https://doi.org/10.48009/4_iis_2016_127-136

Issues in Information Systems
Volume 17, Issue IV, pp. 127-136, 2016

TRACING AND MANAGING A ROGUE CLOUD BASED SERVICE THROUGH A MOBILE APP

Dennis Guster, St. Cloud State University, dcguster@stcloudstate.edu.edu
Paul Safonov, St. Cloud State University, safonov@stcloudstate.edu.edu
Raqeeb Abdul, St. Cloud State University, rabdul@stcloudstate.edu.edu

ABSTRACT

There are many advantages to cloud computing, but there are also new and substantial security risks. Typically, multiple virtual machines (VM) are configured in cloud based architectures and hence this volume makes it difficult to keep track of each service running in the cloud. Numerous commands exist within the LINUX operating system that can be used to evaluate the purpose of the transport layer ports linked to the services running on a VM. However, the level of the entity in terms of rights that started that port sometimes makes it difficult to manage some ports. To illustrate this, a complex remote procedure call (RPC) example that generates several dynamically defined ports will be evaluated using the LINUX command set. While some ports were justifiable ports there were also suspected rogue ports started on the kernel level. To further aid in the identification and management of rogue ports a mobile app was generated.

Key words: Cloud Security, Mobile Apps, System Administration, Rights Management

INTRODUCTION

Cloud computing has been embraced as the architecture of choice by many businesses. While in general it enhances security strategies but it still results a number of security risks (Anthes, 2010). Because of its multidimensional structure managing security in a cloud can be very complex and requires a strategy beyond just invoking simple encryption (Jules and Oprea, 2013). This situation results in part due to the difficulty in tracking security issues within cloud computing environments because of the large number of virtual machines involved (Sun, Chang, Sun and Wang, 2011). The complexity is further exasperated because each virtual machine must support multiple services which are often shared within the cloud. Because of this multilayer complexity, some IT professionals feel that the true security challenges of cloud computing are still unknown (Hyman, 2013). Cloud computing typically involves creating a customized virtual environment to meet the needs of a given company. However, clouds still need to run on hardware. In many cases a small private cloud can be run on a small stack of enterprise level computers. Perhaps as few as a half a dozen units, but each unit might encompass 24 processing cores. To make this hardware configuration practical, virtualization of hosts is applied. This scenario often increases the number of logical computers (virtual machines) to a value in the hundreds. This works well if the goal is to follow green computing strategy because hundreds of physical hosts can be reduced to a stack of just 6 physical hosts. However, security personnel will need to evaluate the cloud as a logical model which is complex and will involve several levels of abstraction because virtual zones, virtual hosts, replication and virtual networks are utilized. It is important to not be deceived by the small physical foot print of a cloud. Many security issues are unique to a cloud while others were a host level problem that became more acute when the host was virtualized within the cloud. The complexity of cloud computing increases the danger of security issues occurring on the services level. This is because the processes involved can often propagate to other hosts within the cloud which ties into a major problem in all of multi-user computing: managing rights. This is complex enough in a traditional computing environment, but more so in a cloud because rights can be inherited across hosts in a cloud (Roberts and Al-Hamdani, 2011). To illustrate this problem an example will be used in which a RPC starts a kernel level process on a host. Because it is a kernel level process it is next to impossible to remove using ordinary means. This paper plans to expand on the work of (Guster, Abdul and Rice, 2015; Abdul, Lebentritt, and Guster, 2016) which create security management tools that deal with complex cloud level security issues. Further, to provide quick an effective usage the tools can be launched via a mobile app. The tool developed herein will be added to the existing framework and become a menu option.

The two prior tools used the router cache to identify attacks that were not caught by the intrusion detection system and mitigated denial of service attacks caused by an over provisioned process. Therefore, in this paper the scope of the investigation will focus on auditing the service level port assignments on a virtual machine within a cloud and providing a mobile apps management interface to deal with identified rogue ports that cannot be removed due to rights issues using common techniques. In cases where the rights limitations cannot be overcome, the importance of a firewall structure in the cloud will discussed.

