What are Capability-Based Systems?

Posted on October 31, 2016 by Gabe Parmer

Composite is a capability-based OS. There have been many capability-based systems in the past, spanning from those that used hardware-capabilities such as Hydra[1], to those that implement them in software such as KeyKOS, Eros, Fiasco, and seL4. An old, but still seminal overview of capabilities is Levy's book. This post discusses capability-based systems in general, only with some references to *Composite* where a concrete example is required.

Operating Systems and Resource Access Abstractions

Operating systems must provide an interface that separates untrusted, user-level computation from raw hardware resources such as memory frames, the CPU processing power, interrupts, and device drivers. There are a great many ways to do this that expose trade-offs along multiple dimensions. Many OSes such as UNIX variants do this by providing high-level abstractions such as processes (with threads, signals, coordination schemes, IPC, and memory management), files (with the hierarchical file-system, mapping, and psuedo-files), and sockets (with networking stacks, protocols, and device options). A core of logic in the kernel intelligently spreads hardware resources among the processes to provide each abstraction. Where shared namespaces exist (i.e. enabling a way for two processes to address the same resource) such as the hierarchical file system namespace, processes have access to and can modify the same resources. That enables them to impact each other's execution, for example by using the shared namespace for inter-process communication IPC. An example of this is printer spool files that are written by processes, and read by the printer daemon.

OS Security. Operations systems have historically primarily been concerned with providing simple and efficient access to resources. UNIX has been successful partially because it provided abstractions that could be relatively efficiently implemented in the kernel. The focus of UNIX was *not* on strictly controlling how data could flow through the system (i.e. confidentiality), or on the fine-grained control of isolation to produce fault tolerant and secure systems. The liberal use of (discretionary access control)[dac] file-system access control including the set uid bit have enabled privilege escalations that often compromise large amounts of a system's data. The co-location of libraries and already bloated monolithic applications makes compromise expose significant fault propagation so that a fault/compromise can impact significant portions of a system. Put in more exacting terms, such systems do not focus on the Principle of Least Privilege (PoLP). This guiding principle for system design dictates that the full set of resources that a computation has access to should be no greater than that which is required for it to accomplish its goals. A little through should make it obvious how much this does not apply to typical UNIX processes that have access to all of a user's files and the network, regardless if they are all necessary or not.

One approach to increasing the application of the PoLP in UNIX systems is to add enough bandages around existing systems until they provide more desirable guarantees. There has been a long history of this being done successfully. Android uses a user and FS subset to constrain the access of individual applications. Modern browsers separate into separate processes compositing and I/O processes from each of the tabs to increase the use of the PoLP. SeLinux attempts to add labels and information for to Linux to limit the flow of data through the system, thus providing multi-level security (MLS). The act of continually adding bandages is, never-the-less, increasing system complexity, and an up-hill battle.

OS adaptation and efficiency. A newer focus on capability-based systems is on their ability to provide controlled access to low-level resources (i.e. those that are less abstract than those in UNIX). This is useful because when you remove abstractions between resource requests and access, you're cutting down on the code that is running, and removing all of the assumptions that code makes. This has the effect of enabling more efficient and more adaptive access to system resources. For example, in UNIX, mapping and unmapping memory into a process is done through a system of system calls that make a number of assumptions about what the abstraction must provide. Unmapping memory requires that the mappings for that memory be expelled from each (TLB) cache for each core in the system. This operation is immensely expensive, and generally unavoidable. To be clear, many of the guarantees that this provides are quite useful, and often what you want when executing as a process. However, it makes it impossible to implement scalable systems that can make different assumptions about how to manage memory mapping.

Why Use a Capability-based System?

Capability-based systems address the issues raised above of OS security. They are fundamentally implemented around the PoLP and around confinement - the ability to erect information flow barriers between different users/processes. Resource access and sharing must be tightly controlled according to specific, and restricting policies with a variety of desirable properties. This is one of the historical motivations of capability-based systems.

Capability-based systems address the issues around effectively harnessing system resources. They enable the safe and controlled access to resources that are very low-level. The expectation is that high-level abstractions such as processes, and UNIX-style mapping and unmapping are implemented in the system within the **components** via the low-level resource access. In this way, high-level abstractions are built on the those at a lower-level, which is often the foundation for good system design (see Hydra, above).

What is a Capability-based System?

The concept of a capability-based system is simple. Each user-level component accesses resources through a level of indirection. Resource accesses are made by providing a **key**, the possession of which denotes the ability to perform a set of operations on a resource. Keys can be passed (delegated) between components to enable the sharing of resources. For example, the ability to communicate with another component is only allowed through the possession of a key, and if a component passes that key to another, then they both can access the resource through it. Capabilities are these keys.

Capability tables implement the lookup $ct_i(c_i) \rightarrow r_i$ where ct_i is component i.'s capability table, c_i is the capability that component is attempting to access, and r_i is the resource associated with that capability. System calls to the kernel from a component can be viewed as $syscall(ct_i, c_i, o)$ where o is a resource-specific operation (map, invoke, delete) to perform on the resource: $o(ct_i(c_i))$. The kernel's job is simply to implement (a) means for identifying the resource for a given capability, and (b) to implement the resource operations. Most kernels provide some version of (b), the operations to perform on their abstract resources, so the unique aspect of capability-based systems is (a).

Capability-based systems are interesting as they

1. constrain the set of resources that each component has access to, and
2. can be implemented efficiently as a means for controlling access to low-level (thus frequently accessed) resources.

