PANDA Grand Tour

Tim Leek `tleek@ll.mit.edu`

Introduction

In this class, you will use PANDA to investigate the dynamic behavior of program code and the operating system that contains it. PANDA is a whole-system dynamic analysis platform based upon the Qemu emulator. An entire operating system runs under PANDA, which makes it especially useful for situations in which concurrent processes and even kernel activity must be understood. Further, emulation means we can run code for a different CPU than we have on our host system. This means PANDA can, e.g., boot an ARM Linux system on a Intel x86 MacOs or Windows host.

Further, whole system activity can be recorded and replayed, which is a powerful ability. Various analysis plugins that run on the execution replay allow visibility into the emulated guest and provide information about what is going on there. Reverse Engineering with PANDA is typically an iterative process, replaying a recorded execution over and over under different plugins and inspecting output to accumulate better and more complete understanding of behavior.

`https://panda.re`

Overview

This course is introductory in nature. We will use PANDA and the set of existing plugins to examine, in detail, goings on inside an operating system. You will gain experience in recording and replaying system activity involving multiple processes and the kernel. You will make use of

operating system introspection to track processes, decode system calls, and monitor file system activity. You will identify instructions of interest using memory taps. You will pull network activity and executables from recordings for analysis. You will use PANDA'S dynamic taint system to follow crucial data flows.

Note that this course does not involve any coding. That is, developing PANDA plugins is beyond the scope of this course. This course is about getting a feel for what kinds of things PANDA can do already. PANDA's plugins are written in C or C++. There is information online (especially in the PANDA manual) that will help you get started developing plugins.

This course is an interactive and lab-like. It is temporally structured around a series of *cues*, which are simply points at which the instructor will ask you, the participant, to do something. The follow-up to a cue may be some comments by the instructor, or perhaps a brief discussion.

Lab Prerequisites

A shell

You will need a shell in which to type commands. On OSX, this is the included `Terminal` program or something like `iTerm`. On Windows, this is the `CMD` window or perhaps powershell. On Linux, this is a terminal, e.g., `xterm` or `gnome-terminal`.

NOTE: for the remainder of this class, we will refer to all of these various command-entering programs as the *shell*.

Class Materials

You'll need a bunch of support files to take this class. Everything in the `panda_class_materials` directory. The class organizer will tell you how to get this either from a USB drive being passed around the class or from a network file share or the internet or whatever. Please copy that directory into your *home* directory. If you are on Linux or OSX, this will mean the materials ends up in `~/panda_class_materials`. If you are on Windows, they should be in `C:\User\YourUsername`, i.e., your home directory.

Docker

You will be running PANDA inside a `docker` container. If you don't already have docker installed, you will need to get it.

1. For OSX or Windows, install docker desktop: osx docker desktop or win docker desktop.
2. For Linux, install with something like `apt-get install docker`. Basically, if you are running Linux you probably don't need to be told how to install docker; you came out of the womb knowing how to do this.

For windows, you may need to log out and log back in after installing docker desktop.

PANDA Docker image

This PANDA class will take place entirely from a shell inside a docker container, which is available on Docker hub. You may be working from a laptop provided by the organizers, in which case you can skip this section. Otherwise, obtain the docker container, now, with

```
docker pull pandare/panda_class_2022
```

PANDA First Gear -- A Whole Operating System At Our Command

In this section of the class, we will boot an entire OS under PANDA and play around with it a bit. First we need a shell inside the PANDA Docker container.

On OSX or Linux, here's how to get that PANDA docker shell.

```
panda_class_materials:/panda_class_materials pandare/panda_class_2022 bash
```

On Windows, here's how to get that PANDA docker shell:

```
panda_class_materials:/panda_class_materials pandare/panda_class_2022 bash
```

The `-v` part, here, makes the class materials directory available inside the docker container at `/panda_class_materials`. Here's what you should see if you get a PANDA docker shell.

```
root@ac6aadd740c8:/#
```

Now, from within that docker container, we will use PANDA to boot an operating system.

CUE #1: Boot Linux with PANDA

Start a timer and then enter the following in your docker shell.

```
/# panda-system-i386 -nographic /panda_class_materials/qcows/wheezy.qcow2
```

After a few seconds, you should see the Linux `grub` boot screen. Just wait a little longer and the Linux operating system will start to boot.

You will see lots of log messages scroll by as the Linux guest boots.

CUE #2: What booting Linux under PANDA looks like

```
Loading Linux 3.2.0-4-686-pae ...
Loading initial ramdisk ...
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.2.0-4-686-pae (debian-kernel@lists.debian.or
[    0.000000] BIOS-provided physical RAM map:
[    0.000000]  BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
[    0.000000]  BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
[    0.000000]  BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
[    0.000000]  BIOS-e820: 0000000000100000 - 0000000007fe0000 (usable)
[    0.000000]  BIOS-e820: 0000000007fe0000 - 0000000008000000 (reserved)
[    0.000000]  BIOS-e820: 00000000fffc0000 - 0000000100000000 (reserved)
[    0.000000] Notice: NX (Execute Disable) protection missing in CPU!
[    0.000000] SMBIOS 2.8 present.
...
```

When boot is complete, you should see a login prompt.
Stop your timer when your panda guest reaches this point.

```
Debian GNU/Linux 7 debian-i386 ttyS0

debian-i386 login:
```

CUE #3: How long did Linux take to boot with PANDA?

