CHAPTER 3

Multics

In this chapter, we examine the first modern operating system, the Multics system [62]. Multics was a large, long-term operating system project where many of our fundamental operating systems concepts, including those for secure operating systems, were invented. For operating systems, these concepts include segmented and virtual memory, shared memory multiprocessors, hierarchical file systems, and online reconfiguration among others. For secure operating systems, the ideas of the reference monitor, protection systems, protection domain transitions, and multilevel security policies (see Chapter 5), among others, were also pioneered in Multics.

We review the history of the Multics operating system, its security features, and then evaluate how Multics satisfies the requirements of a secure operating system developed in Chapter 2. The Multics history has a number of interesting twists and turns that are relevant to those considering the development or use of secure operating systems today. From a technology perspective, the Multics system implementation demonstrates, often more extensively than many of the other secure operating systems that we will study, the secure operating system definition in Chapter 2. In subsequent chapters, we will often compare other operating system implementations to Multics.

3.1 MULTICS HISTORY

The Multics project began in 1963 with the aim of building a comprehensive, timesharing operating system [217, 126]. Multics emerged from the work of an early timesharing system called the Compatible Timesharing System (CTSS) [87], a project led by Fernando Corbató at MIT. Until 1960 or so, computers all ran batch systems, where individual jobs were run in sequence, one at a time. Often users would have to wait hours or days to receive the results of their program's executions. Early timesharing systems could support a small number of programs by swapping the contents of those not running to tape (i.e., there was not enough memory to keep them in memory). Operating systems of the day, whether batch or timesharing, supported a very different set of functions as well, aiming at building programs, loading them into memory, and automating some basic tasks of the system administrators.

CTSS was demonstrated in 1961, and its success motivated the Advanced Research Projects Agency (ARPA) of the US Department of Defense (DoD) to create Project MAC, which stood for Multi-Access Computer among other things [1]. Project MAC proposed to build general-purpose, timesharing services to support large numbers of users simultaneously. It would have to support functions that would enable the multiplexing of devices for multiple processes (i.e., running programs), scheduling of those processes, communication between processes, and protection among processes. Based on a summer study in 1962, the specifications for Multics were developed and submitted for

[1] "MAC stood for Multiple Access Computers on the 5th floor of 545 Tech Square and Man and Computer on the 9th floor [217]."

bid in 1963. Folklore has it that IBM was not interested in Project MAC's ideas for paging and segmentation, so instead General Electric (GE) was chosen to build the hardware for the project, the eventual GE 645. Bell Labs joined the software development in 1965.

The Multics project had very ambitious and revolutionary goals, so it is not surprising that the project had its moments of intrigue. The project called for delivery of the project in two and a half years, but delivery of the GE 645 hardware was delayed such that Multics was not self-hosting [2] until 1968. ARPA considered terminating the project in 1969, and Bell Labs dropped out of the project in 1969 as well. Ultimately, the Multics system itself proved to be bigger, slower, and less reliable than expected, but a number of significant operating systems and security features were developed that live on in modern systems, such as the UNIX system that was developed by some of the associated Bell Labs researchers after they left the Multics project, see Chapter 4.

Multics emerged as a commercial product in 1973. Honeywell purchased GE's hardware business in 1970, and Honeywell sold Multics systems until 1985. As Multics systems were expensive ($7 million initially), only about 80 licenses were sold. The primary purchasers were government organizations (e.g., US Air Force), university systems (e.g., the University of Southwest Louisiana and the French university system), and large US corporations (e.g., General Motors and Ford). Multics development was terminated in 1985, but Multics systems remained in use until 2000. The last running Multics system was used by the Canadian Department of National Defense.

The Multics project was unusual for its breadth of tasks, diversity of partners, and duration under development. While any hardware project requires the development of an ecosystem (e.g., compilers, system software, etc.) for people to program to the hardware, the Multics project was both a substantial hardware project, a revolutionary operating systems project, and a groundbreaking security project. This breadth of tasks would be daunting today. Secondly, the Multics project team represented university and industry researchers in addition to a variety of government and industry engineers. Many members of the project were among our greatest computer minds, so it is not easy to assemble such a group. Thirdly, an astounding thing is that the project persisted for nearly 10 years before any commercial product was released. In today's competitive environment, such a long pre-production phase would be highly unusual. As a result, the Multics project had a unique situation that enabled them to pursue ambitious, long-term goals, such as building a secure operating system.

3.2 THE MULTICS SYSTEM

The Multics system architecture is a layered architecture where executing programs may be permitted to access named system resources that are organized hierarchically. In this section, we first examine the basic principles of the system, then its security features. This information is culled from the many research documents published on the Multics system. The most comprehensive documents written about Multics were Organick's book [237] and the Final Report of the project [280].

[2] A *self-hosting* system can be used to develop new versions of itself.

