Singularity Design Note

9

Security Model

1. Introduction

This document proposes a security model for Singularity. In keeping with the mandate of the Singularity Security Group's charter, this design has the following goals:

- To meet the security needs of a general operating system—but with a particular focus on Singularity's proposed application space (the "in-the-closet" home server), as discussed in [SDN8]. This design places a premium on simple, effective mechanisms for administration and for security.
- To be as simple and as transparent as possible, in keeping with the "Singularity razor" approach to design.
- To facilitate the formulation and expression of clear, understandable security policies by developers, administrators, and users.

In brief, this document describes how *isolation* is accomplished in Singularity through the use of processes, type-safety, and channels. Section 3 discusses how processes can be grouped into *applications* that exercise rights derived from application identity. Section 4 proposes the composition of *security principal* while Sections 5 and 6 discuss how *access rights* to resources can be determined in relation to an overall system policy. Section 7 suggests some possible directions for future work. Section 8 concludes with a list of implementation priorities for Singularity v1.

Singularity is not designed to be a distributed operating system, but it is intended to support the construction of distributed systems via federation. This document describes an initial design for a very early version of Singularity. Much of the design has been made intentionally simplistic in order to allow for forward progress. The intent is to lay an appropriate groundwork for further research, not to solve all the problems at once. For example, many of the difficult distributed systems issues will not be tackled in the initial implementation.

2. Overview

Singularity is a small, modular operating system whose basic tenets support the construction of a secure system. The relevant basic properties of Singularity are as follows.

- Singularity is programmed using a language that restricts access to information and state held in objects and classes. Static and dynamic analyses should be possible to ensure that accesses to information and to state are properly restricted.
- The unit of code isolation is the process. The language does not allow any direct access to information or state outside the process. In particular, sharing of state is disallowed.This isolation enhances integrity, secrecy, and perhaps availability.
- Processes execute with a fixed set of code components, and new code components are not added to processes during their execution.
- Singularity rigorously restricts the interactions between processes (but much less rigorously than the interactions that occur within a single process).
- System extension is achieved by creating and communicating with new, isolated processes, not by adding code to existing processes.
- All application code is declared by precise manifests [see SDN12]. All manifests are in turn declared within a system configuration [see SDN13a]. The system configuration determines where, when, and how application code is run.
- Processes communicate exclusively via channels.
- Channels have two bidirectional endpoints whose movement is constrained. The passage of data through channels is also carefully constrained by type declarations and channel contracts.
- Likewise, kernel services are provided to processes via channels between the threads in a process and the kernel.

This set of characteristics leads naturally to a security design with the following features:

- The smallest granularity of security policy enforcement is the process.
- Security policies can be enforced by regulating the establishment, endpoint transport, and use of channels between processes.
- Policies can be based on the (well-defined concept of the) code running in a process, as well as other attributes such as the user that initiated the process, or the machine on which it runs.
- Policies can also be based on the well-defined concept of application identity.

The Singularity security design emphasizes system integrity and simplicity of management. The security component of Singularity must be able to operate autonomously, with little or no direct administrative management. It must be straightforward to construct and install applications that are highly-resistant to viruses, worms, and other malware. It should be straightforward to run less-trusted applications in restricted environments.

In Singularity, application identity is fundamental to the composition of security principals. Applications, therefore, can be granted rights to access resources and these rights are controlled through conventional access control mechanisms. The rights associated with Singularity application instances will depend on several factors, at least including machine identity, logged-in user identity, code identity, and application identity. Singularity facilitates the categorization

of applications and attribution of rights according to various forms of code identity, including code provenance. In doing so, Singularity integrates elements of application invocation history (similar to the notion of execution history found in the Java and CLR security models) with OS-supported identities and ACL primitives found in traditional operating systems.

Singularity applications are declared through persistent manifests. Manifests include not only the code segments that embody each application, but also describe the application publisher and any certificates pertaining to the publisher and the application. In addition, the system configuration provides essential parameters that describe how applications should be instantiated in the running system.

