Xen-Cap: A Capability Framework for Xen

Yathindra Naik

University of Utah
Salt Lake City, UT USA
ydev@cs.utah.edu

Abstract

Hypervisors provide strong isolation and can be leveraged to disaggregate large software stack of a traditional, monolithic system. Disaggregation can be achieved by running individual applications and kernel components in separate VMs. Existing hypervisors, however, do not offer a fine grained access control mechanism. Without such mechanism, individual VMs still run in one of two extremes—complete isolation or excessive authority.

This project extends Xen with the capability access control model—a mechanism for fine grained, dynamic management of rights. Together with strong isolation, capabilities can be used to create least-privilege services—an environment in which individual applications have minimal rights, that are required to perform their tasks.

We discuss the design and implementation of a capability access framework for Xen. We also demonstrate examples of least-privilege services. Overall, we gained valuable insights for designing a secure system using an industry standard virtualization platform.

1. Introduction

Many of todays modern computer security problems come from the fact that traditional operating systems are monolithic. A compromise in one of the components of the OS makes it easy to gain control of the entire system.

Virtualization has become a de-facto standard in datacenter and web hosting environments. It is also gaining popularity as a mechanism for constructing novel execution environments on desktop machines [13]. Virtualization can isolate individual applications in separate VMs [3, 15]. But isolation alone cannot guarantee security. In addition to isolation, a secure system needs a flexible way to manage privileges available to individual VMs. Security models that are implemented with hypervisors do not offer flexible policies or fine grained access control.

Xen is a full-feature, open-source hypervisor, which provides support for both para-virtualization and hardware-assisted virtualization [1]. It is adopted in the industry as a default component of the cloud datacenter stack [6]. Xen provides mechanisms to run VMs in isolation, which can further be used to disaggregate individual applications or services of an OS and effectively sandbox their execution.

Capabilities offer flexible and fine grained access control mechanism and have been used extensively in many research operating systems [7, 9, 17]. Capabilities have been shown to work well for dynamic privilege management [10]. With capabilities, it is possible to construct least privilege services—an environment in which individual applications have minimal rights, that are required to perform their tasks. By augmenting Xen with capabilities it is possible to achieve isolation and fine grained access control.

Though implementation of capabilities is well understood in the object oriented microkernels [7], it is not clear how we can adopt the same model in a virtualized environment. Microkernel systems are designed from scratch to support capability model where everything in the system is treated as an object. Microkernels provide abstractions to operate on these objects. Xen hypervisor provides a more traditional hardware-centric abstraction such as device drivers, DMA, interrupts etc. It is challenging to map capability model which relies on object-centric abstractions on a system that provides hardware-centric model. In order to map the capability model on such a hardware-centric design, it would require redesigning many of the fundamental abstractions in the system. This will require making changes that can

affect many parts of the system including the application layer and therefore implies lots of code changes.

Our contributions are the following:

- A capability framework to do fine-grained access control in Xen.
- Examples for constructing more secure services using capability framework.

2. Capabilities

2.1 Overview

Capability [5] is a special token that uniquely identifies an object (resource) and allows the application to perform certain operations on that object. Any application that possesses a capability can interact with the object in ways permitted by the capability. Those operations might include reading and writing data associated with the object, and executing a function of the object.

In a classic capability object model, capabilities are implemented as references to system objects. Every resource in such a system is an object, e.g., a memory page, an IPC entry point, a hardware device [7, 17] etc. Applications *invoke* capabilities to perform operations allowed on that object. The capability invocation is the only IPC mechanism in the system.

Hypervisors do not offer objects as a fundamental abstraction. Neither do hypervisors have a unified IPC mechanism which can provide a meaningful interposition interface for enforcing system wide access control.

In a virtualized system, most system resources still run in the unmodified operating system kernels and applications which implement them. Some objects, like hypercall invocation points, memory pages, etc., are implemented and can be protected by the hypervisor. But the high level objects could be anything the application chooses to protect such as a file, or a record in a database. The hypervisor itself cannot make sense of the high level objects. Hence a capability access check cannot be enforced in the hypervisor.

A pragmatic approach to implement a capability access control model on a hypervisor is to separate enforcement of capability rules and access checks. Capability rules—integrity of capabilities, exchange of capabilities—are enforced in the hypervisor, but the access checks are made by a high level code. This allows us to extend the capability model into traditional applications and kernel components with only minimal changes to their code.

2.2 Xen-Cap's view of capabilities

In our system, a *capability* is a record in a hypervisor protected data structure. Please refer to Figure 1 for more details. We call this data structure the *capability address space*, or the *CSpace*. VMs are subjects in our model, i.e, each VM has a private CSpace, and all the code inside it has the same rights.

Each capability is identified by a system-wide unique 64-bit name. The name by itself does not provide access to an object. To obtain an access, the capability with this name must be added to the CSpace of the VM.

In order to guarantee authenticity properties of capabilities, i.e. anyone with capability is authenticated to use a resource, we need to ensure capability names are unique. We create capabilities using the multiply-with-carry (MWC) random number generator invented by Marsaglia [11, 12]. We chose this method for two reasons:

- It is fast since it involves computer integer arithmetic.
- It has extremely large periods ranging from around 2^{60} to $2^{2000000}$.