3.2.1 MULTICS FUNDAMENTALS

The fundamental concepts in the Multics system are processes and segments. *Processes* are the executable contexts in Multics—that is, they run program code. All code, data, I/O devices, etc. that may be accessed by a process are stored as *segments*. Segments are organized into a hierarchy of directories that may contain directories or segments.

A process's *protection domain* defines the segments that it can access. A Multics process's *protection domain* consists of the segments that could be loaded into its descriptor segment and the operations that the process could then perform on those segments. Each segment is associated with its accessibility—i.e., the *subjects* whose processes can access the segment and the operations that they are allowed to perform. Multics has three different ways of expressing accessibility that we will describe in Section 3.2.1.

Segments are addressable either locally within the process's context or by name from secondary storage (i.e., analogous to modern file systems). For segments already in a process's context, Figure 3.1 shows that each process is associated with its own *descriptor segment* that contains a set of *segment descriptor words* (SDWs) that refer to all the segments that the process can directly access. That is, these segments are directly addressable by the process in the system's memory [3].

When Multics process requests a segment that is not already in its descriptor segment, it must name the segment using what is analogous to a file path. Like modern file systems, Multics segments are named hierarchically. For example, the name `/U2/War/NewYearsDay` is processed starting with the root directory, continuing with subsequent descendant directories (i.e., `U2` and `War`), and finishing with the name of the actual segment (e.g., the `NewYearsDay` segment). Thus, Multics segment access provided a blueprint for later hierarchical file systems of UNIX and beyond. If the process's subject has the permissions to perform the requested operation on the segment, then a new SDW is created with those permissions and is loaded into the process's descriptor segment. Note that the process must also have the access to all the directories in the segment's path as well to access the segment.

3.2.2 MULTICS SECURITY FUNDAMENTALS

Multics security depends on some fundamental concepts that we introduce before we detail the protection system and reference monitor. These concepts include the Multics *supervisor*, *protection rings*, and Multics *segment descriptor words*.

Figure 3.2 shows the actions that take place when a user logs into a Multics system. First, a user login requires that a component of the trusted computing base (TCB) verify the user's password and build a process for the user to perform their processing. User logins are implemented by a process called the *answering service*. To authenticate the user, the answering service must retrieve the password segment from the file system by loading the password SDW into its descriptor segment. The loading and subsequent use of the password segment must be authorized by the core Multics

[3] Of course, a segment may have been swapped out to secondary storage, but from the point of view of the process, the segment is available in memory. It will be swapped in invisibly by the Multics kernel.

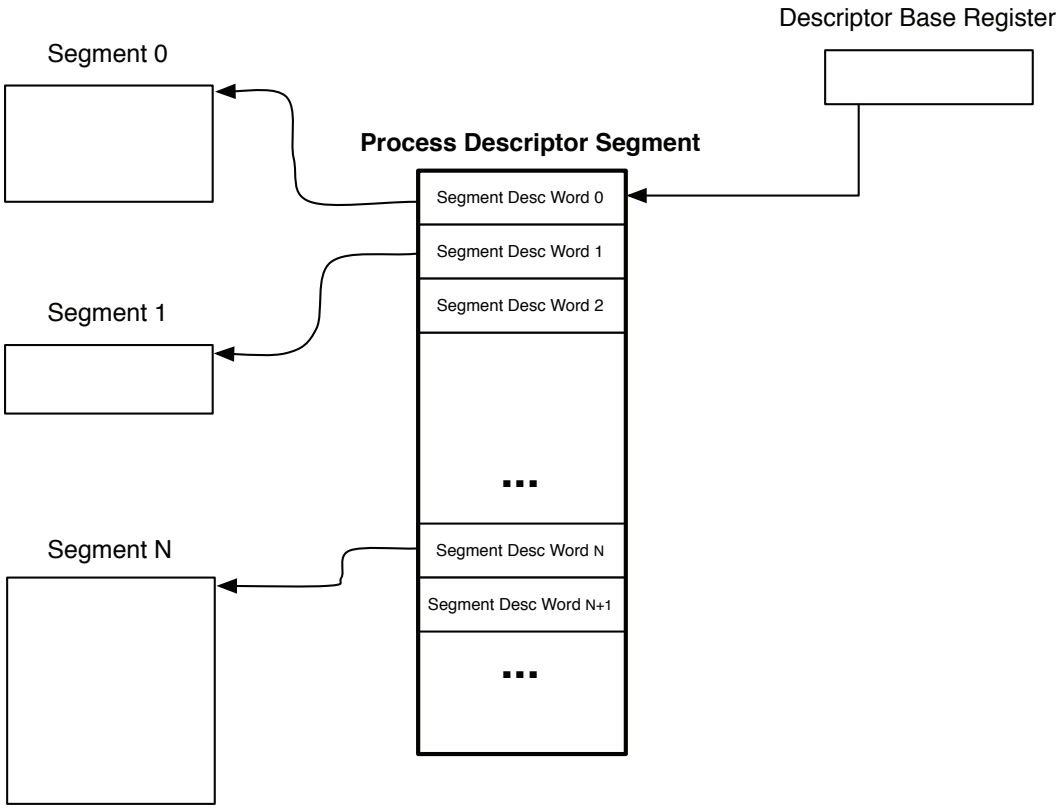

Figure 3.1: Multics process's segment addressing (adapted from [159]). Each process stores a reference to its *descriptor segment* in its *descriptor base register*. The descriptor segment stores *segment descriptor words* that reference each of the process's active segments (e.g., segments 0, 1, and *N*).

