Implementing Middleware for Content Filtering and Information Flow Control

Jeffrey Choi Robinson†, W. Scott Harrison‡, Nadine Hanebutte‡, Paul Oman†, and Jim Alves-Foss†

†Center for Secure and Dependable Systems
University of Idaho
Moscow, ID 84844-1008
208-885-4114
[jeffreyr,oman,jimaf]@csds.uidaho.edu

‡St. John Fisher College
3690 East Avenue
Rochester, NY 14618
589-899-3811
[sharrison,nhanebutte]@sjfc.edu

ABSTRACT
This paper discusses the design and implementation of a middleware guard for purposes of content filtering and information flow control in the Multiple Independent Levels of Security (MILS) architecture. The MILS initiative is a joint research effort between academia, industry, and government to develop and implement a high assurance real-time architecture for embedded systems. The MILS architecture incorporates a separation kernel with formal system security policies that are evaluatable, non-bypassable, tamper-proof, and always invoked. Vendor specific high-level applications are assumed to be untrustworthy components, however, some information flow control needs to be performed by middleware entities external to the applications.

In the MILS architecture, a MILS Message Router and guards are placed between communicating entities to act as message content filters and enforce information flow control. As the MILS architecture does not restrict the protocols that can be employed for communications between applications, a distinct guard is needed for filtering messages within each protocol. Incorporating protocol specific guards in MILS embedded systems aids in the formal certification of those systems or the high-assurance safety critical formally-proven applications. The guards enable formally-proven security policies that guarantee information flow control, data isolation, predictable process control, damage limitation, and resource availability. An example is provided using a multi-level secure file server (MLSFS) that uses a GIOP guard for fine-grained access control. The inclusion of a GIOP guard reduces the complexity and the effort necessary to certify the MLSFS.

Categories and Subject Descriptors
C.2.0 [Computer-Communication Networks]: General–security and protection, C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems, D4.6 [Operating Systems]: Security and Protection-access controls, information flow controls, separation kernels.

General Terms

Keywords

1. INTRODUCTION
Designing and building secure systems is often a complex and challenging task. High-assurance systems require convincing evidence that they support critical safety as well as security properties. If systems fail to meet these critical requirements, the potential for security breaches increases which can lead to the loss of life [2][6].

This paper describes the design and implementation of a middleware guard that filters messages and enforces information flow. It is a modular component that is inserted into communication pathways to mediate information flow in a high assurance system. The inclusion of the guard reduces the need for fine-grained access control in relevant MILS systems. This approach provides a reduction in system complexity. This concept is important because the reduction in complexity makes high assurance system certification processes manageable.

The avionics community has addressed the need for safety-critical systems by developing the DO-178b and DO-255 standards, which provide a set of guidelines for the design, analysis, and evaluation of system safety [9][12]. Though adequate for the safety evaluation of airborne systems, neither is sufficient to address the security concerns of critical security systems, such as weapons systems that protect national security. Such high-assurance systems require the rigorous specification and implementation requirements outlined in the Common Criteria (CC) [5]. The CC provides guidelines for the design, analysis, and evaluation of critical systems defined at seven Evaluation Assurance Levels (EAL). The higher the assurance level, the stricter the requirements mandated by the CC. At the highest levels (EAL 5-7), the CC requires the use of formal methods, mathematical models, and proofs [2].

Formal verification of code is often arduous and time-consuming. In fact, code-bases with a large number of lines of code (LOC) are considered to be unverifiable [7]. The majority of code-bases are
The MILS architecture enables the enforcement of application-level policies by using the security features of the kernel and other lower-level components to create entities that reside along the only authorized communication paths between partitions. A good example of a MILS security component is an application-level protocol filter called a guard. Guards enforce application-specific policies on communication messages. Since MILS guards can be verified independently of other components, they can be built once and inserted into the communication path of any MILS system that needs application-level message filtering. This paper describes the design and implementation of a guard that filters messages using Common Object Request Broker Architecture (CORBA) General Inter-Orb Protocol (GIOP) in a MILS system. The example presented in this paper shows how a GIOP guard can be used to replace the fine-grained access control mechanisms in a multi-level secure file server, as well as the ease of integration into legacy applications.

2. THE MILS ARCHITECTURE

MILS is a verifiably secure architecture for running different security level processes on the same high-assurance system. The MILS architecture accomplishes this by providing two types of separation. First, MILS enforces a separation policy that strictly controls communication between processes of different security levels. This prevents, for instance, a top-secret process from communicating with a neighboring unclassified process if it would violate security policy on that system. Second, MILS separates traditional kernel-level security functionalities into external modular components that are small enough for rigorous security evaluation using formal methods. Verifiable secure systems are then built from multiple certified components that can be developed quickly without great expense by independent vendors.

