DroidScope- Getting Started Guide

Lok Yan

24 Mar 2012
Revised: May 21, 2012

Contents

1 Setting up the working environment

The easiest way to setup up the working environment is not to. That is to go and download the virtual machine image and run it.

1.1 DroidScope VM

The current version of the DroidScope VM is a xubuntu 11.10 image for qemu-kvm. The following command is used to load the image with 1gb of memory and kvm enabled.

```
qemu DroidScope.qcow2 -m 1g -enable-kvm
```

The VM is setup with a 20gb virtual disk and network enabled. It contains the base installation of xubuntu 11.10, so it is probably a good idea to do an update.Although updates sometimes breaks things, so I would suggest saving a copy first.

1.1.1 VM Organization

The username and password for the VM is "user/user."

All of the files you need to get started are located on the Desktop. `android-kernel` is a binary distribution of the 2.6.29 kernel for Android that we used for testing. `android-image` is a binary distribution of the Gingerbread image that we have used for testing. A few benchmarking apps have already been installed. `qemu` contains the DroidScope source.

The Android (Gingerbread) source and sdk are contained within the `Android` directory. `android-gingerbread` contains both the Android source as well as the kernel source (`android-gingerbread/kernel/qemu`. The binary distributions mentioned before are based on this specific source tree. Also, I have already applied the two patches mentioned in Section 1.3 and it has already been lunched for generic-eng. Thus, a simple make will start the building process.

There are also two icons of interest on the top panel. The first is for Eclipse, which is self explanatory. The second is a terminal shortcut which sets the working directory to `/home/user/Desktop/qemu/objs/` and sets the path to `/home/user/Desktop/Android/android-sdk-linux_x86/platform-tools` (where *adb* is). This is done by using the *.androidrc* file in the home directory.

1.1.2 qemu-nbd

While the virtual machine has network enabled, it might still be a good idea to know how to use qemu-nbd to mount the disk directly. Here are the list of commands.

```
1. sudo modprobe nbd
2. qemu-nbd -c /dev/nbd0 DroidScope.qcow2
3. sudo mount /dev/nbd0p1 mnt
4. sudo umount mnt
5. qemu-nbd -d /dev/nbd0 #this is important
6. sudo modprobe -r nbd
```

First, the *nbd* kernel needs to be mounted. Once it is mounted, you can create a nbd device using command 2. It essentially turns the qcow2 file into a new block device called nbd0. Once the new virtual block device is created, it can be mounted. *nbd0p1* just means the first partition of the new block device. Keep in mind that you might have to change the mount point though.

Once the image is mounted, you can do whatever you want with it, like read and write to it. Once you are done you can just umount it (line 4). This might take a while because of synching. Once it is unmounted, you need to disconnect the device **This is important**. As a double check, I also like to unload the nbd driver (line 6) since unload will not work unless the device is disconnected.

1.1.3 VirtualBox

If KVM is too slow, then it might be a good idea to convert it into a VirtualBox disk (vdi). Use the following commands from `http://en.wikibooks.org/wiki/QEMU/Images#Exchanging_images_with_VirtualBox` to do this:

```
qemu-nbd -c /dev/nbd0 DroidScope.qcow2
VBoxManage convertfromraw -format VID /dev/nbd0 DroidScope.vdi
```

If you don't want to use nbd then just convert it to raw first.

```
qemu-img convert -O raw DroidScope.qcow2 DroidScope.raw
VBoxManage convertfromraw -format VDI DroidScope.raw DroidScope.vdi
```

Once this is done, the virtual disk can then be used in a new VirtualBox virtual machine.

1.2 New Setup

If you really want to set up a new clean environment then please follow these instructions. They are all based off of xubuntu 11.10 32bit.

Open up a terminal and install the *g++, libsdl-dev, binutils-multiarch* packages. The first two packages are needed for building *qemu* and the binutils is good for interfacing with ARM binaries and etc. on the host.

```
sudo apt-get install g++ libsdl-dev binutils-multiarch
```

If you are going to be building the Android source as well then also install:

```
sudo apt-get install curl bison flex gperf git
```

You need Java so install it. Gingerbread (at least the version we have tested on) requires Java 1.6 and specifically Sun Java so follow these steps to get it installed.

