Setting Up Android Debug Bridge for Analysis

The Goal

To supplement our packet traces captured through tcpdump, we wish to get timestamps of system calls related to network scanning.

Setting Up Android Debug Bridge

Follow the instructions here to set wireless debugging on your device. It doesn't matter if you use wireless debugging or wired debugging. I chose to stick to wired debugging, to minimize interference with anything network related.

Getting to a Shell

Once you have adb set up, get your serial number by running `adb devices -l` which should print out something like:

```
List of devices attached
R58M32SC7WK            device usb:1-1 product:beyond2qltecs  ...
```

In my case, R58M32SC7WK is my serial number. All we need to get a shell is run `adb -s YOUR_SERIAL_NUMBER shell`.

At this point, we should be inside a restricted shell.

Finding Useful Things Within the Shell

Now comes the tricky part, using this limited set of tools, we must find either the information are looking for or find the means to install a tool such as iw on the device.

Let's compare the output from `netstat --help` on both my android shell and the manpage for netstat found here

Running `adb -s R58M32SC7WK shell "netstat --help"`, we get the following output from running netstat on the android device:

```
Toybox 0.8.4-android multicall binary ...

usage: netstat [-pWrxwutneal]

Display networking information. Default is netstat -tuwx

-r      Routing table
-a      All sockets (not just connected)
```

```
-l      Listening server sockets
-t      TCP sockets
-u      UDP sockets
-w      Raw sockets
-x      Unix sockets
-e      Extended info
-n      Don't resolve names
-W      Wide display
-p      Show PID/program name of sockets
```

Compare this to some of the manpage description:

```
netstat [address_family_options] [--tcp|-t] [--udp|-u] [--raw|-w] ...

netstat {--route|-r} [address_family_options] [--extend|-e[--extend|-e]] ...

netstat {--interfaces|-I|-i} [iface] [--all|-a] [--extend|-e] ...

netstat {--groups|-g} [--numeric|-n] [--numeric-hosts][--numeric-ports] ...

netstat {--masquerade|-M} [--extend|-e] [--numeric|-n] [--numeric-hosts]

netstat {--statistics|-s} [--tcp|-t] [--udp|-u] [--raw|-w] [delay]

netstat {--version|-V}

netstat {--help|-h}
```

We can see that many of the options available on the traditional tool have been removed. Therefore, not only do we have a limited set of binaries to work with, many of these binaries do not have their full feature sets.

One binary of interest is perfetto, which is a system profiling binary. One source of events is system calls.

Using Perfetto to Monitor System Calls

```
adb shell perfetto \
  -c - --txt \
  -o /data/misc/perfetto-traces/trace \
<<EOF

buffers: {
    size_kb: 63488
    fill_policy: DISCARD
}
buffers: {
    size_kb: 2048
    fill_policy: DISCARD
}
data_sources: {
    config {
        name: "linux.ftrace"
        ftrace_config {
            ftrace_events: "raw_syscalls/sys_recvmsg"
        }
    }
}
duration_ms: 100000

EOF
```

Using the above command, we scan for 100 seconds collecting the "recvmsg" system call. If we take a look at some strace output from my laptop scanning for available networks, we see the following:

```
11:46:07 (+     0.000197) sendmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000791) recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000231) recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000254) recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000224) recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000138) sendmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
11:46:07 (+     0.000215) recvmsg(3, {msg_name={sa_family=AF_NETLINK, nl_pid=0 ...
```

Note that each of these messages are related to netlink, which are the types of calls we are interested in. We won't be able to guarantee that each of the recvmsg are messages of interest to us, but we'll be able to filter them once we have the trace.

Now that the trace has been written to `/data/misc/perfetto-traces/trace` we need to extract it to our local machine in order to do the analysis. To do so, we run:

```
adb pull /data/misc/perfetto-traces/trace path_to_desired_location
```

Once we have the file, can either choose to analyze it here or choose to convert it to something that is easy to analyze on the command line such as json. To convert it to json, run:

```
curl -LO https://get.perfetto.dev/traceconv;
chmod +x traceconv;
./traceconv json [input file] [output file]
```

At this point, you should now have a file that you can analyze. As a side note, I completed the last bit of this report with an emulated device, so I did not get an actual file to check, but I did get this:

```
{"traceEvents":[
{"args":{"name":"swapper"},"cat":"__metadata","name":"thread_name","ph":"M", ...
{"trace_processor_stats":{"android_log_num_failed":0,"android_log_num_skippe ...
"androidProcessDump": "PROCESS DUMP\nUSER           PID  PPID     VSZ    RSS ...
  "controllerTraceDataKey": "systraceController"
}
```

Once I test this on a physical device, I will make any required changes to this document.