Project 2: Malware Analysis

CS8803 - O11

Overview

- Goal
  - Analyze real world malware samples and reveal their hidden/true behaviours which normally you don’t see without triggering properly.
- Learning Objectives:
  - Reverse Engineering
    - Manual Binary Reversing through Disassemblers (e.g. Ghidra, Radare2, IDA Pro)
  - Static Analysis
    - Programmatic binary analysis (e.g. data-flow analysis, VSA analysis.)
  - Symbolic Analysis
    - Performing symbolic execution to figure out the whole input or partial of the input
  - Dynamic Trace Analysis (User-level Activity Traces)
    - API Trace (e.g. File, Process, Registry, Network etc. related API trace)
    - BasicBlock Trace (Address of the basic blocks executed while malware is running)
  - Dynamic Binary Instrumentation
    - Manipulating concrete execution using DBI tools (e.g. DynamoRIO)

WARNING

Please **don't run the malware on your own computer!** We are not responsible if you do. Only execute it inside the Windows VM we are provided. These are real world malware samples.

Task Description

- You have 3 real world malware to analyze. Your boss wants you to analyze them and write a report within **two weeks**, and your colleagues are waiting for your results to adjust defense mechanisms to IDS.
- As a malware analyst, to build a strong defense mechanism, your job is to try to reveal all of the behaviours as much as you can. However, most of malware (especially C2-based malware) are only revealed hidden behaviors when the proper inputs (e.g. commands from C2-server) are provided.
- Now, you are going to analyze the three real malware samples with your best weapons (the tools we provide). You are going to perform:
    - Static analysis and reverse engineering on them to find where the triggering logic (e.g. CMD dispatching logic) is located.
    - Symbolic execution to figure out the commands that will trigger hidden behaviors
    - Dynamic binary instrumentation to manipulate execution paths to trigger the hidden behaviors and to collect traces that are corresponding to the behaviors.

Agenda

- Part 1: Analyze malware1.exe(mydoom1.exe) malware which is similar to previously analyzed malware. Trigger the malicious behaviours and observe the effects on the OS. (**60 pts**)
    - Skills :
        - Reverse Engineering,
        - Static Analysis
        - Symbolic Analysis
        - Limited Dynamic Analysis

Agenda

- Part 2: Analyze malware2.exe(win33.exe) malware which is new to your company. Trigger the malicious behaviours and observe the effects on the OS. (**30 pts**)
    - Skills
        - Reverse Engineering,
        - Static Analysis,
        - Symbolic Analysis
        - Extensive Dynamic Trace Analysis
        - Dynamic Binary Instrumentation and Execution Modification

Agenda

- Part 3: Analyze one newly discovered complex malware(unknown.exe) and figuring the CMD Dispatching Logic without having prior investigation.
  - Static Analysis (analyze 10 functions) (**10 pts**)
    - Reverse Engineering
- Bonus: Perform further sophisticated analysis and reveal the malware behaviour:
  - Detecting whole or part of the input (**5 pts**)
    - Reverse Engineering
    - Symbolic Analysis
  - Triggering the malicious behaviour and observe the OS interaction (**5 pts**)
    - Dynamic Binary Instrumentation
    - Dynamic Trace Analysis

Note: Even if you can't figure out any part of the malware,
you can still include any meaningful efforts of revealing the behavior of the malware. We will assign credit accordingly.

High-level Workflow

- Your company has a static analysis tool leveraging Ghidra (developed by the NSA) to detect the important logic such as CMD Dispatching logic which is responsible for determining which malicious behaviour to perform.However, this tool only produces some candidate functions,
- **Your first task** is to perform Reverse Engineering using Ghidra to detect the real CMD Dispatching logic. To do so, you need to analyze each candidate function until you find the actual CMD Dispatching logic.
- **Second**, after discovering the CMD Logic, you are required to construct symbolic execution scripts using angr which will output whole or partial inputs for each behaviour.
- **Third,** you have to properly trigger the malicious behaviours and execute them during concrete execution. The Dynamic Execution Trace will show the malicious activities performed by the malware samples.

Available Artifacts (Free Lunch?)

- Last week, One of your colleagues analyzed a sample malware (sample.exe) last week and created a report.
- That malware looks very similar to the malware malware1.exe which is one of the three malware samples you will analyze for the next two weeks.
- That report is quite informative.
- Fortunately, the report is open to your access and you can take advantage of your colleague’s report and findings.
- Having said that, the report lacks of how to actually execute the behaviour. Your colleague was failed to show the effects on OS such as file system or network operation. Hence, the colleague was failed to prove the nature of the malware sample.

Tutorial (your coworker’s report)

Let’s scrutinize the report for sample.exe.

Before you begin

- Set up a Virtual Machine for Malware analysis
  - Please install/update to the latest version of VirtualBox
    - https://www.virtualbox.org/wiki/Downloads
- Download the VM image
  - Download the project VM from the following link
    - https://drive.google.com/open?id=1a_1U2UQKQ0268NApFnFHfV2HETZZEu4Z
    - We recommend you to download the VM via curl or wget due to the size of the VM. Helpful link: https://stackoverflow.com/questions/25010369/wget-curl-large-file-from-google-drive
    - Note: students have the freedom to adjust VM specs after import

Before you begin

- Open VirtualBox
  - Go to File -> Import Appliance.
  - Select the ova file and import it
- VM user credentials
  - Ubuntu VM password: 123456
  - Windows 7 VM Password: 123456

Project Structure

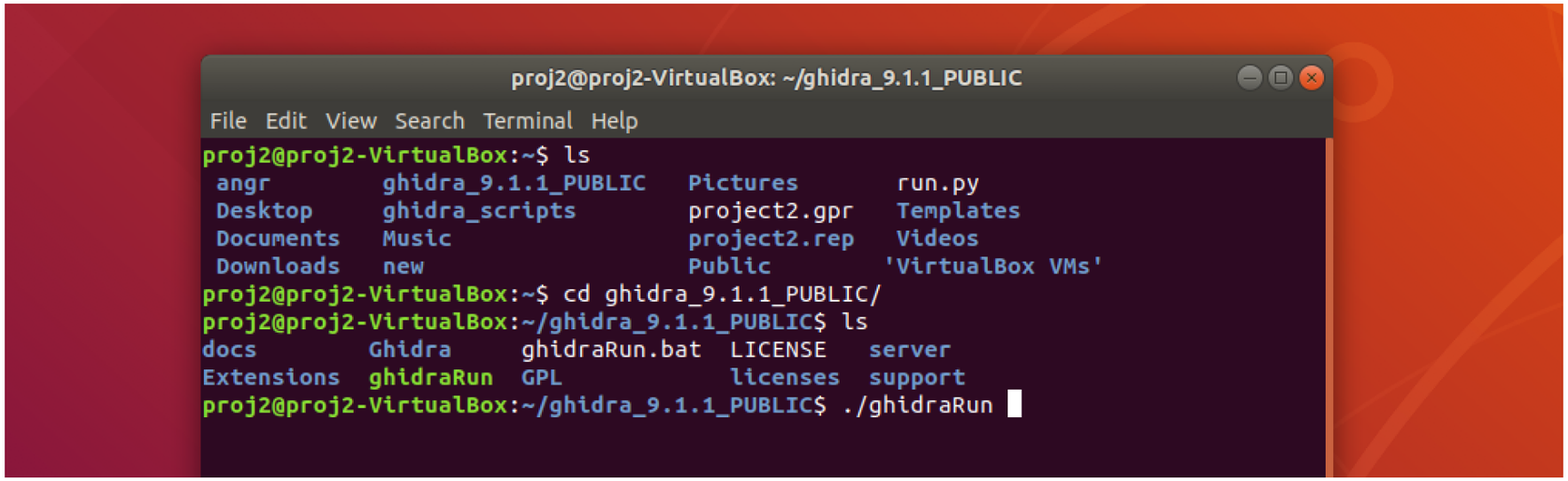
- VM Structure
  - A Windows 7 VM inside Ubuntu 18.04 VM
    - Windows 7 VM for dynamic analysis
      - Target Malware can be executed in Windows 7 VM
    - Ubuntu VM for static analysis
      - A container to block malware from going out