I took this from `www.gaggl.com/2011/10/installing-java6-jdk-on-ubuntu-11-10/`. I did ignore the *sun-java6-plugin* though since the java plugin has a lot of security updates.

```
sudo add-apt-repository ppa:ferramroberto/java
sudo apt-get update
sudo apt-get install sun-java6-jdk
```

You might also want to run `update-java-alternatives -s java-6-sun` just to make sure you are using the right version of java.

Now you are ready to build the DroidScope qemu package. If you don't have the package, you can always just grab it from the virtual machine using *nbd*.

In the off chance that you don't have the Android sdk already installed you can grab it from `http://developer.android.com/sdk/index.html`. Don't forget to run `android` inside the *tools* subdirectory to install the *platform-tools* package. We will use *adb* later.

It is probably a good idea to install Eclipse as well so you can have an IDE for editing DroidScope.

1.3 Patches

During compilation of the Gingerbread source, I came about two errors and the fixes I found online are below.

1.3.1 FORTIFY_SOURCE

There are two build errors related to *_FORTIFY_SOURCE*. The first is *RefBase.cpp:507:67 error: passing 'const android::RefBase::'....*

edit line 60 of *frameworks/base/libs/utils/Android.mk* and add `-fpermissive` at the end so it changes from
`LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1 $(TOOL_CFLAGS)` to
`LOCAL_CFLAGS += -DLIBUTILS_NATIVE=1 $(TOOL_CFLAGS) -fpermissive`

The second is *_FORTIFY_SOURCE redefined [-werror]*. The following fix is grabbed from `code.google.com/p/android/issues/detail?id=20795`.

Edit line 61 of *build/core/combo/HOST_linux-x86.mk* and change it from
`HOST_GLOBAL_CFLAGS += -D_FORTIFY_SOURCE=0` to
`HOST_GLOBAL_CFLAGS += -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0`.

2 First Run

In this section we will show you how to run DroidScope and the `ps` command to list the running processes. This guide is based on the VM, so please adjust the commands accordingly if you are not using the VM.

First, open up a terminal and go to the *qemu* directory where the DroidScope source is. Then simply run `make` to build it.

Once built, go into the *objs* subdirectory. There, you should see the *emulator* executable that was just built. We will now use it to load the Android virtual machine.

To load the virtual machine we run:

```
./emulator -sysdir ~/Desktop/android-image \
  -kernel ~/Desktop/android-kernel/zImage \
  -qemu -monitor stdio
```

In the previous command, *-sysdir* is used to specify the directory that contains the Android virtual machine images. *-kernel* is used to specify the kernel to use. *-qemu* is used to specify that the rest of the parameters are actually directed towards the *qemu* emulator. This means that *-monitor stdio* is passed into qemu using the stdio to interface with the monitor.

If successful, you should see the Android device loaded and a qemu prompt `(qemu)`. You can use the *help* command to get a list of commands. From the long list of commands you should be able to find *linux_ps—ps* entry that "Prints out the process list." In particular this entry specifies that you can use either *linux_ps* or *ps* to print the process list. Use the *ps* command and you should see the process list. Here is a snippet of the one I saw.

```
 PID  TGID  Parent   UID   GID                          COMM         PGD
   0     0       0     0     0  swapper                        0x00000000
   1     1       0     0     0  init                           0x05964000
...
  27    27       1  1000     0  /system/bin/servicemanager     0x059f8000
  28    28       1     0     0  /system/bin/vold               0x05b50000
  29    29       1     0     0  /system/bin/netd               0x059ec000
  30    30       1     0     0  /system/bin/debuggerd          0x05b58000
  31    31       1  1001  1001  /system/bin/rild               0x05b48000
```

```
  32    32       1     0     0  zygote                        0x05b4c000
...
 152   152      32  1000  1000  com.android.systemui          0x053cc000
```

From the output we can see that *servicemanager* is has PID and TGID as 27 (e.g. only one thread) and was spawned by *init* since its parent has pid 1. Also its uid 1000 and its PGD is 0x059f8000.

If you don't see any meaningful output AND you used a different kernel then please see Section 5.1.

