Article can be accessed online at http://www.publishingindia.com

Mitigating Virtual Machine Denial of Service Attacks from Mobile APPS

Dennis Guster*, Raqeeb Abdul, Erich Rice*****

Abstract

With the advances in cloud computing and the use of virtualisation, the complexity of computing systems had never been greater. Due to this greater complexity securing these systems have also become more complex and difficult, especially given the ease with which hackers can bring to bear Denial of Service (DoS) attacks. Luckily, advances in technology have also provided the means of administering these complex computing systems through the use of mobile devices, such as with an Android OS based smartphone. In this paper we provide an option for managing the eradication of rogue processes created through DoS attacks by way of a mobile device application or app. Through the use of this mobile app built on the Android platform a system administrator would be alerted to a potential security incident and be given the tools to kill a rogue process without having to be onsite or initiate a terminal session through secure shell or another terminal program. This type of option could be very appealing to small or mid-sized enterprises which cannot afford the cost of having personnel staffed onsite 24 hours a day, seven days a week. The mobile app was built with security in mind and would provide a system administrator a quicker and more direct ability to curtail DoS attacks before they caused greater harm.

Keywords: Denial of Service (DoS), Mobile APPS, Android, Rogue Processes

Introduction

Denial of Service attacks (DoS) are relatively easy to launch and if the target is not prepared, can be very effective. Therefore it is critical that a potential target site devises a well thought out and timely strategy to combat such attacks. DoS have been around some time and have become easier to launch with the advent of the Internet (Hafner & Lyon, 1996). The Internet, in effect gave the hacking world the access needed to launch DoS attacks remotely with no need to gain physical access to the computer room of the target. There are certainly many classifications of attacks that could originate from the network and foundations of which are described by Cheswick and Bellovin (1994). However, all attacks will become instantiated in the operating system under one or more process identification numbers (PID). Properly identifying such attacks requires a sound monitoring strategy (Kargl, Mair, Schlott & Weber, 2001). Monitoring the PID provides a common element that can tie together the different system resources that might be affected by a DoS attack. Commonly, these resources fall into four categories: CPU, memory, storage, and network.

Identifying Sub-Targets

Tying the Sub-Targets Together with a PID

To illustrate this interrelationship, a temperature conversion service (which converts Fahrenheit to Centigrade) has been instantiated on a VM within a cloud. The Linux netstat (network statistics) command output given reveals that it is running on network port 18002 and available from all networks (: : :) assigned to that host

* Saint Cloud State University, United States of America. Email:dcguster@stcloudstate.edu
** Saint Cloud State University, United States of America. Email:rabdul@stcloudstate.edu
*** Saint Cloud State University, United States of America. Email:rier1201@stcloudstate.edu

and has been given PID 3224 (Fig. 1).

Fig. 1: Linux Netstat Command Output

```
dguster@eros:~$ netstat -apeen | grep java

tcp6       0      0 :::18002                :::*                    LISTEN
1004536945 26213822   3224/java
```

The next command, ps (display processes) as shown in Fig. 2 provides us with the memory (MEM) and CPU usage, in both cases they are well below 1% which indicates that this process in its present state is not causing either a memory or CPU DoS.

Fig. 2: Linux Display Processes Command Output

```
dguster@eros:~$ ps -aux | grep 3224

USER       PID %CPU %MEM    VSZ   RSS
TTYSTAT START   TIME COMMAND

dguster   3224  0.3  0.4 2245236 17656 pts/3    Sl+
14:03   0:02 java TempServer
```

However, in a cloud architecture using virtualisation and symmetric multiprocessing the CPU resources may be distributed across many physical devices. The second ps–ALF command in Fig. 3 shows us that the original PID 3224 has been broken down into 14 "light weight processes" and are assigned to 4 different processors (PSR) numbered 0-3. While the original process 3224 retains the same PID number for the light weight process ID, each subsequent light weight process receives a new and different LWP number. However, because these are assigned hierarchically if one were to kill the root level process, in this case 3224, the whole process stack would be removed. This fact will become important later in the paper where mitigating a DoS attack as quickly as possible becomes imperative. A quick side note about this type of architecture and DoS attack is that the ability to multi-thread tasks and bring multiple processors into the fray when a DoS attack occurs makes it much more difficult for a DoS attack to be successful. Further, it lengthens the time a system administrator has to kill the offending process(s) and mitigate the attack.