Application instances arise through the invocation of application manifests, relative to the system configuration. Application instances are composed of one or more processes. In order to simplify the Singularity programming model, each process executes with the authority of exactly one security principal. The decision to use processes as the finest grain for security decisions is not intended to preclude subsequent research into finer-grain techniques such as data flow analysis. Indeed, the design proposed here has been left intentionally minimal in the expectation that it will be expanded through future research.

Although much of the initial design will be focused on a single-machine system, Singularity will be commonly used as part of a distributed system. Therefore, the proposed security abstractions will be designed with "the network" in mind. Singularity is not a distributed operating system, but it is intended to support the construction of distributed systems via federation. Most of the difficult distributed systems issues will not be tackled in the initial implementation.

There will be a trust hierarchy with a single root for each Singularity machine. This hierarchy is governed by system policy specified in the system configuration [SDN13b]. It names and encodes information such as known users, groups, trust relationships with networked entities, trusted publishers, and installed applications. Local configurations can be extended with policy rules downloaded from trusted network machines or domains. Although *administrators* might manage such policies, the reader should not necessarily equate users and administrators. Singularity will support an environment where useful security policy can be administered remotely through paid or contractual services.

Systems such as NGSCB have emerged that offer hardware support for sealed data storage and attestation. The Singularity design does not require such features, but permits taking advantage of them in the future. They may be useful for administering a distributed environment. There will not be a Digital Rights Management subsystem in the initial design. If DRM is to be necessary in the context of Singularity, there may be many ticklish issues of standards compatibility to be worked out. Innovation in this area might therefore be difficult.

This document deals only with native Singularity applications. It does not propose any model for non-native applications such as those supported by the Singularity virtual machine monitor.

Singularity will not offer any generic mechanism for preventing or mitigating denial-of-service attacks.

3. Processes and Application Instances

In Singularity, the process is the fundamental unit of execution and isolation. Isolation between processes is accomplished through the language type system, so that information or state in one

process is normally unavailable to others. Singularity assumes the correctness of the native language type-safety abstractions, and in particular that native Singularity programs can reference memory as only allowed by the language semantics. All code will execute in hardware "ring 0" without remapping virtual addresses. MSIL (or a variant of it) will be the standard format for executable code.

Each process can call a small amount of privileged code, a *privileged runtime*, that is logically part of the operating system. The privileged runtime provides core kernel functionality such as garbage collection, IPC, and scheduler primitives. This runtime component, which may need to execute unsafe code or access raw hardware devices, is accessible only through a narrow interface and is carefully coded so as to avoid inappropriate application access to privileged functionality without the use of hardware protection mechanisms. Singularity relies on security properties of MSIL for reference safety and scoping. Depending on the implementation details, it might also rely on MSIL for protection of privileged object fields and methods. As the security abstractions for MSIL don't completely correspond to those for C#, getting the privileged runtime right will require considerable care.

The kernel loads the specified executable code into a process, after which the code is immutable. Processes communicate with each other and with the OS kernel (except as noted above) through typed channels. A process can request the creation of a new pair of bidirectional channel endpoints. One or both of these endpoints can be passed across existing channels to other processes. A mechanism for *fixing* endpoints to a specific process so as enable authentication of channels is discussed in Section 4.5.

The execution environment on a Singularity machine can be described as a set of running processes (with the kernel as the initial process). As described earlier, application instances arise through the invocation of application manifests. One or more processes can arise from each such invocation. For each process, the OS maintains the following: a reference to the manifest from which the process was created; a unique application instance identifier that binds all the processes stemming from a specific application invocation; and a security principal that characterizes the authority that the process can exercise. Processes can have only a single, immutable principal identity. The intent is to avoid programming errors that result from managing multiple identities. There is some danger that the performance ramifications or other effects of this decision will outweigh the benefits.

