Sleight of ARM: Demystifying Intel Houdini

Brian Hong
@im_eengineer
Brian S. Hong (@im_eningeer)

- Hardware Enthusiast
- Forward Reverse Engineer
- Like to reverse low-level stuff and break embedded systems
- Android Penetration Testing
- Security Consultant @ nccgroup
- Cooper Union Electrical Engineering
Introduction — Android NDK

- Android is the operating system powering 70%\(^1\) of the mobile devices
- Android supports application development in Java and Kotlin, and additionally in native languages such as C and C++ through the Native Development Kit (NDK)
- ARM is the main hardware platform for Android, with official support for x86 introduced in later versions – Android Lollipop (2014)
  - NDK r6 (2011) added support for x86
  - NDK r10 (2014) added support for 64 bit ABIs, including x86_64
- There is also out-of-tree support for Android on x86
  - Android-x86 (2011)\(^2\)

\(^1\) https://gs.statcounter.com/os-market-share/mobile/worldwide
\(^2\) https://www.android-x86.org/
Introduction — Android on x86

- Two main kinds of x86 devices running Android (neither of them are phones)
  - x86 Chromebooks
  - Commercial Android emulators on x86 hosts
- x86 support is generally lacking across apps
  - ARM is the primary target platform
  - If shipping native code, the Play Store only requires ARM builds
  - Few developers end up shipping x86 binaries for their APKs, but many apps have native code
- So then how are x86 Android devices supposed to support popular apps (optimized with native ARM code)?
Houdini — What is it?

• Intel’s proprietary dynamic binary translator from ARM to x86
  • Co-created by Google for Android
  • Enables ARM native applications to run on x86 based platforms

• A black box shrouded in mystery
  • Little mention of it on Intel’s websites, seemingly not a public-facing product
  • No public documentation
  • Several vendors may be obfuscating their use of Houdini?

• There are three variants:
  • 32-bit x86 implementing 32-bit ARM
  • 64-bit x86 implementing 32-bit ARM
  • 64-bit x86 implementing 64-bit ARM
Houdini — Where’s it used?

- Physical hardware
  - x86-based mobile phones (e.g. Zenfone 2)
  - x86 Chromebooks
    - This is how we got it
- Commercial Android Emulators
  - BlueStacks
  - NOX
- Android-x86 Project
Interpreted emulator

- Essentially a while loop around a switch (but actually more like a state machine)
- Reads ARM opcodes and produces corresponding behavior in x86
  - Doesn’t JIT; no x86 instructions produced at runtime

Two components

- houdini: interpreter used to run executable binaries
- libhoudini: loadable shared object (x86); used to load and link ARM libraries
Runs ARM executable binaries (static and dynamic)

- Uses dynamic libraries precompiled for ARM+Android from:
  - /system/lib/arm
  - /system/vendor/lib/arm

```bash
/data/media/0/Download/arm-bin # uname -a
Linux localhost 4.14.180-15210-gd513939c7dc9 #1 SMP PREEMPT Tue Jul 28 01:21:26 PDT 2020 i686
/data/media/0/Download/arm-bin # file hello_static
hello_static: ELF executable, 32-bit LSB arm, static, BuildID=441f7ee9bafadb1b141d27b82b28569
 stripped
/data/media/0/Download/arm-bin #
/data/media/0/Download/arm-bin # ./hello_static
Hello world!
```

- Loaded in by the Linux kernel binfmt_misc feature
binfmt_misc (Miscellaneous Binary Format) is a capability of the Linux kernel which allows arbitrary executable file formats to be recognized and passed to certain user space applications, such as emulators and virtual machines. It is one of a number of binary format handlers in the kernel that are involved in preparing a user-space program to run.¹

```
localhost ~ # cat /proc/sys/fs/binfmt_misc/arm_exe
enabled
interpreter /system/bin/houdini
flags: P
offset 0
magic 7f454c4601010000000000000000000020028

localhost ~ # cat /proc/sys/fs/binfmt_misc/arm_dyn
enabled
interpreter /system/bin/houdini
flags: P
offset 0
magic 7f454c4601010000000000000000000030028
```

```
./hello -> /system/bin/houdini ./hello
```

