# Creating a powerful Android app context protector

With frida and r2

Giovanni - iGio90 - Rocca

@Ultrapowa | @Defunct | @Bha | @Pepper | @VeronicaPabloOsorio

# $whoami

- 30 yrs old, daddy
- full stack developer @Overwolf
- Passionate reverse engineer (focussed on ARM and nix based os),
  time to time consultant for Supercell, NowSecure and various companies which contact me for internal application audits
- Creator of various cracking and dynamic analysis frameworks, libraries and SDK
  https://github.com/iGio90
- Co-Founder of secure return community

The king of Milan xD
-> source: http://git-awards.com/users/search?login=igio90

# What is this talk about

Quick introduction on how we used Frida to protect our flag from attackers in a competition and how we extended that work with more low level protection techniques, ending up with rising interests from various companies.

Creating a framework that takes a compiled Android app package as input and output a protected package ready for distribution - shipped with a fresh frida based protection.

Credit for work and researches goes to:
**@iGio90 | @Ultrapowa | @Defunct | @Bha | @Pepper | @VeronicaPabloOsorio**

# What is this talk about

## Disclaimer

We believe that the information available in these slides will allow people to replicate our work, which we are not intended to maintain anyway. We believe that this information is “powerful” and can be used to achieve bad stuff, but we also believe that the amount of people that will be eventually able to “fight” something like that using the same knowledge, will be way higher.

Credit for work and researches goes to:

**@iGio90 | @Ultrapowa | @Defunct | @Bha | @Pepper | @VeronicaPabloOsorio**

# Pre-consideration

- All of the content described can be translated to C lang without the real need of Frida.
  We believe that Frida adds an additional security layer and simplify some of the operations.

- Performances obviously matter.
  We tested the solution with one of the most played mobile game, with multiple libraries and heavy code base

- I'm not sure Google would eventually allow the distribution of an application protected with a similar solutions, but i've saw so much bad stuff (specially on TikTok) to believe that their automated security audits are nowhere close to detect such mechanisms (or, in a more real world, they don't care at all?).

## Simple concept:

# Using frida to protect the userspace

- System.loadLibrary(“frida-gadget”);
- Gadget -> Javascript agent with any colorful protection logic

# Focus / Objective

- Provide a solution which attempts to f*@k reverse engineers mind rather than building the “uncrackable” protector
- The time impact must be in term of milliseconds for low end hardware

# Pros/cons of using frida for the purpose

- Accessing low level api / libc with ease
- Using trampolines to protect the protector
- It simplifies multi process anti-debugging solutions
- Frida itself is a huge protection layer

# Pros/cons of using frida for the purpose

~~Calling native api / execute trampoline callbacks in the javascript engine is super slow compared to a native implementation~~

CModule crossed out the issue.

# Hiding a flag with frida (Gadgets / Agent)

- Create the typescript agent
- Obfuscate fields and names with any open source javascript obfuscator
- Hardcode the agent into the gadgets source
- Modify the gadgets code to load the hardcoded agent (as a standard String)

# Hiding a flag with frida (Gadgets / Agent)

Considering mandatory rules of the competition (i.e correct flag must be present in memory at least for a single instruction execution and there must be an input which trigger the correct flag creation, just like protecting sensitive data during a normal app execution.)

- Hook the functions involved with our target creation (what we want to protect)
- Perform additional security tricks and checks
- Alter the code logic (i.e shuffle the result with a different seed) in the hook callback
- Return to app

# Layers

- Debugging inside the duktape engine is painful
- Hooking our target methods will break and/or alter application flow
- Have fun

# HugShell

- Decompile the app with apktool
- Apply protections
- Recompile the app with apktool

# Apply protections

- Native libraries compression
- Gadget and agent encryption
- Elf headers scrumbler
- Encrypt executable and symbol sections in native libraries
- Replace common libc calls to dynamically allocated code which ends up in SVC
- Strip any non-needed symbol

Other “default-enabled” protections will be described later

```
class HugShell {
    static protect(srcPackagePath, destPackagePath, options) {
        // assign default options
        options = {
            'compress-libraries': true,
            'gadget-agent-encryption': true,
            'elf-headers': true,
            'sections-encryption': true,
            'libc-trampolines': true,
            'strip-symbols': true,
            'self-sign': true,
            methods: {},
            ...options,
        };
```