Consider a period of 2^{60}, if the system allocates a capability every 300ns then it would take approximately 10960.4 years for an overflow to occur.

We also create capabilities at runtime and do not use a persistence store. Using persistance store to remember capabilities would require securing the persistence store. We would also need to address file system consistency issues. Therefore, we did not have time to consider such a design in this work.

As we mentioned earlier, capabilities by themselves do not provide any security guarantees in our system. To enforce access control, capabilities are bound to high level objects, by the code which manages these objects, e.g., files in a file system, internet connections in the network stack, memory pages, and even virtual machines. The code which manages the objects is responsible for enforcing the access control checks. Hypervisor protects its own objects such as hypercalls, memory pages, IPC end points, etc. Hypervisor ensures VM-level protection. However, if higher level objects need to be protected, we extend the TCB into the code that implements them.

The hypervisor however makes sure that VMs cannot forge capabilities and can only acquire them through an authorized *grant* hypercall. This simplifies the work of the high level code—to check if access is permitted, the high level code can just invoke the hypervisor *check*

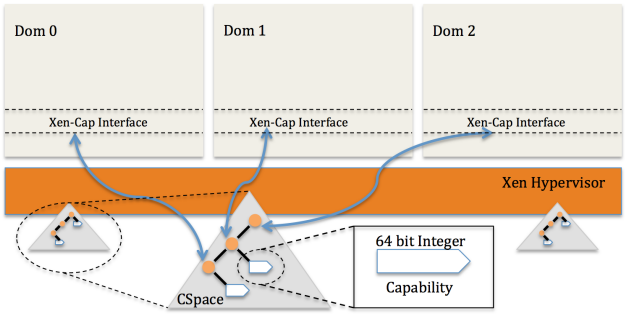

Figure 1. Xen-Cap Capabilities

call. If the capability is present in the capability address space of the virtual machine which tries to perform the access, then *check* returns OK and the high level code can permit access.

3. Xen-Cap Interface

The main components of the Xen-Cap capability interface are three hypercalls - `cap_create`, `cap_grant` and `cap_check`.[1] The only way to interact with capabilities are through these hypercalls. We'll explain how capabilities are initialized in Section 5.3. Now we'll describe how VMs can interact with capabilities.

3.1 Flow of capabilities

- *Creating and binding capabilities*: A capability is created with the `cap_create` hypercall. `cap_create` creates a capability in the calling VM's CSpace and returns the name of the capability to the requesting application. The application then binds the capability with its object. Binding means application now knows which capability to check on access to a particular object. The application decides how to store this information.
- *Distribution of Capabilities*: Application can grant capabilities to other VMs by using `cap_grant` hypercall. The grant will succeed only if VM has a capability with that name in its CSpace, and if VM has the right to grant it according to the rules of the model.

  Distribution of capabilities involves another interesting problem. Initially only the application which created a capability knows about its binding. If we want to distribute rights to this object, we need some querying mechanism for applications to get the names of the capabilities bound to these objects[2].

  For example, consider a shared file system where we want to allow access to specific files. Since only the file system knows which capabilities are bound to which files, we need a way to query capability names to grant them to specific applications. In order to query capability names, we introduce a mechanism called as `get_cap_names` API. `get_cap_names` takes a high level object, i.e, file name, and asks the system which manages that object to return the name of the capability bound to that object. Then the granting VM can perform the grant operation, but of course only if it has a corresponding capability in its CSpace.
- *Checking Capabilities*: Any application process in our model will permit access only if the requester has a valid capability. This is achieved by using the `cap_check` hypercall. The check operation takes a domain identifier of the VM which tries to perform the access and a capability name. It then performs a lookup on the CSpace data structure of the VM which requests the access. If it finds the capability in CSpace, it returns OK.

3.2 Xen-Cap interface—Libxl and Linux Kernel

Xen-Cap interface for capabilities is available from two levels in the guest systems—user level apps, and guest kernel. At the user level, applications can use the xen specific library which in turn invokes the `cap_*` hypercalls.

[1] We did not have time to implement `cap_revoke` which revokes an access to a resource. This can be useful to cut-off communication between entities and also create isolated islands of services. We did not have time to look into take-grant model [10] which deals with issues of granting capabilities to many entities.