component, the *supervisor* [322]. If authorized, a SDW for the password segment is loaded into the answering service's descriptor segment. The supervisor implements the most trusted functionality in the Multics system, such as authorization, segmentation, file systems, I/O, scheduling, etc. Early Multics systems also included dynamic linking functionality in the supervisor, but that was later removed [62] and is also implemented in user-space in modern systems.

The supervisor is isolated from other processes by *protection rings* [281]. Protection rings form a hierarchical layering from the most privileged ring, ring 0 where the most-privilege code in the supervisor runs, to the least privileged ring. There were 64 rings in the GE 645 Multics system, but only 8 were implemented in GE 645 hardware and the rest by some software tricks. The supervisor

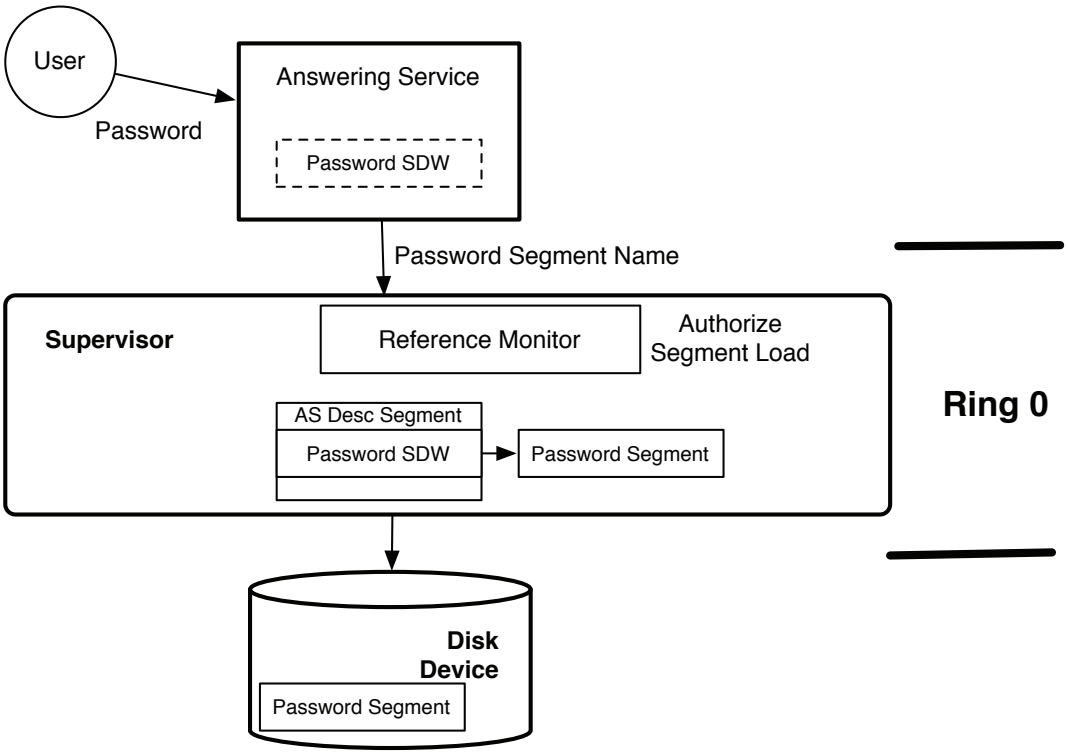

Figure 3.2: The Multics login process. The user's password is submitted to the Multics *answering service* which must check the password against the entries in the *password segment*. The Multics *supervisor* in the privileged *protection ring 0* authorizes access to this segment and adds a SDW for it to the answering service's descriptor segment. The answering service cannot modify its own descriptor segment.

is protected from other processes because only its segments are assigned to rings 0 and 1 [4], and no process running in a higher ring can modify its segments. Thus, processes can only cause a modification of the supervisor's state by invoking supervisor code that runs in ring 0. Multics defines mechanisms to protect the supervisor from malicious input in these calls. The Multics design aimed for layering of function as advocated by other systems of the time, such as the THE system [76], but the rings were ultimately used as a simple, coarse-grained mechanism to protect the integrity of the supervisor and other trusted processes from untrusted code. Of course, modern processors also protect their operating systems using protection rings, although only two levels, supervisor and user, are typically utilized.

[4]The Multics supervisor is divided into ring 0 components, including access control, I/O, and memory management, and ring 1 components that are less primitive, such as accounting, stream management, and file system search.