Project Structure

- In the Windows Virtual Machine(VM)
  - C:\code\concrete_executor
    - $python run.py ##will run the tracer program
    - \tracer ## tracer folder(where you need to write some code)
  - C:\code\dynamorio
    - Dynamorio source code and already built client
  - \\VBOXSVR\VM_Share
    - Shared folder with host VM

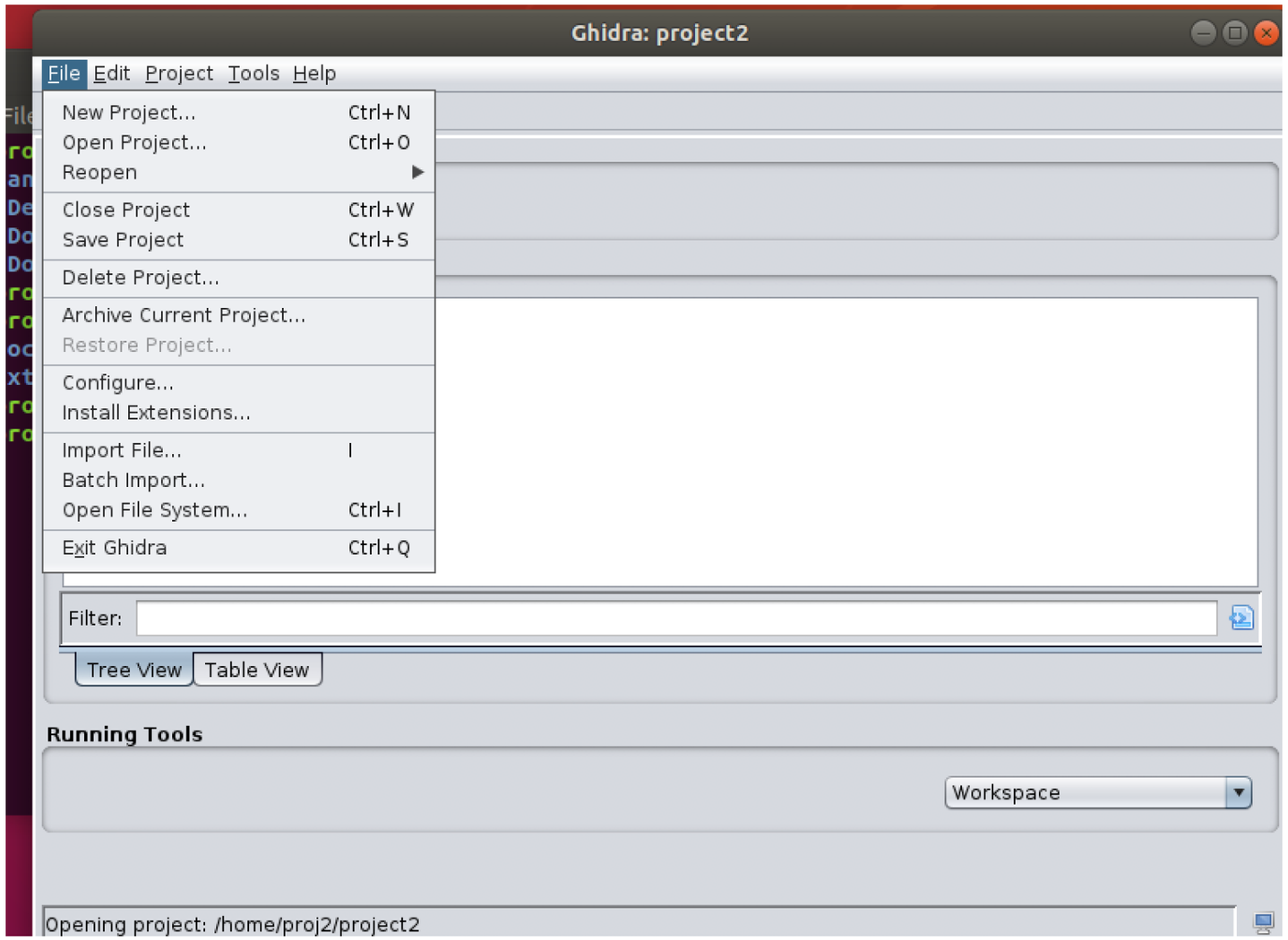
Tutorial - Reverse Engineering

- Go to Ghidra folder and Run Ghidra
    - $ ./ghidraRun



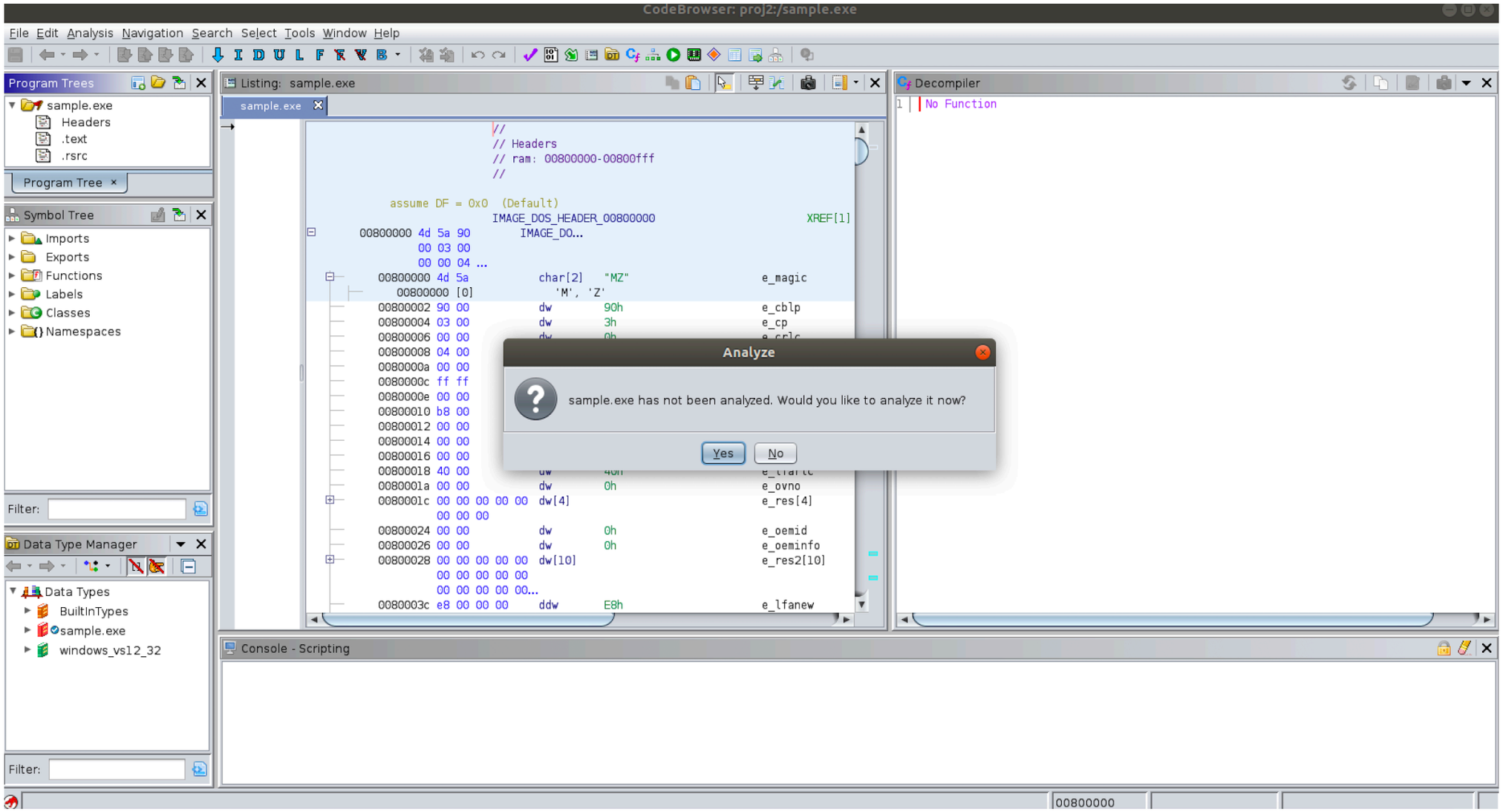
Tutorial - Reverse Engineering

- Use “import file” to import binary into Ghidra project



Tutorial - Reverse Engineering

- Double click on the binary file that appears under the project to open Ghidra GUI and analyze the binary. Then close the GUI and Ghidra completely to save the result into Ghidra database
- There will be an warming about PDBs, which won't affect your analysis



Tutorial - Static Analysis

- In order to have some basic knowledge of the malware, you usually start by analyzing it using a disassembler like Ghidra, Radare2, BinaryNinja or IDA pro, while Ghidra is free and open source (NICE).

