A capability system [181] is an operating system that represents its access control policy from the subjects’ perspectives. Recall from Chapter 2 that Lampson’s access matrix [176] identified two views of an access control policy: (1) an object-centric view, called access control lists, where the policy is defined in terms of which subjects can access a particular object (the columns of the matrix) and (2) a subject-centric view, called capabilities, where the policy is defined in terms of which objects can be accessed by a particular subject (the rows in the matrix).

Although the access control decisions made by capability and access control list systems are the same, the capability perspective provides some opportunities to build more secure systems, but this perspective also introduces some challenges that must be overcome to ensure enforcement of security goals. In this chapter, we identify these opportunities and challenges, and describe capability system designs that can leverage the opportunities while mitigating the challenges.

10.1 CAPABILITY SYSTEM FUNDAMENTALS

A capability is a reference to an object and a set of operations that the capability entitles the holder, first formalized by Dennis and van Horn [72]. Such capability references are extended memory references in that they not only provide location or naming information, but they may also provide access rights for that reference [333]. This form of addressing is known as capability-based addressing [89]. Thus, a capability is like a house key [128] in that it permits the holder the access associated with the key. When a process needs to access an object, it presents the appropriate capability to the system, much like one would select the appropriate key to unlock a door. If the capability includes the requested operations, then the access is permitted.

An important difference between capability and access control list systems is the process’s ability to name objects. In an access control list system, a process can name any object in the system as the target of an operation. Typically, the name space of objects can be searched for a specific object (e.g., file names in a directory hierarchy). Then, access is determined by checking the access control list to determine if the subject associated with the process can perform the requested operations. In a capability system, processes can only name objects for which they have a capability. That is, the process can only reference objects to which they have some access (via a capability), then the system can determine whether the capability authorizes the requested operations on that object.
In his book, Levy compares the access control afforded using the real world analogy of a safe deposit box [181]. Two alternatives that the bank may use to control access to your safe deposit are: (1) keep a list of those persons who are authorized to access the box or (2) provide a set of keys that you can distribute to those persons you wish to give access. In the first approach, the bank uses an access control list to authorize each person’s request to access your box. In the second approach, the bank provides a capability (i.e., the key) that you may use to create copies to distribute to others to which you may give access. The differences are between the two approaches are mainly in the cost of authorization and the ease of revocation. The capability approach results in less work for the bank to perform authorization, as simply possession of the necessary key is sufficient. However, if you want to remove access for one person, the access control list enables immediate revocation, whereas the capability approach requires the retrieval of keys.

A fundamental requirement of a capability system is that processes not be able to modify or forge capabilities. If capabilities could be forged, then a process could build a capability for any access it desires. To protect capabilities, capability systems traditionally store them in memory protected from the process. For example, the kernel stores one or more capability lists or C-lists for each process. When a process wants to invoke an operation using a particular capability, it references this capability in the invocation. There are a variety of ways that capability systems have enabled processes to reference protected memory securely in such invocations. The Plessey System 250 [63] and CAP systems [334] stored process capabilities in special capability segments. Only the kernel could write to these segments, but typical processes could read from their capability segments to invoke their authorized operations. The IBM System/38 uses tagged memory where tag bits are included with every memory word, enabling the system to distinguish capability and noncapability memory [26, 138]. Amoeba and others [215, 12] use password-protected capabilities, where each capability is encrypted using a key only available to the Amoeba kernel.

10.2 CAPABILITY SECURITY

Capability systems have a couple of conceptual security advantages over ordinary systems: (1) they can be used to define process permissions more precisely, enabling least privilege [264], and (2) they enable permissions to be more easily transferred from one process to another, enabling the definition of protected subsystems [264]. First, capability systems can assign a distinct set of capabilities to each process, so it is possible to assign only the capabilities required for each process based on its specific purpose. This contrasts with an access control list system where all the permissions for any possible use of a program must be assigned. In a capability system, we can customize a process’s
10.3. CHALLENGES IN SECURE CAPABILITY SYSTEMS

capabilities based on the particular execution of that process’s program to minimize the set of permissions while still providing the necessary function.

Second, capability systems enable processes to copy their capabilities for other processes (again, like a house key), so a protected subsystem may perform operations on behalf of clients without the need to be assigned the permissions of all clients. For example, a subsystem that provides access to a database does not need to have its own permissions to the database entries. When it is invoked, the client will provide those capabilities with the request, so that the subsystem need not accidentally give the client unauthorized access.