[2] The grant rule checks are not not supported yet—we always grant right now

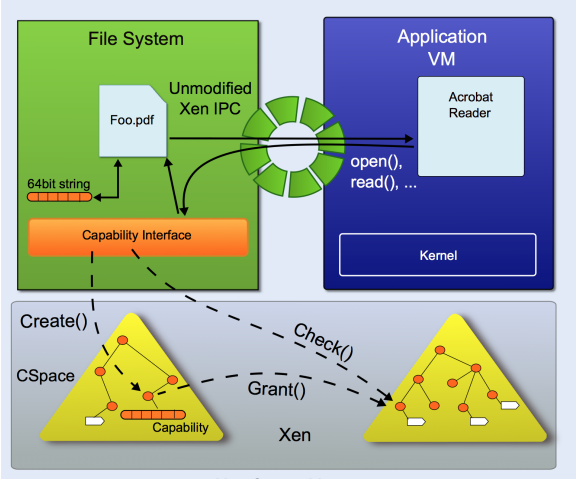

Figure 2. Xen-Cap Architecture

To support capabilities in guest kernel, we introduce our capability hypercalls inside the Linux guest kernel. It follows the same semantics as in Xen. This helped us patch the frontend drivers in Linux guest kernel. The frontend driver code can be found in the Linux source code under drivers/block/ and drivers/net/. We have patched `talk_to_netback()` for network frontend driver via `xenbus_cap_op()`. `xenbus_cap_op()` creates the necessary capabilities via `cap_create` hypercall and does the grant. We also had to patch `connect()` routine in (drivers/block/xen-blkback/xenbus.c) which handles virtual disk backend from guest via `xenbus_cap_op()`.

3.3 Example of Sandboxing a PDF reader

In Figure 2, the capability is shown as a 64-bit string and it is bound to the object Foo.pdf by the file system code in the File System VM. Capabilities are protected by the hypervisor in a fast lookup data structure called CSpace. VMs can access capabilities only through well-defined set of capability hypercalls - `cap_create`, `cap_grant` and `cap_check`. The Application VM hosts a PDF Reader and is created with capabilities to access Foo.pdf. Application VM uses the existing Xen API's to read the PDF file. The Application VM uses the capability interface to protect those files. Any request to access Foo.pdf results in a capability check on the Application VM's CSpace.

Capabilities flow according to `cap_*` API's and this defines the life cycle of the capabilities in the model we have implemented so far.

3.4 CSpace Organization

We create capabilities in the hypervisor protected memory and hold them in a fast lookup data structure—*CSpace*. Designing a fast lookup data structure for capabilities is a research problem in itself. As we mentioned before, in a capability system, access to any resource—every memory page, files, PCI devices in the system is via capability objects. The problem is to design a efficient data structure that can easily scale to billions or more objects in the system. Consider a system with 4GB of physical memory and 4kB page size. To protect every memory page on this system, we would need 1048576 capabilities.

Therefore, a practical way to adopt capabilities on systems such as Xen is to design auxiliary data structures to manage capabilities. In our design, we use an array data structure as CSpace for simplicity. This can be easily replaced with other fast lookup data structures such as a hash table. In order to scale capabilities to a large number, we could think of designing some kind of a sparse tree like data structure where the inner nodes serve to translate the address and leaves are either data or capability pages similar to EROS operating system [17].

Our current implementation, which serves as a prototype, relies on arrays to make things simple and we plan to replace this with other high lookup structure in future.

4. Xen Overview

Xen is a baremetal hypervisor which provides platform to run multiple operating systems. It provides a hypervisor, a special virtual machine that helps manage other

guest VMs (Domain 0) and infrastructure for device discovery, virtual machine creation etc. Please refer to Figure 3 for more details.

A running instance of a virtual machine is called a domain or guest. A special domain, called *Domain 0* contains the drivers for all the devices in the system. Domain 0 also contains a control stack to manage virtual machine creation, destruction, and configuration.

The major components of Xen are as follows:

- **The Xen hypervisor** is a minimal microkernel that runs directly on hardware and is responsible of managing physical CPU, memory, and interrupts. Hypervisor provides a hardware-like interface sufficient for running traditional operating systems inside isolated virtual containers.
- **The Control Domain or Domain 0** is a specialized virtual machine responsible for managing other guest domains. Domain 0 has direct access to physical hardware and typically hosts backend device drivers. It is the first VM started by the system.
- **Toolstack and Console**: Domain 0 contains tools to manage virtual machine creation, destruction , and configuration. The toolstack is either driven by a command line console, by a graphical interface or by OpenStack or CloudStack.
- **XenStore** is a light-weight database that stores VM configuration details. It exposes a file system like interface to write and read from the database. It is also important for the split-device driver architecture that Xen implements to support para-virtualization of device drivers.

The Xen hypvervisor implements a small subset of traditional operating system abstractions: virtual machine scheduling, page-level memory management and address spaces, memory sharing across domains, and interrupt-like notification channels. In contrast to many microkernels, the Xen hypervisor does not implement any IPC mechanism besides a single-bit, interrupt like notification channel. Guest virtual machines are free to implement any IPC mechanism on top of two primitives: the shared memory mechanisms, and the event channels.

A typical guest virtual machine, called Domain U, starts with four virtual devices: console, Xenstore, disk, and network. A guest virtual machine does not have access to real physical devices. Instead, Xen provides the guest with a notion of a virtual device, which is implemented with two components - a backend and frontend split device. The backend and frontend devices run in the Domain 0, and the guest virtual machines, and work as a proxy-stub pair, which provides access to a physical device running inside a privileged device driver domain.

The frontend device driver is the normal device driver that comes with the guest operating system. Instead of talking to the real device, frontend driver talks to the backend device driver.

The backend device driver typically in Domain 0, routes all the requests from the frontend device driver to the actual physical device.

The split-device driver model relies on a shared memory page and event channel communication primitives to establish a high-throughput cross virtual machine communication mechanism.

