Time Division Multiplexing of Network Access by Security Groups in High Performance Computing Environments

Joshua Ferguson

January 24, 2013

Abstract

As the scale of data used in computation increases the utility of High Performance Computing as a Service increases in tandem. Security concerns in such a domain are still being solved using traditional Operating System mechanisms of memory protection and file access privileges [5]. However there exist scenarios where customers demand more intuitive and rigorously verifiable security solutions. This paper presents a robust and application transparent network security solution that is both intuitive and easily verifiable. This paper's main contribution is a mechanism designed to enforce high level time division multiplexing of network access according to security groups. By dividing network access into time windows, interactions between applications over the network can be prevented in an easily verifiable way.

1 Introduction

High Performance Computing (HPC) systems are comprised of numerous individual computing systems networked and administrated together such that they can be used as a single system. Popular examples of these systems include some custom made such as the Cray I and the modern IBM Sequoia [4], though more common are simple Computer Clusters in which Commercial Off The Shelf (COTS) equipment is utilized [5]. Security within these systems is often enforced through traditional Operating System (OS) mechanisms of memory protection and file access privileges [5].

With the increase in scale of data used in computation, and big data problems becoming more common, the utility of High Performance Computing as a Service increases. There exist customers in this domain with sensitive data (government laboratories, national defense contractors, etc.) for which traditional security mechanisms are not satisfactory to all stakeholders. Historically this class of customers devotes entire HPC systems to individual projects. With looming cuts in government discretionary spending there is pressure on these

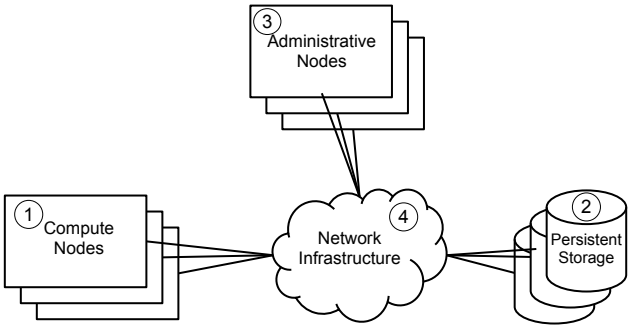

Figure 1: An abstract HPC environment.

organizations to find more cost effective methods for their research without sacrificing security.

This paper presents Time Division Multiplexing (TDM) of network access as one such verifiable security mechanism. By modulating network access between application security groups we can provide an intuitive security mechanism capable of being verified in real-time. Furthermore by implementing this mechanism at the operating system level it becomes transparent to user applications, meaning no modification to existing application code is necessary. [Mention we have built a prototype, and mention formal proof.]

2 Related Work

3 Problem Definition

We begin by defining an abstract HPC environment through which the general case of our security challenge is shown.

3.1 An Abstract HPC Environment

There are four basic resources in most HPC systems represented in Figure 1 as:

1. **compute nodes** - independent computing devices designated to run user submitted applications. These devices are capable of storing temporary data locally. They send and receive data across network infrastructure for three main purposes:*a*) storing or accessing data on the persistent storage devices; *b*) relaying data between other compute nodes working in tandem on the same user application; *c*) and sending or receiving commands (or reports, as the case may be) from the administrative nodes, through which users interact.
2. **persistent storage** - storage devices usually of much larger capacity than the compute nodes are capable of. *a*) ; *b*) .

3. **administrative nodes**
   user accessible front-end or a job scheduler
4. **network infrastructure**

3.2 Security Challenge

The security fear of customers with sensitive data is that a different customer could, through chance or intention, acquire or manipulate their data. There exist three main mediums across which data may be exposed.

4 Design Goals

Here we describe the major goals in the design and implementation of the TDM security mechanism.

4.1 Mechanism and Policy Separation

Envisioning the tool as enabling a more secure form of HPCaaS, the type, number, and scale of jobs assigned to any of the systems can be widely varied. To handle this, at the least, the tool must be capable of receiving and modifying policy at the start of each new job. Furthermore, to provide more fine-grained control an online policy modification scheme is desirable for managing performance tradeoffs between security groups. We define the mechanism and policy split of our tool to be an ability to receive and actuate policy changes at the finest granularity in which the tool operates. [Mechanism Policy split is really along the difference between the formal notion that nodes must be transparently separated from the medium and how it is accomplished (with a queue, at the TCP layer, on ethernet, etc.)]

4.2 Network Fabric Agnostic

Two major technologies are used to network HPC systems: Ethernet and InfiniBand. Any tool for improving security across the broad spectrum of HPC systems must be capable of operating in each. Further, numerous network topologies exist within these technologies; switched fabric and tree structures are the most common for InfiniBand and Ethernet, respectively. For our purposes we define this tool to be network fabric agnostic if the tool is conceptually capable of being implemented in either Ethernet or InfiniBand networks.

4.3 User Application Transparent

A fundamental requirement of the tool is that it be transparent to user applications. Applications written for HPC environments are often quite complex and it is likely that customers would be reluctant to make even minor modifications, especially to programs written in the past that are under re-use. For

our purposes we define user application transparency as the ability to run an application without modification to successful end on an HPC system using our tool, given that it can also do so on a system not using our tool.

4.4 Provably Secure

The final and most important requirement of this tool is that it provably perform its task.

5 Implementation

As a proof of concept we have implemented a version of the tool for the Linux operating system using C++11. In this section we will describe the tool's architecture, operation, and how it adheres to the design goals from the previous section.

5.1 Overview

The tool is composed of four major components: the window scheduler, ingress controller, egress controller, and the state controller. The window scheduler can be located on any administrative node within the system, preferably co-located with the system job scheduler. The remaining controllers are located throughout the HPC environment, with a copy on each compute node that is designated to execute user applications.

