Vol. 20 No. 6 Candle Computer Report Nov./Dec. 1998

1978 Years in Publication 1998

Object Persistence Through Serialization

In 1997 Java 1.1 released an abundance of features and enhancements that demanded the attention of the programming world. Among these new amenities is Java's answer to the object persistence challenge: object serialization, arguably the most important Java feature of all. Through serialization a programmer can funnel an application's entire world of objects into a simple single string of bytes. Unfortunately, little has been written of serialization, suggesting that it still may be a relatively unknown Java feature. Any programmer faced with transporting an object's state across the Web would certainly be grateful to know of it.

This article will take you through the problem domain of object persistence from its definition to the solution that Java provides. Along the way, I will introduce some useful terminology and concepts, borrowed from the world of C++, as well as demonstrate the usefulness of serialization.

THE NEED TO PERSIST

The journey of object serialization begins with the challenge of that need. When the thrust of object-oriented programming descended upon the coding community, the inhabitants were confronted with saving and transporting these constructs as they exist during execution. If an object is to persist beyond the life of the application, it has to be somehow transmogrified into a stream suitable for something external such as a file or network. Before the availability of object-persistent methods, this task was left to the imagination of the programmer.

First, an overview of object persistence as implemented in C++ will give you some helpful insight by demonstrating the different ways that the challenge has been approached. The overview will introduce useful terminology, and present a framework of concepts upon which Java's object serialization

(Continued on page 2)

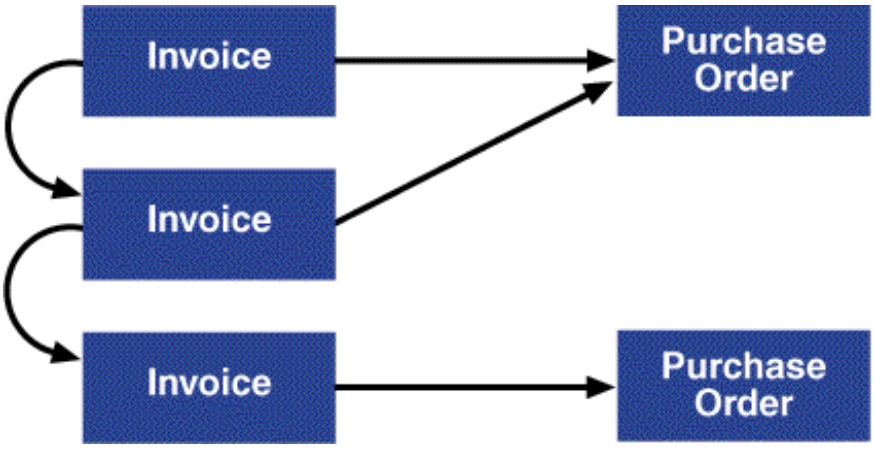

Figure 1 – Original morphology

Securing Messaging Middleware Applications

The introduction of messaging middleware products such as IBM's MQSeries, Microsoft's MSMQ, and BEA's MessageQ has made developing networked applications much more attractive. These products enable companies to leverage their huge investment in legacy applications and bring them into the distributed world without great difficulty. Consumer demand for remote access to applications, which ultimately may read and update mission-critical corporate data, increases the momentum behind strategic business plans to quickly develop networked applications.

An important area of concern for companies deploying networked application solutions is security. That's because the implementation of messaging middleware-based solutions in the networked environment introduces many new security considerations absent in the design of more traditional applications executed on a single platform such as a mainframe.

Secure access to mainframe resources has been implemented successfully for many years in the traditional environments. These mainframe-based applications tightly control access by the users' close proximity to the machine and by security software between the users' terminals and the application software and data. The effectiveness of this security relies on IS personnel assigning an identification to each user along with a set of permissions. This defines the level of user access to various system resources, such as transaction codes, databases, and subsystems.

But with the evolution of applications distributed across multiple processors, connected through internal and external communication channels, ensuring the security of mission critical

(Continued on page 5)

can be staged. To be sure, Java does not offer a complete solution, as do the purveyors of C++ compilers. Nonetheless, Java now enables the programmer to persist and transport application objects and that certainly is a large step towards a mature object-oriented language.

