Part IV
Access Control

Confidentiality and integrity are often enforced using a form of authorization known as *access control*.

- Predefined *operations* are assumed to be 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 is achieved by restricting whether a given principal is authorized to execute operations that reveal information; integrity is achieved analogously by restricting the execution of operations that perform updates.

An *access control policy* specifies whether a *subject* may perform operations associated with a given *object*, including operations to change the policy. Subjects are entities to which execution can be attributed—users, processes, threads, or even procedure activations. And 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 operations typically are associated with distinct privileges. So, for example, there would be a specific read privilege for each object O, and only a subject S that holds this privilige would be permitted to read O.

The Principle of Least Privilege is best served by having fine-grained subjects, 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.

Chapter 7

Discretionary Access Control

7.1 The DAC Model

In a *discretionary access control* (DAC) policy, the initial assignment and subsequent propagation of privileges associated with an object are controlled by the *owner* of that object and/or other subjects whose authority can be traced back to the owner. Simple DAC policies are often implemented for files by commercial operating systems. The users are subjects; a user who creates a file is the owner and specifies which other users are authorized to read, write, and/or execute the file.

The assignment of privileges by a DAC policy can be depicted using a table that has a row for each subject and a column for each object. The entry in the cell associated with a subject S and an object O lists privileges corresponding to those operations on O that are authorized when invoked by execution being attributed to S. 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		
subject	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 control matrix*.[1] However, the term "matrix" here is misleading, because row and column ordering in a matrix has significance but row and column ordering in Figure 7.1 does not. We assume no ordering on the subjects or on the objects. So an access control matrix just specifies an unordered set *Auth* of triples, where $\langle S, O, op\rangle \in Auth$ holds if and only if subject S holds privilege *op* for object O; we will call *Auth* an *authorization relation*.

Any DAC policy can easily be circumvented if subjects are permitted to make arbitrary changes to *Auth*. Yet as execution of a system proceeds, changes to *Auth* will invariably be needed. New objects must be accommodated, object reuse requires changing the set of prinicpals that are authorized to access an object, and trust relationships between principals evolve in response to events outside of the computing system. To characterize permitted changes to *Auth*, a DAC policy includes *commands*. Each command has a Boolean *precondition* and an *action* that alters the set of triples in *Auth*. If the command is invoked and the precondition holds, then the action is performed; if the precondition does not hold, then the command fails. Evaluation of the precondition and performing the action is assumed to be indivisible.[2]

As an example, here is a command that might be found on a system where subjects are 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 defined by *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

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

which adds to *Auth* a triple authorizing operation *op* on object O by user U'. So, the *addPriv* command is consistent with the defining characteristic for a DAC policy—the owner of object O is the subject 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 empowers granting any privilege. So a better command than *addPriv* would be:

[1] Some authors 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. The effect must nevertheless somehow be made to appear 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 Subjects: Protection Domains

The Principle of Least Privilege suggests that the set of operations a principal should be authorized to execute is not the same for all tasks. Users are thus too coarse-grained to serve us well as the subjects in DAC policies. This leads to employing *protection domains* as subjects. We associate a protection domain with each thread of control, and we allow transitions from one protection domain to another as execution of the thread proceeds. A user concurrently engaged in multiple tasks, each as a separate thread, would cause execution in several protection domains and, since different privileges are now being associated with the execution required for the different tasks, the Principal of Least Privilege can be instantiated with *Auth*.

For an efficient implementation, protection-domain transitions must be associated with events that a run-time environment can detect cheaply. Practical considerations also limit what state or execution history would be available for deciding which protection domain is being associated with some execution. 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 system mode). But few processors, for example, could efficiently support entry to a new protection domain that is triggered by branching to a specific instruction or by storing to an arbitrary memory location.

When an operating system provides the support for protection domains, certain system calls are identified with protection-domain transitions. System calls for invoking a program or changing from user mode to system 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/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/pgm* would hold only the minimum privileges needed for *pgm* to execute for U.

Figure 7.2 reformulates the access control matrix of Figure 7.1 in terms of subjects 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` can be written by user `fbs` only

	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

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).

A given protection domain might or might not be appropriate for execution in support of 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 specify such restrictions by defining an `enter` (`e`) 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.

7.1.2 Amplification and Attenuation

We have so far not constrained how privileges change when a transition occurs between protection domains. In practice, though, the set of privileges before and after a transition are likely to be related.

> **Attenuation of Privilege.** Suppose execution in a protection domain D initiates a subtask, and that subtask 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. □

[4] An alternative to having protection domains be objects is having code segments (independent of user) be objects. With this alternative, protection domains from which a user U is allowed to next start execution of code segment *pgm* are granted 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

Amplification of Privilege. Suppose execution in a protection domain D' implements an operation on some object O, as a service to execution in protection domain D. Then D' should grant privileges for O that D 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 play a role in supporting the Principle of Least Privilege. Moreover, amplification of privilege also is key for supporting *data abstraction*, where users of an object are deliberately kept ignorant of how that object is implemented.

The Confused Deputy. Amplification of privilege brings the risk of a *confused deputy* attack. Here, one subject S requests execution by another subject S' in a way that causes S' to abuse privileges it holds but S' does not hold.

Consider a server that processes client requests, where each request names an input file and an output file. In processing a request, the server reads the named input file, computes results, writes these results to the named output file, and records billing information in file `charges.txt`. Further, suppose the server holds a `write` privilege for `charges.txt`, and the server receives as part of each request from a client C the `read` privilege C holds for the named input file and `write` privilege C holds for the named output file.

We might expect that the processing of client requests cannot corrupt file `charges.txt`, because clients lack `write` privileges for this file. But that expectation is naive. A client naming `charges.txt` as the output file for a request would cause the server to corrupt the billing information stored in `charges.txt`, since the server does hold a `write` privilege for `charges.txt` (although this was

not received from the client). What happened was the server functioned as a deputy to the client, and the deputy became "confused" by the client's request.

