Singularity Design Note 35

Security API

In which we describe how to implement access control on protected resources.

1. Introduction

This design note describes the API of the Singularity subsystem.

Singularity SIPs receive a security principal name at creation, and this name is immutable. A principal name is an ordered list of applications, each of which can act in specific roles. The list ordering is intended to reflect the chain of application invocations that led to the creation of the named SIP. Human users are represented as roles of programs trusted to perform user authentication. The design for our *compound principal* grammar has been previously discussed in SDN9.

Because security principals names have structure, we cannot use simple integers or groups of names to specify which principals can access which resources. Instead, Singularity access control lists are regular expressions. Common subexpressions (the equivalent of groups in conventional access control systems) can be named in and expanded from the Singularity directory service. The structure of compound principals makes it appealing for intermediate nodes in a security-relevant operation to simply record their participation and let the end reference monitor decide. Because of this, impersonation is largely unnecessary and we do not support it in Singularity.

For the purposes of this document, Singularity access control is discretionary. Programs that control resources do so by means of explicit access control checks. Since all communication takes place over controlled channels, the subject of such access control checks can be determined by examining the message source principal associated with the incoming channel. Access control decisions are implemented by means of a security library that is bound into every SIP that guards resources. The Singularity kernel provides minimal support by way of a service that maps the shorthand principal identifiers used by the channel implementation into full names.

One resource of particular interest is the directory service. This service implements a naming tree whose names are used by applications to establish communications channels with system components such as files, services, and devices. The directory service, then, employs access checks to allow system policy to regulate which principals can register which names, and, for any given name, which principals can establish channels to it. Thus, while Singularity SIPs could in theory authenticate their channel partners for each new channel, in practice it is sufficient to assume that the directory service is trusted to enforce system channel establishment policy.

In the current Singularity implementation, any one SIP can speak for only one security principal, and this principal cannot be changed. This model is appealing in that it leaves little latitude for *confused deputy* *attacks* or other attacks that depend on security-relevant code dealing (incorrectly) with the authority of multiple different security principals. However, even with the low overhead on SIP invocation that Singularity provides, it may be over-optimistic to assume that all code dealing with multiple principals can be avoided. We expect to introduce controlled delegation of authority between SIP in the form of specially designated communications channels. This is the subject of ongoing research.

2. Principal identifiers and names

The *security service* component of the Singularity kernel is responsible for managing principal identifiers and names. In the Singularity ABI, a principal is represented by a `ulong` value. This `ulong` maps to a principal name managed by the kernel. The following datatype is used to represent a principal within applications. All APIs discussed in this document are defined in the `Microsoft.Singularity.Security` namespace:

```
public struct Principal
{
    public ulong Val { get; }
    public bool Equal(Principal p);
    public string! GetName();

    public static Principal Self();
}
```

The `Equal` operation simple compares the `ulong` values in two different `Principal` structs. Hence, a negative result does not guarantee that the two principals being compared do not have the same name or authority. The Self method returns a `Principal` struct for the current process.

The functionality described so far is part of the standard application runtime. All methods here are thread-safe.

3. Controlling Access to Resources

This section describes how to implement a simple reference monitor. The programmer must define at least the following:

- **Permissions.** These parameterize access control decisions. In order to be granted access, the caller must be granted sufficient permissions to perform a controlled operation.
- **Access control checks.** The programmer must insert access control checks into the appropriate code locations to test whether access is granted.
- **Access control lists.** Any object or resource to be protected must define the set of principals that are granted access to specific objects and permissions.