REVIEW OF LITERATURE

Services in Cloud Security

The maturity process for security strategies in a cloud computing is ongoing. Therefore, the total potential of this architecture is yet unrealized (Hyman, 2013). It seems logical then to devise a strategy to separate traditional security concerns from those inherent within a cloud based architecture (Mell, 2012). To help end users deal with this more complex world that uses several layers of security to protect data, guidelines have been developed. These guidelines assist users in selecting the appropriate security levels and as a result they become more aware of the security ramifications of cloud computing (Roberts and Al-Hamdani, 2011). RPCs provide an excellent example of cloud level abstraction which can lead to complexities in determining the security level of a RPC service. This is due in part to the potential of high level rights being inherited across hosts. At first glance the idea of a remote procedure call is quite simple. Its primary purpose is centered around the transfer of process level control information as well as the data contained within a program running on a single computer. When analyzed in the context of a cloud the concept must be expanded to include the transfer of control and data information via a network which probably will include multiple VMs. This process can be explained by the following sequence: a remote procedure is run, the calling environment is put on hold, the required parameters are sent across the network to the VM where the procedure is to execute and then the desired process or processes are executed there remotely. When the process finishes then the results are returned to the initiating VM. At that point the suspended process restarts and that makes it appear that the service is running on a single-machine (Birrell and Nelson,1984). To make this work often the rights management strategy takes a back seat.

RPCs further complicate things because not only are multiple VMs involved in the computing transaction but often multiple ports too. Each one of those ports has their own process stack as well. The definitive example is NFS (network file system). NFS is a RPC that utilizes multiple daemons (a system level process that displays and monitors service information). Further, each daemon requires its own port assignment. The configuration specifications for each daemon is scattered among a number of startup scripts providing a hacker with a multi attack point environment from which to create a rogue process. That process could then be linked to the port system through the port-map where it could then masquerade it as a legitimate service (Toxen, 2007). This level of complexity can make it difficult to ascertain which user truly owns a given process associated with a rogue port. This is especially true if the returning process is not returned on the user level but, rather handled on return by the operating system kernel (Rushby, 1981). Therefore, it is crucial that RPCs be implemented securely which necessitates using the same isolation logic and rights management that would be applied in basic cloud design (Vahidi, 2013).

Need for Mobile App Management in a Cloud

Given that the existence of rogue ports can be disastrous and must be dealt with in a timely manner (Guster, Schmidt and Paidi, 2015), the interface used to alert the system administrator to such attacks must be agile. To provide a remote management interface that facilitates ease of use, speed and security a mobile app merits consideration. For cost considerations, equipment rooms are staffed at minimal levels which necessitate management through a mobile device. This paper features examples that are based on the Linux operating system and it is appropriate to review those Apps that are compatible with the Linux operating systems. While there are examples of LINUX mobile apps available, however they often depend on secure shell (SSH) or some other type of remote virtual terminal program to link to the operating system. This strategy requires that the desired command set be entered, which may be problematic given the keyboard characteristics of some mobile devices (Geier, 2015). However, the convenience of

a mobile device can't be underestimated because most system administrators/security analysts will carry a smart device with them wherever they go, which makes the APP readily available and easily accessible. The primary goals in adapting a mobile device to manage rogue ports are typically related to identifying and quickly blocking/removing the rogue port, designing an interface that is easy to use, and implementing the management function in a secure manner. If these goals are not met, the solution then would have limited value. The authors devised a similar security APP (Guster, Abdul and Rice, 2015), however that APP was designed to identify and mediate rogue processes. In that mobile app the goal was to protect the system while a true decision concerning the danger of the offending rogue process was being made by the system admin. Often this decision will take time, while the mobile app approach should minimize some of the end-to-end delay associated with logging in remotely via a virtual terminal program or a browser there will be some asynchronous human think time involved. The strategy in (Guster, Abdul and Rice, 2015), was simply to just suspend the rogue process, and if required, the system administrator could kill it via the mobile app. A similar approach will be applied herein where the goal would be to ensure it is being blocked on the firewall level which is temporary solution. Of course a more permanent solution can be invoked by the system administrator by actually killing the socket being used by that port. Within LINUX the fuser command does offer a port kill option which will be investigated.

