IDA

INSTITUTE FOR DEFENSE ANALYSES

Analysis of Secure Wrapping Technologies

Reginald N. Meeson, Task Leader

John M. Boone
Karen D. Gordon
Terry Mayfield
Edward A. Schneider

February 1997

IDA Paper P-3297
Log: H 97-000119

This work was conducted under contract DASW01 94 C 0054, Task A-202, for the Defense Advanced Research Projects Agency. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

PREFACE

This document was prepared by the Institute for Defense Analyses (IDA) under the task order, Secure Wrapping Technology, in response to a task objective to develop a more complete understanding of the types, extent, and performance impact of information survivability protection that can be provided for legacy and commercial off-the-shelf software products. This work was sponsored by the Defense Advanced Research Projects Agency.

The following IDA research staff members were reviewers of this paper: Dr. Edward A. Feustel and Dr. Richard J. Ivanetich.

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

PURPOSE AND SCOPE

This study was undertaken to develop a more complete understanding of information survivability protection that can be achieved by applying "wrapping" techniques. Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that requires no modification of those products. This study investigated the types, extent, and cost-effectiveness of wrapper protection for mission-critical computer systems.

BACKGROUND

Because of the DoD's growing dependence on computer systems and the increase in communication between these systems, there is an increasing exposure and vulnerability to inadequate security. The Defense Advanced Research Projects Agency (DARPA) is developing the technology to protect mission-critical DoD systems against electronic attack on or through their supporting computing infrastructure. Examples of types of attack include unauthorized entrance to systems, interception of critical data in transit, disruption of network services, and corruption of data in repositories or in transit. This protection technology must provide high assurance and measurable security at an affordable price.

A key to making computing systems affordable is the use of COTS components. Former Secretary of Defense William Perry made the use of COTS computer products, including software, a major DoD cost-saving and effectiveness-improving strategy. COTS products, however, are not normally developed to DoD-level security standards and do not meet DoD needs for robustness in critical systems.

A concept advanced at a DARPA Summer workshop to deal with COTS software and survivability was to encapsulate these products within "wrappers" that would monitor and control the effects they were allowed to have on other parts of the system. Investigations of specific wrapping techniques, the weaknesses against which wrappers can provide protection, the degrees of protection that can be achieved, and the cost-effectiveness of wrapping techniques, however, have not yet reported results.

APPROACH

This study proceeded by conducting a series of analyses to define survivability properties, specify the architecture of systems containing COTS components, specify protective wrappings for the COTS components, and analyze the systems' survivability characteristics with and without the protective wrappings. The initial experiment was to specify the properties necessary for a system to survive intrusion attempts and analyze the effects of wrapping key components with authentication services.

DEFINITION OF WRAPPER

A wrapper consists of two parts:

- An *adapter* that provides some additional functionality for an application program at key external interfaces, and
- An *encapsulation* mechanism that binds the adapter to the application and protects the combined components.

INTERFACES AND INTERPOSITION MECHANISMS

Four different interfaces between application programs and their supporting environment were identified as the most accessible insertion points for adapters: library services, operating system services, services provided by separate processes, and services provided by external processes outside the local software environment such as network proxy services. Each of these interfaces provides a different opportunity to intercept and check transactions that applications might attempt to execute. Interfaces that would make better insertion points for adapters may exist internally within an application, but their use is generally not feasible without access to source code and sufficient design information.

SURVIVABILITY PROTECTION MECHANISMS

The principal mechanism investigated in this study is authentication, which allows, for example, client and server applications to verify each other's identity before processing transactions. Other mechanisms that are considered potential candidates for wrapper implementation include: logging and auditing, constraint checking, encryption, access control, fault detection and recovery, and redundancy.

ARCHITECTURE SPECIFICATIONS

A secondary objective of this study was to assess the use of architecture description language (ADL) technology. An example client-server architecture was specified and then

extended to introduce adapter functions that provide authentication services. Two different authentication protocols, Kerberos and public key, were modeled as plug-replaceable components of this architecture.

FINDINGS

Our results show that authentication adapters can be added to client and server applications by substituting library software. Many other survivability functions appear to be implementable with this approach, including those listed above. Adding adapters at lower level interfaces, for example by intercepting operating system calls, may be more difficult because high-level transactions are typically made up of many low-level operations. The difficulty arises in trying to identify high-level transactions, some of which may contain anomalies, within interleaved sequences of low-level operations. Low-level interfaces must be used when higher-level interfaces are hidden within the application and are not externally accessible. While some adapter functions can add significant overhead and impact application performance, wrapper mechanisms appear to add little or no further overhead beyond that of the adapter.

The protecting encapsulation part of available wrapping mechanisms was found to be weak. That is, we found a number of situations where adapters could be bypassed or disconnected from the applications they were supposed to protect.

Architecture description languages work quite well for modeling the system interfaces and connections needed to introduce wrappers. One pitfall we found, though, is that inaccuracies in the interface specifications of legacy or COTS software such as undocumented debug modes can easily conceal security holes. We also found an interesting trade-off between "tight" and "loose" interface specifications. Tight specifications make analysis of properties easier but they restrict the range of plug-compatible components. Loose specifications extend the range of plug-compatibility, increasing potential reuse, but allow connection of incompatible or only partially compatible components. We found the need for both tight and somewhat relaxed or generalized interfaces in modeling our authentication adapters.

A number of open system standards for distributed computing include authentication services and are now incorporating "hooks" for additional services to be provided by plug-in adapter modules.

CONCLUSIONS

Several techniques for wrapping legacy and COTS application software are available and have been employed in commercial products. Practical aspects of wrapping such as the accessibility of interfaces, accuracy of interface specifications, and protecting adapter code from being bypassed, however, are currently open problems that will need to be addressed to enable broad use of these techniques for security and survivability hardening. At present, normal software maintenance processes can probably provide higher assurance for the functions considered to be placed in wrappers.

Architecture Description Languages (ADLs) worked well for modeling system interfaces and adapter functionality. Two questions for the architecture description community were raised, however. One is how to deal with an application program's interfaces at multiple levels of abstraction. The other is how to resolve the issues between "loose" reusable interface specifications and "tight" secure specifications.

Solutions to these questions would enable wrapping techniques to provide valuable security and survivability hardening for important segments of the DoD's software inventory.

1. INTRODUCTION

1.1 PURPOSE AND SCOPE

This paper presents the findings and conclusions from a study undertaken to develop a more complete understanding of information survivability protection that can be achieved by applying "wrapping" techniques. Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that requires no modification of those products. This study investigated the types, extent, and cost-effectiveness of wrapper protection for mission-critical computer systems.

1.2 BACKGROUND

Because of the DoD's growing dependence on computer systems and the increase in communication between these systems, there is an increasing exposure and vulnerability to inadequate security. The Defense Advanced Research Projects Agency (DARPA) Information Technology Office (ITO) Information Survivability Program is developing the technology to protect mission-critical DoD systems against electronic attack on or through their supporting computing infrastructure. Examples of types of attack include unauthorized entrance to systems, interception of critical data in transit, disruption of network services, and corruption of data in repositories or in transit. This protection technology must provide high assurance and measurable security at an affordable price.

A key to making computing systems affordable is the use of COTS components. Former Secretary of Defense William Perry made the use of COTS computer products, including software, a major DoD cost-saving and effectiveness-improving strategy. COTS products, however, are not normally developed to DoD-level security standards and do not meet DoD needs for robustness in critical systems. The Defense Goal Security Architecture (DGSA) recognizes this problem* but the means to solve it completely do not yet exist.

A concept advanced at the 1995 Information Science and Technology (ISAT) Summer Workshop to deal with COTS software and survivability was to encapsulate these

---

* Defense Information Systems Agency (DISA). *Technical Architecture Framework for Information Management (TAFIM), Vol. 6, DoD Goal Security Architecture.* in particular, Section 8.3. (see http://www.itsi.disa.mil/cfs/tafim.html)

products within "wrappers" that would monitor and control the effects they were allowed to have on other parts of the system. It was further suggested that multiple wrappers providing different types of protection might be composed, allowing system components to be hardened selectively against different types of attack. Investigations of specific wrapping techniques, the weaknesses against which wrappers can provide protection, the degrees of protection that can be achieved, and the performance impacts of wrapping techniques, however, have not yet reported results.

