

Scenario-based Security



By: Derek Mezack

Vault Ecommerce - <http://www.vaultecommerce.com>

In the following document I will describe the evolution of digital security technology as well as the threats posed to our organizations. This will include the identification of technology weaknesses and the need for a new method of security modeling for both developers and security professionals, named Scenario Based Security Modeling.

The Evolution of Security Technologies

There has been a shift over the last 10 years in the way we value digital security. In the past generation, we have seen the emergence of various “detection” technologies, including signature and anomaly detection systems, to improved technologies for access control for both data in transit and end point security. With all of these advances and with many buyers, one would wonder how threats to our organizations would have a chance.

At the time these technologies were first conceived, organizations across the world primarily utilized computers for the features provided through commercial technologies, including popular operating systems and commercial applications shared millions of users, such as email services and mainframes. Popular detection technologies are often times effective in both detection and protection of well known attacks through their massive attack knowledge-bases. Protection technologies provided substantial security value against remote attackers, while admittedly having issues with preventing internal malicious threats. It is both with these strengths and weaknesses in mind that threats evolved as technologies role also did in our organizations.

While yesterday’s service technologies still power our world’s IT services, today’s web applications and other proprietarily developed technologies have become the primary source of both our revenue and our concerns for data security. While compromise of an email server or other popular commercially purchased application still poses a cost threat to our organizations, the threat of data asset compromise, usually represented by a database interacted with by proprietary technologies has proven astronomically more severe to both company reputation and liability.

Unfortunately, this shift in organization’s use of computing technology increased the exposure of security technology weaknesses. Detection technologies, depending on their many signatures of popular attacks to commercial technologies, only contained a weak handful of signatures capable of detecting proprietary technology threats.

Protection technologies, usually depending on access control, also suffered. Network services, identified commonly by a port and/or protocol were used usually for a single purpose, such as email, and could be controlled and protected substantially well by access control lists offered in protection technologies. With the exposure of web applications

Copyright © 2007-2012 Vault Ecommerce, Derek Mezack

<http://www.vaultecommerce.com>

All Rights reserved

over the Internet with a variety of capabilities and users types all over a single port and protocol, network access control has become inadequate.

While some technologies have emerged to provide some visibility or access control to our assets, such as linguistic detection systems that detect such things as credit card numbers, social security numbers, PHI and other content, with no tie to our applications to understand the proper use of this data they provide, misuse still goes undetected. For example, a medical web application providing capabilities to Dr.'s that normally must interact with Patient Healthcare Data (PHI). Technologies that have emerged in providing visibility to such content are limited in value by the fact the presence of the content is a part of normal business operation without issue. Instead, it is the *intention and target* of the data's use that can warrant threat.

In recent years, improvements have been made to development technologies to attempt to mitigate emerging threats to proprietary technologies and specifically web applications. Such advances as Data Type Safety and improved validation have aided the reduction of such popularly known attacks as SQL Injection. However, the evolution of threats has led to a new focus by attackers on logistical flaws in applications.

Logistical flaws in applications have many forms, but fundamentally they all exploit exposed points in the application that offer functionality without fully protecting the intention and target of their use. Consider the following popular scenarios:

1. A web application offering services to multiple mutually confidential companies. Once authenticated, the application offers a variety of services interacting with proprietary data to each company. In order to provide reusable code for these services to all companies, the developer chooses to expose a `companyId` parameter to each function. A malicious user simply needs to change the value sent for this parameter to random numbers to see other companies' data. This logistical flaw is very popular in web application in the form of cookie exploitation or direct exploitation of such parameters.
2. Many new credit card gateways use application web services as transaction gateways. A gateway using distributed web services maintains session credentials and privileges in a session object. This "Session" object is used as a parameter to the functions on the web service to maintain session security. The object has a public property, possibly named "IsValid", that is checked by the web service functions for validation, assuming the actual username/password has already been checked by the source application. An attacker takes advantage of XML serialization to simply construct a Session object with the "IsValid" property set to true, bypassing credentials. They then call these backend web services directly to make thousands of transactions.
3. A medical web application allowing doctor allowing doctors to view patient information and make entries concerning their care. A Dr.'s login is stolen to the application by key-logger, mal-ware, or other cause. The credentials are then used by attackers to access thousands of patient records in minutes for identity theft or malicious intention.

In all these cases, intrusion detection technologies are highly unlikely to provide visibility to the described threats because they involve legitimate use of exposed services with the wrong intention or target.

The issue in all these cases is the same; a lack of cohesive protection and monitoring of the capabilities applications offer while interacting with our most critical data assets. The market demand for “turn-key” and “appliance” security technologies has only lead to an increased ignorance of this fundamental problem. While security technologies try to bring solutions to market that require no configuration, this has come with the necessity that he products “protect” without any knowledge of our assets or the capabilities of our applications. Without this knowledge, security technologies cannot effectively prevent malicious misuse, whether by direct logistical attack or theft of credentials. This is demonstrated even by Data Type Safety by its attempt to prevent compromise by enforcing rules around the interpretation of the SQL Language, not the capability the application wishes to offer. While using this security feature is an excellent way to prevent technology based attacks, like SQL injection, it would not prevent any of the above listed logistical threats.

For this reason, the need for those who develop proprietary technologies and those that monitor threats to their operation to also advance in meeting this new generation of threats targeted at them and the critical data assets they interact with.

Logistical security flaws often have their roots in the way developers and security personnel view their best means of protection. The foundation of this view is usually described by the “Security Model” and is often the first thing both security personnel and developers learning necessary security for programming learn. Almost always, the first and sometimes only security-model both developers and professionals learn is the ACL (Access Control List) Model. The ACL Security Model is shared in use by the entire popular operating systems used worldwide, including Windows, Unix/Linux, and Macintosh. Applications almost always run on top of standard operating systems. For both developers wishing to write applications for these operating systems, understanding the security model is necessary in developing applications that interact with the OS and hopefully extend its protection to control access. For security personnel, it is necessary to understand this model in order to understand how to configure Operating Systems with least privilege access and monitor operations.

It is my opinion that the lack of understanding of alternative models has lead both security personnel and developers to continually fall victim logistical vulnerabilities that lead to the compromise of critical assets. For this reason, I will explain both the weaknesses of the ACL Model and propose an alternative that I believe will lead to improved posture and resilience by both developed applications and monitoring personnel to these risks.

The point of a security model is to describe how a system will safely prevent, limit, grant, and revoke the use of a system’s functionality. Operating Systems are designed to be

foundations to allow the use of computer resources for arbitrary functions. Unlike toasters, computers are used for a multitude of different purposes, each with their own applications. The ACL model describes a system of control of general resources through the use of access lists each resource has, identifying the “users” that can “access” them. The types of access are general to such things as read, write, execute, and delete.