If the user and password match, then the answering service creates a user process with the appropriate code and data segments for running on behalf of that user. Each live process segment is accessed via a *segment descriptor word* (SDW) as mentioned above. Figure 3.3 shows the SDW layout [281]. The SDW contains the address of the segment in memory, its length, its *ring brackets*,

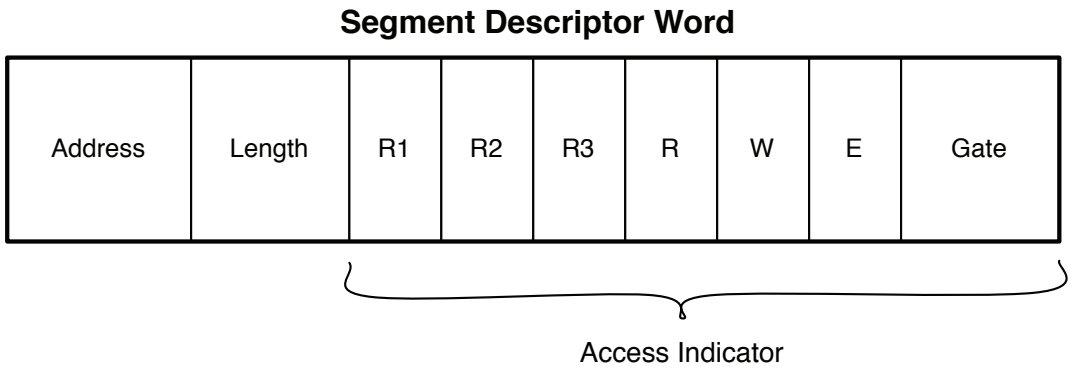

Figure 3.3: Structure of the Multics *segment descriptor word* (SDW): in addition to the segment's address and length, the SDW contains access indicators including *ring brackets* (i.e., *R1, R2, R3*), the process's ACL for the segment (i.e., the *rwe* bits), and the *number of gates* for the segment.

its process's permissions (`rwe`) for this segment, and, for code segments, the number of gates defined for the segment. When the process references an SDW, its ring bracket limits access based on the current ring in which the process is running. The process permissions (`rwe`) limit the operations that the process can ever perform on this segment. We examine the meaning of the SDW access fields below.

3.2.3 MULTICS PROTECTION SYSTEM MODELS

The Multics protection system consists of three different, interacting models that individually provide distinct aspects of the overall Multics protection system. For simplicity, We introduce the models in isolation first, and the describe the overall authorization process.

Access Control List First, each object (i.e., segment or directory) is associated with its own *access control list* (ACL). Each ACL entry specifies a user identity of processes and the operations that processes with that identity can perform on this object. Note that a user may be specified using wild-cards to represent groups of users. Segments and directories have different operation sets. Segments may be read (`r`), written (`w`), or executed (`e`), and directories may be accessed to obtain the status of the entry (`s`), modify an entry (i.e., delete or modify ACLs, (`m`), or append an entry to the directory (`a`). Note that the ACLs for a segment are stored in its parent directory, so access is checked at the

parent. Also, any modification of an ACL for a segment requires the modification permission on the parent directory.

Example 3.1. Examples of ACLs on a segment include:

```
rew    Jaeger.SysAdmin.*
r      Backup.SysDaemon.*
rw     *.SysAdmin.*
```

Also, examples of directory ACLs include:

```
sma    Jaeger.SysAdmin.*
s      Backup.SysDaemon.*
sm     *.SysAdmin.*
```

When a process requests access to a segment, the ACL of the segment is checked to determine if the user associated with the process has an entry in the ACL that permits the requested operations. If so, the reference monitor authorizes the construction of an SDW with those operations.

Rings and Brackets Multics also limits access based on the protection ring of the process. Each segment is associated with a *ring bracket* specification that defines read, write, and execute permissions of processes over that segment. Also, protection domain transition rules are defined by these brackets. First, a segment's *access bracket* defines the ranges of rings that can read and write to a segment. An access bracket is specified by a range of rings $(r1, r2)$ where $r1 \leq r2$ (i.e., $r1$ is more privileged than $r2$). Suppose a process is running in ring r, then the access rights of that process to a segment with an access bracket of $(r1, r2)$ are determined by:

- If $r < r1$, then the process can read and write to the segment.
- If $r1 \leq r \leq r2$, then the process can read the segment only.
- If $r2 < r$, then the process has no access to the segment.

Such a policy ensures that lower rings (i.e., more privileged) have strictly greater access to segments than the higher rings.

Multics also uses rings to control the invocation of code segments. A second access specification, the *call bracket*, is used along with the access bracket to determine how a process in ring r invokes a code segment. The call bracket is $(r2, r3)$, where $r2$ is same $r2$ as in the access bracket and $r2 \leq r3$. If a process at ring r tries to invoke a code segment with an access bracket of $(r1, r2)$ and a call bracket of $(r2, r3)$, the following cases are possible:

- If $r < r1$, then the process can execute the code segment, but there is a ring transition from r to a lower privileged ring $r1 \leq r' \leq r2$ specified by the segment (typically, $r1 == r2$, so the transition is obvious).