3.1. Permissions

Any application that protects resources must define the set of permissions specific to the resources to be controlled and the operations on those resources. The `AccessMode` datatype is available to make these static declarations:

```
public class AccessMode
{
    public AccessMode(string! mode);
    public string! Val;
    public int Num;
}
```

As an example, let's look at the permission definitions used by the Singularity directory service:

```
    public class DirPermissions
    {
        public static readonly AccessMode AccessModeTraverse;
             // can pass through
        public static readonly AccessMode AccessModeRead;
             // can read/enumerate
        public static readonly AccessMode AccessModeWrite;
             // can create/write
        public static readonly AccessMode AccessModeSetAcl;
             // can modify access control list
        public static readonly AccessMode AccessModeRegister;
             // can register name in namespace
        public static readonly AccessMode AccessModeDeregister;
             // can deregister name from namespace
        public static readonly AccessMode AccessModeNotify;
             // can recieve notifications

        static DirPermissions() {
            AccessModeTraverse = new AccessMode("traverse");
            AccessModeRead = new AccessMode("read");
            AccessModeWrite = new AccessMode("write");
            AccessModeSetAcl = new AccessMode("setacl");
            AccessModeRegister = new AccessMode("register");
            AccessModeDeregister = new AccessMode("deregister");
            AccessModeNotify = new AccessMode("notify");
        }
    }
```

The text string in the `AccessMode` constructor is important, since the string defined there appears verbatim on access control lists. As the Singularity access control system evolves, it might become prudent to qualify the textual permission string for clarity or create `AccessMode` objects that are shared between applications.

3.2. AclCore class

The `AclCore` class is the primary interface to access control. The API is simple, the `CheckAccess` method checks a principal and requested permission against the supplied access control list and returns a `bool.` The implementation of `CheckAccess` takes the textual name of the supplied mode parameter and appends it as a role (e.g. `@read` for read permissions) to the principal name implied by the argument principal. This compound name is matched against the supplied access control list. Cached state is used to accelerate the evaluation, but is never used to produce a false negative. Because a cache is used to accelerate positive results, revocation is not instantaneous. Instead, revocation is by timeout.

```
    public class AclCore
    {
        public AclCore(string _coreName, IAclCoreSupport! _support);
        public AclCore(string _coreName, IAclCoreSupport! _support,
                       int aclCacheMaxEntries, int aclCacheExpirySeconds,
                       int groupCacheMaxEntries, int groupCacheExpirySeconds);

        public bool CheckAccess(Acl acl, AccessMode! mode, PrincipalId principal);

        public static Principal EndpointPeer(Endpoint*! in ExHeap ep);

        public bool Disable { set; };
        public void DumpStats(StringBuilder! sb);
        public void ClearStats();
        public void FlushCache();
    }
```

The implementation keeps two caches: one for cached regular expressions (derived during access control list evaluation) and a second for cached group expansions. The more detailed version of the constructor allows the caller to control the timeouts on these caches and the maximum number of entries in each. The current default timeouts are 15 minutes for the regular expression cache and 60 minutes for the group cache. The defaults for maximum entries are 200 for the regular expression (acl) cache and 100 for the group cache.

The `EndpointPeer` method is of note. Applications can use it to derive the identity of the other end of a communications channel. The resultant `Principal` can then be passed to `AclCore.CheckAccess`.

`Disable` allows the client to disable or re-enable the security logic behind. When disabled, `CheckAccess` always returns true. `DumpStats` and `ClearStats` support reset and display of statistics for an `AclCore` instance. `FlushCache` allows the caller to remove all cached data.

When access control checks are performed as a result of channel communications activity, it is important to note that in most cases the channel contract will not allow the partner endpoint to move during the course of the request. This allows the access check code to proceed knowing that the identity of the distant endpoint will not change during the course of the check. In order to get such a guarantee, the channel contract must be written to ensure that the sender is in a receive state during the access check (e.g. the sender must wait for a response before sending again.)

The `AclCore` class can be instantiated with a descriptive `_coreName`. This name is purely for informational purposes (for statistics output) and the name can be `null`. Each class instance must be instantiated with an object that implements the `IAclCoreSupport` interface to provide a caller-specific mechanism to map common subexpression names found in `Acl` expressions (discussed in the next section) to contents.

```
    public interface IAclCoreSupport
    {
        string GetGroupContents(string! path);
    }
```

The `GetGroupContents` method can return `null`. This indicates that an error has occurred.