2.1 Hints

You can always change the virtual machine images to use as well as the kernel. For example, to use the images built in the included Android Gingerbread source just change the *-sysdir* parameter to

```
-sysdir ~/Desktop/Android/android-gingerbread/out/target/product/generic/
```

There is also a case where running the *emulator* command will give you the following error:

```
emulator: ERROR: You did not provide the name of an Android Virtual Device
with the '-avd <name>' option. Read -help-avd for more information.

If you *really* want to *NOT* run an AVD, consider using '-data <file>'
to specify a data partition image file (I hope you know what you're doing).
```

This particular error is most likely due to the fact that the *userdata-qemu.img* file is not present in the sysdir directory. To fix this simply create the file using `touch` or use the *-data* option to point it to the right image file to use for user data as the error suggests.

Important Keep in mind that the *userdata-qemu.img* file is used to keep state information such as installed Apps. This means that truncating the file or creating a new one will automatically revert the system back to the initial state.

3 DroidScope Code Layout

The qemu codebase that DroidScope uses comes directly from the gingerbread source in the *external/qemu* subdirectory. It might be a good idea to do a diff to see the native qemu files that were altered. The list includes

```
target-arm/translate.c - A lot of changes here
target-arm/helpers.h
target-arm/helper.c
qemu-monitor.hx
monitor.c
```

Most of the code for DroidScope is actually located inside the *TEMU* subdirectory. There are five subdirectories. *nativeAPI, linuxAPI* and *dalvikAPI* contains the code for implementing the three APIs. The *plugins* directory contains all of the different plugins - the *trace_custom* needs to be fixed. *utils* contains code for a bunch of useful utilities. We will visit these files in a little bit.

Perhaps the most important file in the *TEMU* directory is *TEMU_Init.c* since this is where you will need to register new initialization functions, e.g. for plugins. Once again we will get to this shortly.

Aside from these files, the other important file is the Makefile called *Makefile.android* in the parent *qemu* directory. The DroidScope declarations begin at around line 316 and it is here that any the source files for any new plugins or utils or whatnot will need to be included.

translate.c translate.c is a very important file located inside the *qemu/target-arm* directory. This is where all of the basic instrumentation code is embedded and it also contains some configuration options. A quick search for `CUSTOM_TRACE` provides a quick overview of all the code that was insertted into the file.

The most important configuration option in this file is `CHECK_IS_USER` which is used as a condition for inserting the instrumentation callbacks. Set the macro to 1 to include kernel code as well as user code and leave it alone for instrumenting userland code only.

4 First Plugin

In this section, we will go through the ropes of building the Hello World plugin that just prints out the "Hello World" message. Then we will immediately follow it with a simple Dalvik Instruction tracer.

4.1 Hello World

The Hello World plugin will familiarize us with the basics of building a plugin and outputting messages to the screen so let's get started.

In this plugin, we will use the *Output* utility. It is simply a wrapper for outputting messages. The basic idea is that the user uses a single function like `TEMU_printf` to do outputs. Depending on whether an output log file is defined or whether the monitor mode is enabled, the output will go to the appropriate location. The order is logfile, monitor and then stdout.

We first create a new file in the plugins directory, lets call it `simple.c`. The file is shown below:

```
#include "TEMU/utils/Output.h"

void simple_init()
{
  TEMU_printf("Hello World\n");
}
```

Next, we need to decide when this code is going to be called, DroidScope is event based. At this point lets simply call this function when the qemu emulator starts. To do this, we will need to edit the `TEMU/TEMU_init.c` file to include a call to *simple_init*. A snippet of the resulting file is shown below.

```
extern void simple_init();
void temu_qemu_init()
{
...
  simple_init(); //this should be at the end of the function
}
```

As can be seen in the code above, we use extern to reference *simple_init* and then call the function inside *temu_qemu_init()*. This function is called when qemu starts, so any calls made by this function will also be called when qemu starts.

The next step is to edit Makefile.androind in the qemu directory to include this new file as part of the build. Then just build the emulator and execute it to see what happens. You should see the "Hello World" message outputed to the monitor on the screen. Furthermore if you remove the *-qemu -monitor stdio* then the message should show up on standard out.

4.1.1 Process List

Now that we have the basics, lets try something a little bit more complicated. We will implement a simple process log - that is a log of the processes that were started.