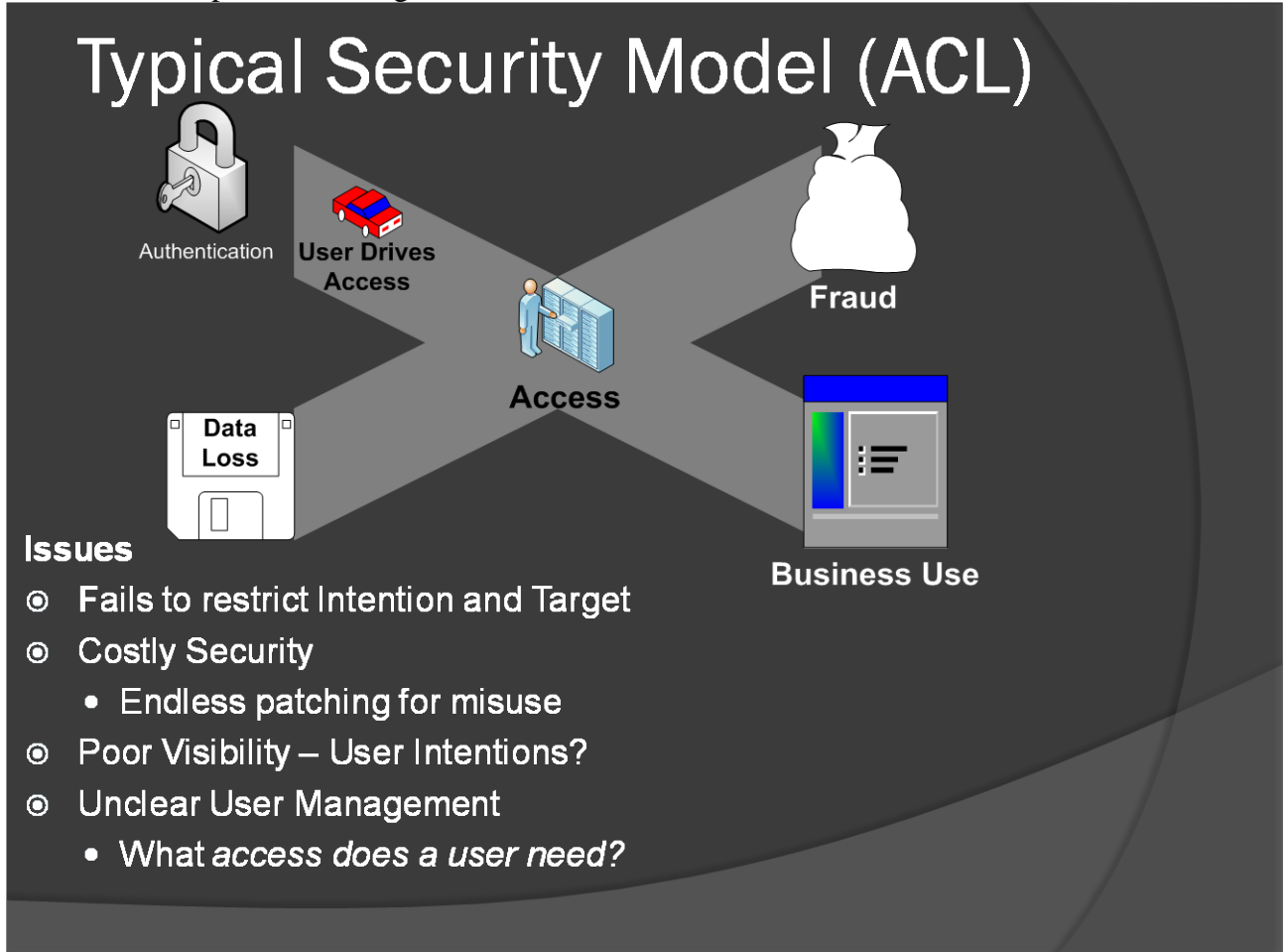
Most all of us are familiar with these systems because many applications that we are familiar with, such as mainframes, database management systems, and web application user management systems mimic these same features. Since an operating system itself has little knowledge of what applications will be used, the ACL model provides a very general means of controlling access to resources. If an operating system required any more knowledge of the applications running to provide its “model” of security, it would be unable to support arbitrary applications as well as they do.

As fitting as the ACL model is for providing a means of providing security through general resource control, its qualities are too general to provide protection from attacks related to misuse. More specifically, this model fails to address the *intentions* and *target* of users. In addition, the concept of “users” in itself suffers from issues when considering the many situations applications must act with higher levels of access than the clients using them, such as network services and web applications. In all these cases the primary weakness is that *programs access objects, not people*. This fundamental risk has been commonly known as “confused deputies” [2]. This distinction is important for the following:

- The same person can run with different user identities (logins).
- The same program can be run by different users at different times with different authorities.
- Special programs sometimes need to have more authority than the user does.
- People do not always know which programs they are running.

The granularity of the control offered by the ACL model is also limited to addressable “resources”. In the case of operating systems, this includes files, networking, pluggable devices, etc. As we know, this is often times insufficient with single files containing thousands of records with different levels of protection required or a single database connection used by a web application with clients of different levels and purposes. For this reason, it has always been left to the applications we use to provide the necessary extensions to this functionality to meet organizational security needs.

The below picture demonstrates the way the ACL model drives security as well as the inherent risks it presents having the user "drive access".



Scenario-Based Security Modeling

Scenario-based security is a concept in the design of secure computing[4] systems. A scenario (known in some systems as a key) is a communicable, unforgeable token of authority. It refers to a value that references a capability along with a single right to perform or use that capability. A user program on a scenario-based operating system must use a scenario to access any resources. Scenario-based security refers to the principle of designing user programs such that they encapsulate resource use according to, intention, target, and principle of least privilege of their purpose, and to the underlying resource infrastructure necessary to make such transactions efficient and secure.

Scenarios and Scenario-based security

Scenarios achieve their objective of improving system security by being used in place of forgeable object references. A forgeable reference (for example, a path name) identifies an object, but does not specify the intention the object is used for or the rights required to perform the task it is associated with. Consequently, any attempt to access the referenced object must be validated by the operating system, typically via the use of an access control list (ACL). ACL's are general to "access" and do not describe the intention of those using them. In addition, each ACL can only reference one object, which is often insufficient to describe a given operation scenario. Instead, in a system with scenarios, the mere fact that a user program possesses that scenario entitles it to *use* a scenario which may access multiple referenced objects in accordance with the rights that are necessary for its intended purpose. In theory, a system with scenarios removes the need for any access control list or similar mechanism by giving all entities all and only the ability to engage scenarios they will actually need.

A scenario is typically implemented as a privileged logical function that consists of a section that specifies access rights, a section that uniquely identifies the object(s) to be accessed, and logic that precisely defines the scenario for which the objects are used. Scenarios may also include required inputs, such as function parameters, that are used to offer greater granularity of intention, privilege set, and target in use. In practice, it is used much like a public library or API, in a traditional operating system, but to perform every task on the system. Scenarios are typically stored within an application, but may be registered for access control with any system, such as operating system, web server, or any other point that provides some mechanism to prevent direct access of underlying resources and some mechanism in place to prevent exposure of the contents of a scenario (so as to forge access rights or change any of the objects it points to).

Copyright © 2007-2012 Vault Ecommerce, Derek Mezack

<http://www.vaultecommerce.com>

All Rights reserved

Programs or users possessing scenarios can perform functions on them, such as passing them on to other programs, converting them to a less-privileged version, or deleting them. The controlling system must ensure that only specific operations can occur to the scenarios, in order to maintain the integrity of the security policy.

Introduction to Scenario-based Security

A scenario is defined to be a protected reference to a task which, by virtue of its possession by a user process, grants that process the ability to interact with one or more objects with a specific intention and target.

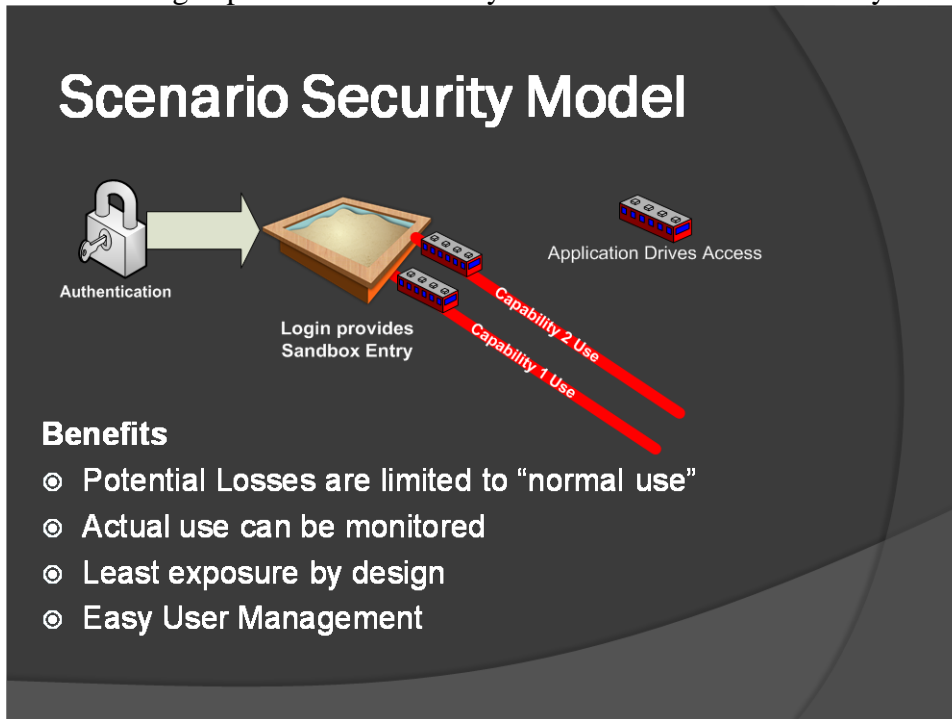
The idea of this model is demonstrated by such products as the toaster and microwave. In considering both of these products, we can see the following qualities of this model:

- Scenarios do not care who uses them
- Scenarios are specific to tasks, not generic to “access”
- Scenarios may use one or more objects to perform their function
- Scenarios use the principle of least privilege in their access of objects to perform their intended purpose
- Two or more scenarios can access the same object(s).
- Scenarios may require inputs, provided internally from the environment or as parameters to determine their appropriate performance and least privilege use
- Scenarios can be delegated

The fundamental idea behind this model is that a system should be designed as specifically a possible around the capabilities or tasks it provides. Using the toaster as an example, we can see how this device provides little means for its ability to be misused in its intended environment.