For refererence, booting a 64-bit more modern version of Linux under Vmware Fusion takes about 20 seconds. Emulation slows things down.

CUE #4: Log into that Linux guest.

Finally, go ahead and log in with username `root` and password `root`. You should see a prompt line like this:

```
root@debian-i386:~#
```

PANDA Second Gear -- Record and Replay All The Things

In this section of the class, we are going to create a recording of something happening in the guest operating system. PANDA records whole-system activity by first taking a guest snapshot, and subsequently logging all interactions with the outside world in an *nondeterminism* log. To replay, PANDA reverts to the snapshot and then taking items from the nondeterminism log, rather than the outside world. This is a core PANDA capability that is very useful.

1. Recordings can be replayed over and over and the same sequence of instructions execute in the same order every time, with pointer addresses, processes, and so on all manifesting in the same way and in the same order as they did during recording. This enables iterative dynamic reversing in which you can build up knowledge.
2. A recording can be replayed with as much and as heavy instrumentation or analysis as desired. This is not true of live execution.
3. Recordings can capture weird, ephemeral, or dangerous activity, such as the activity of malware or anti-virus.
4. Recordings can be shared with other reverse engineers and results will correspond: the same processes with the same PIDs and all the same pointer addresses and all the same instructions at the same times.

Whilst interacting *live* with a PANDA instance, we can create a recording of whole-system activity using its *monitor*. The monitor allows you to interact with the emulation engine via a command-line interface. If you are familiar with Qemu, this monitor is simply Qemu's monitor augmented with additional PANDA commands. There is a lot you can do from the monitor: halt, resume, snapshot or restore a guest, inspect registers or memory, plug in a USB device of CD, etc.

Return to that docker shell in which you ran PANDA and booted Linux. Make sure you are still logged in to the guest.

First things first, you are going to need to know how to enter and exit the monitor. It's very simple.

CUE #5: To enter or exit the monitor, type `Ctrl-a`, followed by the letter `c`.

When you enter the monitor, you will see the following prompt: `(qemu)`. That's how you can

tell that you are in the monitor. Try this a few times right now to get the hang of it: switching back and forth between the guest Linux OS and the monitor. Note: you'll have to hit enter in the Linux guest to see its prompt after switching to it.

Now let's create a PANDA recording. Switch to the monitor and tell PANDA to begin a recording named "commands" by typing the following.

CUE #6: Start a recording using the monitor

```
(qemu) begin_record commands
```

PANDA should respond with

```
(qemu) writing snapshot:    ./commands-rr-snp
opening nondet log for write :  ./commands-rr-nondet.log
```

Now switch from the monitor back to the Linux guest.

```
root@debian-i386:~#
```

Next, execute the following three commands, in sequence, in the guest.

CUE #7: Enter these commands into the PANDA Linux guest

1. `ls -lt /etc`
2. `ps -ef`
3. `netstat -a`

Lastly, return to the monitor and end the recording

CUE #8: End the recording using the monitor

```
(qemu) end_record
```

To which, PANDA will respond with something like

```
(qemu) Time taken was: 29 seconds.
Checksum of guest memory: 0xd4acf418
```

At this point, you can shut down that Linux guest you booted under PANDA with the following command.

```
root@debian-i386:~# /sbin/shutdown -h now
```

Shutdown should take about 10-15 seconds, after which, you will get your docker shell prompt back. Congratulations! You just created a recording of a whole lot of whole-system activity with PANDA.

You can replay all that activity with the following command from the docker shell.

CUE #9: Replay the execution of those three commands you entered into the guest

```
root@ac6aadd740c8:/# panda-system-i386 -replay commands
```

You will see output indicating replay progress: percent of nondeterminism log replayed, instruction count, etc. It will look something like this.

CUE #10: What PANDA replay looks like

```
loading snapshot
... done.
opening nondet log for read :   ./commands-rr-nondet.log
./commands-rr-nondet.log:    34403335 instrs total.
commands:      344035 (  1.00%) instrs.    0.18 sec.  0.15 GB ram.
commands:      688073 (  2.00%) instrs.    0.20 sec.  0.15 GB ram.
commands:     1032103 (  3.00%) instrs.    0.22 sec.  0.15 GB ram.
commands:     1376139 (  4.00%) instrs.    0.26 sec.  0.15 GB ram.
commands:     1720168 (  5.00%) instrs.    0.30 sec.  0.16 GB ram.
commands:     2064203 (  6.00%) instrs.    0.34 sec.  0.16 GB ram.
...
commands:    32339138 ( 94.00%) instrs.    1.82 sec.  0.18 GB ram.
commands:    32683170 ( 95.00%) instrs.    1.83 sec.  0.18 GB ram.
commands:    33027203 ( 96.00%) instrs.    1.84 sec.  0.18 GB ram.
commands:    33371239 ( 97.00%) instrs.    1.86 sec.  0.18 GB ram.
commands:    33715271 ( 98.00%) instrs.    1.87 sec.  0.18 GB ram.
commands:    34059308 ( 99.00%) instrs.    1.88 sec.  0.18 GB ram.
./commands-rr-nondet.log:  log is empty.
./commands-rr-nondet.log:  log is empty.
Replay completed successfully. 1
Time taken was: 2 seconds.
Stats:
RR_INPUT_1 number = 0, size = 0 bytes
RR_INPUT_2 number = 0, size = 0 bytes
RR_INPUT_4 number = 6909, size = 96726 bytes
RR_INPUT_8 number = 13942, size = 250956 bytes
RR_INTERRUPT_REQUEST number = 3753, size = 52542 bytes
RR_EXIT_REQUEST number = 0, size = 0 bytes
RR_SKIPPED_CALL number = 29079, size = 908845 bytes
RR_END_OF_LOG number = 1, size = 10 bytes
RR_PENDING_INTERRUPTS number = 0, size = 0 bytes
RR_EXCEPTION number = 0, size = 0 bytes
max_queue_len = 193
Checksum of guest memory: 0x173e5d95
Replay completed successfully 2.
```