Object serialization arose from the need for object persistence. To the mainframe veterans this may seem as nothing more than saving memory data structures without reformatting them. In its simplest form, that is absolutely true. But as I will explain here, it is much more than that.

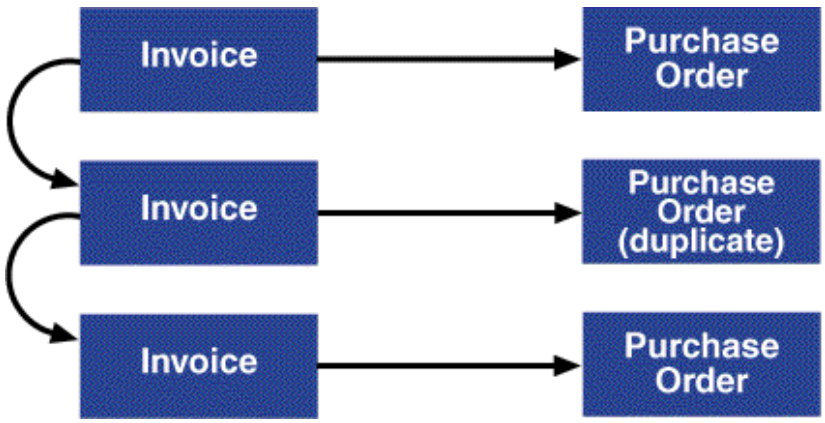

Figure 2 – Altered morphology

THE AMBIGUITY OF SERIALIZATION

Before the internals of serialization are exposed, you need to be familiar with the terminology and the underlying concepts. Let's first look at a working definition of the word "serialization."

Serialization is one of those ambiguous words that never found a concrete meaning in the computing lexicon. Most system programmers of the old school first came across it in the IBM manuals, which used it to describe how certain computer resources (e.g. memory, files, and sections of machine code) could be used by concurrently running tasks. Usage of these resources had to be sequential because they existed in unique states within each running task or thread (task is the old-fashioned name for thread). If more than one thread operated upon a resource, its state could corrupt another state. This would render the resource unusable and cause ABENDs in code that usually did not cause the problem in the first place.

The solution was to serialize the use of the resource. Only one thread at a time could use it and the other threads would use it in their sequences. The specific term in the IBM vernacular was SRR: Serially Reusable Resource.

This form of serialization also exists in UNIX as well as Windows 95 and Windows NT (CP/M and DOS, unfortunately did not provide a multithreaded environment and therefore did not need such a term). In UNIX-speak, however, serial reuse has a different name: mutually exclusive resource. This object acts as a gatekeeper for serially reusable resources and thus is called a MUTEX.

The other use of the word serialization refers to a linear string of bytes representing an object that is potentially multidimensional. Coded in this string of bytes is all the information necessary to reconstitute the original state regardless of the number of dimensions in that state. It is this definition that will be used throughout this article. In Java, serialization specifically refers to a single row of data that was produced through the ObjectOutputStream class and can be read by the ObjectInputStream class.

Pointer Relationships

Next, let's define graphs and morphologies. C++ compiler vendors use one term borrowed from the sciences: morphology; Java uses another: graph. Essentially, they both describe the same thing: the pointer relationships between objects and the structures that those relationships create. (If you insist that Java does not have pointers, please substitute the word "reference.") Do not confuse the term graph with the visual components of Java. Graph can be thought of as the morphology charted on paper. Keep in mind that throughout this article, it is the relationship of objects (i.e., the run-time instance of a class) that composes a graph and not the class-hierarchy defined through inheritance. If you have experience with object-oriented programming, you know the difference between the compile-time class hierarchy and the run-time relationship of objects. Persistence concerns the run-time structure, while the

class hierarchy is viewed as a single instance of an object.

The reason a name is given to an abstract idea like structures of related objects is that these structures are entities in their own right, especially when it comes to serializing them. Traversing morphologies, which the serialization encoding process does, can be a delicate process, though. This is because morphology can be inadvertently altered during the process. If a structure has multiple connections to common object instances, the encoding process can view these connections as unique paths leading to different objects. Unless the encoding is sophisticated, it will not know it has already come across a given object, and this can change the structure. Examples of original and changed morphologies are illustrated in Figures 1 and 2.