Because a namespace provider might need to pass an access control check in the process of implementing `IAclCoreSupport`, there is a potential recursion during subexpression expansion. For this reason, namespace providers should be sure to allow read access to access control expressions whenever possible. To break the access control recursion in namespace providers that cannot do this, the access control library always allows access to the “self” principal. This library currently aborts the entire expansion if a failure occurs during a subexpression expansion (including an access control violation). It should be possible to do better. For instance, failures should be tolerated when they occur in disjunctions.

3.3. Access Control Lists

Access control lists are represented in Singularity through the following struct type:

```
    public struct Acl
    {
        public static Acl nullAcl = new Acl();
        public readonly string val;
        public Acl(string val);
        public Acl(byte[] utf8Encoding);
        public static byte[] ToUTF8(Acl acl);
        public static string ToString(Acl acl);
    }
```

In practice, though, an `Acl` is a regular expression that corresponds to the grammar described in SDN9:

```
      Atom = Arc | “/” | “@” | “+”
      Item = Atom | “.” | “(” ACE “)” | Item “*” | “{” GroupName “}”
      GroupName = “/” Arc | GroupName “/” Arc
      Seq = Item | Seq Item
      ACE = Seq | ACE “|” Seq
      Acl = ACE
```

The matching semantics of the grammar should be interpreted as follows:

- any Atom matches itself;
- “.” matches any single Arc (explicitly excluding “/”, “@”, “+”);
- “(ACE)” matches ACE;
- “Item *” matches zero or more sequential occurrences of Item (greedily);
- “{ GroupName }” matches whatever is matched by the ACE that is the contents of the node GroupName in the naming tree;
- “Seq Item” matches Seq followed immediately by Item;
- “ACE | Seq” matches either ACE or Seq.

As described in SDN9, the “+” delimiter is used to represent process invocation and the “@” delimiter defines a role. As described earlier, the process of evaluating an access check always appends to the principal name a role corresponding to the requested permission, and expressions in the access control grammar take this into account by explicitly representing the permissions granted to principals. It might be worthwhile to represent permissions in positions other that the end of prinicipal names, but we have not done so yet.

The access control language allows common subexpressions to be substituted from file objects in the Singularity directory service.

Here is a simple access control list that grants “any” permission to any logged-in user and the processes invoked by that user:

```
    /groups/path            .(/.)*
    /groups/userrole        login@users/.
    /groups/anyperm         @.
    /groups/app             {path}|{userrole}
    /groups/anyall          {app}(+{app})*{anyperm}
```

The resulting top-level access control list would be "`{/groups/anyall}`". Note that relative subexpression names are resolved using the most recent non-relative prefix in the recursion. In this case, the most recent non-relative prefix is "`/groups`".

If we wanted to define the same subexpression, but for read-only permissions, we would write:

```
    /groups/readperm        @read
    /groups/anyread         {app}(+{app})*{readperm}
```

In this case, the access control list would be "`{/groups/anyread}`".

4. Access Control Policy

It is often the case that access control policy must be enforced across large numbers of structurally-related objects, such as in a file system. Singularity provides an interface for specifying policy rules that apply across such collections.

```
    public interface IAclRule
    {
        bool Valid;
    }

    public interface IAclPolicy
    {
        Acl LookupAndExpand(string! resource,  out IAclRule rule);
        void AddRule(string! resource, string! aclExpander);
    }
```

The `IAclRule` and `IAclPolicy` interfaces define a class of such rule engines. Resources are named with strings and the `LookupAndExpand` method serves both to look up a rule in the current set of rules and also perform limited string manipulation to permit factoring of the name of the requested resource into the resultant access control list. A cacheable IAclRule object is returned so that the caller can reuse the result after checking that a previously fetched rule is still valid.

Naming and expansion semantics are implementation specific, however the security library currently provides an implementation suitable for hierarchical file systems.

```
    public class PathPolicyEngine : IAclPolicy;
    {
        public PathPolicyEngine();
    }
```

This policy engine consists of a collection of rules applicable to paths in a hierarchy. Given a resource path, the engine will generate an access control list. If no suitable rule is found, the engine will return `nullAcl,` which restricts all access. When multiple rules apply to the given resource path, the engine will choose the one that has the longest matching prefix.