Analyzing the survivability properties of a system constructed with COTS components requires methods for defining those properties, describing the system's structure, and specifying the relevant characteristics of its components. Formal techniques for specifying and analyzing security properties such as data integrity and confidentiality of information are readily available. Such formal techniques extend naturally to provide a basis for specifying and analyzing additional survivability properties such as availability of service. A system's structure, component interfaces and behavior (including interfaces and behaviors for COTS products and wrappers), and constraints on interactions among components can be specified formally using DARPA-developed Architecture Description Language (ADL) technology. Robust systems will have direct correspondences between their required survivability properties and their ADL specifications. Lack of such correspondence should be readily analyzable to identify system weaknesses and suggest potential solutions.

1.3 APPROACH

This study proceeded by conducting a series of analyses to define survivability properties, specify the architecture of systems containing COTS components, specify protective wrappings for the COTS components, and analyze the systems' survivability characteristics with and without the protective wrappings. The initial exercise was to specify the properties necessary for a system to survive intrusion attempts and analyze the effects of wrapping key components with authentication services. The steps in this process were:

a. Identify particular wrapping mechanisms and techniques by which wrappers can be applied to COTS and other software products. These techniques should allow COTS products, for example, to be used "straight out of the box" and wrapped and installed in a system with no modification.

b. Identify and formally specify a collection of survivability properties relevant to the protection provided by authentication services. These properties were derived by identifying potential attacks or faults in system components that, if allowed to penetrate or exist and propagate, would allow unknown entities to access information or use system services.

c. Specify the architecture of a small, representative system requiring authentication services for protection using an appropriate ADL, highlighting the interfaces, interconnections, and behavior constraints of COTS and other system components.

d. Determine how and where wrappers might be employed to achieve improved survivability using the architecture specification and the set of desired survivability properties, then specify the necessary wrapper interfaces and behavior in the selected ADL.

e. Assess the level of added protection achieved with the wrapping(s). Identify any remaining weaknesses and, in particular, any new weaknesses that may have been introduced by the wrapping.

f. Identify any shortfalls in the selected ADLs that restrict the ability to capture and analyze desired system survivability properties.

Subsequent analyses added to this collection of information by specifying new required information survivability properties and protection mechanisms (selected in conjunction with the DARPA Program Manager), revising the architecture specifications, repeating the wrapping and protection analyses, and updating our ADL experience notes.

1.4 DEFINITION OF WRAPPER

The concept of wrapping was only vaguely defined by the ISAT working group. To help crystallize our thinking about wrappers and wrapping we have come up with the following definition. A wrapper consists of two parts:

> An *adapter* that provides some additional functionality for an application program at key external interfaces, and
>
> An *encapsulation* mechanism that binds the adapter to the application and protects the combined components.

In addition, wrapping should require no changes to the application program. In this paper we consider the use of adapters to provide security and survivability functions.

Figure 1 illustrates a wrapped application program. The original program uses its environment's interface to a network. We have introduced an adapter, which is plug compatible with the network interface, and have interposed it between the application and the network service. The encapsulation part of the wrapper assures that the adapter stays attached to the application and that the adapter's network-side interface is the only interface accessible from the network. This definition can be easily broadened to include multiple adapters within one encapsulation boundary. Adapters may connect multiple application and environment interfaces.

In general, we expect the encapsulation mechanism to bind adapters to individual application programs. It may also be feasible to bind an adapter to a service interface, making the adapter's protection available to all applications that use the service via that interface. Some service interfaces appear difficult to wrap for all applications, however, such as an operating system's file system interface.

The functionality within an adapter is illustrated by the data flow diagram in Figure 2. Transactions may be initiated by either side of an adapter. The application program plugs into the adapter's service socket (on the left) and the adapter's application plug (on the right) plugs into the original service socket. In the ADL literature these plug and socket interfaces are called *duals* of each other. Interfaces are typically defined in terms of the application's view. The service then simply provides the dual.

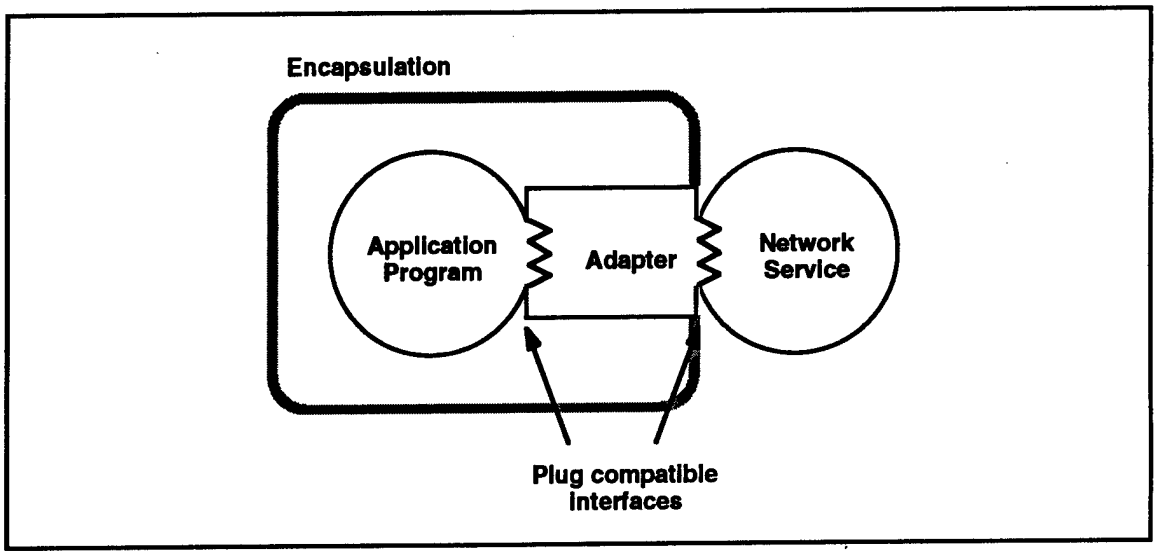

Figure 1. A Wrapped Application Program

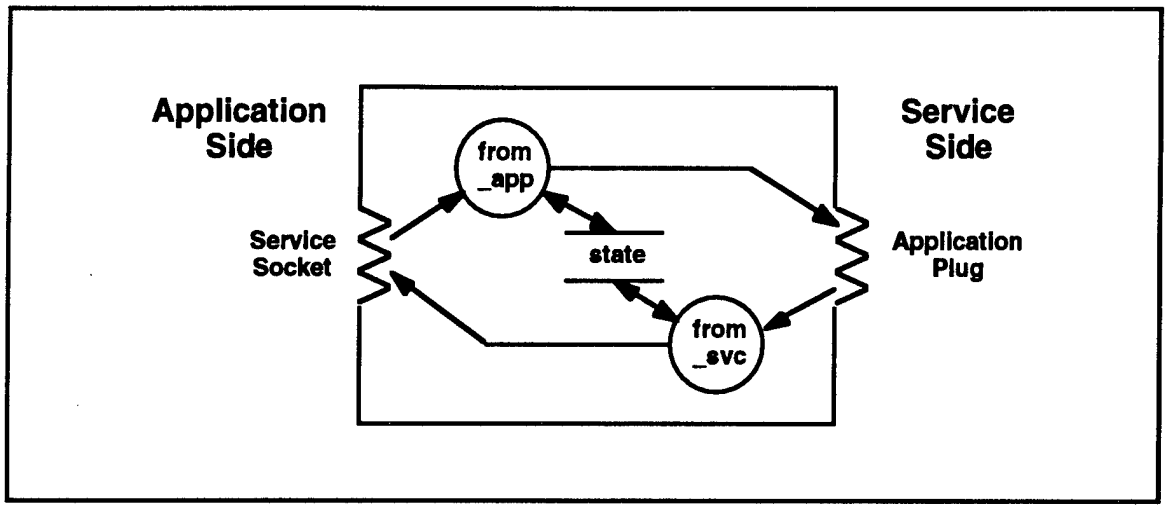

Figure 2. Data Flow of Functionality Within an Adapter

The function from_app handles information passed from the client application to the service. This processing may involve or be controlled by state information maintained within the adapter or in the surrounding environment. For example, the current time of day is considered state information maintained by the operating system. The function may also update the adapter's state and the surrounding environment; for example, by writing to a log file. Likewise, the function from_svc handles information passed from the service to the client. Both adapter functions may share common state information.