- Meanwhile, lucky enough that your coworker sent you a static analysis tool based on Ghidra, which detects the possible dispatching logics. Then all you need to do is to iterate over the candidates and determine the real dispatching logic.

Tutorial - Static Analysis

- To use the provided Ghidra Script
- Close Ghidra project GUI
- In /ghidra_9.1.1_public folder, go to /support, you will find a file called: analyze.sh
- This wrapper script runs the ListFunctionAsCodeBlock.java script with a headless analyzer and pipes the output into output.txt
- Usage: $ ./analyze.sh {malware.exe}
    - (just the file name is enough, no need to append PATH)

```
proj2@proj2-VirtualBox:~/ghidra_9.1.1_PUBLIC$ cd support/
proj2@proj2-VirtualBox:~/ghidra_9.1.1_PUBLIC/support$ cat analyze.sh
#!/bin/sh
./analyzeHeadless /home/proj2 project2 -process $1 -postscript /home/proj2/ghidr
a_9.1.1_PUBLIC/Ghidra/Features/FunctionID/ghidra_scripts/ListFunctionAsCodeBlock
.java > output.txt
proj2@proj2-VirtualBox:~/ghidra_9.1.1_PUBLIC/support$
```

Tutorial - Static Analysis

- To be able to run the script, you **MUST** first analyze the malware in Ghidra and save the result into Ghidra database
- Note: Ghidra must be closed completely, in order to run the headless script

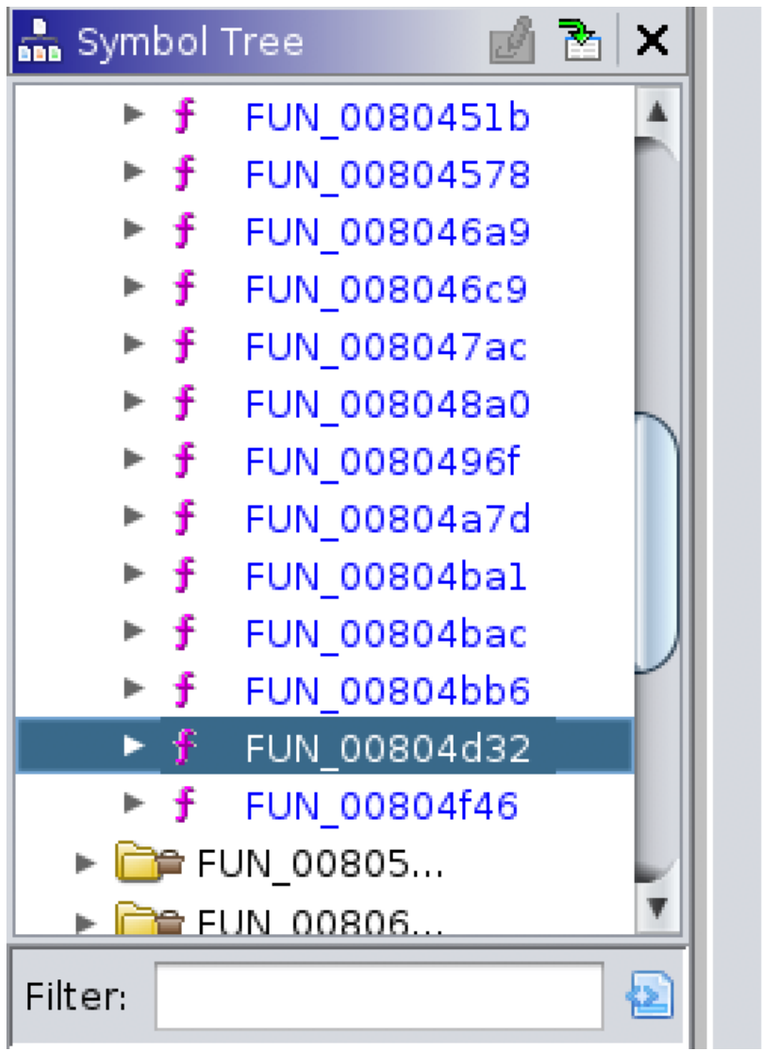
Tutorial - Static Analysis

- The script will output top 50 longest chain of basic blocks that uses certain variable in the end.
- To view these chains, run $tail -53 output.txt
- This will give you a few candidates for where to find the target function that contains the dispatch point that triggers certain behavior (There could be some false positive and may not work for all malware, but for malware1.exe and malware2.exe, target function is within top results)

```
proj2@proj2-VirtualBox:~/ghidra_9.1.1_PUBLIC/support$ tail -53 output.txt
INFO  0: FUN_00804d32, local_120, 00804d6a 00804da2 00804db6 00804dca 00804de5 00804e09 00804e1b 00804e54 00804eae 00804ecb 00804edd 00804efa 00804f1a  (GhidraScript)
INFO  1: FUN_00802fa0, local_28, 00802ff3 00803002 00803005 00803027 00803042 0080304a 00803060 0080306a 008030d4  (GhidraScript)
INFO  2: FUN_0080299e, local_12c, 00802a08 00802a2d 00802a39 00802a53 00802a61 00802aa3 00802ad0 00802af1  (GhidraScript)
INFO  3: FUN_00803265, local_204, 00803265 0080328b 008032b0 008032bf 008032cd 0080330f 00803332 00803338  (GhidraScript)
INFO  4: FUN_00806cf9, local_8, 00806d21 00806d2f 00806d93 00806de9 00806df4 00806dfd 00806e0c 00806e1c  (GhidraScript)
INFO  5: FUN_00804d32, local_8, 00804e1b 00804e47 00804e54 00804e69 00804e6d 00804e79 00804f35  (GhidraScript)
INFO  6: FUN_00804f46, local_1b8, 00805000 00805031 00805039 00805053 00805067 00805082  (GhidraScript)
INFO  7: FUN_008066c8, local_404, 008066d7 00806713 0080671b 0080673a 00806741 00806746  (GhidraScript)
INFO  8: FUN_008075b2, local_198, 008075fb 0080762e 00807641 008076bc 0080776d 008077a1  (GhidraScript)
INFO  9: FUN_00806765, local_8, 008067ee 00806966 00806977 0080698f 008069ba 008069d6  (GhidraScript)
INFO  10: FUN_008050b1, local_108, 008050b1 00805110 00805118 0080512a 00805145  (GhidraScript)
INFO  11: FUN_00805aa7, local_310, 00805be3 00805c43 00805c66 00805c79 00805ccd  (GhidraScript)
INFO  12: FUN_00806b4f, local_114, 00806c1a 00806c53 00806c68 00806c71 00806cbc  (GhidraScript)
INFO  13: FUN_008075b2, local_2f0, 00807693 008076a6 00807735 0080774d 008077a1  (GhidraScript)
INFO  14: FUN_00802db3, local_12, 00802e1b 00802e22 00802e28 00802e2f 00802e35  (GhidraScript)
INFO  15: FUN_0080783e, local_30, 00807852 008078eb 008078f7 0080791f 0080793b  (GhidraScript)
INFO  16: FUN_00804578, local_8, 008045a5 008045db 008045e4 0080467b 00804682  (GhidraScript)
INFO  17: FUN_00805316, local_8, 0080537b 0080546c 00805471 0080548b 008054ac  (GhidraScript)
INFO  18: FUN_00807400, local_8, 00807400 00807428 00807451 00807468 0080746e  (GhidraScript)
INFO  19: FUN_0080418a, local_104, 008041a3 008041c4 008041e0 0080420f  (GhidraScript)
INFO  20: FUN_00804d32, local_260, 00804e1b 00804e47 00804edd 00804f1a  (GhidraScript)
INFO  21: FUN_008062ca, local_7c4, 008062f2 00806324 00806343 008063bb  (GhidraScript)
INFO  22: FUN_008064ec, local_104, 0080659b 008065b1 008065be 008065d0  (GhidraScript)
INFO  23: FUN_00806765, local_10c, 00806765 00806790 008067b8 008068b1  (GhidraScript)
INFO  24: FUN_008071e7, local_17c, 008071e7 00807221 00807227 00807230  (GhidraScript)
INFO  25: FUN_0080317f, local_1e, 008031f9 008031ff 00803206 0080324e  (GhidraScript)
INFO  26: FUN_00805247, local_14, 00805247 0080527a 00805282 00805288  (GhidraScript)
INFO  27: FUN_00805316, local_10, 00805354 0080537b 00805471 008054ac  (GhidraScript)
INFO  28: FUN_00805fd0, local_10, 00806044 0080605b 008060ca 008060e4  (GhidraScript)
```