One obvious defense is for the server to validate each client request, by checking that the client holds appropriate privileges for every file named in the request. This defense, however, requires programmers to include checks in every program that might invoke operations on objects provided by another. Many programmers would regard adding those checks as onerous and not bother. So the defense is unlikely to be deployed.

A more elegant defense is to combine naming and authorization. Instead of names for objects (like files), programs use unforgeable *bundles* comprising the name for an object along with privileges for that object. Bundles are assumed to provide the sole means by which programs name, hence access, objects. In the example above, each client request would convey two bundles—one for the input file (incorporating a `read` privilege) and one for the output file (incorporating a `write` privilege)—and the server would use these client-supplied bundles for reading the input file and writing to the output 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—attempts by the server to perform write operations to `charges.txt` using that client-supplied bundle would fail. The confused deputy is no longer duped into writing the wrong content into `charges.txt`.

7.1.3 *Undecidability of Privilege Propagation

A central concern when designing the commands for changing authorization relation *Auth* is the assurance that certain priviliges cannot be granted to particular subjects. We formalize this concern using a predicate.

> **Privilige Propagation.** $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ is *true* if and only if subject S eventually can be granted privilege op to object O by starting from authorization relation $Auth$ and allowing subjects not in S' to execute commands from set $\mathcal{C}$. □

Typically, S' would be the set of subjects that are both trusted and authorized to grant $\langle S, O, op\rangle$, because finding that $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ holds then would indicate unauthorized propogation of privilege.

To determine whether $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ holds we might write a program that computes the value of $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ by generating all authorization relations that subjects not in S' executing sequences of commands from $\mathcal{C}$ can derive from $Auth$. However, if commands create new subjects 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 about other approaches for evaluating $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$? We prove below that any program whose execution always terminates with the value of $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ could be used to solve the halting

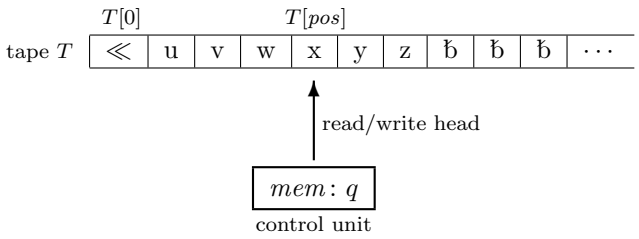


Figure 7.4: A Turing Machine

problem for Turing machines. The latter is an undecidable problem and, therefore, so is the former. This means that there cannot exist a single program that evaluates $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ for any possible arguments S', $\mathcal{C}$, $Auth$, and $\langle S, O, op\rangle$.

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 a 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 some 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 $\Gamma \cup \{\ll, ƀ\}$ are disjoint. The control unit performs execution steps, as specified by a transition function δ. 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, γ' 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 we consider.

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

- When $pos = 0$ holds, no execution step replaces the $\ll$ symbol found in $T[0]$ nor attempts to move the read/write head left (i.e., off the left end of the tape).

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.

The full specification of transition function δ for a Turing machine would specify the value of $\delta(q, \gamma)$ for each possible $q \in Q - \{q_F\}$ and $\gamma \in \Gamma \cup \{\ll, \mathfrak{b}\}$. One way to represent this full specification is with a *transtion table* that has $|Q - \{q_F\}|$ rows and $|\Gamma \cup \{\mathfrak{b}, \mathfrak{b}\}|$ columns; the cell in the row for q and the column for γ contains the value of $\delta(q, \gamma)$. This tablular representation has finite size because Q and Γ 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 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 intial and configruation $\langle T_i, pos_n, mem_n \rangle$ 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 $M(inp) = \uparrow$ if M does not halt on input *inp*.

[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 $i \neq pos$.

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 specification is provided to M as an input. The *halting problem* is concerned with constructing a Turing machine M_{HP} that satisfies the following specification.

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

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:

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

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 $M(inp)$ does halt.

- *Case 2:* $M(inp) = \uparrow$. 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.

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.

Privilige Propagation and Undecidability. We now establish that determining whether $CanGrant(S', \mathcal{C}, Auth, \langle S, O, op\rangle)$ holds is an undecidable problem. We do this by showing that a program P can be used to solve the halting problem if invoking $P(\mathcal{C})$ can determine whether there exists a finite sequence of commands from $\mathcal{C}$ that eventually causes $\langle S, O, op\rangle \in Auth$ 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 subjects. And we simulate M's execution steps using a set $\mathcal{C}$ of commands, so that every execution of M is simulated by a sequence of commands. Moreover, the commands are defined in such a way that $\langle S_0, S_0, q_F\rangle \in Auth$ holds if and only if M halts. Consequently, a program P that determines whether $CanGrant(\emptyset, \mathcal{C}, Auth, \langle S_0, S_0, q_F\rangle)$ holds would constitute a solution to the halting problem.

	object						
subject	S_0	S_1	S_2	S_3	S_4	S_5	S_6
S_0	$\ll$	nxt					
S_1		u	nxt				
S_2			v	nxt			
S_3				w, q	nxt		
S_4					x	nxt	
S_5						y,	nxt
S_6							z, end

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

Figure 7.5 depicts one scheme that *Auth* could use to represent Turing machine configuration $\langle T, 3, \text{q}\rangle$ for a tape T storing: $\ll$ u v w x y z ƀ ƀ ƀ ... . The representation is based on the following.

- $\langle S_i, S_i, x_i\rangle \in Auth$ signifies that symbol $x_i \in \Gamma \cup \{\ll, \text{ƀ}\}$ is stored by tape square $T[i]$.
- $\langle S_i, S_i, q\rangle \in Auth$ signifies that $mem = q$ and $pos = i$ hold, where $q \in Q$ and $0 \leq i$.
- The sequencing on tape squares is encoded with privileges: nxt, end, $\ll$.
  - $\langle S, S, \ll\rangle \in Auth$ implies that subject S is used for representing $T[0]$.
  - $\langle S, S, \text{end}\rangle \in Auth$ implies subject S is used for representing the last non-blank tape square of T.
  - $\langle S, S', \text{nxt}\rangle \in Auth$ implies that the tape square being represented using subject S' immediately follows the tape square being represented using subject S.