5. Securing Services

A Xen VM interacts with other VMs and the hypervisor through the following interfaces:

- **Hypercalls** are well-defined and limited functions to access hypervisor services
- **XenStore** is a system-wide configuration database with a publish-subscribe interface
- **Event-channel** and **shared memory** are two inter-VM communication mechanisms

Xen comes with the `xl` toolstack which provides mechanisms to create, destroy and configure guest virtual machines. Xen-Cap interfaces with `xl` toolstack to grant capabilities to guest VMs during their creation.

5.1 Xen objects

Xen objects are hypervisor-level objects, i.e. hypercall interface, event-channels, PCI devices, shared memory. In order to mediate access to Xen objects, we use XSM framework. Similar to FLASK [19] and SELinux [18], XSM [4] is a generalized security framework for Xen. XSM has security hook functions throughout the hypervisor. XSM provides simple ways to override the default hook functions with one of our own. This allows plugging in different security models without having to re-write a lot of code. The security hooks mediate access to Xen level objects such as hypercalls, event channels, memory pages, etc. When the hook funtion is triggered, control is transferred to the Xen-Cap interface, i.e. `cap_*` interface.

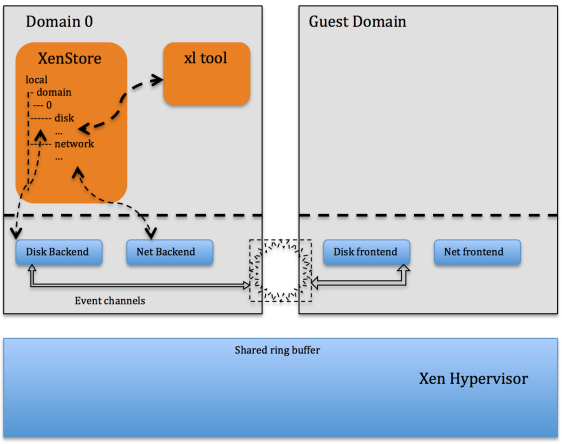

Figure 3. Xen Architecture

5.1.1 Hypercall Interface

The guest virtual machines access Xen services by using the hypercall interface. Hypercalls are implemented like system calls in a traditional operating system. Hypercalls are well-defined and limited in number. In Xen, the hypercalls gives virtual machines access to use event channels, map pages, grant page access to other VM's, send interrupts, reserve memory, etc. Securing hypercalls is important in minimizing authority of a process. By limiting the hypercalls allowed for a VM, we can effectively isolate the system resource that VMs can access.

Creating and binding capabilities: We create capabilities for every hypercall in Xen. For binding capabilities, we use a shared memory page (more about this in Section 5.2).

Distribution of capabilities: The VM config file has an additional parameter we introduced `cap_hypercalls`. It includes the names of all the hypercalls that the VM is allowed to invoke. During the creation of VMs, we parse this parameter and grant the capabilities for hypercalls listed.

Checking capabilities Every hypercall invocation is intercepted via XSM and `cap_check` is invoked. If `cap_check` returns OK, we allow the hypercall, otherwise the hypercall fails.

5.1.2 Event Channels

Unlike traditional microkernels, Xen does not implement a high-level IPC mechanism. Instead, the Xen intervirtual machine communication is built on top of two simple primitives: shared memory [3], and interrupt-like event channels.

Event channels are used as a signaling mechanism in Xen. An event notification is equivalent to a hardware interrupt.

Securing event channels with capabilities allows us to restrict the communication channels available to VMs. We can selectively allow specific VMs to communicate. For example, we allow a PDF Application VM to only communicate with the File System VM but not others.

Creating and binding capabilities: We create a capability for every VM to establish event channel communication with other VMs. In order to establish event channels with other VMs, there is a exchange of capabilities between the two VMs. In order to bind the capability, we use the VM structure in the Xen hypervisor. We introduce a new hypervisor level object in the domain structure called `est_evtchn`. We create a capability with `cap_create` and bind the capability to the VM structure, i.e the `est_evtchn` object. Any VM that wants to communicate with another VM must have `est_evtchn` capability. Since Domain 0 is booted first, we create a `est_evtchn` capability and bind it at boot time. For other guest VMs, we create `est_evtchn` only if its config file specifies it.

Distribution of est_evtchn capability: In order to grant capabilities for event channels, we introduce a new parameter to the VM config file called `cap_evtchn`. It is a list of domain id's for which we grant `est_evtchn` capability. During VM creation, the libxl code parses the

[3] We did not have time to work on shared memory in this work

config file and grants `est_evtchn` capability to every domain listed in the config file.

Checking capabilities: On every event channel operation such as `alloc_channel()`, `bind_interdomain()`, `evtchn_send()`, etc., XSM intercepts these calls and transfers control to Xen-Cap interface. Then, we invoke a `cap_check` to verify if the domain requesting event channel operation has `est_evtchn` capability in its CSpace. If the check returns OK, then the VMs are allowed to use event channels.