The specifics of your replay will likely be different. In the one pictured above, 1.88 seconds and over 34 million instructions were required to replay everything that happened, which is the execution of those three commands in sequence. That may seem a lot. That's because it is. Remember that replay included all program, library, and kernel code that executed to get the business of those commands done. It also included all the background activity, including daemons and the kernel doing its thing.

Take a look at the size of the replay you just created. It consists of two files, really:

1. `commands-rr-snp` : a snapshot of the state of the operating system at the time the recording begain (RAM + registers + a little more)
2. `commands-rr-nondet.log` : the log of nondeterministic inputs that has to be replayed

In my case, the snapshot is about 80MB and the nondeterminism log is about 1.3MB. This means we need just over 80MB to completely specify a perfect recording of over 34M instructions. Practically, what this means is that sharing replays is easy and common.

CUE #11: Is there anything weird about how long the replay of `commands` takes?

Yes. Typing in those commands took 10s of seconds yet replay was more like a second or two. This is because replay can effectively compress or discard periods of inactivity on the guest. It is not uncommon for a replay to run in less time than it took to take a recording.

There are simple things we can do to peek into that replay, to get more out of it than just how many instructions executed. For example, we can add a little to the command-line we just used and PANDA will spit out all the code that was executed plus a little more information.

CUE #12: Replay again, writing out a trace of all the code that executes, system-wide

```
root@ac6aadd740c8:/# panda-system-i386 -replay commands -d in_asm,rr,int |&
loading snapshot
... done.
opening nondet log for read :   ./commands-rr-nondet.log
Servicing hardware INT=0xef
     0: v=ef e=0000 i=0 cpl=0 IP=0060:c10243a0 pc=c10243a0 SP=0068:c13dbfd0
EAX=00000000 EBX=c14170ac ECX=c7ed0f80 EDX=00000003
ESI=00000000 EDI=c13de000 EBP=01989003 ESP=c13dbfd0
EIP=c10243a0 EFL=00000246 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =007b 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
CS =0060 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0068 00000000 ffffffff 00c09300 DPL=0 DS   [-WA]
DS =007b 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
FS =00d8 06a54000 ffffffff 008f9300 DPL=0 DS16 [-WA]
GS =00e0 c7ed5940 00000018 00409100 DPL=0 DS   [--A]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0080 c7ed3780 0000206b 00008900 DPL=0 TSS32-avl
GDT=     c7ece000 000000ff
IDT=     c13de000 000007ff
CR0=8005003b CR2=080ebed0 CR3=06d28000 CR4=000006b0
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00000044 CCD=00000000 CCO=EFLAGS
EFER=0000000000000000
```

```
----------------
IN:
0xc12c4648:  push   0xffffff10
0xc12c464d:  cld
0xc12c464e:  push   gs
0xc12c4650:  push   fs
0xc12c4652:  push   es
0xc12c4653:  push   ds
0xc12c4654:  push   eax
0xc12c4655:  push   ebp
0xc12c4656:  push   edi
0xc12c4657:  push   esi
0xc12c4658:  push   edx
0xc12c4659:  push   ecx
0xc12c465a:  push   ebx
0xc12c465b:  mov    edx,0x7b
0xc12c4660:  mov    ds,edx

Prog point: 0xc12c4648 {guest_instr_count=0}
Prog point: 0xc12c4648 {guest_instr_count=0}
----------------
IN:
0xc12c4662:  mov    es,edx

Prog point: 0xc12c4662 {guest_instr_count=15}
...
```

You can page through this for a while and you will see blocks of x86 code disassembled. If you were very patient you could page through all of it. But there is a lot -- over 7 million lines for the replay pictured above.

That kind of output seems like it would be useful, and sometimes it absolutely is. Mostly it isn't. That's because there is simply too much of it and it is very low level without any context. What instructions correspond to the `ls`, `ps`, and `netstat` processes? We have no idea. Currently, our view of this replay is just the sequence of machine instructions plus a few other things like when interrupts and exceptions happen.

PANDA Third Gear -- Operating System Introspection

Let's use some PANDA plugins to get a little more out of that replay. First, we'll use a stock plugin called `asidstory`, so named because it generates an ASCII art picture of the

temporal *story* of what happened during a replay, in terms of the *asid*s of the processes that executed. `asid` stands for _Address _Space _IDentifier. This is the contents of a specific machine register that points to the virtual page tables for a process. The `asid` identifies a process uniquely. Here is the command line you can use to generate that ASCII art asidstory.

