A Pipeline Development Toolkit in Support of Secure Information Flow Goals

[Extended Abstract] *

Philip Tricca
Air Force Research Laboratory, RIEB
525 Brooks Rd.
Rome, New York 13441
philip.tricca@rl.af.mil

ABSTRACT
The rapid spread of viruses, worms and undesired software in commercial and private networks has been a driving factor behind computer security research. The subject of preventing rogue software and rogue network traffic has been the focus of a number of research areas, most significantly through monitoring with intrusion detection systems (IDS).

These reactive systems speak to our inability as software designers to specify the information flows inherent in our systems. This lack of specification results in computer defenders deploying monitoring technologies to detect and react to known illicit software behavior and network traffic using blacklisting techniques. An opposing approach known as whitelisting is known to be more secure. Network managers are unable however to enumerate all permissible information flows in their systems in order to establish a complete whitelist.

In this work we describe a novel software engineering approach to the design, implementation and management of assured software pipeline systems [2] with the specific goal of specifying and confining information flows. We aim to make information flow goals a first class component of system design, addressing and preserving them throughout development. Further we consider the expansion of our work into policy analysis techniques using established model checking and graph analysis techniques.

Categories and Subject Descriptors
C.2.0 [Computer-Communication Networks]: General—security and protection; D.2.1 [Software Engineering]: Requirements/Specifications—methodologies, tools

General Terms
Design, Management, Security

Keywords

1. INTRODUCTION
The Type Enforcement (TE) policy language and its enforcing reference monitor in the Linux kernel (SELinux) have reached maturity [1]. Both government and private sector partners have spent a significant amount of effort specifying policy to confine the standard Linux environment. As a result, products have been developed and certified for use in both the commercial and government sectors using these technologies as a foundation.

Software development models are largely informal and lack TE policy development at their core. The bulk of the policy development efforts done publicly to date are for preexisting software systems. When confining a preexisting system, policy is often derived from the observed behavior of the target software with the goal of allowing the software to perform its task.

This approach to policy development has been termed learning mode, and produces policy in a style said to provide status-quo encapsulation [4]. The largest problem with this approach is that it presents a simple enumeration of access attempts (including networked) made by the target software. For heavily networked applications exposed to the Internet this style of policy development can result in a policy that has been effected by external influence. While the SELinux community has been one of the biggest critics of this development method, they still supply tools that support learning model policy development (the audit2allow tool).

In this paper we describe a software engineering methodology for the construction of pipelines that addresses these concerns. This method begins with the high-level specification of the desired information flows in the software system (pipeline specification). Our current efforts aim to derive TE policy from a pipeline specification as well as use the specification as a basis for pipeline management and network whitelist generation.

The remainder of this white paper is structured as follows: Background work performed at the Air Force Research Lab relevant to this effort is first discussed in §2. Our approach is
was deployed on standard desktop quality computers and worked nodes enforcing a consistent TE policy. Our pipeline demonstrate the streaming of video data through three net-
were able to achieve our functional goals. We were able as possible.
ble of distribution across multiple systems as transparently
3. to implement these pipelines so that they would be capa-
1. To produce a simple video pipeline confined by TE pol-
on MAC policy enforcing systems. Our goals were as follows:
Little can be proven about the origin of a media stream.
load associated with enumerating all information flows in a
IDS systems continue to evolve into intrusion prevention systems (IPS). These are much like IDSs but with additional risk awareness and adaptive behavior. While more advanced than IDSs, IPSs still suffer from the same fundamental prob-
Mandatory access control (MAC) operating systems like SELinux that enforce policy in the Type Enforcement lan-
guage offer an alternative approach. In this approach, all unknown information flows (including network) are denied by default while flows that are known to be good (explic-
itly allowed by policy) are permitted. When contrasted to the blacklisting approach taken by IDS systems this default deny model is referred to as whitelisting.
Whitelisting approaches are known to be more secure but are employed far less often due to the complexity and work load associated with enumerating all information flows in a system. This problem is only compounded when we consider how disruptive the migration of an existing network to a white list network flow model can be.