The need for $\langle S, S', \text{nxt}\rangle$ privileges might at first seem puzzling. It arises because tape squares for a Turing machine are ordered, we associate tape squares with subjects, but the DAC model does not assume an ordering on subjects.[6] By introducing an ordering relation $\prec$ on the subjects, we induce an ordering on tape squares. And we encode this ordering on subjects by representing $S \prec S'$ using the privilege $\langle S, S', \text{nxt}\rangle \in Auth$.

The set of commands to simulate execution steps for a Turing machine are derived from its transtion function δ. The precondition of each describes a Turing machine configuration; the action updates *Auth* in accordance with the changes to the configuration prescribed by δ. For example, the Turing machine execution step specified by $\delta(q, \gamma) = \langle q'\gamma', 1\rangle$ is simulated by commands C_{R} and $C_{\text{R-end}}$ in Figure 7.6.

[6] You might expect the indices on subjects would suffice to define an ordering on subjects But the ordering defined by having subject names differentiated only by integer subscripts is illusory, being an artifact of notational convenience.

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

$C_{\text{R-end}}(s)$: **command**
pre: $\langle s, s, q\rangle \in Auth \ \wedge \ \langle s, s, \gamma\rangle \in Auth \ \wedge \ \langle s, s, \mathrm{end}\rangle \in Auth$
action: $Auth := Auth - \{\langle s, s, q\rangle, \langle s, s, \gamma\rangle, \langle s, s, \mathrm{end}\rangle\}$;
$s' :=$ newSubject();
$Auth := Auth \cup \{\langle s, s, \gamma'\rangle, \langle s', s', q'\rangle, \langle s', s', \mathrm{end}\rangle, \langle s, s', \mathrm{nxt}\rangle\}$;

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

$C_{\mathrm{HALT}}(s)$: **command**
pre: $\langle s, s, q_F\rangle \in Auth$
action: $Auth := Auth - \{\langle s, s, q_F\rangle\}$;
$Auth := Auth \cup \{\langle S_0, S_0, q_F\rangle$

Figure 7.6: Commands to Simulate a Turning Machine

- C_{R} handles the case where the read/write head position is not at the right-most tape square; this case is distinguished because the current tape square corresponds to some subject S for which there does exist another subject S' where $\langle S, S'\mathrm{nxt}\rangle \in Auth$ holds.
- $C_{\text{R-end}}$ handles the case where the read/write head position is at the right-most tape square; this case is distinquished by having the tape square correspond to a subject S where $\langle S, S, \mathrm{end}\rangle \in Auth$ holds.

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

Finally, C_{HALT} of Figure 7.6 is included in $\mathcal{C}$ so that $\langle S_0, S_0, q_F\rangle$ is granted if ever privilege q_F is granted to any subject. This command does not simulate a Turing machine's execution step, but now $\mathit{CanGrant}(\emptyset, \mathcal{C}, Auth, \langle S_0, S_0, q_F\rangle)$ holds if and only if the Turing machine being simulated halts. Thus, a program that evaluates $\mathit{CanGrant}(\emptyset, \mathcal{C}, Auth, \langle S_0, S_0, q_F\rangle)$ 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 $\mathit{CanGrant}(\emptyset, \mathcal{C}, Auth, \langle S_0, S_0, q_F\rangle)$.

This undecidability result does not imply that a program cannot exist to compute the value of $\mathit{CanGrant}(\emptyset, \mathcal{C}, Auth, \langle S, O, op\rangle)$ for a specific command set $\mathcal{C}$. Such programs do exist, and they even exist for command sets that correspond to DAC policies enforced by actual systems. The undecidability

result nevertheless is important, because it illustrates how reduction techniques from theoretical computer science can be used in reasoning about classes of authorization mechanisms.

7.1.4 Implementation of DAC

The heart of any implementation of DAC is a scheme for representing authorization relation *Auth*. That scheme must provide the means to

- evaluate whether $\langle S, O, op\rangle \in Auth$ holds and, therefore, subject S holds the privileges needed to perform an operation *op* on some object O,
- change *Auth* in accordance with commands the DAC policy defines, and
- associate a protection domain with each thread of control, providing for transitions between protection domains as execution proceeds.

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

The obvious scheme for representing *Auth* is to employ a 2-dimensional array-like data structure resembling an access control matrix. However, access control matrices are likely to be sparse, because the typical subject holds 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 control matrix. We explore these in what follows.

7.2 Access Control Lists

An *access control list* for an object O is a list

$$\langle S_1, Privs_1\rangle\ \langle S_2, Privs_2\rangle\ \ldots\ \langle S_n, Privs_2\rangle$$

of *ACL-entries*. Each ACL-entry $\langle S_i, Privs_i\rangle$ is a pair, where S_i is a subject, $Privs_i$ is a non-empty set of privileges, and $op \in Privs_i$ holds if and only if $\langle S_i, O, op\rangle \in Auth$ holds. Thus, an access control list encodes the non-empty cells in some column of the access control matrix. 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 update; they are also expensive for enforcement mechanisms to scan when authorizing access requests. Therefore, various representations have been proposed for shortening the number of ACL-entries in an access control list and/or 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 group memberships. Students who enroll in a course, for example, are given access to that semester's class notes and assignments because they members of the class.

Group memberships change over time. If membership in a group confers privileges for many objects, then adding or deleting a member requires updating a separate access control list for each of those many objects. That will be tiresome and error-prone. Moreover, updating each individual access control list can be subtle. Suppose, for example, user U is in group G, and membership in G confers privilege *op* for object O. If U is being dropped from G 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 confers *op* for O on its members, in which case U being dropped from G should not cause U to lose *op* for O.

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

Groups in Access Control Lists.