AUDITING THE PORTS

With the promised advent of IPV6 there are commonly UDP and TCP version 6 ports co-existing in Linux based hosts. However, in an effort to reduce the scope of this paper, the focus was limited to a discussion of TCP version 4 ports. The first step in evaluating these ports is to simply use the netstat command to gain basic configuration information.

```
buster@os:~$ netstat -a | more
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:52334                 *:*                     LISTEN
tcp        0      0 *:sunrpc                *:*                     LISTEN
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 *:57526                 *:*                     LISTEN
```

The evaluation typically starts with determining the purpose of globally defined ports (the ones with English names) and determining if they are actually needed. Standard releases of LINUX provide a table in /etc/services that depict the Internet Assigned Numbers Authority (IANA) values which provides a cross reference to their numeric value. For example, the common web port (http) is displayed below and has been assigned port 80. As stated earlier, this IANA port listing is officially recognized and hence the port assignments have world-wide meaning. Services could be created locally and only have meaning on only a given host or within the cloud. These services would typically be defined in a services.local file. The format of this file is similar to the services file, an example is also displayed below. The audit process resulted in two named ports being resolved (sunrpc, http). Each port can be justified as follows: the SUNRPC port was needed to support the centralized file system within the cloud that allowed for centralized storage/management of data and HTTP is needed to support web services. At this point the purpose of the other two ports (52334, 57526) is unclear. However, a look at the concept of dynamic assignment will provide some clues.

```
buster@os:~$ cat /etc/services | grep http
http            80/tcp                  # hypertext transfer protocol

buster@os:~$ cat services.local
udptest         44444/udp    #Buster's special test port for udp traffic
```

So then, how does dynamic assignment of ports work? Because NFS is a complex RPC it uses multiple ports of these ports, only the port map (sunrpc) and the primary NFS service are included in the services file. The remaining ports are assigned dynamically upon host boot-up. While this process can confuse a hacker, it can make access outside of the cloud (or even the host) difficult if some means of updating the firewall after every reboot is not

considered. To further slowdown a hacker the port map can be suppressed outside the cloud zone providing an additional layer of security. RPCs use the client/server model and it is Appropriate to evaluate the port maps on both sides of that model. In the example below 127.0.0.1 is the NFS client and 10.17.59.195 is hosting the NFS service. In both cases SUNRPC provides the port mapping. The purpose of the status service is to provide crash-and-recovery functions for the locking services. The server side is more complex and necessitates running the NLOCKMANAGER. This service manages daemons related to data locking and the mount daemon which facilitates requests from clients related to file system mounts. We can now use the information from the port map below to reconcile of one of the two undefined ports from the original netstat output, 52334 which is the status function on the client side and now can be deemed a legitimate port.

```
buster@os:~$ rpcinfo -p 127.0.0.1
   program vers proto   port  service
    100000    4   tcp    111  portmapper
    100024    1   tcp  52334  status

buster@os:~$ rpcinfo -p 10.17.59.195
   program vers proto   port  service
    100000    4   tcp    111  portmapper
    100003    4   tcp   2049  nfs
    100021    4   tcp  60206  nlockmgr
```