2. BACKGROUND
Like the shortcomings of anti-virus systems which are based on virus signature detection, IDS systems suffer from two distinct problems. First and foremost: IDS systems only detect intrusions after they have succeeded. If an IDS de-
tects malicious traffic within a network, the traffic is the result of a compromised host on the network or was gener-
ated outside the network and permitted entry by a boundary device. Upon detected intrusions or intrusion attempts an IDS will perform some task (reporting typically) and can thus be classified as reactive.
Second is the freshness problem. IDS systems observe traffic patterns comparing them to a “blacklist” of disallowed or suspicious traffic. Intrusions here are detected through the identification of known bad patterns. In the case where an attack uses a unique traffic pattern or the IDS system is not fully up to date, it will likely fail at its task.
This blacklist approach is rooted in the reality that in most networks, not all valid information flows can be enu-
merated. To allow day-to-day operations to continue, we typically permit most, if not all information flows while at-
tempting to detect known bad patterns and react to them.
IDS systems continue to evolve into intrusion prevention systems (IPS). These are much like IDSs but with additional risk awareness and adaptive behavior. While more advanced than IDSs, IPSs still suffer from the same fundamental prob-
lems.
Mandatory access control (MAC) operating systems like SELinux that enforce policy in the Type Enforcement lan-
guage offer an alternative approach. In this approach, all unknown information flows (including network) are denied by default while flows that are known to be good (explic-
itly allowed by policy) are permitted. When contrasted to the blacklisting approach taken by IDS systems this default deny model is referred to as whitelisting.
Whitelisting approaches are known to be more secure but are employed far less often due to the complexity and work load associated with enumerating all information flows in a system. This problem is only compounded when we consider how disruptive the migration of an existing network to a white list network flow model can be.

2.1 Previous Work
This work began with the observation that current de-
ployed media streaming technology is relatively insecure. Little can be proven about the origin of a media stream. We initiated a prototyping effort to deploy media pipelines on MAC policy enforcing systems. Our goals were as follows:
1. To produce a simple video pipeline confined by TE policy with one-way information flow semantics, 2. to develop a simple ad-hoc policy analysis to support our stated information flow goals for each down-stream pipeline element, and 3. to implement these pipelines so that they would be capa-
ble of distribution across multiple systems as transparently as possible.
This effort produced a successful prototype in which we were able to achieve our functional goals. We were able demonstrate the streaming of video data through three net-
worked nodes enforcing a consistent TE policy. Our pipeline was deployed on standard desktop quality computers and each node was able to sustain enough throughput to trans-
port high definition video data at the required frame-rates.
Our information flow goals were also achieved. One-way mechanisms for communication between both OS processes and networked systems are available, if still beta quality. We were successful in developing policy with one-way semantics to confine our pipeline. This policy is also small enough to support ad-hoc analysis back to the source type (the video device).
At the conclusion of this work we were confident that the MAC system offered in SELinux could support the de-
sired information flow properties for a significantly complex pipeline system. We also had developed an appreciation for the work load associated with manual policy development. At this point we began efforts to add some structure to our pipeline development process. We now seek to unify informa-
tion flow specification with pipeline element arrangement at design time as well as provide post-generation analysis techniques to provide assurances that the resultant policy is intact correct and worthy of trust.

3. APPROACH
We begin our next phase by formalizing our pipeline model. Software systems have been using pipeline processing (also termed the pipes and filters design pattern) as far back as the earliest Unix systems [6]. As we are targeting media pipelines explicitly we look to modern architectures for con-
sistent language and relevant work.
We chose to base our work on the model from GSTreamer. GSTreamer is the media framework native to the Linux Gnome desktop. It has seen increasing adoption in open source applic-
ations and has a well documented model. The GSTreamer community has also produced a number of standard en-
coders, decoders and transforms that we leverage allowing us to focus on our specific goals as opposed to the complex task of codec development.

