Moroso: IPC, Security and Userland

Amanda M. Watson, Carnegie Mellon University

December 8, 2014

1 The Moroso Userland

1.1 Moroso IPC

In the Moroso userland, communication across address spaces is done using IPC (Inter-Process Communication). Programs can send messages to one another, ranging from a few bytes to multiple pages. A message is passed when one thread specifies a destination and a chunk of data to send, and another thread agrees to receive it. The Moroso kernel is direct-mapped, meaning that resources are shared between all programs. However, being a microkernel, many resources that would normally be retrieved in the kernel are instead managed by user programs, making communicating over IPC a key ingredient to system operation.

The operating system is responsible for a) ensuring that user programs are protected against mailicious attackers and b) managing system resources. It follows, then, that IPC not be a point of attack by malicious programs. Historically, many IPC systems have allowed attackers to crash or DOS other users. Moroso aims to prevent this with the following features:

- Privileged, per-process identifiers for sending messages
- Centralized management of server permissions
- Message timeouts and truncation

1.2 Servers and IPC

In userland, many resources and services are managed by *servers*, multithreaded operating system programs that run in userland for the duration of the system, and respond to user requests. For instance, if a program P needed to read a file from the filesystem, it would send a request to the file server, and wait for its reply. Requests are made by sending IPC messages to a server mailbox: Program A, for instance, will send a short message requesting the first 100 bytes of file `foo`, and will block until Server S responds. Server S will call `ipc_recv`, read the request, find file `foo`, and reply to P with a message containing the first 100 bytes of `foo`. If P set a timeout, which was reached before S had the chance to reply, `ipc_send` will return with failure, and S will either never see the message, or be told at the time of reply that the exchange has failed, and the message will not be sent.

1.3 Mailboxes

In Moroso IPC, all messages are sent by sending to a mailbox, and received by waiting on a mailbox. A thread creates a new mailbox by invoking `mailbox_new()`, and while any thread (with a valid descriptor) is allowed to send to the new mailbox, only threads that are members of the process that created the mailbox are allowed to wait on it. We call this creator process the mailbox "owner", and while each mailbox has only one owner, processes can own as many mailboxes as the system permits.

Mailbox Descriptors
Mailbox owners control who is allowed to send to them with mailbox *descriptors*. In order to send to or wait on a particular mailbox, the caller must specify a valid u32 descriptor that corresponds to it. A descriptor is valid for all threads in a process (and invalid for all other processes). Retrieving a descriptor for a particular mailbox can be done in three ways: by creating the mailbox (the descriptor is sent as the return value of `mailbox_new()`), one's parent already possesing a descriptor (parent descriptors are copied to the child on `repl()`), or having the descriptor *forwarded* by a thread that already owns it (see *descriptor tables* below).

Implementation of Descriptors
Descriptors are mapped to mailboxes kernelside using a *descriptor table*, a per-process table that translates a descriptor to the address of its mailbox. When a mailbox is created, its address is inserted into an empty space in the owner's descriptor table, and its key is returned to the user as a descriptor. Syscalls that use mailboxes such as `ipc_send()` and `ipc_recv()` will then use this descriptor to retrieve the corresponding mailbox for use (NOTE: this means that there is no way for a user to communicate to a mailbox except through a descriptor. This is done so that there is no "universal" id for a mailbox, and owners have control over who is allowed to send to it). When a process calls `repl()`, its child receives a copy of its current descriptor table (meaning all descriptors that are valid for the parent are valid for the child). Forwarding a descriptor is done using a special kind of IPC: when a descriptor is received over ipc, the kernel adds an entry to the receiver's descriptor table, and returns its key to the receiver as the new descriptor. This new descriptor will not correspond in any way to the sender's descriptor for the mailbox

Descriptors can be removed from the table by calling `mailbox_remove()`, and moved by calling `mailbox_move()`. Note that all threads are initialized to have descriptor 0 correspond to the Name Server.

Each mailbox has a count of how many times it is referenced by descriptors: once all descriptors to a mailbox have been removed, there is no longer any way for a program to be in communication with the mailbox, and the mailbox is freed. Likewise, there is no way for a process – including the mailbox owner – to destroy a mailbox if other processes still hold descriptor entries for it. `mailbox_remove()` simply removes a processes' ability to access it.