Figure 1 is a structure with two invoices that reference the same purchase order. If the encoding logic is not carefully implemented, it will mistake the two references as two objects, resulting in the incorrectly reconstituted structure shown in Figure 2. The common purchase order has now become two distinct purchase orders. This incorrect structure could wreak havoc on the application code if that code misinterprets how the serialization views the structure.

These examples make it clear that an object's persistent state is incidental to its encoding logic. It also suggests that there are probably some underlying rules governing how persistence works.

THE FOUR LEVELS OF PERSISTENCE

Now that we have some working definitions, let's take a look at the different ways C++ has implemented object persistence. We can use these implementations as a basis for some generalizations about the different levels of persistence.

Persistent methods can range from simple to sophisticated. Typically, there are four levels of persistence if absence is included.

No Persistence

This means that the object does not incorporate any persistent methods offered by the compiler or tool kit. In Java, it means the object class did not implement the serializable interface in any object-class definitions.

Simple Persistence

This is equivalent to a deep copy that does not recognize when two or more pointers reference the same object. As shown in the figures, the morphology is not preserved if two or more paths in the morphology lead to the same object instance. That is because simple persistence assumes each referenced object is unique. In some implementations, simple persistence means that only single objects can be processed. Objects that reference other objects, whether they are of the same type or not, are considered invalid. Java's serialization interface does not support simple persistence.

Isomorphic Persistence

This works like its simple counterpart (it also creates a deep copy) except it keeps the original morphology intact. Multiple pointers to the same object are reconstituted to point to a single object. The structure on the receiving end will be identical to that on the sending end. What makes this level isomorphic is that its objects must be uniform in two respects: (1) the receiving end must have a local definition for the object type; (2) all objects are assumed to be the same type. (A morphology of objects that are all the same type is called a homogenous collection.) For example, if the receiver can reconstitute "square" objects derived from the base class "shape," then it cannot accept unknown derivatives of shape such as "triangle." Polymorphism, on the other hand, does not have this restriction.

Polymorphic Persistence

This is perhaps the most interesting level and certainly the only true object-oriented one. Essentially, it behaves like isomorphism but with fewer burdens on the receiver and the ability to deal with heterogeneous collections (i.e., morphologies consisting of different object types). More specifically, the code reconstructing the object does not need to know the target's object type in advance. Similar to the object-oriented concept dynamic polymorphism – not to be confused with parametric polymorphism or overloading – the object upon which the code is operating does not need a full description of the object's class. But here is an important requirement: the target object must be derived from a common base class understood on both ends. For instance, in Rogue Wave the object must be a derivative of RWCollectable and in MFC it must be a derivative of CObject. So, despite the panacea claims about polymorphism, the reality is that something is always known about the object type, even if it is only the base class.

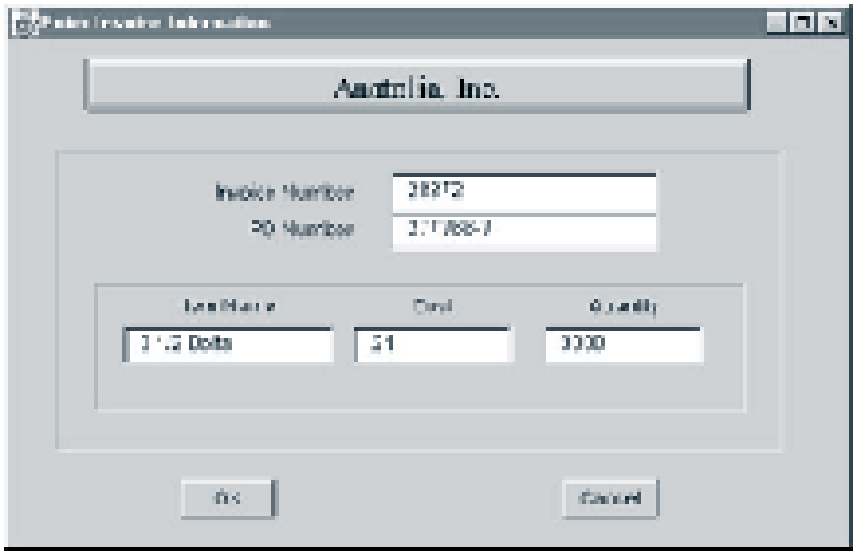

Figure 3 – GUI frame used to collect serialized data and to deserialize it.