Fig. 3: Linux ps-ALF Command Output

```
dguster@eros:~$ ps -ALF | grep 3224
UID        PID  PPID  LWP  C NLWP    SZ   RSS PSR
STIME TTY          TIME CMD
dguster   3224  3120  3224  0  14 561309 17672  3 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3225  0  14 561309 17672  0 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3228  0  14 561309 17672  1 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3229  0  14 561309 17672  2 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3230  0  14 561309 17672  3 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3231  0  14 561309 17672  0 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3232  0  14 561309 17672  0 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3233  0  14 561309 17672  2 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3234  0  14 561309 17672  3 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3235  0  14 561309 17672  1 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3236  0  14 561309 17672  1 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3237  0  14 561309 17672  2 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3238  0  14 561309 17672  1 14:03
pts/3    00:00:00 java TempServer
dguster   3224  3120  3239  0  14 561309 17672  1 14:03
pts/3    00:00:02 java TempServer
```

Last the ls of command (list open files) provides a way of linking the process to storage resources. In Fig. 4 we are able to see respectively: the path to a directory on secondary storage, a library loaded from secondary storage into memory, a temporary file system workspace in memory but writable as a file to secondary storage, a UNIX socket which allows the java class to be linked to the operating system and finally a IP version 6 network connection linked to port 18002. A bottom up approach would require that each of these elements linked to PID 3224 be deleted independently which would more than likely not be quick enough to mitigate a modern DoS attack. Therefore, an effective method would be to kill the root process, in this case PID 3224, and all sub-processes (sometimes referred to as children) would then be eliminated as well.

Fig. 4: Linux lsof Command Output

```
dguster@eros:~$ lsof -p 3224

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

java 3224 dgustercwd DIR 0,21 24576 786911 /rhome/dguster/javaclass

java 3224 dguster 2u CHR 136,3 0t0 6 /dev/pts/3

java 3224 dgustermem REG 8,1 149280 715938 /lib/x86_64-linux-gnu/ld 2.15.so

java 3224 dgustermem REG 8,1 32768 525582 /tmp/hsperfdata_dguster/3224

java 3224 dguster 12u unix 0x0000000000000000 0t0 26213820 socket

java 3224 dguster 13u IPv6 26213822 0t0 TCP *:18002 (LISTEN)
```

Killing the Primary and Secondary PIDs with a Kill Command

Below, in Fig. 5 we seethat the process 3224 is killed. Then, a search using the ps –aux "display process command" reveals only the search argument using the grep filter. Further, a list of open files query returns nothing. So therefore 3224 and its children are no longer instantiated on the VM.

Fig. 5: Linux Kill Command Output

```
dguster@eros:~$ kill 3224

dguster@eros:~$ ps -aux | grep 3224

dguster 6051 0.0 0.0 6500 624 pts/0 S+ 15:22 0:00 grep --color=auto 3224

dguster@eros:~$ lsof -p 3224
```

Identifying a Strategy for Killing PIDs from a Mobile Device

The primary goals in adapting a mobile device to support the remote mitigation of DoS attacks are typically related to identifying and quickly eliminating rogue PIDs, the ease of use, and of course added security. If any one of these goals is not met, then the solution would provide limited value. In terms of identifying rogue processes there are certain parameters that can be monitored and a simple example appears in Fig. 6. A java class is writing a large file into memory and is taking up 98.5% of the available memory available for this type of application. Therefore, if this mobile management strategy is to be effective this obvious violation has to be identified and the associated process (PID 12023) killed in a timely manner. Writing a script or scripts to identify such a violation and sending an alert to the mobile device of a system admin would be a major step in the success of this type of project.

Fig. 6: Example of Identifying Rogue Process

```
dguster@eros:~$ ps -aux | grep java

dguster1202398.5 21.0 3271356 850956 pts/0 Sl+ 09:52 0:22 java MemoryMappedFileInJava2
```

In terms of killing the process, once it is identified the problem will then be greatly affected by end-to-end transmission delay. Once the command is executed and transmitted to the host the delay is relatively minimal, as can be seen in the output shown in Fig. 7. The task itself takes ~ 3.65 milliseconds to run and the elapsed time from command to response is ~ 7.11 milliseconds.