1.5 ORGANIZATION OF PAPER

Section 2 discusses the principal mechanisms available for attaching adapter code to application programs and encapsulating the result. Section 3 describes the survivability properties we have considered as possible uses for wrappers. Section 4 presents a simple client/server architecture that has provided a practical context for our analyses. Section 5 presents our findings and Section 6 presents our conclusions.

2. INTERFACES AND INTERPOSITION MECHANISMS

Four different interfaces between application programs and their supporting environment were identified as the most accessible insertion points for adapters: library services, operating system services, services provided by separate processes (including graphical user interface services), and services provided by external processes outside the local software environment such as network proxy services. Each of these interfaces provides a different opportunity to intercept and check transactions that applications might attempt to execute. Interfaces that would make better insertion points for adapters may exist internally within an application, but their use is generally not feasible without access to source code and sufficient design information.

2.1 LIBRARY SERVICES

Application programs typically use libraries to provide more meaningful, higher-level abstractions of basic system services. These interfaces are often strongly typed, which provides a form of protection and makes analysis of correct usage considerably easier. A prime example of a library service for distributed computing is the remote procedure call (RPC), which allows one program to communicate with another program on another computer as if the second program was a local subprogram. The RPC library turns the first program's remote procedure calls into messages that transfer parameters, invoke the requested operation on the second computer, and then return the results. Another example at a more primitive level is the sockets library, which provides basic transport layer services for setting up network connections and sending and receiving messages.

Several modern operating systems provide late binding of library services. In these systems, libraries reside in the user's local environment. Among other advantages, this allows library functions to be changed without modifying the application programs that use them. For example, the RPC library can be substituted by a new library of adapter functions that have exactly the same callable interfaces. The adapter library can make additional checks for security and survivability protection, and then call the original RPC library to complete those transactions that pass all the checks. If a check fails, the adapter can block the operation or it can log the event and let the operation proceed.

Depending on how library entries are resolved, extra steps may have to be taken to ensure that substitute adapter libraries cannot be bypassed. For example, the substitute library may have to provide "stubs" for all of the original library's functions to guarantee a complete interface, even though these stubs include no adapter functions and immediately hand off the call to the original library entry. The overhead for the additional level of procedure call compared to most library function execution times is not expected to be significant, however, so no changes to an operating system's library entry resolution mechanism should be necessary.

Conventional library linking mechanisms copy library code into executable load modules at program generation time and provide no easy way to modify or substitute library code afterwards. The library call interface is not accessible for insertion of adapters on such systems after program generation.

2.2 OPERATING SYSTEM SERVICES

Operating system calls provide a lower level interface than most library calls. The operating system is a program's most direct interface to basic machine resources such as memory and hardware interfaces, the file system, inter-process communications, and network communications. A single high-level library call may expand into numerous operating system calls.

Operating system calls are typically implemented by hardware "traps" which interrupt normal program execution and transfer control to operating system code. Most operating systems provide mechanisms for redirecting these low-level system traps. Program debugging and tracing tools are probably the most familiar programs that use these mechanisms.

Installation of an operating system call adapter is accomplished by substituting a new trap handler in place of the one provided by the operating system. Each operating system has a slightly different mechanism for this, but typically it involves another system call. The address for the new handler must be supplied as a parameter and the address of the original handler is returned as a result. The next time the trap is triggered, control will be transferred to the new handler.

Typical operation of the new handler is to execute the adapter functions and then pass control to the original handler for normal execution. This mechanism allows adapters to be stacked. The application remains suspended while this processing takes place and will not be able to proceed until the trap handlers return control to it. If the adapter code

detects a discrepancy in the requested operation it can take a number of possible actions, from logging the event and letting the operation proceed to blocking the operation. It can also coordinate with a separate adapter watchdog process that maintains state information and can monitor sequences of an application's system calls. This coordination is often necessary because many useful transactions require execution of multiple system calls.

Another operating system interface is the top-level command interpreter or "shell" from which users are able to launch application programs. In many operating systems the shell is able to pass parameters such as file names and program control flags to an application on start-up. Applications may also return a status code to the shell upon completion. A shell script that screens various combinations of parameters and flags before launching an application, for example, would serve as an adapter between the shell and the application.

2.3 SERVICES PROVIDED BY SEPARATE PROCESSES

Another mechanism for providing library-like services is to place the functions in separate processes that run concurrently with an application program. Applications communicate with such processes through a normal library interface that turns function and procedure calls into inter-process communications (IPC) calls. One service process may serve multiple client applications. The interfaces to service functions can be somewhat more tightly controlled because transactions have to pass through the operating system's IPC service, which may help to enforce a defined set of interface specifications. In addition, separate processes may execute with different access privileges than those of client applications, which can be used to provide applications limited extended capabilities without giving away normal access constraints.

An example service used by many application programs is the X-Windows* graphical user interface system, which provides convenient point-and-click user controls and input. The X Window System includes both a library of high-level display and input services, and a separate server process that provides direct interaction with the user's display(s) and input devices. Other applications then run as clients of the X server, interacting through a well-defined IPC interface. Typically, the X server runs on the user's workstation. X clients may run locally as processes on the same platform as the X server or remotely on other platforms. (A little confusion is sometimes created because X client

---

* cf. Young, Doug. *The X Window System: Applications and Programming with Xt (Motif Version)*, Prentice Hall, 1989. (see also, http://www.x.org/consortium/x_info.html)

applications often run on server platforms.) X servers can typically handle a number of clients at one time. Depending on how the X services are implemented it may be possible to attach adapters and intercept transactions at three different locations: the high-level library calls, the IPC calls, and the remote communications calls.

X-Windows server processes typically run independently of their clients. Other services may need to have processes started each time the application is run. This can be accomplished by a start-up process that spawns the server processes, sets up the necessary IPC connections, and then launches and transfers control to the application. An essential part of the wrapping process is to control how and where these connections are made.

A special case of a start-up process is the user's login process. This privileged process typically authenticates the user's login information, sets the user's access privileges, sets up environment parameters based on the user's profile information, and then launches and transfers control to a command line interpreter or "shell" process. The normal login process can be augmented by inserting another process to collect additional information from a user before transferring control to the normal login process. This mechanism can be used, for example, to substitute different login names for users based on the roles they intend to take. Since different login names can carry different privileges, users would effectively be given access privileges based on the roles they have specified.

2.4 EXTERNAL NETWORK SERVICES

A more distant interface is that provided by an external network proxy server, the most common example of which is a network firewall. Local area networks often have a designated computer that provides the actual physical connection to an external wide-area network service and shares this resource with the other local computers. Since all messages to and from the wide-area network pass through the firewall, it provides a central location for checking and screening all remote transactions for the entire local organization. Placing an adapter within a firewall is an example of binding the adapter to a service rather than an application.

Firewalls play a key role in limiting access to local machine resources and information by unknown users on the external network. For example, an organization may set up a single server for transferring files to and from outside users. The firewall would be set up to allow file transfer requests from the external network addressed to this server to pass through to the local network. Outside file transfers addressed to any other local machine would be rejected by the firewall. File transfer requests initiated by a local

machine to an external address would be allowed to pass unimpeded through the (local) firewall. File transfers within the local network would also be unimpeded since they do not go through the firewall.

3. SURVIVABILITY PROTECTION MECHANISMS

This chapter describes a number of security protection and reliability mechanisms that are considered candidates for potential implementation by wrapping techniques.

3.1 LOGGING AND AUDITING

One of the simplest forms of system defense is to make a permanent record or log of pertinent data from all transactions. While logging will not prevent any unauthorized access or disruption of information, it often provides a way to identify when and how access is gained or disruption is caused, who is doing it, and what may have been compromised. Separate action can then be taken to restore information and restrict further access. Auditing is the process of reviewing log files for instances of anomalous, disruptive, or unauthorized transactions. Key issues in logging and auditing are how frequently transactions occur, what data needs to be collected on each transaction, how much analysis is needed to identify problem transactions, and how to control these functions dynamically. In addition, log files need to be protected to prevent the clever intruder from erasing the record of his entry and exploits.