The benefit of polymorphism is that new object types can be invented on the sending side as long as the receiving side does not depend on the new methods or data types. If the receiver operates correctly on "shape" object types and "triangle" is suddenly sent to it, it will continue to work without ever knowing about "triangle." Polymorphism is truly the heart and soul of object-oriented programming.

PERSISTENCE IN C++

Let's now take a look at some of the implementation details in C++. This will give you an idea of what is available in C++ and allow us to make some comparisons to Java's serialization feature.

C++ implementations typically provide a base class with useful overloaded operators. For example, Borland has its TStreamableBase class, Rouge Wave has its RWCollectable class, and Microsoft has its MFC CObject class, just to name a few. Unfortunately, these are not as straightforward to use as Java's Serializable interface because each requires you to use a macro. Rogue Wave requires RW_DEFINE_PERSISTABLE, Borland requires DECLARE_STREAMABLE, and MFC requires DECLARE_SERIAL.

These macros are awkward and kludge-like. But the advantage over Java is that the macro can generate any constructors the persistence process needs. That is one less method for the programmer to code. Compare this to Java, where a no-argument constructor must either be explicitly coded or allowed to default by omitting a constructor.

JAVA'S SERIALIZATION

We have examined the problem domain of object persistence, so we can now delve into the specifics of the Java solution.

The number of design problems that serialization can solve is probably limitless. Generally, serialization can preserve an any object instance across a network, the life of a thread or process.

Java provides object persistence as an interface to be implemented by the programmer. This interface identifies those classes that can be serialized. Fortunately, it is easy to implement because there are no methods to write.

First, implementing serialization does not make the class persistent. It only makes it a candidate for any code in the application that chooses to persist that object type. It is the ObjectOutputStream class that does the serialization. This stream type is usually wrapped around either a file stream or a socket stream. Wrapping these streams together enables a seamless transfer from the object itself to the file or socket. It is ObjectOutputStream's writeObject() method that does the work of serializing and calling the outer stream's write function. Conversely, it is ObjectInputStream's readObject()method that does the work of reading and deserializing.

Secondly, any subclass that inherits from a serializable base class is automatically serializable. This is not always obvious. For example, if "square" inherits from the base class "shape" and "shape" is serializable, "square" is automatically serializable, as well. That also applies to any subclass of "square." However, any class that neither implements serialization nor sub-classifies a serializable class is not a candidate for serialization and will cause a run-time exception. This principle applies to all objects in the morphology. If a serializable object references any non-serializable object, then NotSerializableException is thrown. So, you must be careful with your "has-a" and "is-a" relationships.

Java only implements one level of persistence, which is similar to isomorphism. To call it that, however, is inaccurate because isomorphism omits heterogeneous collections by definition. On the other hand, calling this level Polymorphic is not quite accurate either because the object on the receiving end must be of the exact type. So, Java's solution is somewhere in between the two. This means that you must ensure that the class definition of the sender/writer is identical to that of the receiver/reader.

For example, even though only the data fields are transmitted – ignoring any member code – the class methods cannot change from sender to receiver. This is because the class signature is written in the serialized string and validated during the readObject( ) operation. If the locally defined class differs from what was sent, an InvalidClassException is thrown.

Another programming requirement is that any serializable classes must have a no-argument constructor, either by code or default. This is because the readObject will reconstruct the object by calling the object's constructor with no parameters. If no constructors are implemented, then the default constructor works just fine, since it does not take parameters anyway.

Java's Externalization

The subject of serialization would not be complete without at least mentioning externalization.

Java's externalizable interface is simlar to serialization, but it's no easy task to implement it. The programmer must write all the code that serializes and deserializes the object or structure. Externalizable interface was designed to give the programmer complete control of the format and content of the serialized string. Describing this type of detailed customization is beyond the scope of this article.

SERIALIZATION IN PRACTICE

With the knowledge of the concepts and terminology object persistence, you are prepared for some practical application.