- A *group declaration* associates a *group name* with a set of subjects. Membership in the set is specified either by enumerating its elements or by giving a predicate that all subjects in the set must satisfy.[7]
- 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 subjects S that are members of G. □

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

Permission and Prohibition. That a given subject does not hold a specified privilege is sometimes what's important. Yet in order to conclude that S does not hold *op* for an object O, we would have to: (i) enumerate all subjects granted *op* by the access control list for O, (ii) check that S is not among them, and (iii) presume the Principle of Failsafe Defaults. This is tedious. So some systems allow a *prohibition* $\overline{op}$ to appear in an ACL-entry; as the term suggests, holding $\overline{op}$ specifies that operation *op* is prohibited. The use of prohibitions does raise a question about the meaning of an access control list that contains both a privilige *op* and a conflicting prohibition $\overline{op}$. Different systems resolve this conflict in different ways, but it is not uncommon to decide an outcome according to the relative order of the conflicting ACL-entries.

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

7.2.2 Pragmatics

Designers and implementors of access control mechanisms are pimarily concerned with three things: flexibility, understandability, and 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 eschew complexity, because it introduces the risk that people will be unable or disinclined to write or understand policies; it also tends to erode assurance in an access control mechanism's implementation. And higher run-time costs are problematic because they lead to favoring access-control policies that involve less checking in preference to access-control policies that enforce what is needed.

Subjects. Flexibility for expressing a policy by using access control lists will depend, in part, on what can be a subject. That set of subjects, in turn, is constrained by the availability of efficient means for attributing (authenticating) accesses, since the name of a subject making a request is what's needed for checking an access control list.

Operating systems typically have cheap mechanisms for authenticating users and processes. Some language run-time environments do even better and can attribute execution of each statement to the current chain of nested procedure invocations—for example, allowing a protection domain $U/pgm_1/pgm_2/pgm_3$ for a subject that corresponds to execution of pgm_3 invoked by a call within pgm_2, itself invoked by a call from pgm_1, running on behalf of user U.

Independent of what constitutes a subject, care should be exercised in recycling subject names. Otherwise, some future incarnation of a subject name could inadvertently receive privileges held by to a past incarnation. One solution is simply not to reuse subject names, but this (i) requires saving enough state to ensure no future subject name duplicates a past name and (ii) constrains the choice of subject names, potentially making policies harder to understand. The more widely-adopted solution is, as part of deleting a subject from the system, to delete that subject'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 requires a reference monitor. The reference monitor intercepts every access to the object, and that restricts possible choices for objects. Some implementations for reference monitors require that all accesses cause traps; other implementations require that checks be in-lined into all code that contains accesses. 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 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 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. First, system calls are then the only way to access an object, and a reference monitor is easily 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 start from a debatable premise that checking shorter access control lists is faster.[8] They advocate employing terse representations for ACL-entries. One approach is to employ patterns and wild-card symbols for specifying names of subjects 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 later is shorter. Terse representations are often harder for humans to understand, though, so there is often a trade-off: human understandability versus computation time for enforcement. And cheap enforcement of policies that nobody understands is arguably a dubious goal.

7.3 Capabilities

Abstractly, a *capability* is a pair $\langle O, Privs \rangle$; any subject that *holds* capability $\langle O, Privs \rangle$ is granted set of privileges *Privs* for operations on object O. Each capability that a subject S holds thus corresponds to some non-empty cell in S's row of the access control matrix. And an authorization relation *Auth* is faithfully represented when, for all subjects S and objects O, S holds capability $\langle O, Privs \rangle$ if and only if $\langle S, O, Privs \rangle \in Auth$ is satisfied.

Compliance with *Auth* requires that

- each subject S invoking an operation *op* on an object O must hold a capability $\langle O, Privs \rangle$ where $op \in Privs$ is satisfied, and
- no subject is able to hold a counterfeit or corrupted capability.

Notice that capabilities could provide the sole means for subjects to identify and access objects, supplanting ordinary names and addresses. This is called *capability-based addressing*. It is what we used above for solving the confused deputy problem; bundles are really just capabilities.

When capabilities are being used, we must somehow prevent unauthorized creation of new capabilities and prevent unauthorized changes to existing capabilities. A variety of schemes have been developed for enforcing this *capability authenticity*; the classics are outlined below. Changes to *Auth* are then supported by providing specific routines that enable an authorized subject S to

- create a new object and, in so doing, receive a capability for that object,

[8] 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.

- transfer to other subjects any capabilities S holds, with attenuation and/or amplification of privilege applied if specified, and
- revoke capabilities that were derived from capabilities S holds.

Consistent with the definition of DAC, privileges are controlled by the owner or by subjects whose authority can be traced to the owner, because all capabilities for an object O are derived from one first received by the subject that created (owns) O.

Object Names. Unless the object name O in a given capability $\langle O, Privs \rangle$ always refers to the same unique object—independent of what subject 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 that designates an object Obj is later recycled to designate object Obj', then by holding $\langle O, Privs \rangle$ long enough, a subject eventually gets privileges to access Obj' (even though access to Obj was what had been authorized).[9] And if an object name O designates object Obj in one subject's address space but Obj' in another's, then transferring $\langle O, Privs \rangle$ unwittingly gives the recipient privileges to Obj' (even though the sender transferred a capability for Obj).

One approach to naming objects is to use the virtual address where an object is stored as the name of that object. This works because 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, and different subjects use the same name only if they are referring to the same object.

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. A software run-time environment can be used. It maintains a directory that maps object names to memory addresses. Subjects are expected to invoke a run-time routine (passing an object name and an operation name) to perform each operation on an object; this 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 symbol table entry—rather than the name field in every capability for that object—suffices. Also, either a real or virtual address can be saved in the symbol table entry for an object.

