# A Tool for Securely Integrating Legacy Systems into a Distributed Environment

Tim Souder and Spiros Mancoridis
Drexel University
Department of Mathematics and Computer Science
3141 Chestnut Street, Philadelphia, PA 19104, USA
{gtsouder,smancori}@mcs.drexel.edu

## Abstract

*Legacy systems provide services that remain useful beyond the means of the technology in which they were originally implemented. Our Legacy Wrapper tool packages the services of a legacy application in order to redistribute as a distributed object. In this new environment, the wrapper provides its own layer of security between the security domains of the host and the distributed object system. This security layer includes a sandbox for the application that is designed to protect the application against malicious users and the host from malicious applications.*

*In this paper we will present the Legacy Access model and the Legacy Wrapper system. The Legacy Access Model is an original system access model that presents a four tiered sandboxing model for wrapping legacy applications: complete encapsulation, shared sandbox, single sandbox, sandboxless operation. The Legacy Wrapper tool is an implementation of the model that combines wrapping an application for security purposes with wrapping it for distribution in a distributed object system.*

## 1. Introduction

Legacy systems are those whose usefulness has extended beyond the expectations of their creators. The Legacy Wrapper attempts to extend the usefulness of those applications by facilitating their integration into modern distributed systems.

As systems age [11], the knowledge base that created them fades. Eventually, the needs that the system addressed change, and, therefore, the system must be changed. Here, the system designers have three options available to them: modify the system (and potentially cause its failure), create a new system with the new functionality, or keep the old system and create a new layer, a wrapper between the original system and the new program interface [13].

The Legacy Wrapper is designed to provide a generic wrapper that extends the usable life of a legacy application without modifications to the legacy application. The wrapper distributes the application's services through a distributed object system. However, merely providing the application services may not be adequate when the application contains or manipulates sensitive data, because its services might be distributed into a hostile environment such as the Internet. Thus, the application wrapper must not only distribute the application's services but also secure the application.

In this paper, we describe the Legacy Wrapper and how it was used to wrap a sample legacy application.

## 2. Background

Security is a method to maintain accountability and control access to system resources. In the early days of computing, both programmers and users of a system were trusted implicitly, because physical access to the computing center was required to access the system. As systems became distributed, physical access was no longer required to the system. In place of the original physical access controls, software security was introduced to the systems.

Since these early access models were an extension of the original physical security models, users were granted trusted access to a host. This created problems when distributed systems were introduced that granted trust to hosts rather than the individual users. All trusted users on one host were implicitly granted access to other systems through the trust relationships those systems had with their original host [5]. In the realm of physical security this would be equivalent to a castle with a moat and draw bridge. People who were granted access across the draw bridge could freely roam the castle including the treasury and throne room. Door locks were introduced to limit one's access within the castle as the draw bridge limited access from without.

Similarly, the view of a user evolved from an entity given a high level of trust to an entity which assumes a set of roles. In this model, the roles are granted access to the system. Thus, the users who perform those roles are granted access for only the activities related to those roles. Thus, Role Based Access Control (RBAC) entered the field of computer security [15].

In the castle, there was an occasional need to confine individuals within a subset of the castle. In software, applications run with the full privileges of the users who invoked them. This is not always desirable in the case where the application is not trusted to perform the tasks for which it was intended. Sandboxing, which creates a virtual environment for the application, was introduced to limit the application's access to the system as RBAC had limited the user's access to the system [19].

In the distribution of computer systems, the information encapsulated within the systems became distributed. Distribution of information generally transfers it across a secondary medium (e.g., an Ethernet cable) between nodes in the system. In this secondary medium, the information is now publicly available to anyone who has physical access to the medium, defeating the RBAC controls that were installed to limit access. Hence, information must be protected in transit. In the pre-computer era, messages were protected by encoding them with a system that was (hopefully) only known to the sender and receiver of the message. Thus, if the message was intercepted, its contents would remain undiscovered. In computer systems, encryption is designed to provide this data protection [14].

As data without context is meaningless, information without the tools to access it is useless. Thus, it became necessary to not only distribute the information encapsulated in the system, but also the means to access it, the legacy applications themselves. However, by distributing the applications, the information concealed by the tools could become discovered through the means that the tools use to access their information. For example, Goldberg and Wagner discovered that the Netscape web browser (version 1.1) used a combination of process id, parent process id, and time of day to generate the random seed for encryption of Secure Sockets Layer (SSL) communication [4]. This discovery would enable them to gather enough information to test for correct key values, if not to completely crack the SSL encryption [2].