Let's look at a simple application example that consists of two programs: one saves invoices in a file with data collected from a GUI dialog frame; the other reconstructs them. Reconstitution has a specific name in Java: deserialization.

To see complete code listings for both programs, visit candle.com and go to the CCR/Publications page.

When program #1 is run, it will display the frame that functions as a dialog box. The user can enter the invoice information and then choose "OK" or "Cancel." If "OK" is pressed, the invoice object is serialized and saved to a local file. The object type at the center of attention is Invoice.

When program #2 is run, it will read the serialized object from the same local file, deserialize it, and display it in the same dialog frame where the user originally entered it. The beauty of this

is that an object living in memory can be saved in one function call without any of the code ever knowing what is in the object. It is not necessary to create new classes or objects for serialization.

The serialized container in program #2 exists for the purposes of the application and not for serialization. Therefore, the only extra code imposed for serialization is the implements clause on the class definition, the file stream instantiation, the object stream instantiation, and the writeObject( ) call. Very little code is actually dedicated to the saving process. Even if the basic Invoice class grows to reference other objects the serialization code does not have to change. When that structure is read back in, it will comprise all the same references that it originally did with the memory addresses automatically relocated. This is where the no-argument constructor comes into play.

Program #1

The main function does four things: (1) instantiates an object of the Invoice class, which will later be serialized; (2) instantiates a dialog frame object into which the user will enter the invoice data; (3) waits for the user to press either "OK" or "Cancel" (if "OK" is pressed then the entered data are transferred from the dialog frame to the object container); (4) serializes and saves the invoice with the saveObject( ) function. The dialog box used in both examples looks like the one in Figure 3.

Program #2

The code is responsible for reconstituting the object saved in the file. It does four things: (1) reads in and deserializes the object; (2) instantiates a dialog frame object into which the user will enter the invoice data; (3) populates the dialog frame with the saved data; (4) waits for the user to press either "OK" or "Cancel."

CONCLUSION

Programmers faced with the challenge of saving an object's state will immediately benefit from Java's serializable interface. As explained in the example, it makes persistence easy by eliminating the hassles of creating and populating data streams apart from the application. The objects that naturally belong to the application can be saved or transmitted without transformation into intermediate containers. This means you can design applications to pick up right where they leave off with very little code. Serialization not only benefits the application programmer, but also the consumer who must use the application day after day.

The ability to do this for an entire object structure is what makes serialization such a powerful feature. It has been said that Java, unlike C++, was built as an object-oriented language from day one. Serialization is further proof of that statement.

Ray Walker
Software Engineer
Candle Corporation

Bibliography

Java in a Nutshell, David Flanagan, O'Reilly & Associates, Inc.

Java Beans: Developer's Resource, Prashant Sridharan, Prentice Hall PTR

Rogue Wave Software: User's Guide, for "Tools.h++ Foundation Class Library for C++ Programming" (Comes with software)

Dr. Dobb's Journal, issue #264 April 1997, "Java & Persistent Objects" by Cliff Berg.

C/C++ Users Journal, issue Vol. 16, No. 9 August 1998 "A Simple Persisence Scheme: Store Objects Effortlessly" by Alberto Florentin.

Securing Messaging Middleware Applications (continued from page 1)

applications and data becomes exponentially more difficult.

What additional considerations are necessary in the brave new world of networked applications where, for example, the home PC-user can access a bank's account applications and trading systems? This article offers background and instruction on the implementation of security where messaging middleware connects functions in a networked application. Throughout the text, you will find terms specific to IBM's MQSeries. However, the implementation described could apply to any commercial messaging application using similar middleware. There are multiple parts to this article; Part 1 is presented in this issue. Parts 2 and 3 will cover Digital Signatures, Digital Envelopes, and Key Management using Digital Certificates. They will appear in a later issue.

THE VULNERABILITY OF NETWORKED APPLICATIONS

Messaging middleware plays a growing role in implementing networked applications, while leveraging existing legacy code. In part, this is because middleware hides the complexity of designing and coding the communications layer, even between disparate operating systems and hardware platforms. It does this via an API to all functions required to communicate between different components in the networked application.