A look at the sockets (net.node.port) supporting the client/server connection provides insight into how data is transferred between the client and server. The display file systems command (df) reveals that the server side of the NFS file system is fs.bcrl.local (10.17.59.195) and is mounted to the client side as /home. To gain the socket level connection information the netstat command is used and the server side port is 2049 which of course is its assigned value in the services file. However, the client port is 1013 which is in the protected range (root required to modify) and not in the services file. This port needs to be protected because it is a "multiplexed" client port which means that multiple users on the client side will use that port rather than forming their own connection when using the NFS service. The advantage is that less connection management overhead is incurred.

```
buster@os:~$ df
Filesystem      1K-blocks     Used Available Use% Mounted on
fs.bcrl.local:  41283968 16691072  22495744  43% /home
buster@os:~$ netstat -tn | grep 2049
tcp        0      0 10.17.59.234:1013      10.17.59.195:2049      ESTABLISHED
```

That leaves one port that needs to be resolved, port 57526. A sound starting strategy to resolve this port would be to obtain the user, process ID and command that started the process/port. To do this a sophisticated configuration of the netstat command can be useful and an example is included below run as the root (via sudo).

```
buster@os:~$ sudo netstat -apeen |more
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User       Inode      PID/Program name
tcp        0      0 0.0.0.0:52334           0.0.0.0:*               LISTEN      106        1544       1081/rpc.statd
tcp        0      0 0.0.0.0:57526           0.0.0.0:*               LISTEN      0          8532       -
```

While the results provide information for the NFS status port (52334) and the port map (111) they only include limited information about the rogue port (57526). The necessary information was gained for port 52334 which is owned by user 106 (statd = status daemon). This process is running with process ID 1081 and was initiated by the rpc.statd command. Unfortunately, the information is limited for port 57526 with only the user ID being provided which is 0 (root). This is problematic because the root starts lots of software and using the ID 0 for the root makes easy for a hacker to run bad code under the auspices of being the root. Also, code can be run by the kernel itself and such case it is often given the root ID. If that is case then all kernel originated ports need to be documented as well. For the purposes of this paper we are assuming that port 57526 is in fact rogue due to the following facts/assumptions:

1. It is not in the services file,
2. It is not in the services.local file,
3. It is not listed in the port maps and
4. It is not an authorized kernel originated port.

LOGIC FOR THE MOBILE APP

The first step is to use the output from the previous section to identify potential rogue ports without process identification numbers. Because these ports have no process ID they can't be readily removed with the kill command, the second step is to verify that they are being blocked by the host level firewall. To search for a rogue process the output from above can be used and parsed for the "-" that would normally contain a process ID/command. If that is found an alert would be generated. At that time it will be important to assess the log files to ascertain that the host is being protected by the host level firewall. A look at the ufw log file reveals the information below. While there is an indication that the port, 57526, is being blocked (dropped). It is a continuing problem. Further, there is no information about how or when the rogue port was instantiated.

```
buster@os:~$ sudo cat /var/log/ufw.log | grep 57526
Apr 30 14:58:12 os kernel: [4382342.956914] [UFW BLOCK] IN=eth0 OUT=
MAC=00:50:56:8c:05:a6:00:50:56:8c:0a:ee:08:00 SRC=10.17.59.195 DST=10.17.59.234 LEN=60 TOS=0x00
PREC=0x00 TTL=64 ID=57322 DF PROTO=TCP SPT=873 DPT=57526 WINDOW=14600 RES=0x00 SYN
URGP=0
```

The mobile app will also be able to display the most recent five entries in either file to verify that although, the rogue process is still running it is at least blocked from outside the host. Further an option will be included that would allow killing the socket associated with that port.The LINUX operating system has mechanisms in place to kill a port without knowing its process ID. To illustrate this scenario we can use a simple java socket call program to generate a port to kill. Because this port was generated by a general user the port is actually available and it could be killed by removing the process, but just assume the process ID is not known. To provide background information on the port in this case 18002, the netstat command (network statistics) can provide basic information. The fuser command (identify processes using files or sockets) can also be used to view this basic information. In both cases process ID 24807 is identified. They each offer additional information that might be useful such as user ID, inode, or access status (F=open for writing). A third command, ss (another utility to investigate sockets) finds the port, inode (UNIX internal object index) and even displays the hex socket address. This would especially be useful in case where there is no process ID.

```
buster@os:~$ netstat -apeen | grep java
tcp6       0      0 :::18002                :::*                    LISTEN      3004536945 13407130    24807/java
buster@os:~$ fuser -v -n tcp 18002
18002/tcp:           dguster   24807 F.... java
buster@os:~$ ss -itoea | grep 18002
LISTEN     0      5                    :::18002                 :::*       uid:3004536945 ino:13407130 sk:ffff8801370a3a00
```