3.1 Pipeline Nomenclature
We adopt the nomenclature used to describe our pipelines from GSTreamer, but note some specific differences. Pipeline elements fall into several categories, some transforming data formats, others combining or separating data streams from composite formats. Specifically we define elements with only data inputs as sink elements. Sink elements may (and usu-
ally do) have outputs but these are invisible to the pipeline. They are typically data presentation end points (screen out-
put) or archival end points (disk output). Similarly there are source elements that have only outputs. These are where data enters the pipeline and may be wrappers for devices like a video camera, microphone or even text streams like news tickers.
GSTreamer pipelines exist within the context of an OS process. A source element introduces data into the process, a sink element outputs data from the process. All pipeline elements pass data between themselves within the process address space.
We view pipelines from the perspective of the OS. To gain increased policy granularity we decompose the pipeline into multiple OS processes. We do this because the OS can only enforce TE policy at the granularity of objects within its domain of control. To express a pipeline as a series of TE domains we must break the processing tasks up into multiple OS processes.
The pipeline element nomenclature in the context of our architecture thus deviates from the GStreamer convention. To prevent confusion we describe each portion of a pipeline that resides on the same computer as a pipeline segment. Each segment is composed of potentially many OS processes. A GStreamer pipeline (composed of GStreamer elements) resides within a single process. As such they are light-weight processing components loaded from system libraries and executed as threads. Here we differ in that we construct pipelines from multiple OS processes and call these processes elements. The processes themselves use GStreamer internally and may be composed of multiple GStreamer elements. To prevent confusion we will explicitly refer to GStreamer elements as “GStreamer elements” when they are discussed.

Similar to GStreamer we then consider each element that introduces data into the pipeline segment a source. The sink element then would be an element that allows data to flow out of a pipeline segment. Pipeline segments may be composed of multiple OS processes and as such these processes must be linked in some way. GStreamer gives us the ability to write custom GStreamer elements to achieve this task. GStreamer elements that join two OS processes in a pipeline segment are referred to as connectors. We describe this pictorially in Figure 1.

![Figure 1: Pipeline segment with connectors](image)

### 3.2 Crossing Process Boundaries

To pass data across process boundaries we employ traditional Unix IPC mechanisms. Access to IPC objects is mediated by the kernel and access decisions are dictated by policy. The different IPC mechanisms available in Unix systems range from slow to extremely fast with the slower of them provide structure and synchronized access while the fastest provide neither.

Our previously stated information flow goals, combined with the nature of the data we are passing, require mechanisms as fast as possible with one-way information flows. Synchronization between sender and receiver is also critical from the perspective of the programmer. With shared memory, we get a fast IPC mechanism which preserves the one way semantics of our information flow (one writer), but data synchronization is not inherent.

While attending the SELinux symposium in 2007 we were introduced to a synchronization mechanism on shared memory that preserves one way information flows in TE policy. This technique, which is implemented in the SIPC library is described at length in [7]. We are currently in the process of wrapping the SIPC library for use as a GStreamer element.

### 3.3 Crossing Machine Boundaries

Now that we have a method for breaking our pipeline up into multiple processes on a single host the next logical step is to span multiple hosts. In this section we discuss the mechanisms used, their impact on our information flow goals and the resultant information flow policy.

#### 3.3.1 Network Protocols

When communicating between networked systems the TCP/IP protocol has become ubiquitous. TE policy can be crafted to restrict communication over TCP sockets such that one end is strictly a sender and the other strictly a receiver, thus preserving our one-way information flow goal.

We are then guaranteed this data flow between processes by the OS. Information flow may also be examined at the protocol level however, and the use of TCP becomes less appealing when we do so. Specifically, TCP is a cumulative acknowledgement protocol, one in which packets sent are acknowledged by the receiver. This acknowledgement requires a packet be sent from the receiver back to the sender.

A stringent analysis of the use of TCP will note that even though the processes exchanging data cannot communicate bidirectionally, the TCP stream is inherently a bidirectional channel. This may seem like a pedantic argument (and it is), but it is a correct one none-the-less. Data flows in one direction but state information about the TCP stream is communicated bidirectionally.

