 1972.

[9] Ole-Johan Dahl. *SIMULA 67 Common Base Language, (Norwegian Computing Center Publication).* 1968.

[10] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference*, AFIPS '65 (Fall, Part I), pages 213–229, New York, NY, USA, 1965. ACM.

[11] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, August 1983. CSC–STD–001–83, Library Number S225,711.

[12] Peter J. Denning, Jack B. Dennis, Butler Lampson, A. Nico Haberman, Richard R. Muntz, and Dennis Tsichritzis. An undergraduate course on operating systems principles. *Computer*, pages 40–58, January/February 1972.

[13] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, March 1966.

[14] James E. (Jed) Donnelley. Managing domains in a network operating system. In *Proceedings of Local Networks and Distributed Office Systems Conference*, pages 345–361, May 1981.

[15] Kattamuri Ekanadham. *Context Approach to Protection*. PhD thesis, State University of New York at Stony Brook, 1976.

[16] Kattamuri Ekanadham and Arthur J. Bernstein. Conditional capabilities. *IEEE Transactions on Software Engineering*, SE-5(5):458–464, September 1979.

[17] D. M. England. Operating system of System 250. In *Proceedings of International Switching Symposium*, June 1972.

[18] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17:403–412, July 1974.

[19] Virgil D. Gligor. Review and revocation of access privileges distributed through capabilities. *IEEE Transactions on Software Engineering*, SE-5(6):575–586, November 1979.

[20] Li Gong. Java security architecture revisited. *Communications of the ACM*, 54:48–52, November 2011.

[21] Scott G. Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, May 1972. ACM.

[22] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22:36–38, October 1988.

[23] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19:461–471, August 1976.

[24] Jan Rune Holmevik. Compiling SIMULA: A historical study of technological genesis. *IEEE Annals of the History of Computing*, 16:25–37, December 1994.

[25] J. K. Iliffe. Surveyor's forum: An error recovery. *ACM Computing Surveys*, 9:253–254, September 1977.

[26] J. K. Iliffe and J. G. Jodeit. A dynamic storage allocation scheme. *The Computer Journal*, 5(3):200–209, November 1962.

[27] A. K. Jones and W. A. Wulf. Towards the design of secure systems. *Software: Practice and Experience*, 5(4):321–336, October 1975.

[28] Anita K. Jones. *Protection in Programmed Systems*. PhD thesis, Carnegie-Mellon University, 1973.

[29] Anita K. Jones and Barbara Liskov. A language extension for controlling access to shared data. *IEEE Transactions on Software Engineering*, SE-2(4):277–285, December 1976.

[30] B. W. Lampson. Dynamic protection structures. In *Proceedings of the Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 27–38, New York, NY, USA, November 1969. ACM.

[31] Butler W. Lampson. Protection. In *Proceedings 5th Princeton Conference on Information Sciences and Systems*, page 437, 1971. Reprinted in *ACM Operating Systems Review 8*, 1 (January 1974), page 18.

[32] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984. Available at http://www.cs.washington.edu/homes/levy/capabook/.

[33] Theodore A. Linden. Operating system structures to support security and reliable software. *ACM Computing Surveys*, 8:409–445, December 1976.

[34] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24:455–464, July 1977.

[35] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20:564–576, August 1977.

[36] David D. Redell. *Naming and Protection in Extendible Operating Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[37] Robbert van Renesse. Personal communication.

[38] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17:365–375, July 1974.

[39] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Cambridge, Massachusetts, May 1986.

[40] Warren Toomey. The strange birth and long live of Unix. *IEEE Spectrum*, 48(12):34–37, December 2011.

[41] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society, May 2000.

[42] Dan Wallach. Personal communication.

[43] Dan S. Wallach, Jim A. Roskind, and Edward W. Felten. Flexible, extensible Java security using digital signatures. In R. N. Wright and P. G. Neumann, editors, *Network Threats*, volume 38 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–74, New Brunswick, New Jersey, December 1996. American Mathematical Society.

[44] D.S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63. IEEE Computer Society, May 1998.

[45] K. G. Walter, W. F. Ogden, W. C. Rounds, F. T. Bradshaw, S. R. Ames, and D. G. Shumway. Primitive models for computer security. Interim Technical Report ESD–TR–4-117, Case Western Reserve University, 1974. NTIS AD-778 467.

[46] W. Wulf, E. Cohen, W. Corwin., A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, June 1974.