Securing Inter-process Communications in SELinux

Spencer Shimko, Joshua Brindle
Tresys Technology, LLC

Abstract

In the modern computing world, a secure system is best implemented with mandatory access control (MAC) mechanisms. One aspect of secure system design is the careful definition of information flows between processes - inter-process communications (IPC). System designers, when weighing the security risks and functionality of different types of IPC, have had to rely on intuition and experience because of the lack of documentation regarding the security properties of the IPC mechanisms. This paper explores the security functionality that the underlying operating system must support to facilitate secure communication between processes in a Linux operating system. Security Enhanced Linux (SELinux) provides the MAC mechanisms used to support and ensure secure communication between processes, as is illustrated through the example secure IPC mechanism presented in this paper. This example is achieved through a combination of SELinux policy and traditional Linux IPC mechanisms, and presents the best combination of security and throughput available in SELinux.

1. Introduction

Inter-process communication (IPC) refers to the exchange of information between processes on a single system or over a network. Many methods exist for communicating between processes on the Linux operating system. The various forms of IPC vary by purpose, such as for synchronization or for passing data and each type of IPC has its merits as well as its drawbacks, typically described in terms of bandwidth and latency. The merits and drawbacks have been tested through benchmarks numerous times and are well documented [Stevens] [Perf]. However, the security of IPC mechanisms is seldom addressed.

The goal of this paper is to discuss the access controls over various forms of IPC in Security Enhanced Linux (SELinux) and describe how those access controls can be used to define information flow in a system. This discussion will address revocation as well as integrity. Information flows that are hidden or often overlooked will be highlighted. The paper begins with a general discussion of the security requirements that must be met by any access control system infrastructure to facilitate secure IPC. Next, selected forms of IPC are analyzed, highlighting the access controls that SELinux provides for each.[1] Finally, a combination of IPC mechanisms that leverages two forms of IPC to create a high-bandwidth, unidirectional, secure channel is presented to illustrate how SELinux policy and Linux IPC mechanisms can work together to provide secure communication. Unidirectionality is the primary goal of this research and the accompanying proof of concept implementation. While a truly unidirectional IPC mechanism is possible it isn't pragmatic since the receiving process generally needs some sort of notification to the sender that more data can be sent. Both of these concerns are addressed in this paper and the implementation.

2. Security Objectives for IPC

The objectives of a security system with respect to IPC are to provide access control on the IPC system objects (sockets, pipes, etc.), to guarantee the desired direction of information flow (bi- or unidirectional) in every instance, to provide for revocation of access, and to protect the integrity of the subjects (processes) and objects (data). All of these objectives need to be considered when designing a system, and must be weighed against both the functional requirements and the risks and threat model of the intended operational environment.

2.1 Fine-Grained Access Control

Subsets of the IPC mechanisms in use today provide certain inherent levels of access control. For the most part, these access controls are discretionary in nature. The general weaknesses of discretionary access controls are understood, so details will not be presented here [Inev].

[1] Although numerous forms of IPC were evaluated, this paper only presents the most common types.

SELinux offers type enforcement (TE), a MAC mechanism, implemented in a modern operating system that offers fine-grained access control over many types of IPC [SELinux]. This allows security administrators to define policy according to their individual security goals. SELinux supports POSIX, SysV, and BSD IPC mechanisms and techniques. Depending on the way in which each of these techniques is implemented,[2] certain types of IPC lend themselves to fine-grained access control more so than others. Fine-grained access control over IPC mechanisms is defined as the ability to control the information flow between two processes using the IPC mechanism.

2.2 Revocation

Even if a security system provides fine-grained access control, revocation of access must be addressed. Revocation is the ability to rescind access to a subject or resource when the policy changes or the entity is relabeled. SELinux supports access time revocation of many forms of IPC, but not all. This can be attributed to the architecture of the Linux kernel and is unlikely to change. Specifically access to memory based IPC, such as mmap()'d files and shared memory segments, cannot be revoked due to the distributed control of memory within the Linux kernel.

2.3 Unidirectional Information Flow

The flexible set of access controls over various forms of IPC provided by SELinux allows one to implement numerous types of policies to achieve various security goals. Sometimes these goals require bi-directional communication between processes. This is well understood as evidenced by the current policies in use today [Refpol]. As a result, this paper will not specifically focus on bi-directional IPC; instead it will focus unidirectional information flow.