IBM's MQSeries and Microsoft's MSMQ, for example, provide asynchronous messaging through a protocol free of connection. That means a communication link between a message's sender and receiver does not have to be active for an application to issue a "send message request" successfully. This is achieved through message queuing, a function which places a message where it will eventually be picked up by the middleware and sent along the communication path, or channel. In MQSeries, these "pick up" areas are called transmission queues, and their processing is transparent to most applications. A message could potentially be queued on any one of several intermediate nodes before reaching its final destination. While this allows for seamless communication, it also makes the message much more vulnerable to security breaches.

This environment differs significantly from a typical 3270-based mainframe transaction. Here, an application's actions are based on a synchronous communication of user input directly to application code. A security package has likely authenticated the user by requesting a password before transaction processing can continue. This authentication includes well-defined access control which determines what resources the user has permission to access.

Compare that to the networked application where communication between different components occurs through asynchronous messaging. The

On encryption: c = m e mod n

On decryption: m = c d mod n

Where: c = message ciphertext

m = message clear text

e = public key exponent

d = private key exponent

n = private/public key modulus

Public Key = (public key exponent, modulus)= (e, n)

Private Key = (private key exponent, modulus) = (d, n)

Figure 1 – RSA approach to the Public Key Cryptosystem.

user logs on to the application and initiates a transaction which will eventually be processed somewhere else in the networked environment. Security software on the initial platform authenticates the user. But how can the component of the application on the target platform gain access to resources it requires based on the permissions granted to the user on the initiating platform? Also, how can the target application code be assured that any data sent with the transaction has not been altered or observed across the communication link? And can the user who initiates the transaction be authenticated at the target end of the link? These are basic questions that any security implementation must address.

Administration of resource access level security to satisfy networked application requirements is inherently complex. This is due to the disparate security packages which exist on different platforms such as RACF, ACF2, and Top Secret on mainframes; and Object Authority Manager (OAM), Windows NT Security Manager or UNIX Security on networked platforms. Mainframe environments are well known for providing industrial-strength access controls, tried and tested over many years, to system resources. But networked platforms, while offering some access controls, are known for being significantly less robust in providing security.

This presents a major problem for securing networked applications because a transaction is typically initiated in the network on a PC or workstation and may eventually end up executing on a mainframe. The result is an Achilles Heel in any security scheme for networked applications based purely on access controls. That's because the overall strength of the security is only as strong as the weakest access point to the enterprise. This weakness has been recognized for many years, but the proliferation of networked application and messaging middleware development has only recently increased the priority to implement strong distributed security. The remainder of this article will be devoted to describing where security breaches might occur and how you can implement security measures in an asynchronous messaging environment.

OPPORTUNITIES FOR ATTACK

Distributed security demands the same level of access control as the non-networked environment does. For example, MQSeries offers the ability to secure Message Queues and the Queue Managers. You would implement this security through SAF-based RACF, ACF2 and Top Secret on MVS; and through the OAM on other platforms. An application's security level is checked when the application attempts to connect to a Queue Manager and subsequently if it attempts to open an object the Queue Manager manages, such as a message queue or process. The request to use such resources succeeds or fails based on permissions of a user.

This approach seems reasonably secure since virtually nothing can compromise security if a Queue Manager or any of its objects cannot be connected to, or opened. As long as the user keeps the log-on password private, MQSeries can ensure that no one is accessing its resources inappropriately. The OAM uses the underlying security mechanism via Access Control Lists (ACLs) to secure all named – and some unnamed – objects. MSMQ also uses this mechanism on UNIX; the OAM uses its own equivalent of ACLs.

But once you go beyond the connection to the Queue Manager and the opening of a Message Queue by an application, opportunities for attacks arise. In a networked application, where messages travel between platforms, any messages sent to a remote platform must pass through a communications channel. As described earlier in this article, MQSeries uses a connection-free protocol: the sending platform channel need not be active for an application to receive a return code from its MQPUT operation (MSMQ equivalent is MQSendMessage). To review: MQSeries places messages destined for remote platforms on a Transmission Queue (called the Internal Outgoing Message Queue on MSMQ). The Transmission Queue is associated with a remote platform message queue, a definition of the remote queue on the local platform, and a channel between the two. If the channel is inactive, any messages requiring the channel will remain on the Transmission Queue until the channel becomes active.