Tutorial - Static Analysis

- Reopen the Ghidra GUI
- Inside the Symbol Tree, you can find a folder called Functions.
- The first function in your coworker's Ghidra script's output is FUN_00804d32, so you decide to double click it



Tutorial - Static Analysis

- The Decompile window on the right gives you the decompiled version of the function that you just double clicked. And you could check the logic of this function in a more readable form

Decompile: FUN_00804d32 - (sample.exe)

```
1
2  /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4  undefined4 __cdecl FUN_00804d32(LPCSTR param_1,int param_2)
5
6  {
7    char *pcVar1;
8    int iVar2;
9    BOOL BVar3;
10   byte local_260 [44];
11   char local_234;
12   char local_233;
13   char local_232;
14   char acStack289 [281];
15   HANDLE local_8;
16
17   if (((param_2 < 1) || (param_1 == (LPCSTR)0x0)) || (*param_1 == '\0')) {
18     return 1;
19   }
20   while (_DAT_00808248 != 0) {
21     Sleep(0x800);
22   }
23   lstrcpyA(acStack289 + 1,param_1);
24   CharLowerA(acStack289 + 1);
25   pcVar1 = strstr(acStack289 + 1,&DAT_00801a7c);
26   if (((pcVar1 != (char *)0x0) ||
27       (pcVar1 = strstr(acStack289 + 1,s_malicoius_00801a70), pcVar1 != (char *)0x0)) ||
28      ((pcVar1 = strstr(acStack289 + 1,s_upload._00801a68), pcVar1 != (char *)0x0 ||
29       (pcVar1 = strstr(acStack289 + 1,s_playyingg_00801a5c), pcVar1 != (char *)0x0)))) {
30     FUN_00803352(param_1);
31   }
32   lstrcpyA(acStack289 + 1,param_1);
33   iVar2 = lstrlenA(acStack289 + 1);
34   if (acStack289[iVar2] != '\\') {
35     lstrcatA(acStack289 + 1,&DAT_00801360);
```

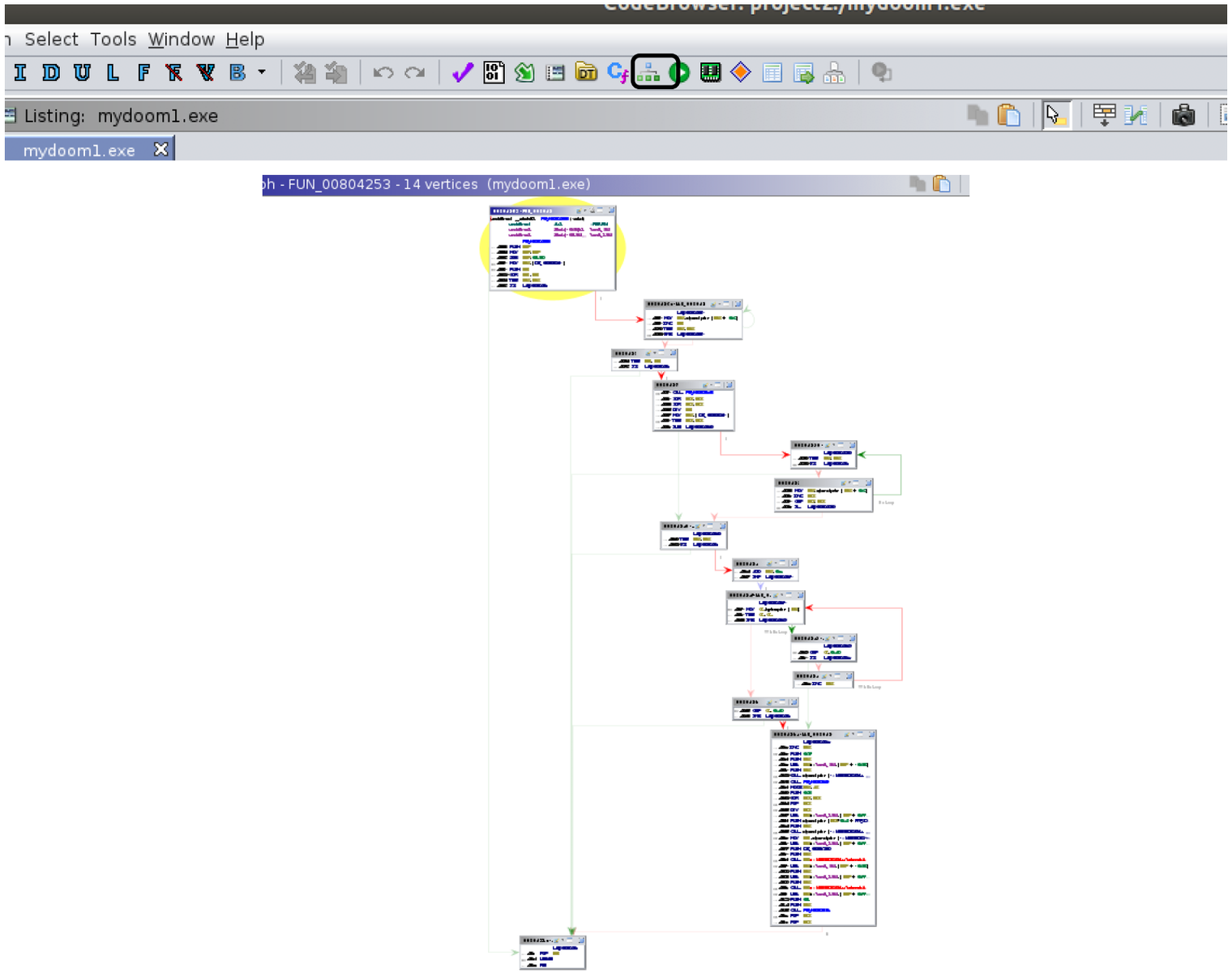
Tutorial - Static Analysis

- You notice that there is the dispatching logic between line 25 to line 31 (lucky!)

```
21      Sleep(0x800);
22    }
23    lstrcpyA(acStack289 + 1,param_1);
24    CharLowerA(acStack289 + 1);
25    pcVar1 = strstr(acStack289 + 1,&DAT_00801a7c);
26    if (((pcVar1 != (char *)0x0) ||
27        (pcVar1 = strstr(acStack289 + 1,s_malicoius_00801a70), pcVar1 != (char *)0x0)) ||
28       ((pcVar1 = strstr(acStack289 + 1,s_upload._00801a68), pcVar1 != (char *)0x0 ||
29        (pcVar1 = strstr(acStack289 + 1,s_playyingg_00801a5c), pcVar1 != (char *)0x0)))) {
30      FUN_00803352(param_1);
31    }
```

Tutorial - Static Analysis

- You want to see a clearer form of the control flow graph, so you click on the Display function graph button and the CFG pops up.

