Network Layer Spoofing on Mobile Devices with Frida: Project Report

JIALE HU*, YANG LI*, and QIHANG SUN*, University of California, Los Angeles, USA

Frida is a powerful instrumentation framework on various platforms. It enables runtime tracing and flexible hooking of system functions. This project aims to implement a spoofing tool on Android-based mobile devices that can perform HTTP/HTTPS hijacking, DNS redirection, etc. The ultimate goal is to support spoofing for arbitrary parts of the packet from/to Android Devices. The implementation of the project is evaluated through a series of attack cases.

Additional Key Words and Phrases: network, Frida, Python, C++, C, JavaScript, Java, Android, Linux

ACM Reference Format:
Jiale Hu, Yang Li, and Qihang Sun. 2021. Network Layer Spoofing on Mobile Devices with Frida: Project Report. 1, 1 (December 2021), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the rapid growth of the number of mobile devices and prevalence of Internet, mobile security has became one of the most important fields to be studied, as mobile security is the protection of portable devices such as smartphones, smartwatches and tables from threats and vulnerabilities.

Study of mobile security not only requires theoretical analysis, practical analysis can also play an important role in finding vulnerabilities in existing systems. Instrumenting tools can be particularly helpful for conducting such analysis. Frida [3] is a dynamic code instrumentation toolkit for native apps on various platforms, which enables run-time tracing and flexible hooking of system functions. In addition to instrumenting, Frida can also be used to perform spoofing through bypassing or modifying mobile security in different layers.

2 BACKGROUND AND RELATED WORK

The network stack is crucial for studying network security. It consists of seven layers, which includes physical layer, data link layer, Network layer, transport layer, session layer, presentation layer and application layer. Each layer of the network stack may have it own vulnerabilities and thus is susceptible to be attacked.

For the physical layer, wireless data link has became dominant in mobile devices. However, because the characteristics of wireless channel are open and broadcasting, it makes wireless network becoming vulnerable to be attacked [6]. For the network layer, there exists some distributed denial-of-service (DDoS) attacks, which are implemented by using source IP address spoofing [7]. Although this kind of attack is in the network layer, it is not our project's focus since DDoS hardly targets

*All authors contributed equally to this project.

Authors' address: Jiale Hu, jialehu@ucla.edu; Yang Li, liyang19@ucla.edu; Qihang Sun, qs97@ucla.edu, University of California, Los Angeles, 405 Hilgard Ave, Los Angeles, California, USA, 90095.

XXXX-XXXX/2021/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

mobile devices. For the transport layer, the author Mnnit provided a improved edition on current Gray Hole attack in mobile Ad-Hoc networks [9]. For the application layer, utilizing HTTP requests to overwhelm victim resources is a common way to perform a application-layer attack [8].

Although there already exists various ways to attack the network stack through different layers, spoofing with Frida can still provide insights in the field of mobile security. Because we can discover what kind of weakness does the kernel code have by using Frida, which can then help fix existing design flaws or help new applications avoid problems caused by those kernel's weakness based on our findings.

3 IMPLEMENTATION GOALS AND INITIAL ISSUES

The goal of this project is to implement network spoofing tool that supports spoofing for arbitrary parts of packet. Specifically, based on network stack, the expected scope of spoofing includes Internet Protocol (IP) packets, Transmission Control Protocol (TCP) segment, Hypertext Transfer Protocol (HTTP) request and response, Hypertext Transfer Protocol Secure (HTTPS) certificate.

IP Spoofing. An IPv4 packet consists of IP header and user data, where header has 14 fields including version, source/destination address, protocol, total length, checksum, etc., and user data contains arbitrary data with size up to 65536 bytes that can be interpreted protocol header field. Spoofing of IP packet can be focused on packet procession pipeline and packet insertion/deletion.

TCP Spoofing. A TCP segment is carried by data section of IP packet, it also consists of its header and user data. TCP header has 11 fields including source port, destination port, sequence number, window size, checksum, etc. TCP provides reliable transmission of byte stream over IP protocol by handshaking, retransmission, and error detection. Spoofing at this layer can be focused on modification of TCP header to break its reliability.

HTTP Spoofing. For HTTP, data exchange is done by a sequence of request-response messages via TCP connection. Each message consists of HTTP header and body. HTTP header has various fields for context, caching, controls, authentication, security, etc. Spoofing of HTTP message mainly focusing on stealing credentials from headers and modifying message body.