3.2 CONSTRAINT CHECKING

Constraint checking is a process of checking, in real time, a set of conditions that need to be satisfied before transactions are allowed to proceed. If any constraint is violated, the checking process may exercise a number of possible options, including:

- Logging the violation and letting the transaction proceed
- Rejecting the transaction
- Requiring further authentication and/or authorization before allowing the transaction
- Restricting or rejecting further transactions from the offending source

Common constraint checks include: bounding program memory address ranges, restricting file access based on user privileges, and preventing simultaneous database updates by multiple users. In addition, a constraint "filter" can be used to log only those transactions that look suspicious. Such filtering can greatly reduce the size of log files.

3.3 ENCRYPTION

Encryption is an essential component of many information protection mechanisms. The most common example of encrypted data on many computers is the system file that contains users' passwords. Passwords are not stored in clear text form because any files that might contain them cannot be adequately hidden. Instead, passwords are encrypted as they are typed in and only the encrypted form is stored.

Encryption is more commonly associated with data in transit within a network or communication system than with stored data. This is partly due to there being little or no other protection available for data in transit. Operating systems typically provide mechanisms to restrict access to information in memory and in files, although the security of these mechanisms depends on the system. Depending on the algorithms used, encrypted files may be substantially larger than the corresponding clear text files, and decrypting them for each use can be time consuming.

3.4 IDENTIFICATION AND AUTHENTICATION

Identification and authentication (I&A) represents the first line of defense against unauthorized access to a system. The most common example of I&A is the user login process. Users identify themselves by a user name and authenticate that they are the owner of that user name by their password. Other identifying information may also be used such as finger print or retina scans. More complex authentication schemes are also in use. Authentication across networks, for example, may involve encrypting passwords in transit, "challenge" protocols that establish one-time passwords, and trusted third-party services that vouch for the authenticity of a remote user or process. Two of these schemes, the Kerberos authentication protocol and a public key authentication protocol are described further in the next chapter.

3.5 ACCESS CONTROL

Access control techniques form a large subset of the more general class of constraint checking methods. Before a user or process is given access to information or other services a check is made to verify that they have the necessary access permissions. Any number of attributes can be used in access control checks, including the user's identity, level of authentication, location, time of day, etc. One particularly important discretionary access control mechanism is the **access control list**.

An access control list confers explicit access permissions to users identified in the list. Information protected by an access control list requires the list to be checked before access is allowed. If the user's name is not on the list, access will be denied. The list of recognized user names, for example, serves as an access control list for the ordinary login process. Additional access control lists may be used to provide access to sensitive information or services to specific users without giving them broad system-wide access privileges. Access control lists are typically protected by operating system file access controls. They may also be encrypted to further preclude disclosure of the names they contain.

One widely-used mechanism for extending a user's access privileges is to temporarily change the process's effective user name for the duration of a privileged transaction. This process name change mechanism must ensure that the user's normal privileges are restored when the transaction terminates, whether the transaction is successfully completed or not.

3.6 LABELING

Labeling is a mechanism by which a sensitivity classification is permanently attached to every object in a system. Labeled objects are typically files, but the concept is more general; for example, individual records or table rows in a database could be labeled objects. Extensive operating system support is needed for labeled files to ensure that correct label information is maintained throughout every possible file transaction.

3.7 FAULT DETECTION AND RECOVERY

Standard practice in robust systems includes the detection of faults or anomalous conditions and initiation of some form of recovery process when such conditions arise. Common faults in distributed computing systems range from message corruption and drop-outs to outages of processing elements or communications links. Low-level communications protocols are usually able to handle detection and re-transmission of corrupted messages. Reliable message transport services include extra acknowledgment messages and watch-dog timers to detect message drop-outs and request re-transmissions. They may also be able to identify failed communication links and attempt to use alternate transmission paths. More commonly, though, recovery from communication system and remote processing element failures has to be handled by application programs.

One of the principal differences between local and remote procedure calls, aside from the communication overhead, is that either the communication system or the remote

service processor may fail while the local processor remains fully operational. When such failures occur, the remote call system cannot recover in any graceful way. Instead, it will return an appropriate failure indication or raise an appropriate exception for the application to handle. Application programs, therefore, cannot simply substitute remote procedure calls for calls to local services without additional changes to handle this new contingency.

3.8 REDUNDANCY

Another standard mechanism used for system fault tolerance is redundancy. A simple example is provided by a local area network that has multiple printers connected to it. If one printer fails or runs out of toner, or if someone is printing an excessively large document, one of the alternate printers can be used.

Similarly, multiple copies of a database and multiple database servers can be set up, providing both faster service under normal conditions and a back-up in case one of the servers fails. The redundant database example is complicated by the problem of making consistent updates on multiple servers. Maintaining consistency and reconciling differences that arise from independent updates may require considerable communication and processing. The improved availability of information, however, can make this extra work well worthwhile. All of the additional processing and coordination can be performed by adapter functions without having to change any application code.

4. EXAMPLE ARCHITECTURE SPECIFICATION

In this section we present a specification of a very simple client/server architecture as an example of how wrapping techniques might be supported by architecture description languages. The survivability protection mechanism addressed is authentication. Two different authentication protocols are described, the Kerberos protocol and the Federal Public Key Infrastructure (PKI) challenge protocol. The adapter interposition mechanism is presumed to be library substitution.

4.1 CLIENT/SERVER SCENARIO

The scenario for this discussion starts with client and server processes that execute on separate platforms and communicate via remote procedure calls over a "trusted" local area network (see Figure 3). In this configuration, the server can respond to requests from all known clients without having to verify the client's identity. The objective is to move these client and server processes to an environment where they communicate over an open, untrusted wide area network. Service is to be provided to the same clients but now, because unauthorized users may attempt to pose as clients, client identities must be authenticated. No changes are to be made to the existing client or server application programs.

The wrapping solution to this problem is depicted in Figure 4. Adapters have been introduced between the client and server application programs and their network services. The function of these adapters is to conduct the necessary authentication process, which involves several additional message exchanges. Although several available RPC libraries support authentication, we assumed our client and server applications made no use of these services.

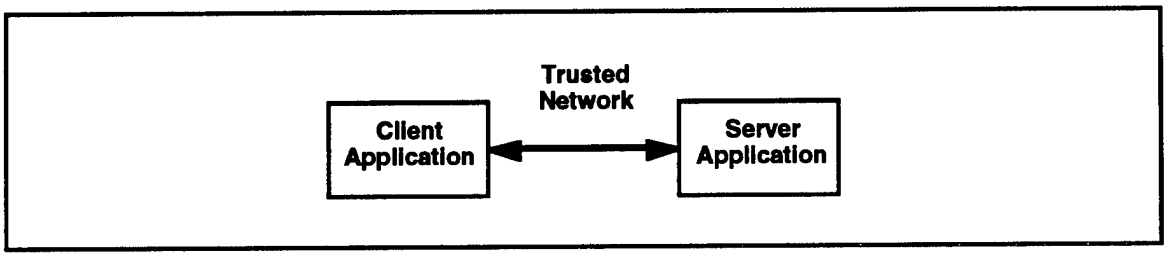

Figure 3. Initial Client/Server Configuration With No Authentication

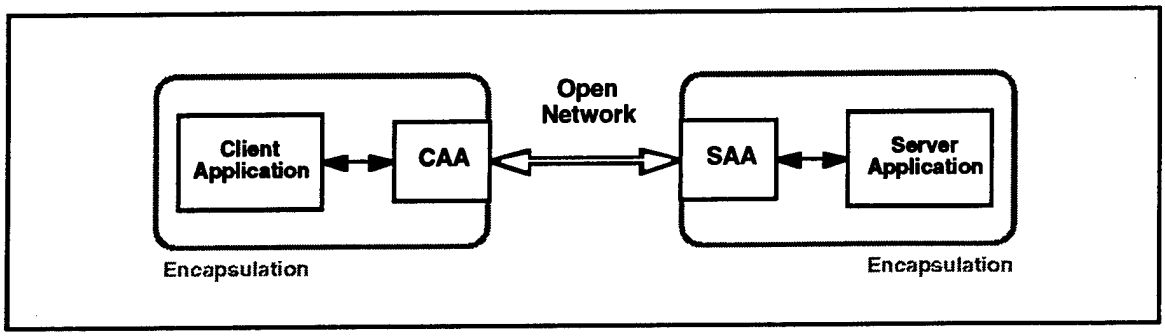

Figure 4. Wrapped Client and Server With Authentication Adapters

4.2 ARCHITECTURE SPECIFICATIONS BEFORE WRAPPING

The top-level architecture specification for the initial configuration using the Rapide architecture description language* is shown in Figure 5. The first part of this specification includes declarations for the three components: the Client, the Server, and the Network. The types of these components are given as Client_Application, Server_Application, and Network_RPC_Service. Client_App() and Server_App() represent simulations of the actual client and server application programs in the Rapide environment. System_-RPC_Lib(1,1) represents a simulation of the operating system's remote procedure call services with connection points for one client and one server.