When a process is started, it is handed a channel that it can use to obtain core OS services such as naming and communication with other processes. The fact that all communication to and from a process springs from an initial channel gives the opportunity to interpose on naming and channel creation, and therefore control the behavior of a child process. Possible future work in this area is discussed in Section 7.1.

When an application is invoked, there is no default transfer of authority from the invoker to the invokee. In this case, the invoked application will execute with only its inherent authority derived from the application identity. This enables the implementation of traditional protected subsystems that own all the resources they manage. An administrative option in the application manifest will indicate whether the target application receives rights from the invoker. As described in Section 4, the invoked process will acquire a compound principal identity that identifies both invoker and invokee.

Although the identity of an application can determine its rights, the same application installed on two different machines can derive two different sets of rights. Similarly, an application's rights on one machine can change over time. Such differences and changes are due to administrative actions.

Application invocation is a fundamental mechanism for obtaining channels to local services that these application provide. Depending on the administratively setparameters defined in the target manifest, it might be acceptable for the invocation of an application manifest to bind to an existing application instance already derived from the same manifest. This can be the case when the target manifest describes a protected subsystem that does not inherit authority and a process instance for that manifest already exists. However, it might instead be necessary to start a new application instance, for example when authority must be inherited. Based on the target manifest and the state of the system, Singularity will make this decision at invocation time. Note it might be desirable to invoke a new (dedicated) process instance even if inheritance of authority is not required. Per-invoker instances promote isolation by eliminating the need for shared state.

It might be practical for an application publisher to package multiple different applications within the same manifest. This might be useful, for example, to describe closely related programs that operate in the same general execution environment. Thus, application manifests will support multiple invocation points (e.g. verbs) to enable this functionality.

In order to run in Singularity, an application manifest must first be installed into the system configuration. The policies imposed on applications at installation time and at execution time are described in Section 6. For example, the start-up environment for each application is carefully specified in the system configuration so as to minimize the possibility of undefined or unspecified behavior, such as when an application instance might attempt to gain improper access by starting a privileged application in an unexpected environment. Thus, application extensions that require different privileges are typically implemented as separate applications.

4. Principals

Processes receive a principal identity at creation, and this identity is immutable. Singularity principals are compound entities that take the following form.

Principal ::=	AppRole | Principal '+' AppRole
AppRole ::=	App | App '.' Role
App ::=	Path
Role ::=	Entity | Role '.' Entity
Path ::=	Entity | Path '.' Entity
Entity ::=	STRING

Principals are ordered lists of applications, where applications can act in specific roles. As discussed below, human users are modeled as application roles. The list ordering is intended to reflect an immutable log of application invocations.

Compound principals arise during application invocation. If an invoked program inherits authority from the invoker, that is reflected in the resultant compound. So, for example if Apps.Microsoft.IE invokes Apps.Adobe.AcroRd the resultant principal might be Apps.Microsoft.IE + Apps.Adobe.AcroRd. Note, however, that as previously described, application invocation does not necessarily involve inheritance of authority. This might be modeled as Apps.Microsoft.IE.Nothing + Apps.Adobe.AcroRd, or simply as Apps.Adobe.AcroRd, each of which has its advantages.

4.1. Trust Hierarchy

Each Singularity machine will have a trust hierarchy that is rooted in its system configuration, and that is therefore subject to the policies defined there. Nodes in the trust hierarchy name security principals or groups. (Groups are discussed in Section 5.) Principal entities are represented by arcs in the hierarchy; each such name is valid within the namespace of its parent node in the tree. At the root of the trust hierarchy is the local configuration, which is the authority of the local running system. A Path in the grammar above enables the holder to trace back through local names for all the intermediate trusted entities to the local system root.

Nodes in the trust hierarchy can reference supporting information in the system configuration. For example, a node naming a user role might use a name in the system configuration to reference configuration information for authentication of that user. A group node might reference that group's name in the configuration, and from that name its membership information. An application node might reference the corresponding application manifest, as represented in the system configuration.