5.2 Boot protocol

The Xen hypervisor grants capabilities to all the hypervisor-level objects to the first booting domain. In a default Xen setup, Domain 0 which contains all the tools and backend device drivers, boots first and receives capabilities on all the hypervisor-level objects. We therefore grant capabilities required to access hypervisor-level objects.

During system boot, Xen discovers and creates capabilities for resources such as PCI devices, memory page table, device drivers, etc. In the later stages of boot, Domain 0 is created. XSM intercepts domain creation function and allows Xen-Cap to allocate memory for CSpace. Once the CSpace for Domain 0 is allocated, hypervisor queries capability names for all the hypervisor-level objects such as hypercalls, PCI devices, memory pages, etc., and grants them to Domain 0.

Next, we need the `get_caps_name` interface to query capabilities bound to these hypervisor-level objects. We implement this as a shared memory page called BOOT_CAP_INFO during boot phase. Once the capabilities are created for the hypervisor-level objects discovered during boot, we store them in BOOT_CAP_INFO memory page.

5.3 XenStore

5.3.1 Overview

Xen store is a light-weight key-value store. It mainly has configuration information of each virtual machine. XenStore aims to simplify development of additional drivers by providing higher level abstractions like read, write, list and so on. XenStore offers a hierarchical namespace for keys which gives users the ability to group similar keys together under a single directory. Every key-value pair is also known as a node internally.

XenStore has a particular format for entries and they are grouped under particular directories. There are 3 paths:

- /vm - This holds information related to VM specific configuration. Number of processors, typer of kernel, boot parameters etc.
- /local/domain - This holds information about various device drivers, memory stats, state of VM etc.
- /tools - This holds information related to various tools.

XenStore provides high-level semantics which look like this,

```
write('/local/domain/4/NFS/1', '/file1')
write('/tool/guest/foo', 'xs')
```

where the user provides the key-value pair. Note that internally, node names are key names. Please refer to Figure 4 for more details.

5.3.2 XenStore's default permission model

XenStore has a notion of security for nodes which is associated with an access control list of VM id's that are allowed to read/write nodes or do both.

```
set_perms('/tool/mytool', { 'dom'   : 0,
                            'read'  : True,
                            'write' : True,
                            'dom'   : 1,
                            'read'  : True,
                            'write' : False})
```

The first <domain id,permission> is the owner of the node and the permissions on the first pair are the default permissions for any of the domain id's not specified in the list. Owner domain has both read/write permissions by default. The other pairs that follow have the permissions that are specified.

Generally, each domain makes entries under /local/domain/-<domain-id>/. Guest domains grant permissions to Domain 0. This is because Domain 0 needs to know the state of the guest domains. More important reason is that backend drivers are typically hosted in Domain 0 and they need to communicate with the frontend drivers that come with the guest.

5.3.3 XenStore and split-device driver model

XenStore provides a mechanism for the initial handshake process between frontend and backend drivers. When guest boots up, libxl code initializes the entries

for frontend virtual disk driver, virtual network driver, console etc., depending on the specification provided through config file. Backend drivers put a watch on these nodes prior to their creation. Watch is a notification mechanism provided by XenStore. Any updates to these nodes trigger the watch callback function on the virtual machine that put a watch. Once the guest frontend entries are written, backend is notified and both of them change the state accordingly.

5.3.4 Capabilities for XenStore

Creating and binding capabilities: The first thing we needed was to find a way to bind capabilities to each node in XenStore. XenStore does not have a general security framework similar to XSM or SELinux. We introduce capabilities straight to the XenStore code, when the nodes are created in XenStore in `construct_node()` via `cap_create`. The capability is bound to the nodes (inside the node struct) and also stored in Domain 0's CSpace.

Mapping XenStore permissions to Xen-Cap capabilities: We needed to map the permission model used by XenStore to our capability model. We create a read capability and a write capability for each node. We take care to grant the least amount of privileges necessary for the correct functionality of existing tools. We grant both read/write capability to owner domains. We converted the existing XenStore permission model by introducing our routine `xs_strings_to_caps()`. This understands the semantics of XenStore permissions and does the necessary capability creation and grants the same. XenStore keeps track of the creation of new guest VMs using its own domain structure. We maintain a flag in this structure to indicate if the domain is in capability mode. This takes care of VM's that are not in capability mode and can can fall back to the default XenStore permission model.

Distribution of capabilities: In the default XenStore permission model, when a new node is created, it inherits the permissions from its parent node. We felt that this kind of permission inheritance limits our reasoning about authority. Therefore, we disable inheritance and change all `xl` tools to explicitly grant capabilities to each node in XenStore. We introduce `xs_get_caps(path)` API to query the capabilities needed to access a particular node. With this in place, we can explicitly grant capabilities rather than follow an inheritance model. We had to patch a number of places in libxl code path where we call `xs_get_caps()` and then do `cap_grant`.

Checking capabilities: Whenever a XenStore node is created, we invoke `cap_check` on the parent node. This ensures child node is created only if the parent node has write capability on it. By default, reading the value of the node in XenStore requires the read capability. XenStore communication abstractions dictate if we need to check read or write capability. For example, looking up a value of a particular node invokes a `cap_check` for the read capability on that particular node.