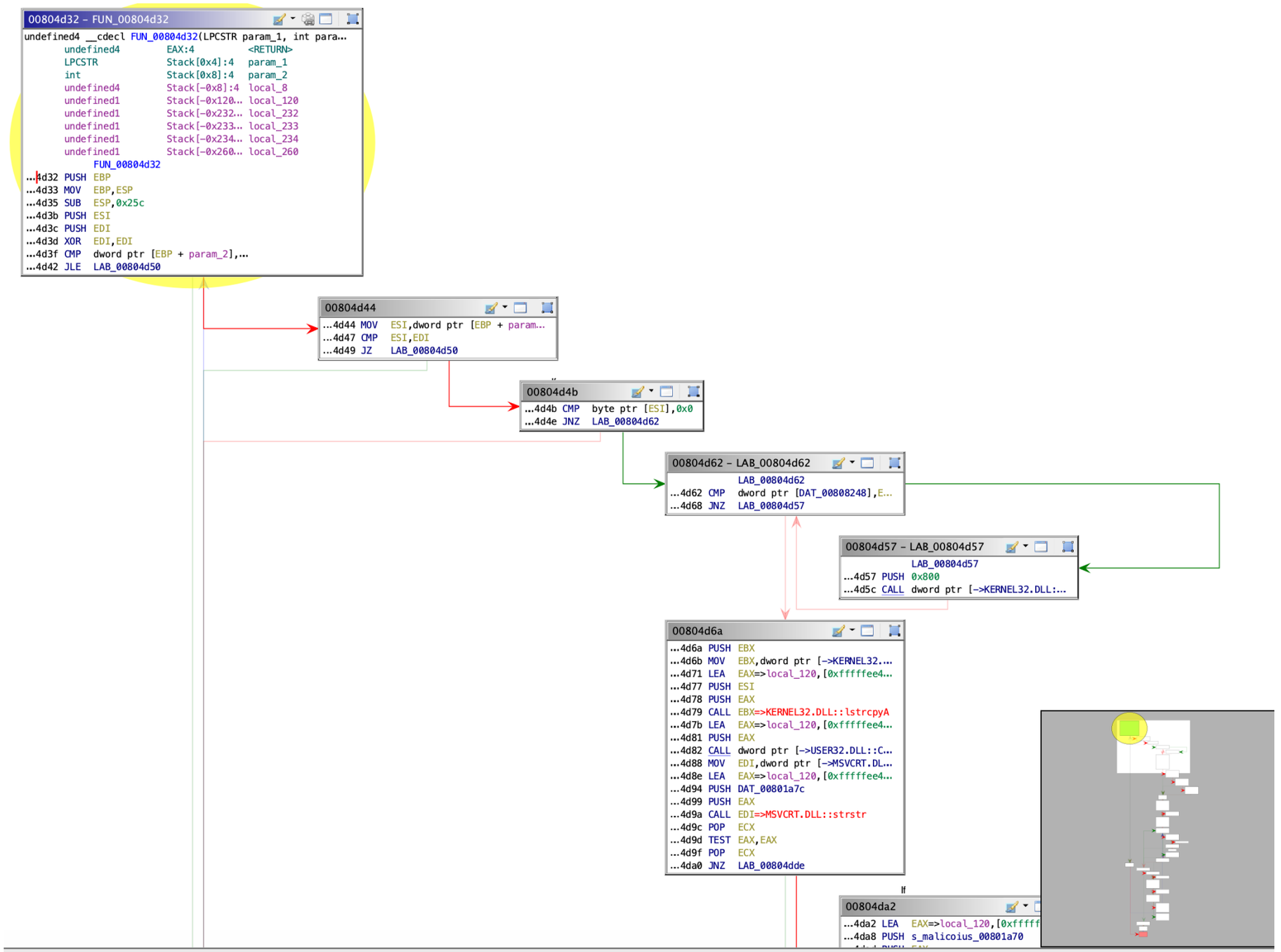


Tutorial - Symbolic Analysis

- After knowing some possible information about the dispatching logic, you want to construct symbolic execution script using angr to output whole or partial inputs for each behaviour.
- Note: **Your script should run with python3**
- In essence, You want to find some keywords that will trigger the malware’s activities

