Portcullis : Secure TCP/IP Port Reservation

Benjamin Ellerby, Raymond Weiming Luo
Evan Ricks, Oliver Smith-Denny
Department of Computer Science, Western Washington University

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus nec nulla eu odio tincidunt tempus. Ut consequat euismod purus, nec aliquet mi porttitor ac. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas et pulvinar dui, sit amet aliquam nisi. Integer tempor, nisl a sodales volutpat, eros dui vehicula purus, quis pellentesque libero tellus id neque. Curabitur aliquam hendrerit dui et sagittis. Sed dictum purus fermentum erat gravida, sed dictum risus finibus. Vestibulum placerat nunc vitae nibh convallis, eget ullamcorper ante sodales. Aliquam eget sem rutrum, eleifend diam vel, fringilla leo. In tempus arcu dolor, quis semper erat iaculis nec.

keywords: secure port reservation, TCP/IP, computer networking

1. INTRODUCTION

As computer network services have exploded over the past decades, the fundamental technology for computer networking ports has stayed unchanged. The current implementation of a Transmission Control Protocol and Internet Protocol (TCP/IP) port is ambiguous and does not apply any form of local security pertaining to the acquisition of ports. A TCP/IP port is ambiguous as it operates on a first-come, first-served principle. The first-come, first-served methodology allows TCP/IP server sockets to bind onto any unbounded ports, leaving future server sockets unaware that a port is in use and of remaining ports.

Additionally, the current technology of TCP/IP ports does not provide any form of port security before binding onto the port. The present implementation of a TCP/IP server socket creates a new socket descriptor which can bind onto specific or all IP Addresses using any available port number. This method only verifies the IP Address associated with the socket without validating the port number or the socket permissions to utilize the port.

2. RELATED WORK

As might be suspected of a concept as seemingly basic as applying traditional permission schemes to ports, a variety of actual and potential solutions exist, each with limitations/drawbacks that motivated this project.

One naive approach to consider is out of the box kernel tuning. On both Linux and the BSDs, it is possible to change the range of ports randomly allocated by the system to processes calling bind(2). Similarly, the range of ports only root can bind to (by default through 1024) can be tuned. However, while this approach resolves the issue of a process not acquiring a required port due to the system's random allocation of ports, it leaves open the prospect of an unprivileged user acquiring a port âĂIJbelongingâĂİ to another unprivileged user. Similarly, upping the root-only port range does not address the core issue and, in fact, invites security issues through essentially promoting the running of networked server processes as root when they would be better off running as their own users.

SELinux offers a concept known as restricted ports that, at first glance, may appear to solve the problem at hand. It allows services to bind only to a set of âĂIJwell-knownâĂİ ports for that service–e.g. port 80 for an httpd process. Due to this paradigm, this system lacks the sort of fine-grained user/group-based control we generally associate with Unix permission systems and wish to enable through this project. In a multi-user environment, such as a shared server in a networking class, the ability to permission ports by user and group is a vastly more useful and flexible concept than well-known ports on a per service basis.

The concept of well-permissioned port reservation exists in IBM's proprietary mainframe Unix, z/OS. Lacking an IBM mainframe, our ability to evaluate this system was limited to perusing publicly available documentation. However, said documentation indicates that the features we deem important in a port reservation system are present, z/OS concepts like JOBNAMEs aside. Unfortunately, due to the proprietary nature of z/OS and likely significant differences between it and open Unix systems, it is not possible to perform a direct port of this system. Instead, we can consider this project an attempt to incorporate this element of z/OS functionality in the free software ecosystem.

[Albert thing, going to talk to him tomorrow]

With existing solutions and their shortcomings in mind, our focus must shift to avenues of implementation. The naive approach towards implementing a port reservation system is to do what, in essence, SELinux and z/OS do: modify the base system behavior to suit our purposes. In the context of a free Unix-like operating system, this manifests as directly modifying the kernel. It would be preferable to use a dynamically loadable kernel module, but as implementing a port reservation system requires redefining behavior in system call implementations, a kernel module's potential for defining new system calls and device drivers is not useful. Therefore, we are left with direct kernel modification. Theoretically, shimming in our desired behavior is not an enormously difficult task from a programming perspective. However, downsides to this approach become apparent when other factors are considered. First, assuming the changes are made, getting these changes incorporated into mainline kernels is a daunting proposition. This leaves custom kernel building as the only way of adopting the system. Such a barrier to deployment would likely severely dampen interest and impede adoption. Secondly, portability suffers with this approach, as unique code would have to be produced for each desired supported kernel. This is a potential major drain of development resources both now and in the future. This hints at a third obstacle: extended support. While the system call interface for networking is unlikely to change meaningfully in the near future, the odds of changes to the kernel necessitating a rework of the port reservation system implementation are likely higher. For all of these reasons, we elected to implement our system as a user-space daemon process, despite the inherent limitations of such an implementation.

3. IMPLEMENTATION

Having the daemon exist in user-space has many advantages that are auxiliary to software engineering principles. However, there is a major caveat to be considered: through user-space, we are unable to redefine the bind system call. Any users of our daemon would have to call our `secure_bind` and `secure_close` functions, as opposed to the bind and close system calls. We have a greater audience for our daemon, but cannot force any arbitrary binary to use the daemon (without a hack). We believe the security provided by the daemon is still a valuable attribute and that this system could find use in many applications. The daemon is Linux and FreeBSD compatible (though other compatibilities likely exist but have yet to be tested).