We address this concern by using UDP streams exclusively. UDP does not give the same guaranteed delivery as TCP but it does give us a mechanism for strict one way networked communications. The TE policy similarly has unidirectional semantics as in the previous case but we are given the added one way guarantee at the protocol level as well.

We acknowledge that in some data pipelining applications the guaranteed delivery of TCP is necessary. One way information flow semantics at the policy level are achievable using TCP and its use does not invalidate this stated goal. For the purposes of our demonstration system we are able to use UDP as the video codecs employed are resilient to some data loss.

#### 3.3.2 Label Exchange

The information flow mechanisms discussed above are necessary for achieving our stated information flow goals. Little security is guaranteed, however, unless the communicating end points share a consistent view of the security policy. As a result we must assume a consistent policy on all systems hosting segments of our pipeline. Further, SELinux provides integration with the IPsec key exchange mechanism [5] extending type labels across the network at runtime.

Labeled IPsec adds to the policy language the association object which always bears the label of the communicating peer process. This is the only place in the policy for a pipeline element where the type of the processes are exposed. This is a significant improvement in policy confinement over restricting a domain access to specific IP addresses and ports which has known limitations [3]. Keeping with good practice, we expose these types through policy interfaces.

### 3.4 Pipeline Specification & Policy Generation

We are currently implementing a small XML syntax as specification for pipeline element structure. We intend for this specification to contain sufficient information to support two tasks: First, it must support the generation of SELinux file context policy as well as interfaces between ad-
jacent pipeline elements. Second, it must be descriptive to the point that it can be used by a program for pipeline management purposes.

Pipeline elements are specified as XML elements. Policy generation requires each element be given a distinct type. This may be done either through random assignment by the policy generation tool or through user specification. In the interest of preserving an understandable policy we allow the user to specify a type name while the tool validates the name against a best-practice naming convention.

Pipeline elements define child XML elements describing their inputs and outputs. These IPC mechanisms may be created explicitly by the element in which case they will be a child of the element or they may be created by an external domain specifically instantiated to manage the IPC object as is the case with the SIPC library. For this reason an element's input and output may also be an XPath reference to an IPC mechanism exposed by another element in the pipeline or by a supporting domain. We generate TE policy by examining the interfaces described in the XML specification, translating these into policy interface definitions.

4. STATUS

Currently we have developed and demonstrated a significant amount of the machinery necessary to establish a media pipeline spanning multiple networked systems. We have also demonstrated the ability to decompose the pipeline on each system into separate OS processes to allow for varying levels of policy granularity. Our current focus is on finalizing the port of our initial prototype to use GStreamer internally. This task (which initially seemed trivial) is made significantly more complex by the extremely flexible nature of GStreamer and the very rigid nature of our information flow goals.

4.1 Future Direction

Future work plans include significant improvements in our policy generation. Our current approach lacks support for generating policy beyond the interfaces between pipeline elements. Generation of IPsec policy and IKE keying daemon configuration is also desirable and could be foundational for constructing whitelists for networked IDS systems.

We are also planning work with significantly more complex demonstration applications. Our prototypes thus far have been limited to simple point-to-point media streams. We expect our future prototyping efforts to include a pipeline to transport video stream from source device to multiple streaming servers in differing networked domains. Even further in the future we expect to take on more complex applications like VoIP infrastructure. We believe VoIP will provide an interesting case study to test the limitations of both our policy generation as well as the pipelining infrastructure.

5. CONCLUSION

We have discussed a pipeline architecture that supports one-way information flows (enforced by SELinux) of varying granularity across OS processes and networked systems. We’ve discussed the use of the GStreamer media pipelining system which supplies us with codecs and an infrastructure to support our custom machinery for communicating across TE domains. Further, we have presented a method for describing these pipelines in a way that allows for their implementation from a specification. We plan to use the same specification for basic TE and IPsec policy generation as opposed to deriving policy from observed behavior. The end result is a development methodology supporting explicit information flow specification at design time.

6. REFERENCES