The flag flying above a castle's ramparts will immediately tell an approaching army whether its occupants are friends or foes. To protect the anonymity of the castle's inhabitants, its flag must be lowered. Likewise, to ensure the distributed object system's roles are maintained and the information remains encapsulated by the legacy tool that accesses it, the tool must be protected from accidently revealing the information it encapsulates. Our research focuses in this area, and this paper describes how we seek to protect the tool as others before us have sought to protect the information it encapsulates.

## 3. Legacy Access Model

### 3.1. Security Domains

When legacy applications are wrapped, they are commonly designed to provide a service to a trusted set of users [13]. Before the application is wrapped, users are permitted direct access to the system (i.e. they can telnet [18] to the host system). The host defines which users were permitted to access its services through a locally-defined login procedure [18]. When the application is wrapped, however, it essentially runs outside the hosts authentication mechanism (i.e. users can access an application on the host without logging into it first). To limit this remote access, distributed object systems impose their own user identification mechanisms, and the host trusts the distributed object system for proper user identification. Essentially, then, the application sits in an area between the host's security and the distributed object model's security system *(figure 1)*.

This model is successful when a host trusts the Distributed Object Model (DOM) and the DOM trusts the host. However, when one or the other is not trusted, problems quickly arise. If the DOM's security is breached, the host may be compromised through the user-level access that is given to the application (many host attacks require only user-level access to the system [5]). Likewise, if the host is compromised, the application may be used to attack the DOM and transitively other hosts that are accessible through the DOM.

Thus, both the host and the DOM need to be shielded from the access granted to the application. That is, when one system fails, the application cannot be used as an access point to cause the failure of the other system. By running the application within a sandbox (described in the next section), an application's access to both the host and the DOM can be controlled. Since neither the host nor the DOM can directly access the application, one system's failure is not extended into the other *(figure 2)*.

Since the sandbox has access to both systems, the sandbox's access may be questioned in the same manner as the application's access had previously been questioned. Since the sandbox is a single application, its integrity may be verified [14]. Therefore, once the sandbox is verified to work as intended [17], it can be used to run applications that are unverified or cannot easily be verified. For example, the sendmail 8.9.3 distribution contains over 52,000 lines of source code. An application of that size has a large num-

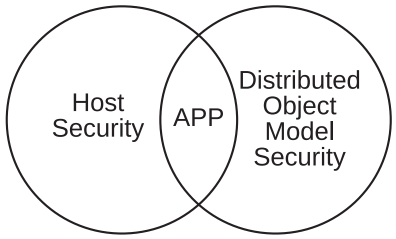


**Figure 1. Overlap of Host and DOM Security Models** When an application is directly packaged as an object, it gains privileged access to both the host and distributed object model (DOM) security domains.

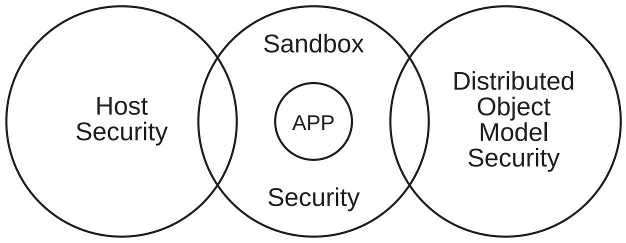


**Figure 2. Sandbox Security Layer** The application is isolated from both the Host and DOM security domains by encapsulating it within the sandbox.

ber of potential system states. Thus, it would be difficult, or impossible, to verify that all of the states are secure [1].

Implementing a security model in the sandbox extends the idea that neither the host's nor the DOM's security model is completely trusted.

## 3.2. Sandboxing

To sandbox an application, a virtual environment is created for the application. Then, the application is executed within this environment. Ideally, the sandbox provides a complete set of virtual system resources to the application creating a virtual machine for the application. The MVS operating system, successor to the IBM OS/360 system software, provides this service [5]. It creates a complete virtual machine in the Virtual Machine Monitor (VMM). This virtual machine is so complete that other VMM systems may be recursively installed within a VMM logical partition.

