Partitioned Application

(a.k.a. Security Kernel, Insulation)

Abstract

The *Partitioned Application* pattern splits a large, complex application into two or more simpler components. Any dangerous privilege is restricted to a single, small component. Each component has tractable security concerns that are more easily verified than in a monolithic application.

Problem

Complex applications often require dangerous privileges in order to perform some of their tasks. These privileges are often only needed by a small part of the application, but the existence of these privileges makes the entire application dangerous. If any part of the application can be compromised, the dangerous privileges could be misused. As a result, it is very difficult to have confidence that these privileges won't be abused – the entire application must be free of exploitable flaws.

There are many examples of this. The IIS Web server requires administrative privileges in order to execute. It is a large, complex application in which mistakes are frequently found. Because it executes with administrative privilege, every one of those mistakes could potentially lead to remote compromise of the entire system. Similarly, the UNIX sendmail program is a large, complex program that runs with administrator privilege in order to be able to write e-mail into any user's incoming mail files. For years, errors were found in sendmail that would allow remote attackers to gain complete control over the system.

Even when this approach does not lead to compromises, it causes the quality assurance burden to be significantly increased. Developers and quality assurance engineers are never really sure that some minor change won't have some unanticipated effect on security. As a result, routine maintenance activities are often inhibited by the need to perform full-blown security assessments of even the smallest change.

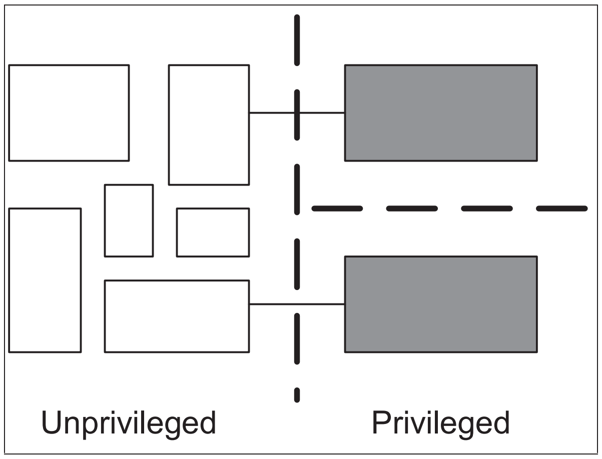
Solution

The *Partitioned Application* pattern divides a complex application into smaller components that minimize and isolate the elements that require dangerous privileges. Existing protection mechanisms are used to assign privileges to each component and prevent the various components from interacting with one another except through the published interfaces. The privileged components each provide some single privileged service to the rest of the application.

Each privileged component publishes an interface that the remainder of the program can use to perform privileged actions. For example, the postfix application (a more secure alternative to sendmail) contains a "maildrop" component that is responsible only for writing mail to user accounts. The maildrop component has administrative privileges, but it is a small component that thoroughly checks all inputs it receives and writes only valid data to valid mail user's mail queues. Because the maildrop component is so simple, it has been extensively analyzed for possible security problems. The remainder of the application can be updated as necessary without concern that the administrative privileges might somehow be misused.

There is no one-size-fits-all approach to developing a partitioned application. Much like the problem of assigning functionality to specific objects, the problem of developing an appropriate application partition is a craft. The models presented here are general suggestions; specific examples provide far more concrete details.

By isolating the privileged elements into one or more relatively small components, significant security issues can be restricted to just those components. The privileged components can be subjected to rigorous quality assurance, and the functionality of the non-privileged components can be updated with less worry about the security impact.



Issues

Depending on the partition, it may be important that one component be able to have confidence that it is talking to a specific component and not an impostor. This can generally be ensured using queues in which only a specific object is able to read from the queue. If each object has such a queue, mutual authentication can take place by using simple handshakes delivered through the queues. The appropriate queue should be read from a controlled configuration file, not accepted as provided by the caller.

A partitioned application can impact performance, as inter-process communication is introduced and synchronization is required. Benchmark and understand any performance penalties before committing to a partitioned design.

A partitioned application introduces logs and other intermediate message queues that can fill up; appropriate responses to jammed queues should be specified, implemented, and tested.

Installation / Configuration

This pattern also encompasses the installation and configuration scripts and files used to assign privileges and allow components to locate one another at run-time. Using operating system privileges, this could include the creation of user accounts, the establishment of privileges, and the deployment of scripts that ensure that all the components will start correctly and be able to find one another.

Installation of a partitioned application should address the following:

- The components need to be registered with the underlying protection mechanisms. Typically, this means creating multiple user accounts for each of the major components.
- The components will each have a configuration file that identifies each of the other components and interfaces. These names should be absolute, not relative. Specifically, never rely on the PATH or other environment variables to locate a component – hardcode the entire pathname into the configuration file.
- Create any resources needed by the privileged components. Changes the access rights on all the resources and configuration files so that they can only be accessed by the owning account.
- Configure the components so that they will run under the appropriate identities.
- Install a startup script that permits each of the privileged components to be brought up in the appropriate order.

A partitioned application will often require that components are initialized in a particular sequence. Determine the correct startup sequence and ensure that it is enforced. Depending on the design, it may be necessary to stop the entire application if any one component fails to start.

Possible Attacks

There are a number of possible attacks that could be perpetrated against this pattern:

- Direct invocation of an internal interface – early versions of postfix contained an internal interface that if manipulated directly would allow privileged commands to be invoked.
- Race condition problems – if an attacker can deliberately slow some component, it may be possible to cause inconsistencies that never appeared in testing. Partitioned applications are susceptible to race condition bugs. There is an entire class of security vulnerability called "time of check, time of use" (TOCTOU) errors.
- Man-in-the-middle attack – numerous mistakes have been made in which applications invoke other components by depending on an environment variable (or the path) to locate the appropriate file. When this approach is used, it is possible to intercept the invocation request and cause the invoking application to start an untrusted process. From this it is possible to conduct a man-in-the-middle attack.