Intermediate nodes in the trust hierarchy can represent delegation of trust. For example, the system root might trust the publisher Microsoft to specify applications that it publishes. Thus, Microsoft could specify the meaning of the name Microsoft.IE, although the rights given to that application could still be determined locally. Or, the system might trust a domain authentication service (e.g. com.microsoft.corp.northamerica) to authenticate user names within that domain, with the meaning of such authentication defined elsewhere.. There might be symbolic links in the hierarchy to give an application its own local namespace, and to enable local renaming of principals and groups in this namespace. Again, the shape of the trust hierarchy is always subject to the policies (e.g. rules and certificates) specified in the system configuration, because the system configuration ties trust declarations to their effects.

Imagine that every application has a subtree of the trust hierarchy that reflects context specific to that application. It would be extremely desirable for applications to make most security decisions based on this sort of application-specific context. This helps enforce the principle of least-privilege since each application's configuration can constrain the elements of the trust hierarchy that it must itself trust. In Singularity, the majority of configuration information is supplied off-line by application publishers. Since application publishers presumably understand their applications, it makes sense to allow publisher-supplied configurations to effectively express security policy, although such configurations must of course be localized within the context of the importing configuration.

The relationship between the system configuration and the trust hierarchy is not yet understood. The trust hierarchy might best be embedded within the system configuration, but the details have not yet been worked out. For example, it is unclear at this time how configuration data that can change dynamically should be represented in the system configuration (such as group membership).

4.2. Roles

Roles represent a strict diminution of authority for applications. As expressed in the grammar, the authority of an application in a role can be further diminished through a sub-role, and so on. Roles of a single application do not commute. In other words, an application running without a role has access to resources granted to any possible role of that application. However, an

application in a specific role can only access resources granted under that role and any possible subroles. Roles exist only in the trust namespace of an application that might adopt them. Symbolic links in the trust namespace allow roles to be shared between applications.

Roles allow applications to pass on some subset of their authority to the applications they invoke. This can be a valuable tool for implementing protected subsystems that must manage resources accessed by mutually distrusting entities. Roles can be adopted only by creating new processes. This will maintain the invariant that process principals are static for the life of a process.

When considering the adoption of roles during process creation, there are two distinct cases. In the first, a process invokes code from the same application. In this case the transition of authority might be.

```
Apps.App1 -> Apps.App1.WeakRole
```

However, the situation is different when the target application is not the same application. Suppose App1 wants to start App2, with inheritance, but in a weaker role.

```
Apps.App1 -> Apps.App1.WeakRole + Apps.App2
```

This dichotomy seems intrinsic, and must be reflected in the process creation API.

4.3. User Authentication

Human users are expressed as roles of applications. So, for example, there might exist a variety of system login-shell programs that perform user authentication one such shell that authenticates via user passwords; another for remote authentication via RSA keys; and still another for biometric methods. In each case, the original application, which might run as System.PwdLogin will take a role upon successful user login so as to run as System.PwdLogin.ThisUser. Alternatively, the new process might run as System.PwdLogin.Users.ThisUser where users is a link to a system-wide set of user-roles.

Given that Singularity is targeted at home appliance applications, it might prove more suitable for certain end-user applications to perform authentication directly. This could be useful to avoid losing application context while switching users. Imagine a PVR application that wants to access files private to Ted. In order to do so, it might run as the principal Apps.Microsoft.PVR.Users.Ted.

4.4. Remote Authentication

In some cases, such as for compatibility with legacy (non-Singularity) systems, it will be necessary to authenticate communications channels (as opposed to Singularity channels) and associate them with principals in the local trust hierarchy. So, for example, a physically remote administrator of a local application must be able to establish an authenticated channel with credentials that map to a local identity with administrative privileges for the target application.