Using fine-grained access control, a truly flexible secure IPC system is capable of supporting unidirectional information flow, i.e. by only permitting data to flow in one direction through a pipeline. Unidirectional IPC has been used in cross-domain solutions to provide an assured, non-bypassable pipeline [MacMillan]. Unidirectional IPC is also valuable in other implementations, as evidenced by the prevalent use of pipes (FIFOs) and UNIX datagram sockets. Unfortunately, true unidirectional information flow is difficult to achieve. Unidirectionality ensures that one process can pass data to another without the possibility of the second process receiving data back from it – a backchannel.

For example, consider any IPC mechanism that facilitates unidirectional information flow from process A to process B. It is nearly impossible to have a zero bandwidth back channel, since any implementation has to provide hard guarantees that information has passed from A to B. Even in the simplistic case of UNIX datagram sockets, where A is only permitted to send data and B is only permitted to receive data, B's socket buffer might be full, which would result in a blocking condition. The blocking action itself is a form of back channel. The block status can either be "blocked" or "not blocked," a binary condition. By combining this status bit with a few other things, such as the passage of a specific amount of time, it is possible to transmit data backwards through the pipeline.

This paper describes the IPC mechanisms that are capable of providing a full-bandwidth forward channel while minimizing the bandwidth of the back channel. It does not explicitly address covert channels or channels not intended to transmit any information. However, limiting the bandwidth of overt back channels can be achieved in many ways in an SELinux environment. Each method has its benefits and drawbacks, just like the IPC mechanisms themselves.

2.4. Integrity

Traditional confidentiality models, such as Bell-LaPadula found in Trusted Solaris, do not address integrity [TSol][TCSEC]. The Biba model applies a hierarchical model inverse to Bell-LaPadula to ensure integrity and has the same issues with coarseness and inflexibility that Bell-LaPadula does for confidentiality [Biba]. On the other hand, TE is capable of maintaining fine-grained object and process integrity. An access matrix defines the exact relationship between all subjects and all objects. The kernel enforces this access matrix ensuring that the trusted binary cannot be altered through malicious means.

By ensuring the integrity of the processes acting on data, and restricting data access to specific processes, the TE mechanism supports data integrity. TE cannot ensure that the process is modifying the data correctly, but it can ensure that only the permitted proc-

[2] Typically, implementation differences appear when comparing those supported in userspace to those with true in-kernel support.

esses are making any data modifications. Unidirectionality and fine-grained access controls provide mechanisms to support data integrity. Integrity is ensured by policy since process A will only be able to send data to process B, and not other processes. Integrity is provided by SELinux and type enforcement regardless of the type and directionality of the IPC chosen.

3. Security Attributes of Common IPC

There are numerous forms of IPC available in SELinux. Some of the most common include pipes, UNIX domain sockets, shared memory, and message queues. The following sections elaborate on the security benefits and drawbacks to each of these mechanisms.

There are several types of IPC that will not be addressed in detail in this document. These include files, signals, memory mapped files and semaphores. Research into these forms of IPC is continuing and will be presented in a later work.

3.1. Pipes

Pipes, or FIFOs, provide an easy way to send data between processes using the familiar read and write operations. SELinux access controls can ensure unidirectionality only to a certain point; there are certain ways the pipeline can be manipulated that SELinux cannot control without breaking the long-used semantics of FIFOs or possibly the API itself.

The directionality of a FIFO is determined by the applications. When a FIFO is created there are two file descriptors, one for read and one for write, but a process can use either file descriptor to read and write the pipe. Linux only supports unidirectional pipes, but it is arbitrary which process is the reader and which process is the writer. In practice, bidirectional flow is usually achieved by creating two half-duplex FIFOs: one for process A to write to and process B to read from, and the second for process B to write to and process A to read from.

Since all FIFOs have the possibility of being used by a single process for reading *or* writing, it is possible for an exploit to reverse the direction of the pipeline. A pipeline originally intended to send data from A to B could then be used to send from B to A. SELinux policy can enforce the directionality of any FIFO, thereby ensuring proper directionality of the pipeline and eliminating this attack vector.

Even if SELinux policy is used to ensure the directionality of a FIFO, there is still the possibility of a timing back channel. If the FIFO uses the default blocking mode, a write operation will block until it succeeds. This blocking operation can be combined with the passing of a specific amount of time and thus used to transmit data through a limited bandwidth back channel. SELinux cannot reasonably control this aspect of FIFOs without breaking API compatibility.

The write system call provides another back channel for non-blocking FIFOs. The write system call is implemented in such a way that it writes as many bytes as possible and returns the number of bytes successfully written. Using this knowledge, it would be possible for B to manipulate the size of the buffer by only reading a certain amount of data, thereby causing a value to be transmitted back to A via the write call. This channel could transfer large amounts of data even with SELinux enforcing unidirectionality on the data pipeline.