¹ https://en.wikipedia.org/wiki/Binfmt_misc
libhoudini.so

- Is a shared object (x86)

```bash
:/ # file /vendor/lib/libhoudini.so
/vendor/lib/libhoudini.so: ELF shared object, 32-bit LSB 386
```

- Loads in ARM shared objects
- Mainly designed to be used with Android NativeBridge to run ARM native code
Android NativeBridge

- Main interface from Android to libhoudini
- Part of the Android Runtime (ART)
- Supports running native libraries in different processor architectures

Android NativeBridge — Initialization

- Initialized on boot by Android Runtime (ART)
- NativeBridge reads system property ro.dalvik.vm.native.bridge
  - Disabled if set to "0"
  - Otherwise, it provides the name of the library file to be loaded with NativeBridge (e.g. "libhoudini.so")
  - Android-x86 project uses "libnb.so" instead, which is a shim that loads libhoudini

- NativeBridge defines interface with callbacks
  - NativeBridgeRuntimeCallbacks
  - NativeBridgeCallbacks
The JNI is an FFI for calling between JVM code (e.g. Java) and native code (e.g. C/C++). Java native methods are mapped to native symbols. The native functions receive a `JNIEnv*` from the JVM, which is a bag of function pointers providing a low-level Java/JVM reflection API, including object allocation, class lookups, and method invocations. It also provides a type mapping between Java primitives and C types.

```c
typedef const struct JNINativeInterface* JNIEnv;
struct JNINativeInterface {
    jint (*GetVersion)(JNIEnv *);
    jclass (*DefineClass)(JNIEnv*, const char*...);
    jclass (*FindClass)(JNIEnv*, const char*);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jclass (*FindClass)(JNIEnv*, const char*);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEnv*, jclass, jmethodID...);
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...);
    jobject (*AllocObject)(JNIEnv*, jclass);
    jobject (*NewObject)(JNIEn...
NativeBridgeRuntimeCallbacks provide a way for native methods to call JNI native functions.

```c
// Runtime interfaces to native bridge.
struct NativeBridgeRuntimeCallbacks {
  // Get shorty of a Java method.
  const char* (*getMethodShorty)(JNIEnv* env, jmethodID mid);

  // Get number of native methods for specified class.
  uint32_t (*getNativeMethodCount)(JNIEnv* env, jclass clazz);

  // Get at most 'method_count' native methods
  // for specified class.
  uint32_t (*getNativeMethods)(JNIEnv* env, jclass clazz,
                                JNINativeMethod* methods, uint32_t method_count);
};
source
```

source

1 https://android.googlesource.com/platform/art/+master/runtime/native_bridge_art_interface.cc
NativeBridge can interact with libhoudini via
NativeBridgeCallbacks

Fetched from libhoudini via symbol
NativeBridgeItf

- initialize()
- loadLibrary() "dlopen()"
- getTrampoline() "dlsym()"

// Native bridge interfaces to runtime.

```
struct NativeBridgeCallbacks {
    uint32_t version;
    bool (*initialize)(const NativeBridgeRuntimeCallbacks* runtime_cbs,
                        const char* private_dir, const char* instruction_set);
    void* (*loadLibrary)(const char* libpath, int flag);
    void* (*getTrampoline)(void* handle, const char* name,
                            const char* shorty, uint32_t len);