This allows a determined attacker to view messages on the Transmission Queue and alter the contents, compromising message privacy and integrity. Any changes to the message could then lead to inadvertent and possibly destructive actions on receipt of the message. Also, the attacker might alter the user ID in the message header and impersonate an authorized application user. This would allow the attacker to avoid security access controls on the Queue Manager and Message Queues.

Even if a message is not compromised on the Transmission Queue, the attacker still has other opportunities to infiltrate the security. The next one presents itself through "sniffing" the communication channel between the sending and the receiving platforms, or an intermediate node. Many techniques exist to observe messages transmitted across communication links. Software to accomplish this is freely available on the Internet. For example LOpht Heavy Industries, at www.lopht.com, builds an application to test system security. Amazingly, an attacker can use it to compromise message privacy, authenticity, or integrity if messages are not properly secured beyond access controls. A common attack is to insert entirely new messages directly into the channel. This completely bypasses access controls implemented on the sending platform.

The receiving platform is another spot vulnerable to an attacker. Receiving Queue Managers resolve the queue name to which a message is sent and, if the queue is local to that platform, place the message on the destination queue. The security weakness here is similar to that of the Transmission Queue, because the application that reads the message may not be active. That means the message can be compromised the same way it is on the sending platform. It's a sitting duck, so to speak.

A security weakness also exists for messages delayed on intermediate nodes during transmission when the forward communication channel is inactive. These nodes could even be on somebody else's network! While a message's exposure on an external network is obviously a threat, it is interesting to note that most reported attacks occur within the local network. MQSeries resolves destination queues based on Queue Manager name and queue name. If the destination queue is on another platform, the Queue Manager will search for a queue with the same name as the destination Queue Manager. This will be a Transmission Queue that will hold a message on an inactive channel until that channel becomes active. This gives an attacker an opening on a platform that is neither the one where the message started or the one that is its final destination. It means that intermediate nodes handling messages are particularly vulnerable to attack if there isn't very tight access security at either platform. Chances are, a platform may be left unattended since access security is typically focused on the end points of a message communication path. The unattended platform provides easier access to the messaging network for the attacker.

It is possible to set an "alternate" user ID in an MQSeries message. This ID is designed for access security to resources on the receiving platform. Networked environments typically have multiple user IDs defined across them for a single user. The alternate user ID allows an application to set the appropriate user ID for access to remote resources. The MQSeries access controls regulate the alternate user ID. But even if the ID is accepted, it can still be reset in transit as a different ID for resource access on the target platform.

If setting access security controls and using the alternate user ID fail to secure a networked application, further steps must be taken in the design and implementation phases to ensure message security.

SECURITY BEYOND ACCESS CONTROL

Aren't access security controls enough? The answer to this question, you may have guessed by now, is almost always no. To ensure security beyond simply access to resources, an application must deploy a security mechanism which allows an application receiving a message to answer questions in the following categories:

- Authentication: Are users, or systems, associated with messages really the users, or systems, that sent the messages?
- Integrity: Is message content exactly as originally sent?
- Privacy: Is the information in the messages, or parts of the messages, readable by the intended recipients only?
- Non-repudiation: Can senders of the messages later deny that they sent them?

The four categories – Authentication, Integrity, Privacy, and Non-repudiation – are basic security capabilities that must be available to any secure application. To correctly answer the questions in each category, a secure application must use software or hardware that implements cryptographic techniques. A widely used technique in today's secure environments is Public Key Cryptography.

Diffie-Hellman and RSA

In 1976, Whitfield Diffie and Martin Hellmann invented this technique. The basic idea is that two keys are generated: a Private Key, known only to its owner; and a Public Key, which is related to the Private Key but mathematically different. This Public Key is distributed to all users who wish to communicate securely with the owner of the Private Key. By incorporating hashing techniques and symetric key encryption into this mechanism, the issues of Integrity, privacy and non-repudiation are addressed along with authentication.

In 1977 three mathematicians, Ron Rivest, Adi Shamir, and Leonard Adleman developed what is still today the most widely, commercially used Public Key Cryptosystem. In their approach (RSA), the Public and Private keys are connected by a mathematical relationship based on prime number theory and modulus arithmetic. The details of the mathematics are published in the Frequently Asked Questions (FAQ) section of the RSA Web page at www.rsa.com. The relationship between the keys, and how they are used to encrypt and decrypt, is encapsulated in Figure 1.