When a remote computation is authenticated over a network, the authenticating party is in practice evaluating an policy that takes into account the administration of the remote machine, the trust hierarchy linking local and remote machines, and perhaps the software that is running on the remote machine. The policy required for this authentication is embodied in the local trust hierarchy, that is to say that the local trust hierarchy only links in remote namespaces that are trusted under local policy, and the system configuration specifies the parameters for any required authentication protocols. This is one reason for retaining a log of selected security-related actions in compound principals, since the eventual meaning of these actions cannot always determined at the time and place that they occur.

Suppose a remote shell client seeks to authenticate channels from users holding pre-shared public keys (as in ssh). Such keys might be named in the trust hierarchy by a hash of the public key (e.g. Keys.ab456cefa). The remote shell namespace would then link to the key namespace and, after carrying out the appropriate authentication protocol, produce principals such as Apps.RShell.Keys.ab456cefa. However, using the key hash as a principal might be inconvenient. Instead, the actual pre-shared public key, stored in the system configuration, might be named so as to attribute the user holding it: Apps.RShell.Users.Fred.

Similar indirection tricks in the trust hierarchy can be used to link large hierarchical naming authorities into the local namespace.

Compound Singularity principals might ultimately contain remote host components to allow seamless remote invocations using native Singularity naming. This will not be considered for the initial implementation.

4.5. Principals and Channels

In Singularity, processes send messages over channels. Singularity provides a mechanism that allows a process to obtain the principal corresponding to the owner of the distant endpoint for a given channel. The resulting principal identity can then be used to make access control decisions pertaining to messages received on that channel. However, the fact that a request was made by a principal over a channel does not give the receiver the right to act as the sender in subsequent communications. Authority can only be inherited by application invocation as described in Section 3.

Singularity channel endpoints can move from process to process. In order to determine that a distant endpoint is bound to a particular process, that endpoint must first be *fixed* by the owning process. Singularity therefore provides the primitive:

```
FixEndpoint(ep: Endpoint);
```

This primitive associates the argument endpoint with the current process, and guarantees that further attempts to move the endpoint will fail. The corresponding primitive for getting the principal associated with the far end of a channel is:

```
GetPrincipal(c: Channel) : Principal;
```

This primitive retrieves the principal associated with the distant endpoint of the channel. If the distant endpoint has not yet been fixed, an error is raised. Note that these primitives require the constraint that a processes has only one principal that is fixed for the process lifetime. If this were not the case, then it would be necessary to provide a principal when fixing endpoints.

Credentials that grant authority to a principal can become invalid over time due to certificate revocation and timeouts. Rather than exposing this fact through the GetPrincipal interface, invalidation of credentials will be handled at access control check time. This almost certainly means that the Principal object described above will be opaque to clients, and have an implementation that is shared between the process/channel security subsystem and the authorization subsystem.

It seems unlikely, at this point, that intra-machine channels as defined in Singularity will extend unchanged to inter-machine operation. However, there are ample examples in the literature of systems that provide authenticated and encrypted channels over which inter-machine messages can be transmitted. If Singularity adopts one of these schemes, the security architecture offered

in this document might then allow a receiver to model the authenticated identity of a remote message originator just as in the local case.

5. Access Control

The set of resources that might be controlled by access control policy in Singularity is diverse. Here is a non-comprehensive sample of some possible resources that might be controlled or encoded through traditional access control mechanisms:

- user files in one or more file stores;
- access to hardware devices by driver software;
- special rights for privileged programs (for example authentication modules);
- services available to applications (e.g., authentication, file systems, network services);
- ability to modify aspects of the name server space; and
- permission to install programs.

Singularity will provide conventional *access control lists* and *access control groups* as a primary method for controlling runtime access to protected resources. Singularity will also support capabilities, as described in Section 5.2, in the form of typed channels.

5.1. Access Control Lists

The Singularity system configuration subsystem will offer strong mechanisms to help ensure the integrity of system and application code, static data, and parameters. This means that many of many system and application-relevant objects might be protected without resorting to traditional access control. There will certainly be resources that fall outside of the scope of the system configuration, especially those that require confidentiality. The hope is that the number of these resources will be relatively few.