In addition to these qualities concerning Scenarios themselves, a scenario-based security system *prevents all direct access to resources except by the scenarios that use them.*

The following depiction shows the way this model structures security:



The primary difference between this and the ACL model, depicted earlier, is the "application drives access", rather than the user. The application knows the least access privileges required to perform its business function at runtime.

This differs from a "capability" as defined by the "Capability Security Model" in that this model accounts for runtime factors that may influence the access permitted. For instance, for the capability security model to be effective, each "capability" must be able to have all decisions concerning the access it requires decided regardless of things such as what user or other factors that can differ while the application is running and they are used. A scenario therefore differs from a capability by including both handling and security standards to receive and utilize runtime information, such as their user.

A scenario most typically is implemented as a function exposed by some interface for client use. The reason for this can easily be seen by the following example of a Scenario as a structure:

Scenario

Reference to one or more objects

```
Necessary inputs to describe required object privileges
Privilege set to the objects in light of inputs
Logic describing use and intention of the Scenario
```

Suppose that, in a user process's memory space, there exists the following objects:

```
TableName: Orders
TableName: Customers
TableName: CreditCardTransactions
```

Although this identifies unique tables in the system, it does not specify access rights or the intention and target for which these resources can be used and hence is not a scenario.

Suppose there are instead the following functions:

```
AddCustomer(CustomerName, CustomerAddress)
AddOrder(Customer, Product, OrderAmount)
MakePayment(CreditCardNumber, Amount)
```

These identify the objects along with the set of access rights to perform their functions. It, however, is still not a scenario because the ability for the user process to call these functions would provide direct access to the objects with specific privileges but without any specified intention for which they operate. Now suppose that the user program successfully executes the following statement:

```
Int NewOrderNumber = PlaceOrder(Customer X, Product Y, Quantity Z,
CreditCardNumber A)
```

The scenario function `PlaceOrder` is a reference to a scenario the process has the ability to engage in. Its existence in this call to the scenario function in a processes code assumes that the process does indeed have legitimate access to engage in the scenario, and therefore access to *use* the objects the function refers to in the way it describes. A key feature of this arrangement is that the `Orders` table and the privileges used to interact with it are not directly exposed. There is no means to “access” objects, only “use” them as part of engaging in a scenario. Another key feature that is not directly shown in the above arrangement is that the `PlaceOrder` scenario logic makes use of the minimal access privileges required to perform its intended purpose, following the least privilege model.

A final key feature of this arrangement is that the scenario function requires all the necessary inputs that make up its *use* when it is engaged. This effectively allows it to validate and determine its appropriate use, including the least privileges it requires.

Part 2: Application Scenario-based Security

Scenarios achieve their objective of improving system security by being used in place of publically exposed points of access, including forgeable references to protected resources or general access functions. By modeling all exposed points of access to an application to only provide the opportunity for users or user programs to engage in scenarios, misuse of resources can be effectively prevented.

Logic within defined scenarios may make use of unexposed general functions, allowing code reuse. As long as these functions are not exposed to the systems user or user program directly, such code may even be shared as libraries or other sharing techniques, but should be protected by the system from presentation layer access.

Adoption Criteria of a Scenario-based Security System

A Scenario-based security system requires that no direct access to objects/resources be exposed through its interface. The scenario based security model's application has been hindered over history in its practical code application and adoption to security operations by the overwhelming access given by most popular systems directly to resources. Most user programs in the past were accessed directly from an operating system user session or shell network services. These interfaces commonly provided user credentials in an ACL model that did little to prevent direct resource access by means outside the application. When Scenario-based security is implemented on a system with direct resource access, such as an ACL Model system, enhancements to application security pale in value compared for the need for operating system hardening and distributed architecture. For developers, this could be compared to trying to build a child-safe toaster that requires exposure of its wires and electric outlet to its user.

The evolution to distributed architecture was the first necessary step in removing this hindrance. However, problems still persisted at least for client applications that could be directly manipulated running as full operating system applications. The introduction and widespread movement to sandbox technologies, such as the web browser, added the final component to create a system capable of hosting Scenario-based security applications while protecting resources from direct access.

The web browser, at least in concept, provides a "sandbox". This can be effective for both the application, removing direct exposure of resources along with the code instructions and operating system that make up the architecture of applications. While shell network services and client side applications begin with user account access, providing accessibility to many of the resources of the underlying operating system, web browsers provide no direct access or accessible logic to an organizations system unless explicitly developed and made accessible by the presentation tier in the browser. A web

page that contains only <HTML></HTML> offers no functionality to the application whatsoever.

Although direct access to the same resources a scenario-based security system uses may be offered through a separate resource management interface, this resource management interface would require strong segmentation from the scenario-based security systems point of access. For instance, an Internet web application implementing a scenario based security system would not provide any means of resource access to the Internet except through the exposed scenarios access is permitted for engagement. However, underlying resource components, such as the relational database system or web server may provide local means of direct resource access to authenticated users of the operating system or other relational database management system for resource management.

Application Scenario Security Modeling

Benefits and Requirements

By combining scenario/task based encapsulation with a capability security model, developers can effectively reduce the exposure of any access and authority utilized by their applications to business tasks they perform. This combination, referred to herein as Scenario based security modeling, packages the use of access for program operation into protected capability functions, designed to perform the specific business task the application offers, with little to no room for misuse.

This model also provides more meaningful auditing by security operations, granting visibility to business tasks as they are performed, rather than bulk data logging access. By changing or augmenting the way we perform monitoring, analysis, and response as part of information security management to work cohesively with the capabilities and related assets of our environment, threats related to misuse and proprietary code vulnerabilities can be effectively managed. Since no research we are aware of exists as to the application of capability security models in information security management, we will later discuss Scenario Based Security Management with the same exemplary application to follow in thorough detail.

It would wisely be admitted that components utilized by developers which only provide ACL model security, such as popular operating systems, relational database management systems, and other components are unlikely to be expelled or redeveloped in scope of normal application development projects. Therefore, in order to attain the benefits described, while maintaining interaction with such components, a hybrid system must be adopted with addressed points of weakness. To make such a system easily adoptable, we will later discuss some fundamentals and techniques that will provide immediate improvement of security posture to both existing and newly developed applications built with today's popular components. References to more scientific and complete implementations of capability security models will be provided to allow developers to take things beyond the scope of the exemplary application model we will discuss shortly if desired.

In order to effectively take advantage of the scenario-based security model's benefits in an environment having components using the ACL model, these components must be hidden from exposure to un-trusted interfaces. Fortunately, both of these feats have already been substantially accomplished by the widely adopted distributed architectures and web based application technology. Currently accepted distributed architectures for web applications hold the following security requirements:

- No direct access to the data tier, such as database, mainframe, file system, etc. is permitted from the un-trust zone (usually the Internet).
- Interaction with the application is only provided over a web service, which offers no direct resource access to the underlying operating system hosting the application except those explicitly developed and provisioned to the thin-client (web browser).
- Strong segmentation is enforced between the 3 tiers, including the presentation, application, and data tier.

Starting with these well known rules, which have already been adopted by many applications, we will discuss both the fundamentals and techniques that still rely on the developer's implementation.

Scenario/task based encapsulation

The primary goal security-wise for a developer implementing a scenario is to encapsulate the specific logic used to perform a given task with the necessary authority to perform it. The product is the packaging of specific logic and necessary inputs to both encapsulate the intention and target of the task and minimize the access required to perform its function. The result of this packaging makes misuse of the final scenario substantially more difficult.

Presentation Tier Requirements

The primary goal of the presentation tier is simply presentation. Scenario based security modeling focuses on security *in the core*. What this means is that the presentation tier should almost operate as a pluggable façade outside of the Scenario based Security Model. In order to accomplish this, the following requirements must be met, of which some we've already discussed:

- The presentation tier must not contain logic requiring trust by the application tier. For instance, calls and parameters made by the presentation tier should never be allowed in design or treated in use any different than if a user themselves were typing them directly to the application tier.
- The presentation tier should provide encapsulation from the client's Operating System, as it likely uses the ACL Model. Sandboxes, such as a browser, provide potentially adequate levels of this encapsulation.