Capability Archives. Often, a main memory representation is used for capabilities held by executing subjects and an *archive* is kept on secondary memory to store capabilities for objects (e.g., files) that persist even when subjects holding those capabilities are not executing. Capabilities for accessing the archive

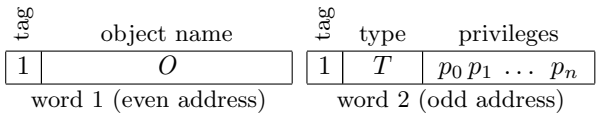
[9] This problem can be avoided if the run-time environment's object-deletion routine also eliminates or revokes all capabilities naming the object. But, as will become clear, some implementations of capabilities are not amenable to the bookkeeping necessary for this.

are held by a single, special, trusted subject. This trusted subject always executes (it might be part of the operating system), and it retrieves appropriate capabilities to populate main memory whenever execution of a new subject is initiated. The retrieval function often allows human-readable names as input, so the archive resembles a conventional 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 employ 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 bit string $p_0\, p_1\, \ldots\, p_n$. The type defines how each of the bits in $p_0\, p_1\, \ldots\, p_n$ is interprted.. 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 the privileges field 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 also 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[10]

`cap_copy` @*src*, @*dest*

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 subject 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 impose restrictions. For example, if *cap* is a capability for object O and *op* is an integer then execution of

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

`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) stores *retCap* someplace accessible to execution by *O*, (iv) pushes @*retCap* onto the run-time stack, and (v) 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 `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 system mode can suffice for supporting most other functionality involving capabilities, 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 system mode would likely be provided for the following functionality.

- New objects and their capabilities are created by invoking 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 are propagated from one subject to another that do not share an address space (so `cap_copy` cannot be used) by invoking system routines to send and receive capabilities. The operating system presumably has access to every subject's address space and can execute the `cap_copy` instructions needed.
- The functionality of the `cap_copy` instruction is extended to perform attenuation and amplification, where desired, by having the source and destination subjects 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

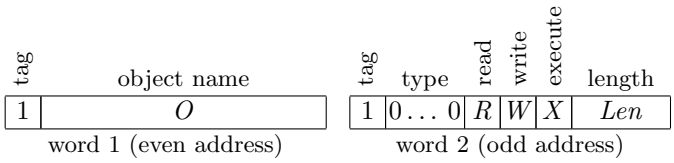


Figure 7.7: Example Format of Capability for a Memory Segment

> O with some type T and a set $Privs_O$ of privileges, and the second capability gives type T as its name[11] (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.

As part of this scheme, capabilities can be used to ensure that appropriate privileges are held for each and every memory access that a subject makes. Figure 7.7 suggests a format for such a memory segment[12] capability; it is an instance of the capability format given above. The type (0...0) signifies that the capability is for a memory segment; O gives the starting address of the memory segment; privileges bits p_0, p_1, and p_2 (labeled R, W, and X in Figure 7.7) specify privileges for operations read, write, and execute; and suffix $p_3\, p_4\, \ldots\, p_n$ of the privileges specifies the segment length.

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

- A set of *segment capability registers* is provided to store capabilities for memory segments.[13] And a memory access is allowed to proceed only if the address (i) names 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 privilege bit is set in that capability. An *access-fault* trap occurs, otherwise.

- The processor provides a system-mode instruction

  `load_scr` *scr*, @*cap*

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

The operating system then provides routines that allow execution to *map* and *unmap* memory segments. The set of mapped memory segments at any

[11] When typed objects are not supported, then the hardware might require that the two input capabilities name the same object.

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

[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 accessing a significantly larges number of segments.

given time defines a protection domain by establishing what memory can be addressed, hence what set of capabilities the executing subject holds. In some systems, the set of mapped memory segments for a given subject is partitioned into subsets: memory accessible to every subject, memory accessible only to this subject throughout its execution, and memory accessible because some operation on a given object is being executed.

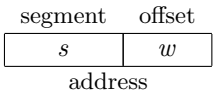
An operating system might support having a larger number of memory segments be mapped at a given time than there are segment capability registers. To accomplish this, system software multiplexes the segment capability registers in much the same way that a small set of page frames is multiplexed to create a much larger virtual memory. Specifically, the operating system maintains a set *MappedSegs* of the capabilities for memory segments that currently are considered 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.

7.3.2 Capabilities in Protected Address-Spaces

Modern processor hardware invariably enforces some form of memory protection, if only to protect operating system integrity by isolating its code and data from user programs. This is achieved by allowing memory to be partitioned into one or more regions and, for each, enforcing access restrictions. Although coarse-grained in comparison to tagged memory, even this simple form of memory protection suffices for implementing capability authenticity.

The basic strategy is to segregate capabilities and store them in memory regions that cannot be written by execution in user mode. Operating system routines, which execute in system mode, are granted 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 subjects, and the invocation and return from operations on objects (with attenuation and amplification).

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



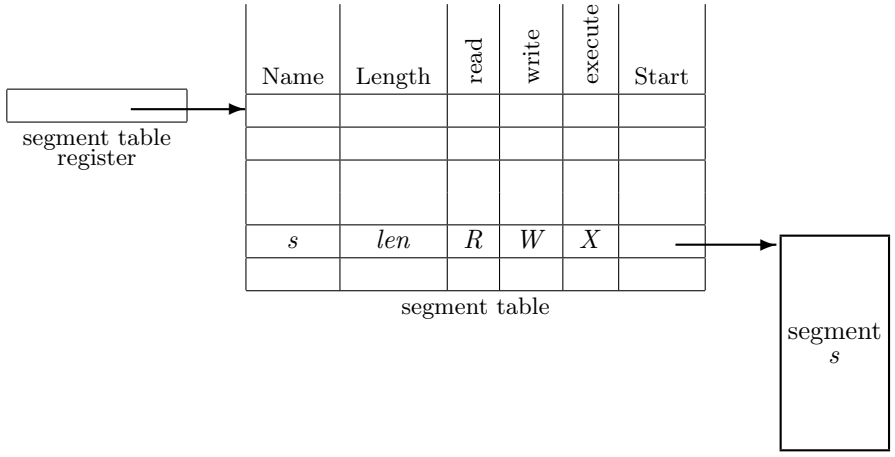


Figure 7.8: Addressing with a Segment Table

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 should now come as no surprise. The access bits in segment descriptors allow us to construct one or more virtual address spaces for which writes to those segments are prohibited. So, by compelling 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. Subjects are implemented as user-mode processes, with the set of capabilities held by a subject 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

[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.