The foundation component of MILS is the Separation Kernel (SK). The SK segregates processes and their resources into isolated execution spaces called partitions. Processes running in different partitions can neither communicate nor infer each other’s presence unless explicitly permitted by the SK in accordance with a security policy. The SK enforces data isolation, controlled information flow, periods processing, and damage limitation security policies on a single microprocessor. Within one or more partitions of each MILS system is the MILS Message Router (MMR). The primary function of the MMR is to route communication between partitions while supporting the data isolation, controlled data flow, periods processing, and damage limitation policy of the SK. The MMR also has the ability to grant or deny untrusted inter-partition communication according to a MLS policy, which it enforces in tandem with the SK. However, it should be noted that the MMR’s role is only in place when there needs to be a routing decision made. For instance, communication between two partitions operating at the same level within the same system does not need to be routed through the MMR. This setup is referred to as trusted inter-partition communication or a trusted enclave.

In addition to message flow control imposed by the SK and the MMR, a MILS system must also contain fine-grained application-level filters. In a system which supports multiple communications
CORBA AND GIOP

CORBA is an open, vendor-independent standard for distributed object-oriented applications. It was introduced by the Object Management Group (OMG) in 1991 and has matured into an industrial strength standard for developing n-tier distributed applications. CORBA runs within the classic client/server application framework. Client applications employ the services of objects (called servants) that are owned by remote server processes (Figure 3).

3.1 CORBA ORBs

Object Request Brokers (ORBs) mediate the communication between clients and servant objects. When a client wishes to invoke a servant object, it first sends the request to its own ORB. The client ORB then locates the servant object and forwards the request to the servant's ORB. Finally, the servant ORB relays the request to the servant object. The servant object's response is handled in similar fashion. Since ORBs are positioned between clients and objects, CORBA is often referred to as middleware.

Because inter-ORB communication is well defined and machine independent, ORBs from different vendors, written in different languages, and running on different platforms are designed to be interoperable. Likewise, this same interoperability is extended to CORBA clients and servants since ORBs always mediate their own communication.

3.2 GIOP and IIOP

CORBA ORBs communicate using the General Inter-ORB Protocol (GIOP). GIOP is specifically designed to be a well-defined and machine-independent protocol to provide interoperability between all ORBs. OMG has done this by incorporating the following three core elements into the GIOP specification.

1. Common Data Representation (CDR) - GIOP defines an on-the-wire format for every CORBA Interface Definition Language (IDL) primitive. CORBA IDL is used to allow object compatibility and interaction across varied implementations, e.g., C++ and Java. This ensures that all data types passed between clients and servant objects have a consistent representation across different platforms and programming languages.

2. Message Formats - GIOP defines eight specific message types with which ORBs can communicate (described in the next subsection).

3. Transport Assumptions - Some assumptions are made about the network transport-layer protocol which GIOP runs above. The underlying transport mechanism must be a semi-reliable, connection-oriented, and byte (octet) stream protocol.

GIOP can be mapped onto any transport-layer protocol that meets the transport assumptions. The most common mapping of GIOP is onto TCP/IP and is called Internet Inter-ORB Protocol (IIOP). Our research has focused solely on the IIOP variant of GIOP, so GIOP and IIOP are used interchangeably throughout the rest of this paper.

3.3 GIOP Messages

GIOP messages are categorized according to their function as either administrative or for object invocation. Briefly, the administrative messages are:

- LocateRequest - locate a servant object
- LocateReply - response to a LocateRequest
- CancelRequest - cancel a pending Request
- CloseConnection - gracefully abort the connection
- MessageError - response to a malformed message

The object invocation messages are:

- Request - invoke a servant object method
- Reply - response to a Request
The MLSFS does not verify that a request message originates from the specified requester or that the requester has the security credentials claimed in the request message. In other words, the system does not check to see if request messages are spoofed.

The MLSFS follows the assumption that services such as CORBA and GIOP offer in the MILS architecture and store and distribute information with different security classifications, e.g., top secret, secret, confidential, and unclassified. Remote processes, e.g., user level applications, from different partitions can invoke read and write methods on the system. The MLSFS utilizes transparency that middleware services such as CORBA and GIOP offer in the MILS architecture. It is an assumption that the MLSFS exists on top of a secure and safe configuration of the MILS architecture and uses MILS components to provide requirements such as authentication. For instance, the MLSFS does not verify that a request message originates from the specified requester or that the requester has the security credentials claimed in the request message. In other words, the system does not check to see if request messages are spoofed.