This problem is known as the confused deputy problem [129], due to Hardy, and it is inherent to ordinary operating systems. In an ordinary operating system, permissions are assigned to objects and it is inconvenient to change which subjects can access an object at runtime. Thus, the subsystem would be assigned the rights to operate on behalf of any client (i.e., subsystems have the union of all their clients’ rights to the objects that they serve). As a result, if the subsystem is confused, it could provide unauthorized access to another client’s data. Whereas in a capability system, this would not be possible because the subsystem would only be able to perform operations using the capabilities that the client could provide.

Many in the operating systems community saw capability systems as the proper architecture for constructing fail-safe operating systems. The capability abstraction was first defined by Dennis and Van Horn [72], and was quickly adopted in a variety of systems [342, 178, 4]. Initially, capabilities were implemented in software, but performance concerns led to the development of several systems that implemented capabilities in hardware [63, 334, 26, 237]. In the 1970’s, it was envisioned that capability systems may provide better flexibility and security than ordinary operating systems, such that they would supplant ordinary operating systems [186]. In more recent years, capabilities have been used to support single address space operating systems [133, 45] and distributed operating systems [215, 65].

10.3 CHALLENGES IN SECURE CAPABILITY SYSTEMS

The fundamental question is whether a capability system architecture is suitable for implementing a secure operating system. Definition 2.5 requires that a secure operating system meet mediation, tamperproofing, and verifiability requirements when enforcing a mandatory protection system. While the requirements for mediation and tamperproofing can be met, the flexible distribution of capabilities fundamental to capability systems present some problems in verifying the enforcement of security goals (i.e., the protection system is not mandatory). In this section, we identify these problems, and we describe how capability systems have been modified to address these problems in the following section.
First, we describe how capability systems satisfy the secure operating system requirements of mediation and tamperproofing. Since capabilities must be used to name objects, access control is inherently bound to access operations in capability systems. For example, in order to access a file, a process must provide a capability that both identifies which file to access (i.e., naming the file as described above) and specifies the rights that the process has over that file. Without the capability, the file cannot even be found, so mediation is fundamental to capability systems.

Tamperproofing requires that the trusted computing base, including the reference monitor and protection state itself, cannot be modified by untrusted processes. Like an ordinary system, a capability system provides untrusted processes with access to services offered by trusted processes, such as the kernel and protected subsystems. For example, if an untrusted process has a capability that enables the execution of a trusted process, the trusted process must protect itself from any input the untrusted process may provide. These problems are not significantly different than for ordinary operating systems, as they may accept requests from untrusted processes (e.g., network communications) or be executed by an untrusted process (e.g., via `setuid`). Capability systems are also fundamentally aware of the need to prevent unauthorized modification of the protection state. As discussed above, the prevention of capability forgery are design principles of such systems.

The flexibility that processes have to distribute capabilities in capability systems conflicts with the desire to prove that particular security goals are truly enforced, so ensuring that security goals are verifiable is problematic in capability systems. Researchers have identified three specific problems in ensuring the enforcement of security goals in capability systems: (1) care must be taken to ensure that the $*$-property in the Bell-LaPadula policy is correctly enforced [23]; (2) capability systems require additional mechanism to ensure that each protection state is *safe* [130], enforcing the system security goals; and (3) changes in security requirements (i.e., policy) require that capability systems *revoke* all newly unauthorized capabilities. In general, these problems are created by the discretionary distribution of capabilities inherent to the model, so solutions focus on how to add mandatory boundaries that ensure operation that satisfies system security goals.

10.3.1 Capabilities and the $*$-Property
Boebert [32] and Karger [158] note that traditional capability systems fail to implement the $*$-security property of the Bell-LaPadula policy [23], as shown in Figure 10.1. Suppose there are two processes, a high secrecy process $A$ that has access to high secrecy data and low secrecy process $B$ that is not authorized to access that data. If our goal is to implement a multilevel security (MLS) policy such as Bell-LaPadula, then the high secrecy process may read data in the low secrecy process. For example, $A$ uses
10.3. CHALLENGES IN SECURE CAPABILITY SYSTEMS

its legal capability $B1$ read to read segment $B1$. Since capabilities are data, the high secrecy process $A$ can read the capabilities (e.g., $B2$ write in the low secrecy process’s segment $B1$. Then, high secrecy process $A$ has a capability to write its secrets (e.g., data from segment $A1$) to a low secrecy segment $B2$, violating the $*$-security property.

While it may be unlikely that an error in a high secrecy process may result in such a leak, remember that secure operating systems must prevent any code running in a high secrecy process, including malware, such as Trojan horses, from leaking data. A Trojan horse could be designed that retrieves write capabilities to low secrecy files to enable the leak.