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.

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 tests 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) store *retCap* in a segment of capabilities readable when executing in the invoked operation, (iii) push @*retCap* onto the run-time stack, (iv) load the segment table register with the address of the new segment table, and (v) transfer control to the code for *op* in *obj*.

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

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

Capabilities Stored in Kernel Memory. Even if a segmented virtual memory is not supported by the processor, some form of memory isolation usually will be. Hardware support for a single memory region that cannot be corrupted by user-mode execution is typical, because this suffices for protecting an operating system's integrity. To implement capabilities here:

- The memory region stores all capabilities and identifies which subject(s) hold each capability.
- Each subject is implemented by a user-mode process.
- Operating system routines provide the sole means by which user-mode execution can read or change the memory region.

Various schemes have been employed for identifying what capabilities a process holds. Invariably, these schemes associate a table, herein called a *c-list* (for "capability list"), with each process. Each entry in a c-list stores a single capability and is identified by its location in the table. References to capabilities give this location, so they are indirect and relative to the c-list implicitly associated with the process making the reference. Some schemes have separate c-lists for every process; in others, c-lists are shared by multiple 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 provided to pass capabilities from one process to another, whether or not those processes share a c-list. Specifically, a process P might invoke send, giving a destination process P' and locations for some capabilities that would then be buffered at P' for receipt; by invoking receive, P' would move any buffered capabilities to its c-list and obtain the locations where they are stored.[16]

The operating system would also provide routines for creating and managing c-lists. The routine for new process creation would presumably take an argument specifying whether the new process will share the caller's c-list or have a new one (and, for a new c-list, what subset of the caller's capabilities and privileges to include). And invoke/return operations might be implemented as operating system routines, in order to allow execution of an invoked routine to use a new c-list. This new c-list would be populated from the caller's c-list with copies of capabilities the caller indicates in arguments as well as capabilities obtained though attenuation and amplification of capabilities the caller indicates.

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—limit the applicability of

[16] An operating system call to examine the capability stored for each index would then be used by P' to determine what the new capabilities authorize.

this approach, 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 something is a k-signed bit string.

We then implement capabilities as signed bit strings, because the Unforgeability and Tamper Resistance properties imply capability authenticity[17] and the Validity Checking property gives a way for system components to ascertain whether a bit string purporting to be a capability is authentic.

Specifically, for *cap* a bit string that gives the name, type, and privileges for an object O, the k-signed bit string $\mathcal{S}_k(cap)$ serves as a capability that grants its holders the specified privileges for O provided:

(i) private key k is known only by component(s) authorized to generate capabilities for O, and

(ii) corresponding public key K is available to any principal needing to check the authenticity of a capability $\mathcal{S}_k(cap)$.

Notice that code to generate capabilities or to check capability authenticity need not execute in system mode—user mode works just fine for performing the necessary cryptographic calculations. Confidentiality of private keys used in such user-mode computations is required, but this confidentiality can be implemented through memory isolation typically provided by an operating system. And digital certificates signed by some well-known trusted authority are an obvious means for making public keys available for Validity Checking.

If, as usual, each component has a distinct private key, then capabilities generated by different components require different public keys to validate them. This built-in means of attribution allows capabilities to be ignored when they have been generated by components lacking the authority. In some systems, only one component, such as the operating system, is authorized to create capabilities. So a single public key valdiates the authenticity of all capabilities. In

[17] 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.

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 type, to select that public key. Capabilities for all objects of each given type T might, for example, be validated by a corresponding well known public key K_T.

Capabilities implemented as signed bit strings are easy to transfer between subjects, protection domains, and even between computers. It is just a matter of copying the bits (assuming infrastructure is in place to disseminate public keys that can be trusted 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 from scratch a capability with modified privileges or to modify the privileges in $\mathcal{S}_k(cap)$ directly. But sharing private keys is risky, and cryptographic computations create performance bottlenecks. So system designers often do not provide support for amplification and attenuation when capabilities are being implemented cryptographically. The Principle of Least Privilege now becomes harder to support, leading to applications that are not as secure as they could be.

Three costs are noteworthy when a digital signature scheme is used to support capability authenticity: the amount of space required to store a k-signed bit string, the amount of time required to generate one, and the amount of time required for validity checking. To facilitate comparisons with schemes that use 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 k-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 a k-signed bit string $\mathcal{S}_k(b)$ using a 2048-bit RSA key takes orders of magnitude longer than the time required for a kernel call; validity checking takes somewhat less time but the 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 implemented 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 the operating systems at those computers to communicate. The integrity and authenticity of that communiation must be ensured. Digital signatures are the usual defense here, but signature generation and validity checking now become part of the cost of transmmitting a capability between computers.

In addition, cryptographic protection allows the authenticity of a capability to be checked locally in user mode, which can be considerably cheaper than querying the operating system on the (possibly remote) computer that generated that capability. Also, transferring a capability between two principals executing on the same computer does not require the operating system to serve as an intermediary. So when cryptographic protection is used, the execution times for capability authenticity checking or transfer operations need not incur the expense of inter-processor communication.

7.3.4 Capabilities Protected by Type Safety

Programs written in type-safe programming languages declare a *type* for each variable. Execution is then restricted accordingly. Since support for capabilities also involves enforcing restrictions on execution, 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 type safety implies (i) possession of a suitable capability is necessary for executing each operation defined on an object, and (ii) capability authenticity is enforced.

Type Safe Execution. 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 exececution, only operations in ops_T are invoked for values in $vals_T$. □

And 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. Thus, $T \preceq T'$ 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 always complies 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.** Assignment statement $v := Expr$ exhibits type-safe execution provided $type(v) \preceq type(Expr)$ holds. □

The condition is statically checkable because the declaration of v provides $type(v)$, and the declarations of the variables and operators in *Expr* suffice for deducing a type for the value *Expr* produces.