The second part of the specification defines how these components are connected together. The Client's RPC_Service interface is connected to the first slot in the Network's

```
//*************************************************************************

architecture  Initial_Client_Server()  for Demonstration is

   Client: Client_Application is Client_App();
   Server: Server_Application is Server_App();
   Network: Network_RPC_Service is System_RPC_Lib(1,1);

connect
   Client.RPC_Service to Network.Client_RPC_Service(1);
   Server.RPC_Handler to Network.Server_RPC_Handler(1);

end architecture;

//*************************************************************************
```

Figure 5. Initial Client/Server Architecture Specification in Rapide

* David Luckham, James Vera, and Sigurd Meldal. *Three Concepts of System Architecture.* CSL-TR-95-674, Stanford University, July 1995. (see http://anna.stanford.edu/rapide/rapide.html)

Client_RPC_Service interface. The Server's RPC_Handler interface is connected to the first slot in the Network's Server_RPC_Handler interface. The specifications for these interfaces are shown in the next several figures.

The client's interface with the network's RPC services allows it to issue RPC calls requesting services from some remote server and to receive its replies. The server's interface allows it to pick up and handle incoming client requests and then issue replies to the appropriate return address. These actions are specified in Figure 6. Additional application program interfaces, for example for file system and user interface services, would be specified in a similar way.

The type declarations for the client and server applications are specified as shown in Figure 7. These declarations show some of the additional interfaces that allow connections to services provided by the operating system or separate commercial products. Note that we have enabled the server application to connect to the network as a client of other RPC services. Since our focus is on client-server interaction, the simplified architecture specification in Figure 5 does not show connections to these additional interfaces.

The interface for the network RPC service is shown in Figure 8. This interface was modeled as two arrays of "slots" where clients and servers could be plugged into the network. The type of the client slots is the dual of the client RPC service interface and, therefore, these slots accept connections from client application programs. Similarly, the type of the server slots is the dual of the server RPC handler interface and, therefore, these

```
//*************************************************************************

type  Client_RPC_Service  is interface
   action
      out Issue_RPC_Call(svr_addr: Network_Address;
                         request: Service_Request);
      in Receive_RPC_Reply(reply: Service_Reply);
   end interface;

type  Server_RPC_Handler  is interface
   action
      in Handle_RPC_Call(rtn_addr: Network_Address;
                         request: Service_Request);
      out Issue_RPC_Reply(rtn_addr: Network_Address;
                          reply: Service_Reply);
   end interface;

//*************************************************************************
```

Figure 6. Type Specifications for Client and Server Interfaces

```
//*****************************************************************************

type  Client_Application  is interface
   service File_Service: System_File_Service;
   service GUI_Service: COTS_GUI_Service;
   service RPC_Service: Client_RPC_Service;
end interface;

type  Server_Application  is interface
   service DBMS_Service: COTS_DBMS_Service;
   service GUI_Service: COTS_GUI_Service;
   service RPC_Handler: Server_RPC_Handler;
   service RPC_Service: Client_RPC_Service;    // servers may also be clients
   end interface;

//*****************************************************************************
```

Figure 7. Type Specifications for Client and Server Applications

slots accept connections from server application programs. Rapide provides ways to model architecture connections that change dynamically but we did not use these features.

The principal functions of the network component in our model are to:

(1) Map the server's network address, which clients use to identify the desired server, to the appropriate server interface slot;

(2) Pick up the client's return address, based on its slot number, and pass it on to the server RPC handler along with the client's request; and

(3) Map the client's return address back to the appropriate client interface slot for the server's reply.

4.3 ADAPTER AND WRAPPER SPECIFICATIONS

As shown in Figure 4, two adapters need to be specified, the client authentication adapter (CAA) and the server authentication adapter (SAA). Because of the objective of

```
//*****************************************************************************

type  Network_RPC_Service ( Num_Clients: integer;
                            Num_Servers: integer )  is interface
   service Client_RPC_Service(1..Num_Clients): dual Client_RPC_Services;
   service Server_RPC_Service(1..Num_Servers): dual Server_RPC_Handler;
   end interface;

//*****************************************************************************
```

Figure 8. Type specification for network RPC service

```
//*************************************************************************

type  Client_Authentication_Adapter  is interface
   service Client_Side: dual Client_RPC_Services;
   service Network_Side: Client_RPC_Services;
   end interface;

type  Server_Authentication_Adapter  is interface
   service Server_Side: dual Server_RPC_Handler;
   service Network_Side: Server_RPC_Handler;
   end interface;

//*************************************************************************
```

Figure 9. Type Specifications for Authentication Adapters

plug-compatible interfaces and plug-and-play adaptability, the interface specifications for these adapters are quite straightforward (see Figure 9). The client adapter has a client-side interface, which accepts client RPC service connections, and a network-side interface, which plugs into the network's client RPC services. The server adapter has a corresponding server-side interface, which accepts server RPC handler connections, and a network-side interface, which plugs into the network's server RPC services.

Wrapping the client and server application programs consists of creating new client and server sub-architectures that bind the appropriate adapters to the original applications. Figure 10 shows how the original client's RPC service interface is connected to the

```
//*************************************************************************

architecture  Wrapped_Client()  for Client_Application is
   Original_Client: Client_Application is Client_App();
   Adapter: Client_Authentication_Adapter is Kerberos_Client_Adapter();
connect
   Original_Client.RPC_Service to Adapter.Client_Side;
   Adapter.Network_Side to RPC_Service;
end;

architecture  Wrapped_Server()  for Server_Application is
   Original_Server: Server_Application is Server_App();
   Adapter: Server_Authentication_Adapter is Kerberos_Server_Adapter();
connect
   Original_Server.RPC_Handler to Adapter.Server_Side;
   Adapter.Network_Side to RPC_Handler;
end;

//*************************************************************************
```

Figure 10. Client and Server Sub-Architecture Wrapping Specifications

adapter's client-side interface and the adapter's network side interface is exported as the wrapped client's RPC service interface. Similar "wiring" changes are made for the wrapped server. The adapter behavior modules shown in Figure 10 are for Kerberos authentication. Public key authentication can be supported by simply substituting the public key versions of these modules.

Although the adapters' network-side interfaces are the same as the applications' network interfaces, the adapters exchange a new set of authenticated RPC service requests. The function of the adapters is to translate the original form of requests into authenticated requests and then back again. Only authenticated RPC requests are exchanged over the open network. The client and server applications see only the original requests. If an unwrapped client attempts to access the server directly, using the original requests, the server's adapter will block the transaction and return an "authentication required" or "access denied" error reply.

4.4 KERBEROS AUTHENTICATION ADAPTERS

The Kerberos authentication protocol is an encryption key distribution protocol that requires the exchanges of information illustrated in Figure 11. We modeled these exchanges in terms of remote procedure calls. The official Kerberos protocol* is specified at a lower level of abstraction with discrete messages and specific message formats for requests and replies.

Under the Kerberos protocol, a client must present a "ticket" and an "authenticator", which represent its credentials, along with each service request. These credentials are time-tagged and encrypted so that both clients and servers (or, in our case, the adapters) can verify each other's authenticity. Tickets are obtained from a trusted third-party, the Kerberos key distribution center (KDC). A client obtains an initial ticket by identifying itself to the KDC and then with this ticket can obtain tickets for other servers from the KDC ticket granting service (TGS). The KDC holds copies of both the client's and the server's secret encryption keys, which is essential to the service it provides.