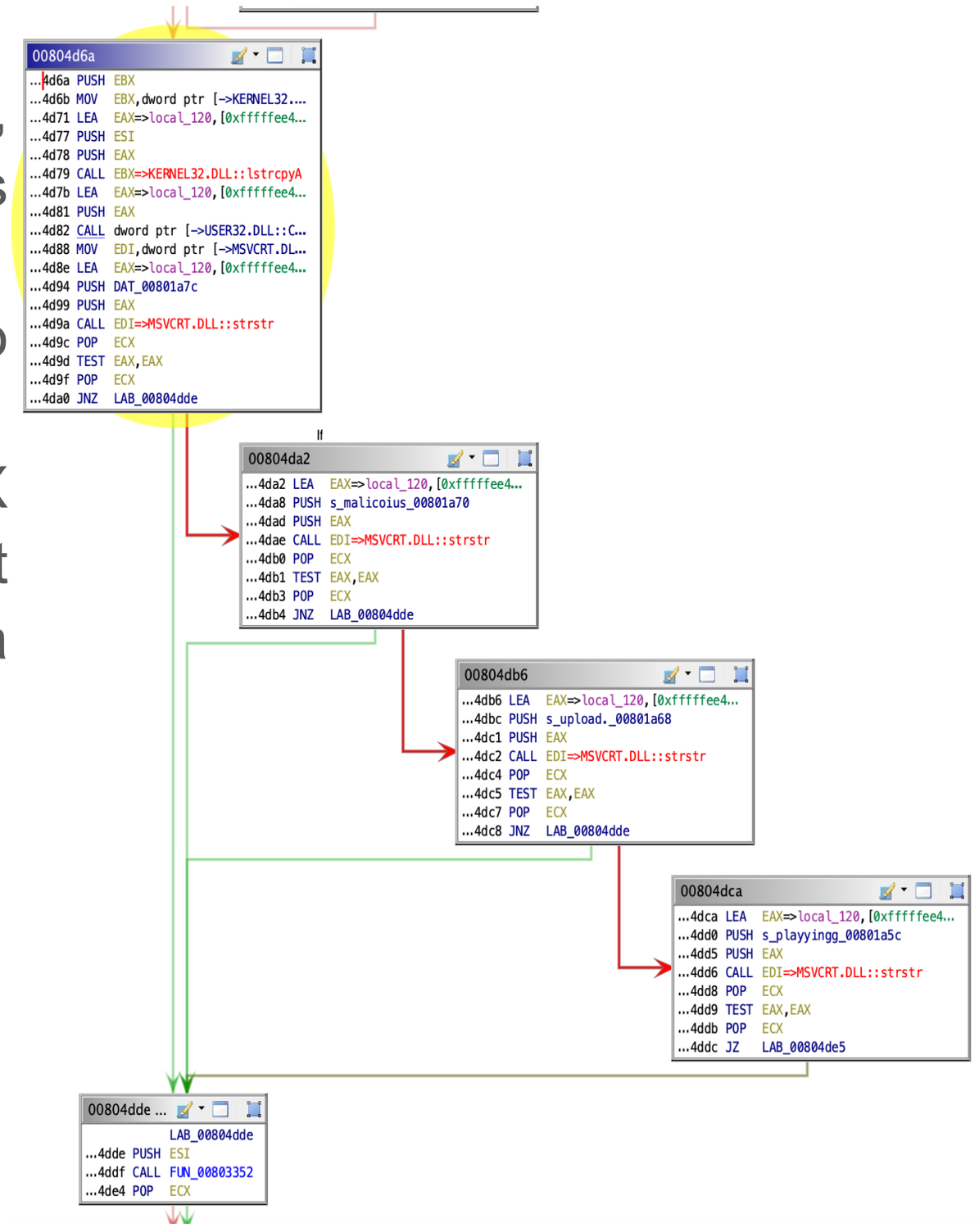
Tutorial - Symbolic Analysis

- Let’s take another look at our CFG for the Dispatching Logic



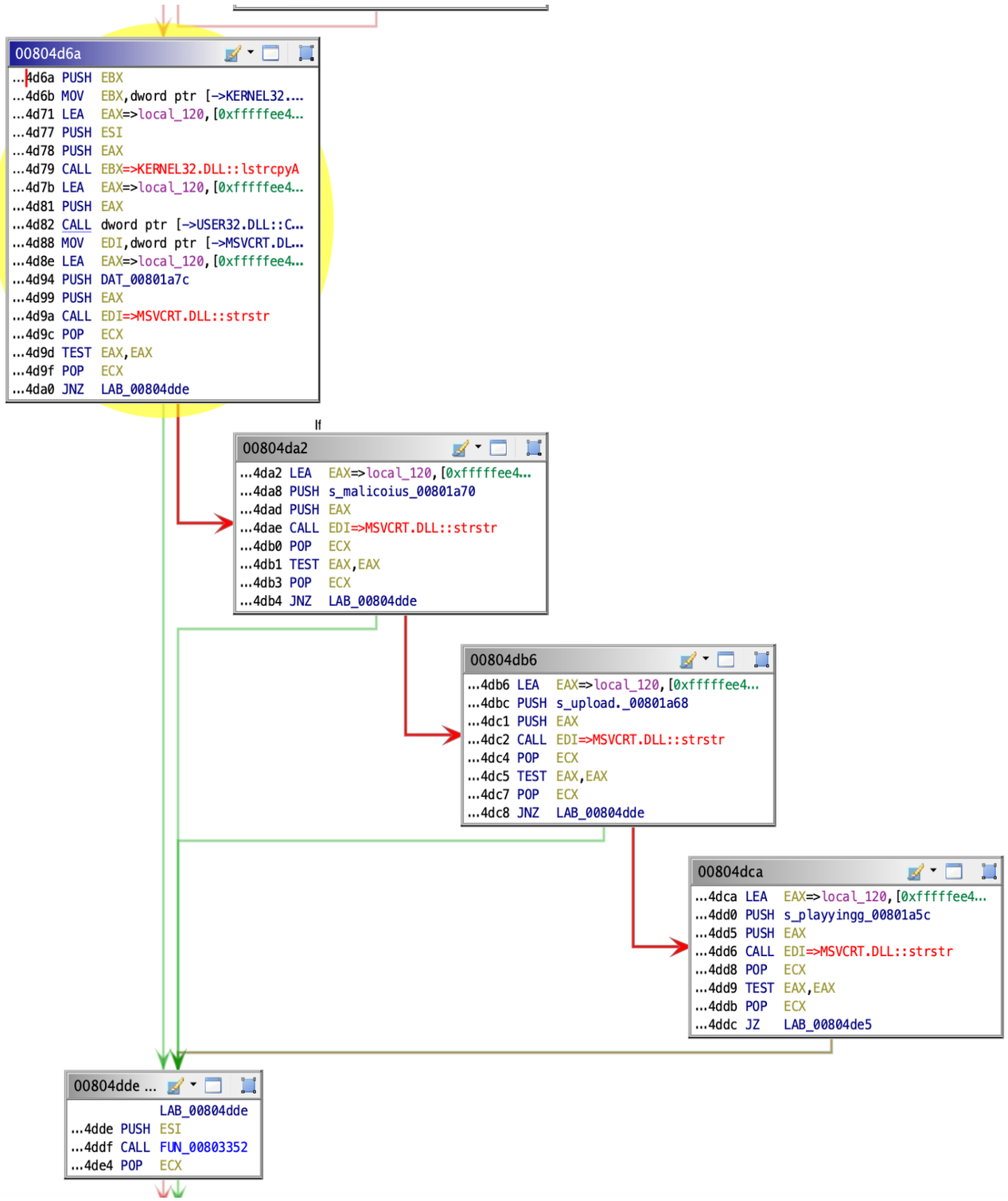
Tutorial - Symbolic Analysis

- This chain of basic blocks(0x804d6a, 0x804da2, 0x804db6, 0x804dca) is responsible for the dispatching logic.
- The chain uses a function to determine the triggering logic -- strstr.
- All the strstr functions takes the EAX register as a pointer to the first parameter and some constant as a second parameter.



Tutorial - Symbolic Analysis

- Since all the second parameters are constants, we should focus on the first argument which is pointed by EAX.
- If you go backward from the chain of basic block, you can see EAX value is determined in basic block 0x804d6a.
- The call instruction to lstrcpyA at 0x804d79 will copy the buffer pointed by ESI to the address held by EAX -- EAX being the first argument and ESI being the second argument for lstrcpyA.



Tutorial - Symbolic Analysis

- Now we know where EAX is assigned.
- However, the source of the copy operation is also another register.
- We need to perform further static backward slicing to detect the original source of the command.
- While going backward in the CFG, you will see the last update to the ESI register is performed at 0x804d44 with **mov ESI, [EBP + 8]** .
- In x86 calling conventions (e.g. cdecl, stdcall), EBP+8 points the the first argument passed to the function.
- Hence, we now know the symbolic analysis needs to solve the value of the first parameter (EBP + 8) passed to the function 00x804d32.
- Now, we know where the variable that is passed to the malware resides, and can find keywords that reach the target function in the malware!

Tutorial - Symbolic Analysis

- Fortunately, your colleague also had a symbolic execution script for sample.exe (NOTE: sample.exe is similar to malware1.exe!).
- The report also shows the start and end addresses for the symbolic execution -- start:0x804d32 end:0x804dde
- From your terminal go to **~/Desktop/symbolic_execution/** folder and run
  - **python sample_inputs.py --start 0x804d32 --end 0x804dde**
- It will print out one of the concretized inputs that malware requires to expand its malicious behaviour.

Tutorial - Symbolic Execution

- However, there is no free lunch.
- Using the same start and end addresses, you need to find all the commands.
  - **Hint:** You need to add constraints to the state which inverse the path conditions to explore other possibilities.The angr documentation may be of use.
  - Let's analyze the code snippet below. When performing symbolic execution, symbolic execution engine can find the path condition for **ddos_attack** as message[0] = 101
  - Once we already explored **ddos_attack**, you need to explore other branches now.
  - To explore other branches, you need to add constraint such as message[0] != 101, hence the path conditions for **ddos_attack** becomes infeasible.
  - Once, you are done with send_spam, keep adding the condition inverses to explore other behaviours.
  - Copy sample_inputs.py as malware1_inputs.py, and modify the code.
  - Add a constraint for each found input such as
    - **parameter[0] != found_input[0]**
  - **Extra hint:** you should use
  - an inverse_constraints array to output all strings at one
  - run

```
int cmd = m->message[0];
if (cmd == 101)
    ddos_attack(m);
else if (cmd == 142)
```

Tutorial - Symbolic Execution

Extra Help for Angr:

Angr’s Solver Engine: Claripy && How to add constraints

https://docs.angr.io/advanced-topics/claripy

https://docs.angr.io/core-concepts/solver

Tutorial - Dynamic Analysis

- Now, it is time to practice Dynamic Binary Instrumentation and Dynamic Trace Analysis.
- For this purpose, we already have a DynamoRIO client to capture the API traces and Basic Block addresses captured during concrete execution.
- Your colleague’s report also shows how to use and manipulate the values of the function arguments.
- The report shows the execution successfully reaches to the 0x804dde.
- However, your colleague again failed to capture and prove what malware is doing.