5.4 Network File System

5.4.1 Overview

File systems have a large code base and represents one of the major subsystems in the OS. Users rely on file system to share files and directories for collaborative work. Sharing often brings the question of what to share with whom. In other words, we need a way to secure the file system even when there is collaboration. The default Linux ACL access control model is inflexible to implement the principle of least privilege, e.g. share a single file with another VM like our PDF reader example. Capabilities provide a flexible and simple mechanism to construct controlled communication flow without requiring complex policies.

In order to isolate the file system, we run the file system in a separate VM and export the directories to application VMs. One of the best ways to export a file system to another VM is via Network File System (NFS). NFS is a distributed file system protocol used to export file systems over a network connection. It offers the most simple way to share files over network. Please refer to Figure 5 for more details.

5.4.2 Capabilities for NFS

Creating and binding capabilities: In order to bind capabilities to files, we needed some metadata attached to the files to store capability names bound to each file. One option we considered was to bind capabilities inside file's inode. Though this seemed like a good option, we did not want to deal with different filesystems and their representation of inodes. Hence, we introduced our own metadata for binding capabilities to files. We needed something unique about a file that could serve as a binding between the file and the capability. The inode number of a file seemed to be a viable option. The inode number is unique during the lifetime of a file.

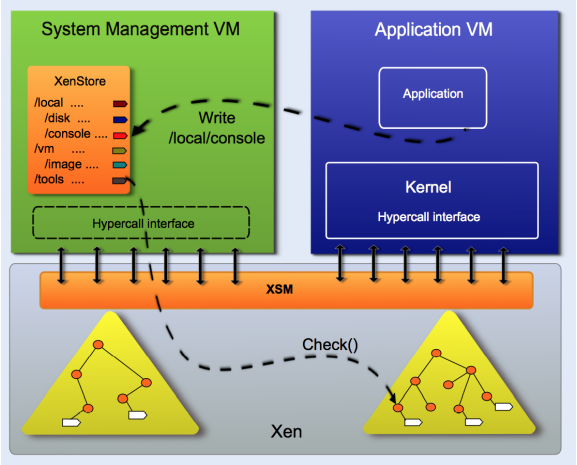

Figure 4. XSM under Xen-Cap. Every node in XSM is binded with a capability. Access to these nodes will result in checking capabilities.

We created a mapping for inode number of a file to the capability and maintain this mapping in a hash table. The key to this hash table is the inode number. We use a hash table implementation called uthash developed by Hanson [20]. It is a very flexible implementation that allows any C structure to be a part of the table. We had to make few changes to it for using it in our kernel modules.

Distribution of capabilities: To grant capabilities for exported directories and files, we explicitly mention their full path names in the VM config file. For each of these files, we grant the capability to the guest VMs. See more details in section 5.4.3.

Checking capabilties: One issue here is that NFS exports files to client VMs and the only way we can grant access to files is by verifying client VMs capabilities. But our capability interface works with virtual machine as its subject. Hence, we needed a way to map the client VM's IP address to it domain id.

To resolve VFS request to domain ids, we maintain IP address to domain id mapping in a hash table. Once a NFS request for a file comes from the client VM, we can track the client ip address. And then we use the mapping table to get its domain id and we can invoke `cap_check` on the client VM. Once the client has the required capability, we grant access to the file. This is the overall picture of how things flow for file system in our capability model.

5.4.3 Implementation details

NFS server domain: We modify the config file to enable exporting files from the VM that acts as the NFS server. We call this config parameter `cap_files`. The NFS server lists the files that it wants to export to the guest VMs through `cap_files`.

```
cap_files = "/files/foo /files/bar"
```

The guest VMs specify the NFS server VM's id along with the files it needs to access.

```
cap_files =
  "sharedfs backend:/files/foo,/files/bar"
```

Once we are in the process of creating the VM, libxl routine `initiate_domain_create()` invokes `libxl_domain_setcapfiles()`. This routine parses the `cap_files` from the config file and iterates through the list of files and writes them in XenStore under NFS server VMs directory. Specifically, under `/local/domain/<NFS server>/NFS/backend/`. As soon as these entries are made in XenStore, a watch is fired that starts the xen-cap-nfs module in the NFS server domain.

The xen-cap-nfs module reads the list of files to be exported from XenStore. For each of the file, we convert the file pathname into inode numbers. Once the conversion succeeds we invoke `cap_create` for each file and add the capability name and the inode number in the hash table. This completes the binding process for NFS sever domain.

Once the NFS server domain comes up, we rely on the Linux initialization script (rc.local) to indicate that the file system is up and running. This is because the routine `kern_path()` has to convert the file pathnames in the NFS server domain to their inode numbers.