Along with each ticket issued, the KDC and TGS generate a unique encryption key, called the session key, which is used to sign and, optionally, encrypt transactions between the client and the server. The session key, along with a time stamp, is encrypted using the client's secret key and returned to the client along with the ticket. Only the client, therefore,

---

* J. Kohl and C. Neuman. *The Kerberos Network Authentication Service (V5).* Internet Engineering Task Force, RFC 1510, September 1993. (http://ds.internic.net/rfc/rfc1510.txt)

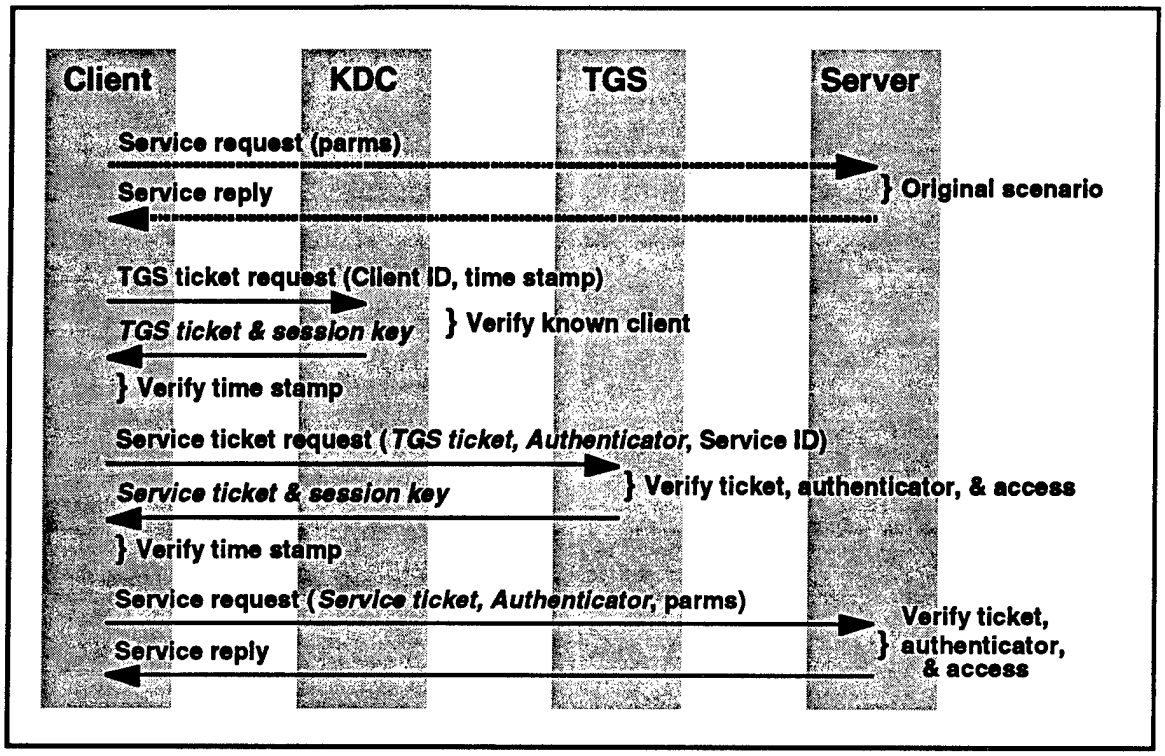

Figure 11. Steps in the Kerberos Authentication Protocol

should be able to decode and extract this information. The client can also check the time stamp to ensure the session key is fresh.

Tickets contain the client's identification, the session key generated by the KDC or TGS, the time the ticket was issued, and its expiration time — and all this is encrypted using the server's secret key. Only the server, therefore, should be able to decode and unpack this information. The server can check the issue and expiration times for validity.

Authenticators contain the client's identification and a fresh time stamp, and are encrypted using the session key. The server can decode this information using the session key, which is contained in the ticket. The ticket and authenticator, together, allow the server to check the validity of the client's credentials and the timeliness of the request.

In a wrapper implementation the Kerberos authentication adapters handle all the tickets, session keys, authenticators, encryption and decryption processing, and validity checking.

4.5 PUBLIC KEY AUTHENTICATION ADAPTERS

Public key authentication uses digital signatures, which are based on asymmetric encryption using pairs of keys, one public and one private. Either key can be used to encrypt a message. The other is needed to decrypt the message. A signed message is encrypted using the sender's private key. The receiver can then decrypt the message using the sender's public key. The message may include a time stamp and other information to ensure, for example, that it is not a copy of a previous message. The receiver may generate a random "challenge" message and require the sender to include it in its signed messages. This is the standard approach for authentication using public key encryption.*

In our client/server scenario we have the server generate a new challenge message for each client session. This allows one challenge to be used for multiple transactions within a limited period. To authenticate the server to the client, we have the server time-stamp and sign a digest of each transaction's results. The sequence of messages required is shown in Figure 12. These challenge, signing, and signature validation steps can be handled easily by our client and server adapters.

Where does a server get a client's public key? This information is distributed in the form of an X.509 "certificate." A user's certificate contains their name, public key, and validity time stamps, among other information, and is signed by a Certification Authority. Certificates can be obtained from an X.500 directory by searching for the desired name. The client can obtain the server's certificate the same way. A significant part of the Federal Public Key Infrastructure is dedicated to keeping directory information current. Note that the public key directories contain no private keys and perform no encryption, as in the Kerberos approach.

Digitally signed messages can also be used to exchange a session key like the one used in the Kerberos protocol. Symmetric session key encryption algorithms are significantly more efficient than asymmetric public key algorithms. We did not model this key exchange protocol.

---

* National Institute of Standards and Technology (NIST). *Technical Specifications for the Federal Public Key Infrastructure (PKI), Part C: Proposed Federal PKI Concept of Operations.* February 1996. (see http://csrc.nist.gov/pki/welcome.html)

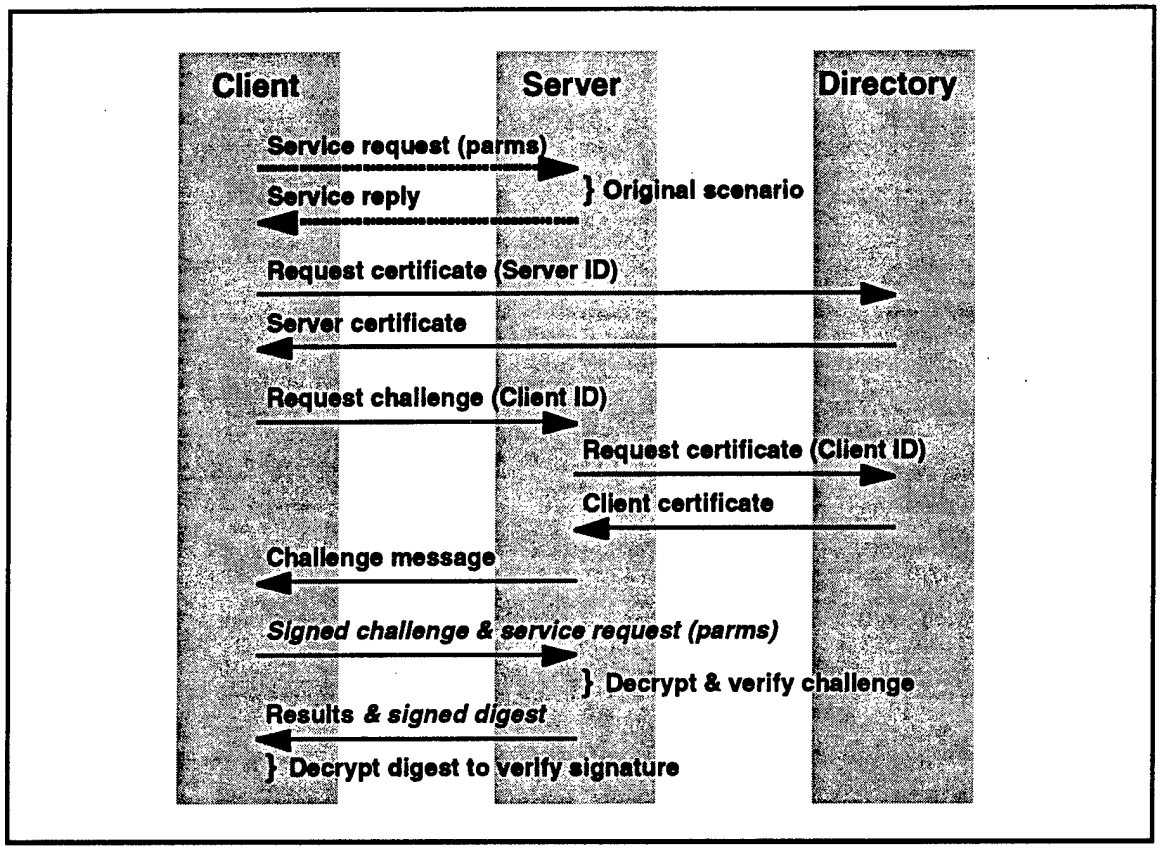

Figure 12. Steps in the Public Key Authentication Protocol

5. FINDINGS

5.1 ADAPTER FUNCTIONS

The previous chapter shows that authentication adapters can be added to client and server application programs relatively easily by substituting remote procedure call libraries. Different authentication techniques can be plugged in simply by substituting adapters and connecting the necessary components into the architecture.