Providing a complete set of system services through the sandbox may be expensive, because those services are replicated for each sandbox. For example, if an operating system requires 32 M memory and 512 M disk, running 3 operating system partitions requires 128 M of memory and 2 G of disk (the fourth OS partition is the partition that contains the sandboxes). Thus, it may be more efficient to restrict unprivileged application access to system resources than to provide a complete environment for each application [19]. UNIX follows this approach to control application access to system resources.

Restricting system access has the disadvantage that unprivileged applications may retain access to critical system resources. To limit this access and enforce security policies, additional system configuration is required. As application complexity leads to uncertainty in the validity of application security, system complexity leads to uncertainty of the formal validity of system security. For example, the Solaris 2.7 distribution installs over 360 configuration files in one system directory (/etc). This level of complexity requires a highly knowledgeable system administrator to maintain system consistency, let alone enforce a security policy.

With a trusted user set, the ambiguity in the system's security policy caused by the complexity in its configuration may be acceptable. However, when an application is wrapped and distributed to a set of users unknown to the host, the host needs to be protected from attacks on the application that originated from the distributed object system. Thus, the application needs to be sandboxed to limit the damage caused by an attack on the host. However, the system may not have enough available resources to support many complete virtual machines running in parallel. Thus, the protection provided by the sandbox is required without the overhead incurred by creating a complete virtual machine. If the file system and other concurrently executing applications are protected from the sandboxed application, then most of the benefits of the sandbox may be provided without the cost of creating a complete virtual operating environment.

In this limited virtual environment, the external configuration and filesystem resources required by the application must be made available to the application. An example of an external filesystem resource in some UNIX systems is shared libraries. Without access to these libraries, many applications on those systems cannot function. Thus, the applications have an implied dependencies with the library files. To make these resources available through the virtual filesystem, the originals must be duplicated to or referenced in the virtual filesystem. Even the lightweight sandbox may still incur a heavy initialization penalty by resolving the external filesystem dependencies. However, this initial penalty still does not incur the extended system resource requirements of a full virtual machine. To compensate for this initial penalty, we have designed four virtual sandbox strategies for wrapped applications:

1. Complete Encapsulation *(figure 3)*. Each application instance is given its own virtual sandbox. At the expense of high initialization time for each wrapped ap-

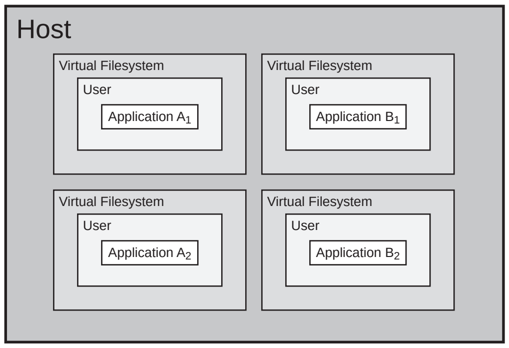


**Figure 3. Complete Encapsulation** All instances of wrapped applications run in their own sandboxes. Applications can neither directly access one another nor access the host system.

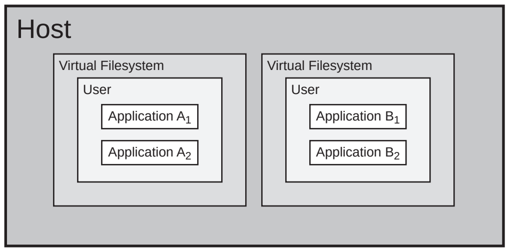


**Figure 4. Shared Sandbox** All instances of one application run within the same sandbox. Increased efficiency is gained by not recreating the sandbox for each instance of the application. While different instances of the application can interfere with one another, they are still prevented from interfering with other wrapped applications and the host itself.

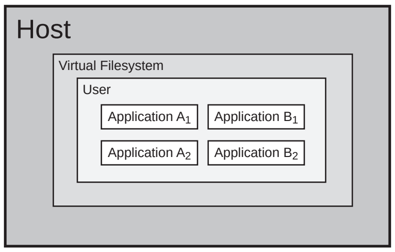


**Figure 5. Single Sandbox** All wrapped applications run in a single environment. The applications can interfere with one another, however, their access to the host system is limited by the sandbox.