Expanders are access control lists with placeholders. Each placeholder is substituted with an nth arc of the tested path. A placeholder is specified using "{index}", where index is the index of the arc to be used.

Here is one example.

rule:	/restricted/more
expander:	{/users/{2}}
resource:	/restricted/more/aydan/test

Matching "resource" against "rule" will produce the access control list: {/users/aydan}.

Another rule engine example comes from the in-memory directory service implementation. In this implementation, each `Node` object has an `Acl` that might come from either a rule lookup or a explicit `StoreAcl` operation. An explicitly-set `Acl` takes precedence. An explicit `Acl` is present when `rule == null`,. otherwise `acl` contains the results of evaluating the derived policy rule for the node. Here, `rule` must be checked for validity in case it represents superseded policy.

```
        private Acl acl = Acl.nullAcl;
        private IAclRule rule = null;

        public bool CheckAccess(AccessMode! permission, PrincipalId prId)
        {
            Acl acl = GetObjectAcl();
            return core.CheckAccess(acl, permission, prId);
        }
```

```
        private Acl GetObjectAcl()
        {
            lock (this) {
                if (acl.val != null) {
                    if (rule == null || rule.Valid)
                        return acl;
                }
                acl = policy.LookupAndExpand(nodeName, out rule);
                return acl;
            }
        }
```

5. PrincipalImpl

The security service implementation in the Singularity kernel exports a specialized class for use by other elements of the kernel:

```
    public class PrincipalImpl
    {
        public static void      Initialize();
        public static void      Export();

        public static Principal Self();
        public static Principal MakePrincipal(ulong id);
           // this is a kernel-only constructor for the type Principal

        public static Principal NewInvocation(Principal parent,
                                         Manifest! manifest, string role);
        public static string!   GetPrincipalName(Principal p);
        public static void      DisposePrincipal(Principal p);

        public static void      RegisterAclCore(Object core);
           // "core" is really an AclCore object
    }
```

This class provides functionality to the kernel process management implementation for creating new principals when processes are forked. The `NewInvocation` method takes a `Manifest` as argument as well as an optional `role`. Thus, Singularity manifests contain sufficient information to name applications. There are also

`Initialize` must be called once before any other security function for initialization of the kernel security service. `Export` should be called near the completion of kernel initialization to export diagnostic channels to clients. `RegisterAclCore` is called by the `AclCore` constructor to inform the security service implementation of new `AclCore` objects from which statistics might be available.

6. Diagnostic Interfaces

The class `SecurityDiagnostics` contains a collection of methods that are intended as diagnostic aids for getting and setting state within the kernel security service. This class is implemented as a client library that communicates with the security service via a channel. This class will not normally be available for production versions of the system.

```
    public class SecurityDiagnostics
    {
        public static void Disable(bool yes);      …
        public abstract string! GetStatistics();
        public abstract void ClearStatistics();
        public abstract void FlushCaches();
    }
```

The `Disable` method can be used to disable (or re-enabled) all security checking in the kernel. The other methods get kernel security statistics, clear them, and flush the access control caches.

7. Practical Considerations

Timings for calls to `AclCore.CheckAccess` vary according to the state of the caches. When all caches are primed, that is the access control list has been previously evaluated against the argument principal and permission, the elapsed time should be on the order of a microsecond. If the target regular expression is in the cache, but hasn't been evaluated against the principal and permission before, the times should be on the order of 10-20 microseconds. If the access control list is not in the cache, or if re-evaluation is being performed due to an access control failure, we don't have good measurements. It obviously depends on the state of the subexpression cache and the number of subexpressions, but timings in the 500-1500 microsecond can be expected.

At present, subexpressions are kept in the “/group” subtree of the in-memory directory service. The group definitions and in-memory directory service policy rules are defined in the “distro” XML configuration. The XML configuration is parsed by the directory service initialization code, and the groups and policy rules are instantiated in that code. Any or all of these practices are likely to change over time.

Access control expressions are currently limited to alphanumerics. It should be straightforward to eliminate this restriction in future.