Fig. 7: Linux Kill Command Output

```
dguster@eros:~$ perf stat -B kill -9 12161

Performance counter stats for 'kill -9 12161':

        3.651294 task-clock                #    0.514 CPUs utilized

                1 context-switches         #    0.000 M/sec

                1 CPU-migrations           #    0.000 M/sec

              183 page-faults              #    0.050 M/sec

<not counted> cycles

                0 stalled-cycles-frontend  #    0.00% frontend cycles idle

                0 stalled-cycles-backend   #    0.00% backend cycles idle

                0 instructions             #    0.00  insns per cycle

                0 branches                 #    0.000 M/sec [19.34%]

<not counted> branch-misses

      0.007110059 seconds time elapsed

dguster@eros:~$ ps -aux | grep java

dguster  12184  0.0  0.0   6500   624 pts/3    S+   10:02   0:00 grep --color=auto java
```

The challenge then will be to protect the system while a true decision about the offending rogue process is being made by the system admin. Once again this decision will take time, although the mobile APP approach should minimise some of the end-to-end delay associated with logging in remotely via a virtual terminal program or a browser there still will be some degree of asynchronous human think time. Therefore, a strategy needs to be pursued to have the potentially rogue process suspended by the identifying script and if determined to be a rogue process, then killed by the sys admin via the mobile APP. In cases in which the process is determined not to bea rogue process then it could be restarted from the same APP. While suspending a process stops it from using CPU resources, the memory content remains in place but cannot grow. The LINUX command series in Fig. 8 shows how a given process can be temporarily disabled and restarted using the kill command. A PID (14439) from a potentially rogue process is identified and disabled with the signal stop switch via the kill command. When the process is then displayed its status flag changes from S (suspended waiting for resources) to T (stopped). In the last part of the results below it is then changed back to S (if indeed it is not a rogue process), but of course it couldalso be killed if it truly is a rogue process. This type of logic will need to be incorporated into the design of the mobile APP to minimise the effect of false positives.

Fig. 8: Linux Command to Temporarily Disable and Restart a Process

```
dguster@eros:~$ ps -aux | grep java

dguster14439  38.6  25.7  3271356  1043744  pts/0  Sl+  12:45   0:11 java MemoryMappedFileInJava2

dguster  14455  0.0  0.0   6500   624 pts/3    S+   12:46   0:00 grep --color=auto java

dguster@eros:~$ kill -SIGTSTP 14439

dguster@eros:~$ ps -aux | grep java

dguster   14439   8.8  25.7  3271356  1043744  pts/0  Tl   12:45   0:12 java MemoryMappedFileInJava2

dguster  14463  0.0  0.0   6500   624 pts/3    S+   12:47   0:00 grep --color=auto java

dguster@eros:~$ kill -SIGCONT 14439

dguster@eros:~$ ps -aux | grep java

dguster   14439   7.3  25.7  3271356  1043744  pts/0  Sl   12:45   0:12 java MemoryMappedFileInJava2

dguster  14490  0.0  0.0   6500   620 pts/3    S+   12:48   0:00 grep --color=auto java
```

Identifying a Strategy for Authentication

In order to facilitate the authentication process, an existing library will be used (GitHub, 2015). By using this library a HTTP Basic Auth object which deploys AES-128 encryption can be easily deployed. Therefore, every request from the mobile device can be authenticated using the HTTP Basic Auth object. For more security, the username and password are also encrypted using

AES-128, then transmitted to server. Furthermore the passwords that will be stored in the MySQL database will be stored as a MD5 hash. In an effort to speed up requests, the encrypted credentials can then be saved on the mobile device in persistent memory.

Identifying a Strategy for Encryption

It is critical that the data and commands to-and-from the mobile device remain secret. Knowing the process name and its associated PIDs may be useful to hacker. Certainly if hacker could read the data stream and determine its purpose it would at least alert them that the system is already in a vulnerable state and additional attacks may successfully complete an intended DoS attack. Thus it would be wise to use a state of the art encryption algorithm such as AES-128 (Advanced Encryption Standard with a 128 bit key) on both the client and server side. This algorithm is still considered a safe standard and not currently vulnerable to brute force attacks (Miller, Vandome & McBrewster, 2009). This algorithm also is readily available in a library designed to support mobile APPS for Android mobile devices (GitHub, 2015).

Thus it will be easy to implement because in this library, the initial vector and secret key are shared between the client and server. A brief explanation of how the objects can be created using this library appears below:

In Java:
MCryptmcrypt = new MCrypt();
String encrypted = MCrypt.bytesToHex(mcrypt.encrypt("Sample data"));
String decrypted = new String(mcrypt.decrypt(encrypted));

In PHP:
$mcrypt = new MCrypt();
$encrypted = $mcrypt->encrypt("Sample data ");
$decrypted = $mcrypt->decrypt($encrypted);

Review of Literature

Traditional Methodology

Killing a process under normal condition is typically not a problem. However, if there is a need to kill a process, once a DoS attack has been launched it can be problematic because the kill command might not be able to get the resources it needs to run including an interrupt. Further, computer equipment rooms (in small and medium sized companies) that house the hardware for the server side of processing are typically not manned 24 hours a day 7 days a week and hence console access to the host(s) is not readily available. This necessitates some type of remote access for system admins. How quick and secure this access is will be paramount to killing the DoS process in a timely manner.