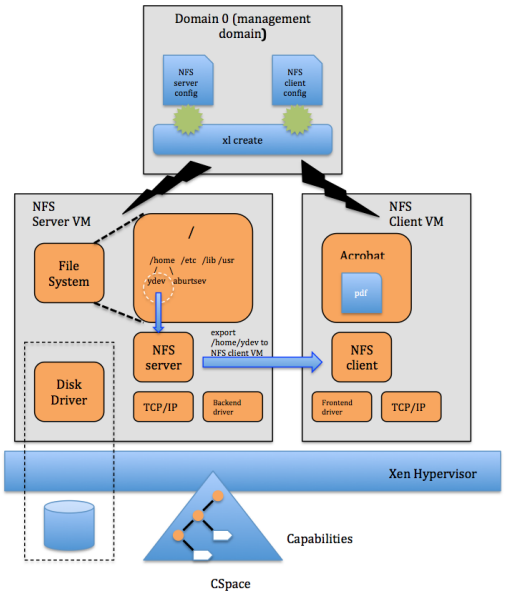

Figure 5. NFS under Xen-Cap. NFS server VM exports a part of the file system to client VMs. The client VM is created with capabilities to only a part of the file system. Once client tries to access files from the server, capability checks are made on the incoming request and access is granted.

The initialization script invokes the xen-cap-nfs module which starts the process of creating capabilities and binding them to file inode numbers. The module goes through all the files that needs to be exported, converts them to their inode numbers and creates capabilities and adds them to the hash table.

NFS client domain: The libxl code while creating the NFS client domain, write its IP address and domain id to XenStore. This will later help in converting the NFS client request to their domain id. We maintain a mapping of the NFS client IP address to its domain id in a hash table.

The libxl code parses the `cap_files` parameter from the config file. It iterates through the domain id:<file pathname> list, queries the capability names for each file and grants the same.

Once the NFS client domain comes up, it mounts the exported file system from NFS server domain. The client sends NFS file requests and the NFS server first determines whether it has a mapping for the incoming client request by looking up its IP address from the hash table. If a mapping is found, we get the domain id of the client VM and then the server looks up the capability for the requested file name by its inode number. After fetching the capability, server invokes `cap_check` on the client VM. If it succeeds then access to file is granted to the client.

5.5 Xen-Cap changes to VM config file

We made several changes to Xen's VM config file. We needed some way to specify hypercalls allowed to a particular VM. Hence, we introduced the `cap_hypercalls` parameter, which is a list of hypercall names allowed for a particular VM. Please refer to Figure 6 for more details. We introduced a new routine called `libxl_domain_setcapfiles()` that parses the `cap_hypercall` list and grants capabilities to these hypercalls. We have to grant capabilities to the VM early during the boot process otherwise the VM creation will fail. We have complete control over a VM through config file and once we make sure config files can only be edited by the creator of the VM, we can be confident that we have secured the VM.

We also added the `cap_files` parameter to config file which lists the directories and files that is shared by the NFS server VM. We have distinguished the server and client here. The VM which exports the files has a particular format. The client VM which accesses the files also lists the server domain names and the files it wants to access from them.

```
#
#   XEN-CAP GUEST CONFIG FILE (SERVES AS BACKEND FOR VIF DRIVER)
#

# This config file has additional parameters to support Xen-Cap
# The additional parameters are cap_files, cap_hypercalls, cap_evtchn

# cap_hypercalls - List of hypercalls allowed for this VM
# cap_files      - List of file pathnames that are being exported to other domains
# cap_evtchn     - List of domain id's which can establish event channel communication

kernel      = "/boot/vmlinuz-3.7.5+"
ramdisk     = "/boot/initrd.img-3.2.16emulab1"
memory      = 192
name        = "backend"
blkif       = "yes"
netif       = "yes"
vif         = ['ip=155.98.36.197']
disk        = ['phy:/dev/loop0,xvda1,w']
root        = "/dev/xvda1"
extra       = 'xen-cap-lsm=y security=xencap earlyprintk=xen console=hvc0'
cap_hypercalls = "memory_pin_page getdomaininfo setvcpucontext  pausedomain
unpausedomain resumedomain  max_vcpus  destroydomain  vcpuaffinity  scheduler
getvcpucontext  getvcpuinfo  domain_settime  set_target  domctl  set_virq_handler
setdomainmaxmem  setdomainhandle grant_mapref  grant_unmapref  grant_setup grant_transfer
 grant_copy  grant_query_size  memory_adjust_reservation  memory_stat_reservation
console_io  profile schedop_shutdown evtchn_unbound evtchn_interdomain evtchn_send
evtchn_status   evtchn_reset get_pod_target set_pod_target map_domain_pirq irq_permission
iomem_permission pci_config_permission"}
cap_files   = "/files/foo /files/bar"
cap_evtchn  = "0"
on_crash    = "preserve"
```

Figure 6. Xen-Cap VM config file

And finally, `cap_evtchn` lists domain id's which can establish event channel communication with one another. Here is an example config file,

6. Related Work

Securing large stacks of untrusted software is an ongoing effort. There have been numerous research effort in finding the right security model. As experience has shown, none of the security models have been able to solve all the problems. There have been a number of mandatory access control frameworks (MAC) such as SELinux [18], AppArmor [2], Smack [16], and Solaris Trusted Extensions [8], that help but does not completely solve the problem. MAC frameworks depend on security policies and it becomes a problem to come up with a policy to deal with every situation.

Capsicum is the latest research work that implements capability access control framework in FreeBSD at the level of file descriptors [21]. Capsicum helps sandbox applications and disaggregate privileges at the user-level applications. However, similar to MAC framework, Capsicum does not protect against a number of attacks on the operating system kernel.