The first step is to identify the appropriate event. In this case since we want to know when a process begins we will use the *process_begin* event. As this is a Linux event, it is included inside the *LinuxAPI* so we must include *LinuxAPI.h* in our source file. The resulting code is shown below:

```
#include "TEMU/LinuxAPI.h"
#include "TEMU/utils/Output.h"

void simple_process_begin_callback(CPUState* env, int pid)
{
  TEMU_printf("Hello World - Process [%d] started\n", pid);
}

void simple_init ()
{
  registerProcessBeginCallback(&simple_process_begin_callback);
}
```

Making and then running this new file will result in a long list of Hello World messages with the increasing process IDs. If you don't see output then see Section 4.1.2.

Now if you take a look at the documentation for LinuxAPI, you should see a list of access functions such as *printProcessList*. One of them is *findProcessByPID* which returns a pointer to a `ProcessInfo` structure. From the structure definition, we see two strings `strName` and `strComm`. Lets now change the code to print out the `strName` and the `strComm` name as well as the PID. The resulting code is below:

```
#include "TEMU/utils/Output.h"
#include "TEMU/LinuxAPI.h"

void simple_process_begin_callback(CPUState* env, int pid)
{
  ProcessInfo* pInfo = NULL;
  pInfo = findProcessByPID(pid);
  //double check the return value
  if (pInfo != NULL)
  {
    TEMU_printf("[%d] %s (%s)\n", pid, pInfo->strName, pInfo->strComm);
  }
}

void simple_init ()
{
  registerProcessBeginCallback(&simple_process_begin_callback);
}
```

Notice that the `strName` is empty in the print out. This is because the processBegin callback is called when the process is initially added to the shadow process list. At this point in time, the arg name has not been populated yet. In the interest of learning more about the code, lets try to fix this in two ways. The first way doesn't work, but its still a good exercise.

First is to change the API logic so that the argument name is populated before the processBegin callbacks are called. To do this, we will edit the `TEMU/linuxAPI/Context.c` file. As can be seen, the process name is originally populated around line 266. Notice that there is a call to *get_name* which is then used to call *addProcess* followed by a call to *callProcessStartedCallbacks*. Afterwhich, there is a call to *get_arg_name*. This is the function used to get the argument name and is also what we will use.

The idea is to make a call to *get_arg_name* before the call to *addProcess*. The prototype for *addProcess* shows that the second to last parameter (currently `NULL` is a pointer to the strName) is what we need to pass the arg name by. Given this information, we have the following code:

```
264    if (processMark(pid) == 1) // i.e. it doesn't exist
265    {
266      get_name(i, name, MAX_PROCESS_INFO_NAME_LEN);
267      char name2[MAX_PROCESS_INFO_NAME_LEN];
268      get_arg_name(i, name2, MAX_PROCESS_INFO_NAME_LEN);
269      //addProcess(pid, parentPid, tgid, glpid, uid, gid, euid, egid, pgd, NULL, name);
270      addProcess(pid, parentPid, tgid, glpid, uid, gid, euid, egid, pgd, name2, name);
271      processMark(pid);
272      callProcessStartedCallbacks(env, pid);
273    }
```

After making the change, you can make and then run the emulator again. This time you should see some non-sensical names as the arg name. In some ways this is expected since *updateProcessList* function is called when some interesting syscalls are made where it is possible that the whole process structure is not fully setup yet. Now lets look at the second way of doing this.

The second way to fix the argument name problem is to register for `processUpdated` events. If you look at the original code for *updateProcessList* in line 284 you will notice that when the argument name is updated, the processUpdatedCallbacks will be called. So this is the perfect opportunity for us. To do this, we will simply register for this new event. The resulting `simple.c` file is:

```
void simple_init ()
{
  registerProcessBeginCallback(&simple_process_begin_callback);
  registerProcessUpdatedCallback(&simple_process_begin_callback, INV_ADDR);
}
```

Notice that we used `INV_ADDR` as the second parameter to register for the processUpdated event. `INV_ADDR` (-1) is just a catch all pid value meaning call this callback for all processUpdated events and not only when a particular process has been updated.

After making and running this new code, you will notice that the arg names will appear correctly. The only problem is that there are a lot of lines of output (its very repetitive). This can be suppressed by comparing the previous argname with the new argname inside the *updateProcessList* function (around line 283) and then making the processUpdated events only when they are different. Now that you have looked at our failed first attempt, this other change should be easy.