    ...}

int (*unloadLibrary)(void* handle);
void* (*loadLibraryExt)(const char* libpath, int flag,
                        native_bridge_namespace_t* ns);
```

source¹

¹ https://android.googlesource.com/platform/art/+/master/runtime/native_bridge_art_interface.cc
NativeBridge — Libhoudini

$ objdump -T libhoudini.so
libhoudini.so: file format elf32-i386

DYNAMIC SYMBOL TABLE:
...
004f8854 g DO .data 0000003c Base NativeBridgeItf
NativeBridge — Summary

- dlopen(libhoudini.so)
- dlsym(NativeBridgeItf)
- initialize()

- loadLibrary() "dlopen()"
- getTrampoline() "dlsym()"

- Houdini provides a ARM version of JNIEnv
  - Handled via trap instructions
Houdini Emulation — Memory

- Dual architecture userland (separate ARM binaries; e.g. libc, etc.)
- Shared virtual address space
- Real world view of memory
- Maintains a separate allocation for ARM stack
Houdini Emulator — Execution

- State machine (switch inside while loop), fetch/decode/dispatch shown below

```
MOV EBX, dword ptr [EDI]; read PC (fetch instruction)
LEA EAX, [EDI + 0x8]; read PC+8
MOV dword ptr [ESI + 0x19c], EAX; store it somewhere
MOV EAX, EBX; instruction now in EAX & EBX
MOV dword ptr [ESI + 0x8f0], EDI; get condition code (bits 28-31)
ADD EDI, 0x4; check condition code
CMP EAX, 0xe; jump if condition code != OxE (always)
JNZ LAB_0030fe10ad; instruction now in EAX & EBX
ADD ESP, 0x8; push processor_state struct as argument
MOV FAX, EBX; bits 4-7
PUSH ESI; push instruction as argument
PUSH EBX; mask bit 4-7
PUSH EBX; mask bits 20-27
MOV EDX, dword ptr [EBP + EAX*0x4]; combine the above fields: instr[27:20]_instr[7:4]
CALL EDX; get offset into instruction handler table
```
Instruction bits 27-20 concatenated with bits 7-4 is used as the offset into the table.

```c
uint32_t instruction = memory[state.pc];
uint8_t condition_code = instruction >> 24;
if(condition_code != 0x0E) goto 0x3100AD;
uint32_t offset =
    (((instruction >> 16) & 0xFF0) + \[20:27\]
    ((instruction >> 4) & 0x00F)); \[4:7\]

void **instruction_table = 0x4BB9C0;
int (*instruction_handler)(uint32_t, struct proc_state*);

instruction_handler = instruction_table[offset];
instruction_handler(instruction, state);
```
int instr_mov_1(uint instr, proc_state *state)
{
    int iVar1;
    byte bVar2;
    uint uVar3;
    uint Rd;
    uint newPC;

    Rd = (Instr & 0xffff) >> 16;
    if (Rd == 0x0f) {
        s_000559(state);
    }
    uVar3 = (Instr & 0xffff) >> 8;
    if (uVar3 == 0x0f) {
        s_000559(state);
    }
    if (!(Instr & 0xf0)) {
        s_000559(state);
    }
    bVar2 = (byte)state->reg[uVar3];
    uVar3 = state->reg[Instr & 0x0f] << (bVar2 & 0x0f);
    if (iVar1 < bVar2) {
        uVar3 = 0;
        state->reg[Rd] = uVar3;
    }
    if (Rd == 0x0f) {
        newPC = state->reg[0x1f];
        if (state->isThumb == 0) {
            branch_something();
        } else {
            state->ldrstr = 0x11;
            state->reg[0x1f] = newPC & 0xffffffff;
        }
    }
    if ((newPC == 0xfffffff)) {  // (newPC == 0xfffffff)
        iVar1 = (*DAT_0088774) (DAT_0088775);
        FUN_003555f0/*(undefined4 *(*(int *))(iVar1 + 8) + 0x730));
    }
}
return 0x086;
Houdini Emulator — Processor State

- Stores ARM registers, as well as other processor states

/* Processor state of libhoudini's emulated ARM */
struct proc_state {
    unsigned int   reg[16];  /* Register values for r0, r1, r2... */
    unsigned char  unk[300]; /* Unknown fields */
    unsigned int   isThumb;  /* Whether in thumb mode or not */
    unsigned int   svcNumber; /* Pending SVC call number */
    unsigned char  unk2[40]; /* Unknown fields */
    unsigned int   pc8;      /* PC + 8 */
    unsigned int   ldrstr;   /* ?? (used for ldr/str instructions) */
    unsigned char  unk3[84]; /* Unknown fields */
};

- ARM registers can be read/written from both ARM and x86
ARM syscalls are handled by userland x86 code that issues x86 syscalls
Houdini Emulator — fork(2)/clone(2)

- Intercepted and reimplemented by Houdini
- Houdini clones the process
- The child process handles the child fork/clone logic
- The parent process handles the fork/clone logic
- clone(2) child_stack not passed to the kernel
- Instead an empty RWX page is passed as child_stack
Java architecture checking
- `System.getProperty("os.arch");`
- `/proc/cpuinfo`

Memory mapping checking
- `/proc/self/maps`
- Dual x86/ARM shared libraries

Detection from noisy to quiet
The best implementation is one that issues no otherwise discernable syscalls
- JNIEnv magic pointer detection

Houdini hides these
`System.getProperty("os.arch") -> armv7l`