- If $r1 \leq r \leq r2$, then the process invokes the code segment in its current ring r (i.e., no ring transition).
- If $r2 \leq r \leq r3$, then the process can execute the code segment, there is a ring transition from r to the higher privileged ring r' if authorized by the *gates* in the code segment's SDW.
- If $r3 < r$, then the process cannot invoke the code segment.

The call brackets not only define execute privilege based on the process's current protection ring, but they also define transition rules describing the requirements for protection domain transition (e.g., if authorized by all gates) and the resultant ring number for the executing code. Call brackets are the only means of describing transition state in the Multics system.

Multilevel Security Multics pioneered the enforcement of *Multilevel Security* [23, 326] (MLS) in operating systems [5]. An MLS policy prevents a subject from reading data that is more secret than the subject or writing data to less secret objects. A detailed description of MLS and its semantics is provided in Chapter 5.

In Multics, each directory stores a mapping from each segment to a secrecy level. Also, Multics stores an association between each process and its secrecy level. A request is authorized if one of three conditions are met:

1. **Write**: The process requests write access only and the level of the segment/directory is greater than (i.e., *dominates*) or equal to the level of the process.
2. **Read**: The process requests read access only and the level of the segment/directory is less than (i.e., *dominated by*) or equal to the level of the process.
3. **Read/Write**: The process requests read and write access and the level of the segment/directory is the same as the process or the process is designated as trusted.

Intuitively, we can see that a process can only read a segment/directory if its level is more secret or the same as the level of the object and write a segment/directory if its level is less secret or the same as that of the object. This prevents information leakage by preventing a process from reading information that is more secret than its secrecy level and preventing a process from writing its information to objects of a lower secrecy level. In Chapter 5, we formally defines MLS secrecy enforcement.

3.2.4 MULTICS PROTECTION SYSTEM

Multics's protection system consists of these three policies. When a segment is requested, all three policies must authorize the request for it to be allowed. If the requested operation is a *read*, the ACL

[5] MLS is called the Access Isolation Mechanism (AIM) in Multics documentation. We will use the current term of MLS for such access control systems.

is checked to determine if the user has access, the MLS policy is checked to verify that the object's secrecy level is dominated by or equal to the process's, and the access bracket is checked to determine whether the process has read access to the object's segment ($r \leq r2$). When the requested operation is a *write*, the ACL is checked for write access, the MLS policy is checked to verify that the object's secrecy level dominates or is equal to the process's, and the access bracket must permit the current ring write access ($r < r1$).

An *execute* request is handled similarly, except the call bracket is used instead of the access bracket, and the request may result in a protection domain transition. The process must have execute permission in the segment's ACL, the MLS policy must permit reading the segment, and the call bracket must permit execution.

Execution of a segment may also result in a transition from the process's current ring r to the ring specified by the segment (we call this r') based on the call bracket. There are two cases. First, when this process invokes a code segment with a call bracket where $r < r1$, then the process must transition to r' (i.e., a lower-privileged ring). Second, when this process invokes a code segment with a call bracket where $r2 \leq r \leq r3$, then the process must use one of the valid segment gates as an entry point and transition to r' (i.e., enter a higher-privileged ring if the gates allow).

As described in Chapter 2, a secure protection system consists of a protection state, a labeling state, and a transition state that may only be administered by trusted subjects. Multics defines its protection state based on these three models. The ring brackets define the allowed protection domain transitions in the system. There are no object transitions specified in the Multics policy. Labeling is not specifically defined in the Multics policy. Presumably, new segments are assigned the MLS labels and ring brackets from their creator, but this is not specified.

Both the ACL and ring bracket policies are *discretionary access control policies*. That is, the ACLs and ring brackets for a segment may modified by any process that has the modify privilege to the segment's parent directory. Only the MLS policy is *nondiscretionary* or *mandatory*. The MLS policy is loaded with the system at boot-time and is otherwise immutable.

3.2.5 MULTICS REFERENCE MONITOR

The Multics reference monitor is implemented by the supervisor. Each Multics instruction either accesses a segment via a directory or via a SDW, so authorization is performed on each instruction. Originally, the supervisor performed such authorizations, but eventually hardware extensions enabled most SDW authorizations to be performed directly by the hardware [281], as we now are accustomed. The supervisor then became responsible for setting up the process's descriptor segment and preventing the process from modifying it.

In addition to protection state queries, the supervisor also performs protection domain transitions by changing the process's ring as described above. Accessing a code segment has three allowed cases, two that result in a ring transition. Invoking code in a ring below (i.e., more privileged than) the access bracket results in a ring transition to a more-privileged ring. Such transitions require entry through a special gate segment that verifies: (1) the number of arguments expected; (2) the data type

on each argument; and (3) access requirements for each argument (e.g., read only or read-write). The gate segment, also called a *gatekeeper*, aims to protect the invoked code from potentially malicious input from lower-privileged code. The called procedure must also not depend on the caller for stack memory, and it must return to the calling code in the proper ring number r.