# HugShell - Work cycle

```
// decode apk
HugShell.decode(srcPackagePath, packageWorkingDir);
```

- Decompile the APK using straight **apktool d app-name.apk**
- Prepare temporary and output directories

# HugShell - Work cycle

```
// copy agent and run npm install
HugShell.importAgent(agentSourceDir);
```

We used frida-compile to build our agent, here we run an install in the agent source

# HugShell - Work cycle

```
// apply modules protections (backend-ed by r2 <3)
HugShell.applyModuleProtections(options, packageWorkingDir, agentSourceDir);
```

We used **r2** to backend a static analysis on Native modules automated with **r2pipe**.
We used it to:

- get offsets of common libc calls
- user defined offsets for custom method protections
- dlopen calls
- Segments offsets and length for encryption

We planned (and I did some positive research on it) to strip out native code (ARM only) by matching 24+ bytes patterns of instructions and abuse that space to create some sick trampolines (EA Games like)

# HugShell - Work cycle

```
// patch application libraries elf headers with 32 bytes of crap
if (options['elf-headers']) {
    HugShell.patchHeaders(packageWorkingDir, agentSourceDir);
}
```

Here we collect the ELF headers of natives and store them somewhere accessible by the agent (i.e, hardcoded  in code or read-able from somewhere else).
We replace original headers with 32 bytes of craps.
This breaks any decompilers. It is still possible to read through assembly code by scripting, considering that:

- most common libc calls are nopped out in the libraries and replaced in runtime after the agent has been loaded
- user protected methods are a bunch of random instructions with no sense, real code is moved to agent mapped space
- you won't have symbols, xrefs or any feature from decompilers until you fix headers

# HugShell - Work cycle

```
// compile the agent
HugShell.buildAgent(agentSourceDir, agentBuildDir);

// copy compiled agent from build dir to work dir
HugShell.copyAgent(agentBuildDir, packageWorkingDir);

// copy compiled gadget from build dir to work dir
HugShell.copyGadget(packageWorkingDir);
```

# HugShell - Work cycle

```
// copy our shield loader smali, patch smali launcher activities
HugShell.injectGadgetInSmali(packageWorkingDir, loaderClassName);
```

Here we patch the Application class, or, if not present, we patch any Launcher Activity declared in the AndroidManifest to initialize an instance of our java class. That class is in charge for decrypting the gadgets, store the SO lib somewhere in the app context, load it through **System.load()** call which allows loading  from custom app context path, and remove it.

# HugShell - Work cycle

```
// first agent xor. mandatory as gadget will default dexor agent with an hardcoded placeholder key
HugShell.xorAgent(packageWorkingDir);
```

We run a first xor on the agent with an hardcoded static key.

Better encryptions can be implemented, we went for something fast (shuffles, xor) as we are going to run another cycle later with another key.

This key is hardcoded in the gadgets. During our work on the protector, we prefered to keep the agent outside the gadgets as this would force us to compile the gadgets for each test

# HugShell - Work cycle

```
// first gadget xor and second agent xor.
// both mandatory as smali default dexor both with an hardcoded placeholder key
if (options['gadget-agent-encryption']) {
    HugShell.encryptGadgetAgent(packageWorkingDir, loaderClassName);
}
```

Here we xor the gadgets and the agent again.

The key changes at each run of the protector as we can quickly replace the parts of the key in our smali class loader. A little protection to this key is added as well. We wanted to keep those layers as small/easy as possible to avoid overengineering and to give the false illusion that everything is very easy.

From an attacker perspective, reverse engineering until the load of the gadgets would take a couple of hours.

# HugShell - Work cycle

```
// iterate over smali files, find System.loadLibrary call and wrap it with a zlib decompress + load
if (options['compress-libraries']) {
    HugShell.findAndCompressLibraries(packageWorkingDir, loaderClassName);
}
```

Among the main protections, we were working on other small “features”, which add anyway lot of values when all-together-in-place. One of that feature was to wrap all the loadLibrary calls with a custom method on our injected java class. That method was basically in charge of doing the same as it did for the gadgets loading but with compression instead of encryption.