A commonly used method would be to gain remote access via a secure virtual terminal service such as secure shell (ssh). This common scenario and associated problem is described in Killing a Process on a Remote Machine When the Machine is Stuck(2015). In this example a user has run a complex problem using MATHLAB that has locked up the system. The solution offered is to use secure shell via the two scenarios listed below.

Scenario 1: the secure shell command string

dguster@eros:~$ ssh -l dgustereros 'pkill -9 java'

--

Do not login to this system without permission.

--

READ all prompts to change password if required.

--

Error opening terminal: unknown.

Scenario 1: the java process is killed

dguster@eros:~$ netstat -a | grep java

Scenario 1: the console on the server side

dguster@eros:~/javaclass$ java TempServer

Waiting for connection...

Killed

Scenario 1: the command level performance statistics

Performance counter stats for 'ssh -l dgustererospkill -9 java':

91.506240 task-clock # 0.154 CPUs utilized

38 context-switches # 0.000 M/sec

10 CPU-migrations # 0.000 M/sec

1263 page-faults # 0.014 M/sec

223611024 cycles # 2.444 GHz [50.18%]

0 stalled-cycles-frontend # 0.00% frontend cycles idle [55.94%]

0 stalled-cycles-backend # 0.00% backend cycles idle [57.02%]

0 instructions # 0.00 insns per cycle [54.89%]

0 branches # 0.000 M/sec [48.80%]

0 branch-misses # 0.00% of all branches [44.38%]

0.592828639 seconds time elapsed

Scenario 2: the secure shell command string

dguster@eros:~$ ssh -l dguster2 eros 'pkill -u dguster'

Obviously the first solution would be preferred because the process ID linked to the java runtime is killed and the user shell should then be unaffected. In the second scenario the user shell is killed which of course kills all children processes including the java runtime. Note, the pkill command allows one to kill by process name rather than the process ID. This is critical because the remote user doesn't need to do a PID lookup which would further slowdown the process. The performance statistics for the ssh command string are interesting in that even though the command was run from the same host the process still took almost .6 of a second and 223 million CPU cycles.

Using a Mobile APPS Approach

Given that the goal of any remote process management software would be ease of use, speed and security whatever the client side software might be it makes sense to deploy it on a mobile device. Because the examples herein are based on Linux platforms it makes sense to evaluate the available mobile Apps designed to be compatible with Linux systems. While there are numerous mobile APPS available, they typically would still depend on ssh or some type of remote virtual terminal program to gain access to the operating system. Then it would require that the appropriate kill command be entered, which may not be all that fast given the keyboard characteristics of some mobile devices (Greier, 2015). However, a mobile device will be crucial in an effective solution because it is readily available and easily accessible.

Proactively Identify a Rogue Process

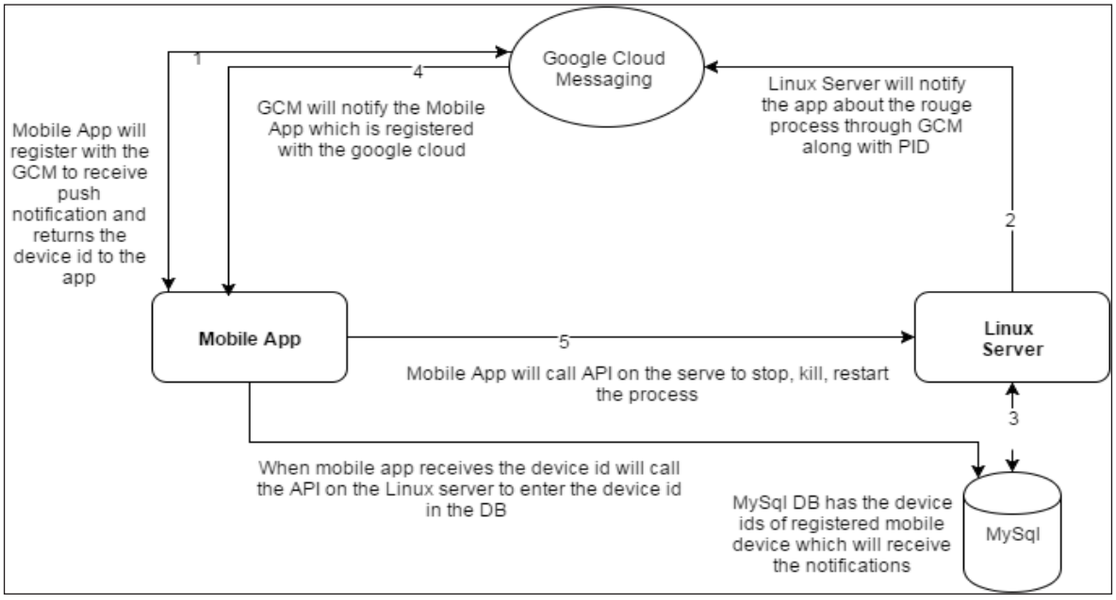
In order to effectively manage DoS processes it is important to devise a proactive rather than a reactive policy. As state earlier there are four main categories of denial of service: CPU, memory, disk, and the network. For the sake of simplicity we will focus on the CPU. There are many ways of using the Linux tools to identify a process that is overusing CPU resources, but in most cases they are a variant of the ps command. To illustrate this process we picked the method from How to Find Which Process is Causing High CPU Usage (2015), and the results appear as shown in Fig. 9.