The transition to a lower-privileged ring also generates some security issues. In this case, we may leak information as a result of the call to a lower-privileged ring and that the higher privileged code must protect itself on a return. In the first case, we need to ensure that the called procedure in a high ring (i.e., less-privileged ring) has access to the procedure arguments. Since the granularity of control is a segment, each segment in which an argument is contained must be accessible to the called procedure. Multics can enforce protection on segments, such that the called procedure does not get unauthorized access, but that may result in program failures. Thus, some form of copying is necessary. For example, the supervisor copies arguments from its segment to another segment accessible to the called procedure. However, the caller must be careful not to copy unauthorized information, such as private keys, that the less-privileged code may be able to use to impersonate the higher-privileged code.

In the second case, Multics enables the caller to provide a gate for the return, called a *return gate*. This mechanism is similar in concept to a call gate, except multiple calls may result in a stack of return gates. Thus, the SDW is unsuitable for return gates. The supervisor must maintain the stack of return gates for the process.

While supervisor functions are implemented in rings 0 and 1, the fundamental reference monitor services are all in ring 0. For example, the file system search utility has been moved to ring 1, such that the determination of a directory or segment from a name is performed there, but authorization of whether this access is permitted is done in the ring 0 supervisor [279]. That is, the code in ring 1 running due to a user's process, may not have an ACL that permits it access to the segment. Thus, ring 0 can limit the actions of code in ring 1. Decisions about what code belongs in ring 0 and ring 1 was an ongoing process throughout the Multics project. Modern operating systems have generally not made such fine-grained distinctions, potentially to their detriment for security. Nonetheless, programming is much simpler in the modern case.

Some services running in less-privileged rings also must be trusted by the supervisor for some functions. For example, the answering service (see Section 3.2.2) performs authentication, so it assigns the user of a process. Clearly, if it is malicious, the process could get unauthorized permissions by being assigned to the wrong user. Also, the administrator must be entrusted with several operations, supported by code that must then be trusted, such as measuring storage usage, performing backups, and changing permission assignments [264]. A TCB was defined for Multics' B2 evaluation (see Chapter 12 for a discussion on system security evaluation), but the Multics architecture continually evolved, such that its TCB evolved over time. In 1973, Saltzer stated that 15% of Multics programs ran in ring 0 [264], so these programs plus administrative and authentication programs minimally defined the Multics TCB. The Multics team recognized that this was a large number of trusted

programs, but the resolution of what should be in or what should be out of the TCB remained an ongoing issue until the end of the project.

3.3 MULTICS SECURITY

We evaluate the security of Multics system using the reference monitor principles stated in Chapter 2: complete mediation, tamperproofing, and verifiability. Unlike the commercial operating systems discussed in the next chapter, Multics performs well on these metrics. Nonetheless, we will see that it is difficult to completely achieve these requirements. In the next section, we will discuss how the implementation may cause breaches in security, even in well designed systems.

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?

   Since Multics requires that each instruction accesses a segment and each segment access is mediated, Multics provides complete mediation at the segment level. Thus, all security requirements that can be effectively expressed in segments can be mediated in Multics.

   MLS labels for segments are stored in their directories rather than directly in the segments, so Multics must ensure that the mapping between segments and their access classes is used correctly. That is, Multics must prevent a TOCTTOU attack [30] where the attacker can switch the segment assigned a particular name after the access class assigned to the name has been authorized. Traditionally, this is done by restricting a directory to contain only segments of a single access class.

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

   Since Multics mediates each segment access at the instruction level, Multics mediates memory access completely. Multics also mediates ring transitions, in both directions. Thus, the reference monitor provides mediation at memory and ring levels. Multics' ring transitions provide argument validation via *gatekeepers*, which is not part of the reference monitor in modern systems (although argument validation is performed procedurally in modern operating system). In practice, the Multics *master mode* permits code to run in a higher ring level without the full ring transition, see Section 3.4 below.

   Also, TCB servers may have finer-grained access control (i.e., within segments), but this is beyond the ability of Multics. If Multics had a server that is trusted to support clients of multiple secrecy levels, it must also ensure that there is no way that an unauthorized information leak can occur (e.g., the *confused deputy problem*, see Chapter 10). In general, such servers must be trusted with such permissions.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

To verify complete mediation, we need to verify that ring transitions and segment accesses are mediated correctly. These operations are well-defined, so it is straightforward to determine that mediation occurs at these operations. However, the complexity of these operations still made verification difficult. The complexity of addressing resulted in some mediation errors in segment mediation, see Section 3.4.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, from modification?

   The Multics reference monitor is implemented by ring 0 procedures. The ring 0 procedures are protected by a combination of the protection ring isolations and system-defined ring bracket policy. The ring bracket policy prevents processes outside of ring 0 from reading or writing reference monitor code or state directly.