The system provides enhanced security by time dividing access to the network according to security groups. As jobs are scheduled on the system, the window scheduler must be informed of the intended location and their assigned security group. As jobs are run on compute nodes throughout the system the window scheduler communicates with the state controller on each node to designate time windows. During any individual time window only one security group has authorization to access the network. For any given time window in which a security group does not have access to the network, outgoing network packets are stored in a local queue while incoming packets are just ignored and deleted. The window scheduler is tasked with alternating time window authorization between security groups.

The following subsections describe each components operations in further detail.

5.2 State Controller

The state controller has three major tasks:

1. Securely send and receive communication with the window scheduler for the system
2. Transit both the ingress and egress controllers between states of network access and denial

3. Collect and store performance data on the egress queue's memory usage

5.3 Ingress Controller

The ingress controller is a firewall of incoming network packets and has two states:

1. Open access of network packets to applications on the node
2. Closed to network packets except for those from explicitly allowed sources (a whitelist style of firewall)

During the open state incoming packets are processed normally. During the ingress closure, incoming packets, except those allowed by the whitelist, are dropped. The whitelist is designated to allow only necessary infrastructure communications such as Network Time Protocol (NTP), performance measurements, and especially packets from the window scheduler.

5.4 Egress Controller

The egress controller, similar to the ingress controller, has two major states:

1. Open flow of packets onto the network
2. Diversion of outgoing packets into a blocked queue

5.5 Window Scheduler

The window scheduler has three major tasks:

Determine system network access states for the next time window Validate the closure of the previous time window Communicate the next time window states to compute nodes To determine the network access states for the next time window the scheduler must run a scheduling algorithm on a few historical inputs. The base case scheduling algorithm is round robin (i.e., equal window size for each security group). To improve performance, a number of heuristics have been considered for the creation of a dynamic priority scheduling algorithm: the egress controller memory usage of compute nodes, number of TCP timeouts, and externally imposed priorities.

6 Case Study

6.1 Performance

6.2 Security Validation

7 Formal Definition

Suppose S is a finite set of security groups, s.t. each security group $s \in S$ is made up of a number of compute nodes. Given S, the language our mechanism

operates on can be generated by a context free grammar. Because the language is dependent on the security groups S, this grammar must be generated based on it. This is done in two steps:

First, we define the base grammar:

$$
\begin{array}{lll}
G^1 = (V^1, \Sigma^1, R^1, \mathcal{A}), \text{ where} & & \\
V^1 = \{\mathcal{A}, W\} & & \text{non-terminal symbols} \\
\Sigma^1 = \{\emptyset\} & & \text{terminal symbols} \\
R^1 = \{ & \mathcal{A} \to \varepsilon, & \text{rules of production} \\
 & \mathcal{A} \to W\mathcal{A}|W\} &
\end{array}
$$

This base grammar, through the non-terminal symbols and production rules, establishes a means of generating the base language form of unordered windows ($W \in V^1$) in an arbitrary length such as WW or $WWWWW$.

Next, we generate the S specific definitions. To do so it is first necessary to define notation for two special terminal symbols and three special sets:

$o_{s,i}$ - an open command issued to node i within security group s,
$a_{s,i}$ - an acknowledgement received from node i within security group s,
θ_s - the set of all $o_{s,i}$ terminals for security group s,
α_s - the set of all $a_{s,i}$ terminals for security group s, and
$\pi(A)$ - the set of all permutations of the set A.

These definitions allow us to define a final, special set:

$$\Lambda_s = \pi(\theta_s) \times \pi(\alpha_s)$$

Intuitively, Λ_s is a set of ordered sets expressing each permutation of the θ_s set matched with each permutation of the α_s set. For example, given a security group s made up of two elements s.t. $s = \{1, 2\}$, Λ_s is defined:

$$
\begin{array}{ll}
\Lambda_s = \{(o_{s,1}, o_{s,2}, a_{s,1}, a_{s,2}), & (o_{s,2}, o_{s,1}, a_{s,1}, a_{s,2}), \\
\quad (o_{s,1}, o_{s,2}, a_{s,2}, a_{s,1}), & (o_{s,2}, o_{s,1}, a_{s,2}, a_{s,1})\}
\end{array}
$$

The sets within Λ_s represent all legitimate command sequences within a window (W) for security group s. The key property of the sets within Λ_s is that each node within the security group is issued an open command, in any order, followed by acknowledgements from each node within the security group, once again in any order.

With these definitions established we can now formally define an S specific grammar:

$$
\begin{array}{l}
G^2 = (V^2, \Sigma^2, R^2, \emptyset), \text{ where} \\
V^2 = \{W\} \\
\Sigma^2 = \{[o_{s,i}, a_{s,i}] : \forall i \in \forall s \in S\} \\
R^2 = \{[W \to \lambda] : \forall \lambda \in \Lambda_s : \forall s \in S\}
\end{array}
$$

These definitions add new terminal symbols and the necessary production rules to generate them. Note the use of Λ_s in the production rules. These rules provide every possible command sequence possible for any window W s.t. every node issued an open command is required to report back with an acknowledgement before continuation onto another window.

Finally, the language our mechanism accepts for security group S can be formed using the union of the previous two grammars:

$G = (V, \Sigma, R, \mathcal{A})$, where
$V = V^1 \cup V^2$
$\Sigma = \Sigma^1 \cup \Sigma^2$
$R = R^1 \cup R^2$

8 Conclusion

8.1 Further Work

Implement dynamic scheduling algorithm on the window scheduler using memory usage statistics from compute nodes.