4.1.2 Hint

If you don't see output, then double check that the *simple_init* call is after the *context_init* call inside *TEMU_Init.c*. This is important because you need to initialize the APIs (context is part of the LinuxAPI) before you can actually register for events.

Also if you want to try you can start qemu in monitor mode and then use the `set_out_file testout.txt`. You should notice that the messages that used to get printed out to the monitor (or stdout) are now being redirected to `testout.txt`.

5 VMI And Symbol Information

Before we continue to the next and final example, we will see how Virtual Machine Introspection and Symbol information are gathered.

5.1 Kernel Offsets

While we can follow the kernel stack to get the `threadinfo` structure which then leads to the `task_struct` list, we still need to know the data structure layouts in order to correctly interpret the data structures in the guest memory. All of the offset information are contained and defined inside a file called *kernelinfo.conf* inside the *objs* directory of qemu. A sample is shown below:

```
{  "Android-x86 Gingerbread", /* entry name */
   0x00000000, 0x00000000, /* hooking address */
   0xC02B6F90, /* task struct root */
   1000, /* size of task_struct */
   448, /* offset of task_struct list */
   492, /* offset of pid */
```

The first entry is a name (this is not necessary), the second and third entries are not used and thus are 0. The fourth entry `0xC02B6F90` is the location of `init_struct` which is not necessary either since we traverse the threadinfo structure to get to the task list. The remaining three entries consists of the size of the task struct (so we know how many bytes of guest memory to read), the offset into the object to obtain the next pointer and the offset to obtain the pid.

While it is possible to obtain these values manually, it is much easier to obtain them dynamically using a kernel module for the guest. The actual instructions for doing so are embedded at the end of *read_linux.c* inside the *TEMU/linuxAPI* subdirectory. The basic idea is to simply use a kernel module to print out the values using *printk*. The kernel messages are then obtained through adb using *adb shell* and then *dmesg* or simply *adb shell dmesg* on the host.

Note that the kernelinfo.conf in the VM is preconfigured for the binary distribution in the VM so this is not necessary. Also note that the data structure sizes and offsets should remain the same through different compilations of the same source. This means that you can include the code for dumping these offsets in one build and then remove the code and rebuild the kernel to get a cleaner kernel image.

5.2 Symbol Information

DroidScope relies on good Symbol information. Without knowing where `dvmAsmInstructionStart` is, it is not easy to rebuild the Dalvik context. Once again while it is possible to obtain the symbol information manually it is better to obtain it through automally. To do this, we will use the *setupdumps.sh* script in the *objs* directory in qemu.

This script uses adb to grab all of the shared libraries from */system/lib* and all of the dalvik files from */data/dalvik-cache*. These files are copied into a directory called *dumps* on the host into their respective directories. After a successful pull, the script will then try to disassemble these files using *objdump* and *dexdump* and place the disassembled files into the *dumps/out* directory. DroidScope parses the dumps in this directory to obtain the symbol information. Thus, if you want to make a change to the directory structure for the dumps, you will also need to make the same change in the `TEMU/linuxAPI/ModuleServer.cpp` code.

5.3 Android Source

DroidScope also includes some of the Android source into its source tree. This is all included inside the *TEMU/dalvik* directory tree. These files have been changed from their original implementations (see the comments) so they can be included into DroidScope's source tree. These are included so that it is easier to interpret data structures. While it is possible to rewrite DroidScope to not rely on the Android source, it was easier (and more correct) to simply include the source.

If you are using a different Android source - it is also a good idea to update these files.

5.3.1 Hint

Ice Cream Sandwich uses a *.odex* extension for the dalvik-cache whereas Gingerbread uses *.dex*. So if you are using ICS, make sure you change the setupdumps script.