10.3.2 Capabilities and Confinement

Karger states that the violation of the $*$-property implies that capability systems fail to enforce process confinement [158]. Lampson defined confinement in terms of [177]: (1) processes only being able to communicate using authorized channels and (2) process changes not being observable to unauthorized processes. The failure above in implementing the $*$-property does result in an unauthorized communication channel, but the problem is even broader than this; we must ensure that no unauthorized communication is present for any security policy.

Consider a second example from Karger [157]. An attacker may control a program $P$. When an unsuspecting victim provides a capability $C$ to $P$, the malicious program can store the capability. This enables the attacker to use this capability, presuming that the attacker can run at the same clearance as the victim. Clearly, the attacker should not have access to this capability, but how does the kernel know that a capability stored by a program cannot be used in another context?

Confinement is not achieved in the example above because the program $P$ has the discretion to give the capability to the attacker. Like other mandatory access control systems, we want to define a mandatory policy that ensures that the program $P$ cannot give the victim’s rights away.
CHAPTER 10. SECURE CAPABILITY SYSTEMS

10.3.3 Capabilities and Policy Changes
The third problem is the revocation problem; capabilities are difficult to revoke. Recall Levy’s safe-deposit box example. When keys are distributed among the authorized people, the owner of the safe-deposit and the bank lose the ability to restrict who has access. Should the owner or bank try to change who can access the safe-deposit box after the keys have been distributed they have a couple of challenges in enforcing this change. First, they have to locate all of the keys that were given out. While they may know how many keys were created and to whom they were initially distributed, the keys may not longer be in the possession of those people and they may not remember what they did with them. Second, keys may have been copied, such that it may not be possible to determine whether all the keys have been revoked. In general, it may be easier to change the lock and start again.

This analogy means that should we decide to change the security goals that we want to enforce in a capability system, we may not be able to determine whether we have accounted for all the capabilities necessary to prove that the new goal can be achieved. We not be able to find all of the capabilities to all objects or some unknown copies may have been made. Mechanisms to bring the set of capabilities back into some approved state may be expensive (search all memory) or disruptive (delete all capabilities and start again).

10.4 BUILDING SECURE CAPABILITY SYSTEMS
Work in secure capability systems aims to address these problems to enable effective verification that the system enforces a well-defined set of security goals. While a variety of capability system designs have been modified to solve these problems, we focus on two capability systems: SCAP [157] and EROS [285]. Both SCAP and EROS are capability system designs based on existing designs, CAP [135, 222] and KeyKOS [128], respectively, but extended to solve these fundamental problems. Table 10.1 shows a summary of how EROS and SCAP address the three problems in capability systems. We develop and compare these solutions below.

10.4.1 Enforcing the \( \ast \)-Property
The SCAP design to ensure that the \( \ast \)-property is not violated in capability systems leverages two key insights [157]: (1) capabilities must be loaded into a capability cache prior to use and (2) we simply need to remove unauthorized access from any capability loaded into the cache to prevent leakage. SCAP requires that a process must load a capability into its capability cache (i.e., its capability list or C-list) prior to using it. This load operation provides the operating system with a point of complete mediation to inspect the capabilities being loaded. This mediation can be used to determine whether the capability provides write access to an object with a lower access class than the pro-
Table 10.1: Summary of SCAP and EROS solutions to the major security issues in capability systems.

<table>
<thead>
<tr>
<th>Security Issue</th>
<th>SCAP Solution</th>
<th>EROS Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>*-Property</td>
<td>Convert to read-only capabilities by MLS policy</td>
<td>Define weak capabilities that transitively fetch only read-only capabilities</td>
</tr>
<tr>
<td>Confinement</td>
<td>Use Access Control List to define confinement</td>
<td>Define safe environments for confined processes or test via authorize capabilities</td>
</tr>
<tr>
<td>Revocation</td>
<td>Revocation by eventcounts (single page entry) or revocation by chaining (multiple page entries)</td>
<td>Indirect capabilities that permit later revocation of all descendants (similar to Redell [251])</td>
</tr>
</tbody>
</table>

cess (i.e., where write permission would violate the Bell-LaPadula policy). Enforcement of such an approach requires the SCAP kernel to include labels with capabilities, labels with processes, and an MLS access policy, so that the kernel can assess whether the capability may be loaded legally.