```
$ cat /proc/cpuinfo
Processor : ARMv8 processor rev 1 (aarch64)
processor : 0
processor : 1
BogoMIPS : 24.00
Features : neon vfp half thumb fastmult edsp vfpv3 vfpv4 idiva idivt tls aes sha1 sha2 crc32
CPU implementer : 0x4e
CPU architecture: 8
CPU variant : 0x02
CPU part : 0x000
CPU revision : 1

Hardware : placeholder
Revision : 0000
Serial : 0000000000000000
```
Houdini Emulator — Escape to x86

- mprotect(2) + overwrite code
  - Not subtle
- x86 stack manipulation
  - Find and clobber x86 stack with ROP payloads
Multiple RWX

- We can write x86 code to these pages and jump to it
- Shared memory, which means we can write code from either x86/ARM

```
00008000-0000a000 rw-p [anon:Mem_0x10000002]
0e094000-100000000 rwxp [anon:Mem_0x20000000]
10000000-10003000 rw-p [anon:Mem_0x10002002]
10003000-10004000 ---p [anon:Mem_0x10002002]
10004000-10015000 rw-p [anon:Mem_0x10002002]
10015000-10016000 ---p [anon:Mem_0x10002002]
... 
10128000-120000000 rw-p [anon:Mem_0x10002002]
12000000-121000000 rwxp [anon:Mem_0x10001000]
12100000-121220000 rw-p [anon:Mem_0x10001000]
1215a000-121930000 rw-p [anon:Mem_0x10001000]
ca6e8000-ca6e90000 ---p [anon:Mem_0x10000004]
ca6e9000-caae80000 rw-p [anon:Mem_0x10000004]
caae8000-caae90000 ---p [anon:Mem_0x10000004]
caae9000-cabe80000 rw-p [anon:Mem_0x10000004]
... 
```

ARM JNIEnv

ARM stack
Houdini ignores the execute bit entirely

- ARM libraries are loaded without the execute bit on their pages
- No DEP/NX\(^1\) for ARM
- Trivial to abuse (write to anywhere writable, and jump/return to it)

\(^1\) https://en.wikipedia.org/wiki/NX_bit
$ cat nx-stack.c
#include<stdio.h>

int main(){
    unsigned int code[512] = {0};

    code[0] = 0xE2800001; // add r0, r0, #1
    code[1] = 0xE12FFF1E; // bx lr

    printf("code(1) returned: %d\n", ((int (*)(int))code)(1)); // Normally, this causes a segfault
    printf("code(5) returned: %d\n", ((int (*)(int))code)(5));
}

$ arm-linux-gnueabi-gcc nx-stack.c -static -Wl,-z,noexecstack -o nx-stack-static
$ file nx-stack-static
nx-stack-static: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked
7323f32a36, for GNU/Linux 3.2.0, not stripped
$ ./nx-stack-static
  code (1) returned: 2
  code (5) returned: 6
DEMOS
Libhoudini-aware Malware

- App stores and security researchers often run apps in sandboxed environments to check for malicious behaviors
- Mainly 3 different environments for running/analyzing apps
  - Real ARM devices
  - Fully virtualized ARM environment (like QEMU)
  - x86 Android emulators (VMs)
- Apps that express different behaviors depending on which environment it is running on can, for example, be benign during analysis but malicious otherwise
  - Harder to detect
  - Inconsistent behavior is harder to analyze
Using one of the detection methods discussed earlier, we can write JNI-loaded native Android code that does different things based on whether or not it is running through libhoudini

- x86 Android emulator VMs, such as ones based on Android-x86, may use libhoudini for ARM compatibility
  - This is one possible approach used by app stores, so any form of fingerprinting can become a problem\(^1\)
  - If you know that your apps are only going to be analyzed in such environments, you could key malicious behaviors to the lack of libhoudini