- If higher privileged technology is needed, all privileged functions should be as small as possible and only possess privileged in encapsulated functionality that specifically needs it. In addition, such encapsulated pieces of functionality should follow the same requirements described for the application tier in exposure. However, any efforts on the client side to wrap privileged functions into capabilities themselves should not make this code a trusted piece of the server side security model. This means that no matter how well wrapped, critical assets will not likely have adequate protection if ever passed to privileged pieces on the client. Some technologies, such as Java applet and un-trusted code, provide easy strategies using scope for least privilege operation while still allowing increased functionality over other HTML and scripted technologies.

By meeting the above requirements, an applications presentation tier can be treated as a pluggable façade, making the application tier the first point of security model interest, and allowing the presentation tier to function independently.

One of the most important aspects of implementing a security model is to avoid or explicitly manage integration with other security models. In the case of the first tier, the client or presentation layer represents the first point of potential exposure. Since operating systems do not share the same security model as we have designed, we must deal with this exposure. It is for this reason we earlier discussed the importance of the web browser adoption.

The web browser provides us a sandbox, which frees us from the overwhelming task of preventing our application from misusing local authority to the client. While client applications on a typical system share the immediate and implicit authority of the user, opening up a plethora of risks from the client application that must be overcome with recommended environment hardening and ACL configuration, the browser's default settings typically suffice the mitigation of such exposure. The browser, at least in theory, strips our application of all authority beyond basic presentation on the client system. By stripping us of all authority, we operate outside the security model of the operating system. Our application no longer shares the authority with the user, which the ACL Model of most operating systems is centered around. We will discuss integration techniques for dealing with such issues as ACL Model component dependencies later at other tiers.

Since we are not dealing with integration with the client ACL Model, it should be noted that client side code in this example implementation should not make use of logic to attain local privileges, such as Active X technologies, or other logic which breaks the sandbox provided by the browser. Such technologies may be successfully utilized in other implementations, but would require manual sandboxing to shield our application from the ACL model using some of the same techniques we discuss later in other tiers.

Obviously, there are aspects of this implementation that tend to favor thin-client architecture. This is simply because the web browser undergoes frequent and widespread

security testing in its ability to provide a sandbox on the vast majority of hosts in the world. For this reason, it is not likely that the undertaking of developing an alternative client that must also adequately sandbox its contained code from the Operating System and ACL model would be feasible for typical development project scopes.

However, the techniques utilized for integration with the relational database system, which typically uses the ACL model in commercial products, may also be used to successfully protect privileged client code. This is outside the scope of what we will discuss in this example.

Application Tier Requirements

The application tier represents the real entry point in our security model from un-trust to a certain level of trust. To begin with, the business requirements of an application should be examined to identify a list of the scenarios it must offer to its clients. These scenarios offer us a list of the first, and perhaps most important, point of encapsulation our application should use. For example, in considering an online banking application, we may identify the following tasks we will use in Scenario Based Security Modeling:

- A client should be capable of viewing their account information
- A client should be capable of making a deposit to their account
- A client should be capable of making a withdrawal from their account
- A client should be capable of transferring funds between accounts they own

There are some things we should notice about the scenarios we have listed. Each scenario describes both the specific functional task they perform and the target for which they operate. Each of these business tasks are then translated to scenario functions, one for one. These scenario functions will have the following requirements:

- The code for each scenario should perform the specific business task required without the dependence of any other scenario calls.
- Each scenario should utilize its own least privilege authority to perform the task it is designated for.
- Parameters to exposed Scenario functions are highly recommended to be Data-Use-Safe, discussed later. Generally, this means that no logic, such as parameter interpretation or other logic, that is not core to the performance of the task the scenario describes should share scope with the authority attained to perform the task. If such code exists, authority along with its core logic should be encapsulated and stripped of privileges from core scenario logic.
- No scenario should provide direct forms of access to resources for its users.
- Resources requiring subject/user access must be sandboxed before provision.
- Resources of the applications hosting operating system or data access tier must be encapsulated to their specific use as it related to the scenario accessing it.

Each scenario is required to be atomic from each other as this form of security is centered on accomplishing a single purpose.

Each scenario function exposes only parameters that are specific in both data type *and* business property, making them “Data Use Safe”. This is easily demonstrated by the typical toaster or microwave. The idea is as a developer is to make the inputs to authoritative code precise to the business and data intention, meeting its business requirement. Just as it would be difficult in normal or suggested environments to misuse a toaster or microwave, scenario functions operate in such a way that unwanted input simply can't fit.

Since all applications must typically provide resources or assets to clients in some form, capability functions must offer them in a sandboxed fashion as to avoid any access to underlying components that may not share its security model. For instance, the application tier must adequately perform data access for its clients, providing permitted data free of any ties to its source system. For this reason, data access languages, such as SQL, must be fully encapsulated from capability parameter inputs. Secure caching of asset or resource content at the data access layer would be encouraged as break the tie of externally prompted data access as a task from underlying components. This significantly reduces risk related to denial of service types of misuse.

Tier separation between the application and data access tier would be strongly encouraged in this model to aid in the prevention of direct influence from capability inputs or prompting.

This implementation assumes the backend data to be its primary assets. Users in some situations may also be considered assets. Consider a collaboration application that allows users to share content. Content may be safely provided and accessed by users utilizing the applications capabilities in this scenario based model. However inputted content that is Data Use Safe to the application may not be safe to the user. Thus, such things as Data Use Safety may also need to be applied to the output of capabilities.

Web applications typically have less to deal with as it relates to integration with the ACL model of its hosting operating system. This is because most web services now attempt to sandbox web service related applications themselves, stripping them of most privileges. However, any use of file I/O or other resources of the operating system does present risk and should be encapsulated the same way capabilities are as to prevent misuse.

Often times, data-driven applications consider only the database for its resources. Although databases often provide the most critical assets to our application, local resources to the application tier may lead to compromise of data access if their misuse can compromise the host of the application. Most internal (private) application functions must be free to operate without consideration of the above requirements to be efficient and practical in rapid development. However, internal functions that make use of process or login authority to access external resources; such as Operating System files and other resources or external web services, are encouraged to encapsulate this use as to reduce exposure from both developer misuse and tier bypass.

For instance, multiple vulnerabilities have been discovered over the years related to file output an application made use of for various reasons, such as logging, that could be misused to overwrite system files and compromise the Operating System. To ensure the safety of such functions, scenario based security modeling techniques can be applied to their own functionality. Instead of providing a function that generically provides File I/O access, the function should be specific, offering to write a specific format and type of file to a specific or restricted location of a limited size. General “utility” functions like this are often provided in shared libraries in development environments and used without consideration by many developers. Misuse of such functions further exposes misuse opportunities to attackers.

As another example, application web services have quickly become an adopted technology utilized in many distributed application environments. Some commercial components, such as PDF generators and other utilities, now offer web service packaging, allowing applications to freely call their functions through a standard web service connection. Unfortunately, many implementations of such services have been found to be exposed on the internet through the same hosting web service their applications, thus exposing direct call access from clients. The typical damage resulting from this form of tier bypass is often amplified by such utility functions that attempt to provide general forms of use without consideration of intention.

The encapsulation these functions should endure is the same as described by the data access tier, which they belong in terms of the Scenario-based security model.

Data Access Tier and ACL Model Integration

The data access tier represents privileged resource access of any kind from a security perspective. Although an application may primarily use the database as it’s only “resource” from a user perspective, resources include files, email, external web services, and any other forms of resources that may play a role in capability use. The data access tier is not exposed to direct client input in a secured architecture, but must consider exposure in the case of application tier compromise. For instance, if an exposed web server to the internet hosts both presentation tier content and the application process, compromise of this system through any means may represent direct access to the data access layer.

One benefit of the Scenario-based security model is that the technique of capability encapsulation can be replicated to the data access tier with often little impact to performance or efficiency. In fact, the data access tier as well as developers often benefit from this replication as a result of its organization and exposure impact.