A more complex adapter would be necessary to connect a client to multiple servers that require different authentication protocols. It should be possible to build a dispatching adapter that routes requests and replies to the appropriate authentication adapters. The Kerberos and PKI adapters could then be used as components that plug into the dispatcher.

Many other security and fault-tolerance enhancing functions appear to be implementable using plug-in adapters. Transaction auditing and logging, access control checking, and simple fault detection and recovery can all be modeled as intermediate actions taken by an adapter either before forwarding a request or before returning a reply from a service. It should also be possible for an adapter to dispatch a request to multiple servers to provide redundancy. The first reply from a dispatched request might be returned to improve overall system performance. Another possibility is to collect and compare multiple replies and then retry or reject discrepancies.

5.2 LEVELS OF ABSTRACTION

Implementing adapters is greatly facilitated by interfaces that provide a high level of abstraction. Ideally, the level of abstraction would be at the application's semantic level, where the meaning of each transaction and the relevant parameters are clearly evident. Of the wrapper implementation mechanisms we studied, the library substitution mechanism provides the highest level of abstraction. By comparison, operating system calls and network firewalls provide much lower-level interfaces, where it may be much more difficult to piece together observed sequences of calls or messages into recognizable transactions. Reconstructing the semantic levels of a transaction from low-level interfaces is made even more difficult if the higher-level protocols and the mappings to their lower-level implementations are not rigorously defined.

Logging transaction information illustrates some of the difficulties of using low-level interfaces. Ideally, we would like to selectively log a subset of transactions that may contain potential high-level transaction anomalies. There may be a number of attributes of the high-level transactions that could be used to filter out uninteresting transactions. For example, we may be looking for database queries of a particular kind. Identifying and extracting this information from a low-level interface such as operating system calls, however, would require significant effort to reconstruct recognizable parts of the high-level query. The usual approach to logging information from low-level interfaces is to record large volumes of data and then weed through this data off line.

5.3 STRENGTH OF ENCAPSULATION

Application programs may use high-level library calls some of the time but at other times may go around the library and execute equivalent functions directly on its own. Many database management systems, for example, bypass higher-level file system services to gain improved performance. An application that uses remote procedure calls may bypass some RPC library services by using lower-level socket calls. Of course, any security or survivability checks introduced by adapters linked to bypassed library calls would be bypassed as well.

Where high-level source code is available it may be possible to analyze this code and recognize bypassing operations. A tool that analyzes the library linkage editing process may also be able to recognize bypassing calls at the object code level.

Another approach is to cut off the low-level services completely and provide the high-level services through a separate process. For example, to restrict an application to a particular file system interface, all direct file operations would have to be rejected by intercepting the application's file-related operating system calls. The library that defines the file system interface would then be substituted by a library that translates these calls into IPC calls to a separate file service process. This separate process would provide a virtual file system for the application, and could perform a number of other useful security and reliability functions in addition to enforcing the desired interface. Example functions might include encrypting and decrypting files, and creating backup files on separate media.

These examples show that the encapsulation around combinations of adapter and application code, for most of the wrapping mechanisms we studied, is relatively weak. Even the mechanism of providing services through a separate process depends on the level of protection provided by the operating system. This technique relies heavily on being able

to completely sever the application's direct access to the protected service; for example, by moving it into a separate address space. It may be desirable, in addition, for the service process to execute with different privileges than the application.

5.4 WRAPPER CONFIGURATION MANAGEMENT

The weakness of encapsulation mechanisms for wrappers suggests a need for increased attention to configuration management of the entire platform. Any changes to application code or the operating environment, including "transparent" library or system upgrades that do not affect normal applications, could affect the correct functioning of wrappers. There appear to be no easy ways to ensure that adapters remain attached and continue to operate as intended or to warn users that their operation may have been altered. For example, if a transaction auditing adapter or an intrusion detection adapter were disconnected, it would simply never report any transaction anomalies or intrusion attempts.

5.5 LOOSE VERSUS TIGHT SPECIFICATIONS

There is some debate within the software architecture and reuse communities about whether interfaces should allow broad plug-and-play adaptability, which requires a very general or "loose" specification, or whether interfaces should be tightly constrained so that only perfectly matching duals are plug compatible. For example, connections to the internet can be considered to have a loose interface because there are very few restrictions on the types of messages that can be sent. A tighter specification might constrain the same interface to a particular protocol, say FTP or HTTP. An even tighter specification might constrain the interface to a particular set of application-specific remote procedure calls.

Tight specifications make analysis of survivability properties easier, they simplify the development of adapters that check for anomalies, and they restrict the range of plug-compatible components. Loose specifications extend the range of plug-compatible or reusable components, which facilitates system development, but they introduce the potential for connecting incompatible or only partially compatible components.

We found the need for both tight and loose specifications in our modeling of authentication adapters. We started with tight specifications for one set of RPC messages that were exchanged between the client and server application programs. We then introduced a new set of RPC messages for authenticated transactions, also with tight specifications. This meant, however, that the two sides of the adapters could not be matching dual interfaces — the application side fit the original specifications, while the network side fit the new specifications. Our network interface specification, therefore, was

loosened to accommodate any components that exchange RPC messages. This is more accurate, in the sense that networks indeed carry all kinds of messages, and it does allow network users to attempt to exchange incompatible RPC messages. The normal RPC mechanisms and the authentication protocols, however, combine to prevent any incompatible clients and servers from achieving successful transactions.

5.6 ACCURACY OF INTERFACE SPECIFICATIONS

Overall, we found that modeling systems using architecture description languages worked quite well. Interface specifications derived for COTS or legacy software, however, may not reflect the actual interfaces with sufficient accuracy. Among the inaccuracies there may be undocumented entry points, additional parameters that change the interface's behavior, and debug modes that provide additional controls or access to additional information. Any of these extra features can expose security holes. Analysis of the interface specifications will never reveal these holes because, according to the specifications, they do not exist.

We ran into similar difficulties when we started to put together the architecture specifications discussed in Section 4. The problem was the level of abstraction of transactions represented in the model, not incorrect information. At the applications' semantic level, transactions are represented by remote procedure calls. At a lower level these calls are transformed into sequences of message exchanges. The difficulty arises when anomalies are introduced at the lower level; for example, by an intruder who is not going to play by the rules established for RPC transactions. Analysis of how such an intruder might interfere with the authentication activities required mapping the higher-level transactions into the lower-level message exchanges. Only then could we show that the exchange of session keys, for example, was safe under the assumption of an adequately random key generator and adequate encryption.

5.7 ACCESSIBILITY OF INTERFACES

Wrapping works well when adapters can be attached to the most appropriate interfaces. The best insertion point for a desired adapter function, however, may be internal to an application. Adapting the accessible external interface may at best be awkward and possibly not feasible at all. An example problem is to restrict access to certain rows, columns, and tables of a database. It may be relatively easy to apply such restrictions within a database management system, based on user access privileges. Attempting to impose the same restrictions using an external adapter would require parsing

the input queries, checking the user's access privileges, possibly modifying the query before passing it to the database system, and then filtering the query results to eliminate any disallowed information before returning the results to the user. Such an adapter would be nearly as complex as the database management system and just as difficult to verify or test.