Mailboxes and Servers
A server will have one or more mailboxes associated with it; user programs send requests to servers by communicating with server mailboxes. This means that only programs who own a descriptor for a server mailbox are allowed to make requests to it. Server access is gained by sending a request to the Name Server (see below).

1.4 The Name Server

So far, the only server integral to system function is the Name Server. The Name Server is brought up during system init, and acts as a trusted public directory for servers in the system. A server can opt to register a mailbox with the Name Server: this operation forwards a descriptor to the Name Server with a unique string id; the Name Server then creates an entry for the descriptor (probably including information such as permissions data, though no such features are currently implemented) in a userland table, with the id as its key. Then, when a program wishes to send to the new server, it sends a request to the Name Server with the string id originally sent on registration. The Name Server looks up the entry for the ID and (if the requesting thread has valid permissions and such), forwards the server's descriptor to the requestor.

Note that, without a permissions system, any mailbox can register with the Name Server with any id that has not already been registered, meaning that a user program could theoretically "pose" as an OS service that isn't launched on boot. This is valid behavior within the current security model. A more refined permissions system would allow a class of privileged (trusted) servers, and grant descriptors only to requesting programs that met a certain criteria. Since mappings between mailboxes and descriptors are opaque to the user, the

Name Server can also respond to an untrusted request by forwarding the descriptor of a different mailbox, such as a Proxy Server that will screen and sanitize requests before sending them to their target.

1.5 IPC and Security

As the EROS IPC paper points out, a naive design can allow a malicious client to bring a server to a standstill. Implementations in previous systems have suffered from the following vulnerabilities:

Callee Protection When invoking threads are able to send directly to recipient threads with no form of protection or authentication, the system is unable to place restrictions on requests. Even if the server thread is made responsible, for determining whether a request is valid, it has no way of refusing to acknowledge a malicious program's messages.

Moroso IPC solves this in two ways. First, rather than send directly to a thread, `ipc_send` takes a mailbox descriptor. A thread only has access to a mailbox if it is *forwarded* a descriptor for it by a program that already has access. While this can be done by the mailbox's owner or any thread the owner has forwarded the descriptor to, access will generally be granted by the Name Server, which holds descriptors for all operating servers and handles message permissions. If the Name Server approves a request, it invokes the `forward` syscall to forward a descriptor for the mailbox to the requesting thread (an entry is added to the requesting thread's descriptor table, and the new descriptor is returned). By restricting server access to threads that have been granted access by a trusted party, we ensure that the system can arbitrate who is allowed to send, and untrusted, malicious programs can be denied access to resources.

Second, rather have one mailbox per server, a single server can have several. They can differ in priority and permission level. For instance, a less trusted client may only be granted access to a low-priority mailbox for the server, meaning that a denial-of-service attack by untrusted parties will not bring the entire server server to a standstill.

Denying Replies A key denial of service attack outlined in the EROS paper occurs when a malicious client makes itself unable to receive replies, either through timeouts or failing to receive again after it sends its payload. The result is the server, attempting to reply to the client, blocking until it times out. Moroso IPC solves this by having synchronous sends block until the reply occurs; this way, the client has no choice but to wait, ready to receive, until the server replies. In the case that `ipc_send` times out between the server receiving the message and sending a reply, it signals this to the mailbox, and `ipc_reply` will return immediately with error.

Although asynchronous messages do exist in the system, they do not require replies; thus, the server's behavior when replying to a client that may be malicious is up to the discretion of the server; as the server is in control of how the client is handled, rather than being at the mercy of the client's action, this seems like a success against the given case.

Invalid Transfers While perhaps less of a problem, it's noted that some IPC systems do not safely handle message transfers when the receiver does not have enough room to take the send: either the server trusts the client to hand them a payload they're able to receive, or it silently truncates the message, and the client has no way of getting the information across. Moroso IPC solves this problem by truncating the message to the receiver's buffer length and returning the length of the buffer that was copied to the receiver. Futhermore, checks made by the kernel on both ends ensure that the memory being sent is not (and cannot become) invalid, meaning that a malicious client is unable to cause a server to fault by handing them memory that no longer points to a valid frame. During the transfer, the kernel grabs the sender's VM lock (ensuring pages cannot be unmapped by the user), and truncates the transfer when an address is not backed up by a valid page.