CUE #13: Run your commands replay with the `asidstory` plugin using this command-line

```
root@ac6aadd740c8:/# panda-system-i386 -replay commands -os linux-32-debian
```

You should see the replay scroll by as before. It will take a little longer this time, since now an analysis is happening alongside thre replay. After it completes, you can inspect the output which is in the file `asidstory`. Your output should look somewhat like the following.

```
root@ac6aadd740c8: cat asidstory
   Count   Pid              Name/tid      Asid     First          Last
     172  2506          [ps  / 2506]   6f34000  12249520  ->  24314310
     129  2505          [ls  / 2505]   6f3c000   1963133  ->  10908122
      91  2463        [bash  / 2463]   5d7f000     80041  ->  32376444
      85  2507     [netstat  / 2507]   57eb000  26004830  ->  31797780
       7     1           [init  / 1]   6d28000    418250  ->  11724599
       3  1711     [rpcbind  / 1711]   7a8f000   1225712  ->   1243670

      [bash  / 2463] : [#####                    ####
         [init  / 1] : [##                        #
   [rpcbind  / 1711] : [  #
        [ls  / 2505] : [    ######################
        [ps  / 2506] : [                           ######################
   [netstat  / 2507] : [
```

Let's look at the bottom half of that output first, which is a kind of 2D ASCII plot. The vertical axis of the plot has a row for each process identified during the replay. The horizontal axis is time. The far left corresponds to the start of the replay and the far right is the end of the replay. Time is divided up into a number of cells. A hash mark, #, in a cell indicates that a process was seen to be executing at least once during that interval of the replay. If you scan back to the sequence of commands you executed, you should see them reflected in that same sequence in your `asidstory`. First, `ls` executes, then we see `ps`, and, finally, `netstat` runs. The shell, `bash`, executes in between.

This richer understanding of what is going on inside the guest is made possible by *operating system introspection* (OSI). The plugin `asidstory` requires it (invisibly) and this is why part of the command line provides PANDA with information about the guest OS: the kernel version,

the CPU and whether this OS is 32 or 64 bits (that's this part of the command line: `-os linux-32-debian:3.2.0-4-686-pae`). OSI is used all over the place in PANDA plugins as it is so useful. In this case, it allows the code for `asidstory` to be written once, in a few hundred lines of C++, in an operating system neutral way. The same plugin code works for Windows OS replays.

The top part of the output of `asidstory` provides more precise information. You can determine the range of instructions over which any process was observed during the replay, e.g. You can also see the PID and `asid` as well as a count of the number of times each process was observed. Note that the three commands you ran by hand make up the lion's share of the replay.

PANDA Fourth Gear -- Dynamic String Search

In this section, you will use the `stringsearch` PANDA plugin to do a little light reverse engineering.

`stringsearch` allows you to identify exactly what code performs some calculation, given that you can control or at least predict the input or output of that calculation (or some intermediate data). This might mean you can control the password typed in to a program (even if you don't know the password). Or it might mean you know that the uncompressed mp3 data for Rick Astley's "Never Gonna Give You Up" will be output and thus played out of your speaker.

You first create a PANDA recording in which that input or output is guaranteed, by construction, to be read from or written to memory at some point. Then, you replay the recording with the `stringsearch` plugin, providing, as a parameter, the known data. The plugin reports any string matches it encounters, corresponding to it having seen that data read or written by some instruction.

What good is all this? Sometimes, merely isolating the code performing some calculation is a big reverse engineering win. For instance, precisely identifying the code that de-obfuscates a string in malware, or that encrypts exfiltrated data. Another use for `stringsearch` is building introspection gadgets without requiring any domain knowledge or any reverse engineering work. That's what we are going to do now.

You will recall that, when we booted that Linux OS image before, it output a bunch of messages. These are useful, since they provide insight into what is going on with system set up and when. However, if you took a recording of boot and tried to play it back, you wouldn't have them. Or, if someone handed you a recording purportedly of boot and you replayed it, you'd just see replay progress messages. Note that you can forget about using `asidstory` to help here since it

won't tell you much of use until boot is nearly complete. There are no processes during most of boot, just a kernel.

There is a recording of boot for the same guest you have been working with in `panda_class_materials/replays/boot`.

CUE #14: Use what you've learned so far to replay that `boot` recording

Note that the replay is annoying in that you dont get those nice boot messages like you did before.

Scroll back in your terminal to see what some of those boot messages were. Note that some of them are pretty unique. Choose one, preferably near the beginning of the replay, and replay with something like the following commands.

