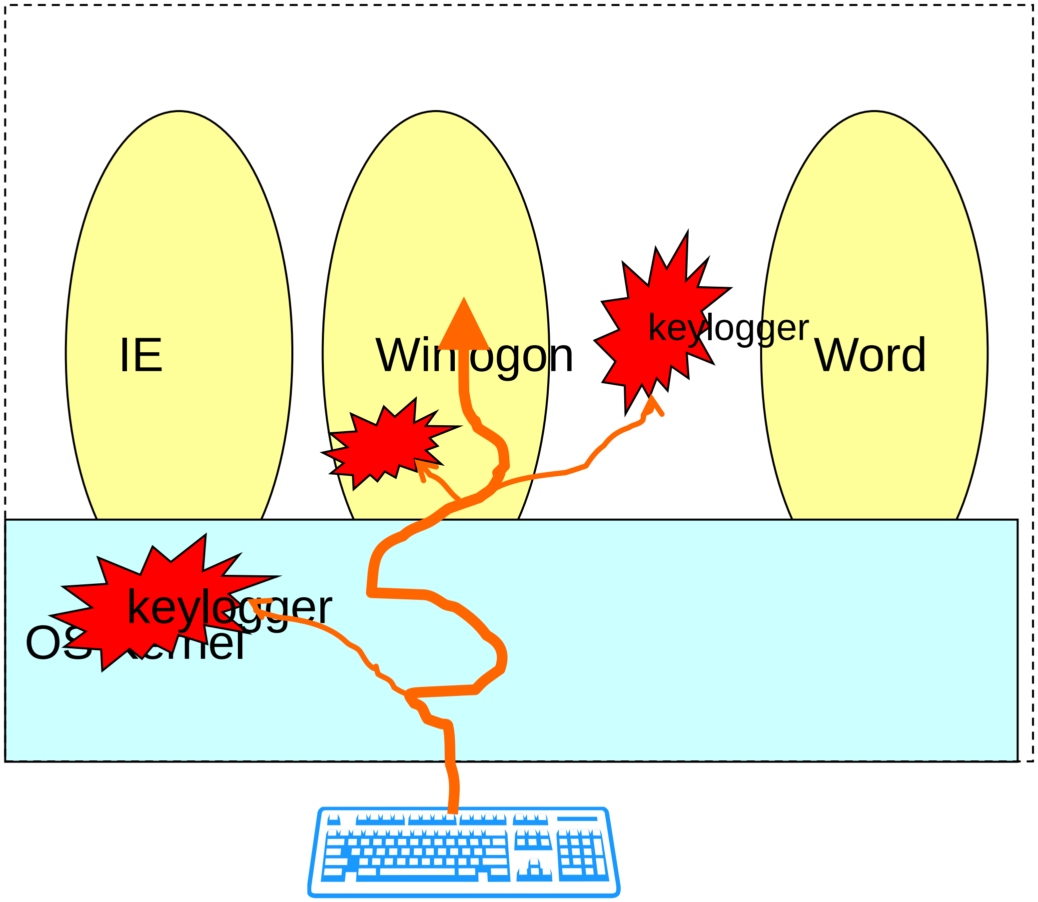
CS 250: Software Security

Full-System Dynamic Binary Analysis

Why whole-system?

- Malware analysis
  - Resides in the kernel space; Scatters in multiple processes
- Vulnerability analysis
  - For the OS kernel and device drivers
- Embedded systems
  - Contains an OS kernel and user-level programs

Full-System Tainting for Malware Analysis



[CCS'07] Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis

What is needed?

- Dynamic Taint Analysis
  - Tracking important information flows for entire system
  - Implement DTA in QEMU
- Hooking APIs/System Calls
  - Understand API-level behaviors
- Current Process & Modules
  - What processes/modules are currently executed

- Question:
  - How do I know this OS-level knowledge from hardware-level execution (QEMU)

The Answer: Virtual Machine Introspection

- Definition:
  - **Virtual Machine Introspection (VMI)** is a technique that observes and analyzes the state of a virtual machine (VM) from the **hypervisor,** without modifying the guest OS itself.

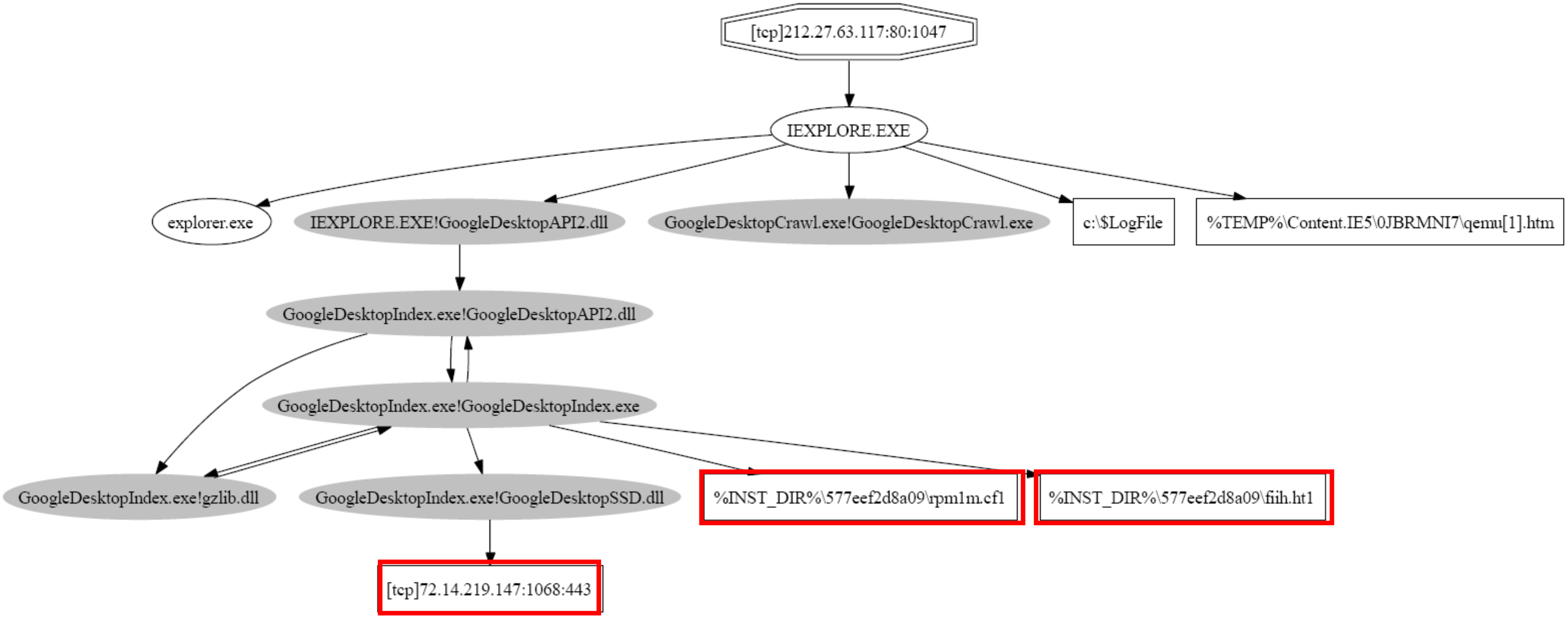
- How it works in general:
  - Intercept important events (e.g., syscall, context switch, page fault, breakpoint)
  - Parse important data structures in memory

Identifying the Current Process

- Each process has its own page directory base register
  - CR3 for x86; TTBR for ARM

- Parse kernel data structures
  - EPROCESS for Windows; task_struct for Linux
  - VMI tools have “profiles” describing where these structures are in memory
    - Current process pointer is at a known offset in kernel stack (Windows) or “gs” segment in Linux
  - Parse the structures to identify process name, PID, loaded modules, etc.

An Example: Google Desktop

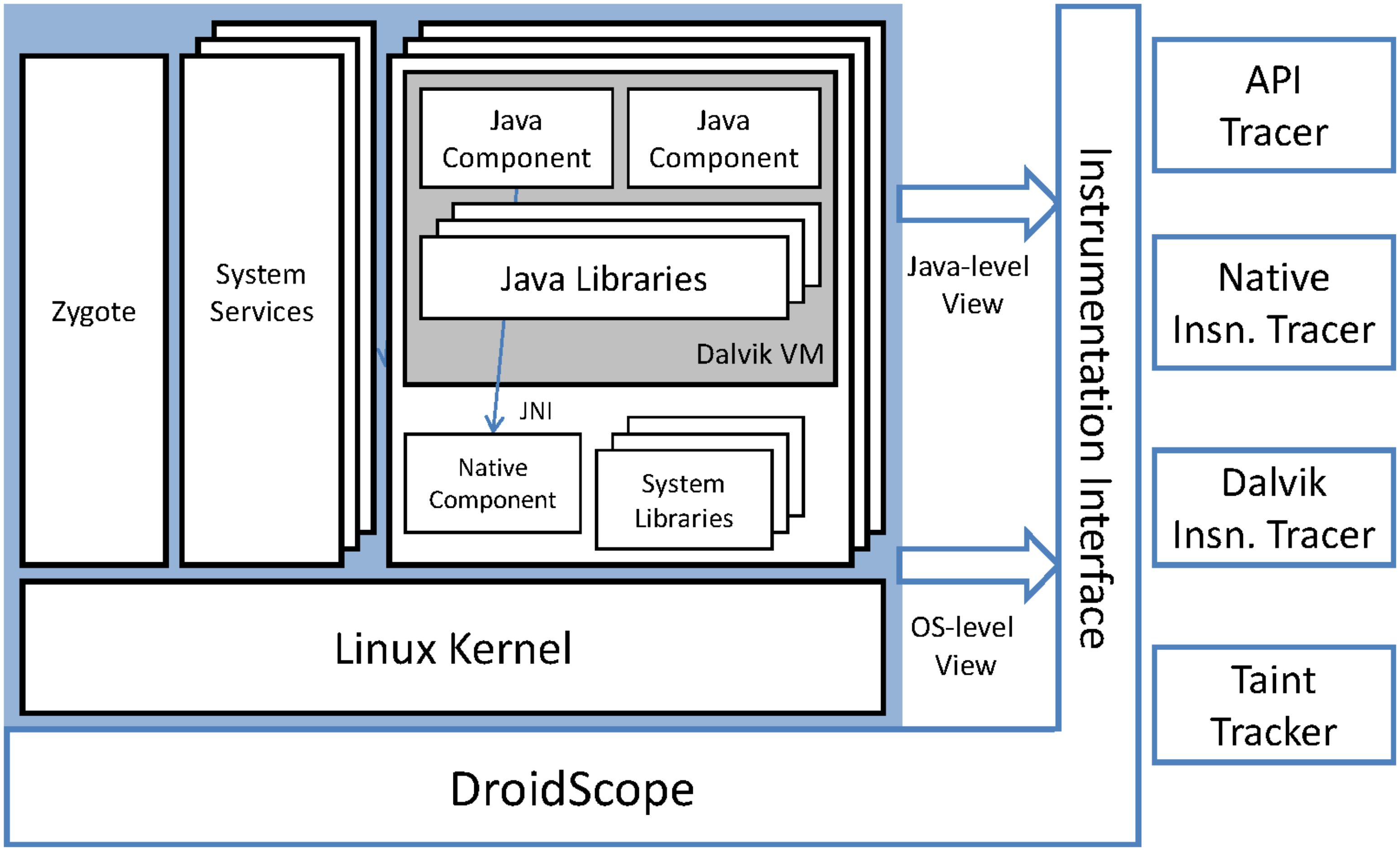


Google Desktop obtains the incoming HTTP traffic, saves it into two index files, and then sends it out though an HTTPS connection, to a remote Google Server

Dynamic Binary Analysis for Android System

[USENIX Security 2012] DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis

DroidScope Overview

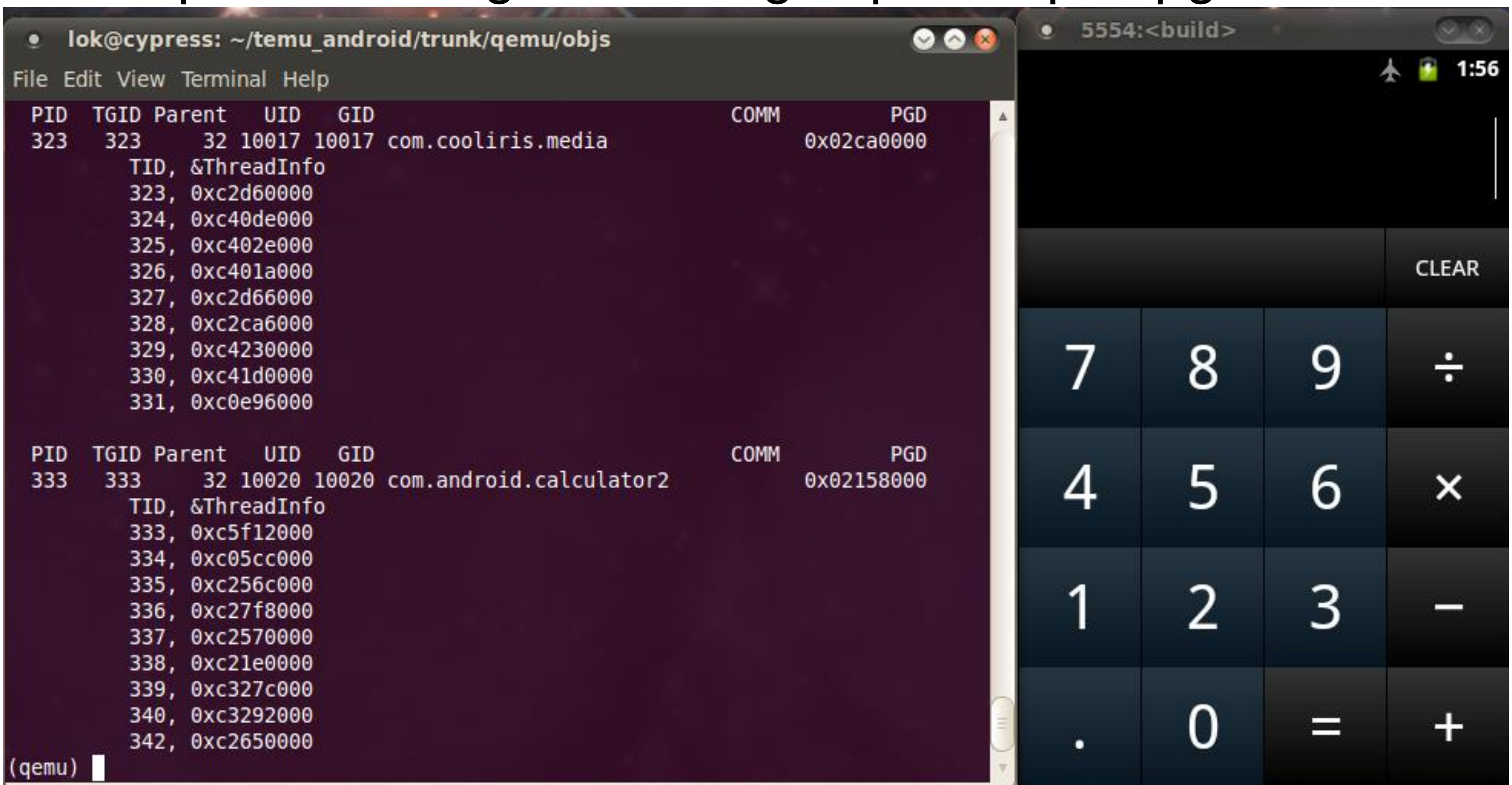


Goals

- Dynamic binary instrumentation for Android
  - Leverage Android Emulator in SDK
  - No changes to Android Virtual Devices
  - External instrumentation
    - Linux context
    - Dalvik context
  - Extensible: plugin-support / event-based interface
  - Performance
    - Partial JIT support
    - Instrumentation optimization

Linux Context: Identify App(s)

- Shadow task list
  - pid, tid, uid, gid, euid, egid, parent pid, pgd, comm



Java/Dalvik View

- Dalvik virtual machine
  - register machine (all on stack)
  - 256 opcodes
  - saved state, *glue*, pointed to by ARM R6, on stack in x86

- mterp
  - offset-addressing: *fetch opcode* then jump to *(dvmAsmInstructionStart* + *opcode * 64)*
  - *dvmAsmSisterStart* for emulation overflow

- Which Dalvik opcode?
  1. Locate dvmAsmInstructionStart in shadow memory map
  2. Calculate opcode = (R15 - dvmAsmInstructionStart) / 64.

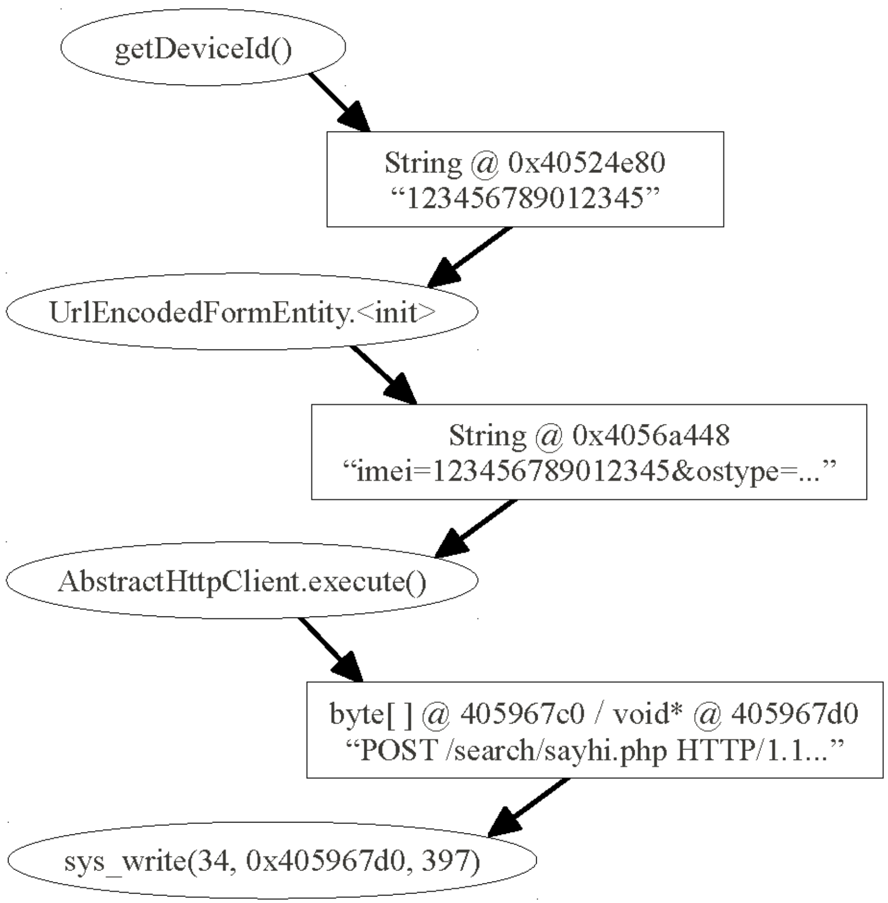
Just In Time (JIT) Compiler

- Designed to boost performance
- Triggered by counter - mterp is always the default
- Trace based
  - Multiple basic blocks
  - Multiple exits or *chaining cells*
  - Complicates external introspection
  - Complicates instrumentation

Droid Kung Fu

- Three encrypted payloads
  - ratc (Rage Against The Cage)
  - killall (ratc wrapper)
  - gjsvro (udev exploit)
- Three execution methods
  - piped commands to a shell (default execution path)
  - Runtime.exec() Java API (instrumented path)
  - JNI to native library terminal emulator (instrumented path)
  - Instrumented return values for *isVersion221* and *getPermission* methods

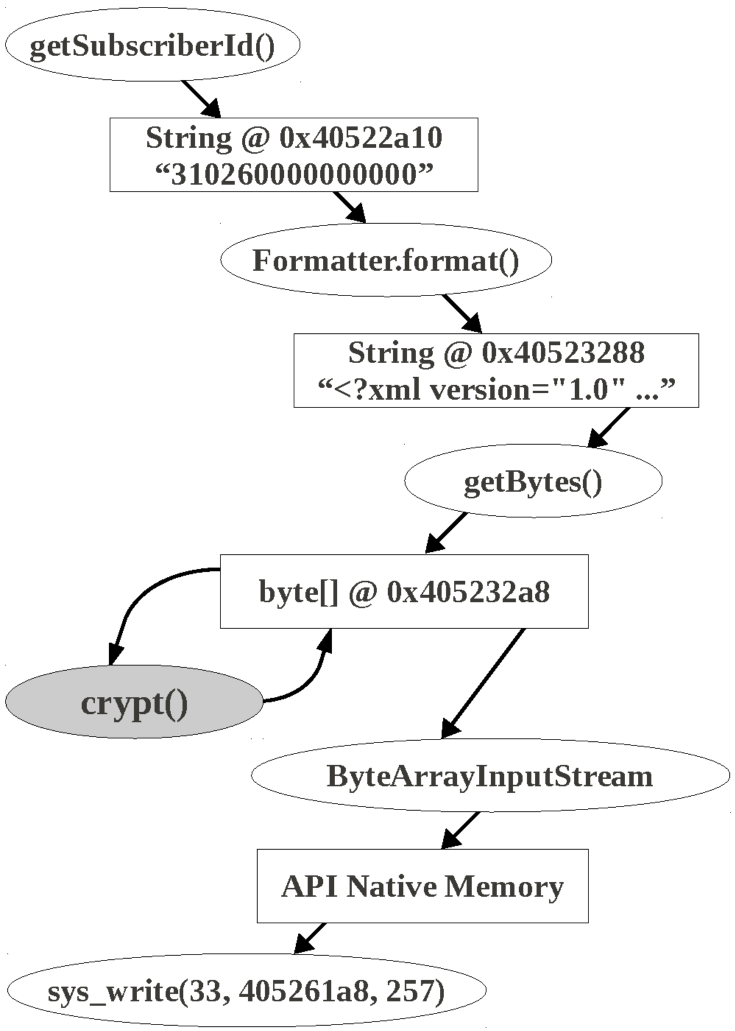
Droid Kung Fu: TaintTracker



DroidDream

- Same payloads as DroidKungFu
- Two processes
  - Normal *droiddream* process clears logcat
  - *droiddream:remote* is malicious
- xor-encrypts private information before leaking
- Instrumented *sys_connect* and *sys_write*

Droid Dream: TaintTracker



DroidDream: crypt trace

```
[43328f40] aget-byte v2(0x01), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 40523372(2401372)
  Adding M@410accec(42c5cec) len = 4
[43328f44] sget-object v3(0x0000005e), KEYVALUE// field@0003
[43328f48] aget-byte v3(0x88), v3(0x4051e288), v1(58)
[43328f4c] xor-int/2addr v2(62), v3(41)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@410accec(42c5cec) len = 4
[43328f4e] int-to-byte v2(0x17), v2(23)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@410accec(42c5cec) len = 4
[43328f50] aput-byte v2(0x17), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 410accec(42c5cec)
  Adding M@40523372(2401372) len = 1
```

ratc

- Vulnerability
  - *setuid()* fails when RLIMIT_NPROC reached
  - *adbd* fails to verify *setuid()* success
- Three generation (stage) exploit
  - Locate *adbd* in */proc* and spawns child
  - Child *fork()* processes until *-11 (-EAGAIN)* is returned then spawns child – continues *fork()*
  - Grandchild *kill() adbd* and waits for process to re-spawn

ratc: exploit diagnosis

```
;;;setgid returns from kernel back to adbd
00008l3c: pop {r4, r7}
00008140: movs r0, r0
00008144: bxpl lr : Read Oper[0]. R14, Val = 0xc3a5
;; Return back to 0xc3a4 (caller) in Thumb mode

;;;adbd_main sets up for setuid
0000c3a4: movs r0, #250
0000c3a6: lsls r0, r0, #3 : Write Oper[0]. R0, Val = 0x7d0
  ;; 250 * 8 = 0x7d0 = 2000 = AID_SHELL

  ...

  ;;;Start of setuid section
  ;;;  213 is syscall number for sys_setuid
  00008be0: push {r4, r7} : Write Oper[0]. M@be910bb8, Val = 0x7d0
    ;; push AID_SHELL onto the stack
  00008be4: mov r7, #213
  00008be8: svc 0x00000000
    ;; Make sys call

    ;;; === TRANSITION TO KERNEL SPACE ===

    ;;;sys_setuid then calls set_user in kernel mode

      ;;;inside sys_setuid
      ;; Has rlimit been reached?
      c0048944: cmp r2, r3  : Read Oper[0]. R3, Val = 300 Read Oper[1]. R2, Val = 300

      ;;; RLIMIT(300) is reached and !init_user so return -11
      c0048960: mvn r0, #10 : Write Oper[0]. R0, Val = 0xfffffff5
          ;; the return value is now -11 or -EAGAIN
      c0048964: ldmib sp, {r4, r5, r6, fp, sp, pc}

      ;;;Return back to sys_setuid which returns back to userspace

    ;;; === RETURN TO USERSPACE ===

  ;;;setuid continues
  00008bec: pop {r4, r7}
  00008bf0: movs r0, r0 : Read Oper[0]. R0, Val = 0xfffffff5
      ;; -11 is still here

;;;Return back to adb_main at 0xc3ac (the return address) above
;;; Immediately starts other work, does not check return code
0000c3ac: ldr r7, [pc, #356] : Read Oper[0]. M@0000c514, Val = 0x19980330
  Write Oper[0]. R7, Val = 0x19980330
    ;; 0x19980330 is _LINUX_CAPABILITY_VERSION
```

Symbol Information

- Native library symbols - Static
  - From *objdump* of libraries
- Java symbols - Dynamic
  - Dalvik data structures -> address of string
  - Given address, load from
    - Memory
    - File mapped into memory
  - *dexdump* as backup

Discussion

- Emulation Fidelity and Transparency
- Relevance to Memory Forensics
- Full-system or Kernel Fuzzing