   Some ring 0 code must respond to calls from untrusted processes (e.g., system calls). The only way that ring 0 can be accessed by an untrusted process is via a gate. As described above, gates check the format of the arguments to higher-privileged, supervisor code to block malicious inputs. Thus, if the gates are correct, then untrusted processes cannot compromise any ring 0 code, thus protecting the supervisor.

   Multics *master mode* code was not designed to be accessed without a ring transition to ring 0, but this restriction was later lifted, resulting in vulnerabilities (see Section 3.4 below). Thus, a secure Multics system must not include the unprivileged use of master mode as Karger and Schell identify.

5. **Tamperproof**: Does the protection system protect all the trusted computing base programs?

   The Multics TCB consists of the supervisor and some system services in rings 1–3. Multics relegates standard user processing to rings 4 and higher, so trusted code would be placed no higher than ring 3. If we assume that all the code segments in rings 0–3 are part of the trusted computing base, then the TCB is large, but can be protected in the same manner as the supervisor in ring 0.

   The integrity of the TCB depends on its system-defined ring bracket policy. However, the ring bracket policy is a discretionary policy. It can be modified by any subject with modify access to a directory containing a TCB code segment. Should any process in the TCB be compromised, it could undo protections at its ring level, thus potentially compromising the entire ring. If more-privileged rings contain any code that depends on trust in a less-privileged ring that is compromised, then the compromise may spread further. Thus, Multics tamper-protection is "securable" as Saltzer stated, but discretionary access control makes its tamperprotection brittle. See Chapter 7 to see why the use of discretionary access control is problematic.

6. **Verifiable**: What is basis for the correctness of the system's trusted computing base?

The implementation of the Multics TCB is too large to be formally verified [279]. The project's goal was to minimize the Multics implementation as much as possible, such that most, if not all, of the TCB can be verified using manual auditing. This goal was not achieved by the completion of the Multics project, and in fact this limitation motivated the subsequent work in security kernels (see Chapter 6). As we will see in the next section, this resulted in some security problems in Multics.

7. **Verifiable**: Does the protection system enforce the system's security goals?

   Verifying that the system's security goals are enforced correctly involves ensuring that the policy: (1) protects the secrecy and integrity of the system and user data by the *protection state*; (2) assigns subjects and objects to the policy labels correctly by the *labeling state*; and (3) ensures that all protection domain transitions protect the secrecy and integrity of the system and user data based on the *transition state* defined by the call bracket rules.

   First, the protection state ensures MLS secrecy protection is enforced, although the discretionary management of the ring bracket policy limits integrity protection to the system TCB at best, and only if no TCB process can be compromised. The MLS secrecy policy is a mandatory policy of information flow secrecy goals, so the secrecy goals are enforced by the Multics system given a trusted TCB.

   Second, verifying the correct labeling of segments and processes is challenging since much of this labeling is specified manually. The Multics policy does not explicitly state how new processes and segments are labeled, although we would expect that the norm is to inherit the labels of the creating process.

   Third, the transition state permits low integrity code in a less-privileged ring to transfer control to high integrity code in a more-privileged ring through either gates or return gates based on the call bracket rules. As discussed above, the security of these transitions depends on the correctness of the gates, but most systems do not even have this level of enforcement.

This informal analysis shows that Multics security is largely very good, but risks remain. In this analysis, we describe powerful mediation, expressive tamperproofing, and verifiable secrecy controls and system integrity controls. However, challenges still remain in managing the scope of the TCB, verifying the correctness of system integrity policies, ensuring integrity protection for all processes, labeling processes and segments correctly, and verifying the correctness of all gates. Saltzer identifies nine areas of security risk in Multics as well [264]. In addition to the issues above, Saltzer mentions the need for secure communication between systems, control of physical access to machines, the weakness of user-specified passwords, the complexity of gate protections for the supervisor, the possibility of leaking secrets via reuse of uncleared memory or storage, excessive privileges for administrators, and others. These issues and challenges are not unique to Multics—as we will see, every secure operating system design will fight with these challenges. For a first attempt at building a secure operating system, the Multics project did an admirable job of identifying issues

and proposing solutions, but many difficult issues must be addressed. As Saltzer states, Multics was "designed to be securable," not a single secure configuration.

Of course, building any operating system also requires that the designers consider usability, performance, and maintainability in their design. To a large extent, an operating system is supposed to be invisible to applications. While applications have to use the system's interface to obtain service, the interface should just implement the requests, so programs can run as expected. Of course, the addition of security enforcement may cause programs to no longer work as expected. Requests may be denied for security reasons, and applications may not be prepared to handle such failures. Also, security is supposed to be effectively invisible from a performance perspective. This was a significant problem for Multics, especially given the limited computing power of that time. The small number of deployed Multics systems probably also prevented the usability model of Multics from spreading widely enough to become an accepted norm. As the UNIX community grew to numbers that dwarfed the number of Multics administrators, the computing community came to accept the open, but insecure, approach. Finally, operating systems are complex software components, so they undergo a fair amount of evolution. This was particularly true in the case of Multics, but any system maintenance must still preserve the security guarantees offered by the system. As we will see in the next section, this was not always the case.