Access control in Singularity seeks to control access based on three specific sorts of rights:

- *rights propagating from the head of a principal chain*. These rights will typically be associated with a specific user or application that allows inheritance of rights through multiple invocations.
- *rights to access a specific type of protected resource*. These rights will typically involve the most recent invocation in a principal. For example, only the Word application might be allowed access to .doc files.
- *rights of intermediate principal components.* These rights can be used to specify what intermediate applications are allowable on a invocation chain.

Access control lists in Singularity are patterns. A principal requesting access to an protected object must match such a pattern to gain access to the object. It is likely that there will be several standard paradigms for constructing ACL patterns corresponding to common usage scenarios.

Singularity supports groups in order represent common shared components of ACLs. Groups are named in the trust hierarchy and represented in the system configuration. The exact form of the ACL and group pattern language is TBD. For now, thinking of ACLs as regular expressions and groups as fragments of regular expressions is sufficient.

Singularity will include a standard runtime component that allows for the maintenance and encoding of ACLs, and provides the ability to check whether a principal is a granted a privilege by an ACL. Stable storage for encoded ACLs will be handled by the code responsible for the resource, for example the file system service might store ACLs in the file system itself. (Alternatively, the file system might define all ACLs as groups and store only group names along with file data.) This facility, taken together with the ability of any Singularity process to reliably determine the sender of a message, provides sufficient mechanism to perform discretionary access control on requests that flow from other processes, or even within the same process.

5.2. Capabilities

Channels in Singularity can be viewed as a simple form of capabilities. Possession of a channel endpoint gives the caller the right to pass messages over a channel. Whether the caller can successfully invoke a service by doing so is of course at the discretion of the service provider. Hence, such access controls are ultimately discretionary.

Capabilities and access control lists have long been discussed as alternative methods of implementing an access control matrix. In some sense, the two schemes are duals of each other. Both have their benefits and drawbacks. Since Singularity is a research OS, it seems acceptable to offer both. In practice, this means that a process can give a channel endpoint to a second process and infer that messages received over the channel are from that process or from a third process to whom the endpoint was intentionally given. This train of logic might be used in lieu of an explicit access control check on messages received over a channel. Usually, a process will also perform some discretionary access control check prior to granting a capability. For example, file descriptors (as in Unix) can be thought of as capabilities that are granted only after an access control check at file open time.

6. Policy Management

One serious problem with existing systems is that security policy is maintained in a hundred different places. It's very hard to enumerate all the elements of a machine's security policy, or to ensure its consistency. This can be different in Singularity, since it offers a single place to stand, the system configuration, where all security policy is rooted. It will be critical that the system configuration presents a coherent model for representing security policy.

Applications are first-class objects in Singularity. It seems very appealing that access control enforcement might be dictated, to large degree, by which application is running. This is especially true in the home server environment, where it is likely that relatively few running processes will be associated with any given user, meaning that there will be relatively few different access control decisions to be made per user. Instead, most access control decisions will likely be based on the application-id of the requesting code.

It would be a great improvement if most access control policies could be expressed through a small number of well-defined rules, with explicit exceptions as appropriate. The following sections describe some of the policy statements that need to be expressed in any Singularity machine.

It is at this time unclear how best to implement the enforcement of policy in all cases. There will certainly be a discretionary access control system governing most runtime resources. The system configuration will certainly embody much of the installation-time policy mentioned below, and

perhaps other mechanisms will also exist. The correct blend of mechanisms for the initial version of Singularity is not yet well understood.

6.1. Credentials and Groups

The system configuration must contain sufficient information to support the authentication of machines, users, and applications. In general, authentication requires the validation of credentials offered by the authenticating party in the context of system policy. Successful authentication in Singularity will result in a name for the authenticated entity that is unique in the context of the local trust hierarchy.