Fig. 9: Results of Ps Command

```
dguster@eros:~/javaclass$ ps -eopcpu,pid,user,args | sort -k1 -r | head -10
%CPU   PID USER     COMMAND
82.8  3748dguster  java MemoryMappedFileInJava2
12.0  3764 root     [flush-0:21]
0.1  1395 root     /usr/sbin/vmtoolsd
 0.0 31281 root     [kworker/0:0]
 0.0 30824 root    /usr/lib/policykit-1/polkitd --no-debug
  0.0  30757  root          /usr/sbin/console-kit-daemon --no-daemon
 0.0 30697 syslog   rsyslogd -c5
 0.0 30344 root     /usr/sbin/sshd -D
 0.0 29030 root     [kworker/3:2]
```

The resulting list displays the 10 processes that are using the most memory. The first entry in the list is really problematic in that it is using 82.8% of available memory. Note that the Linux operating system has processes in place to help safeguard against a denial of service. For

Fig. 10: Graphical Overview of the Interaction Among the Mobile App, Google Cloud Messaging Server and Linux Server



example, in the list above the flush process is freeing memory by writing dirty memory pages out to disk.

Devising a Strategy to Deal with False Positives

When dealing with a true rogue process there certainly is a need to kill it. However, when scripts are written and contain arbitrary values such as a maximum CPU utilisation value as a means to identify a rogue process it is not an exact science. An instance in which an event with a legitimate high CPU value triggered a kill could have a disastrous effect on a production application. The killing of a legitimate process would be termed a false positive. In the literature related to intrusion detection, the effects of false positives and the need to prevent them is well documented (Denning, 1987; Baayer, Regragui, & Baayer, 2014).

The literature does indeed show that false positives can occur in dealing with the killing of processes via their process name or PID. This is particularly true in cases in which the PID of a given application keeps changing (because it keeps ending and restarting). Hence, one would need to find an effective way of keeping track of the current PID or better yet use the pkill command to kill it via its process name (Process ID, 2015). Further, the strategy of utilising a short term process stop, with the ability to kill or restore upon inspection, is also mentioned in the literature (Hayden, 2009). This solution is attractive because it offers an immediate pause to a detected rogue process giving a Sys Admin time to truly determine if it should actually be killed.

Implementation Strategy

The implementation strategy was designed to provide an easy, quick and flexible means to deal with rogue processes. Fig.10 provides a graphical overview of the interaction among the mobile app, google cloud messaging server and the Linux server.

To provide a logical strategy from which to evaluate the project the design will be presented from the client side and server side separately.

Server Side

The host in this case is using a popular release of Linux called Ubuntu. The services on this host are provided by an Apache Web Server. This service facilitates the communication with the mobile application. Also, on this host a MySQL database has been configured to provide secondary storage to store the mobile device IDs. Of course a modular approach has been used to create the code. Below are all the components of the server which are used to implement this application:

cProc.sh: This bash script is used to restart the stopped process

register.php: This is the rest API exposed to mobile app used to store the device id of the Android Phone in the MySql DB.

This API accepts the GET Method and takes 'id' as an argument.

unregister.php: This is the rest API exposed to the mobile app used to delete the device id of the Android Phone in the MySql DB.

This API accepts the GET Method and takes 'id' as an argument.

db_config.php: This file has the configuration used to connect to the MySQL DB. This is not an API and it is used internally by other files which need to connect to DB.

cronjob.sh: This is a bash script which basically acts a cronjob which calls the process.sh every five seconds.

MCrypt.php: This is a library which uses AES-128 internally for encrypting and decrypting data. This is not exposed as API and it is only used internally with other files.

startProcess.php: This is a Rest API which is displayed to the user by the mobile application. This is used to start the process. It will internally call the 'cProc.sh' internally.