Tutorial - Dynamic Analysis

- We need a secure experiment environment to execute the malware.
- Why?
  - Insecure analysis environment could damage your system
- You may not want:
  - Encrypting your file during a ransomware analysis
  - Infecting machines in your corporate network during a worm analysis
  - Creating a tons of infected bot client in your network during a bot/trojan analysis
- The solution:
  - Contain malware in a virtual environment
    - Virtual Machine
    - Virtual Network
      - Conservative rules(allow network traffic only if it is secure)

Tutorial - Dynamic Analysis

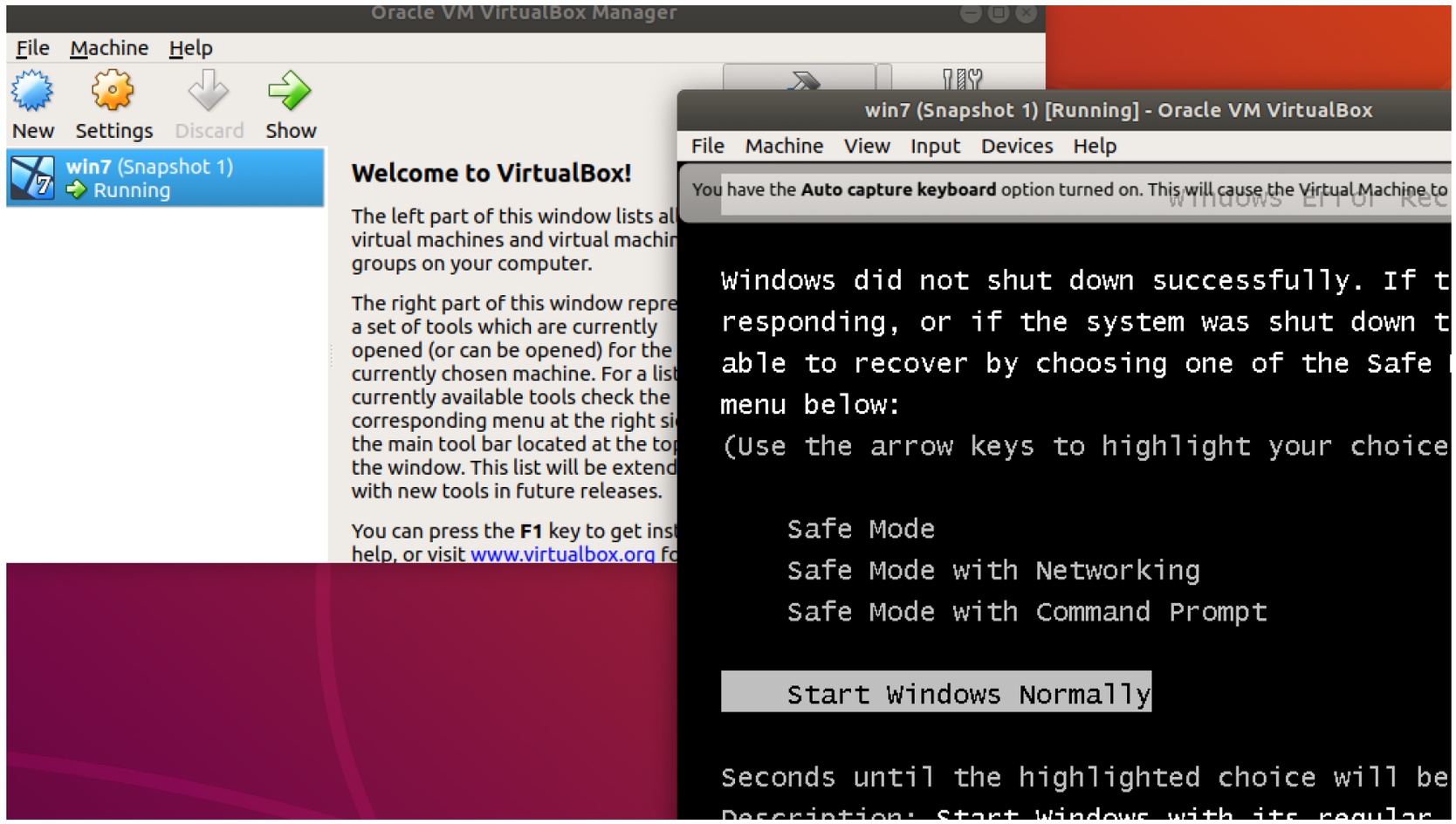
- **WARNING: We must reiterate that you make sure that no malware traffic goes out from the virtual machine!!!**
- **Run $sudo iptables -A OUTPUT -o enp0s3 -j DROP:**
  - DROP all -- anywhere anywhere
- You can check if the rule exists by typing
  - $sudo iptables -L
- Keep the firewall rule to prevent malware traffic going out.

Tutorial - Dynamic Analysis

- Although static analysis could give a lot of information about the malware, it’s important for malware analyst to see how the malware behave when it’s executed
- Your next task is to use Dynamorio to perform dynamic analysis.
- With provided code in C:\code\concrete_executor
- You can feed the malware different input to see what the malware will do

Tutorial - Dynamic Analysis

- Run Windows 7 VM in the VirtualBox in ubuntu VM
- Password: 123456

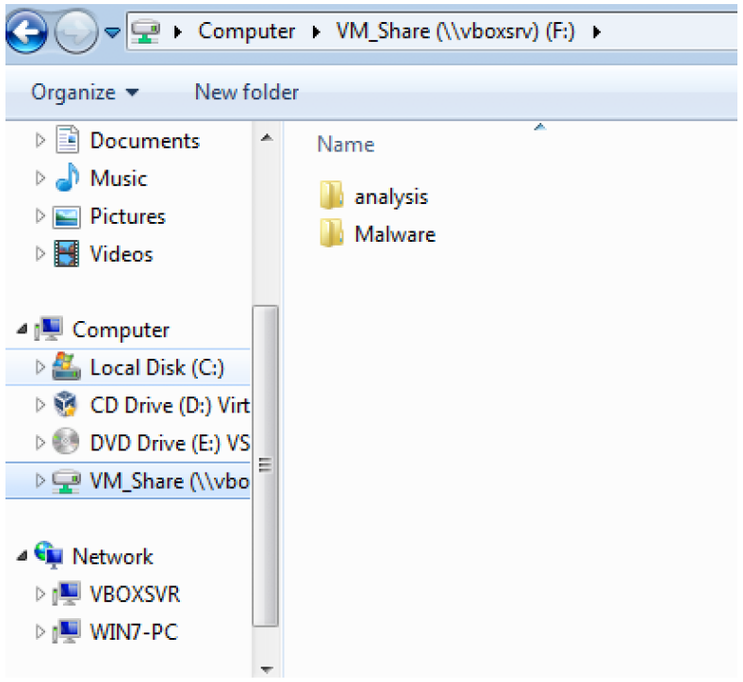


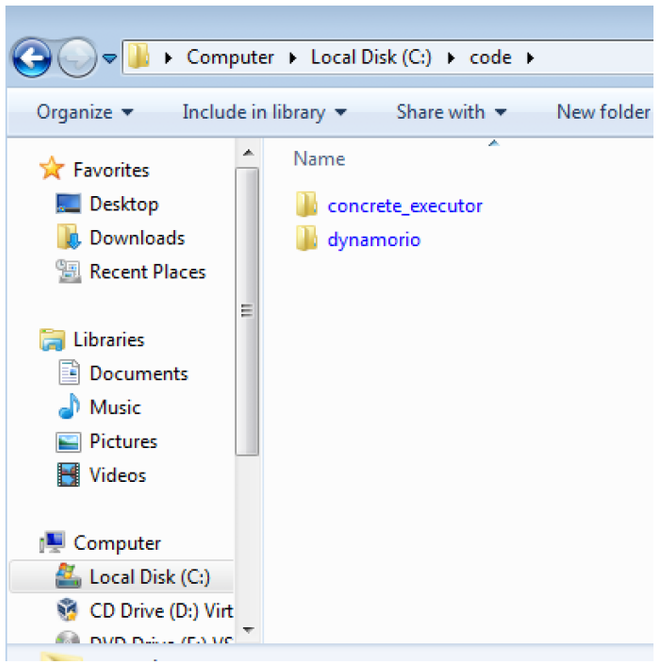
Tutorial - Dynamic Analysis

- You are given a snapshot (Base) to act as a backup for your analysis
- If something bad happens on your Windows7 testbed, always revert back to the basecamp snapshot.