CUE #15: Replaying using the `stringsearch` plugin

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

You should see output something like this.

```
...
boot:    47432553 (  1.00%) instrs.    5.27 sec.  0.09 GB ram.
boot:    94865100 (  2.00%) instrs.   10.55 sec.  0.09 GB ram.
boot:   142297662 (  3.00%) instrs.   16.11 sec.  0.10 GB ram.
boot:   189730201 (  4.00%) instrs.   21.48 sec.  0.10 GB ram.
boot:   237162751 (  5.00%) instrs.   27.07 sec.  0.11 GB ram.
boot:   284595306 (  6.00%) instrs.   31.08 sec.  0.11 GB ram.
boot:   332027849 (  7.00%) instrs.   35.12 sec.  0.11 GB ram.
READ Match of str 0 at: instr_count=346437193 :  01510e00 0150edb8 (asid=0x
WRITE Match of str 0 at: instr_count=346437193 :  01510e00 0150edb8 (asid=0
READ Match of str 0 at: instr_count=347091916 :  c116354a c1165818 (asid=0x
READ Match of str 0 at: instr_count=347092742 :  c1164178 c1163572 (asid=0x
WRITE Match of str 0 at: instr_count=347092743 :  c1164178 c1163575 (asid=0
READ Match of str 0 at: instr_count=347095970 :  c12bf1e5 c10390c0 (asid=0x
READ Match of str 0 at: instr_count=347095974 :  c12bf1e5 c10390a8 (asid=0x
WRITE Match of str 0 at: instr_count=347095980 :  c10390b0 c1038840 (asid=0
READ Match of str 0 at: instr_count=347095999 :  c12bf1e5 c10390b0 (asid=0x
READ Match of str 0 at: instr_count=347100561 :  c103912c c1038bdf (asid=0x
READ Match of str 0 at: instr_count=369467817 :  c1039776 c1038bdf (asid=0x
READ Match of str 0 at: instr_count=369471989 :  c1038654 c11dd4d9 (asid=0x
WRITE Match of str 0 at: instr_count=369471993 :  c1038654 c11dd4e2 (asid=0
READ Match of str 0 at: instr_count=369471998 :  c1038654 c11dd4f1 (asid=0x
READ Match of str 0 at: instr_count=369472004 :  c1038654 c11dd5c5 (asid=0x
READ Match of str 0 at: instr_count=369472008 :  c1038654 c11dd5d5 (asid=0x
READ Match of str 0 at: instr_count=369999831 :  c1039776 c1038bdf (asid=0x
READ Match of str 0 at: instr_count=370003178 :  c11e43e9 c11e076a (asid=0x
READ Match of str 0 at: instr_count=370003180 :  c11e43e9 c11e077b (asid=0x
boot:   379460398 (  8.00%) instrs.   42.67 sec.  0.16 GB ram.
boot:   426892948 (  9.00%) instrs.   53.04 sec.  0.18 GB ram.
...
```

You can stop the replay after you have seen that big block of read/write matches and start seeing just the percent progress indicators. You won't see any more matches.

What that output is telling you is that `stringsearch` worked as advertised. It found a bunch of replay instruction counts at which the string you gave it to search for was observed passing through a memory read or write instruction. The hex numbers near the end of each match are the program counter of the read or write instruction and the first address on the callstack, indicating some of the context from which this read or write instruction was executed. Note that what we really have here in the output of `stringsearch` is a trace of the data-flow journey this particular log message experienced. Probably the later matches correspond to kernel print function handling of the string which we may want to hook as our gadget.

If we were being scientific, we might run `stringsearch` twice, with two different seemingly unique log messages and look for the intersection, since that would be more likely to contain the boot log tap point we want. In this way, we might tease apart the different dataflow journeys of the different messages and find where they converge, thus indicating a good tap point for extracting boot logs.

Now we are going to use the `stringsearch` output to fashion a boot log extractor that will work on this or any other replay from this operating system image. We will use the `textprinter` plugin (a companion to `stringsearch`) which logs anything read or written by the instructions you specify in a magic file: `tap_points.txt`. Create that file, now, and put the following in it, which identifies that last read match as the one from which to extract log messages.

CUE #16: Create `tap_points.txt` file required by `textprinter`

```
root@ac6aadd740c8:/# cat > tap_points.txt
0
c11e43e9 c11e077b
Ctrl-C
```

And then run the `textprinter` plugin on your replay.

CUE #17: Replay using `textprinter` to collect tap point data, i.e., boot log messages.

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

You can safely halt replay at around 25% by pressing `Ctrl-C`. Don't worry you will still get plenty of log messages. The output of `textprinter` is in the file `read_tap_buffers.txt.gz` which you can look at with `zcat`, if you like, but it is a bit hard to follow. Instead, we'll use the `render_log.py` script provided in the `panda_class_materials` to reconstruct the log. Here's the contents of that script, in case you are curious. It's quite simple.

```
import sys
import fileinput
for line in fileinput.input():
    parts = line.split()
    c = int(parts[-1], 16)
    if c>0 and c<256:
        sys.stdout.write(chr(c))
```

Use it now, using the following cmdline, to extract boot log out of `textprinter`'s output.

CUE #18: Reconstruct boot logs using script `render_log.py`.

```
root@ac6aadd740c8:/# zcat read_tap_buffers.txt.gz | python /panda_class_mat
```

Congratulations again! You just built your first introspection gadget for PANDA.

PANDA Fifth Gear -- Reverse Engineering

As we noted earlier, PANDA's record/replay allows one to share interesting whole-system activity with another reverse engineer. There is just such a recording in `/panda_class_materials/replays/commands_wtf`. This was created by a colleague trying out an early version of the activities in this class. She followed the directions above to create a recording of executing those three commands, in sequence, just as you did earlier. However, she was concerned about the output she got from the `asidstory` plugin. Let's look into it.

CUE #19: Run PANDA with `asidstory` on the `commands_wtf` replay now.

Now take a look at the output in the file `asidstory`. It looks pretty different from before, right? You can still see the three commands running: `ls`, `ps`, and `netstat`. But now there's some periodic activity layered on top and a bunch more `bash` processes than before. We also see a program called `ignorame` running, which sounds weird. Plus a few we may find familiar: `nc` and `sleep`.

CUE #20: Does this output look at all worrying to you?