The system policy needed for authentication of users and machines is well understood in the distributed systems literature. Singularity will adopt some collection of existing techniques according to the needs of its emerging prototype applications.

Application credentials are carried in the application manifest (see Section 6.4). They describe the code that is permissible for the OS to load under a given application identity. These credentials also include cryptographic keys and signatures that certify the code hashes in the manifest and vouch for the provenance of the application.

An application identity is a name in the local trust hierarchy. The leaf arcs in this name are typically defined by the application publisher. The universe of publisher names is specific to the local trust hierarchy, but will probably be standardized to correspond with some more global notion of publisher certification.

6.2. Application Installation

The installation of an application is in general a somewhat sensitive task, since it allows the application access to resources—in some cases, critical ones. Installation must therefore be authorized.

Installation can only be performed by an application with the specific permission to install applications. Initially, a specific administration utility (the *installation utility*) that is part of the system has this permission, although other applications may eventually be granted it. Installation is controlled by policy represented in the system configuration and involves decisions based upon the meta-data provided in the application manifest (see Section 6.4.)

Singularity will maintain two types of information related to application installation:

- An *administration policy* for establishing a set of privileged administrators (perhaps per application);
- An *installation policy* for evaluating and acting upon requests to install applications.

Administration policy determines what user credentials are necessary to run the installation utility. Note that the administrator might be physically present and operating a prompted user interface, or might be a remote entity communicating over a secured channel.

Installation policy determines which application credentials represent permissible applications for the local machine, and it maps application identities into groups of applications that can be managed in a similar fashion (see the next section).

Installation policy is represented in the form of trusted public keys and certificate-chains, simple predicates on certain information in manifests (application name, version number, publication

date, and so on), and groups of public keys and applications identities that represent the fundamental trust relationships in the system. The system configuration subsystem will specify a language for defining policy.

6.3. Application Groups

The key to allowing more manageable control of application permissions is to define a few categories (*standard application groups*) of application to which new applications can be assigned on installation, according to a set policy. Possible standard application groups include:

- *Normal applications.* This would be the default category for an installed, trusted application.
- *Applets.* Untrusted "network clients" would be installed with very limited privileges, equivalent to those granted today's Java applets.
- *System utilities.* Highly trusted applications installed for system administration functions (such as backup—although backup utilities have a specific enough set of privileges that they might get their own category) would get privileges equivalent to those associated with kernel code.
- *Sandboxed applications.* "Guest applications" (say, borrowing cycles and disk space for grid computing applications), or applications performing some function that can be completely separated from the rest of the computer, get permissions equivalent to normal applications—except that they are confined to a VM-like shadow version of the rest of the computer, in which all the persistent state associated with other applications is off-limits.

When an application is installed, it is assigned, based on the installation policy, to one or more standard application groups. These groups can then be listed in ACLs to define access rights to resources. For example, the group, "normal applications" might be included on the ACL for most data files, as well as standard system resources (printer, network, and so on). "Applets", on installation, would be given access to their "home" network address, as well as only those resources associated with applet operation (perhaps only UI devices).

Note that once installed, applications can then create application groups, own resources, and set access to those resources to include itself or other applications, groups (included those it created), or standard application groups. However, the application cannot change its own membership or non-membership in any of the standard application groups.

6.4. Application Manifests

The Singularity application manifest will contain a wealth of information about an application, some of which reflects on security policy. The definition of the application manifest is the domain of the Singularity Application Abstraction [SDN12]. Application manifests should support at least the following properties:

- Application name;
- identity of and hashes for code components;
- signatures and public keys to certify code hashes;
- shape of process sub-tree (e.g. subprocess/application creation rights);

- versioning information – equivalence classes of component versions;
- intended installation policy relevant to this application.

And, of course, any or all of the above information should be attestable by signatures, and public key data so as to establish a trust relationship between the executable code and a set of software publishers trusted by system policy.