Pipes provide a reasonably unidirectional IPC when SELinux policy is used and only blocking FIFO’s are allowed, unfortunately SELinux policy cannot differentiate blocking and non-blocking FIFO’s

3.2. UNIX Domain Sockets

UNIX domain sockets are very simple; each application participating in the IPC creates an endpoint, or a socket, of one of two types. The two types are stream, or session-oriented sockets, and datagram, or sessionless sockets. There are no revocation issues with domain sockets and the permissions do not migrate. Migration occurs when the enforcement of a security decision moves from the security sever to another location. Any changes in policy that alter the permissions on the socket files or the socket pipeline will be enforced at the next access.

UNIX stream sockets, like all session-oriented protocols that provide reliable delivery of data, must be bidirectional. While SELinux access controls are sufficiently fine-grained to enforce unidirectionality, it would render the socket connection unusable. As such, there are very few limits on the bandwidth of the back channel introduced - since each participant will need read/write access to the socket, process B could send arbitrary amounts of data back to A, not just the data used to ensure delivery. For unidirectional information flow, stream sockets should be used with care, or not at all.

Datagram sockets are sessionless, yet, unlike user datagram protocol (UDP) connections, reliability is still ensured by the kernel. The back channels presented through the use of datagram sockets are limited and difficult to exploit; however, they still exist.

Unlike other forms of IPC, such as FIFOs, there are no “partial writes” when the socket is configured to be non-blocking. The message is either completely sent when it can fit in the buffer, or an error code is returned if it cannot fit in the available space. Similar to the case of blocking FIFOs, error codes can be combined with the passage of time to transmit information.

The use of UNIX domain sockets as a form of IPC is widespread due to their history, convenience, and flexibility. There are drawbacks to using this form of IPC for secure communications including the socket files themselves and the back channel resulting from the blocking of sockets. Stream or session-oriented sockets present additional security complexities that must be included in the evaluation of the architecture. The back channels described above are difficult to exploit, are low bandwidth aside from the file back channels, and, depending on the threat model, might not present a significant attack vector.

3.3. Shared Memory

Most forms of IPC require the kernel to transfer data through kernel space, which adds overhead and latency. Shared memory has the benefit of not having to go through kernel space - it can be shared between processes entirely in user space.

Access controls on shared memory in SELinux are fine-grained but non-revocable. A process must have permission to create the shared memory and then permission to attach it to the process’ address space. When the segment is attached, another access check is performed based on the type of mapping: read or write. While any process allowed to write to a shared memory segment must also be allowed to read the segment, a process can be granted read only access. Thus a unidirectional pipeline can still be achieved: process A is given read and write access, but process B is restricted to read access.

Some form of back channel must be introduced if the memory segment itself is to be unidirectional. Process A and process B usually use a bit in the memory as a means of synchronizing the access. This requires process B to be able to write to the segment. If process B requires write permission on the segment, a full bandwidth back channel is created.

If the system design does not require revocation at a particular point in the pipeline, i.e. if shutting down the network interfaces is sufficient, shared memory might be an acceptable choice. A control channel that has a limited back channel bandwidth is presented later in this paper.

3.4. Message Queues

Message queues are a means of IPC often used to send control information, as opposed to large data sets. They are priority based (or can be treated as such) linked lists used to send records between processes. They are persistent across process termination due to the manner in which the kernel stores them. This means that one process can create the message and terminate; a second process can start and send a message then terminate; and finally a third process can start, read the message, and finally destroy the message queue. This is a complex case but illustrates a possible way to communicate using message queues.

The SELinux access controls on SysV message queues are sufficient to eliminate all but a very limited bandwidth back channel that is hard to exploit in a meaningful way. Access controls exist for creating the queue, querying and setting attributes, destroying the message queue, and sending and receiving messages. The back channel results from the blocking on the send. As has been shown earlier, the act of blocking when combined with careful manipulation of timing could be used to transmit data through this back channel. However, this is a difficult back channel to use due to the noise introduced into any timing back channel and the limited bandwidth of a blocking state at the outset.

The message queue structure carries information about the last time a message was read and what process ID (PID) read the message. This could feasibly represent a back channel, but as long as process A cannot read the attributes of the message queue this back channel is prevented. SELinux can prevent process A from reading the attributes and thus eliminate this possible back channel.