# HugShell - Work cycle

```
// build package
HugShell.buildPackage(packageWorkingDir, destPackagePath);
```

# HugShell - Agent

```
const Linker = require('./linker');
const Protector = require('./protector');
```

The 2 main modules of the agent

- Linker
- Protector

# HugShell - Agent - Linker

Abusing something in the linker adds several layers of protection

- It is the lowest zone of os involved things in charge of loading modules
  (That's what it's built for. Linker. Linking stuff.)
- It cross out the issue of protecting modules which are loaded from another native module using dlopen rather than the java call System.loadLibrary (if you question yourself, why should I run my stuff in the linker rather then keeping my self at an higher level)
- An attacker can't go any lower to debug (within the os)

# HugShell - Agent - Linker

```
        const m = new CModule(`
int xxor(char* buf, int size, char* key, int keySize) {
  int i = 0;
  while (i < size) {
    buf[i] = buf[i] ^ key[i % keySize];
    i++;
  }
  return 0;
}
`);
        const xxor = new NativeFunction(m.xxor, 'int', ['pointer', 'int', 'pointer', 'int']);
```

Setup CModules for any operation. Even a xor sees decrements - js: 0.0X cmodule: 0.0000000X

# HugShell - Agent - Linker

```
Interceptor.attach(Linker.getSymbol( name: 'do_dlopen').address, callbacksOrProbe: {
```

Setup an hook to do_dlopen

Allows to retrieve module name from register 0.
Aka we are aware that the app is loading a native library that could be (or not) protected by us.

# HugShell - Agent - Linker

```
const readElfInterceptor = Interceptor.attach(Linker.getSymbol( name: 'ReadElfHeader').address,
    onEnter() {
        Logger.logDebug('[linker] reading headers');
        let headerRead = 0;
        let headerSize = 0;
        const preadInterceptor = Interceptor.attach(Linker.getSymbol( name: 'dl_pread64').address,
```

We setup 2 additional hooks to **ReadElfHeader** and **dl_pread64** (note that all of this has been tested across multiple os versions and multiple arch (x86, arm 32/64 - android 5-9, 10 wasn't out yet but we believe in no major changes to the linker))

Those 2 hooks basically allow us to know when the Linker is reading the headers of our target module, giving us the last possible window to make the swap with the correct headers.

# HugShell - Agent - Linker

Superfun fact:

Once the whole **do_dlopen** trees is completed, the real ELF headers are not needed anymore by the app (this can be terrible wrong depending on the behavior of the protected target).

So we swapped again the real headers with the crap data right after **do_dlopen** and with big surprise it wasn't only working superfine, but it was also breaking any tools attempting to access the module space, as any tools relays on reading the elf headers to perform some initializations (symbols load etc).

It turns out that neither frida, nor other popular debugging/dynamic instrumentation tools, did successfully dump a line of assembly from the protected module space. We didn't perform additional attacks trying to avoid this issue, might be easy or not.

# HugShell - Agent - Linker

```
if (ModuleSections
    && typeof ModuleSections[Android.arch] !== 'undefined'
    && typeof ModuleSections[Android.arch][doDlopen.currentLibName] !== 'undefined'
) {
    let linkImage = Linker.getSymbol( name: 'prelink_image');
    if (linkImage === null) {
        linkImage = Linker.getSymbol( name: 'soinfo_link_image');
    }
    if (linkImage !== null) {
        const linkImageInterceptor = Interceptor.attach(linkImage.address,  callbacksOrProbe: {
```

One additional hook to **prelink_image** (if existing - android 7 or 8 made some changes) or **soinfo_link_image** once again to be in the lowest possible place to perform a secure segment decryption.

# HugShell - Agent - Linker

Kind of a fact:

Debugging any of this in a higher level, has highchance to break what is expected from the agent. Hooking in the linker to perform debugging, has high chance to replace the hooks performing the operations. All of this sounds ease knowing how it is acting, but from the perspective of an attacker with zero knowledge we believe this would be an hard challenge.