It is easy to kill the test port by using the fuser command. A killed statement appears on the server side and when the ss command is rerun the process/socket no longer appears.

```
buster@os:~$ fuser -k 18002/tcp
18002/tcp:           24807
buster@os:~/javaclass$ java TempServer
Waiting for connection....
Killed
buster@os:~$ ss -itoea | grep 18002
```

Now the real test begins to see if this method will work with a port started on the kernel level. The port below, 57526 was started as part of a loading process of nfs modules. Note the port is available from any interface on the host, receives user ID 0 (typically root on passwd) and displays a "-" where the process ID and command that started the process would normally appear. An attempt with the fuser kill command is made, but it is unsuccessful even when run as the root. This is because the port and its unknown process ID were started on the kernel level and even the root doesn't have rights on that level. A good solution to eliminate the port just using standard LINUX commands has yet to be found by the authors'. The next step of their planned investigation would center around, recompiling the kernel and creating a shell interface with kernel level rights.

```
buster@os:~$ sudo netstat -apeen | grep 57526
tcp        0      0 0.0.0.0:57526           0.0.0.0:*               LISTEN      0          8532       -
buster@os:~$ sudo fuser -k 57526/tcp
buster@os:~$ sudo netstat -apeen | grep 57526
tcp        0      0 0.0.0.0:57526           0.0.0.0:*               LISTEN      0          8532       -
```

DEVELOPMENT STRATEGY FOR THE ROGUE PORT TRACE MOBILE APP

The Android Mobile application is designed to facilitate the availability of information pertaining to rogue ports and their associated firewall logs. To implement the APP in the easiest manner, the design will be segmented into the following modules: identifying the rogue port, notifying the Mobile Device about the rogue port, killing the rogue port and viewing the Uncomplicated Firewall logs. An opening screen appears below.

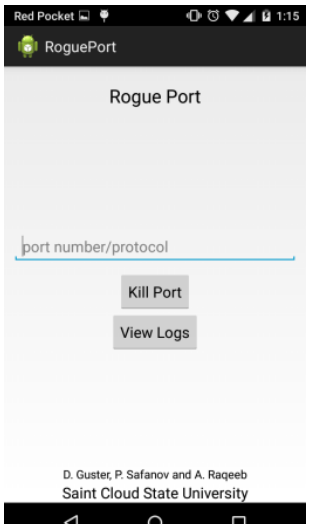

Figure 1. Home Screen of the Mobile App

All of the modules above are discussed from two perspectives i.e. client side which is an android mobile application and server side which is LINUX host running the Ubuntu flavor. The server side basically includes shell and php scripts.

1) Identifying the rogue port

To identify the rogue port on the server side we have created a script which acts like a UNIX cron job (chronological job scheduler). This script in turn calls another script where the actual logic to identify the rouge port is defined. The logic used, is to parse the netstat command and look for processes with no process id i.e. one with "-". Below is the command which was used to achieve this task.

```
buster@os:~$ sudo netstat -ntap
Proto Recv-Q Send-Q Local Address           Foreign Address         State       User       Inode      PID/Program name
tcp        0      0 0.0.0.0:52334           0.0.0.0:*               LISTEN      106        1544       1081/rpc.statd
tcp        0      0 0.0.0.0:57526           0.0.0.0:*               LISTEN      0          8532       -
```

rogue.sh – script to identify the rouge port

```
#! /bin/bash

netstat -ntap |sort -r |awk '{if(length($7) ==1 && $7 == "-"){{system("php notify.php "$1" "$4)}}};'
```

So, whenever this script finds ports with no process id it will call another script named notify.php which will be discussed more in the next section.