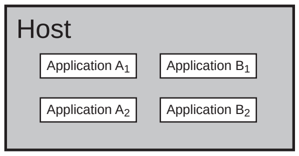


**Figure 6. No Protection** All wrapped applications run as typical user applications. They are granted full user-level access to the host system. They can interfere with both local applications and the host itself.

plication, there is a high level of protection between applications (e.g. both can modify the same temporary storage without interfering with one another). This is the mode of encapsulation proposed by the CORBA Security Services model (described in the next section) but not currently implemented in the specification.

2. Shared Sandbox *(figure 4)*. All instances of an application share the same virtual filesystem. As long as the application can be trusted to run in multiple, concurrent sessions, the benefits of protecting the system and other wrapped applications remain. The performance penalty for creating the sandbox occurs only during the first initialization of the application. A set of related applications that need to share a set of files can run within a shared sandbox, where they can share resources, yet remain isolated from other applications executing on the host.

3. Single Sandbox *(figure 5)*. There is a single, common sandbox that is used for all wrapped applications. Wrapped applications initialize more quickly than the previous two methods. However, this speed comes at the cost of more difficult configuration (the sandbox must contain configuration information for all wrapped applications). In addition, the applications must be programmed so that they will not interfere with one another. This sanboxing mode and the shared sandbox mode of encapsulation were not proposed in the CORBA Security Services model.

4. No Protection *(figure 6)*. Wrapped applications run as full user-level applications with the full filesystem access granted to their user and group identities. In effect, wrapped applications are equivalent to those that are run by local users. None of the performance penalties associated with creating the sandbox are incurred. However, the wrapped applications can interfere with one another. For example, one application running as the unprivileged user ‘nobody’ is able to interrupt or modify the execution thread of another application

running as ‘nobody’ (as a debugger is able to trace and alter another process running in parallel with the debugger). Thus, this mode of execution should only be used in environments with controlled external access (for it places the application in the security domain mentioned in *figure 1*).

In the Legacy Wrapper model, we choose to allow users to select one of the four modes of execution. In more secure environments, the first two methods are available. In trusted environments, the last two methods are available (though they are not recommended to be used in an open environment). In the current implementation of the Legacy Wrapper, we have only implemented the first method, complete encapsulation. However, soon all four methods will be available in the implementation.

## 4. The Legacy Wrapper Tool

The Legacy Wrapper is designed to be a generic object wrapper. This wrapper meets the criteria of security and ease of use described in the previous sections.

The Legacy Wrapper can be thought of as three components: the Legacy Client and Legacy Server with a distributed object system linking the two. Next, we describe the rationale behind the selections of the platforms for each of these three components beginning with the distributed object system.

### 4.1. Distributed Object System

For portability and integration with current object technologies, the Legacy Wrapper should use a publicly available distributed object technology. Briefly, the distributed object system should meet the following criteria:

- *Cross-platform:* The clients and servers can run on a variety of platforms (particularity Java, UNIX, and PC clients).

- *Heterogeneity:* To ease the implementation of the system on multiple platforms, the object system would need to provide clients and servers for multiple operating systems and software written in various programming languages.

- *Availability:* The distributed object system should be readily available to all clients and servers. An open source-based system would be preferable, because it would enable us to reference a known system with which the wrapper would work.

A review of the literature describes OMG’s CORBA as a reasonable technology for wrapping legacy applications [20] [10]. In addition, CORBA is available for many versions of UNIX, NT, Java, and other platforms. Its governing consortium, the Object Management Group, is comprised of over 700 organizations [9], meeting the standards-based criteria. Finally, it is available in such products as the JavaIDL from Sun for the Java 1.2 platform and MICO CORBA for POSIX based systems [12].

Security services are available through the CORBA Security Services Specification [8]. The services provide strong user identification and session control. The specification also includes a provision for the creation of a virtual machine per object invocation (meeting the criteria of the complete application encapsulation, c.f. *figure 3*). However, implementation of this service is left for a future version of the security specification. In addition, the CORBA Security Specification includes no provisions for the shared and single sandbox modes of execution included in the legacy access model. Thus, the Legacy Wrapper must provide the sandbox for wrapped applications, while CORBA Security Services provide the transport-level encryption and external user authentication.

After we selected CORBA as the distributed object system, our development focused on the selection of server and client platforms and the implementation of the Legacy Wrapper system.