This API accepts the GET Method and takes the 'pid' as anargument.

stopProcess.php: This is a Rest API which is displayed to the user by the mobile application. This is used to stop the process. It will internally call the 'sProc.sh' internally.

This API accepts the GET Method and takes the 'pid' as anargument.

sProc.sh: This bash script is used to stop the running process.

killProcess.php: This is a Rest API which is displayed to the user by the mobile application. This is used to kill the process. It will internally call the 'kProc.sh' internally.

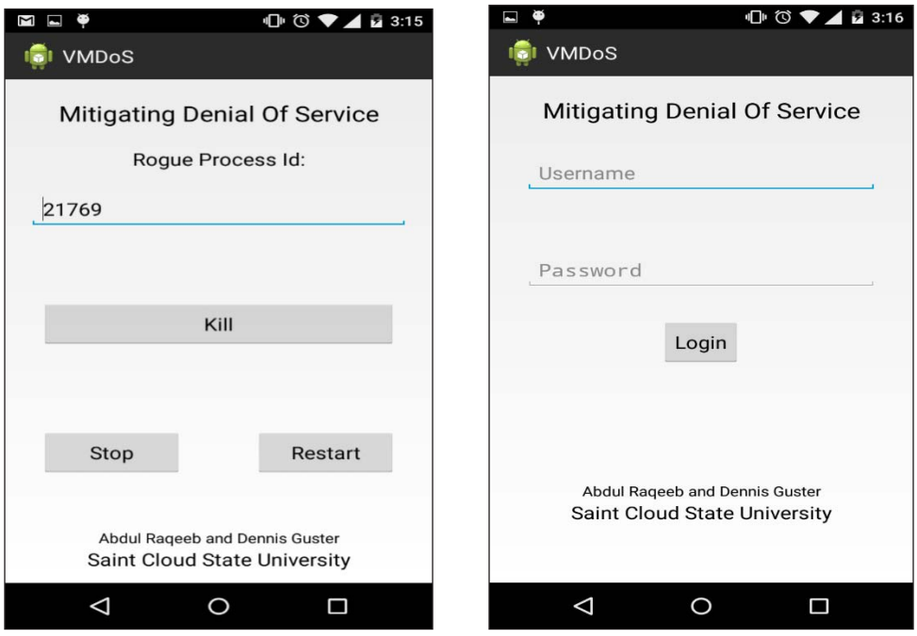
kProc.sh: This bash script is used to kill the running process.

process.sh: This is a bash script which is used to identify the process with a high CPU usage. This will also call the 'kill.sh' to suspend the process.

kill.sh: This is a bash script which actually suspends the process and calls the 'notify.php'.

auth.php: This is not an API and it is the authentication library which contains the code for HTTP Basic

Fig. 11: Login Screen and Administration Screen



Authentication along with AES-128 encryption. The credentials are transmitted as encrypted data.

notify.php: This is API calls the GCM (Goggle Cloud Messaging) to send the push notification. It gets the device ids from the MySQL DB.

Client Side

In our design, the client is an Android Application. This application has two screens i.e. the Login screen and the Administration screen which are shown as screenshots in Fig.11.

The login screen is simply used to authenticate the application with the server. The login screen will be shown only once i.e. when the user installs the application. Therefore, when the user opens the APP for the second time it will not show the login screen again, instead the administration screen would then be displayed. Also, when the user logs in successfully for the first time the credentials are encrypted using AES-128 and then stored on the mobile device to support future communications.

Push Notifications

Fig.12 illustrates how the push notification is used within this application.

The push notification, also called the server push notification, is the delivery of information from a software application to a computing device, without a specific request from the client. In our design, push notifications are used to alert the user about the existence of a rogue process.

Fig. 12: Push Notification

After a successful login, the mobile device will register with the GCM. Upon a successful registration, the GCM will then return a device id to the mobile application. This ID will be used to uniquely identify the device within the cloud. The mobile application will then send a request to the server along with this device id and then store it in the database (at this point theregister.phpapi is called). Next, when the server finds any rogue process it can, it will then send the push notification using the device id (again the notify.phpapi is called to notify the registered devices). Of course the object Notify.php has the code needed to send the request to the GCM. Below is an example of a request to the GCM:

GCM API :https://android.googleapis.com/gcm/send

Method : POST

Parameters:

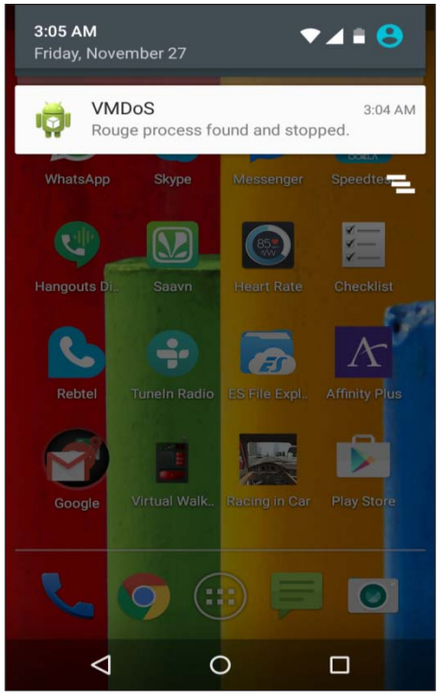
registration_ids : is the array of device ids

data : is the array of key value pair (Ex: “message” =>”Rogue process found”)

Header : Authorisation: key=”This key can be found on the Google API Console

Fig.13 shows an example of the Android mobile device screenshot after the rogue process has been killed.

Fig. 13: Screenshot After the Rogue Process Has Been Killed



Authentication

In this application authentication is accomplished using the HTTP Basic Auth object which deploys AES-128 encryption. The container “auth.php” holds the main file used for handling the authentication and login APIs on the server. Every request from the mobile device is authenticated using the HTTP Basic Auth object. Below is the example of applying the Auth header on the mobile

application:

```
credentials = MCrypt.bytesToHex(mcrypt.encrypt("raqeeb")) + ":"
        + MCrypt.bytesToHex(mcrypt.encrypt("superman2"));

String credBase64 = Base64.encodeToString(
        credentials.getBytes(), Base64.DEFAULT).replace("\n",
        "");

HttpClient httpclient = new DefaultHttpClient();
HttpGet httpGet = new HttpGet(params[0]);
httpGet.setHeader("Authorization", "Basic " + credBase64);
HttpResponse response = httpclient.execute(httpGet);
```

The code below illustrates how Auth requests are handled on the server side.

```
function pc_validate($user,$pass) {
    $mcrypt = new MCrypt();
    $username =$mcrypt->decrypt($user);
    $password = $mcrypt->decrypt($pass);
    $pass_md5 = md5($password);

    $query = "select * from users where username = '$username' and password = '$pass_md5'";

    $result=mysql_query($query);
    $count=mysql_num_rows($result);

    if($count==1)
    {
        return true;
    }
    else
    {
     return false;
    }
}
```

For more security, the username and password are also encrypted using AES-128 and then transmitted to the server. Once they are transmitted to the server the credentials are decrypted to verify the identity of the user. The users are stored in 'users' table in the MySQL database. Let's say you want to give access to a user, you can then simply add the user to the 'users' table. A desirable future addition to this project would be adding an administration page which could be used to manage access to the users. The passwords in the MySQL database are stored as a MD5 hash for added security. The encrypted credentials are stored on the mobile device in persistent memory to support future requests.

Encryption

The library described in GitHub (2015) has been used to support encryption on both the mobile device as well as the server side. This library makes use of AES-128. When a mobile application sends a request to the server the data is encrypted and then it is transmitted. As stated earlier some of this data may in fact be sensitive such as Process ID. When this method is used every process id is encrypted and then sent over the REST API. Of course on the receive side the REST API is capable of decrypting the data.

The Initial Vector and Key both are shared between mobile app and the server. Below is sample implementation of

```
WebRequest wr = new WebRequest();
String encProcess = MCrypt.bytesToHex(mcrypt.encrypt(process));
wr.execute(SERVER_URL + "stopProcess.php?pid=" + encProcess);
```

Below is a sample implementation of the decryption process on the server side:

```
$mcrypt = new MCrypt();
$pid = $mcrypt->decrypt($_GET['pid']);
```

Discussion and Conclusions

Denial of service attacks are a real problem on many cloud based virtual machines. In some cases they may even be a result of internal software that is not correctly designed, configured or tested. While object oriented programming should help minimize such attacks there are still numerous legacy systems, which may be on a lower layer in a SOA (service oriented architectures) approach. Last, typically a hacker's ultimate goal is to trick a user into downloading and executing "bad code". While dealing with "bad code" was not a primary discussion point herein, it is another concern that could warrant mobile APPs management as a possible solution.