\(^1\) Dissecting the Android Bouncer (Oberheide, J., & Miller, C. (2012, June). SummerCON, Brooklyn, New York)
Conversely, a malicious app could do bad things only when it detects the presence of libhoudini, then abuse libhoudini to further obfuscate itself

- For example, while we don’t know what the Play Store actually uses these days, its automatic app testing did not appear to run ARM APKs on x86 with libhoudini
Recommendations to Vendors and Platforms

Drop RWX pages
- Where necessary perform fine-grained page permission control

Implement efficient NX/userland page table implementation
- Checking page permissions for each instruction would incur significant overhead
- Instead, keep track of mappings and permissions in-process
- Perform checks if instruction is from different page than the previous instruction’s
  - e.g. jumps or serial instructions across a page boundary

Use virtualization
- And ensure that ASLR is implemented/used to protect sensitive structures
This could be done in a couple of ways:

1. Trust only ARM .so .text sections on load
2. Check /proc/self/maps on each “new” page that hasn’t been added to the data structure
3. Instrument memory mapping-related syscalls (e.g. mmap, mprotect) to track page permissions

An ideal solution combines 2 and 3, with the checks for 2 performed as a catch-all:
   - Supports dynamic .so loading via dlopen(3)
   - Supports legitimate JITing
     - And removes JIT pages when cleared/reset/freed to prevent page reuse attacks

This data structure acts as a page table and should be heavily protected (writeable only when being updated, surrounded by guard pages, not accessible to ARM, etc.)
Recommendations (cont’d)

For anyone doing analysis of Android applications

- Dynamic analysis should also run apps through libhoudini
- Static analysis should look for access to Houdini RWX pages and attempts to execute from non-executable pages
  - and anything scanning the JNIEnv function pointers
Conclusion

- Houdini introduces a number of security weaknesses into processes using it
- Some of these impact the security of the emulated ARM code, while some also impact the security of host x86 code
- These issues overall undermine core native code hardening
- Houdini not being well-documented publicly nor easily accessible may have prevented wider security analysis and research into it that could have caught these issues earlier
Disclosure — Timeline

[04/24/21] Findings (discussed in this talk) sent to Intel PSIRT via secure@intel.com
[05/05/21] Intel PSIRT confirms receipt of findings, and sends a few questions
[05/07/21] NCC Group sends a response answering Intel's questions
[05/07/21] Intel PSIRT confirms receipt of the additional information
[05/17/21] Intel PSIRT provides an update that the product team is looking into the findings
[06/25/21] Intel PSIRT provides an update that a fix release is planned for the end of July
[07/16/21] Additional findings (not discussed in this talk) sent to Intel PSIRT
[07/19/21] Intel PSIRT confirms receipt of the additional findings and that they will be sent to the Houdini team
[07/21/21] NCC Group previews this talk for Intel PSIRT
Big special thanks to...

- Jeff Dileo
- Jennifer Fernick
- Effi Kishko
Questions?

brian.hong@nccgroup.com
@im_eningeer