Other capability systems, the Secure Ada Target (SAT) [34] and the Monash capability system [12] implement similar enforcement semantics, albeit with markedly different approaches. SAT implements semantically similar checks as SCAP to ensure that any capability being loaded adheres to an MLS policy, but the enforcement is done entirely in hardware. Monash’s solution is also semantically similar, but Monash is a password capability system which limits access to write capabilities by keeping the encryption keys used for these capabilities secret. Only authorized processes can obtain the key from the system.

Rather than just providing a point of mediation to decide whether to reduce capability permissions, the EROS system defines a capability that automatically generates the correct permissions [286]. EROS defines the notion of a weak attribute for a capability, the combination of which we will call a weak capability. If a weak capability is used to fetch some other capabilities, all the retrieved capabilities are automatically reduced to read-only and weak capabilities. Like SCAP, the reduction is performed when the capabilities are loaded into the capability cache. Unlike SCAP, EROS enforces the *-property without the need to consult an MLS policy at runtime, a potentially significant performance advantage [287].

Instead, EROS requires that an MLS policy be consulted when capabilities are created (e.g., at load time) to determine when a process should be given a weak capability. In order to ensure that this works, the system must ensure that any capabilities
constructed that enable a process to access lower-secrecy memory must be weak capabilities. As EROS does not define how capabilities are initially distributed, a higher-level service is necessary that understands the labeling of processes and capabilities, and has access to a Bell-LaPadula policy, so the \( * \)-property is enforced correctly.

The idea of a weak attribute is based on sense capabilities in KeyKOS which only enable retrieval of read-only capabilities (i.e., in most cases), even if the capability fetched has read-write privileges. The EROS design generalizes this by making the semantics uniform and transitive. First, weak capabilities have the same effect regardless of the type of the object referenced by the capability. Second, if a sequence of capabilities is required to retrieve some target capability, the target capability is reduced to read-only and weak if any capability in the retrieval sequence is weak.

### 10.4.2 Enforcing Confinement

The confinement problem was identified by Lampson in 1970, so several capability system designs in the 1970s aimed to provide confinement guarantees. The HYDRA capability system provided confinement by defining confined protection domains that were not allowed to store capabilities or regular data into potentially shared objects. This prevents leakage of secret data, but does not permit sharing that is legal under Bell-LaPadula model.

The Provably Secure Operating System (PSOS) is a design for a capability system that also provides an approach for confinement. In PSOS, secure documents are defined such that write capabilities may not be stored in such documents. This prevents propagation of write capabilities that would enable leakage. The rules for checking whether a write capability may be stored are complex, as they pre-date the definition of the Bell-LaPadula model.

In practice, there are two ways to enforce confinement: (1) build an execution environment for the process that satisfies confinement requirements or (2) verify that confinement is preserved whenever an access right is obtained. Both HYDRA and PSOS take the former approach. They define restrictions on execution environments that enforce confinement requirements. In the second approach, the system ensures that whenever a process obtains a capability (or access right in general) the confinement requirements of the system are met, or the capability is revoked.

While the first approach is conceptually cleaner and more efficient to execute, it has a fundamental limitation. This limitation is the intractability of the safety problem. A system is said to be safe if all future protection states in a protection system only grant authorized permissions to processes (i.e., confine the processes correctly). Harrison, Ruzzo, and Ullman showed that determining whether an arbitrary

---

1 EROS does check that the capabilities available to a process belong to an authorized set, which also is defined external to EROS, as described below in Section 10.4.2.

2 PSOS was the basis for the Secure Ada Target mentioned above.
protection system prevents a principal from obtaining an unauthorized permission is undecidable. That is, given a current protection state and the operations in a general protection system, we cannot determine whether there may be some future protection state in this system which some principal may obtain unauthorized access.

In addition to HYDRA and PSOS, general access control models have been defined in which verifying safety is tractable, such as the Take-Grant model [155, 31], Typed Access Matrix [269], and Schematic Protection Model [268]. However, all these models require limitations in the possible policy designs that have proven unacceptable in practice.

Unlike these systems, SCAP enforces confinement by mediating each change in protection state. In SCAP, changes in protection state require the loading of capabilities into SCAP’s capability cache, so this serves as an effective point to enforce confinement as well. As described above, SCAP verifies that a process can load a capability into the cache by verifying that the process label permits access to write objects with capability’s label using an external policy. To enforce the $\ast$-property, an MLS policy is checked, but any confinement policy may be used in general. Interestingly, SCAP uses an access control list to define confinement policies, resulting in SCAP enforcement being based on a combination of capabilities and access control lists.