2) Notifying the Mobile Device about the rogue port (Push Notifications)

The push notification is simply a message that is delivered by server to the mobile application that can be generated without any request from the mobile application. In this implementation, the push notification is used to notify the user about any rogue ports that are running on the LINUX host in real time. Below is the basic architecture diagram for the push notifications:

The mobile device will register with the GCM upon successful login to the Application. If the registration is

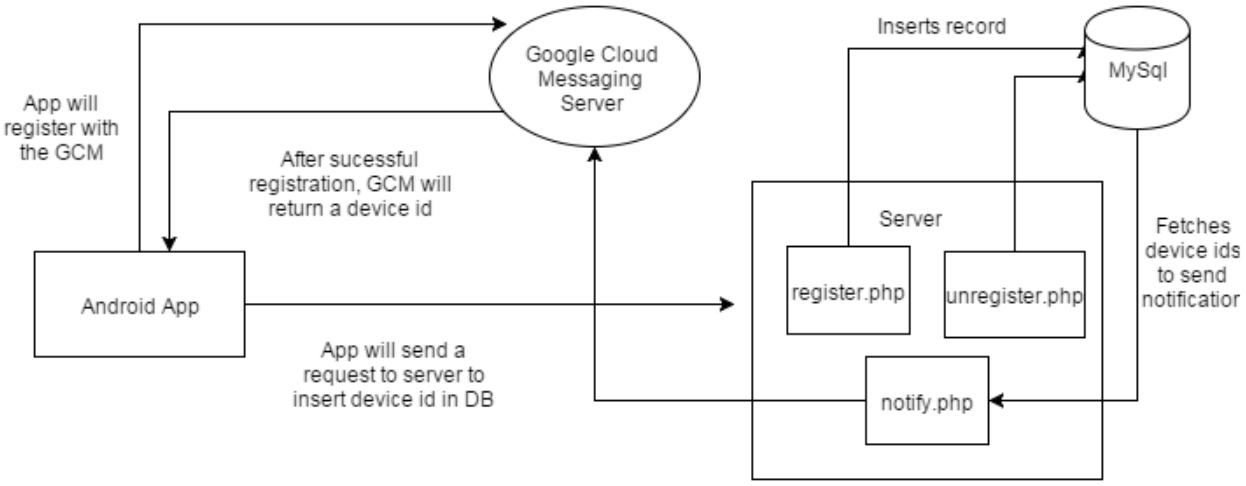

Figure 2. Push Notification Architecture

successful, the GCM will return a device id which is a unique id to identify the device and application in the cloud. To send a notification to the mobile application a device id is needed. For future use this device id is saved in a MySQL database. This process is accomplished while the registration API is called. To send a notification, a script named notify.php was created which in turn has a function termed "notify" which contains an argument message and the associated port number that is captured from the netstat command. The rogue.sh script discussed above can then call this "notify" function along with arguments message and port number / protocol.

3) Killing the rogue port

As discussed above, the processes which are started by the kernel do not have a process id and cannot be killed by root as well. But this can be achieved by recompiling the kernel and give the user access to kernel. For this paper, we have created a Rest API that can kill the process using the port number which is further defined by the protocol it is using. Later, we can extend this to killing the process on the kernel level. To achieve this we have created a Rest API that is consumed by the mobile application.

4) Viewing the Uncomplicated Firewall logs

The log files play vital role in identifying if the port is block on firewall level. So, in the mobile application herein an option was provided to view the ufw.log which can be used to verify that the port is actually blocked. For this purpose a REST API was created that will parse the ufw.log from the LINUX host and return the data in JSON format. This in turn can be consumed on the mobile device as formatted data. Here is the screenshot from the mobile application which displays the logs.