HTTPS Spoofing. HTTPS is an extension of HTTP, it exchange data with bidirectional encryption to avoid man-in-the-middle attack. The initiation of encryption requires server to provide a signed certificate for client to validate. One of the spoofing at this layer is to prevent client from validating certificate, then a fake certificate may be used on a fake HTTP server to serve a client.

Initial Issues. With a list of initial spoofing plan for each network layer, the initial obstacles of the project includes connection from Frida to Android device, lack of understanding of network processing pipeline in Android kernel, identifying processing native functions to hook for spoofing at each layer, and designing their modification to implement spoofing.

4 SOLUTIONS

4.1 Solutions to Initial Issues

Frida setup and its connection to Android device is done through following steps:

- Root Android device.
- Install frida-server executable to device with Android Debug Bridge (adb).
- Run frida-server on Android device in background with root privileges.

where the privilege issues prevented us from connecting to the device successfully.

With device connected, functions to be hooked are to be determined for each layer. We start with analyzing source code of Android kernel [2] with a focus on network processing. We then

discovered that majority of network processing logic resides in Linux source code that Android depends on. To find the specific version of Linux kernel, [1] is referred. After reading through Linux packet processing pipeline, we determined following native functions to be hooked with Frida:

- IPv4 packet input processing (*/net/ipv4/ip_input.c*):
```
int ip_rcv(struct sk_buff *skb,
           struct net_device *dev,
           struct packet_type *pt,
           struct net_device *orig_dev)
```
- TCP segment input processing (*/net/ipv4/tcp_ipv4.c*):
```
int tcp_v4_rcv(struct sk_buff *skb)
```
- DNS resolution (*/net/getaddrinfo.c*):
```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res)
```
- Linux Socket (*/net/socket.c*):
```
int *recvmsg(struct socket *sock,
             struct msghdr *m,
             size_t total_len,
             int flags)
```

In addition to native functions, Java level functions are also explored since most of application layer (i.e. HTTP/HTTPS) network processing is implemented in Java. We first looked into Java codebase and identified *java.net.Socket* as a common class of socket connection, which has the following method for potential spoofing of response data:

```
public InputStream getInputStream() throws IOException
```

Beyond Java built-in classes, open source libraries for Android application development are also examined. We identified OkHttp [5] as one of the most popular implementation of HTTP client in Java and Android development. Its usage is also verified through hooking *getInputStream()* of *com.android.okhttp.internal.huc.HttpsURLConnectionImpl* on various apps in our testing Android device.

4.2 New Issues and Solutions

With a plan of function hooking for each network layer, new issues aroused as we start hooking the aforementioned functions. The first roadblock is that the dynamic library storing native functions are unknown in Android system. Even with this issue solved and hook is successful, the next roadblock is that Frida only support decoding of limited C/C++ types, which does not include c/c++ *struct*. In case that input *struct* of a native function is intercepted, Frida is only capable of showing the memory address, which limits our implementation to modify native functions.

To examine available dynamic library loaded in Android, we first tried to find the path in which dynamic libraries reside in Android file system, which is determined to be */system/lib* and *system/lib64*. To further validate this finding and search for symbols (i.e. classes and functions) in dynamic libraries, a Frida snippet for symbol search and discovery is implemented (Appendix A.1). With this snippet, we are able to locate and load symbol *getaddrinfo()* and *recvmsg()*. However, *ip_rcv()* and *tcp_v4_rcv()* are not found despite their occurrences in Android source code. Although progress are made to access the functions, this issue prevents us from implementing modification of TCP/IP processing pipeline. Without further details about Android dynamic library loading, we assumed that lower layer functions are hidden for security reasons in the specific Andriod device we have.

To support C/C++ *struct* reading, we looked into memory allocation for *struct* and determined that the memory address of fields of *struct* are allocated sequentially. With this finding, we proposed to add manual offset from memory address of the *struct* to find the address of the field-of-interest. The sample implementation can be found in Appendix A.2.

Based on solutions of the issues, we are able to proceed with designing detailed spoofing attack case (Demonstrated in next section) above TCP layer. For lower layer spoofing, the main limitation is the hidden native functions in our specific Android device. Although unable to perform actual spoofing, we briefly designed spoofing cases of IP packet and TCP segment insertion by modifying header fields (along with header checksum) and creating fake packet/segment *struct* to passed into native functions.