If multiple queues were created - say nine: one to represent each bit in a byte and one to signal the start of a new byte - and the length of the queues was set to one, process B could remove the message from the queue indicating the bits set to one in a byte of data. B could then pop the message off the queue used to signal the end of a byte. A would be able to determine the byte transmitted through this back channel. Since each queue would require a size of one for successful transfer, this channel's throughput is more limited than that of the forward channel.

Although the last back channel may not be optimal, it becomes more relevant because message queues are persistent across the termination of processes. A helper process, process H, could be used to create the message queue with a specific label. SELinux policy can prevent processes A and B from creating message queues. This means that process A would not be able to arbitrarily create message queues to create a bit-field. This is an effective means to eliminate the back channel just described.

Given the fine-grained access controls and the limited bandwidth of the back channels, message queues represent a useful IPC mechanism to use for passing control data between processes. Due to their size limitations and the passage of the messages from userspace, through the kernel, and back into userspace, they provide little opportunity for transferring large amounts of data.

4. Constructing Unidirectional IPC

The previous section addressed security and performance considerations of a number of IPC mechanisms. In this section, shared memory will be combined with message queues to create a high bandwidth forward channel with a very limited back channel.
Message queues can be used as a side channel to facilitate unidirectional high bandwidth transfers of data from process A to process B. Using two forms of IPC introduces complexity into the application, complexity that the developer must handle, and possibly introduces another back channel. SELinux policy can be developed to ensure that the shared memory is unidirectional between two and only two processes, leaving a single low-bandwidth back channel. A library is being developed to address application complexity. This library will provide all the functionality required while including an intuitive interface for the developer that is no more complicated than the API for the traditional IPC mechanisms described earlier. The functionality that must be provided by any secure IPC library for creating secure IPC from an insecure base is described below.

To use message queues securely, the `msgget(2)` function should be called from a helper process with the `IPC_CREAT` flag indicating that the helper process is creating the message queue. A helper application is required to eliminate the possibility of compromised senders and receivers creating multiple queues for use as a bitmap. SELinux policy will ensure that this application is permitted to create the message queue, but not use it for any other operation. The pertinent parts of policy to support this helper application are shown in Figure 1 - Helper.

After the message queue is created, the application sending the data, process A, will associate with the message queue and set the queue's size to contain the necessary control information. The pipeline described below only requires the queue size be limited to the size of a single message buffer. SELinux policy can prevent the process from creating the message queue while permitting it to associate and subsequently use it for data transfer. This reduces the possibility of an attacker creating message queues for use as a bit-map to transmit data through a back channel leveraging error conditions. Similarly, process B, the receiving application, should associate with the message queue. SELinux policy will prevent process B from setting any attributes on the message queue thus ensuring the attributes themselves cannot be used for unintended data transfer. Once both applications have associated, the control channel is ready for use.

The message queue that will be used for control data transfer has been created at this point, but the efficient data channel has not. Process A creates the shared memory segment and attaches the segment to its own address space. Relying on process A to create the segment eliminates the back channel introduced through the `EEXIST` error condition; furthermore, SELinux policy can enforce this principle. Process B attempts to associate and attach the shared memory segment in a read only mode. This creates the high-throughput data channel. After attaching the segment to its address space process B will call `msgrcv(2)` and immediately block.

After the data channel and the control channel have been created, the infrastructure is in place to support IPC capable of providing high-throughput forward data transfer while minimizing the bandwidth of the back channel. Process A begins to fill the data segment and process B waits for a message from process A indicating the shared memory is ready for reading.

When process A completes the task of writing data to the shared memory segment, it must tell process B that the segment contains data and is ready for reading. Process A places a `DATA_READY` message on the queue. It immediately attempts to place a second message, a `NOOP`, on the queue but the `msgsnd` call blocks.

While process A has been processing the data and filling the segment, process B has been blocked after calling `msgrcv,` waiting for a message from A. As soon as A places the first message on the queue, the call to the `msgrcv` returns, B receives the `DATA_READY` message and begins processing the shared memory. By B receiving, or “popping”, the message from the queue, A’s previous attempt to place a second message, `NOOP-1`, on the queue is unblocked and succeeds.

At first glance, it seems logical that only `NOOP-1` is needed. A second No Op seems redundant. However, since A’s attempt to place the first No Op on the queue has succeeded, there must be a way to prevent A from starting the process over again, which occurs later. From a technical standpoint, the second No Op exists to prevent A from exiting its data send function prematurely, creating a race condition. If it exits the data send function before B has completed its processing of the shared memory, process A would begin to send the next segment. The second No Op prevents this race and inevitable data corruption.