### 4.2. Legacy Wrapper Server

The Legacy Wrapper Server must provide the wrapping and encapsulation services described above. Since the wrapper is designed to support existing legacy applications, it must be designed to use a standard operating environment. To wrap the applications as described in the Legacy Wrapper model, the following services are required of the operating system that the legacy application runs on: a concept of multiple users, user-based access control (i.e. RBAC), multiple concurrent applications, and a coherent, rooted filesystem. POSIX-compliant operating systems provide all of these services [6], Hence, the Legacy Wrapper server is designed to be implemented on a POSIX-compliant operating system. Current POSIX-compliant operating systems include Sun’s Solaris, Linux 2.0, Windows NT, Open VMS, and many others.

By lightweight sandboxing of an application, we mean running it in a virtual filesystem with its own set of system privileges assigned specifically to that application. Thus, this lightweight sandbox requires an independent filesystem and RBAC access controls (c.f. *figures 3–5*). POSIX (IEEE standard 1003.1) provides three mechanisms for limiting system access to users and application’s [6]. When executed by a privileged process, `setuid()` changes the privilege level of the process to the privilege set that is granted to a specific user. Similarly, `setgid()` changes the privilege level to that of a group of users (implementing a form of RBAC). As `setuid()` and `setgid()` restrict an application's system privilege level, `chroot()` restricts the application's access to the file system by creating a virtual file system that starts within a branch of the original file system (e.g. if `/` is the top of the filesystem, `chroot(/tmp/root)` is called for an application, `/tmp/root` becomes `/` for that application).

Using `setuid()` and `setgid()` with `chroot()` controls the application's access to most system services (i.e. the only available system services are those specifically given to the application). However, applications retain access to system library functions that were available at the time of the compilation (e.g. network access). Restricting access to these functions leaves three options that do not require modifications to the application's source code:

1. Recompile the legacy application with a new limited-functionality system library.

2. Create a library that is binary-compatible with the original system libraries with which the legacy application was originally created.

3. Use available, but not standardized (i.e non-POSIX) operating system-based user limits to prevent these accesses.

The first two options require creating new system libraries or modifying existing libraries. Because of their inherent complexity and dependence upon internal system structures, reverse engineering these libraries would be impractical (for example, reviewing the over 540,000 lines of code that comprise the GNU project's libc, which has the source code freely available). Hence, by creating limited versions of these libraries, the wrapper would be prevented from meeting two of its primary goals: system and application independence.

Within their system libraries, some UNIX variants provide mechanisms to control the behavior of running applications. For example, AT&T System V Release 4 and BSD 4.3 support `rlimit()` and `quotactl()` which can limit the processor usage, disk usage, and the number of open files per user. Generally restricting an application's ability to make network connections is highly system-dependent, thus, it is currently not automated by the Legacy Wrapper.

Hence, the Legacy Wrapper can provide a lightweight sandbox for existing legacy applications using POSIX services with CORBA as the underlying distributed object system. The applications do not require modification, but administrative awareness is required to verify that applications do not attempt to create denial of services. In the next section, the legacy client will be described.

## 4.3. Legacy Wrapper Client

The Legacy Wrapper client is an API (Application Programmer's Interface) designed to interface with the remote legacy server. For it to be a practical addition to user applications, it must meet the following criteria:

- *Platform-independence:* To achieve maximum usability of the client, the API should be available for a large number of platforms.

- *Compactness:* Since the API is designed to be included in third-party applications (or user applications with respect to the API), it must not add significant overhead to the applications. That is, there should only be a small additional speed and memory overhead for including the client in an user application.

- *Transparency:* The user application should not need to know how the client communicates with the server, only that the application must follow the given API to access the external application object.

- *Simplicity:* The API should not be cluttered with a large interface that may confuse application programmers. Thus, API calls should be logically grouped and have no unintended side effects.

Since Java includes both compactness and platform-independence, it was chosen as the implementation language. In addition, in Java 1.2, Sun includes the JavaIDL ORB, a CORBA implementation. Thus, with Java 1.2, we had platform-independence, compactness, strongly-typed object-orientation that would let us oblique the Client's interaction with CORBA all included within the language itself [16].