Does anyone have a theory about what UNIX pattern might have caused this output? Is anything weird or worrying here?

Let's peer a little closer into the activity of the new program `ignorame`. First, we'll extract some necessary info from that `asidstory` output.

CUE #21: Determine pid, asid, and instruction counts for `ignorame`

Use these commands to focus on just `ignorame` in that output.

```
root@ac6aadd740c8:/# grep ignorame asidstory | grep ">"
  60  2797  [bash ignorame  / 27   632d000  53546067  ->  58119005
  50  2792    [ignorame  / 2792]   632a000  23583905  ->  27478909
  48  2788    [ignorame  / 2788]   57d0000   3774483  ->   7493465
```

The second column, here, is the pid or process id. The 4th column is the `asid` for the `ignorame` process, which runs three times in our replay. The instruction counts during which each invocation was observed are at the end of the line. Thus, the *first* invocation of `ignorame` has pid `2788`, asid `57d0000`, and executes from instruction count `3774483` to `7493465`.

Now use the plugin called `syshist`, which outputs a histogram of counts for system calls, by asid.

CUE #22: Run PANDA with the `syshist` plugin to get per-process system call stats

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

This plugin, like `asidstory`, produces output in a file of its own name. Let's look at the subset of output for that 1st `ignorame` process.

CURE #23: Examine histogram of system calls for `ignorame`

```
root@ac6aadd740c8:/# grep 57d0000 syshist
57d0000 sys_read 2
57d0000 sys_write 2
57d0000 sys_open 4
57d0000 sys_close 2
57d0000 sys_access 3
57d0000 sys_brk 3
57d0000 sys_munmap 1
57d0000 sys_mprotect 4
57d0000 sys_getdents 2
57d0000 sys_mmap_pgoff 7
57d0000 sys_stat64 4
57d0000 sys_fstat64 3
57d0000 sys_set_thread_area 1
57d0000 sys_exit_group 1
```

This provides a high level view of what `ignorame` does. It reads and writes files, reads directory and file information, asks for memory. The number after each call is a count of the number of times that system call was observed. So it's not making tons of system calls.

`ignorame` seems pretty innocuous. Well, maybe.

We can get a little more insight into what is going on by using the `filemon` plugin which monitors reads and writes to files. You can probably guess where its output goes: into a file called `filemon`.

CUE #24: Running PANDA with the `filemon` plugin to capture file activity.

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

This plugin will log all file read and writes. We can restrict our attention to just output for that first version of the process `ignorame` using its pid.

CUE #25: Examine file activity for `ignorame`

```
root@ac6aadd740c8:/# grep 2788 filemon
read-enter-2788-2788-3-ignorame-26 "/lib/i386-linux-gnu/i686/cmov/libc-2.13
read-return-2788-2788-3-ignorame-26
read-enter-2788-2788-4-ignorame-31 "/home/joeuser/.ssh/id_rsa"
read-return-2788-2788-4-ignorame-31
write-enter-2788-2788-1-ignorame-34 "pipe:"
write-enter-2788-2788-1-ignorame-35 "pipe:"
write-enter-2788-2788-1-ignorame-36 "pipe:"
write-enter-2788-2788-1-ignorame-37 "pipe:"
```

Ok we are getting somewhere. This output is pretty sketchy. The program `ignorame` appears to be reading someone's ssh private key and then writing something to a pipe. But wait, there's more. The plugin `filemon` *captures* the contents of those read/writes, in files named `read-return-2788-2788-4-ignorame-31` etc. Let's look at one of those files.

CUE #26: Examine the file reads and writes by `ignorame` pid 2788 and hypothesize what might be going on.

```
root@ac6aadd740c8:/# cat read-return-2788-2788-4-ignorame-31
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA0KXYGCTbeIgv+ljVh4QoiskVjJhG0+L+hudlTkOzpE3w8YSP
SKUdx5jWsxe4I/8JthcCqMA/zvbIGWIE5ciYmwGgqnGQiEyzRmHk39857FrxM8gL
oinRpIQz0oJxwNLqGDoPBKjk/mvXyL35shK/tK1SyhNCT04J37wKdd0sZCTCju9W
gjseM3inEmm6CXymC2twAEkqcnKc4+Zme8CBvc1wH3UJPaJt657X+Z/pDNX1aH7S
DgCVzuFtrGMT6LbzHs5WUBkuOiR+CumL8QH2RbsXlBVw/6RSqjCg7s+WLdx1vViR
fQnOq5Wd35ZsiPtTgXxOx0gnklD3ga5jpJbELwIDAQABAoIBAHEGxi8eGD4NGGB3
qwSEVSoJkEDfZM73koYl570j87a3+iP7eVsTBzTO2M+fs8LcL7iScoBT1L0a9n65
mImZxwGBOjXa2z7avesMsN2NoWUmRq+UWKj6GnUSoLgSreiU2PdSzDvr+lDQvs4i
...
root@ac6aadd740c8:/# cat write-enter-2788-2788-1-ignorame-35
00 31 33 19 39 01 14
73 7b 41 67 49 24 27 2d 17 18 4c 18 37 27 42 3a 0c 1f 3a 0b 30 23 42 21 1b
06 2f 01 09 56 12 15 2b 1b 3d 1d 29 0a 2d 01 5e 75 0c 01 2f 5c 25 20 1c 3d
...
```

Well we certainly see the ssh private key being read. And we also see a bunch of hex numbers separated by spaces being written to a pipe.