5 EVALUATION

In the previous section, we discussed various ways of spoofing by hooking into different system functions and Java methods. In this section, we will showcase how the spoofing works.

5.1 HTTPS Response Spoofing

The team used the native *Weather* app to verify HTTPS response spoofing. During our experiments, we found that this app uses Java library class *HttpsURLConnection* when retrieving weather data. Therefore, we were able to hook onto it, intercept the JSON response and modify its content (Appendix A.4). For example, as Fig. 1 showed, we were able to change the temperature shown to the number we want.

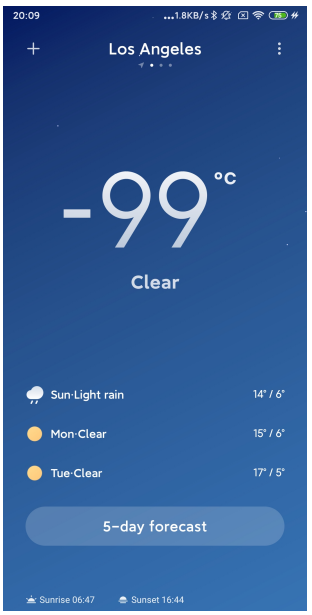

Fig. 1. HTTPS Spoofing with Weather App

5.2 HTTPS Certificate Unpinning

With HTTPS Certificate Unpinning, we were able to attack apps (e.g. native *Music* app) that uses certificate pinning to protect HTTPS traffic. To demonstrate the attack, we used a HTTPS debugging proxy (*HTTP Toolkit*) to intersect the Android device. Before we run our script, there will be errors

shown on the Toolkit log because the app rejected the proxy certificate of the Toolkit (Fig. 2) [4]. After we ran the script (Appendix A.5), we were able to remove the pinning and made the Music app to trust the proxy certificate and thus made it possible to inspect its traffic.

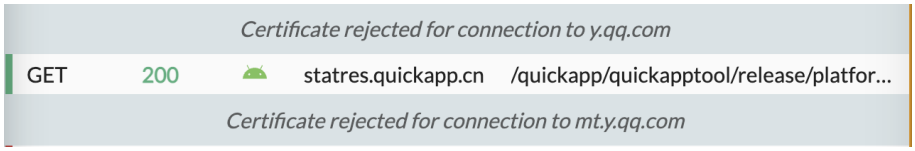

Fig. 2. Certificate rejected e.g. *mt.y.qq.com*

Fig. 3. HTTPS certificate unpinning (successfully connected to *mt.y.qq.com*)

5.3 DNS Redirection

In order to perform DNS redirection spoofing, we ran dynamic library scanning (Appendix A.1) for several apps and found that the native Browser app called the dynamic library function `getaddrinfo()` in `libc.so`. After we ran our script (Appendix A.3) to replace the `name` argument, say into `www.ucla.edu`, we were able to redirect every URL to `www.ucla.edu` (Fig. 4).

```
CS211 on  main [!] via  frida took 2m 19s
➜ frida -U -l hookNative.js Browser
     ____
    / _  |   Frida 15.1.8 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
Attaching...

[MI 6X::Browser]-> ----------------------getaddrinfo----------------------
Original Host: exp.sug.browser.miui.com
null
Original retval 0
>>>>>>>Going to: ucla.edu
New retval 0
----------------------
```

Fig. 4. DNS Redirection on Frida

6 DISCUSSION AND FUTURE WORK

The initial ultimate goal of our project was to implement a spoofing tool on Android platform, which enables spoofing for arbitrary parts of packets sent to/from the mobile device. The team started our work by exploring the Linux kernel and Java codebase to figure out what functions and methods we could hook on. We successfully performed HTTPS response spoofing, HTTPS certificate unpinning and DNS redirection. However, due to lack of support in decoding C/C++ struct with Frida, especially nested structs, we had difficulties in modifying the contents of TCP/IP packets. Futhermore, while doing dynamic library scanning for various Android apps, we noticed

that some kernel functions failed to be discovered. For instance, the function `getaddrinfo()` are called by the native Browser app, but not used by Chrome. One possible explanation is that Chrome built its own wheels in achieving those functionalities, or the function calls were hidden due to security reasons.