# HugShell - Agent - Protector

Syscall wrappers setup

```
module.exports = {
    socket: new NativeCallback( func: (domain, type, protocol) => (
        libc.syscall3(libc.SYSNO_SOCKET, ptr(domain), ptr(type), ptr(protocol)).toInt32()
    ), retType: 'int', argTypes: ['int', 'int', 'int']),
    sendto: new NativeCallback( func: (sockfd, buf, len, flags, dest_addr, addr_len) => (
        libc.syscall6(libc.SYSNO_SENDTO, ptr(sockfd), ptr(buf), ptr(len), ptr(flags), ptr(dest_addr), ptr(addr_len)).toInt32()
    ), retType: 'int', argTypes: ['int', 'pointer', 'int', 'int', 'pointer', 'int']),
    recvfrom: new NativeCallback( func: (sockfd, buf, len, flags, dest_addr, addr_len) => (
        libc.syscall6(libc.SYSNO_RECVFROM, ptr(sockfd), ptr(buf), ptr(len), ptr(flags), ptr(dest_addr), ptr(addr_len)).toInt32()
    ), retType: 'int', argTypes: ['int', 'pointer', 'int', 'int', 'pointer', 'int']),
    close: new NativeCallback( func: (fd) => (
        libc.syscall1(libc.SYSNO_CLOSE, ptr(fd)).toInt32()
    ), retType: 'int', argTypes: ['int']),
    connect: new NativeCallback( func: (sockfd, addr, addrlen) => (
        libc.syscall3(libc.SYSNO_CONNECT, ptr(sockfd), ptr(addr), ptr(addrlen)).toInt32()
    ), retType: 'int', argTypes: ['int', 'pointer', 'int']),
    openat: new NativeCallback( func: (dirfd, path, flags, mode) => (
        libc.syscall4(libc.SYSNO_OPENAT, ptr(dirfd), ptr(path), ptr(flags), ptr(mode)).toInt32()
    ), retType: 'int', argTypes: ['int', 'pointer', 'int', 'int']),
    read: new NativeCallback( func: (fd, buf, size) => (
        libc.syscall3(libc.SYSNO_READ, ptr(fd), ptr(buf), ptr(size)).toInt32()
    ), retType: 'int', argTypes: ['int', 'pointer', 'int']),
};
```

```
const syscalls = {
    'armeabi-v7a': {
        SYSNO_SOCKET: 281,
        SYSNO_SENDTO: 290,
        SYSNO_RECVFROM: 292,
        SYSNO_CLOSE: 6,
        SYSNO_CONNECT: 283,
        SYSNO_OPENAT: 322,
        SYSNO_READ: 3,
    },
    'arm64-v8a': {
        SYSNO_SOCKET: 198,
        SYSNO_SENDTO: 206,
        SYSNO_RECVFROM: 207,
        SYSNO_CLOSE: 57,
        SYSNO_CONNECT: 203,
        SYSNO_OPENAT: 56,
        SYSNO_READ: 63,
    },
    x86: {
        SYSNO_SOCKET: 359,
        SYSNO_SENDTO: 369,
        SYSNO_RECVFROM: 371,
        SYSNO_CLOSE: 6,
        SYSNO_CONNECT: 362,
        SYSNO_OPENAT: 295,
        SYSNO_READ: 3,
    },
    x86_64: {
        SYSNO_SOCKET: 41,
        SYSNO_SENDTO: 44,
        SYSNO_RECVFROM: 45,
        SYSNO_CLOSE: 3,
        SYSNO_CONNECT: 42,
        SYSNO_OPENAT: 257,
        SYSNO_READ: 0,
    },
};

const syscallPtr = Module.findExportByName( moduleName: 'libc.so', exportName: 'syscall');
```

```
module.exports = {
    ...syscalls[Android.arch],
    syscall0: new NativeFunction(syscallPtr, 'pointer', ['int']),
    syscall1: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer']),
    syscall2: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer', 'pointer']),
    syscall3: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer', 'pointer', 'pointer']),
    syscall4: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer', 'pointer', 'pointer', 'pointer']),
    syscall5: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer', 'pointer', 'pointer', 'pointer', 'pointer']),
    syscall6: new NativeFunction(syscallPtr, 'pointer', ['int', 'pointer', 'pointer', 'pointer', 'pointer', 'pointer', 'pointer']),
};
```