Once B has finished processing the data, it reads the second message, `NOOP-1`, on the queue. As soon as `NOOP-1` is read process A’s attempt to write `NOOP-2` is unblocked. Process A places `NOOP-2` on the queue, which process B immediately pops. Process A resets the shared memory as soon as its attempt to place `NOOP-2` on the queue succeeds (`msgsnd` returns).

At this point, the process repeats. Process B blocks, waiting to receive the DATA_READY message, while process A begins refilling the shared memory segment with data.

Policies to support the receiver (or reader) and the sender (or writer) are shown in Figure 2 - Shared Memory Reader and Figure 3 - Shared Memory Writer, respectively. These policies are not complete but show the parts necessary to define the pipeline and support near unidirectional data flow with minimal back channel.

This implementation provides a good combination of security and throughput available in SELinux. Shared memory, a high-throughput, traditionally insecure IPC mechanism due to its lack of a limited back channel for control has been combined with another form of IPC used for process control to create a more secure information flow pipeline while retaining most of the efficiency of the original mechanism.

5. Summary

Exploration into the various forms of IPC in SELinux has revealed numerous information flows between processes. Most of these information flows are part of the IPC mechanism themselves, the forward channel. In addition, there are back channels typically used to transfer data related to the control of the forward channel. An optimal solution would provide IPC that is high bandwidth with no back channels.

These back channels do not preclude the use of any of the mechanisms described here. Rather, the system architect must take the back channels into account and strike a balance between security and functionality. A similar balance is shown in the development of a combination of IPC mechanisms that possesses a high bandwidth forward channel with a low bandwidth back channel.

A proof of concept of the implementation outlined in section 4 has been written and is available at http://oss.tresys.com. This library provides an API to create unidirectional IPC between processes and SELinux policy to ensure unidirectionality.

In the proof of concept implementation IPC keys are determined at application build time so that there is no need for a heavy and bidirectional synchronization protocol.

References

[Biba] Biba, K.J. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[Bishop] Bishop, Matt. Computer Security: Art and Science. Addison Wesley, 2003.

[Drepper] Drepper, Ulrich and Molnar, Ingo. The Native POSIX Thread Library for Linux. February 2005.

[Inev] Smalley, Stephen et al. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. October 1998.

[MacMillan] MacMillan, Karl et al. Lessons Learned Developing Cross-Domain Solutions on SELinux. Proceedings from the 2006 SELinux Symposium.

[Perf] Immich, P. K., Bhagavatula, R. S., and Pendse, R. 2003. Performance analysis of five Inter-process communication mechanisms across UNIX operating systems. J. Syst. Softw. 68, 1 (Oct. 2003), 27-43.

[Refpol] Pebenito, Chris J. "Reference Policy." Open Source Software. Tresys Technology. 20 Oct. 2006 http://oss.tresys.com/

[SELinux] Peter Loscocco, NSA, Stephen Smalley, NAI Labs. Integrating Flexible Support for Security Policies into the Linux Operating System. FREENIX Track: 2001 USENIX Annual Technical Conference. June 2001.

[Stevens] Stevens, W. Richard. UNIX Network Programming Volume 2: Inter-process Communications. Prentice Hall, 1999.

[Stevens2] Stevens, W. Richard. Advanced Programming in the UNIX Environment. Addison Wesley, 1992.

[TCSEC] DOD 5200.28-STD. Department of Defense Trusted Computer System Evaluation Criteria, December 1985.

[TSol] Sun Microsystems. Trusted Solaris Developer's Guide, Technical Document 805-8031-10, pp. 223-286, August 1998.

Policy Examples

```
# the creator only needs to be able to create and insert in a queue
allow shm_creator_t self:msgq  { create enqueue };
```

Figure 1 - Helper

```
# the reader needs to receive messages from the creators queue
allow shm_reader_t shm_creator_t:msg receive;
allow shm_reader_t shm_creator_t:msgq { associate read unix_read };

# the reader gets permissions to read the senders shm
allow shm_reader_t shm_sender_tmpfs_t:file read;
allow shm_reader_t shm_sender_t:fd use;
allow shm_reader_t shm_sender_t:shm { associate read unix_read };
allow shm_reader_t shm_sender_tmpfs_t:file read;
```

Figure 2 - Shared Memory Reader

```
# send messages through the creators queue
allow shm_writer_t shm_creator_t:msg send;
allow shm_writer_t shm_creator_t:msgq { associate unix_write write
      getattr setattr unix_read };

# write to its own shared memory segment (requires read as well)
allow shm_writer_t shm_writer_tmpfs_t:file { read write };
allow shm_writer_t self:shm { create read unix_read unix_write write
      };
```

Figure 3 - Shared Memory Writer