Future work on this project include figuring out the way to decode and edit values in nested C/C++ *struct*. This can be designed and implemented at Frida level such that a uniform API can be developed for user to access data in an arbitrary C/C++ *struct* Another work can be done in the future is to dig out the kernel functions hidden in RAM. Lastly, We can also investigate the source code of open source apps like Chrome to find useful functions/Java packages to hook.

7 CONCLUSION

In conclusion, in the project, the team implemented a spoofing tool on Android-based mobile devices using Frida. The team studied the way Linux kernel handles packet processing by investigating it's kernel code base, as well as Android Java APIs to figure out useful functions and packages to hook. The team identified these function calls with dynamic library scanning, successfully performed spoofing on the network stack, but also encountered some challenges such as reading and editing values in nested C/C++ structs. With future work addressing these issues, Frida could be turned into a more powerful tool in exploiting Android vulnerabilities, and could make developers aware of possible security pitfalls in designing their software.

ACKNOWLEDGMENTS

Thanks to our mentor Zhehui Zhang for explaining Frida frameworks and providing Android device for testing.

REFERENCES

[1] [n.d.]. *Android Common Kernels.* https://source.android.com/devices/architecture/kernel/android-common
[2] [n.d.]. *Android Open Source Project.* https://source.android.com
[3] [n.d.]. *Frida - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* https://frida.re
[4] [n.d.]. *HTTP ToolKit.* https://httptoolkit.tech/
[5] [n.d.]. *OkHttp.* https://square.github.io/okhttp/
[6] Weidong Fang, Fengrong Li, Yanzan Sun, Lianhai Shan, Shanji Chen, Chao Chen, and Meiju Li. 2016. Information security of PHY layer in wireless networks. *Journal of Sensors* 2016 (2016).
[7] IB Mopari, SG Pukale, and ML Dhore. 2008. Detection and defense against DDoS attack with IP spoofing. In *2008 International Conference on Computing, Communication and Networking*. IEEE, 1–5.
[8] Tongguang Ni, Xiaoqing Gu, Hongyuan Wang, and Yu Li. 2013. Real-time detection of application-layer DDoS attack using time series analysis. *Journal of Control Science and Engineering* 2013 (2013).
[9] K Vishnu. 2010. A new kind of transport layer attack in wireless Ad Hoc Networks. In *2010 IEEE International Conference on Wireless Communications, Networking and Information Security*. 198–201. https://doi.org/10.1109/WCINS.2010.5541733

A FRIDA SNIPPETS IN JAVASCRIPT

A.1 Android Dynamic Library and Symbol Discovery

```
function getAllModules() {
    var modules = Process.enumerateModules();
    const res = [];
    for (var i in modules){
        var module = modules[i];
        let name = module.name;
        if (name.indexOf("target.so") > -1 ){
            name = module.base;
        }
        res.push(name);
    }
    return res;
}
```

```
function getSymbols(moduleName) {
    var symbols = Process.findModuleByName(moduleName).enumerateSymbols();
    const res = [];
    for (var i in symbols) {
        var sym = symbols[i];
        res.push(sym.name);
    }
    return res;
}

function findSymbolModule(symbol) {
    const res = [];
    const modules = getAllModules();
    for (var m of modules) {
        const symbols = getSymbols(m);
        for (var s of symbols) {
            if (s === symbol) {
                res.push(m);
                break;
            }
        }
    }
    return res;
}
console.log(findSymbolModule(symbolName));
```

A.2 C Struct Reading

```
typedef struct {
    int size;
    char* data;
} some_struct;
```

Its fields can be read by:

```
const pointer = "ADDR_TO_STRUCT";
const pSize = Process.pointerSize;

let field1 = pointer;
let field2 = pointer.add(pSize);

let size = field1.readInt();
let data = field2.readPointer().readCString();
```