The MLSFS follows the assumption that services such as authentication occurs somewhere within the communication path by MILS components and that if a process sends a spoofed request message that it will be rejected. The GIOP protocol would fail to meet this requirement. A client classified to access secret data, e.g., Secret_Read(), could attempt to invoke a read method on top secret data, e.g., TopSecret_Read(). As previously mentioned, there is no stipulation in the GIOP protocol to prevent a low level user from invoking a higher level method. One solution to this problem would be to pull the authorization functionality into the MLSFS. However, doing so would complicate the system and introduce unwanted issues of code complexity and an unverifiable code-base. Instead, using MILS middleware guards to assist in this process provides the benefits of the component-wise certification process discussed in earlier sections. By pushing out requirements like authentication and relying on MILS middleware guards, it makes the verification process much easier [10].

4.1 Implementing the MMR and GIOP Guard
The introduction of MILS architecture greatly simplifies the verification requirements of the MLSFS. In this section, we discuss in more detail how to implement information control flow and message filtering through a combination of the MMR and GIOP guard. The GIOP guard exists outside of both the client...
and the servant object and has all GIOP messages routed through it by the MMR. The MMR need only recognize GIOP messages in order to route them to an appropriate guard. The MMR does not need to be capable to fully parse messages; it only needs to identify the type. When the GIOP guard receives a message from the MMR, it parses the message, consults a GIOP specific security policy, and then notifies the MMR whether to allow or deny the message. If the message is allowed by the GIOP policy, the MMR sends the message on to its destination. Otherwise, the message is discarded.

It should be noted, however, that simply discarding the message will generally be insufficient. If the client never receives a response from the server with a result, the client will likely continue to make the same request. Thus, one function of the GIOP guard is to generate "error" packets, which look like standard GIOP error messages (e.g., OBJECT-NOT-EXIST). These will be sent back to the client. It should be noted that in a GIOP message, it must be specified whether a response is expected or not. If no response is expected, obviously, an error message need not be returned and the offending request can simply be dropped.

We will illustrate the fine-grained access control with an example of the MMR and GIOP guard working together to enforce MLS and transport policies. Client A is a CORBA client application running in secret-level partition 1 and client B is a CORBA client running in unclassified-level partition 2. A CORBA MLSFS object is running in multi-level secure partition 3, which has both secret and top-secret clearances. The MLSFS object has two methods, Secret_Read() and TopSecret_Read(). Figure 7 illustrates this situation. The partitions which contain the MMR and GIOP guard are labeled as a MILS partition, as it has no specific security level itself.

### Table 1: MMR Communication Paths

<table>
<thead>
<tr>
<th>Source Partition</th>
<th>Destination Partition</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclassified</td>
<td>Secret</td>
<td>No</td>
</tr>
<tr>
<td>Secret</td>
<td>Unclassified</td>
<td>No</td>
</tr>
<tr>
<td>Secret</td>
<td>MLSFS</td>
<td>Yes</td>
</tr>
<tr>
<td>MLSFS</td>
<td>Unclassified</td>
<td>Yes</td>
</tr>
<tr>
<td>MLSFS</td>
<td>Secret</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The MLS policy, for this system, is that an unclassified process can only communicate with other unclassified processes, and that secret or top-secret processes can communicate with any higher-level process. This is the portion of the policy which will be enforced by the MMR, and is shown in Table 1. The MLSFS access policy, however, requires that data access within the MLSFS must be done only by a client at an equal or dominated security level. Specifically, the Secret_Read() can only be called by a secret or top secret-level process, and the TopSecret_Read() can only be called by a top secret-level process. This is a GIOP-

### Table 2: GIOP Guard Policy

<table>
<thead>
<tr>
<th>Source Partition</th>
<th>Destination Partition</th>
<th>Method</th>
<th>Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secret</td>
<td>MLSFS</td>
<td>Secret_Read()</td>
<td>Yes</td>
</tr>
<tr>
<td>Secret</td>
<td>MLSFS</td>
<td>TopSecret_Read()</td>
<td>No</td>
</tr>
<tr>
<td>Top Secret</td>
<td>MLSFS</td>
<td>Secret_Read()</td>
<td>Yes</td>
</tr>
<tr>
<td>Top Secret</td>
<td>MLSFS</td>
<td>TopSecret_Read()</td>
<td>Yes</td>
</tr>
</tbody>
</table>

specific refinement of the overall communication policy, and this will be enforced by the GIOP guard. Table 2 illustrates the GIOP guard policy. The table lists only secret and top secret partitions. Since the MMR disallows communication between unclassified partitions and the MLSFS, unclassified source partitions are not explicitly addressed in the GIOP guard Policy. It is important to note that, for purposes of the GIOP policy, anything not explicitly allowed would be denied. In addition, while there is no top secret-level partition in Figure 7, the GIOP guard Policy is setup to support a top secret-level partition if it were introduced into the architecture.

Figure 8 illustrates the MMR and GIOP guard working in unison to enforce the overall security policy. As shown, the MMR allows client A to communicate with the MLSFS object because partitions 1 and 3 both have secret clearance. The GIOP guard, however, restricts client A's communication with the MLSFS object to only invocations of the Secret_Read() method since A does not have the top-secret clearance required to invoke TopSecret_Read().

Client B is not allowed to access the MLSFS object at all, because of our policy that unclassified partitions are not allowed to communicate with any other partitions which are not unclassified.

It should be noted that the applications are not aware that any of this filtering is taking place. The MMR is the only route (via SK mechanisms) by which communications can take place. Further, since only the MMR can access the guards, the guards and the filtering they perform are both non-bypassable and transparent to communicating partitions.

### 4.2 Legacy Applications

A legacy application (one that was not written to perform access at different security levels) could easily be integrated into a MILS system. Assume there is a legacy client which existed in conjunction with a secret database. If the client and the database were in a secret-level system, there would be no need to create
different types of read calls within the client, e.g., the only access to the database was Read().

If information in this secret-level database were integrated into the multi-level secure database as described above, it would not be easy, without the MILS components, to use this client, as the calls are different. The database is expecting a Secret_Read() and the client only issues Read() calls. While the database can be rewritten to account for this, this requires increasing the LOC of the database, potentially making it unverifiable, as well as potentially having to add database functionality for every legacy client that will be integrated into the system.

With MILS, none of this is necessary. Using the components described above, the legacy client can be placed in a secret partition. This allows the MMR to enforce basic communication policies transparently via the SK (which is possible with many commercial SKs). The GIOP, on analyzing a message from the legacy client can simply replace the Read() calls with Secret_Read() calls within the GIOP message. This is possible because the MMR is aware of where the message came from. Then, the MMR routes the message to the database. The database receives only Secret_Read() calls from the legacy client.

When a call from a legacy client is received, the MLSFS responds and the GIOP guard must again change the content of the message, as the client is expecting a Read() response whereas the database is sending a Secret_Read() response. The guard, through looking up in a history table what this message was a response to, is able to then change the parameters of the response to a Read() response.

Finally, in addition to changing object names, parameters may be added, deleted, or reordered to allow for the integration of legacy applications by the guard.

5. CONCLUSION

In this paper we have demonstrated how a security policy can be enforced on GIOP messages sent between MILS partitions. A policy that allows or disallows method invocations based upon the requesting client partition, the servant object, and the object method can be used. We have also shown how guards can allow a MILS system to enforce both MILS and application-specific policies, thus providing fine-grained access control of inter-partition communication.

In creating the MILS GIOP guard, we have identified some issues that will be crucial in developing formally certified MILS applications. First, most application-level protocols employ an acknowledgment scheme for some or all messages sent. This implies that a guard must be seamlessly integrated into the protocol’s communication scheme to provide policy enforcement for different protocol components. Second, simply having the MMR drop disallowed messages (even for the case when no reply was expected) is not a very graceful way to handle policy violations. In the case of protocols (including CORBA IIOP) that run over TCP/IP, dropped messages will be re-sent repeatedly until a timeout occurs at the TCP stack level. This can cause applications to hang until the timeout and could also be used in inference attacks. A better solution would be to send a "spoofed" error message to the application that sent the illegal message. The error message would be an application protocol error message, and would appear to have come from the illegal message's destination. This would make it impossible for the offending application to know that filtering by a guard is being performed, that is, the application has no knowledge which messages the guard allows or disallows. This prevents the application from hanging.

The MILS initiative is devoted to finding an efficient way to build verifiable secure systems. By separating security functionality into modular components, high-assurance systems can be engineered and evaluated rapidly and independently, e.g., the MLSFS [10]. The MILS architecture is an approach to system design triggered by the DOD's increasing need for secure, high-assurance, embedded systems. Separation kernels, the lowest layer of the MILS architecture, are already being deployed by multiple real-time operating system vendors for embedded systems [4]. Common criteria protection profiles are currently being developed for both the separation kernel [8] and MILS components [13].

6. ACKNOWLEDGMENTS

This material is based on research sponsored by AFRL and DARPA under agreement number F30602-02-1-0178 and the NSF SFS grants DUE-0114016 and DUE-0416757. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFRL, and DARPA or the U.S. Government.

7. REFERENCES