- C++ language protections – when partitioned applications are built using the C++ language protection mechanisms, it is possible to bypass these protections and access private functions directly. A number of advanced C++ programming texts provide guidance for directly manipulating C++ virtual function tables.

Examples

There are numerous examples of the *Partitioned Application* pattern. Interestingly, examples can be found at different abstraction levels. This section presents examples at the code, system, and network levels.

In addition, we will eventually include the code from our password management object from our Web repository application as an example of a partitioned application. The Java object for password management will be called by less trusted servlets and will be the only object able to access the application password file.

Code Level

At the code level, many object-oriented programs are constructed as partitioned applications. Most applications written in an object-oriented language use the Private keyword to hide implementation details of an object. These features are useful in protecting against unintentional errors. For example, a dangerous function call can be wrapped using an object that checks consistency before invoking the function. *These features can often be circumvented by a savvy attacker though.* In C++, for example, the application as a whole must have the privileges to execute the dangerous instruction, and there is nothing preventing some other part of the code from invoking the function.

Java, on the other hand, protects objects from one another and allows specific privileges to be granted on a per-object basis [2]. Thus, Java can be used to develop partitioned applications if object access modifiers are used appropriately. This makes Java less susceptible to the attacks that C++ programs are.

System Level

Kernel-based operating systems are the archetypal system-level example of partitioned applications. Operating systems must be able to execute privileged instructions and directly manipulate the hardware resources of the machine. But operating systems also contain millions of lines of code that perform no such privileged work. Operating system design has embraced the notion of a kernel, in which only a very small subset of the system runs privileged code [5]. Functions such as direct manipulation of the system memory tables and swapping of running processes are performed within the kernel. Other functions, such as managing the file system, run as unprivileged code but are able to make carefully controlled requests of the kernel.

As a counterexample of this, Windows NT v.4 moved its entire graphics and user interface subsystems into kernel mode code, where any bug is a potential breach of system security. The rationale given was that performance was increased, and any bug in that code would eventually lead to a system crash in either case [4].

Another set of applications that are partitioned as described in this pattern are replacements for sendmail. Sendmail is an extremely complex UNIX mail processing system that has been the home of innumerable security-relevant bugs. As a result, several replacements to sendmail have been developed, most notably qmail [3] and postfix [1]. Both of these programs are instances of the *Partitioned Application* pattern. The "maildrop" elements of the program, which need access to any user's e-mail queue, are carefully restricted to very small, separate executables that have been subjected to intense scrutiny.

Network Level

By definition, most distributed network applications are partitioned applications. While they are not usually partitioned for security reasons, security does impact their design. Most client-server applications place all of the low-risk user interface activities on the client and ensure that the server will only execute authenticated, validated transactions.

Web front ends to banking applications are very common examples of network-level partitioned applications. Many Web systems isolate the application that performs electronic funds transfers or credit-card transactions on a separate system. Access to this system is through a minimal interface, often implemented using an internal firewall. While the rest of the banking application must still be careful to ensure that only valid transactions are initiated, the partitioned model guarantees that audit rules and certain safeguards cannot be circumvented. For example, charges for negative values can be discarded, a shipping address can be validated against the billing address, and the credit card number can be prevented from ever being disclosed to a client.

Trade-Offs

Accountability	No direct effect.
Availability	In some cases, this pattern can increase availability. For example, the sendmail replacements can continue to receive mail even if an internal processing component has died. However, the interfaces between components might also introduce new avenues for resource consumption attacks that result in a denial of service. In addition, the increased processing overhead can also affect availability adversely under heavy load conditions.
Confidentiality	This pattern can increase confidentiality by simplifying the security-critical aspects of the design and containing the damage that could be caused by many bugs.
Integrity	See Confidentiality.
Manageability	It is generally harder to deploy a partitioned application appropriately than its monolithic cousin. Each of the interfaces between the components must be understood by the administrator in order to debug problems or avoid dangerous misconfiguration. It can be very difficult

	to ensure that elements are all started in the correct sequence. While installation scripts can help ease this burden, they cannot eliminate it entirely. The maintenance of a partitioned application can be easier though, if the components and interfaces are well designed. Monolithic applications can be exceedingly difficult to maintain and enhance; a partitioned application could be easier to understand due to its modular structure.
Usability	No direct effect.
Performance	The primary drawback of this pattern is its impact on performance. Each of the restricted interfaces introduces overhead and processing delays. For example, the message queues used by sendmail alternatives introduce significant processing overhead.
Cost	A partitioned application can have both negative and positive affects on development costs. Structuring an application around security requirements can have an adverse impact on other development priorities. Complex installation scripts and documentation requirements introduce further costs. But enforcing a discipline of application decomposition can permit more focused testing and greatly improve quality assurance.

Related Patterns

- *Log for Audit* – a related pattern alerting of the need to maintain consistent logs across components.
- *Password Propagation* – a related pattern stating that, where possible, trust in any single trusted proxy should be minimized.
- *Server Sandbox* – a related pattern advising that sandboxes be built around individual components of a partitioned application, such as servers that should not be trusted.
- *Trusted Proxy* – a related pattern where components protecting critical resources are viewed as trusted proxies.

References

[1] Blum, R. *Postfix*. Sams Publishing, 2001.

[2] Gong, L. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[3] Nelson, R. "The qmail home page". http://www.qmail.org/top.html, June 2002.

[4] Solomon, D. *Inside Windows NT Second Edition*. Microsoft Press, 1998.

[5] Tanenbaum, A. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.