The motivation behind this project was to create a fast, secure and convenient means for dealing with DoS attacks. Further, within the production zone of the cloud used testing purposes there was a desire to minimize the effects of false positives. Under ideal conditions the equipment room that hosts the cloud based services would be manned 24 hours a day, 7 days a week. However, the high cost of personnel, especially in small or medium sized businesses often precludes this from happening. So typically, the best case scenario is having a Sys Admin on call with their mobile device with them. The App developed herein, is not designed to be all inclusive in regard to remote management, but rather to illustrate some concepts that would help reach the design goal of fast, secure and easy to use.

First, the persistent sign-on reduces the time it would take to get through the connection and authentication phases when compared to using a traditional approach such as secure shell (ssh). Second, to support this persistent image and the transfer of sensitive data the authentication

and encryption phases took advantage of the robust AES-128 algorithm. Further, from a programming perspective this was easy to implement via library calls. Of course passwords stored on the system's database would need to be protected as well, so in our example an md5sum hash was used to provide greater security.

Third, the means to identify and kill or stop a rogue process was evaluated carefully. It was clear that a rogue process needed to be identified quickly and stopped as soon as possible to mitigate damage. The example used herein was based on CPU utilisation, one of the four typical DoS attacks. Once the utilisation threshold was exceeded the process was stopped and an alert was generated. For this example, we selected a 5 second poll interval to alert the Sys Admin, although it could be set to a shorter or longer value as needed. The logic was that because the script monitoring the host stopped the rogue process it is far less dangerous and not an immediate threat. Based on the trials from our test-bed the stop process can often be done in less than 10 milliseconds. Humans do not operate in a millisecond world so a poll interval of 5 seconds seems reasonable. Hopefully the Sys Admin on call would get the alert and make a decision to either restart or kill the offending process in a couple of minutes.

Last, distributed processing much less a cloud infrastructure is not well understood by many end-users as well as undergraduate college students. While the primary goal in this project was to develop a prototype mobile APP to manage rogue processes quickly, securely and conveniently this project could be used as an educational case study. Too often a user assumes that all the code needed for an APP is running on the client side device, in this case an Android phone. Because of the modular design employed herein, this project would be ideal to provide students with a meaningful example related to what runs on the server, what runs on the client and what runs in the cloud. The fact that the algorithms needed for authentication and encryption were available via library calls further illustrates the power of the object oriented programming concept.

It is planned that this project will serve as a stepping stone to a more sophisticated APP with greater functionality. The authors are also considering adapting this project to serve as a model for a culminating project in a second semester programming class as well.

References

Baayer, J., Regragui, B., & Baayer, A. (2014). False positive responses optimization for intrusion detection system. *Journal of Information Security*, 5(2), 19-36. Retrieved from http://file.scirp.org/Html/1-7800184_43038.htm.

Cheswick, W. R., & Bellovin, S. M. (1994). *Firewalls and internet security*, Addison Wesley Longman.

Denning, D. (1987). *An intrusion-detection model.* IEEE Transactions on Software Engineering, SE-13, 222-232. Retrieved from http://dx.doi.org/10.1109/TSE.1987.232894.

Geier, E. (2015). *10 Android apps for Linux server admins*. Retrieved from http://www.linuxplanet.com/linuxplanet/reviews/7301/2.

GitHub. (2015). Retrieved from https://github.com/serpro/Android-PHP-Encrypt-Decrypt.

Hafner, K., & Lyon, M. (1996). *Where Wizards Stay Up Late, Simon & Schuster,* New York.

Hayden, M. (2009). *Two great signals: SIGSTOP and SIGCONT*. Retrieved from https://major.io/2009/06/15/two-great-signals-sigstop-and-sigcont/.

High CPU Usage of Flush Process. (2015). Retrieved from http://serverfault.com/questions/294505/high-cpu-usage-of-flush-process.

How to Find Which Process is Causing High CPU Usage. (2015). Retrieved from http://unix.stackexchange.com/questions/20483/how-to-find-which-process-is-causing-high-cpu-usage.

Kargl, F., Mair, J., Schlott, S., & Weber, M. (2001). *Protecting web server from distributed denail of service attacks.* Retrieved from http://www10.org/cdrom/papers/409/.

Killing a Process on a Remote Machine When the Machine is Stuck. (2015). Retrieved from http://unix.stackexchange.com/questions/25599/killing-a-process-on-a-remote-machine-when-the-machine-is-stuck.

Miller, F., Vandome, A., & McBrewster, J. (2009). *Advance encryption standard*. Alpha Press.

Process ID and Killing Process – ps Command. (2015). Retrieved from http://unix.stackexchange.com/questions/94430/process-id-and-killing-process-ps-commmand.

Reed, D. A. & Dongarra, J. (2015). Exascale computing and big data. *Communications of the ACM*, 58(7), 56-68.