Note that we also know that `nc` was running periodically, from our inspection of the `asidstory` output. This means something might be being written to the network. We can use PANDA's `network` plugin to inspect that activity. This plugin sends its output, in binary form, to a `pcap` file.

CUE #27: Run PANDA with the `network` plugin.

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

One way of looking at this network capture is with `tcpick`:

```
root@ac6aadd740c8:/# tcpick -C -yP -r commands_wtf.pcap | less -R
Timeout for connections is 600
tcpick: reading from commands_wtf.pcap
1       SYN-SENT        10.0.2.15:36637 > 192.168.80.1:5144
1       SYN-RECEIVED    10.0.2.15:36637 > 192.168.80.1:5144
1       ESTABLISHED     10.0.2.15:36637 > 192.168.80.1:5144
00 31 33 19 39 01 14
73 7b 41 67 49 24 27 2d 17 18 4c 18 37 27 42 3a 0c 1f 3a 0b 30 23 42 21 1b
...
```

CUE #28: Examine this output. Anything look familiar?

That's right. The output of `ignorame` in the form of those hex numbers is getting sent out on the network. The `network` plugin actually provides a little more info in the form of comments attached to packets. These comments indicate the instruction count at which network packets went out. If we examine this pcap using `less`, we see a lot of garbage but also what is clearly those hex numbers being output on the network and, most important, we can see where in the replay this is happening.

```
root@ac6aadd740c8:/# less commands_wtf.pcap
...
...Guest instruction count: 9369201...
...
... 00 31 33 19 39 01 14
73 7b 41 67 49 24 27 2d 17 18 4c 18 37 27 42 3a 0c 1f 3a 0b 30 23 42 21 1b
```

This tells us that the first time a bunch of `ignorame` 's output goes out on the network is at instruction count 9369201.

CUE #29: See if you can use the output of `asidstory` to figure out which process is running at that instruction count.

If you aren't good at comparing large numbers, there's obviously a better way to do this, which is to write a python script to analyze that `asidstory` output, in other words to determine which processes were operating at a particular instruction count. Such a script is provided in the class materials directory, which you should study at your leisure. For now, here's how to use it.

```
root@ac6aadd740c8:/# python /panda_class_materials/scripts/find_procs_at_in
      73  2444        [bash  / 2444]   533d000   2100932  ->  52321843
      65  2772        [bash  / 2772]   6f20000    168545  ->  59594525
      58  2789     [bash nc  / 2789]   784c000   3831722  ->  10075072
       9     1           [init  / 1]   6d28000   2071859  ->  52468906
```

The `bash` and `init` processes aren't likely to be the culprit. So that means `nc` is probably the one writing out those hex numbers on the network.

CUE #30: `filemon` captured reads and writes for all processes. You may also want to examine what it collected for `nc`.

So to sum up, we know that

1. `ignorame` reads `joeuser` 's `id_rsa`, which is naughty
2. `ignorame` writes a bunch of hex numbers separated by spaces to a pipe
3. Sadly, these hex numbers aren't just the ASCII for the private key. They look like junk.

4. `nc` writes those same hex numbers out on the network

One explanation that fits these facts is that `ignorame` is encrypting `id_rsa` before sending it out on the network. We have two ways of exploring this possibility with PANDA. The first involves extracting the binary from the replay and then spending time doing static analysis. PANDA's `memsavep` plugin can dump the guest's physical memory at a particular replay instruction. You can use this dump with another tool (not part of PANDA) called `volatility` (built for forensic analysis) in order to extract the binary for `ignorame` from that dump. We aren't going to pursue that avenue in this class since it veers off into static analysis and our bailywick is dynamic analysis. If you are interested in that kind of thing, we can reccommend `Ghidra`, a software reverse engineering tool, which you can obtain here: https://www.nsa.gov/resources/everyone/ghidra/.

PANDA Top Gear (Ludicrous Speed) -- Dynamic Taint Analysis

This piece of malware, `ignorame` happens to be tiny and quite simple. But what if it weren't? What if it were a few megabytes worth of deliberately convoluted code of which only a small portion were of concern to you? Say you wanted to understand the encryption used on `id_rsa` before sending it out on the network?

PANDA has a sophisticated dynamic taint analysis built in that can help here. The taint system allows you to apply labels to data in a replay and then track those labels as data flows through the system. Labels are tracked through copies, between processes, into and out of the kernel. When computation happens involving tainted data, the system collects sets of labels indicating derivation from labeled taint sources.

Let's make this concrete by using PANDA's taint to analyze this encryption a little. Be warned that taint analysis is slow since it is powerful and precise. It will slow down your replay by 20-40x. Thus, first, we will want to pull out a small portion of the replay to work with, if possible. Let's use PANDA's `scissors` plugin to do that. We merely specify the start and end instruction count (see `asidstory` subset from CUE #21) and a name for the new replay files. Probably, you want to subtract a few 10s of thousands of instructions from the start instruction and add a few tens of thousands to the end instruction, just to be safe.