The daemon begins at startup and reads a configuration file. This configuration file is formatted as a list of TCP port numbers and the corresponding user IDs (uids) and group IDs (gids) that are allowed to access the port. The ports, uids, and gids can all be listed as ranges (e.g. ports 4000-5000 by uids 11111-11133). This configuration file is read by the daemon upon startup of the machine it is deployed on. The daemon then proceeds to create sockets and bind them to every port listed on its reservation list. It then stores in its internal data structures a list of all the uids and gids that can access a given port and if that port is currently in use.

Upon encountering any errors, such as an invalid port given (e.g. -7), the daemon reports this information to syslog, the logging daemon. All errors and matters of note in the daemon are reported through syslog and use syslog conventions on messages and alert types. Any system administrator can view, understand, and use the logs to help fix any errors or to know the state of the daemon.

After its startup routine, the daemon opens up a first-in-first-out special file (FIFO) that it blocks on awaiting another process to write to it. The FIFO is bound to a known pathname in the file system so that other processes are always able to write to it. A process that wants to request a reserved port calls a function that in part writes to the FIFO to wake the daemon.

A process that wishes to bind to a reserved port makes a call to our `secure_bind` function (separate from the daemon). Inside this function, a Unix domain socket (uds) is created. A uds acts like an internet socket, except that it is used for inter-process communication. In addition, a uds supports passing ancillary data such as file descriptors and process credentials. The final caveat is that rather than being given just an integer file descriptor, a uds is bound to a path in the file system. The caller of the `secure_bind` function specifies what this pathname will be. After creating the uds, the requesting process listens on the uds and writes the pathname of the uds to the daemon's FIFO.

When the daemon awakens to input on its FIFO, it reads the name of the uds provided by the requesting process and connects to the uds. The daemon then sends a message to the requesting process informing it that it is the daemon. Because this message is sent through a uds, the kernel writes the message to one of its buffers and creates a set of credentials (uid, gid, and process ID) for the daemon and includes that in the message sent to the requesting process. The requesting process is setup to expect credentials to be sent to it along with a small message from the daemon. It then verifies that the daemon's credentials are correct. If the credentials received do not match the daemon's credentials, the requesting process drops that connection and awaits another connection that might be the daemon. After verifying that the daemon is on the other end of the uds, the requesting process then sends the port it wants to the daemon across the uds. In the same way as before, the kernel inserts credential information for the requesting process into its message. This is critical, because it means that all credentialing is left up to the kernel, which we can assume is correct and if it has been compromised then our system is not presupposed to work. The daemon then receives the credentials and port request from the requesting process and checks its data structure to see if the process has access to the requested port and whether that port is currently in use. If access is denied or the port is in use, reports are made to syslog. Otherwise, the daemon takes the unused socket bound to the port that has been requested and marks it as in use. Then the daemon sends the socket (already bound to the requested port) across the uds to the requesting process. This is supported behavior through a uds, in that the kernel recognizes a file descriptor is being passed and updates file tables accordingly. The requesting process receives this socket and is able to use it as it sees fit. The daemon then services other requests or blocks awaiting further requests.

When a process is done using a socket, it calls our `secure_close` function (separate from the daemon). This function closes the uds and bound socket for the requesting process. By closing the uds, the daemon will be alerted. It wakes up and closes down its end of the uds. It then calls shutdown on the bound socket which clears any connections across that port. The daemon then marks the port as free again and resumes blocking until further input. If an auxiliary process does not call `secure_bind` before exiting, the daemon will still be alerted that the other process shut down its side of the uds (via file descriptors being closed on exit) and the whole process will continue regardless of `secure_close` being called or not. The possibility exists that a process could stay open indefinitely and not call `secure_close`, but be done using the port it received. In this case, the daemon would lose access to reassigning that port because it would never be alerted that the other process was done with the port. In that case, the port would be lost until machine restart, in which case the port will be closed as part of the shutdown and the daemon will reclaim the port upon startup. We feel this is a fringe case and is acceptable because root has allowed that particular uid or gid to access that port and what they do with it is their choice.

This process repeats itself with many different applications requesting access to ports and being granted access or being denied if either their credentials fail or the port is in use. Because the credentials are provided by the kernel, all the security inside of the daemon consists of is making sure no one attempts buffer overflow or similar style attacks, in sending bad information across the FIFO or a uds. With careful monitoring of these two input styles, we can ensure that ports are handed to out to uids and gids that have been granted access and not to others.

4. CONSLUSION

Secure port reservation is a feature that is currently absent from modern UNIX-based operating systems. The lack of such a feature exposes these systems to spoofing attacks from malicious users and generates significant logistical overhead for the administrators of multi-user systems.

Our proposed userland solution allows for an administrator to define a set of reserved ports such that access to the communications through these ports is only available to the uids and gids as defined by the administrator. By reserving ports for only authenticated users/groups, system administrators can prevent bad actors from spoofing potentially sensitive communications on the system. Furthermore, this secure port reservation functionality prevents users on a multi-user system from unintentionally acquiring a port that another user or group depends on.

The major drawback of our solution is that it requires a process's source to be modified to call our `secure_bind()` function when acquiring a reserved port instead of the bind() system call. Potential future work includes extending our tool to patch an arbitrary process's object file to call our secure bind function rather than the standard bind system call.