A.3 DNS Redirection

```
function hookNativeAddr() {
  const tcpSocketFDs = new Map()

  const getaddrinfo = Module.getExportByName("libc.so", "getaddrinfo")
  let new_func = new NativeFunction(getaddrinfo, "int", ["pointer", "pointer", "pointer", "pointer"]);

  var argss = [];
  var preHost;
  Interceptor.attach(getaddrinfo, {
    onEnter(args) {
      console.log("----------------------getaddrinfo----------------------")
      preHost = args[0].readCString();
      console.log("Original Host: " + preHost);
      console.log(args[1].readCString());

      argss = [args[0], args[1], args[2], args[3]];
    },
    onLeave(retval) {
      console.log('Original retval ' + retval.toInt32())

      const host = "ucla.edu";
      if (!preHost.endsWith(host)) {
        console.log(">>>>>>>Going to: " + host);
        let newHost = new NativePointer(argss[0]);
        newHost.writeUtf8String(host);

        let ret = new_func(newHost, argss[1], argss[2], argss[3]);
        retval.replace(ret);

        console.log('New retval ' + retval.toInt32());
      }
      console.log("----------------------\n");
    }
  })
```

```
}

hookNativeAddr();
```

A.4 HTTPS Response Spoofing

```
const httpsUrlConnection = function() {
    const HttpsURLConnection = Java.use("com.android.okhttp.internal.huc.HttpsURLConnectionImpl");
    const ByteArrayInputStream = Java.use("java.io.ByteArrayInputStream");
    const ByteArrayOutputStream = Java.use("java.io.ByteArrayOutputStream");
    const BufferedReader = Java.use("java.io.BufferedReader");
    const InputStreamReader = Java.use("java.io.InputStreamReader");
    const JavaString = Java.use("java.lang.String");

    console.log("-------HttpsURLConnectionImpl")
    HttpsURLConnection.getInputStream.overloads[0].implementation = function() {
        console.log("  -----getInputStream")
        try {
            var methodURL = "";
            var responseHeaders = "";
            var responseBody = "";
            var Connection = this;
            var stream = stream = this.getInputStream.overloads[0].apply(this, arguments);

            var requestURL = Connection.getURL().toString();
            var requestMethod = Connection.getRequestMethod();
            var requestProperties
            methodURL = requestMethod + " " + requestURL;
            if (Connection.getHeaderFields) {
                var Keys = Connection.getHeaderFields().keySet().toArray();
                var Values = Connection.getHeaderFields().values().toArray();
                responseHeaders = "";
                for (var key in Keys) {
                    if (Keys[key] && Keys[key] !== null && Values[key]) {
                     responseHeaders += Keys[key] + ": " + Values[key].toString().replace(/\[/gi, "").replace(/\]/gi, "") + "\n";
                    } else if (Values[key]) {
                        responseHeaders += Values[key].toString().replace(/\[/gi, "").replace(/\]/gi, "") + "\n";
                    }
                }
            }
            console.log("--Https Header:\n" + responseHeaders)

            var retval;
            if (stream) {
                var baos = ByteArrayOutputStream.$new();
                var buffer = -1;

                var BufferedReaderStream = BufferedReader.$new(InputStreamReader.$new(stream));
                while ((buffer =stream.read()) != -1){
                    baos.write(buffer);
                    responseBody += String.fromCharCode(buffer);
                }

                var s2 = JavaString.$new(modifyJson(responseBody));

                BufferedReaderStream.close();
                baos.flush();
                retval = ByteArrayInputStream.$new(s2.getBytes());
            }
            console.log("--Https Body:\n" + responseBody);
            if(retval) return retval;
            return stream;
        } catch (e) {
            console.log(e);
            this.getInputStream.overloads[0].apply(this, arguments);
        }
    }
}
Java.perform(httpsUrlConnection);
```