CUE #31: Use PANDA `scissors` plugin to extract a smaller section replay just containing `ignorame` execution

```
root@ac6aadd740c8:/# panda-system-i386 -replay /panda_class_materials/repla
```

This should create a scissored recording in the current directory called `ignorame`. We are

now going to use two plugins that rely upon the `taint2` plugin's underlying taint analysis. `file_taint` will provide us the ability to apply taint labels to a particular file being read. The first byte in the file gets the label `1`, the second gets the label `2`, which we call *positional* labels. We will be applying those labels to the bytes read from `id_rsa`. The `tainted_instr` plugin identifies and logs all instructions that use that tainted data.

CUE #32: Replay `ignorame` with `file_taint` and `tainted_instr` plugins

```
root@ac6aadd740c8:/# panda-system-i386 -replay ignorame -os linux-32-debia
```

The output of `tainted_instr` goes in a file named `ignorame.plog`. This log is in the `pandalog` format, which uses Google protocol buffers under the hood but adds compression as well as the ability to rewind and fast forward. You can examine this log directly with the script `plog_reader.py`:

CUE #33: Simple pandalog viewing with `plog_reader`

```
root@ac6aadd740c8:/# python -m pandare.plog_reader ignorame.plog | less
{
  "pc": "3077637219",
  "taintedInstr": {
    "callStack": {
      "addr": [
        "3077637219",
        "134514782",
        "3076918854",
        "134513937"
      ]
    },
    "taintQuery": [
      {
        "tcn": 0,
        "uniqueLabelSet": {
          "ptr": "140220515913728",
          "label": [
            0
          ]
        },
        "ptr": "140220515913728",
        "offset": 0
      }
    ]
  },
  "instr": "928658"
},
```

It isn't advisable to use `plog_reader` to peruse this log, since it is a lot of information. Here, it generates tens of millions of lines of output. Instead, one typically writes a small script that uses the class `PLogReader` from `plog_reader` to navigate the log, collect, filter, and summarize its output. There is such a script in `/panda_class_materials/scripts/taint.py` which you should run now.

CUE #33: Pandalog analysis via script.

```
root@ac6aadd740c8:/# python /panda_class_materials/scripts/taint.py ./ignor
tainted instr @ pc=80486f7 count=75555
tainted instr @ pc=80486fa count=50370
tainted instr @ pc=8048709 count=75555
tainted instr @ pc=804870d count=75555
tainted instr @ pc=804870f count=50370
tainted instr @ pc=8048738 count=75555
tainted instr @ pc=8048768 count=75555
tainted instr @ pc=804876e count=25185
tainted instr @ pc=8048777 count=75555
tainted instr @ pc=8048800 count=5037
tainted instr @ pc=8048803 count=10074
tainted instr @ pc=8048809 count=6716
tainted instr @ pc=b768c03c count=10074
tainted instr @ pc=b768c0d5 count=23952
tainted instr @ pc=b768c0d8 count=20958
tainted instr @ pc=b768c0dc count=20958
tainted instr @ pc=b768c0e1 count=5988
tainted instr @ pc=b768c0e3 count=2994
tainted instr @ pc=b768d838 count=11753
tainted instr @ pc=b768d83f count=1679
tainted instr @ pc=b768f980 count=8395
tainted instr @ pc=b769012b count=10074
tainted instr @ pc=b7690140 count=6716
tainted instr @ pc=b76b99a0 count=8976
tainted instr @ pc=b76b99a4 count=5984
tainted instr @ pc=b76bc440 count=6
tainted instr @ pc=b76bc444 count=4
tainted instr @ pc=b7713463 count=1679
tainted instr @ pc=c10d2b7d count=9
tainted instr @ pc=c10d2b81 count=2
tainted instr @ pc=c1165e56 count=7981
```

The output of this script is the set of instructions seen to process tainted data, i.e., the contents of `id_rsa`. Those whose program counter start with `0x8` are part of `ignorame`. Those that start with `0xb` are library code. And those that start with `0xc` are kernel code. Thus, only 12 instructions in `ignorame` actually involve tainted data. This should narrow reverse engineering down quite a bit.

Another output of the script is a dump of the so-called *taint compute number* or *tcn* as a function of instruction count. This is a measure of how computationally distant (depth of the tree of computation) a tainted result is from its taint source. This is output in the file `tcns` in this current directory in two columns: instruction count and tcn. We can plot this in something like R

to see how the tcn changes over time to gain insight.

One final note. The script to process this pandalog is a less than thirty lines thanks to the nice abstraction of protocol buffers. Here it is in its entirety.

```
import sys
import itertools
from google.protobuf.json_format import MessageToJson

from pandare.plog_reader import PLogReader

f = open("tcns", "w")
pcc = {}
with PLogReader(sys.argv[1]) as plr:
    for i,m in enumerate(plr):
        if m.HasField('tainted_instr'):
            if not m.pc in pcc:
                pcc[m.pc] = 0
            pcc[m.pc] += 1
            for tb in m.tainted_instr.taint_query:
                f.write("%d %d\n" % (m.instr, tb.tcn))

f.close()

pcs = list(pcc.keys())
pcs.sort()

for pc in pcs:
    print ("tainted instr @ pc=%x count=%d" % (pc, pcc[pc]))
```

PANDA: Plaid Mode

The real power of PANDA is in writing plugins. This requires programming in C, C++, or (coming soon!) Python. We won't cover that in this class, but you are encourage to read the PANDA manual, and read the code used to implement the various plugins you used in this class to get started. Plugins live in `/panda/panda/plugins/NAME_OF_PLUGIN`.

Feel free to get in touch if you have questions, accolades, or complaints about PANDA.