3.4 MULTICS VULNERABILITY ANALYSIS

In 1974, a couple of Air Force researchers, Paul Karger and Roger Schell, performed a vulnerability analysis on the Multics system [159]. Unfortunately, the Multics system was too complex for them to do any kind of analysis that may prove the security enforcement of the system (i.e., mediation, tamperproofing, or verifiability), but they examined the system looking for implementation flaws. That is, they were, and still are, firm believers in the Multics approach to building a secure operating system, but they found a number of vulnerabilities in the Multics implementation that raised questions about how to build and maintain secure operating systems.

Karger and Schell's vulnerability analysis investigates whether specific hardware, software, and "procedural" (i.e., configuration) vulnerabilities are present in the Multics system. They found vulnerabilities in each area.

First, a hardware vulnerability was found that would permit an execute instruction to bypass access checking using the SDW. That is, complete mediation could be circumvented due to this vulnerability. The details of the vulnerability require a deeper knowledge of Multics addressing than we provide, but the basic problem is that the Honeywell 645 hardware [6] did not check the SDW access if the segment was reached by a specific format of indirect addressing. Thus, access to the segment containing the indirection was checked for access, but not the segment containing the actual address to be executed. It was found that this error was introduced in a field modification made at MIT and later applied to all processors. While this is was simply an erroneous update, the pressures of balancing performance and security, makes such updates likely. Further, the Multics

[6] This analysis was done after Honeywell had purchased GE's computer division in 1970.

project had no tools to enable the verification of security impact of such changes, so errors should not be unexpected.

Second, a variety of software vulnerabilities were reported by Karger and Schell. One of the more significant vulnerabilities was an error caused by misuse of a supervisor mode of execution, called *master mode*. Master mode is an execution state that permits any privileged processor instructions to be executed in the current ring. The original Multics design required master mode code to be restricted to ring 0 only [322]. However, this design choice resulted in all faults (i.e., divide-by-zero, page faults, etc.) incurring a ring transition from the user ring to ring 0 where the fault handler was located and then back to the user ring. A proposal to reduce the overhead on the system was to enable execution of some fault handling (e.g., divide-by-zero and access violations) in the user ring. These faults are reported to user programs anyway by a *signaller* module, so the proposal was to run the signaller in user rings. But, the signaller uses some privileged instructions, so it must run in master mode.

Permitting the signaller to run in master mode in a user ring was deemed secure because of the restricted manner in which the signaller must be invoked, but this code was not designed to protect itself from malicious calls. The problem is that the signaller's code expects a register to be loaded with a reference to a section of the signaller's code when it is called. Unfortunately, the signaller does not check that the register value is legitimate, so when the code became addressable in user rings, it became possible for a malicious user program to set the register to an arbitrary location "permitting him to transfer to an arbitrary location while the CPU was still in master mode [159]." Thus, a significant vulnerability was created in the Multics system. The problem was that the signaller code had been written with other design rules in mind. This is why it is important to: (1) have clear design rules and (2) have approaches and (automated) tools to verify that the implementation meets the design rules. Unfortunately, most operating systems are implemented without clear design rules for security, and few approaches are available to verify compliance with such rules.

Third, Karger and Schell demonstrated that software vulnerabilities, such as the one above, then enable compromise of all Multics security through further "procedural" vulnerabilities. They demonstrate how an attacker can: (1) take control of the Multics patch utility enabling modification of trusted programs; (2) forge the user identification of processes under the control of the attacker; (3) modify the password file; and (4) hide the existence of the attacker by modifying the audit trail and installing backdoors into the system. This work demonstrates many of the challenges that modern operating system designers face of hidden threats, such as *rootkits*. Even if the design is secure and comprehensive, implementation mistakes or poor maintenance decisions can introduce significant vulnerabilities.

3.5 SUMMARY

The Multics designers were the first to tackle the challenge of building an operating system that enables comprehensive enforcement of practical secrecy and integrity requirements. This challenge was just one of several that the designers were faced with, as Multics was also one of the first,

structured, time-sharing operating systems as well. As the security analysis shows, the Multics design addressed many facets of building a secure operating system, including defining a reference monitor to enforce a mandatory secrecy policy and developing a protection ring model to protect the integrity of the trusted code, among several innovations. Multics set the foundations for building secure operating systems, but our security analysis and the vulnerability analysis of Karger and Schell show that many difficult issues remain to be addressed. Subsequent work, described in Chapter 6, aimed to address many of these problems, particularly reduction in TCB complexity. First, to clarify the idea of a secure operating system further, we will examine why ordinary operating systems, such as Windows and UNIX, are fundamentally not secure operating systems in Chapter 4.