# HugShell - Agent - Protector

Writing assembly instructions to the offsets we previously scanned and nopped with r2 to jump in our syscall functions wrapper pointers and return the result.

```
static createTrampoline(targetAddr, wrapper) {
    if (Android.arch === 'x86' || Android.arch === 'x86_64') {
        const writer = new X86Writer(targetAddr);
        writer.putJmpAddress(wrapper);
        writer.flush();
        writer.dispose();
    } else if (Android.arch === 'armeabi-v7a') {
        const writer = new ArmWriter(targetAddr);
        writer.putLdrRegAddress( reg: 'r12', wrapper);
        writer.putMovRegReg('pc', 'r12');
        writer.flush();
        writer.dispose();
    } else if (Android.arch === 'arm64-v8a') {
        const writer = new Arm64Writer(targetAddr);
        writer.putLdrRegAddress( reg: 'x16', wrapper);
        writer.putBrReg( reg: 'x16');
        writer.flush();
        writer.dispose();
    }
}
```

# HugShell - Agent - Protector

We were also testing a loop on a separate thread, which would anyway needs  to be ported with either CModule or an additional native library to achieve better performances.

The loop was basically check for ptracer. We were also testing a crc cycle on protected modules.

Note in the code some tricks that “should” technically prevent lower level break or debugging of the loop.

```
static loop() {
    Protector.asyncProtectionLoopState = 1;
    while (true) {
        Protector.asyncProtectionLoopRun = Date.now();
        Protector.checkPtracer();
        Protector.asyncProtectionLoopState = 2;
        Thread.sleep( delay: 3);
    }
    Protector.asyncProtectionLoopState = 0; // eslint-disable-line no-unreachable
}

static loopCheck() {
    while (true) {
        const now = Date.now();
        if (now - Protector.asyncProtectionLoopRun > 10 * 1000) {
            Logger.logDebug('[loopCheck] time based attack detected');
            Process.crashUi();
        }
        Thread.sleep( delay: 10);
    }
}
```

# HugShell - Agent - Protector

Last, but not least

Find colorful ways to crash the process.

```
Process.crashUi = () => {
    memcpy.attach( callbacks: (args) => {
        const p = ptr(args[0]);
        args[0] = ptr(args[1]);
        args[1] = p;
    });
};
```

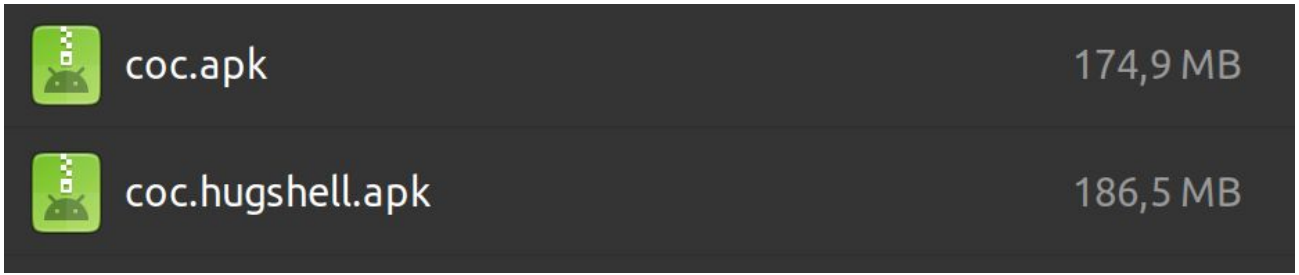
# HugShell

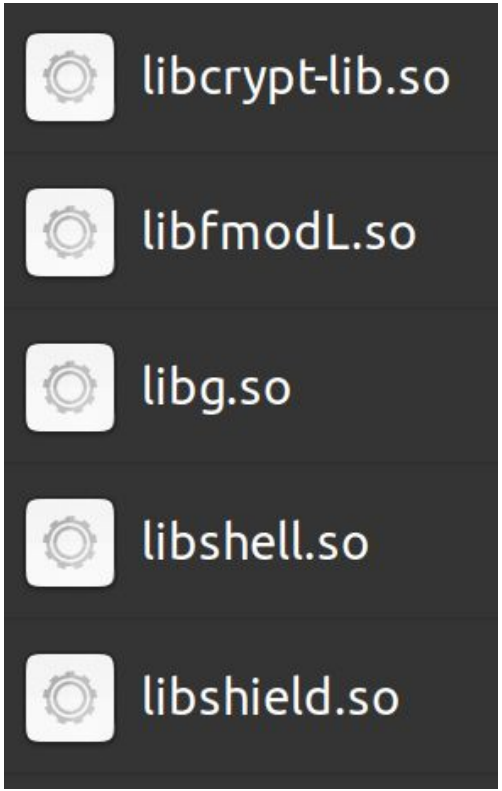
Additional misc

- Reduced gadget size from 20 to 4mb by stripping v8 and lot of non-used code. *4arch = -84mb
- Java field obfuscation (Obfuscapk as an awesome references)