A.5 HTTPS Certificate Unpinning

```
const unpinning = function() {
    console.log("------------Unpinning Android app--------------");

    try {
        const HttpsURLConnection = Java.use("javax.net.ssl.HttpsURLConnection");
        HttpsURLConnection.setDefaultHostnameVerifier.implementation = function (hostnameVerifier) {
            console.log('  --> Bypassing HttpsURLConnection (setDefaultHostnameVerifier)');
            return;
```

```
    };
    console.log('[+] HttpsURLConnection (setDefaultHostnameVerifier)');
} catch (err) {
    console.log('[ ] HttpsURLConnection (setDefaultHostnameVerifier)');
}
try {
    const HttpsURLConnection = Java.use("javax.net.ssl.HttpsURLConnection");
    HttpsURLConnection.setSSLSocketFactory.implementation = function (SSLSocketFactory) {
        console.log('  --> Bypassing HttpsURLConnection (setSSLSocketFactory)');
        return;
    };
    console.log('[+] HttpsURLConnection (setSSLSocketFactory)');
} catch (err) {
    console.log('[ ] HttpsURLConnection (setSSLSocketFactory)');
}
try {
    const HttpsURLConnection = Java.use("javax.net.ssl.HttpsURLConnection");
    HttpsURLConnection.setHostnameVerifier.implementation = function (hostnameVerifier) {
        console.log('  --> Bypassing HttpsURLConnection (setHostnameVerifier)');
        return;
    };
    console.log('[+] HttpsURLConnection (setHostnameVerifier)');
} catch (err) {
    console.log('[ ] HttpsURLConnection (setHostnameVerifier)');
}

try {
    const X509TrustManager = Java.use('javax.net.ssl.X509TrustManager');
    const SSLContext = Java.use('javax.net.ssl.SSLContext');

    const TrustManager = Java.registerClass({
        name: 'dev.asd.test.TrustManager',
        implements: [X509TrustManager],
        methods: {
            checkClientTrusted: function (chain, authType) { },
            checkServerTrusted: function (chain, authType) { },
            getAcceptedIssuers: function () { return []; }
        }
    });
    const TrustManagers = [TrustManager.$new()];

    const SSLContext_init = SSLContext.init.overload(
        '[Ljavax.net.ssl.KeyManager;', '[Ljavax.net.ssl.TrustManager;', 'java.security.SecureRandom'
    );

    SSLContext_init.implementation = function (keyManager, trustManager, secureRandom) {
        console.log('  --> Bypassing Trustmanager (Android < 7) request');
        SSLContext_init.call(this, keyManager, TrustManagers, secureRandom);
    };
    console.log('[+] SSLContext');
} catch (err) {
    console.log('[ ] SSLContext');
}

try {
    const array_list = Java.use("java.util.ArrayList");
    const TrustManagerImpl = Java.use('com.android.org.conscrypt.TrustManagerImpl');

    TrustManagerImpl.checkTrustedRecursive.implementation = function(a1, a2, a3, a4, a5, a6) {
        console.log('  --> Bypassing TrustManagerImpl checkTrusted ');
        return array_list.$new();
    }

    TrustManagerImpl.verifyChain.implementation = function (untrustedChain, trustAnchorChain, host, clientAuth, ocspData,
    ↪  tlsSctData) {
        console.log('  --> Bypassing TrustManagerImpl verifyChain: ' + host);
        return untrustedChain;
    };
    console.log('[+] TrustManagerImpl');
} catch (err) {
    console.log('[ ] TrustManagerImpl');
}

try {
    const okhttp3_Activity_1 = Java.use('okhttp3.CertificatePinner');
    okhttp3_Activity_1.check.overload('java.lang.String', 'java.util.List').implementation = function (a, b) {
        console.log('  --> Bypassing OkHTTPv3 (list): ' + a);
        return;
    };
    console.log('[+] OkHTTPv3 (list)');
} catch (err) {
```

```
        console.log('[ ] OkHTTPv3 (list)');
    }
    try {
        const okhttp3_Activity_2 = Java.use('okhttp3.CertificatePinner');
       okhttp3_Activity_2.check.overload('java.lang.String', 'java.security.cert.Certificate').implementation = function (a, b) {
            console.log('  --> Bypassing OkHTTPv3 (cert): ' + a);
            return;
        };
        console.log('[+] OkHTTPv3 (cert)');
    } catch (err) {
        console.log('[ ] OkHTTPv3 (cert)');
    }
    try {
        const okhttp3_Activity_3 = Java.use('okhttp3.CertificatePinner');
        okhttp3_Activity_3.check.overload('java.lang.String', '[Ljava.security.cert.Certificate;').implementation = function (a,
        ↪ b) {
            console.log('  --> Bypassing OkHTTPv3 (cert array): ' + a);
            return;
        };
        console.log('[+] OkHTTPv3 (cert array)');
    } catch (err) {
        console.log('[ ] OkHTTPv3 (cert array)');
    }
    try {
        const okhttp3_Activity_4 = Java.use('okhttp3.CertificatePinner');
        okhttp3_Activity_4['check$okhttp'].implementation = function (a, b) {
            console.log('  --> Bypassing OkHTTPv3 ($okhttp): ' + a);
            return;
        };
        console.log('[+] OkHTTPv3 ($okhttp)');
    } catch (err) {
        console.log('[ ] OkHTTPv3 ($okhttp)');
    }
}
Java.perform(unpinning);
```