The data access tier in many environments involves a resource management system that typically uses the ACL model for its function. Integration of the Scenario-based Security Model and these systems is crucial to its practical acceptance and benefit. Many ACL Model resource management systems encourage the creation of individual user accounts for each client identity to the application. With the growth of Internet applications, it can

quickly be seen that this is not practical in considering the number of potential Internet users. In addition, many of the protected resources these systems identify, such as tables in a database, host inseparable data of differing levels of access. For instance, an “orders” table may host records that can be seen by some users and not others. To normalize such data any further would often be impractical to performance and usability. Row level ACL options add further complication to user management, development, and support.

In order to integrate successfully with ACL controlled resource systems, resource access must be encapsulated in the same way application capabilities are, with fewer constraints. The goal of this encapsulation is not to necessarily prevent misuse, but to limit the scope of data access to the necessary provision each capabilities “use” requires. In considering a relational database management system, such as SQL Server, much of this can be accomplished by creating one or more user accounts that do not have any direct table access. Instead, all data access is provided through stored procedures related to each of the applications capabilities, and providing the specific data each capability necessitates. While compromise of the application tier may lead to unauthorized access to these procedures, the scope of damage can often times be drastically reduced by removing direct table access and limiting them to stored procedures. This also fits perfectly with the scenario-based security model, by allowing developers to write stored procedures that meet most of the application tiers requirements along with its security posture. The following requirements can be observed as it relates to data access tier functions in the Scenario-based Security Model.

- The data access code (whether via stored procedure or application SQL) for each capability should provide only the resources necessary to the capabilities function. This limit should be applied to resource content, frequency, and availability of resource access.
- Data access functions related to the same capability should share the same ACL Model Security Account if the resource manager operates on the ACL model. This account should have least privilege authority to perform the task it is designated for. In the case of a RDBMS, this account should have “execute” permissions to the necessary stored procedures for the capability only.
 - If the number of scenarios in the application makes user requirements for the resource manager infeasible, scenarios can be grouped when shared by application users of the same role. Interface user accounts to ACL controlled resources, such as the database, may also share a resource manager user account. While this does not offer the same level of security, it offers substantially more security than failure to implement this point in the model.
- Parameters to expose to the application tier should require little to no interpretation. This includes such parameters as long varchars that are exec'd as SQL by a stored procedure.
- Direct forms of resource access should be restricted to resource administrators, preferably not part of the application. This can be accomplished in RDBMS by

disallowing any direct table access from application related database user accounts.

- Resources of the resource manager's hosting operating system must be prevented or encapsulated to their specific use as it related to the capability accessing it.

Consider a medical application that requires a client to be able to search all patients and view individual patients in full detail. The data access tier would need to provide patient related resource data for the application to fulfill its requirements. However, the data access tier could be limited in prompting to providing a list of all patients with no detail, and provide a single patient in full detail per user within a certain time period. Assuming the client is human, fast access to patients in detail could be measured by typical use and limited as closely as possible to these requirements. Even such constraints as time can be implemented within the limited logic provided by stored procedure technology and would have excellent benefits when considering the impact of complete asset loss.

While these functions will rarely be one for one with the capabilities that use them, they do benefit from atomic design and more general forms of data use safety. By identifying and encapsulating internal functions that directly access resources, the damage resulting from tier bypass can be minimized and developer misuse can be made less likely.

Data Use Safety

Data Type Safety has significantly aided developers in reducing the risks to applications as it relates to SQL Injection and other forms of type exploitation. However, data type safety does not protect applications against logistical exploits and other point's proprietary points of input interpretations. As a result, many applications still suffer from misuse due to superfluous parameters or parameters whose interpretation can be exploited to exploit application flow or authority.

Data Use Safety is type system in which the values are structured specific to the business properties and task they represent. The goal of this system is to provide a meaningful process of encapsulating the intention and target of the parameters to our application's business capabilities to prevent misuse or exploitation.

When considering the validation of parameters to functions, there are two forms of assurance that must be met in order to protect "data use". There is the assurance of validity for each parameter as a business property or value to our functions. There is also the collective assurance of the business properties together for the capabilities intended use. For example, consider a banking application offering the capability of a client to transfer funds between accounts. Some parameters to this function may be the Source Account ID to transfer funds from, a Destination Account ID to transfer funds to, and a Money Amount to transfer.

To encapsulate the intention and target of these parameters, we may first validate each individual parameter as a Data Use Element. This could be accomplished by making each

parameter an object, representing the business property it represents to the capability, and offering implicit validation on value set to ensure its proper use. For instance, the Account ID object could include validation of the following:

- Account IDs data type safety - Ensuring the ID contains no special characters, etc, that may pose a threat to backend systems, such as a database in the case of SQL Injection.
- Account ID business property safety - Assurance the Account ID is of the proper format of system Account IDs in the system.

Another example of Data Use Element validation could be demonstrated when adding or importing a new customer's information. The customer may fill in information to a form that includes their address. Suppose the name or address includes such things as first and last name, street name, number, city, and zip, put together as a single item. Combining elements like this often times requires superfluous characters, such as space, comma, and other special characters to be allowed as input to put elements together. Attackers often take advantage of such inputs to allow many forms of injection, such as cross-site-scripting and SQL injection. Some times, normalizing items like this is not as simple as splitting interface inputs up. For instance, search screens often must handle dynamic input strings that cannot escape normalization and interpretation. In such cases, much of this normalization can be handled in the "set" functions of these Data Use Element objects. By handling this interpretation here, we encapsulate dangerous routines, such as parsing, from our capability functions that utilize any authority.

Although this validation may seem sufficient, there are many situations where valid parameters may be used together to perform malicious tasks. For instance, what if valid source and destination account IDs in addition to denominations are passed, but the existing destination account or source account does not belong to the client and has no permissions for them to transfer funds from or to. Another possibility is that the bank has a policy that accounts that do not belong to an individual can only have a certain denomination of money transferred to them, due to tax reporting or other business requirements that exist. In such a system, correct use can only be determined fully by examining the properties passed to the capability together.

For this reason, an additional encapsulation must be required that is not validated until within our capability function and may make use of privately passed information, such as a database connection, session object, or other privately held information the function may have in order to provide collective business validation.

In order to ensure data use safety, individual property value assurance is provided by DataUseElement encapsulation, while collective use assurance is provided by a ScenarioUsePropertyBag.

To demonstrate a framework of for Data Use element in a modular fashion, the following code is presented:

```
using System;  
using System.Collections.Generic;
```

```

using System.Text;
using System.Collections;
namespace AppShared
{
    public abstract class DataUseElement
    {
        private string sScenarioParamName = null;
        private System.Collections.Generic.List<ElementValueDef>
alValueTypeMap = null; // Populated by subclasses with the map of values and
their types this parameter will hold
        private Dictionary<string, object> dictSafeValues = null;
        /// <summary>
        /// Initializes a new instance of the DataUseElement class.
        /// </summary>
        public DataUseElement(string sScenarioParamName)
        {
            this.sScenarioParamName = sScenarioParamName;
            if(!InitalizeValueTypeMap()) {
                throw new Exception("The definition of values this element
represents could not be initialized.");
            }
        }
        /// <summary>
        /// This function is overridden by subclasses of DataUseElement to
populate the map of values this parameter
        /// will have and their respective types. Each value may be identified
with both a DataType and a BusinessType
        /// for respective validations
        /// </summary>
        /// <returns></returns>
        protected virtual bool InitalizeValueTypeMap() {
            alValueTypeMap = new List<ElementValueDef>();

            return true;
        }

        /// <summary>
        /// This function is overridden by subclasses to provide parsing and
normalization
        /// of raw parameter values, if necessary into a ArrayList of their
values for validation
        /// This may do such things as parse a First and Last name in a single
string into two string, separate a zipcode from a address string, etc
        /// </summary>
        /// <param name="bSuccessfulParse"></param>
        /// <returns></returns>
        protected virtual ArrayList TranslateSetParamToProperty(ref bool
bSuccessfulParse, object Value)
        {
            ArrayList alParsedValues = new ArrayList();
            // Parsing code
            // ...
            alParsedValues.Add(Value);
            bSuccessfulParse = true;
        }
    }
}