Next, consider an *invocation statement*

$$\textbf{call } obj.m(Expr_1, \ldots, Expr_i, \ldots, Expr_N) \tag{7.3}$$

Here, *obj* is a program variable that designates some object and m names an operation. The definition for $type(obj)$ presumably contains declarations for all methods supported on instances of $type(obj)$, where a declaration for a method m would have the following form.

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

Execution of invocation statement (7.3) 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 (7.3), we must be concerned with Type-Safe Assignment Restriction and with Type-Safe Invocation Restriction. Assume that checking has established $body_m$ will exhibit 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 (7.3) is then implied by Type-Safe Assignment Restriction for $p_i := Expr_i$ where $1 \leq i \leq N$ which, according to Type-Safe Assignment, requires $T_i \preceq type(Expr_i)$ to hold 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
>
> $$\textbf{call } obj.m(Expr_1,\ Expr_2,\ \ldots,\ Expr_N)$$
>
> for a method
>
> $$m\text{: } \textbf{method}(p_1\text{:}T_1,\ \ldots,\ p_i\text{:}T_i,\ \ldots,\ p_N\text{:}T_N)\ \ body_m\ \ \textbf{end}$$
>
> exhibits type-safe execution 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.

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

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

defines a set of values identical to the values of type T and defines a set of operations that contains only those operations both supported by T and appearing in list m_1, m_2, ..., m_N of operations the capability type authorizes:

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

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 better understand type-safety for capabilities, consider capabilities *cap1* and *cap2* for objects of type *dbase*, which supports three operations: read(x, *val*), update(x, *val*), and reset().

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

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

An 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 $ops_{type(cap1)} = \{\text{read}, \text{update}\}$). But **call** *cap2*.update(...) cannot satisfy Type-Safe Invocation, since update $\in ops_{type(cap2)}$ does not hold; and this is exactly what we should desire—*cap2* does not convey priviliges for operation update and type-safety is prohibiting the attempt to invoke update through *cap2*.

Assignment statement *cap2* := *cap1* 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 privilige. Assignment statement $cap1 := cap2$ does not satisfy Type-Safe Assignment since *cap2* $\preceq$ *cap1* does not hold. This is desirable, because it means that program fragment

cap1 := *cap2*; **call** *cap1*.update(...)

is not type safe, as we should want—allowing the fragment to execute would enable a subject holding a capability (*cap2*) for *db2* that authorizes only read

operations to invoke an update operation on *db2* (since invocation statement **call** *cap1*.update(...) is type safe).

So knowing that $obj \neq \perp$ will hold before a type-safe invocation statement is executed suffices to conclude that a subject possesses not just any capability but possesses a suitable capability to proceed with the invocation. No run-time checks concerning priviliges need to be performed; that aspect of capability semantics is enforced through type-checking analysis of program text before program execution starts. Moreover, when a program-flow analysis determines that $obj \neq \perp$ is necessarily true prior to reaching a given invocation statement, then no run-time check at all is required for an invocation.

The other defining characteristic for an implementation of capabilities is that subjects are prevented from altering or forging capabilities—capability authenticity. Type-Safe Assignment is what prevents a subject from altering a capability to increase priviliges. In particular, Type-Safe Assignment ensures that an assignment statement never stores a capability into a variable allowing more operations on some object than were allowed by the variable originally storing the capability.

Finally, we prevent subjects from forging capabilities by restricting the class of expressions whose evaluation produce values with capability types. This class of *capability expressions* must, at a minimum, include (i) expressions that manufacture a capability whenever a new object is created and (ii) expressions that materialize a capability already held by the subject evaluating the expression. 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 methods for that new object. And, to satisfy (ii), we define all variables declared with capability types to be capability expressions, thereby allowing existing capabilities to be retrieved for copying (perhaps between subjects) and to be used for invocations.

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

Type-Safe Assignment allows attenuation (as was noted above), but it does not allow amplification. However, a form of amplification is intrinsic in the usual scope rule for variables declared within an object. This scope rule stipulates that such variables may be named within that object's methods but not outside. For example, Figure 7.9 gives a definition for type *dbase*. It declares a single variable *dbCntnt*, which stores the state of a *dbase* instance; the scope 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 methods, because the body of a method can directly access the object's variables. Moreover, if variables with capability types are declared within an

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

Figure 7.9: Definition of type dbase

object, then only by executing a method can these capabilities be exercised. So, a subject executing a method has amplified priviliges relative to what it had when executing outside the method.

7.3.5 Revocation of Capabilities

Revoking a subject's authorization can be subtle when *Auth* is implemented using capabilities. First, deleting $\langle S, O, Privs\rangle$ from *Auth* requires finding and invalidating not just one copy but all copies of capability $\langle O, Privs\rangle$ that S holds. Second, S could have passed copies of $\langle O, Privs\rangle$ to other subjects, and they in turn could have passed those copies still further. The rationale for revoking S's authorization to O might well apply to these other subjects, so we would need to find and invalidate their copies of $\langle O, Privs\rangle$ too. In sum, support for revocation requires an efficient means to *invalidate* all copies of a given capability held by some set of subjects whose elements might not be easy to enumerate.

Brute-Force Search. Brute-force searching for copies of a capability and deleting them is one obvious approach to revocation. However, brute-force search is feasible only for relatively small storage regions. Capabilities implemented by hardware-implemented tagged memory or by cryptographic protection can be stored anyplace in a subject's address space; brute-force search is infeasible here. But brute-force search is feasible when capabilities are implemented by protected address-spaces or by strong typing and declarations, because then all capabilities are stored in a small number of easily identified memory locations.

Revocation Tags. An alternative to finding and deleting invalidated capabilities is simply to block access attempts that use them. This is implemented by deploying a reference monitor not only to check capability authenticity but

also to check whether a capability has been invalidated.

One approach is to include a *revocation tag* in each capability. A capability is now a triple $\langle O, Privs, revTag\rangle$. And, for each object O, we define a set $RevTags_O$ to store those revocation tags appearing in invalidated capabilities for O.