6 Dalvik Instruction Trace

Please see *TEMU/plugins/DalvikInstructionTrace.c* for the full source. We will only focus on the events in this manual. The important code for us is shown below:

```
void dalvikInstructionTrace_module_updated_callback(CPUState* env, int pid, ModuleNode* node)
{
  //if (!isInWatchContext(env))
  if (pid != getWatchPID())
  {
    return;
  }

  TEMU_fprintf(outputfp, "\n  --  MEMORY MAP CHANGED -- \n\n");
  printModuleList(outputfp, getWatchPID());
  TEMU_fprintf(outputfp, "\n  --  END MAP CHANGED -- \n\n");

  if (!bInitialized)
  {
    //this is to disable JIT
/*
    gva_t startAddr = INV_ADDR;
    gva_t endAddr = INV_ADDR;
    if(findProcessClassesDexFile(pid, &startAddr, &endAddr))
    {
      return;
    }
*/

    gva_t startAddr = 0x43317000;
    gva_t endAddr = 0x43344000;

    addDisableJITRange(pid, startAddr, endAddr);
    addMterpOpcodesRange(pid, startAddr, endAddr);

    //if we are here, these should not fail - perhaps I should check the return
    ibase = getIBase(pid);
    dalvikMterpInit(env, ibase);

    getcodeaddr = getGetCodeAddrAddress(pid);
    disableJITInit(getcodeaddr);
    //this is to get mterp opcodes set up
    //here we can try and find ibase
    bInitialized = 1;

    setTaintFile(outputfp);
    setSyscallFile(outputfp);
    //enableInsnCallbacks();
    //enableLoadStoreCallbacks();
    TEMU_mprintf("INITIALIZED\n");
```

```
  }
}

void dalvikInstructionTrace_init(const char* filename)
{
  processWatcherInit();

  outputfp = fopen(filename, "w");

  registerDalvikInsnBeginCallback(&dalvikInstructionTrace_insn_begin_callback);
  registerWatchProcessModulesUpdatedCallback(&dalvikInstructionTrace_module_updated_callback);

  gInstrWidth = dexCreateInstrWidthTable();
  gInstrFormat = dexCreateInstrFormatTable();

  //setWatchProcessByName(cpu_single_env ? cpu_single_env : first_cpu, "com.greenecomputing.linpack");
}
```

We begin by looking at the *dalvikInstructionTrace_init* function, which should be called in *TEMU_Init.c*. The first thing the tracer does is to initialize the `processWatcher`. processWatcher is a util that registers for process related events and filters them to issue related events for a single process. For example, the last line (which is commented) makes a call to *setWatchProcessByName* to watch for *com.greencomputing.linpack*. This sets the processWatcher up to register for processBegin, processModulesUpdated and etc. events and issue watchProcessBegin, watchProcessModulesUpdated events if those events are generated for *linpack*. This is why we register for watchProcessModulesUpdated events instead of simply modulesUpdated events. We also register for the DalvikInstructionBegin event.

In the *dalvikInstructionTrace_modules_updated_callback* function we do the following. We first do a sanity check to make sure that the process that was updated is the same as the process being watched - this should always be true. We then log the module list (which might or might not have changed). We then determine whether the tracer state has been initialized. If not then we must initialize it. Looking at the code, you will notice that *startAddr* and *endAddr* uses hardcoded addresses. This is for testing. The more general approach has been commented out. In that approach *findProcessClassesDexFile* is used to populate the start and end address variables. What that function does is to first lookup the process name using the pid, then it will run through the shadow module list to find an entry for the processname followed by "@classes.dex" (This won't work for ICS or for some samples where the process name is not the same as the class file name.

Once the *startAddr* and *endAddr* have been set up for the memory region that needs to be instrumented - in this case where the Dalvik Instructions are begin dumped. We will then need to disable JIT for that memory region (*addDisableJITRange*) as well as setup the DalvikAPI to issue Dalvik events for that memory range (*addMterpOpcodesRange*).

Since the DalvikAPI needs to know the address of *getCodeAddr* to disable JIT and *dvmAsmInstruction-Start* (iBase) so the API can rebuild the dalvik opcodes, we use two helper functions to grab those addresses. They are then used to initialize the JIT and Mterp portions of the DavlikAPI accordingly. Please feel free to explore the rest of the code.

One thing to note is that *enableInsnCallbacks* and *enableLoadStoreCallbacks* are commented out in this implementation. These two calls enable instruction level events generation so if they are commented out, the taint propagation logic in *taint_custom.c* will not be invoked. It is an example of dynamically enabling certain events for different purposes.

7 isObjectTainted - exercise

To get a little familiar with the source, we will now discuss how to write a new function called *isObjectTainted*. Inside the file *TEMU/utils/taint_custom.c* are two functions of interest. The first is *do_taint_object* and the

second is *do_check_object_taint*. The first is called as a result of the *taint_object* command in the qemu monitor and the second as a result of the *check_object_taint* command.

