Portcullis : Secure TCP Port Reservation

Benjamin Ellerby, Raymond Weiming Luo
J Michael Meehan, Evan Ricks, Oliver Smith-Denny
Computer Science Department
Western Washington University
Bellingham, WA 98225
meehan@wwu.edu

ABSTRACT

Although there are many paradigms in use today for networked computer applications, the fundamental underlying foundation of all such applications is still the socket, originally implemented for BSD UNIX. The TCP and UDP port numbers associated with these sockets were divided into reserved and ephemeral, those above and below 1K. In UNIX this effectively meant that in order to acquire a port number below 1K the process needed to be executing as root. If you desire a network service to not run as root and use a port above 1K there is no way to guarantee that only specific user ids or group ids are allowed to acquire a given port. We have implemented a system that allows the system administrator to control access to ports based on process user ids and group ids. It is common practice to create special login ids used only to run certain services. Our software provides an extra degree of security, assuring that only the account or group created to run a particular network service is able to acquire the port(s) designated for that service.

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

1. INTRODUCTION

Even as computer network services have exploded over the past decades, the fundamental technology for computer networking using ports has remained unchanged. The Transmission Control Protocol and Internet Protocol (TCP/IP) defined a set of well know ports specified by the Internet Corporation for Assigned Names and Numbers (ICANN). Oringinally, the port numbers below 1K were reserved for assignment as so-called well known ports for standard applications. In the UNIX world this manifested itself as ports in the reserved range could only be bound to if the process id was root. In practice there are many ports above the original 1K range that are routinely used by well-known applications. The actual dividing line between the ports reserved only for root and the so-called ephemeral ports is a configurable constant that can be modified when build the operating system from source files. There are a number of weaknesses in this root-only and the rest of the world model. Running server processes as root open the system to extreme security risks. If any server process running as root contains a flaw that can be exploited the entire system can be compromised. On the other side of the dividing line the port are effectively first-come, first-served. There is no way to reserve a given port for a specific application and to ensure that only that application is able to bind to that port.

The current UNIX technology of TCP/IP ports does not provide any form of port security before binding onto the port other than the root and everyone else model. 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 provided the motivation for this project.

On both Linux and the BSD varieties of UNIX, it is possible to change the range of ports allocated by the system to processes calling bind(2). Similarly, the range of ports only root can bind to (by default below 1024) can be tuned. While modifying this boundady and restricting the overall range of ports available makes it possible to ensure a specific port you are interested in can only be accessed by processes with root privaledges this does not address the issue of first come forst serve for the ephemeral ports, i.e. those above the boundary. It leaves open the prospect of an unprivileged user acquiring a port you wish to belong to another unprivileged user. Similarly, upping the root-only port range invites security issues through essentially promoting the running of networked server processes to root when they would be safer running as a less privaledged user.

SELinux offers a concept known as restricted ports that, at first glance, may appear to solve the problem. It allows services to bind only to a set of "well-known" ports for that service–e.g. port 80 for an httpd process. The problem is that in order to use this to attempt to provide enhanced secutity for acces to ports you have to buy into the entire SELinux security model paradigm. You cannot run a regular Linux version and simply utilize a feture to achieve the security por ports you must change your entire system to utilize the

SELinux policy paradigm. In addition, SELinux out of the box only has type definitions for a few standard services. In order to use your custom service integrated with SELinux is a serious configuration exercise. [1] In other words, the SELinux approach lacks the sort of fine-grained user/group-based control we generally associate with Unix permission systems and wish to enable through this project. One example of the use of our approach would be to assign each user in a computer networking class as personal port number of numbers to use. We wish to prevent others from being able to access those port so that they are exclusively available for that student's programs when developing and debugging. The SELinux paradigm is totally inappropriate for this situation. The ability to assign permission to access ports by user and group is a vastly more useful and flexible concept in this situation rather than a registered service based model.

With existing solutions and their shortcomings in mind, our focus is on implementation for open source UNIX systems. One approach would be to implement a port reservation system by modifying the kernel to suit our purposes. It would be preferable to use a dynamically loadable kernel module if possible, 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 perhaps drivers is not sufficient. Therefore, we are left with direct kernel modification. The 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 variant. 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 pursue a practical implementation as a user-space daemon process, despite the inherent limitations of such an implementation. This will provide a less intrusive solution that can be deployed easily and will continue to be function in the face of future OS updates and releases.

3. IMPLEMENTATION

Having the daemon exist in user-space has many advantages. However, there is a major caveat to be considered. In user-space, we are unable to redefine the bind system call as it exists in the kernel. Any users of our daemon will 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 hacking the executable). We believe the security provided by the daemon is still a valuable resource and that this system could find use in many practicle applications in industry. The daemon is Linux and FreeBSD compatible. Other OS variants are likely compatible 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 can be accessed 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. Thus, using a UNIX doamin socket we are able to pass credentials for the socket to be used by the client.

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 out to uids and gids that have been granted access and not to others.

4. CONCLUSION

Secure port reservation is a feature that is currently absent from current UNIX-based operating systems. The lack of such a feature exposes these systems to spoofing attacks from malicious users and leaves the system administrator with no way to ensure that only the intended user ids created to run particular network services are the only ones that can aquire the port.

Our 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 limitation 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 the applicability of our tool by creating a program to patch an arbitrary process's object file to insert calls our secure bind function rather than the standard bind system call. This would allow the port reservation system to be used by applications for which one does not possess the source code.

This software has been tested and used on Linux and BSD style kernels. The software and accompanying man pages can be downloaded from the following site. (URL To be specified in final draft).

5. REFERENCES

[1] S. Vermeulen, *SELinux System Administration*. Packt Publishing, 2nd ed., 2016.