- *Revocation.* A capability $\langle O, Privs, revTag\rangle$ is invalidated by invoking an operation that adds revocation key $revTag$ 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 $\langle O, Privs, revTag\rangle$ has been invalidated.

Capability authenticity is presumed to prevent subjects from changing the revocation tag in a capability; the operating system is presumed to protect the integrity of $RevTags_O$ and to provide the support for adding elements to $RevTags_O$ (but preventing other changes to $RevTags_O$).[18]

Capabilities that incorporate revocation tags can be used to support *selective revocation*. Here, operations are provided to delete various subsets of the capabilities that subjects hold for a given object. It suffices for there to be a capability *facsimile generation* operation, with a corresponding privilege (say) fg. The holder of a capability $\langle O, Privs, revTag\rangle$ with $fg \in Privs$ invokes facsimile generation to obtain a new capability $\langle O, Privs', revTag'\rangle$ for O, where $Privs' \subseteq Privs'$ holds and $revTag'$ is a fresh revocation tag.

Sets of subjects whose authorization for an object O might have to be revoked together are then given capabilities that share the same revocation tag. Notice, if some subject S passes a capability $\langle O, Privs, revTag\rangle$ to another subject, and unbeknown to S that capability is forwarded further, then all those copies of $\langle O, Privs, revTag\rangle$ also would be invalidated when $revTag \in RevTags_O$ holds. Revocation tags thus support selective revocation but only to the extent that (i) prior to disseminating capabilities to subjects we can anticipate what sets of capabilities should together be invalidated (because all capabilities in such a set must share a revocation tag), and (ii) these sets are non-intersecting (because a capability may contain only one revocation tag).

Capability Chains. Indirection is the basis for a second approach to blocking the use of invalidated capabilities. The idea is simple. We permit the object named in a capability to be another capability. Now chains of capabilities that lead to the capability for an object O can be constructed. And an access

[18] In the implementation just sketeched, $RevTags_O$ grows without bound. This could be problematic given finite memory. But an element $revTag$ can be removed from $RevTags_O$ once all capabilities containing that revocation key have been deleted. It is possible to ascertain that those capabilities have all been deleted, for example, when a subject that cannot share its capabilities with other subjects is terminated and all of its storage is reclaimed. In addition, $RevTags_O$ 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.

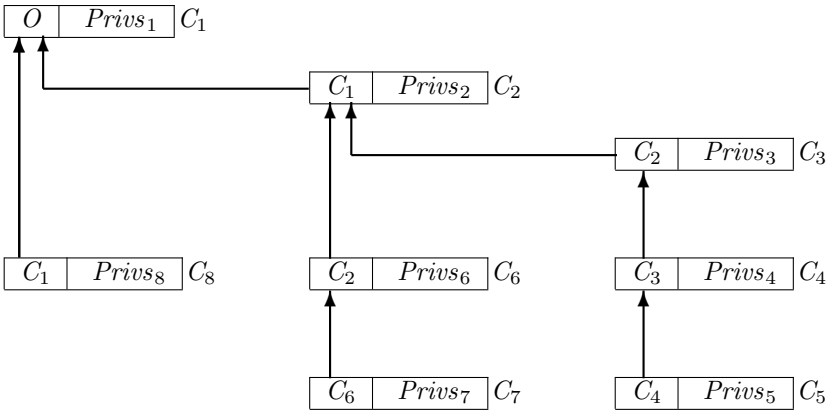


Figure 7.10: Chain of Capabilities

to O is permitted using a capability C if there is a chain of capabilities that starts from C, ends with a capability for O, and each capability in the chain satisfies capability authenticity as well as containing privileges that authorize the requested access to O.[19]

Figure 7.10 gives an example. Each of the capabilities there (including C_1) starts a chain that ends with a capability for object O. For an access using one of the capabilities C_i depicted in the Figure 7.10, a reference monitor would follow the chain from C_i to C_1 (the capability for O), checking authenticity for each capability it traverses and checking that the requested access is authorized by that capability.

Deleting all instances of a capability C that appears in a chain severs the chain. Afterwards, access attempts that required traversing C in order to reach some capability $\langle O, Privs \rangle$ no longer succeed. For example, if all copies of C_2 in Figure 7.10 are deleted, then access to O from capabilities C_3, C_4, C_5, C_6 or C_7 is no longer authorized; access to O from C_1 or C_8 is unaffected, though. The authorization for a subject S to delete a capability C might derive from S being authorized to update the memory that contains C. Or it might derive from S holding a capability $\langle C, Privs \rangle$ where *delete* $\in$ *Privs* holds and the *delete* privilege is required by system routines that delete objects.

Capability chains support richer forms of selective revocation than revocation tags do. Deleting (all copies of) a capability C invalidates a set of ca-

[19] There are other sensible interpretations for what operations are authorized by a chain of capabilities. We might, for example, require that the last capability in the chain authorize the operation but other capabilities traversed in the chain grant an *indirect* privilige. We might also require that the subject making the access hold all capabilities in the chain.

pabilities, namely the set comprising all capabilities on chains that contain C. Because chains can overlap, deletion of (all instances of) a single capability could invalidate a union of sets of capabilities, where each set could have each been invalidated by itself. This is illustrated in Figure 7.10. Deleting all copies of C_3 also invalidates C_4 and C_5; deleting all copies of C_6 also invalidates C_7; and deleting C_2 invalidates 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 invalidated.

A particularly attractive way to implement capabilities that name other capabilities is to employ the address of a capability as the name for that capability. This would mean there can be multiple copies of the first and last capabilities in a chain but only one copy of every other capability appearing in the chain. For the structure depicted in Figure 7.10, for example, multiple copies of C_1, C_5, C_7, and C_8 are possible but not of the other capabilities. With this implementation, deleting a single interior capability C of a chain always deletes all copies of that capability. Subjects whose authorization to access an object O might have to be revoked should be given capabilities to other capabilities rather than being given capabilities directly for O; and each subject that might have to revoke authorization from holders of a given capability C should be given a capbility that appears someplace on the chain from C to the capability for O.