5.8 AVAILABILITY OF MECHANISMS

The specific details of each wrapping mechanism depend on system configuration and particular details of the hardware and operating system. The library linking mechanism is a primary issue. Dynamic linking supports a very important high-level interface. Without it, wrapping is reduced to using only low-level interfaces. Porting application code to run on a system with dynamically linked libraries should mean that these libraries will be used. This is not automatically guaranteed, however. There may be applications that are "binary compatible" with the dynamically linked system but retain their original statically linked library code.

5.9 OPEN STANDARDS AND COMMERCIALLY SUPPORTED PLUG-IN FUNCTIONS

We found a number of open standards for distributed computing that already include authentication services and are now incorporating "hooks" for additional services to be provided by plug-in adapter modules.

- DCE* — Distributed Computing Environment — is a remote procedure call standard developed by the Open Group (formerly the Open Software Foundation). In addition to remote procedure calls, DCE supports directory services, network time services, and a distributed file system built using RPC's. DCE currently supports Kerberos Version 5 authentication and plans to support PKI with a general capability for plug-in adapters. DCE is supported on a large number of commercial platforms, including most Unix systems. The client-side interface is available for MS Windows, Windows NT, Windows 95, and OS/2 on PC's and for the Macintosh.
- CORBA† — Common Object Request Broker Architecture — is an object interface specification standard developed by the Object Management Group. Its security services are still being developed. The general architecture,

---

* cf. Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE.* O'Reilly & Associates, Inc., 1993. (see also, http://www.opengroup.org/tech/dce/info/)

† cf. Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice Hall, 1996. (see also, http://www.omg.org/)

however, has "filter" connectors for plugging in arbitrary adapters on both the client and the server side of an object request broker.

- SOCKS* — (the name is not an acronym) — is a network proxy mechanism for firewalls that allows local users full access to external network services, while preventing unauthorized access to local resources from the external network. SOCKS Version 5 supports multiple authentication techniques. The reference implementation supports Kerberos using the Generic Security Service Application Program Interface (GSS-API). SOCKS server software, which allows a machine to serve as a network firewall, is available for Windows NT and Unix platforms from NEC. Client applications operating on the external network must support the SOCKS protocol to gain access to services provided by machines that are behind a firewall. Dynamic library substitution is used to add SOCKS adapters to client applications on Unix platforms.
- SSL† — Secure Sockets Layer — is a network transport layer encryption protocol developed by Netscape Communications Corporation. SSL supports public key authentication. It also provides a framework into which additional exncryption methods can be incorporated. It is not clear if SSL intends to support plug-in extensions, however. SSL has been submitted for consideration as a draft internet standard. A number of Netscape products include support for SSL.

5.10 PERFORMANCE IMPACTS OF WRAPPING

Adapters that operate at high-level interfaces and are introduced by library substitution—for the survivability functions they provide—have the lowest impact on application performance. For authentication protocols that require several extra message exchanges to establish client-server connections, for instance, the difference between adapters added by library substitution and custom built-in implementations should be imperceptible. Adapters that operate at low-level interfaces and perform relatively simple checks should also impose little or no significant performance impact. As the amount of processing performed by these adapters increases, particularly for operations that normally take little time, they may begin to introduce noticeable performance degradation. For example, passing file input and output operations through a separate process to limit an application's access to the file system may slow it down noticeably because of the extra

---

* Marcus Leech, et al. *SOCKS Protocol Version 5.* Internet Engineering Task Force, RFC 1928, March 1996. (see http://www.socks.nec.com/rfc/rfc1928.txt and http://www.socks.nec.com/introduction.html)

† Alan Freier, Philip Karlton, and Paul Kocher. *The SSL Protocol, Version 3.0.* Netscape Communications Corp., March 1996. (see http://home.netscape.com/eng/ssl3/index.html)

process swapping this would introduce. Logging large quantities of data from low-level interfaces can have significant performance impact. This can occur when the relevant information cannot be easily recognized in its low-level form and, therefore, cannot be easily filtered out. In this situation there would be little performance difference between a wrapper and a custom built-in implementation. Overall, therefore, we expect wrapping as an implementation mechanism to have relatively minor performance impacts above and beyond the survivability functions they provide.

6. CONCLUSIONS

Several techniques for wrapping legacy and COTS application software are available and have been employed in commercial products. Adapters for a number of security and survivability functions can be readily constructed and interposed at accessible interfaces using these techniques. Practical aspects of wrapping existing software such as the accessibility of interfaces, the accuracy of interface specifications, and protecting wrappers from being bypassed by applications, on the other hand, appear to be significant open problems that will need to be addressed to enable broad use of these techniques for security and survivability hardening.

If there is sufficient market pressure for security and survivability functions in commercial software products, vendors should be able to incorporate those functions more easily and with higher assurance than can be achieved by a third party using current wrapping techniques. Functions incorporated directly into applications and system software have access to the most appropriate internal interfaces at high-levels of abstraction. In addition, opportunities for bypassing functions within an application can be controlled by source code analysis.

Some application developers may choose not to support security and survivability functions themselves. As an alternative, they may provide special interfaces for third-party plug-in adapters. This would provide more flexibility for adding functions that provide different types and, perhaps, different levels of protection. Even so, vendors may not be willing to warrant the appropriateness or security of these interfaces for any particular add-on wrapper services. Careful configuration management will be necessary to keep adapters plugged-in and functioning as intended.

Wrappers for products that are not enhanced by their developers may become a necessity for some DoD uses. In this case, the current problems of interface accessibility, accuracy of interface specifications, and protecting the integrity of wrapped code come into full play. The market for such wrapping is expected to be small (a larger market would justify the product developer's participation). Reusable adapter components will have to be developed rather than individual custom solutions to control cost.

Legacy application software that is no longer being actively maintained is another category where wrappers offer perhaps the only viable security and survivability enhancement options. Software in this category is known to be a liability, independent of security and survivability issues, and should be high on the DoD's list to be upgraded, replaced, or eliminated. Wrapper development time and the useful life-span of the application should, therefore, be considered in wrapping decisions.

Modeling systems using architecture description languages worked well for specifying adapter interfaces and functionality. Two questions for the ADL and formal methods communities were raised, however. One is how to deal with an application program's access to an interface at multiple levels of abstraction. Network services may be accessible, for example, through both a high-level RPC interface and a low-level sockets interface. Can RPC operations be protected from potentially interfering socket operations? Can this be done without cutting off all access to the sockets interface?

The second question is how to resolve the issues betweem loose and tight specifications. Are there systematic ways to relax or generalize tight specifications to allow reuse without giving up key security or survivability properties? Without having to repeat the analysis? Is there any way to derive tight specifications (implying assurance of system properties) for systems composed from loose-fitting reusable components?

Solutions to the questions raised here would enable wrapping techniques to provide valuable security and survivability hardening for important segments of the DoD's software inventory.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1997	Final

4. TITLE AND SUBTITLE
Analysis of Secure Wrapping Technologies

5. FUNDING NUMBERS
DASW01-94-C-0054
DARPA Assignment A-202

6. AUTHOR(S)
John M. Boone, Karen D. Gordon, Reginald N. Meeson, Terry Mayfield, Edward A. Schneider

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-3297

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
Information Technology Office
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited: 7 July 1997.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

Networked computer systems are increasingly being threatened by inadvertent disruption and, at time, malicious attacks from other network users. This paper analyses the types, extent, and performance impacts of protection that might be achieved by applying software wrapping techniques to DoD legacy software systems and to commercial off-the-shelf (COTS) software products. Wrapping is a technique by which adapters are interposed at software interfaces, allowing new functions to be added. Architecture modeling techniques were used to verify that wrapping could be used to add services such as encryption, authentication, access controls, transactions logs, fault detection and correction, and redundancy to existing software systems. While the best solution for security in legacy systems and COTS software may ultimately be provided by their complete reengineering, our investigations confirm that wrapping techniques can provide an effective, near-term solution to a number of existing security problems. Techniques for ensuring the integrity of wrappers and wrapped software systems, and possibilities for additional functionality provided through wrappers require further investigation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer Security, Wrapper, Authentication, Commercial Off-the-Shelf (COTS), Legacy Systems.	53
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR