# Put a Sock(et) in it!

*Understanding and Attacking Sockets on Android*


Jake Valletta

April 16, 2016

# Who Am I

- Senior Consultant at Mandiant
- Mobile security researcher
- Beer drinker
- @jake_valletta

# Agenda

- Introduction to Sockets
- IPC and Sockets on Android
- Attacking Android Sockets
- Questions

# What's a Socket?

# Sockets are…

- A mechanism for Inter Process Communication (IPC)
  - Across a network
  - On a local system
- Typically bi-directional
- Foundational to the Internet and modern operating systems

# Why do we Care?

- Sockets are used by critical components of the Android operating system
  - OEMs don't always do things correctly
- Sockets routinely lead to security vulnerabilities on Android Devices
  - CVE-2011-1823: *vold* accepted AF_NETLINK connections from arbitrary processes
  - "weaksauce": Exposed /dev/socket/dmagent AF_UNIX socket can root HTC phones
  - /dev/socket/fotabinder (LG), /dev/socket/init_runit, …
  - …

# Using Sockets

- APIs exist for most programming languages
  - Python (“socket” module)
  - Java (“java.net.*” classes)
  - Go (“net” package)
  - Perl (“IO::Socket::*” module)
- All roads lead to ~~Rome~~ system calls
  - <sys/socket.h>

# Socket System Calls (Server)

- socket(…)
  - return file descriptor
  - domain, type, and protocol arguments
- bind(…)
  - Associate file descriptor with PID/IP/port/file (depends!)
- listen(…)
  - Indicate listening for connection on socket file descriptor
- accept(…) **(Blocking!)**
  - Accept connections from socket file descriptor
  - Returns new client connection file descriptor

# Socket System Calls (Client)

- socket(…)
  - return file descriptor
  - domain, type, and protocol arguments
- bind(…)
  - Associate file descriptor with PID/IP/port/file (depends!)
- connect(…)
  - Connect socket file descriptor to supplied socket IP/port/file
  - Not always required

# Socket System Calls (Data Transfer)

- Sending data:
  - send(…) / sendto(…) / sendmsg(…) / write(…)
- Receiving data:
  - recv(…) / recvfrom(…) / recvmsg(…) / read(…)

# Socket Domains

- Maintained by kernel
  - include/linux/socket.h
- AF_INET + AF_INET6
- AF_UNIX (also called AF_LOCAL)
- AF_NETLINK
- …

# Socket Types

- Maintained by kernel
  - include/linux/net.h
- SOCK_STREAM (e.g. “TCP”)
  - Sequential, reliable, two-way, connection-based
- SOCK_DGRAM (e.g. “UDP”)
  - connectionless, unreliable
- SOCK_SEQPACKET
  - Sequential, reliable, two-way, connection-based, fixed-max length
- SOCK_RAW
  - “you’re on your own”
- …

# Socket Domain: AF_INET + AF_INET6

- Internet Protocol (IP) sockets
  - Designed to traverse networks
- What most people simply call "sockets"
- "bind()" call expects IP address and port
- Types: SOCK_STREAM for TCP, SOCK_DGRAM for UDP

```
import socket

sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)

sock.connect(("1.2.3.4", 80))

try:
   message = 'Hi. I am a silly TCP socket.'
   sock.send(message)
finally:
   sock.close()
```

# Socket Domain: AF_UNIX

- UNIX domain sockets (UDS)
  - Local only
  - More lightweight than AF_INET
  - process <-> process
- "bind()" call expects socket-type filename **or** abstract name
  - Abstract always starts with null byte ('\0')
- Types: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET

# Socket Domain: AF_UNIX

## *File UNIX Socket*

```
import socket

sock = socket.socket(socket.AF_UNIX,
                     socket.SOCK_STREAM)

sock.connect("/dev/socket/silly")

try:
    message = 'Hi. I am a silly UNIX socket.'
    sock.send(message)
finally:
    sock.close()
```

Note the \0

## *Abstract UNIX Socket*

```
import socket

sock = socket.socket(socket.AF_UNIX,
                     socket.SOCK_STREAM)

sock.connect("\0very_silly")

try:
    message = 'Hi. I am a silly abstract-UNIX socket.'
    sock.send(message)
finally:
    sock.close()
```

# Socket Domain: AF_NETLINK

- Communication between kernel and user space
- "bind()" call expects PID and netlink group
  - Groups are maintained by kernel
    - include/uapi/linux/netlink.h
- Types: SOCK_DGRAM, SOCK_RAW (doesn't actually matter)

```
import socket

PID_KERNEL = 0              # PID of the kernel
NETLINK_SILLY = 31          # New Group

sock = socket.socket(socket.AF_NETLINK,
                     socket.SOCK_RAW)

sock.connect((PID_KERNEL, NETLINK_SILLY))

try:
    message = 'VERY silly netlink socket .'
    sock.send(message)
finally:
    sock.close()
```

# Sockets on Android

# IPC on Android

- Android is built on Linux kernel so…
  - Sockets, FIFO, pipes, ioctl, system calls, shared memory
- Android also provides Binder/Intents for doing IPC between applications
  - Primary app <-> app IPC mechanism
  - Java and Java Native Interface (JNI)

# Non-Application Socket IPC

- AF_UNIX sockets
  - App <-> native daemons
    - /dev/socket/*
- AF_NETLINK sockets
  - Native daemon <-> kernel
    - /system/bin/vold
- AF_INET sockets
  - Others…?
    - …but why?

# Finding Sockets

- Kernel maintains lists of sockets in /proc/net/ directory
- Formats are not easy to parse
- '/system/bin/netstat' utility parses these files but…
  - Android uses a stripped down version of netstat

```
06:21:38 ~$ adb shell cat /proc/net/tcp
  sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
   0: 0100007F:13AD 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 1837 1 00000000 100 0 0 10 -1
   1: 00000000:15B3 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0        0 1841 1 00000000 100 0 0 10 -1
   2: 0F02000A:15B3 0202000A:D48E 01 00000000:00000000 00:00000000 00000000     0        0 1842 3 00000000 21 4 29 10 -1
06:21:40 ~$
```

?

# Finding Sockets (cont.)

- Solution #1: Compile your own `netstat`
  - Meh.
- Solution #2: Upload `busybox` binary
  - https://busybox.net/downloads/binaries/latest/

```
06:32:32 ~$ wget https://busybox.net/downloads/binaries/latest/busybox-armv7l -q
06:32:43 ~$ adb push busybox-armv7l /data/local/busybox
1207 KB/s (1109128 bytes in 0.896s)
06:32:56 ~$ adb shell chmod 775 /data/local/busybox
06:33:06 ~$ adb shell /data/local/busybox netstat -h
netstat: invalid option -- h
BusyBox v1.21.1 (2013-07-08 10:26:30 CDT) multi-call binary.

Usage: netstat [-ral] [-tuwx] [-enWp]
```

```
06:34:47 ~$ adb shell /data/local/busybox netstat -lt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:5037          0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:5555            0.0.0.0:*               LISTEN
```

# Finding Sockets (cont.)

- Flags, flags, flags
  - -l – Listening sockets
  - -t – TCP sockets
  - -u – UDP sockets
  - -x – UNIX sockets
  - -e – Extra information
  - -p – Print process ID and program name (only if you have permission!)

# Finding Sockets: AF_INET

- *busybox netstat -pletu*

```
07:17:45 ~$ adb shell /data/local/busybox netstat -pletu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 localhost:5037          0.0.0.0:*               LISTEN      66/adbd
tcp        0      0 0.0.0.0:5555            0.0.0.0:*               LISTEN      66/adbd
tcp        0      0 :::9999                 :::*                    LISTEN      11471/busybox
```

# Finding Sockets: AF_UNIX

- *busybox netstat -plex*

```
07:08:17 ~$ adb shell /data/local/busybox netstat -plex
Active UNIX domain sockets (only servers)
Proto RefCnt Flags       Type       State         I-Node PID/Program name    Path
unix  2      [ ACC ]     STREAM     LISTENING       1838 66/adbd              @jdwp-control
unix  2      [ ACC ]     STREAM     LISTENING       4085 1053/com.jakev.loca @silly_unix
unix  2      [ ACC ]     STREAM     LISTENING       4833 1615/com.android.br @webview_devtools_remote_1615
unix  2      [ ACC ]     STREAM     LISTENING       1655 1/init               /dev/socket/property_service
unix  2      [ ACC ]     STREAM     LISTENING       1683 51/vold              /dev/socket/vold
unix  2      [ ACC ]     STREAM     LISTENING       1866 54/debuggerd         @android:debuggerd
unix  2      [ ACC ]     STREAM     LISTENING       1699 57/zygote            /dev/socket/zygote
unix  2      [ ACC ]     STREAM     LISTENING       1721 55/rild              /dev/socket/rild-debug
unix  2      [ ACC ]     STREAM     LISTENING       2517 355/system_server    /data/system/ndebugsocket
unix  2      [ ACC ]     STREAM     LISTENING       1724 55/rild              /dev/socket/rild
unix  2      [ ACC ]     STREAM     LISTENING       1736 53/netd              /dev/socket/mdns
unix  2      [ ACC ]     STREAM     LISTENING       1739 53/netd              /dev/socket/dnsproxyd
unix  2      [ ACC ]     STREAM     LISTENING       1741 53/netd              /dev/socket/netd
unix  2      [ ACC ]     STREAM     LISTENING       1748 60/installd          /dev/socket/installd
unix  2      [ ACC ]     STREAM     LISTENING       1784 62/qemud             /dev/socket/qemud
```

# Finding Sockets: AF_UNIX

- *busybox netstat -plex*

```
07:08:17 ~$ adb shell /data/local/busybox netstat -plex
Active UNIX domain sockets (only servers)
Proto RefCnt Flags       Type       State         I-Node PID/Program name    Path
unix  2      [ ACC ]     STREAM     LISTENING       1838 66/adbd              @jdwp-control
unix  2      [ ACC ]     STREAM     LISTENING       4085 1053/com.jakev.loca  @silly_unix
unix  2      [ ACC ]     STREAM     LISTENING       4833 1615/com.android.b   @webview_devtools_remote_1615
unix  2      [ ACC ]     STREAM     LISTENING       1655 1/init               /dev/socket/property_service
unix  2      [ ACC ]     STREAM     LISTENING       1683 51/vold              /dev/socket/vold
unix  2      [ ACC ]     STREAM     LISTENING       1866 54/debuggerd         @android:debuggerd
unix  2      [ ACC ]     STREAM     LISTENING       1699 57/zygote            /dev/socket/zygote
unix  2      [ ACC ]     STREAM     LISTENING       1721 55/rild              /dev/socket/rild-debug
unix  2      [ ACC ]     STREAM     LISTENING       2517 355/system_server    /data/system/ndebugsocket
unix  2      [ ACC ]     STREAM     LISTENING       1724 55/rild              /dev/socket/rild
unix  2      [ ACC ]     STREAM     LISTENING       1736 53/netd              /dev/socket/mdns
unix  2      [ ACC ]     STREAM     LISTENING       1739 53/netd              /dev/socket/dnsproxyd
unix  2      [ ACC ]     STREAM     LISTENING       1741 53/netd              /dev/socket/netd
unix  2      [ ACC ]     STREAM     LISTENING       1748 60/installd          /dev/socket/installd
unix  2      [ ACC ]     STREAM     LISTENING       1754 62/qemud             /dev/socket/qemud
```

Abstract contains '@'

File shows path

# Finding Sockets: AF_NETLINK

- busybox's netstat doesn't parse AF_NETLINK
- You'll need to parse /proc/net/netlink file
  - (more on this later)

# Connecting to Sockets

- Method #1: `adb` forwarding
  - Allows tcp, localabstract, localfilesystem, and more!
  - Write a Python client that talks to forwarded or use `nc`

```
07:58:21 ~$ adb forward tcp:9999 tcp:9999
07:58:42 ~$ telnet 127.0.0.1 9999
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

root@generic:/ #
```

# Connecting to Sockets (cont.)

- Method #2: Compile and upload `socat`
  - Super flexible command-line utility
  - TCP, TCPv6, UDP, UDPv6, UNIX (file + abstract)
  - http://core.nctritech.com/do/socat-arm-static **(WARNING – Not mine!)**

# Connecting to Sockets (cont.)

- Method #3: Create a Java application
  - Public "LocalSocket" and "LocalSocketAddress" classes (UNIX)
  - Non-Public "NativeDaemonConnector" class (UNIX)
  - Public "Socket" class (INET)

```
LocalSocket sender = new LocalSocket();
sender.connect(new LocalSocketAddress("silly_unix"));
sender.getOutputStream().write("Let's send a silly message".getBytes());
sender.getOutputStream().close();
```

```
mConnector = new NativeDaemonConnector(
        new NetdCallbackReceiver(), "netd", 10, NETD_TAG, 160);
```

# Connecting to Sockets (cont.)

- Method #3: Create a Java application
  - Public "LocalSocket" and "LocalSocketAddress" classes (UNIX)
  - Non-Public "NativeDaemonConnector" class (UNIX)
  - Public "Socket" class (INET)

Abstract UNIX

```
LocalSocket sender = new LocalSocket();
sender.connect(new LocalSocketAddress("silly_unix"));
sender.getOutputStream().write("Let's send a silly message".getBytes());
sender.getOutputStream().close();
```

'/dev/socket/netd' UNIX

```
mConnector = new NativeDaemonConnector(
        new NetdCallbackReceiver(), "netd", 10, NETD_TAG, 160);
```

# Connecting to Sockets (cont.)

- Method #4: Create a C/C++ program
  - Slowest method, but also most powerful
  - Works with all types of sockets, including AF_NETLINK

# Attacking Android Sockets

# Device Testing Framework

- Modular framework for testing Android devices
  - https://github.com/jakev/dtf
  - https://github.com/jakev/dtfmods-core
- Modules for data collection and processing
- Modules for determining potential security flaws

```
07:49:05 ~$ dtf -h
Android Device Testing Framework (dtf) v1.3.1-dev
Usage: dtf [module|command] <arguments>
   Built-in Commands:
    archive     Archive your dtf project files.
    client      Install/remove the dtf client.
    help        Prints this help screen.
    init        Initializes a project.
    local       Display all local modules.
    pm          The dtf package manager.
    prop        The dtf property manager.
    reset       Removes the dtf config from current directory.
    source      Used for sourcing additional commands.
    status      Determine if project device is attached.
```

# Who is Listening?

- Java applications
  - AF_INET
  - AF_UNIX
- Native binaries (“daemons”)
  - AF_INET
  - AF_UNIX
  - AF_NETLINK
- The kernel
  - AF_NETLINK

# AF_INET Example

- Uncommon..
  - Why would you want an application to listen on a socket locally anyways?

```
08:48:56 /x$ dtf_busybox netstat -pletu
netstat: showing only processes with your user ID
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:1668            0.0.0.0:*               LISTEN      -
tcp        0      0 localhost:5037          0.0.0.0:*               LISTEN      -
```

- …but still happens!

# AF_INET Example

- Determining *who* is listening can be a challenge*
- Searching in Java applications:
  - grep application DEX for ServerSocket, or use dtf 'dexsearch'

```
09:41:22 /DevTesting/[redacted]$ dtf dexsearch s "java/net/ServerSocket" -f apps
Matches in applications:
Match(es) in com.v[redacted].db
  Ljava/net/ServerSocket;
Match(es) in com.[redacted].db
  Ljava/net/ServerSocket;
09:41:27 /DevTesting/[redacted]$
```

  - Reverse application and determine protocol

**** Without root privileges***

# AF_INET Example

- Searching in native binaries:
  - Determine running processes
    - *adb shell ps*
  - Pull binaries from "/(vendor|system)/(s|x)?bin/.*"
    - *adb pull …*
  - Look for "socket", "bind", "listen", "accept"  dynamic symbols
    - *nm –D …*
  - Reverse binary to determine domain/type and bind parameters
    - Radare2, IDA Pro

# AF_INET Example

- turbohacker.sh

```
10:27:31 /DevTesting/[redacted]$ for bin in `ls system-bins` \
> do if [ "$(nm -D system-bins/$bin 2>/dev/null \
> |grep -E "(socket|bind|listen|accept)"|wc -l)" = "4" ]; then \
> echo "HIT: $bin"; fi; done
HIT: dhcpcd
HIT: dnsmasq
HIT: gsensorCalibrate
HIT: hcidump
HIT: ip
HIT: mdnsd
HIT: nc
HIT: netperf
HIT: netserver
HIT: racoon
HIT: storeintenttranslate.x
```

?

# AF_INET Example

- Use 'adb forward' with `nc` to interact with TCP socket
- Use `socat` on device to interact with UDP socket
- **Pro-tip:** Always check the logs with `adb logcat`

```
10:49:28 /DevTesting/[redacted]$ dtf_busybox netstat -pletu
netstat: showing only processes with your user ID
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
tcp        0      0 0.0.0.0:1668            0.0.0.0:*
tcp        0      0 localhost:5037          0.0.0.0:*
tcp        0      0 :::9999                 :::*
10:49:34 /DevTesting/[redacted]$ adb forward tcp:1668 tcp:1668
10:49:37 /DevTesting/[redacted]$ nc localhost 1668 -v
Connection to localhost 1668 port [tcp/*] succeeded!
asdfasdf
10:49:52 /DevTesting/[redacted]$
```

```
10:50:23 /$ adb logcat|grep 3046
E/storeintenttranslate( 3046): Error: Only support GET! "asdfasdf"
```

# AF_UNIX Example

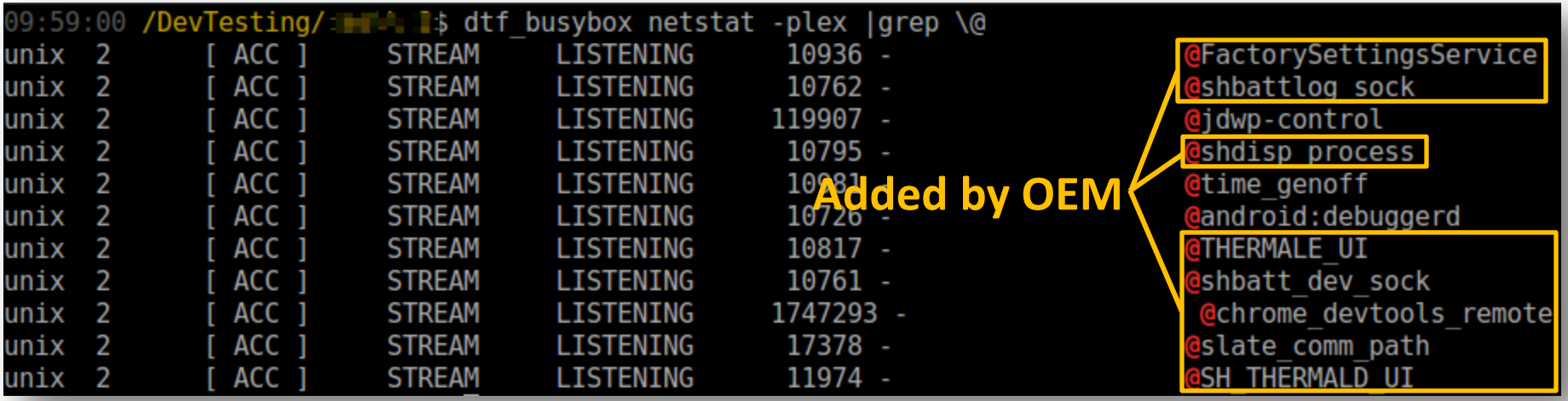
- UNIX sockets are very common and easy to search
  - Especially abstract!

```
09:59:00 /DevTesting/[redacted]$ dtf_busybox netstat -plex |grep \@
unix  2      [ ACC ]     STREAM     LISTENING      10936 -                    @FactorySettingsService
unix  2      [ ACC ]     STREAM     LISTENING      10762 -                    @shbattlog_sock
unix  2      [ ACC ]     STREAM     LISTENING     119907 -                    @jdwp-control
unix  2      [ ACC ]     STREAM     LISTENING      10795 -                    @shdisp_process
unix  2      [ ACC ]     STREAM     LISTENING      10981 -                    @time_genoff
unix  2      [ ACC ]     STREAM     LISTENING      10726 -                    @android:debuggerd
unix  2      [ ACC ]     STREAM     LISTENING      10817 -                    @THERMALE_UI
unix  2      [ ACC ]     STREAM     LISTENING      10761 -                    @shbatt_dev_sock
unix  2      [ ACC ]     STREAM     LISTENING    1747293 -                     @chrome_devtools_remote
unix  2      [ ACC ]     STREAM     LISTENING      17378 -                    @slate_comm_path
unix  2      [ ACC ]     STREAM     LISTENING      11974 -                    @SH_THERMALD_UI
```

- Ask yourself – *Is this socket part of the AOSP, or OEM added?*

# AF_UNIX Example

- UNIX sockets are very common and easy to search
  - Especially abstract!




- Ask yourself – *Is this socket part of the AOSP, or OEM added?*

# AF_UNIX Example

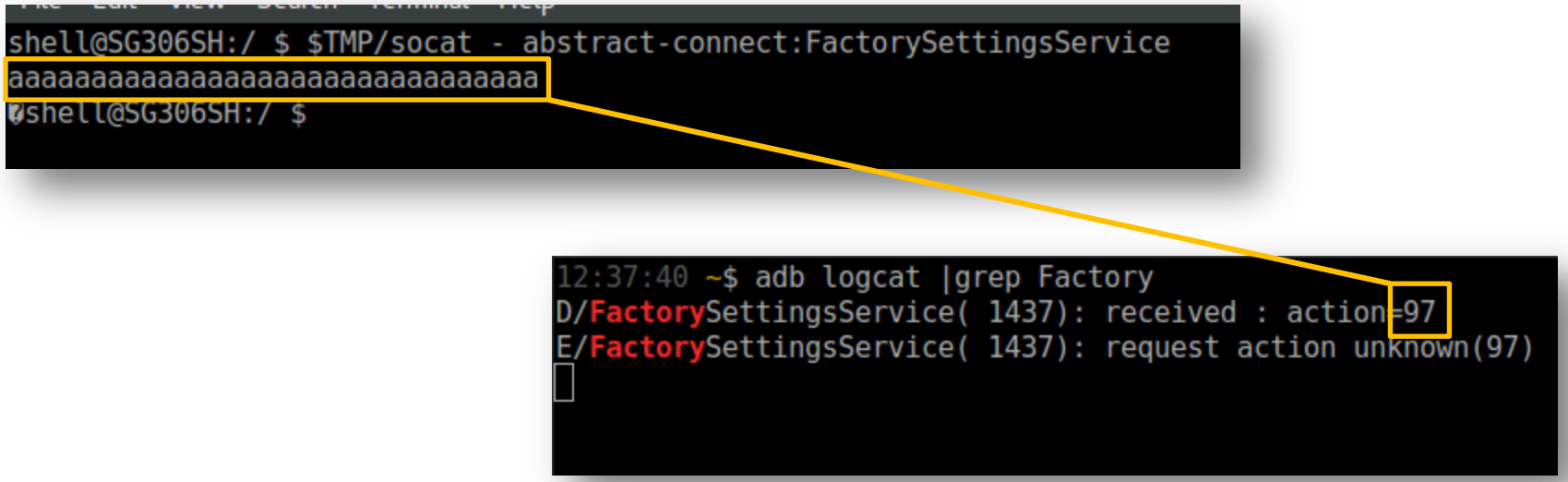
- Searching in Java applications:
  - Use grep or "dexsearch" module

```
10:39:21 /DevTesting/[redacted]$ dtf dexsearch s FactorySettingsService
Matches in applications:
Matches in frameworks:
Match(es) in jp.co.sharp.android.factorysettings.db
  FactorySettingsService
10:39:39 /DevTesting/[redacted]$ grep -rl FactorySettingsService \
> unframework/jp.co.sharp.android.factorysettings/
unframework/jp.co.sharp.android.factorysettings/jp/co/sharp/android/factorysettings/Ipc.smali
10:39:56 /DevTesting/[redacted]$
```

- Searching in native binaries:
  - Use one-liner from previous slide

# AF_UNIX Example

- Use ‘adb forward’ or `socat`
- **Pro-tip:** Always check the logs with `adb logcat`



# AF_NETLINK Example

- Most difficult to trace and replicate
  - Could have a whole presentation on netlink sockets alone ☺
- Parsing "/proc/net/netlink" contains netlink group and PID

```
11:33:11 /DevTesting/Note$ dtf socklist --filter netlink|grep -vE "(None|kernel)"
[Thu Apr 14 23:33:13 EDT 2016] socklist/I - Waiting for device...
AF_NETLINK Sockets:
   /system/bin/wpa_supplicant group=0 PID=20213
   /system/bin/rild group=0 PID=294
   flipboard.app group=0 PID=30221
   com.google.android.apps.magazines group=0 PID=30732
   /system/bin/dhcpcd group=0 PID=21888
   com.google.android.googlequicksearchbox:search group=0 PID=9593
   /system/bin/netd group=0 PID=290
   /system/bin/cnd group=4 PID=341
   /system/bin/netd group=5 PID=290
   /system/bin/keystore group=7 PID=312
   /system/bin/servicemanager group=7 PID=264
   /init group=7 PID=1
   zygote group=7 PID=336
   /system/bin/auditd group=9 PID=1358
   /sbin/healthd group=15 PID=261
   /system/bin/vold group=15 PID=265
   /system/bin/netd group=15 PID=290
   /sbin/ueventd group=15 PID=223
   system_server group=15 PID=928
   com.samsung.android.MtpApplication group=15 PID=29027
   /system/bin/mpdecision group=15 PID=3250
```

Added by OEM

# AF_NETLINK Example

- Determine group mappings from kernel source
  - *Usually* in "include/uapi/linux/netlink.h", but doesn't have to be
- Review netlink documentation on group protocol format

```
11:46:56 /DevTesting/Notes$ dtf socklist --filter netlink|grep auditd
   /system/bin/auditd group=9 PID=1358
```

```
#define NETLINK_XFRM        6   /* ipsec */
#define NETLINK_SELINUX     7   /* SELinux event notifications */
#define NETLINK_ISCSI       8   /* Open-iSCSI */
#define NETLINK_AUDIT       9   /* auditing */
#define NETLINK_FIB_LOOKUP  10
#define NETLINK_CONNECTOR   11
#define NETLINK_NETFILTER   12  /* netfilter subsystem */
#define NETLINK_IP6_FW      13
#define NETLINK_DNRTMSG     14  /* DECnet routing messages */
#define NETLINK_KOBJECT_UEVENT  15  /* Kernel messages to userspace */
```

# AF_NETLINK Example

- Determine format of netlink messages and permission checks from kernel source
  - "netlink_kernel_create(…)" kernel function to create netlink sockets in kernel

*drivers/misc/qseecom.c*

```
#ifdef CONFIG_BBRY
static int __init qseecom_nl_init(void)
{
#if LINUX_VERSION_CODE >= KERNEL_VERSION(3, 10, 0)
    struct netlink_kernel_cfg cfg = {
            .groups = QSCNLGRP_MAX,
            .flags  = 0,
        };
    qscnl = netlink_kernel_create(&init_net, NETLINK_QSEECOM, &cfg);
#else
    qscnl = netlink_kernel_create(&init_net, NETLINK_QSEECOM,
            QSCNLGRP_MAX, NULL, NULL, THIS_MODULE);
#endif
    if (qscnl == NULL)
        pr_err("cannot create netlink socket\n");

    pr_info("netlink socket created\n");
    return 0;
}
```

**Qualcomm-added netlink protocol**

# AF_NETLINK Example

- User space netlink sockets will use “socket()/bind()” but ***not*** “listen()/accept()”
  - “bind()” might not be used, depending on group/protocol

```
86      // Socket to listen on for uevent changes
87      memset(&nladdr, 0, sizeof(nladdr));
88      nladdr.nl_family = AF_NETLINK;
89      nladdr.nl_pid = getpid();
90      nladdr.nl_groups = 0xffffffff;
91
92      if ((uevent_sock = socket(PF_NETLINK,
93                              SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
94          LOGE("Unable to create uevent socket: %s", strerror(errno));
95          exit(1);
96      }
97
98      if (setsockopt(uevent_sock, SOL_SOCKET, SO_RCVBUFFORCE, &uevent_sz,
99                     sizeof(uevent_sz)) < 0) {
100         LOGE("Unable to set uevent socket options: %s", strerror(errno));
101         exit(1);
102     }
103
104     if (bind(uevent_sock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
105         LOGE("Unable to bind uevent socket: %s", strerror(errno));
106         exit(1);
107     }
```

# Conclusions

# In conclusion…

- Sockets are an important component of Android platform security
  - AF_INET, AF_UNIX, and AF_NETLINK are the most common forms of sockets used on Android
- Sockets can be enumerated, fuzzed, and researched mostly without a rooted device
- Unprotected sockets can be used to interact with the kernel, privileged applications, or native binaries

# Questions? Comments?

# Contact Me!

- GitHub: https://github.com/jakev/
- Blog: http://blog.thecobraden.com
- Website: https://www.thecobraden.com/
- Twitter: @jake_valletta
- Email: javallet@gmail.com

# The End

*Thanks!*