EROS aims to provide confinement with a pure capability approach, rather than the hybrid approach of SCAP [286]. Also, unlike SCAP, EROS provides confinement through the creation of safe execution environments. EROS defines a safe execution environment as one that contains only safe capabilities. A safe capability meets the following requirements:

1. It trivially conveys no mutate authority, or
2. It is a read-only, weak capability, or
3. It is a capability to a constructor that (recursively) generates confined products (i.e., environments).

These capabilities preserve confinement because they do not change the protection state by adding rights (#1 and #2) or they only add rights by generated new confined environments.

These requirements for a confined environment are very restrictive, so EROS includes a fall-back position that is similar to that of SCAP. If a process has some capabilities that are not safe, EROS checks whether these capabilities are authorized. That is, it checks whether the capabilities that the confined environment may ultimately obtain are a subset of the authorized capabilities that define confinement. If so, the process may be executed. Determining all the capabilities that a process may need in advance is nontrivial undertaking, so in some cases, we envision that runtime checking similar to SCAP, but using authorized capabilities to define confinement would be
necessary. Since EROS also mediates capability loads to enforce weak capabilities, this would be feasible.

The SCAP design also considers the impact of covert channels on confinement carefully. SCAP prevents storage channels by eliminating system-wide views of system state. For example, domains are limited to storage quotas and page tables are unshared. Addressing timing channels is a more ad hoc procedure, but is a consideration throughout the design.

10.4.3 Revoking Capabilities

Redell defined the first comprehensive approach to revocation in capability systems [251]. In this system, the owner of an object has a choice whether to grant normal capabilities (i.e., with no hope of revocation) or grant capabilities that are associated with a special revoker capability. A revoker capability is a level of indirection that enables the owner to revoke all the capabilities that reference the object through that revoker capability. Simply by deleting the revoker capability, the other capabilities lose access to the object. This is because the object is only available via the revoker capability.

The idea is shown in Figure 10.2. An owner creates a revoker capability and grants

Figure 10.2: Redell’s revoker capability approach: When the revoker capability is revoked all the capabilities that were based on it are also revoked.

a capability that points to that revoker capability to Process 1. Revoker capabilities may be used by other subjects as well. Process 1 can also create a revoker capability and create a capability that points to it for Process 2. Thus, the owner will revoke both Process 1 and Process 2’s access should she delete her revoker capability. However, Process 1 can only revoke access from Process 2. Note that the capability for Process 3 cannot be revoked (i.e., without examining Process 3’s memory) because it points directly to the object.

Redell’s scheme could result in a deep nesting of revoker capabilities, so SCAP defines two different schemes, called revocation with eventcounts and revocation by chaining. Revocation with eventcounts is appropriate for systems that use the same page table for each shared object. In revocation by eventcounts, an event, such as revoking a
capability, causes an eventcount to change. Eventcount values are stored with capabilities as well, so that should a revocation occur, the eventcount between the capability and the page table will differ, triggering a verification whether the capability is still valid.

If there are multiple page table entries that point to the same physical page (because it is shared by multiple processes), the revocation by eventcounts cannot be used. Revocation by chaining creates a ring of capability records for the same page by adding a pointer field to each capability. Thus, the revocation of any capability in the chain enables triggers a reassessment of the validity of the remaining capabilities in the chain. All such capabilities are accessible because they are chained together.

Both the revocation by eventcounts and revocation by chaining approaches are rather complex and potentially expensive to implement, so the later EROS system reverted to an indirection mechanism similar to Redell [251] to revoke capabilities. An indirect (revoker) capability may be obtained that enables later revocation, as described above [287]. The memory usage problems cited by SCAP as a reason for seeking alternative revocation schemes had become less of an issue by the late 1990s.

10.5 SUMMARY

In the chapter, we examine the construction of secure operating systems from capability systems. Capability systems have conceptual advantages in enforcing security because they can be used to define protection domains specific to a particular execution of a program easily and they enable permissions to be distributed with program invocation preventing the confused deputy problem [129] by limiting the user of others’ permissions. However, capability systems also have some inherent security problems brought about by the discretionary nature of capability management.

The SCAP and EROS capability systems address these limitations by adding mandatory restrictions on the use of capabilities to ensure safe system behavior. They each define mechanisms to limit the capabilities that a process can receive to only those within the system’s security goals (e.g., weak capabilities of EROS), but restricting the system’s behavior in a sufficiently flexible manner requires runtime checks (e.g., on capability loading). Revocation is a conceptual problem for capability systems, but in practice simple ideas, such as Redell’s indirect revoker capabilities, appear sufficient. Thus, the biggest challenge for capability systems, like many systems, is providing a practical execution environment that can be proven to ensure system security goals.