A code component can be associated with one or more applications. However, since code components are only loaded into new processes with fresh and isolated process state, there will be no runtime sharing of state between applications due to sharing of code components

7. Future Work

7.1. Interposition

As discussed in Section 3, every new process receives an "initial channel" from which it can obtain core OS services such as the ability to communicate with other services. This architecture suggests a that it might be possible to allow a parent to *interpose* on a newly started child process by totally controlling its interactions with other processes (and the OS). Although this has yet to be proven, such interposition should make it easy to implement sandboxing environments in which untrusted child processes or applications can be run without risk of corrupting the rest of the system. In this context, the child process (or application) is encapsulated by the interposing agent and is dependent on the parent's continued operation.

Interposition services can be designed explicitly to allow sandboxing of portions of applications or encapsulated applications. When a parent process chooses to interpose, it can then proxy requests from the new process to the kernel or other services, possibly with modifications enroute. Arbitrary interposition policies can be implemented by proxying communications in this manner, but certain standard interposition policies might become popular. Examples include "read-only", forbidding changes to the file system or other persistent state beyond the application, or "virtual machine", defining some sandboxed collection of state (including some persistent state) outside of which the application has no access. Since objects (including channel endpoints) can only be transported via channels, controlling a process' ability to obtain a channel endpoint determines to which resources (including other processes) it has access.

7.2. Constraints on Capabilities

Many security problems are caused by code that allows attackers to exploit rights unexpected and unintended by the author. This will probably be the case for a long time to come, but Singularity can offer the possibility of tools that make secure code easier to write. Some of the ideas mentioned here represent speculative, but perhaps promising research directions.

Channels in Singularity are typed. Because the Singularity runtime has sufficient knowledge of all the runtime types statically known to an application, it is impossible for an application to obtain a channel endpoint for a type that is not included in the runtime scope of the program. If channel typing is implemented at a sufficient fine level of granularity, it will be possible to constrain the behavior of an application by the simple fact that the programmer didn't include the code to access specific resources. In some sense, the application manifest carries this security policy as part of the type system meta-data derived at compilation-time.

Within the confines of a single machine, it is not difficult to track the propagation of endpoints. Therefore, it is plausible to design policies that constrain the propagation and use of channels. It might be wise, in some cases, for the programmer to explicitly forbid propagation of an endpoint further than a specific destination process. This constraint could be easily implemented by the operating system. More complex application behavior can be modeled along the same lines. Suppose that an application typically obtains a channel to a portion of the network stack that can make HTTP requests. Suppose further that, in its manifest, the application code specifies a limited set of URLs to which access is required. It would be impractical to embed URLs within the type of the HTTP channel, however it is quite plausible to mechanically map from the application manifest to a set of constraints on channel usage. If automatically and securely conveyed, these constraints can be enforced by the service provider. By applying such defensive techniques, the programmer can intentionally limit the resources an application requires, and hence reduce the severity of unforeseen attacks.

Since inter-process communication in Singularity relies exclusively on channel propagation, it might be feasible to implement a mandatory access control policy dictating the acceptable usage of resources by controlling the propagation of endpoints. The main difficulty here is in establishing an acceptable policy given a collection of applications. It is possible that better compile-time understanding of program behavior will help in doing so.

8. Next Steps

The following tasks are considered high priority items in terms of the initial Singularity implementation:

- API and implementation for the OS “principal” abstraction
- Integration of principals with OS processes and channels
- API and implementation for groups and access control lists
- Agreement over initial design of installation policy w.r.t. system configuration
- Agreement about relationship between trust hierarchy and system configuration
- Implementation of user abstraction and rudimentary secure (off-machine) channels
- Agreement with application abstraction team over the requirements for application manifests

The following tasks will be of explicitly lower-priority for the first version:

- Capability constraints
- Interposition / sandboxes
- Mandatory controls on endpoint propagation
- Machine attributes in principals