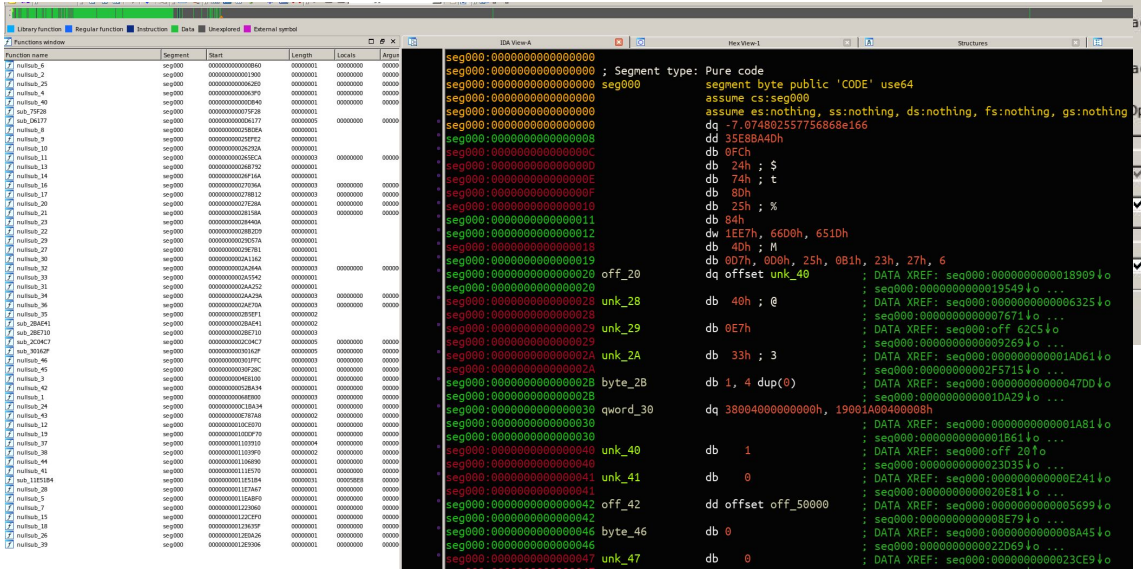
- Native method protections is a WIP research ended up with 4/5 fails already. It will get done.

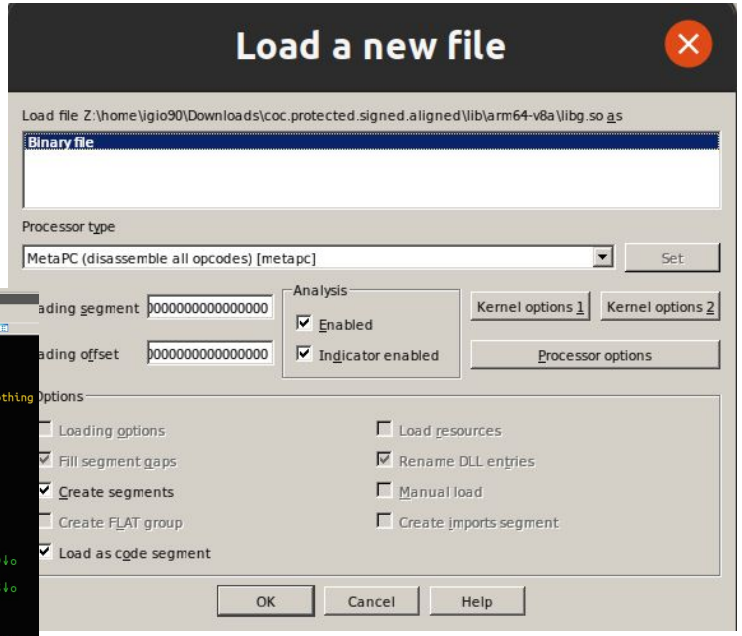
# HugShell result



# HugShell result





# Thanks

Creating a powerful protector with frida

Giovanni - iGio90 - Rocca

@Ultrapowa | @Defunct | @Bha | @Pepper | @VeronicaPabloOsorio