There have been numerous research operating systems [3, 7, 15, 17] that implement capability systems from scratch. However, there have been no solutions to traditional desktop and server environments.

There are solutions that leverage hypervisors to provide isolation like Xen-Cap. Some of them are Qubes [15], Bromium [3], and XenClient [22]. These solutions attempt to sandbox applications by executing them in a separate VM. Complete isolation works well in environments where sharing is not required. But many subsystems in the operating system are inherently designed to provide sharing of resources (e.g. file systems, block storage, network stack), require mechanism for secure sharing to avoid operating with reduced functionality or with excessive privilege. Xen-Cap provides ways to secure environments where is sharing is necessary. Securing NFS under Xen-Cap was one such ways of achieving the same.

Disaggregation of core operating system services, device drivers, and hardware devices is important. Sandboxing applications in individual VMs helps to reduce privilege attacks but does not address the attacks against devices or device drivers. Xen [1], VMWare [14] and Bromium [3] by default run all the device drivers in a single privileged domain. Therefore, a vulnerability in one of the device drivers could lead to a compromise of the entire guest VM.

7. Conclusions

Xen-Cap describes our approach to implement the capability model on the Xen hypervisor. Xen-Cap serves as a simple prototype to examine the issues one can face while designing a capability model on a virtualized system. Our approach was aimed to reduce re-engineering effort which resulted in simplifying a lot of details. We were able to use Xen-Cap to secure critical services running on Xen without significant amount of code changes. The nice feature of Xen-Cap is that it is simple to use (just three hypercalls).

We have some more areas that needs work in future. The capability rules that we have implemented works well for simple scenarios. The grant rule requires more thinking as we do not address issues involving transitive grants. We did not have time to implement revoke rule. Revoking capabilities will require mechanisms to track which domain has those capabilities in order to do it recursively. The CSpace data structure needs to be replaced with a more efficient structure. We could make use of read-only capability caches inside the hypervisor to enhance CSpace lookup. Naming resources with capabilities will require a different design. We may have to adopt a capability table similar to seL4 or EROS. Using persistence store for capabilities is another area we haven't looked at. To make isolation completely transparent and quick, we need mechanisms to spawn light-weight VMs.

Overall, this is a step towards realizing a production system that offers isolation of services with flexible and fine-grained access control model.

8. Acknowledgments

The author would like to thank Anton Burtsev, Robert Ricci, Mike Hibler and all the members of the Flux Research Group for their contributions. Anton Burtsev spearheaded the idea and envisioned the entire project. Without his guidance, the project would not have seen the light of the day. Robert Ricci, my advisor, would always sit with us and help us breakdown the problem and provided valuable suggestions. Mike Hibler was last stop for any issues to get resolved. His experience with systems helped us in many situations.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] Bauer, M. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148):13, 2006.

[3] Bromium. Bromium Micro-virtualization. *whitepaper*, 2010. `http://www.bromium.com/misc/BromiumMicrovirtualipdf`.

[4] G. Coker. Xen security modules (xsm). *Xen Summit*, pages 1–33, 2006. `http://mail.xen.org/files/summit_3/coker-xsm-summit-090706.pdf`.

[5] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of The ACM*, 9:143–155, 1966.

[6] Amazon EC2 underlying architecture. `http://openfoo.org/blog/amazon_ec2_underlying_architecture.html`.

[7] D. Elkaduwe, G. Klein, and K. Elphinstone. *Verified Protection Model of the seL4 Microkernel*. 2008.

[8] Faden, G. Solaris trusted extensions. *Sun Microsystems Whitepaper, April*, 2006.

[9] N. Hardy. KeyKOS architecture. *Operating Systems Review*, 19:8–25, 1985.

[10] R. J. Lipton and L. Snyder. A Linear Time Algorithm for Deciding Subject Security. *Journal of The ACM*, 24:455–464, 1977.

[11] G. Marsaglia, B. Narasimhan, and A. Zaman. A random number generator for PC's. *Computer Physics Communications*, 60:345–349, 1990.

[12] G. Marsaglia and A. Zaman. A New Class of Random Number Generators. *Annals of Applied Probability*, 1:462–480, 1991.

[13] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *Architectural Support for Programming Languages and Operating Systems*, pages 291–304, 2011.

[14] Rosenblum, M. VMwareâĂŹs Virtual PlatformâĎć. In *Proceedings of Hot Chips*, pages 185–196, 1999.

[15] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[16] Schaufler, C. Smack in embedded computing. In *Proceedings of the 10th Linux Symposium*, 2008.

[17] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.

[18] Smalley, S. and Vance, C. and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.

[19] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *USENIX Security Symposium*, 1999.

[20] uthash: A hash table for C structures. `http://troydhanson.github.com/uthash/`.

[21] Watson, R.N.M. and Anderson, J. and Laurie, B. and Kennaway, K. Capsicum: practical capabilities for UNIX. In *USENIX Security*, 2010.

[22] XenClient. `http://www.citrix.com/products/xenclient/how-it-works.html`.