```

```

        return alParsedValues;
    }
    // This function is overridden by subclasses to provide validation of
    type safety in a fail-closed manner of parsed values this element is being set
    to
    // It will use the index to determine the value to validate in the
    alParsedValues arraylist
    // It will also use the private value type map (alValueTypeMap) to
    determine the data type of value for validation
    protected virtual bool ValidateTypeSafety(int nParsedValueIdx)
    {
        return false;
    }
    // This function is overridden by subclasses to provide business level
    validation in a fail-closed manner of parsed values this element is being set
    to
    // It will use the index to determine the value to validate in the
    alParsedValues arraylist
    // It will also use the private value type map (alValueTypeMap) to
    determine the business type of value for validation
    protected virtual bool ValidateBusinessPropertySafety(int
nParsedValueIdx)
    {
        return false;
    }

    public object Value
    {
        get
        {
            // This may not be implemented by the developer here since
            normalization of parameters may cause multiple DataUseElement values to be set
            // or by default this may return the first element value to
            provide shorthand for single value element access
            throw new Exception("DataUseElements must be retrieved through
the ScenarioUse Property bag that holds them to ensure validation of
collective use before they are utilized");
        }
        set
        {
            bool bSuccessfulParse = false;
            ArrayList alParsedValues = null;
            alParsedValues = TranslateSetParamToProperty(ref
bSuccessfulParse, value);
            if(!bSuccessfulParse) {
                return;
            }
            for(int nValueIdx = 0; nValueIdx < alParsedValues.Count;
nValueIdx++)
            {
                ElementValueDef cThisValueDef = alValueTypeMap[nValueIdx];
                object oParsedValue = alParsedValues[nValueIdx];

```

```

        if (ValidateTypeSafety(nValueIdx) &&
ValidateBusinessPropertySafety(nValueIdx))
        {
            dictSafeValues.Add(cThisValueDef.ValueName,
oParsedValue);
        }
        else
        {
            dictSafeValues.Clear();
            throw new Exception("Invalid value given for
DataUseElement. Please check the value and try again.");
        }
    }
}

}
/// <summary>
/// This property provides the Scenario's Parameter name for use by
the property bag to allow
/// parameter name based access to the verified values. This can be
demonstrated looking at the
/// ScenarioPropertyBag object and how it builds its dictionary after
verification
/// </summary>
public string ScenarioParamName
{
    get
    {
        return sScenarioParamName;
    }
}
/// <summary>
/// This property allows unaware users this DataUseElement to retrieve
the map providing
/// the number and types of values this element holds so that they can
be properly interpreted
/// By default, we have made it and the ElementValueDef "internal"
because it is not absolutely
/// necessary that it be shared, but both the ElementValueDef and this
property could be made
/// public if the ValueTypeMap is desired to be shared.
/// </summary>
internal List<ElementValueDef> ValueTypeMap
{
    get
    {
        return alValueTypeMap;
    }
    set
    {
        // No set is offered here since the type map is initialized by
the subclass
        // and cannot be changed by its users
    }
}

```

```

    }
}
/// <summary>
/// This function returns the value(s) that have been set and
validated by the DataUseElement object
/// and are ready for collective validation by the ScenarioUse object
before retrieval
/// by the scenario function
/// </summary>
/// <returns></returns>
internal Dictionary<string, object> GetSafeValues()
{
    return dictSafeValues;
}
}

```

```

internal class ElementValueDef
{
    private string sValueName = null;
    private Type dtDataType;
    private string sBusinessTypeName;
    /// <summary>
    /// Initializes a new instance of the ElementValueDef class.
    /// </summary>
    /// <param name="sValueName">The name of the DataUseElement value this
represents</param>
    /// <param name="dtDataType"> The raw datatype this value should
represent.
    /// This is used for type-safety validation
and parsing</param>
    /// <param name="sBusinessTypeName"> The business type name, such as
Credit Card, this value should be.
    /// This is used for business
property validation</param>
    public ElementValueDef(string sValueName, Type dtDataType, string
sBusinessTypeName)
    {
        this.sValueName = sValueName;
        this.dtDataType = dtDataType;
        this.sBusinessTypeName = sBusinessTypeName;
    }

    public string ValueName
    {
        get { return sValueName; }
    }

    public Type ValueDataType
    {
        get { return dtDataType; }
    }
}

```

```

    public string ValueBusinessTypeName
    {
        get { return sBusinessTypeName; }
    }
}

```

Business data types that are used as parameters to our capabilities, such as Bank Account ID, Credit Card Number, Address, etc., would be created as inheriting classes from the above DataUseElement object. The framework provided by this object allows a shared library to be created to hold both this object and the sub classes to be used by one or more of our applications. This enhances secure application development from a process standpoint alone by forcing all our applications to use this type for its exposed capability function parameters and centralizing the development of these parameters to a common library. I have seen in many assessments redundant validation code copied or redeveloped and maintained in several locations, increasing the risk of vulnerabilities throughout the application over the course of time and application extension. By explicitly requiring the override of the above virtual functions, we enforce the forms of validation we need to ensure correct use of our parameters. We also encapsulate much of the dangerous logic, such as parameter parsing and interpretation, commonly exploited by attackers when placed in the same scope as authoritative logic.

The following things should be noted about the above implementation of a DataUseElement:

- The separation between the overall Set and Get of the elements value. The public "get" functionality of a DataUseElement Value throws an exception from direct accessibility in order to require that the element first pass through validation of its container object, the ScenarioUsePropertyBag, before becoming accessible to the capability functions that will use it. For this same reason, the DataUseElement exposes separate containers for raw values used in sets and the validated values that are only made available after both type and use validation. This enforces both forms of assurance before the values can be used, making Data Use Safety a requirement.
- Each DataUseElement value may be set with general input types of data, such as Address, Name, etc. It is assumed though that the TranslateSetParamToProperty will be called to parse/normalize such data to the values necessary for proper datatype and business property validation before being validation and retrieval from the ScenarioUse property bag. This allows the flexibility of the inputs we often see applications using, but separate complicated parsing code that's often exploited to a safe location.
- Although the DataUseElement object and its subclasses could be used directly as the parameters to our exposed capability functions, this would assume that the callers would be able to construct one of these objects, which is sometimes not the case. Therefore, by accepting general inputs as was mentioned in the previous

statement, our capability functions may expose "public" versions of themselves that take general raw data parameters, but immediately use those parameters to set DataUseElement values that are then passed to our actual capability functions internally. As a rule in such a case, these exposed functions should have no logic associated with the actual capability other than the necessary code to create these elements and pass them to our capability function. This also enhances security by hiding the logic we use for element validation from the caller.

- Since the exposed input "Value" of a DataUseElement may correspond to multiple normalized safe values, the type map includes a ValueName property to allow the ScenarioUse property bag to appropriately pull such values out of the object by name for validation and use.
- While the use BusinessTypeName is encouraged for validation of each elements value, it may be null, to allow generic use values that have no individual business validation needs to be passed, but still utilized in collective business use validation.

To demonstrate a framework of for Data Use element in a modular fashion, the following code is presented:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace AppShared
{
    public abstract class ScenarioUsePropertyBag : DictionaryBase
    {
        private string sScenarioName = null;
        private Dictionary<string, DataUseElement> dictUnverifiedElements =
null;
        private Dictionary<string, DataUseElement> dictVerifiedElements =
null;
        private bool bUseTainted = false;

        public DataUseElement this[String key]
        {
            get { return ((DataUseElement)Dictionary[key]); }
            set {
                bUseTainted = true;
                Dictionary[key] = null;
                dictUnverifiedElements[key] = value;
            }
        }

        public ICollection Keys
        {
```

```

        get
        {
            return (Dictionary.Keys);
        }
    }

    public ICollection Values
    {
        get
        {
            if (bUseTainted)
            {
                return null;
            }
            return (Dictionary.Values);
        }
    }

    public void Add(DataUseElement value)
    {
        dictUnverifiedElements.Add(value.ScenarioParamName, value);

        Dictionary.Add(value.ScenarioParamName, null); // We add a null
placeholder to ourselves
        // so that our keys can be freely viewed before verification
        // and duplicate naming prevention, etc., works
        bUseTainted = true;
        return;
    }

    /// <summary>
    /// This function will be overridden with the proper use verification
code.
    /// Upon verification, the bUseTainted property is set to false and
the verified DataUseElements
    /// are made available in the dictionary
    /// </summary>
    /// <param name="alBusinessInformationObjects">
    /// This is an array of objects the verification process may require
    /// to verify the Scenario's use such as a database connection,
session object, account info, etc.</param>
    /// <returns></returns>
    public virtual bool VerifyScenarioUse(ArrayList
alBusinessInformationObjects)
    {
        // Verification Code
        // ...
        UseVerified();
        return true;
    }
    /// <summary>
    /// This private function just does the heavy lifting of what is
required to make the verified elements accessible
    /// </summary>
    private void UseVerified() {

```

Copyright © 2007-2012 Vault Ecommerce, Derek Mezack

<http://www.vaultecommerce.com>

All Rights reserved