The strength of this encryption is that the derivation of the Private Key is considered mathematically unfeasible even with the Public Key known by multiple users or machines. The modulus is the product of multiplying two large prime numbers, i.e., numbers which can only be divided by themselves and one. An attack to obtain the Private Key based on knowing the corresponding Public Key would require factoring the modulus into its two prime numbers. In practice, a modulus is 512, 768, or 1024 bits long, with the longer key presenting a much larger range of possible primes to derive the modulus Primes from.

An interesting fact is the number of primes that can be represented by 512 bits is 10^{150} more than the number of atoms known in the Universe. This is why a longer modulus provides stronger encryption than a shorter one. Public Key Cryptography Key length is often referred to by the length of the modulus. But the keys are actually a combination of the exponent, which varies between the Private and Public Key, and the modulus, which is common to both keys.

Message privacy through Public/Private Key Encryption requires that the sender of a message has the Public Key of the intended recipients. The sender encrypts the message data with the Public Key and transmits the generated ciphertext. Upon receiving the encrypted message, the recipients decrypt the ciphertext using the Private Key only known and accessible to them.

A necessary characteristic of this type of encryption is that the message text is encrypted in blocks equal to, or less than, the length of the modulus of the Public/Private Key pair, minus an additional 11 bytes. That means in a modulus of 1024 bits, a message would be decrypted in blocks no longer than 1024-(11*8) bits = 936 bits or 117 bytes. Therefore, for large messages, the encryption process goes through many iterations to encrypt the entire message.

Since the key exponents – especially the Private Key – may be extremely large, you might ask how this encryption performs in commercial applications. For example, raising a 100-byte hexadecimal value to the power of the exponent value, which itself can be an extremely large number, requires specialized large number arithmetic, known as modular multiplication. This arithmetic is beyond the capabilities of standard, commercial hardware instruction sets.

However, you can optimize implementations of Public Key encryption algorithms by establishing the exponent as a small number, such as three. This leads to relatively fast Public Key operations also used for encrypting messages and authenticating the Digital Signature attached to a message. You will find more details about the Digital Signature in part 2 of this article, featured in the next issue.

Other Public Key Technologies

Beside RSA, there are several other Public Key technologies that are either available today or are being evaluated for commercial implementations. Recently, interest has been increasing in Elliptic Curve (EC) Technology. Its security relies on the difficulty of solving the elliptical curve discrete logarithm puzzle. Early results indicate that EC offers the same security as RSA using a shorter key – 160 bits versus 1024 bits. The reduced storage requirement makes a difference in small, limited-capacity devices that deploy a Public Key Cryptosystem. The storage reduction for implementation is not as great as the difference in key lengths suggests, but for a more detailed explanation, visit RSA Laboratories Web site (Technical Notes section).

An even more recent alternative to existing Public Key cryptosystems was presented by two IBM researchers, Ronald Cramer and Victor Shoup, in May, 1998. The security of their approach relies on the difficulty of solving the Diffie-Hellman decision problem. It also offers the advantage of being both practical and secure against adaptive chosen ciphertext attacks, which can be potential problems for today's commercial Public Key cryptosystems. But any new approach to Public Key cryptography must endure extensive scrutiny by the cryptographic community over an extended period of time before becoming widely accepted, especially in commercial applications. Because of this, it may be some time before the Cramer-Shoup Public Key cyptosystem is adopted commercially.

RSA Public/Private Key encryption is a form of Asymmetric Key encryption because the key used to encrypt differs from the key used to decrypt. Another technique is Symmetric Key encryption, in which the keys used to encrypt and decrypt are the same. Common terms for such keys in messaging environments are Secret Keys, or Session Keys. In the next part of this article, we will see how combining Asymmetric and Symmetric encryption provides Message Privacy in message queuing environments. Look for Part 2 in the January issue.

Laurence Hart
Development Advisor
Candle Corporation

ISSN 1071-2976

Read the CCR on the WWW and "talk" to the authors...http://www.candle.com