Next, the API needed to be designed to hide its interaction to the remote application object through CORBA. Since the remote application input and output is streamed, the API was designed to model the remote application I/O as local file I/O. Thus, to send data to the application was as simple as writing to an input stream, and receiving data from the application required reading from an output stream.

In contrast, if the application user interface wanted to access the legacy server directly, it would need to be aware of several low-level implementation details of both CORBA and the Legacy Server:

- initialization state of the JavaIDL ORB and its naming context (so that it can resolve the names of CORBA servers including the Legacy Server),

- whether or not the client's version of the Legacy Server CORBA interface definition is accurate and up to date (otherwise, it will not be able to communicate with the server),

- Separate mechanisms for dealing with low-level details of the multiple application output streams. For example, handling data underflow when the server cannot send enough data to satisfy a read request and data overflow when the user application wants to send more data than the legacy application can accept.

However, when a user application wants to access a remote application through the Legacy Client, it follows a three step process:

1. Ask the Legacy client to start the application.
2. Ask the Legacy client for the application input, output, and error streams.
3. Read from the application output and error streams and write to the application input stream.

For both ease of use and consistency with the Java 1.2 API, we subclass the `InputStream`, `OutputStream`, `InputReader`, `OutputWriter` classes. Thus, the user application can use familiar methods to access the remote application through our subclassed Java streams.

As with the Legacy Wrapper server, the goals of the Legacy client remained the same: a clean, consistent module that lets the user access the remote application. It does this through a small API that follows the conventions in the Java standard libraries after the initial setup that selects the remote application.

### 4.4. Complete Client-Server Model

Now that we have described the details of the Legacy Client and Server, we describe how they integrate to become the Legacy Wrapper *(figure 7)*. The client (user) works with an application-specific user interface. When the user interface wants to send data to, or receive data from, the legacy application, it communicates with the Legacy Client. The Legacy Client packages the request and sends it to the remote Legacy Server. Communication between the Legacy Client and Server objects is handled by the JavaIDL and Host ORBs. When it receives the request, the Legacy Server unpackages it and passes it through the Legacy VM (the sandbox described earlier) to the application.

Now that we have described the Legacy Wrapper as an abstract tool, we continue in the next section with an example of its use in the wrapping of a legacy graph visualization tool, called dot.

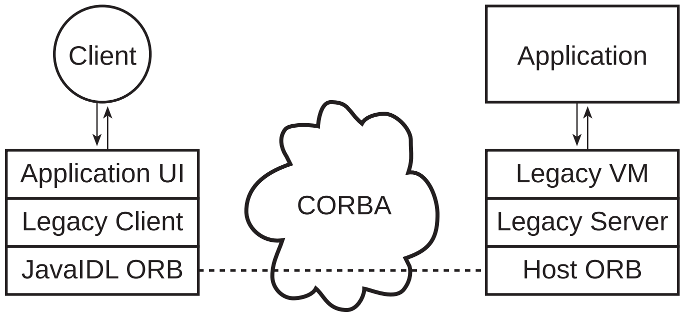


**Figure 7. Complete Legacy Wrapper Layered Model** The client's (left) interface into CORBA distributed object system is divided among three principle sections: the application user interface, the Legacy Client, and the JavaIDL ORB. The server's (right) interface into CORBA is divided among the Legacy VM, the Legacy Server, and the Host ORB. Low-level network communication is coordinated through the ORBs. The Legacy Client and Server package application data to allow it to flow between the ORBs. The Legacy VM creates the sandbox for the legacy application.

## 5. Distributed Graph Visualization Service

### 5.1. Background

The dot graph visualization tool [3] creates nicely formatted partitioned graphs such as those produced by Bunch [7], a reverse engineering tool created by our research group.

As input, dot uses a text graph description file. It contains descriptions of the nodes in the graph including style information (e.g., a "ball" is a circular, red node). Similarly, edges link node pairs (e.g., green ball $\rightarrow$ red ball), and they may also contain style attributes.

While dot produces nicely formatted graphs, it has the side effect of creating an external dependency to Bunch. Rather than recreating the functionality of dot, we needed to securely distribute its services to Bunch users.

Thus, dot is an example of an application that needs a secure, generic wrapper. We need to distribute dot's services to a large set of known and anonymous thin clients (Bunch users). Considering that the clients may be anonymous, dot needs to be protected from malicious users.