```

        foreach (KeyValuePair<string, DataUseElement> kvpVerifiedPair in
dictUnverifiedElements)
        {
            DataUseElement dueVerified = kvpVerifiedPair.Value;

            if (this.Contains(dueVerified.ScenarioParamName))
            {
                this[dueVerified.ScenarioParamName] = dueVerified;
            }
            else
            {
                this.Add(dueVerified);
            }
        }
        bUseTainted = false;
    }

    public bool Contains(String key)
    {
        return (Dictionary.Contains(key));
    }

    public void Remove(String key)
    {
        dictUnverifiedElements.Remove(key);
        Dictionary.Remove(key);
        bUseTainted = true;
    }

    protected override void OnInsert(Object key, Object value)
    {
        bUseTainted = true;
    }

    protected override void OnRemove(Object key, Object value)
    {
        bUseTainted = true;
    }

    protected override void OnSet(Object key, Object oldValue, Object
newValue)
    {
        if (newValue.GetType() !=
Type.GetType("AppShared.DataUseElement"))
            throw new ArgumentException("value must be of type
DataUseElement.", "value");
        Dictionary[key] = null;
        dictUnverifiedElements[key.ToString()] = (DataUseElement)newValue;
        bUseTainted = true;
    }

    protected override void OnValidate(Object key, Object value)
    {

```

```

        // We can't do use validation here because we need to validate
        values all together and with proper business info

    }

    /// <summary>
    /// Initializes a new instance of the ScenarioUsePropertyBag class.
    /// </summary>
    /// <param name="ReferenceID"></param>
    public ScenarioUsePropertyBag()
    {

        SetScenarioName();
    }
    // On startup this class sets its Scenario name to make error
    messaging or exceptions useful
    protected virtual void SetScenarioName()
    {
        sScenarioName = "My App Scenario";
    }