Right now the source in the VM does not have the code for *do_check_object_taint* so it needs to be implemented. To do so we will first look at the code for *do_taint_object* and then based on our observations, implement *do_check_object_taint*.

The code for *do_taint_object* is shown below:

```
void do_taint_object(Monitor* mon, uint32_t addr) //we are only trying String objects
{
  StringObject* pSO = NULL;
  ClassObject* pClazz = NULL;

  //first we get the string object at addr
  int ret = 0;
  ret = getObjectAt(addr, (Object**)(&pSO), &pClazz);
  if (ret != 0)
  {
    return;
  }

  //get have the object at that addr we check to see if its a string object
  // and then taint the resulting char array

  uint32_t addr2 = (uint32_t)pSO->instanceData[STRING_INDEX_VALUE];
  size_t count = pSO->instanceData[STRING_INDEX_COUNT];

  //now we get the array object
  Object* pObj = NULL;
  ClassObject* pClazz2 = NULL;

  ret = getObjectAt(addr2, &pObj, &pClazz2);
  if (ret != 0)
  {
    free(pSO);
    free(pClazz);
    return;
  }

  //now this guy should be an array object, so we will have to check that by
  // making sure that the descriptor string is "[C"
  char s[5] = "";
  TEMU_read_mem((uint32_t)pClazz2->descriptor, s, 5);
  if ( (s[0] != '[') || (s[1] != 'C') || (s[2] != '\0') )
  {
    free(pSO);
    free (pObj);
    free (pClazz);
    free(pClazz2);

    return;
  }

  //now that we know its a Character Array, lets make sure that the length is greater
  // than or equal to count - which I AM NOT DOING YET
  //ArrayObject* pAO = (ArrayObject*)(pObj);
```

```
  addTaint(MEM_TYPE_MEM, addr2 + offsetof(ArrayObject, contents) + 4, 1, count * 2);

  free(pSO);
  free(pObj);
  free(pClazz);
}
```

As can be seen from the comments, this function is only used to taint String objects [1]. It first tries to grab a copy of the Java Object at address `addr`. The function takes in two pointer references, one to the Object and the second to the ClassObject. These objects are dynamically allocated by the *getObjectAt* function so make sure you free the memory when you are done with them.

By the time `addr2` is defined, *getObjectAt* has successfully created two new object structures on the host and copied the contents from the guest. Here we simply treat the Object as a StringObject (which might be wrong) and try to get the ArrayObject pointed to by this StringObject. If this Object wasn't a String Object then most likely `addr2` will not be a valid address and thus *getObjectAt* will return an error code. In this case we free the two temporary objects and return [2].

Once the Object has been deemed a char array, we then taint the char array. This is done by the call to *addTaint*. So in essense to check the taint, what we need to do is change the *addTaint* call into a *getTaint* and return the result. Don't forget to free the memory though.

7.1 Tainting All Objects

To taint all objects we need to recognize that *do_taint_object* checks to see if the object is a String and if it does not check out then it simply frees dynamically generated memory and returns. So to taint all objects, we have to taint the object's instanceData field before the function returns. Please refer to *TEMU/dalvikAPI/AndroidHelperFunctions.c* for more hints if necessary.

8 Notes and FAQs

TEMU_mem.c TEMU_mem.c is located within the *qemu/TEMU/nativeAPI* directory and contains the code for accessing guest memory. In most cases this does not need to be changed, although there is one special case where changing it could help with performance.

The TEMU_get_phys_page function is implemented in two ways, one of which is commented out by default. First, it is simply a wrapper for *cpu_get_phys_page_debug* which is the QEMU function that does the same thing. This is the fallback method. Second, the default behavior, is optimized for data reads. Taking a look at the code will show that we first try to obtain the TLB entry for the particular virtual address. If it exists then we translate the virtual to physical directly and if it does not then we simply return -1 or `INV_ADDR`. The reasnoning behind this is that in cases like tainting where we propagate the taint after the instruction has executed, all data access by the instruction should guarantee that the virtual to physical mapping is in the TLB, thus if it is not in the TLB we simply assume that there is no mapping. This behavior can be changed of course.

[1] See Section 7.1 for hints on implementing a more general function.

[2] This is a good opportunity to taint the Object's InstanceData if you want to taint all Objects instead of just String objects.