### 5.2. Distributed dot Implementation

Once the Legacy Server had reached its alpha version, we began working on the Legacy Client. During initial testing of the server, a small test C++ client program was developed to test the server locally. Through the local client, communication was successful.

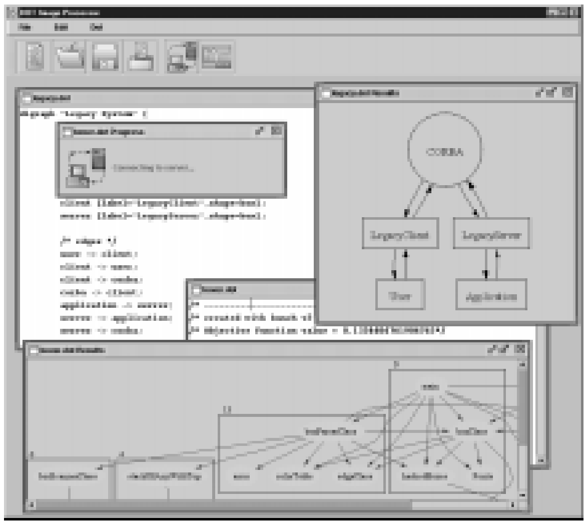
Next, we created the Legacy Client API. The API encapsulates the functionality described earlier. To test the API, we created a small Java program that includes a text editor *(figure 8)*. Graph description files are loaded (and written) in the editor. When the graph description is ready, the user selects a menu item (Dot → Process) that sends a message to the Legacy Client to create an instance of the application.

The Legacy Server creates a complete encapsulation style sandbox for dot. First, it verifies that the client has the proper privileges to use the application. If the client does not have the proper privileges to run dot, or any other permissions error occurs, an “Access Denied” error code is returned to the client so that a malicious user cannot use a more descriptive error code as a subtle means to attack the system. If the client is permitted to run dot, the Legacy Server starts creating a new sandbox for dot. It creates a temporary directory that will hold the virtual filesystem. In the virtual filesystem, it places references for all of the configuration, library, and executable files needed by dot (including the application itself). Then, it creates data pipes that will be used to read from and write to the application. Next, it `fork()`’s to create a new process. In the new process, it `chroot()`’s to the root of the virtual filesystem. It uses `setgid()` and `setuid()` to change to permitted role set of the application. Finally, the dot application is executed in the child process.

Once the application instance is created, the client sends the graph description to the legacy application using the Legacy Client output stream. The legacy application (dot) processes the graph description and produces a gif image of the graph. Then, the client waits for the image data. Finally, when all of the data are received, the Java application displays the image, and it asks the Legacy Server to shutdown its instance of the dot application (it cannot request for any other application instance to be shut down).

When dot completes, or the authenticated client asks for its instance of application to be shutdown, the Legacy Server kills the application process. Then, it frees up the filesystem space that was taken by the virtual filesystem for that instance of dot.

Thus, through the Legacy Client, the services of the dot application (over 70,000 lines of source code) were accessed. The largest portion of the Java application is user interface code. A dozen lines of source code are needed to communicate with the Legacy Client class instructing it to create a graph from the user’s graph description (this is approximately equivalent to the number of lines of C source code required to run the dot application as a subprocess on the host itself).



```
digraph "Legacy System" {
client [label="LegacyClient",shape=box];
server [label="LegacyServer",shape=box];

/* edges */
user -> client;
client -> user;
client -> corba;
corba -> client;
application -> server;
server -> application;
server -> corba;
```

**Figure 8. Distributed Dot Tool** The Legacy client was used to connect to a remote server which wrapped the dot graph visualization tool. On the right and bottom of the image are the two graphs produced by the files shown on the top-left and center of the image, respectively. The small pop up window (inset from the top-left) displays a new connection to the Legacy server.

## 6. Conclusion and Future Work

The Legacy Wrapper is designed to address a current need for a secure generic distributed object wrapper that secures the intersection between the distributed object model and host security. Through the wrapper, application services are distributed in a secure manner so that neither malicious applications can use the host as a means to attack the distributed object system nor can malicious users can use the wrapped application as an entry point to attack its host.