    }
}

```

Code Reuse

Often, applications expose generic forms of “access”, such as “Access Customer Data/Table”, “Access Order Data”, etc with no regard to intention. This is usually in an effort to retain reusability in their coding efforts without regard to logic exposure and misuse. It is important to note that logic within a defined scenario may make use of other more general functions, allowing code reuse, as long as these functions are not exposed to the systems user or user programs.

In order to still make use of reusable functions and code, developers utilize interfaces, scope, and object oriented encapsulation so that each scenario can be exposed in a single function, but private logic can be efficiently used internally without breaking the model. For instance, a public scenario function would exist for “A client should be capable of making a deposit to their account“, but unexposed functions to do such things as make database connections, recalculate balances, etc.

Scenarios should not be confused with a general “function” in designed structure. A function in an application may perform any technical operation our application needs and is often designed with reusability in mind. For this reason, the parameters to functions are often made as general as possible as to promote their reusability. Scenario functions are developed with the exact opposite in mind.

Conclusion

The adoption of Scenario Security Modeling in application development will significantly increase the security posture of applications. Applications implementing this model reduce the exposure of resources/privileges they use in a meaningful fashion and provide a natural means of management, by allowing administrators to focus on managing the tasks personnel need to function, rather than the access they need to perform it.

In addition, such applications are more resilient to compromise, limiting the scope of any compromise to the limited functionality a scenario defines about its intended “use”. This limit in scope can be seen by contrasting the theft of a gun *vs.* the theft of a microwave. Both of these items have dangerous components, but a microwave is not typically seen as an immediate threat if stolen because its *use* is well encapsulated in its design.

The proposed framework for this model also provides many cost savings for development by centralizing security related code and pairing it with the intentions they are used, providing lowered QA costs. In addition, savings in operations management can be seen with growing significance with the growth of application functionality by minimizing user management to *task control* and allowing monitoring to be performed of such applications on a capability level, providing better visibility to a user’s intention, rather than generic logs of “access”.

Part 2: Operating System Contrast and Adoption

Many of the documents concerning capabilities have strong focus on operating systems in their application. Although the focus of this document has been in web application, it is important to understand at least how the Scenario Security Model might be utilized at an operating system level in order to understand its difference to other existing security models.

Operating systems demand the ability to host applications offering a wide variety of capabilities that may not be preordained by the operating system developers. As mentioned before, this is one reason that it is difficult to successfully implement capabilities as defined in this document when developing an operating system. Without specific knowledge of the tasks a user will perform with any applications developed, capabilities would be limited in scope to controlling arbitrary resources that rarely relate to intention without further knowledge. In fact, arbitrary resources controlled by Operating Systems are rarely understood by the typical user or how they relate to the capabilities applications offer. It is also for this reason that ACLs are rarely configured by typical users in widely adopted Operating Systems. Operating Systems have made some progress in combating this issue by providing better default levels of access and special user accounts for services to reduce management and configuration risks.

Currently, operating systems such as Windows have added warnings that frequently pop-up for user-decisions relating to the use of such resources as “network access”. Without understanding the intention of the application, users are often left with the general question at the moment of the warning as to whether or not they want to program as a whole to “work”. In addition, such questions allow no means of the user to qualify the use of such resources with any limitations. For instance, popular applications that sometimes have no need to network access, still request network access for such things as automatic updates that are not core to its functionality. With automatic updates

being such a popular feature, users are often left to believe that practically all applications should be allowed to use the network.

Intelligent Decisions

One method the Scenario Security Model can be utilized is to provide a means and requirement for applications to register their scenarios. In presentation, this registration would not necessarily require the registration of the inputs of each scenario, only their capability. Note that the following refers to these registered capabilities, being described scenarios without mention of their inputs, not to be confused with capabilities in the capability security model. Registration would provide a listing of the resources each capability requires and a description of its intention, target of use, and recommended limitation that can be enforced by the Operating System. Registration of this nature would provide a means for the Operating System to present the user a meaningful interface for them to make access decisions.

Presentation of registered capabilities could be utilized in decision making a number of ways or combinations. One possible way would be during installation, allowing them to make such decisions before the application is even installed. Registered capabilities could also be presented during runtime upon startup or resource use. Obviously, both of these options are suitable primarily for desktop applications. Finally, registered capabilities may centrally controlled by application policies.

Intelligent Control

The use of application policies for centralized control of scenarios and resource limitations is similar to application firewall technologies currently in adoption. However, the difference between such technologies and this feature has significant value. Application firewalls often assume little knowledge about the applications they govern. For this reason, resources are often generally controlled and governed by policy that requires significant effort by security managers in order to have any granularity in control. For instance, application using the Internet for any reason, such as updates, force security policy makers to generally “allow” such access as soon as the application displays any issues. Often times, client applications are simply allowed network access after enough times of seeing them break in operations due to a more stringent policy. The proposed adoption of Scenario Security Modeling would instead allow application providers that are far more fluent with the reasons behind their resource use and capabilities to give policy makers a standard means of attaining decision making knowledge through registration, along with the ability to make recommendations for the limits that should be placed on resource access.

For instance, an application requiring network access for automatic updates could provide a suggested limitation of network access to their update server on the Internet at the time of registered scenario presentation, which could be enforced by most Operating Systems. This would also provide policy makers with a standard information source concerning the safety and risks associated with each application they may be deciding for corporate adoption.

Overall Benefits

There are many benefits to applications registering their capability functions with the operating system:

- The security posture of applications would be significantly increased with the adoption of the Scenario Security Model.
 - Applications utilized in a company environment would have a reduced risk of misuse and exposure by the encapsulation Scenario Security Modeling provides.
 - The scope of any compromise would be significantly reduced by accepted limitations recommended by software providers during capability registration.
 - Granular and more accurate access control would be more frequently adopted with greater ease by recommendations provided by applications in capability registration
 - Fewer outages or application failures related to stringent ACL policies would occur by allowing security personnel to take advantage of scenario descriptions and resource limitations provided by applications concerning resource use, leading to fewer errors and less testing to secure applications while allowing them to operate.
- Users or Installers of the application would be presented an applications list of capabilities and their requested privilege/resource use with the opportunity for the application developers to describe the intentions of their privilege and resource use. This would lead to more educated decisions by users or policy makers concerning application privileges.
- Operating Systems would be able to protect resources and privileges by limiting the scope of privilege use to registered flows of logic that use them. In addition, capabilities could even be “accepted with limitations” during installation, allowing an installer to add restrictions, such as Source/Destination IP restrictions or other possible limits to how requested resources are acceptably used.
- Application providers, whom are most fluent with the privileges and resource limitations their application can operate with would have a common means of making such recommendations and ensuring secure implementation.
- General utility applications could be freely used at a reduced risk of misuse by allowing users to place limits on their resource consumption, such as network access, at the time of installation.
- Application providers would have a means of demonstrating security concern and diligence for their users by providing scenario design for registration, limiting privilege use, and recommendations during registration for safe use of their application. While some providers may not desire to participate in this change, the competitive desire of many enterprise applications to prove their safety and compliance would drive its adoption.
- Operating System providers, who are unaware of the uses and minimal privilege requirements of installed applications would have a means of passing such intelligence and benefits to users and free them of the inadequate inconveniences currently imposed by general warning pop-ups that plague user operation.
- Costs associated with enterprise security management, specifically relating to application and network security policy development and management, would be significantly reduced.

- By applications providing recommendations regarding resource allowance and limitations, security personnel would not have to perform hours of research and ACL configuration in order to provide adequate security posture.
- Operating systems could potentially provide export capabilities of scenario policies for import into other protective infrastructure, saving configuration and management costs. For instance, IT could setup a machine with all the applications required for use in a company, accepting recommended or user defined resource limitations on registered scenarios. They could then export the complete listing of network related resource limitations as a whole for firewall import and enforcement.

Scenario functions exist, as shown in the example architecture herein, at the top-level of the application flow, where business tasks and intent are best understood. For this reason, the task of such registration would not carry a major cost for developers, as most applications offer a small manageable list of scenarios. Scenarios, including their description and privilege/resource use could then be accepted during installation. This puts the decision-making back in the installer's hands, whom is more likely to be suited for such concerns. Distributed applications would require more thought in how such scenarios would be accepted and managed.

Conclusion

Adoption of the scenario-based security model would allow operating systems to provide enhanced resource security, while also simplifying access management. By allowing developers a standard means of communicating their application resource needs, resource control and limitation can be accomplished more intelligently with less effort to achieve a stronger security posture. Costs associated with security management in access control are significantly reduced, saving configuration time and research related to machine "hardening" and policy development. With recent changes to popular operating systems, including additional pop-ups warning and decisions being presented to users during operation, the proposed recommendations would eliminate such annoyances, and provide a means of better decision making by access decision makers.

Part 3: Contrasting Other Security Models

The ACL Model

The ACL model provides for a descriptor to accompany each object, specifying the entities that can access it and the types of access allowed. This model differs from the scenario security model by allowing direct forms of access to objects and having no required privilege set and intentional logic to limit access when used. This model also differs by requiring each objects access to be managed individually by a set of ACLs, while the scenario-based security model allows for multiple objects to access with least privilege use within a specific logic that encapsulates intention.

The popular ACL security model for application development and security management does not prevent many forms of logistical exploitation. Currently, 90% or more of applications implement some form of access control list model or depend solely on components implementing ACLs themselves. The ACLs for applications are often configured to loosely control or limit data access. Even when configured with the best granularity offered by ACLs, generic forms of “access” to not adequately allow intention or target to be encapsulated and often lead to the plethora of resources requiring access control management for each task the application offers to exhaust management efforts and be ignored.

Most tasks performed by application users involve reference to multiple resources. For instance, the creation of a simple order in an order-entry system typically requires reference to a customer, a product, a method of payment, and a denomination. Applications that make great efforts in allowing granular configuration of individual resource access are often left uncontrolled due to the effort required to determine the resources and levels of access required to perform each scenario.

Resources are also mismanaged because business personnel think about an application for the scenarios the users will engage in, not the resources each task will reference. When faced with short deadlines to configure a user to be able to do their job, many administrators will superfluously grant privileges, not knowing which precise *access* privileges are required to allow their tasks to be performed. This is like giving the keys to their car away without being able to specify where the driver will take it for what reason. Resources used by application to accomplish tasks are often not understood by the administrators responsible for user management. This is especially true when considering an installed application that wasn't written by those who will administrate it. Even with granular access control being offered, the resources used to perform tasks in the application are not always obvious. Consider an application with hundreds of files or tables that may be used for each scenario a user engages in. Most of such applications are left with little configuration of access control in light of the effort required to know the resources and types of access each scenario requires. This issue relates to nature of relationships between users and resources as implemented in most ACL modeled systems [1].

In the same regard, most security professionals monitor security data coming from technologies that have little knowledge of their assets or the tasks users perform with their applications. For this reason, the data most often monitored from such systems as IDS or Firewall measure generic traffic, system vulnerabilities, and popular threats without any regard to the assets or intentions of individuals accessing organization services.

This negligence can be primarily attributed to the fundamental limitation of the ACL model in controlling resources with general access. This method of control requires knowledge transfer to occur between developers that write applications that use our resources and administrators responsible for protecting and limiting the use of these resources. Aside from the lack of understanding of alternative models developers and security architects often times depend on and/or choose example “frameworks” to follow

in their own designs, copy, or purchase. Exemplary systems using anything other than the ACL model are not popular and rarely known by developers or security administrators alike.

The Capability Security Model

One security model known to fewer individuals in the computer world is the capability system security model. The Capability was introduced by Dennis and Van Horn in 1966, in a paper entitled, *Programming Semantics for Multiprogrammed Computations*. Capability-based security is a concept in the design of secure computing systems. A capability, as defined in this model, is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights.[3]

A capability is defined to be a protected object reference which, by virtue of its possession by a user process, grants that process the capability (hence the name) to interact with an object in certain ways.[3] Most examples of this system demonstrate an operating system, where capabilities are representing by data-structures hosting an object, such as a file, and a set of privileges for the file, to be assigned to a user or program.

This system has many benefits, including the potential shedding of the ACL Model's method of resource control in a perfect world. It depends however on the correct management of capability assignment, requiring individual programs and users to be assigned only the capabilities they specifically need. Based on the definition of the term "capability", each capability has a reference to *one* object and a set of privileges for that object. Consider how many resources are on a popular operating system which already requires management. Now consider the creation of capabilities, each one having a reference to one of these resources and a set of privileges allowing a given program that uses it to work. The work required to research and management of these capabilities is one reason this system has often not been adopted by developers.

While capability-based security has advantages, its issues in practice can often be attributed to its lack of body in a defined *capability*. A capability refers to a value that references an object along with an associated set of access rights. An "object" as singularly referenced by each capability is often insufficient in practice to determine the intention or task of its user, even when paired with a set of privileges. Without accompanying logic and potentially required input from its environment, a resource may still be misused in the access provided by the capability security system.

Both Models

Both of the above security models only allow access to be controlled to objects individually. Often, systems today reference many resources at a time to perform a single task. By allowing "access" to resources individually, misuse can often occur by accessing such resources in malicious ways.

In addition, any given object may be used in multiple scenarios with different intentions. While the capability security model does reduce the risks associated with change of role risks by defining capabilities that reference both the object and set of privileges needed at a given time, both systems do not account for how the order in which such privileges are

used may sometimes allow exploitation. For instance, an order requiring customer data access, order entry, and credit card transaction requires access to all of these objects. By relying on individual object access control, one may be able to perform an order entry without credit card transaction.

Both of these prescribe that access be controlled without consideration of the inputs that may pertain to the scenario they are used in. The provided inputs to a given scenario can play a role in defining the intention of the user or user program. In addition, inputs often influence the correct least privilege use of objects it references. For instance, a scenario defined to allow access to customer's information, which may be stored in different locations depending on the company the customer belongs, would best be served by utilizing the customer's id or company as validated input in order to determine the data privileges required to perform the scenario in use. While such input is not forbidden by either systems approach to security, the definition provided by scenario-based security systems for input to scenarios affords greater immunity to injection based attacks.

Conclusion

We can see from the above recommendations that Scenario Security Modeling provides a strong differentiation from other models, including the ACL Model and Capability Security Model, by addressing the intention objects are limited to in access.

References

1. Mark S. Miller, Ka-Ping Yee, Jonathan Shapiro - Capability Myths Demolished.
Online at: <http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf>
2. N Hardy - The Confused Deputy
Online at: <http://www.pdos.lcs.mit.edu/6.828/2004/readings/hardy88confused.pdf>
3. Wikipedia – Capability-based Security
Online at: http://en.wikipedia.org/wiki/Capability-based_security
4. Wikipedia – Secure Computing
Online at: http://en.wikipedia.org/wiki/Secure_computing