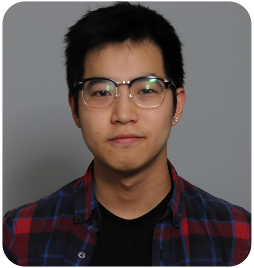
Sleight of ARM: Demystifying Intel Houdini

Brian Hong
@im_eningeer

whoami

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

Houdini — How's it work?

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

./houdini

Runs ARM executable binaries (static and dynamic)

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

```
:/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=441f7ee9bafadb1b141d27b82b28569e
 stripped
:/data/media/0/Download/arm-bin #
:/data/media/0/Download/arm-bin # ./hello_static
Hello world!
:/data/media/0/Download/arm-bin #
```

- Loaded in by the Linux kernel `binfmt_misc` feature

./houdini — binfmt_misc

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 7f454c46010101000000000000000000020028
```

```
localhost ~ # cat /proc/sys/fs/binfmt_misc/arm_dyn
enabled
interpreter /system/bin/houdini
flags: P
offset 0
magic 7f454c46010101000000000000000000030028
```

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

0x02	ET_EXEC
0x03	ET_DYN

[1] https://en.wikipedia.org/wiki/Binfmt_misc

libhoudini.so

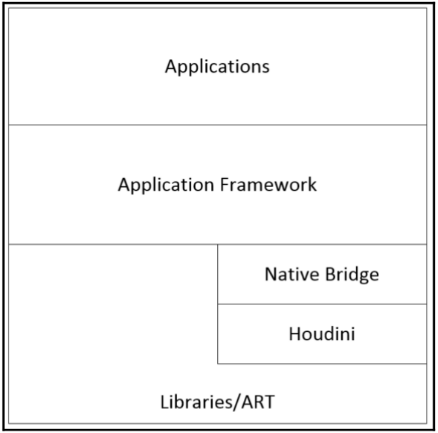
- Is a shared object (x86)

```
:/ # 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



Native Bridge in Android architecture [1]

[1] Ye, Roger. Android System Programming: Porting, Customizing, and Debugging Android HAL. Packt Publishing, 2017.

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`

Android NativeBridge — Java Native Interface (JNI)

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.

```
typedef uint8_t  jboolean; /* unsigned 8 bits */
typedef int8_t   jbyte;    /* signed 8 bits */
typedef uint16_t jchar;    /* unsigned 16 bits */
typedef int32_t  jint;     /* signed 32 bits */
typedef int64_t  jlong;    /* signed 64 bits */
```

```
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...
    ...
    jmethodID (*GetStaticMethodID)(JNIEnv*, jclass...
    jobject   (*CallObjectMethod)(JNIEnv*, jobject...
    jboolean    (*CallBooleanMethod)(JNIEnv*, jobject...
    ...
    jbyte     (*GetByteField)(JNIEnv*, jobject, jfieldID);
    jchar     (*GetCharField)(JNIEnv*, jobject, jfieldID);
    jint      (*GetIntField)(JNIEnv*, jobject, jfieldID);
    ...
```

source[1]

[1] https://android.googlesource.com/platform/libnativehelper/+/refs/heads/master/include_jni/jni.h

Android NativeBridge — Callbacks

`NativeBridgeRuntimeCallbacks` provide a way for native methods to call JNI native functions.

NativeBridge -> libhoudini

```
// 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[1]

[1] https://android.googlesource.com/platform/art/+/master/runtime/native_bridge_art_interface.cc

Android NativeBridge — Interface

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);
};
```

source[1]

[1] https://android.googlesource.com/platform/art/+/master/runtime/native_bridge_art_interface.cc

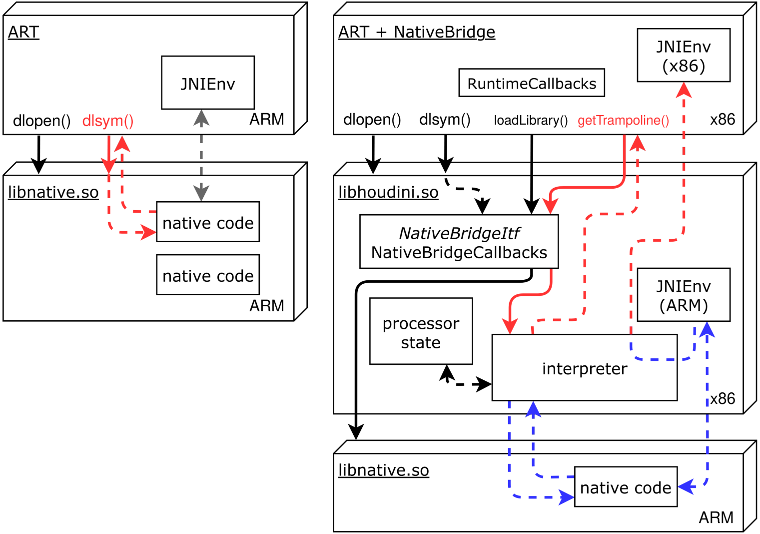
NativeBridge — Libhoudini

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

DYNAMIC SYMBOL TABLE:
...
004f8854 g    DO .data 0000003c Base     NativeBridgeItf
```

```
                    NativeBridgeItf                      XREF[1]…Entry Point(*)
00508854 03 00 00     NativeCallbacks
         00 60 03
         2a 00 10…
  ├─00508854 03 00 00 00          3h                  version
  ├─00508858 60 03 2a 00          initialize          initialize
  ├─0050885c 10 fe 29 00          loadLibrary         loadLibrary
  ├─00508860 50 fe 29 00          getTrampoline       getTrampoline
  ├─00508864 40 00 2a 00          isSupported         isSupported
  ├─00508868 80 00 2a 00          getAppEnv           getAppEnv
  ├─0050886c c0 00 2a 00          isCompatibleWith    isCompatibleWith
  ├─00508870 e0 00 2a 00          getSignalHandler    getSignalHandler
  ├─00508874 10 01 2a 00          unloadLibrary       unloadLibrary
  ├─00508878 20 03 2a 00          getError            getError
  ├─0050887c a0 01 2a 00          isPathSupported     isPathSupported
  ├─00508880 f0 02 2a 00          initAnonymousName…  initAnonymousNames…
  ├─00508884 40 02 2a 00          createNamespace     createNamespace
  ├─00508888 90 02 2a 00          linkNamespaces      linkNamespaces
  ├─0050888c c0 02 2a 00          loadLibraryExt      loadLibraryExt
  └─00508890 ff ff ff ff          ffffffff            getVendorNamespace
00508894 01           ??                 01h
```

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

```
00008000-0000a000 rw-p 00000000       [anon:Mem_0x10000002]
0c000000-0c001000 r--p 00000000       /vendor/lib/arm/nb/libdl.so
0c001000-0c002000 r--p 00000000       /vendor/lib/arm/nb/libdl.so
0c200000-0c203000 r--p 00000000       /data/app/com.nccgroup.research.../lib/arm/libnative-lib.so
0c203000-0c204000 r--p 00002000       /data/app/com.nccgroup.research.../lib/arm/libnative-lib.so
0c204000-0c205000 rw-p 00003000       /data/app/com.nccgroup.research.../lib/arm/libnative-lib.so
0c500000-0c5d6000 r--p 00000000       /vendor/lib/arm/nb/libc.so
0c5d6000-0c5da000 r--p 000d5000       /vendor/lib/arm/nb/libc.so
0c5da000-0c5dc000 rw-p 000d9000       /vendor/lib/arm/nb/libc.so
...
0e094000-10000000 rwxp 00000000       [anon:Mem_0x20000000]
12000000-12100000 rwxp 00000000       [anon:Mem_0x10001000]
12100000-12122000 rw-p 00000000       [anon:Mem_0x10001000]
12153000-1218c000 rw-p 00000000       [anon:Mem_0x10001000]
e5502000-e598d000 r-xp 00000000       /vendor/lib/libhoudini.so
e598d000-e59bf000 r--p 0048a000       /vendor/lib/libhoudini.so
e59bf000-e59ff000 rw-p 004bc000       /vendor/lib/libhoudini.so
ecdb0000-eceaa000 r-xp 00000000       /system/lib/libc.so
eceaa000-eceae000 r--p 000f9000       /system/lib/libc.so
eceae000-eceb0000 rw-p 000fd000       /system/lib/libc.so
ee0da000-ee0dc000 rwxp 00000000       [anon:Mem_0x10000000]
ee1b5000-ee303000 r-xp 00000000       /system/bin/linker
ee303000-ee309000 r--p 0014d000       /system/bin/linker
ee309000-ee30a000 rw-p 00153000       /system/bin/linker
ff26d000-ffa6c000 rw-p 00000000       [stack]
```

Houdini Emulator — Execution

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

```
                    ****************************************************
                    * EDI contains PC                                  *
                    * ESI points to processor_state struct             *
                    * EBP points to instruction handler table          *
                    ****************************************************
                    LAB_0030fe1b
0030fe1b 8b 1f               MOV      EBX,dword ptr [EDI]              ; read PC (fetch instruction)
0030fe1d 8d 47 08            LEA      EAX,[EDI + 0x8]                  ; read PC+8
0030fe20 89 86 9c 01 00 00   MOV      dword ptr [ESI + 0x19c],EAX      ; store it somewhere
0030fe26 8b c3               MOV      EAX,EBX                          ; instruction now in EAX & EBX
0030fe28 c1 e8 1c            SHR      EAX,0x1c                         ; get condition code (bits 28-31)
0030fe2b 89 be f0 08 00 00   MOV      dword ptr [ESI + 0x8f0],EDI
0030fe31 83 c7 04            ADD      EDI,0x4
0030fe34 89 7e 3c            MOV      dword ptr [ESI + 0x3c],EDI
0030fe37 83 f8 0e            CMP      EAX,0xe                          ; check condition code
0030fe3a 0f 85 6d 02 00 00   JNZ      LAB_003100ad                     ; jump if condition code != 0xE (always)
0030fe40 83 c4 08            ADD      ESP,0x8
0030fe43 8b c3               MOV      EAX,EBX                          ; instruction now in EAX & EBX
0030fe45 56                  PUSH     ESI                              ; push processor_state struct as argument
0030fe46 c1 e8 04            SHR      EAX,0x4                          ; bits 4-7
0030fe49 53                  PUSH     EBX                              ; push instruction as argument
0030fe4a c1 eb 10            SHR      EBX,0x10                         ; bits 16-27
0030fe4d 83 e0 0f            AND      EAX,0xf                          ; mask bit 4-7
0030fe50 81 e3 f0 0f 00 00   AND      EBX,0xff0                        ; mask bits 20-27
0030fe56 03 c3               ADD      EAX,EBX                          ; combine the above fields: instr[27:20]_instr[7:4]
0030fe58 8b 54 85 00         MOV      EDX,dword ptr [EBP + EAX*0x4]    ; get offset into instruction handler table
0030fe5c ff d2               CALL     EDX                              ; jump to instruction handler
```

Houdini Emulator — Instruction Table

Instruction bits 27-20 concatenated with bits 7-4 is used as the offset into the table

```
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);
```

```
                instr_table
004bb9c0 20 96 31 00        ddw     LAB_00319620
004bb9c4 a0 65 31 00        addr    LAB_003165a0
004bb9c8 d0 96 31 00        addr    LAB_003196d0
004bb9cc 10 60 31 00        addr    LAB_00316010
004bb9d0 b0 91 31 00        addr    LAB_003191b0
004bb9d4 c0 62 31 00        addr    LAB_003162c0
004bb9d8 e0 93 31 00        addr    LAB_003193e0
004bb9dc e0 60 31 00        addr    LAB_003160e0
004bb9e0 20 96 31 00        addr    LAB_00319620
004bb9e4 80 09 32 00        addr    LAB_00320980
004bb9e8 d0 96 31 00        addr    LAB_003196d0
004bb9ec 00 39 37 00        addr    LAB_00373900
004bb9f0 b0 91 31 00        addr    LAB_003191b0
004bb9f4 e0 3a 37 00        addr    LAB_00373ae0
004bb9f8 e0 93 31 00        addr    LAB_003193e0
004bb9fc 40 3e 37 00        addr    LAB_00373e40
004bba00 30 93 31 00        addr    LAB_00319330
004bba04 a0 63 31 00        addr    LAB_003163a0
004bba08 a0 94 31 00        addr    LAB_003194a0
004bba0c 70 66 31 00        addr    LAB_00316670
```

Listing: libhoudini.so

```
004bc034 80 45 37 00    addr    LAB_00374580
004bc038 50 89 31 00    addr    LAB_00318950
004bc03c 90 49 37 00    addr    LAB_00374990
004bc040 f0 1c 38 00    addr    instr_mov_0
004bc044 d0 b5 31 00    addr    instr_mov_1
004bc048 90 97 31 00    addr    instr_mov_2
004bc04c 70 b1 31 00    addr    instr_mov_3
004bc050 f0 a4 31 00    addr    instr_mov_4
004bc054 30 a4 31 00    addr    instr_mov_5
004bc058 80 3b 38 00    addr    LAB_00383b80
004bc05c 10 a9 31 00    addr    LAB_0031a910
004bc060 70 a6 31 00    addr    LAB_0031a670
004bc064 60 36 38 00    addr    LAB_00383660
004bc068 90 97 31 00    addr    instr_mov_2
004bc06c 90 3a 37 00    addr    FUN_00373a90
004bc070 f0 a4 31 00    addr    instr_mov_4
004bc074 d0 3d 37 00    addr    LAB_00373dd0
004bc078 50 b5 31 00    addr    LAB_0031b550
004bc07c d0 40 37 00    addr    LAB_003740d0
004bc080 80 1d 38 00    addr    DAT_00381d80
004bc084 d0 b6 31 00    addr    LAB_0031b6d0
004bc088 10 98 31 00    addr    LAB_00319810
004bc08c a0 b0 31 00    addr    LAB_0031b0a0
004bc090 70 a5 31 00    addr    LAB_0031a570
004bc094 a0 a3 31 00    addr    LAB_0031a3a0
004bc098 80 3b 38 00    addr    LAB_00383b80
004bc09c 60 a8 31 00    addr    LAB_0031a860
004bc0a0 10 a6 31 00    addr    LAB_0031a610
004bc0a4 90 28 38 00    addr    LAB_00382890
004bc0a8 10 98 31 00    addr    LAB_00319810
004bc0ac b0 43 37 00    addr    LAB_003743b0
004bc0b0 70 a5 31 00    addr    LAB_0031a570
004bc0b4 d0 45 37 00    addr    LAB_003745d0
004bc0b8 b0 b4 31 00    addr    LAB_0031b4b0
```

Decompile: instr_mov_1 - (libhoudini.so)

```c

int instr_mov_1(uint instr,proc_state *state)

{
  int iVar1;
  byte bVar2;
  uint uVar3;
  uint Rd;
  uint newPC;

  Rd = (instr & 0xffff) >> 0xc;
  if (Rd == 0xf) {
    s_000059(state);
  }
  uVar3 = (instr & 0xfff) >> 8;
  if (uVar3 == 0xf) {
    s_000059(state);
  }
  if ((instr & 0xf) == 0xf) {
    s_000059(state);
  }
  bVar2 = (byte)state->reg[uVar3];
  uVar3 = state->reg[instr & 0xf] << (bVar2 & 0x1f);
  if (0x1f < bVar2) {
    uVar3 = 0;
  }
  state->reg[Rd] = uVar3;
  if (Rd == 0xf) {
    newPC = state->reg[0xf];
    if (state->isThumb == 0) {
      branch_something();
    }
    else {
      state->ldrstr = 0x11;
      state->reg[0xf] = newPC & 0xfffffffe;
    }
    if ((newPC == 0xffff0fa0) || (newPC == 0xffff0fc0)) {
      iVar1 = (*DAT_006807c4)(DAT_006807c0);
      FUN_003c55f0(*(undefined4 *)(*(int *)(iVar1 + 8) + 0x730));
    }
  }
  return 0x86;
}

```

Houdini Emulator — Processor State

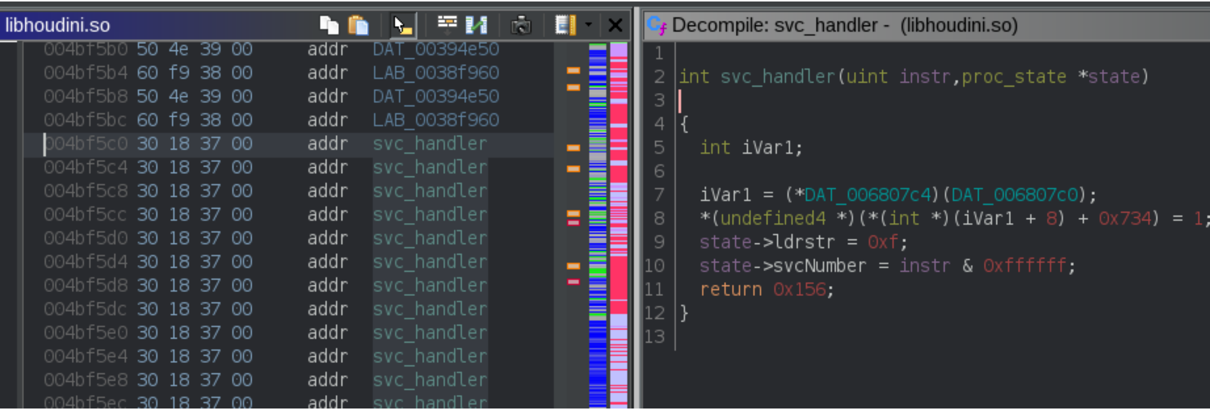
- Stores ARM registers, as well as other processor states

```
/* 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

Houdini Emulator — Syscall

- 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

Houdini Emulator — Detection

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

Security Concerns — RWX + Other Interesting Pages

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]
0e094000-10000000 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]
...
10128000-12000000 rw-p     [anon:Mem_0x10002000]
12000000-12100000 rwxp     [anon:Mem_0x10001000]
12100000-12122000 rw-p     [anon:Mem_0x10001000]
1215a000-12193000 rw-p     [anon:Mem_0x10001000]
ca6e8000-ca6e9000 ---p     [anon:Mem_0x10000004]
ca6e9000-caae8000 rw-p     [anon:Mem_0x10000004]
caae8000-caae9000 ---p     [anon:Mem_0x10000004]
caae9000-cabe8000 rw-p     [anon:Mem_0x10000004]
...
e4f99000-e4f9a000 ---p     [anon:Mem_0x10000004]
e4f9a000-e4f9f000 rw-p     [anon:Mem_0x10000004]
e8cb4000-e8cb6000 rwxp     [anon:Mem_0x10000000]
```

ARM JNIEnv → 10004000-10015000

ARM stack → caae9000-cabe8000

Security Concerns — NX Ignored

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

Page Permissions — A Matter of Interpretation

```
$ 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

Libhoudini-aware Malware (cont’d)

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)

Libhoudini-aware Malware (cont’d)

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

Recommendations (cont'd) — Custom NX Validation

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
[07/30/21] Intel provides NCC Group with the following statement:

> "Intel would like to thank the NCC Group for responsibly reporting their findings to our PSIRT. The weaknesses identified in the research have been addressed in the Intel® Bridge Technology (Intel® BT) hotfix release Houdini_9.0.6b.51725. Updates were released at the end of July 2021."

Big special thanks to...

- Jeff Dileo
- Jennifer Fernick
- Effi Kishko

Questions?

brian.hong@nccgroup.com
@im_eningeer