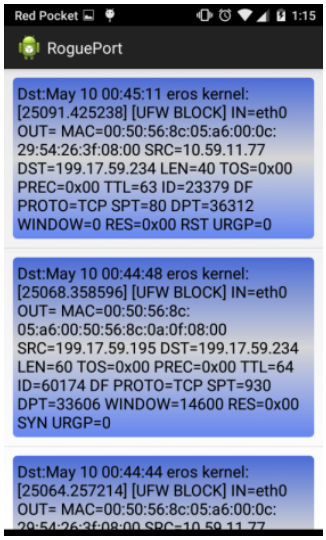

Figure 3. Mobile Screen for Firewall Logs

5) Encryption

The library https://github.com/serpro/Android-PHP-Encrypt-Decrypt/_ has been used to implement encryption on both the mobile device and server. This library makes use of the AES 128 algorithm. When the mobile application sends a request to the server the data is encrypted and thereafter it can be transmitted safely. In this example the data being transferred is the process id. So therefor, every process id is encrypted and then sent over the REST API and the process stack within the REST API is capable of decrypting the data. The Initial Vector and Key both are initially shared between the mobile app and the server.

6) Authentication

In this mobile application the authentication is done with HTTP Basic Auth in conjunction with AES-128 encryption. “auth.php” is the main file used for handling the authentication and the login function within the API on the server. Every request from the mobile app is authenticated using the HTTP Basic Auth process. The credentials created from this process are stored within the persistent memory of the mobile device once the user has successfully logged in. This will facilitate future user requests to the server. In order to achieve more security, the username and password have been encrypted using AES-128 before transmission to server. The server will handle the request by decrypting the request and following the rest of authentication process. A future idea to be dealt with in this project would be to add an administrative control panel that could be used to manage user accounts via the mobile application. Keeping this in mind the information is stored in a table called ‘users’. The passwords for this are then stored as a MD5 hash.

Discussion and Conclusion

Throughout computing the importance of documentation cannot be over looked. So perhaps, the first step in effective port management within a cloud architecture is to adopt sound policy. In many situations, the purpose of

the ports is defined through a services file which provides the required documentation that describes their need and purpose. With RPCs often non-typical ports are needed. The example used herein illustrated that when complex remote procedure calls are used that additional ports are generated dynamically and can be a real challenge to the defined security policy. In fact, if started remotely under the supervision of the kernel the unintended port may be next to impossible to remove.

For example, port 57526 running on a VM in the author's cloud Appeared to have been created by malware used by hackers to provide a "back door" to the host. However, further review revealed that this port was created by the installation of the NFS software. So, based on this observation it is important when installing/updating software to checkpoint the port structure and compare the before and after state of the ports. This especially hold true when ports are being defined dynamically. Of course in basic LINUX structural terms there needs to be a way of linking the port to a process, startup command, user id, hex socket id and inode. From the coded example used herein it is clear that this auditing process required using several LINUX commands. Of course, if this strategy was well documented the process could be streamlined and performed quicker. Because of the status of port 57526 the audit could not be effectively complete because both the server-side and client side were started on the kernel level. So in effect the available information was limited to residual error message files and no other data could be linked to the port via its available inode.

Of course within a cloud there are mechanisms in place to combat the effects of potentially rogue ports. In this paper, it is the host level firewall that prevents any connectivity to the rogue port from the public interface. More specifically, in the case of port 57526 the host level firewall prevented the client process from a remote host from forming a connection and this blocked connection was recorded in the UFW firewall log. The UFW log files besides detecting intrusion problems can be used to evaluate if policy is being followed as well as its degree of effectiveness. Given the basic policy used in which everything is blocked and ports are opened selectively as needed. Given the state of the internet this is certainly the correct policy. At times, it results in a legitimate port being blocked. In the case of NFS this is true when a port is generated dynamically and ports as well as the resulting processes are started on the kernel level. As stated earlier there is no information in regard to the process id or the program that started the port. The implication of this is that, while UNIX based system have numerous advantages they require more sophisticated personnel to manage which demand higher salaries.