If memory were of no concern, one could imagine each of the capability tables being implemented as a simple array, each entry holding a reference to the associated resource (or `NULL`), a tag denoting which type of resource is referenced, and a bitmap including which operations can be performed on the resource. It is clear that the overhead of looking up a resource in this system is nearly minimal (array indexing). It is less clear how useful how this is to control access to system resources and to provide confinement.

Capability Delegation and Revocation

So how are the capability tables of each component populated? This boils down to two factors:

1. How is a component given access to resources (i.e. how are resources added to a capability table)? We call this capability *delegation*.
2. How is a resource that a component has access to, removed from the component's capability table (i.e. how are resources removed from a capability table)? We call this capability *revocation*.

Delegation. Abstractly, we can view delegation as providing at least $Dc(ct_s, cl_s, c_d)$ which takes a capability c_l in the source capability table ct_s where $ct_s(c_s) = r$, and creates $ct_d(c_d) = r$ in the destination capability table (assuming that before the delegation $ct_d(c_d) = NULL$). Different systems will provide different operations that provide different arguments to this function, but the core is the same.

Observe a few key properties:

1. A component (with capability table ct_i) can only delegate access to resources that it has access to. Thus, there is no *expansion* of resource access through delegation.
2. The destination component will be able to access the resource (via $ct_i(c_i)$) as efficiently as the delegating component.
3. Delegation can be recursive, meaning the destination component can now be the source for a future delegation to another component.

Each capability often also has a *permission set* associated with it. This indicates the set of operations that can be performed on the resource through that component. *Delegation* often also takes an argument p, which delegates those permissions to the destination component. If a capability's permissions are $perm(ct_s, c_s) \rightarrow r$, then $Dc(ct_d, ct_s, c_d, c_s, p)$ is allowed if and only if $p \subseteq perm(ct_s, c_s)$. Resource operations $o(ct_i(c_i))$ are allowed only if $o \in perm(ct_s, c_s)$. All of this has an important effect related to (2) above: a component can only delegate access rights to resources it has itself.

An interesting use of the permissions set is that *delegation itself* can be considered a permission set-controlled operation. For example, only if $D \in perm(ct_s, c_s)$ can a component perform $Dc(ct_d, ct_s, c_d, c_s, p)$. Thus a component can delegate access to a resource to another component, **but** prevent that component from delegating it itself. This enables specific components to be charged with managing the resources. A component wanting access to a resource might have to ask a specific component that mediates all access to the resource.

Revocation. The other side of the coin is how we can remove access to a resource from a component. This is often used for very mundane reasons. If a component is done executing (e.g. `main` returns, or `exit` is called), then we can use revocation to remove resource access, and reclaim resources (e.g. memory) that are only accessed by the terminating component.

The minimal *implementation of revocation* performs the following: $Rc(ct_s, c_l)$ where before R, $ct_s(c_l) \rightarrow r$, and after, $ct_s(c_l) \rightarrow NULL$. The difficult question is who should be allowed to revoke a capability from a component? This is actually a historically difficult question.

How a number of systems handle this:

- Some systems track which components have made a delegation to which other components (a per-resource tree of delegations), and revocation actually revokes all capabilities derived from a capability in the source component. Modern L4 variants and Barrelfish use this approach.
- Some systems use indirection of capability c_l access through another capability c_i. A delegating component maintains access to c_i, and the main operation it can perform is R – to revoke c_j. Eros is the main system that used this. Some versions implemented it with a "epoch" counter per capability and object. Revoking capabilities involves incrementing the objects epoch, and any resource access through a capability must have a matching epoch, or an error is thrown.
- The last option is that capability tables are themselves referenced as kernel resources through capability tables. The capability-based access to a capability table denotes some level of access to it, including added and removing capabilities to it. To maintain the confinement properties, all created capabilities must be a delegation from the component's own capability tables. Put another way, a component can only create capabilities for resources they already have access to. Importantly, this enables revocation on any capability tables you have a capability to. Composite uses this model through what we call "higher-order resource tables".

Implementing an OS as a Capability-based System

The post on the design of capability-based systems contains details on how these systems (and specifically *Composite*) are implemented. A key to understanding how a full operating system could be implemented using capability-based resource access, is understand what the resources are, and how they are composed together to create higher-level abstractions. For example, how are individual capabilities to memory pages composed to provide the expected semantics for `fork`? How are individual thread capabilities composed together with a component to provide system scheduling?

First, the kernel resources are all very low-level. They are close to actual hardware abstractions. This includes pages, virtual address spaces, threads, communication end-points, and interrupts. To build the higher-level abstractions we expect from fully-featured OSes, we simply create components with limited capability tables that give them access to exactly the resources they require (but no more – PoLP). When a component needs more resources, it typically ask for them (via a communication channel accessed through its capability table) from another component whose job is to manage and multiplex some set of resources.

The big question is one of bootstrapping. The initial component that gains control upon boot typically has access to all system resources. It must partition and delegate resources to the system resource managers (i.e. the main scheduler) so that they can continue to build the tower of abstraction.

Acknowledgments

Gregor Peach motivated this post by expressing confusion for the capability design article. Great feedback as I realized there was a huge scaffolding gap to understand the system.

Updates

- Cleaned up the links (thanks Luke Baier!).
- Fixed `md -> html` conversions (thanks Gregor Peach).
- Cleaned up and clarified some areas (thanks Teo Georgiev!).

1. "The separation of mechanism and policy in Hydra" is the foundational paper from which much of **Composite**'s design is derived.↩