The Legacy Client and Server are written to be portable. The server follows the POSIX standard C++ library calls and it uses the freely available MICO ORB. Similarly, the Legacy Client uses the freely available JavaIDL ORB from Sun. Portability is achieved through the use of standard libraries and cross platform programming languages.

Future work on the Legacy Wrapper will be in four areas:

- Implement the complete legacy access model.
- Better integrate with the CORBA Security model.
- Provide better service support.
- Provide a more complete implementation of the Legacy wrapper model.

Currently, the server only supports the per application instance method of sandboxing. However, during testing, the overhead incurred by recreating the sandbox for each application instance became apparent. In the future, we will implement the shared and single sandbox models on the server.

However, it is unlikely that we will implement the no sandbox model, because of its inherent security flaws.

The earlier version of the MICO ORB that was available at the beginning of development did not provide full support for CORBA Services. Thus, the server has little current integration with CORBA domain security and favors its internal security instead. As both MICO and the Legacy Server evolve, the server will contain better support for CORBA domain security.

As the legacy access model evolves, it will be able to provide a more comprehensive evaluation of the denial of service threats to the host computer.

In this paper, we have described the Legacy Wrapper, a model of its function, and how the tool was used to integrate a legacy application into a distributed system. For more information, please visit the Legacy Wrapper's web page on the Drexel University Software Engineering Group (SERG) web page at http://serg.mcs.drexel.edu/legacy.

## 7. Acknowledgements

This work is sponsored by a CAREER Award from the National Science Foundation (NSF), under grant CCR-9733569. Additional support was provided by a grant from the research laboratories of AT&T.

Any opinions, findings, and conclusions, or recommendations expressed in this paper are exclusively those of the authors and do not necessary reflect the views of the NSF, the United States government, or AT&T.

All copyrights and trademarks are the property of their respective owners.

## References

[1] M. Aldrich. Secured systems and Ada: A trusted system software architecture. In *Proceedings of the 1994 Conference on TRI–Ada '94*, pages 282–292, New York, NY, 1994. ACM.

[2] E. English and S. Hamilton. Network security under siege: The timing attack. *IEEE Computer*, 29(3):95–97, Mar. 1996.

[3] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–231, 1993. http://www.research.att.com/sw/tools/graphviz.

[4] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr. Dobb's Journal*, pages 66–70, Jan. 1996.

[5] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, Sept. 1994.

[6] D. Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, 1991.

[7] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings Sixth International Workshop on Program Comprehension*, Ischia, Italy, 1998. IEEE Computer Society Press. http://www.mcs.drexel.edu/~serg/Projects/Bunch.

[8] Object Management Group, Farmingham, MA. *CORBA Services Specification 1.2*, 98-12-09 edition, 1998. http://www.omg.org/corba/sectran1.html.

[9] Object Management Group, Farmingham, MA. *CORBA/IIOP 2.2 Specification*, 98-01-07 edition, 1998. http://www.omg.org/corba/corbaiiop.html.

[10] F. Olken, H.-A. Jacobsen, C. McParland, M. Piette, and M. F. Anderson. Object lessons learned from a distributed system for remote building monitoring and operation. In *ACM SIGPLAN Notices* [20], pages 284–295.

[11] D. L. Parnas. Software aging. In *Proceedings of the 16th Annual International Conference on Software Engineering*, Sorrento, Italy, 1994. IEEE. Keynote Address.

[12] K. Römer, A. Puder, and F. Pilhofer. *MICO Is CORBA*. University of Frankfurt, 2.2.6 edition, 1999. http://www.mico.org.

[13] S. Rugaber and J. White. Restoring a legacy: Lessons learned. *IEEE Software*, 15(4):28–33, July–Aug. 1998.

[14] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, July 1983.

[15] R. S. Sandu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[16] Sun Microsystems, Palo Alto, CA. *Java® 2 Platform API Specification*, 1999. http://www.javasoft.com/products/jdk/1.2/docs/api/.

[17] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, Aug. 1984.

[18] University of California, Berkeley. *BSD Unix 4.2*, 1983.

[19] R. Wahbe, S. Locco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS O/S Review*, 27(5):203–216, Dec. 1993.

[20] E. Wallace and K. C. Wallnau. Situated evaluation of the object management group's (OMG) object management architecture (OMA). *ACM SIGPLAN Notices*, 31(10):168–178, Oct. 1996.