While the basic structure of the LINUX operating system commands provide a sound basic frame work, determining which commands are needed and their appropriate options is not a simple process. This process would be very difficult for an inexperienced user. Even for experienced users there is a degree of trial and error required to reach a solution. Besides papers such as this the internet has a wealth of information, particularly on forums. However, to speed up the process it was decided to integrate the port management related information described herein into a mobile app. This APP not only has the appropriate command structure built in, but provides a new level of convenience to the management process. This is especially important, given the fact that equipment rooms are very sparsely manned. It allows systems administration/security personnel to manage high level security problems while on call or from home. Of course, this remote management technique would necessitate that sensitive data traverse the airways. So to be effective the APP design needs to effectively adopt encryption strategies and AES-128 was used. Hackers are known to often be successful in part due to the use of a dynamic attack strategy. Protecting the port structure is critical in preventing access to a host particularly on a shell level. Not only are they constantly devising new attack techniques, but they are keenly aware that it is important to minimize their attack fingerprint. That is why kernel level ports with no directly defined process ID are so disturbing. While this is an unconventional strategy and a hacker would really need to think outside the box to successfully implement it the results could be devastating. This paper provided background information leading up to killing rogue ports and defined the potential problem with kernel level ports. Further, the authors' recognized the need to manage killing rogue ports in a secure and timely manner, hence the need for the mobile app described herein.

REFERENCES

Abdul, R., Lebentritt,L. and D. Guster (2016). Developing a mobile app that uses the routing cache table to detect brute force secure shell attacks not detected by the intruder detection system. *Proceedings of the 49th Midwest Instructional Computing Symposium,* Cedar Falls, IA.

Anthes, G. (2010). Security in the cloud. *Communications of the ACM,* 53(11), pp.16-18.

Birrell, A., Nelson, B. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems, 2*(1), pp39-59.

Dennis Guster, Mark Schmidt and Paidi Karthik. (2015). Auditing complex cloud based services with inodes to trace the origins of potentially rogue processes within a Linux host (Virtual Machine). *Proceedings of the IABPAD Conference,* Las Vegas, NV.

Geier, Eric. (2015). 10 Android Apps for Linux Server Admins. Retrieved from http://www.linuxplanet.com/linuxplanet/reviews/7301/2.

Guster, D., Abdul, R., and Erich Rice. (2015). Mitigating Virtual Machine Denial of Service Attacks from Mobile apps. *Journal of Network and Information Security*, 3(2). pp. 21-31.

Hyman, P. (2013). Augmented-reality glasses bring cloud security into sharp focus. *Communications of the ACM, 56*(6), pp. 18-20. Retrieved from http://bit.ly/MGQGyW

Jules, A., Oprea, A. (2013). New Approaches to security and availability for cloud data. *Communications of the ACM, 56*(2), pp. 64-73.

Mell, P. (2012). What's Special about Cloud Security?. *IT Professional, 14*(4), pp. 6-8, Retrieved from http://www.computer.org.ezproxy.umuc.edu/csdl/mags/it/2012/04/mit2012040006.html

Roberts, J., Al-Hamdani, W. (2011) *Proceedings from the 2011 Information Security Curriculum Development Conference*. Who can you trust in the cloud? A review of security issues within cloud computing. Pages 15-19. Retrieved from http://bit.ly/1bfH3jn

Rushby, J. (1981). Design and verification of secure systems. *Proceedings of SOSP, the eighth ACM symposium on Operating systems principle,* pp. 12-21.

Sun, D., Chang, G., Sun, L., Wang, X. (2011). Surveying and Analyzing Security, Privacy and Trust Issues in Cloud Computing Environments. *Procedia Engineering, 15*, pp. 2852-2856.

Toxen, B. (2007). The seven deadly sins of Linux security. *Security, 5*(4).

Vahidi, A. (2013). Secure RPC in embedded systems: evaluation of some Global Platform implementation alternatives. *Proceedings of the Workshop on Embedded Systems Security, 4.*