Tutorial - Dynamic Analysis

- Basic Structure:
  - C:\code\concrete_executor
    - Directory contains Dynamorio scripts
    - run.py: python script to run the Dynamorio server
  - \\VBOXSVR\VM_Share
    - Shared folder with host ubuntu VM
    - You can find VM_Share on ubuntu desktop





Tutorial - Dynamic Analysis

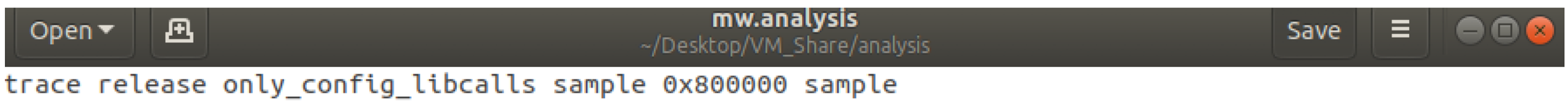
- In VM_Share
  - Malware/
    - where you need to put the malware sample
  - analysis/
    - mw.analysis : dynamorio task assignment file
    - sample/ : sample folder
      - sample/: contains a copy of sample.exe and trace files that dynamorio returned by running the sample.exe in windows VM

Tutorial - Dynamic Analysis

- To assign a task to the Dynamorio script server, first write the following into mw.analysis file:

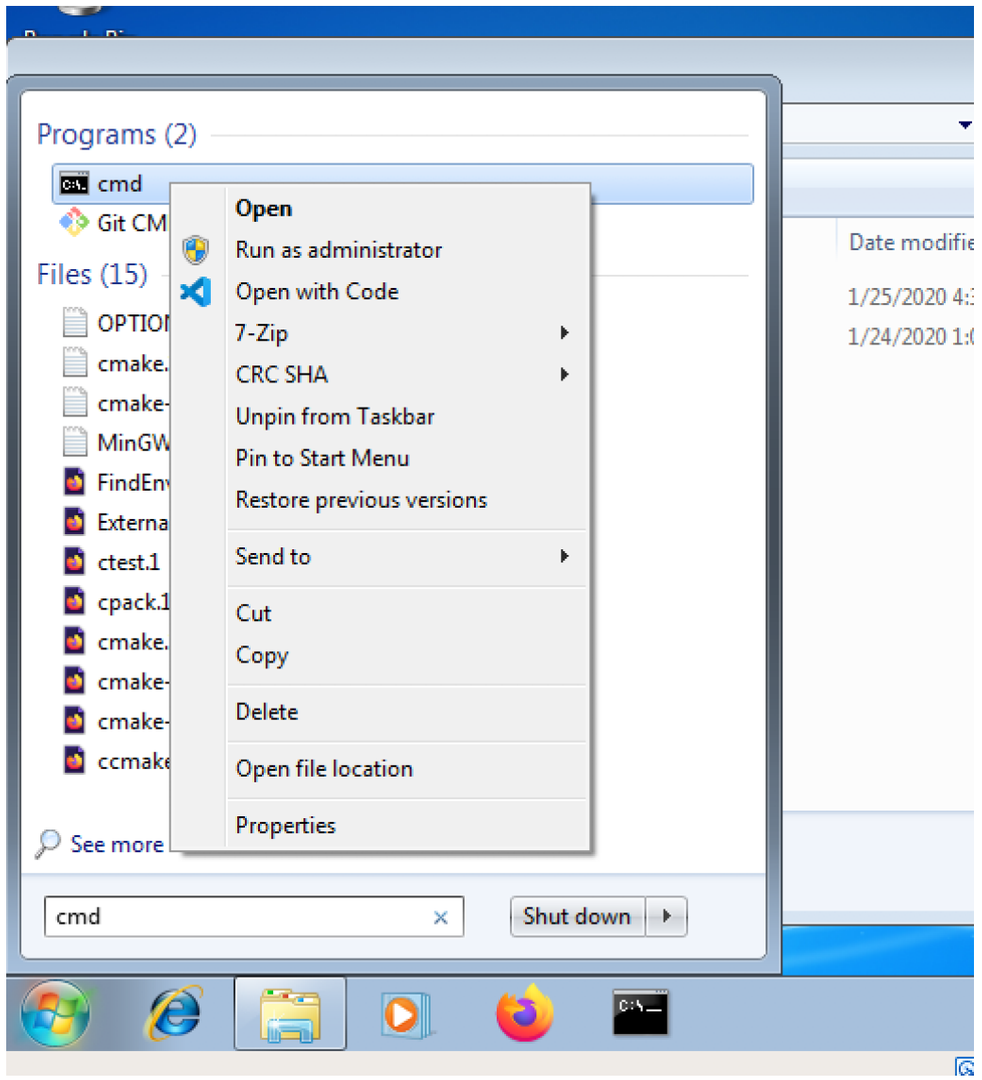
  trace release only_config_libcalls {binary name} {starting address} {folder name}

  e.g. trace release only_config_libcalls sample 0x800000 sample
  (as the program starts from 0x800000)



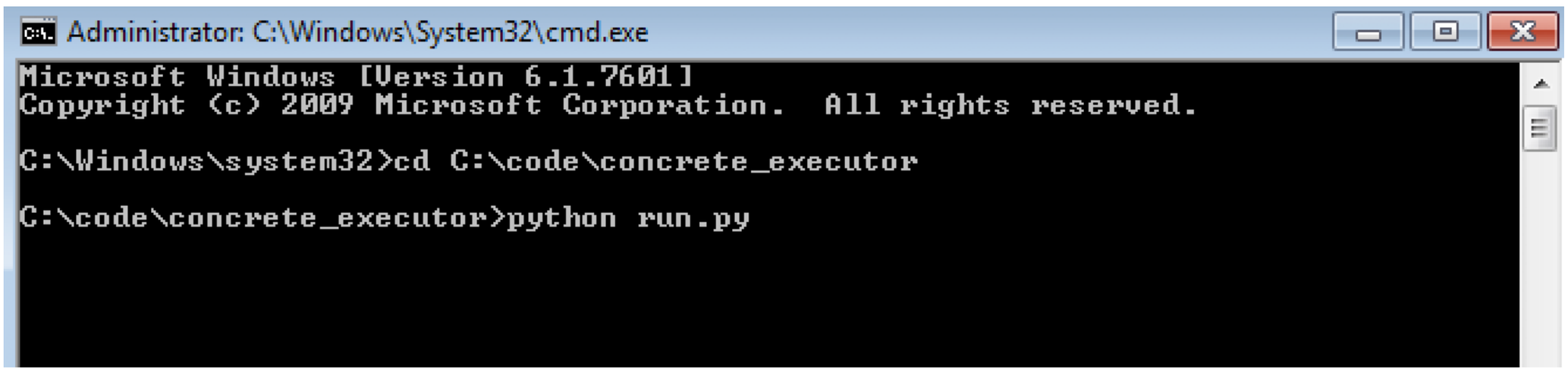
Tutorial - Dynamic Analysis

- To run the Dynamorio server: open WIN7 VM
- open command prompt with admin privilege
- Search “cmd” and right click on the cmd
- Select “run as administrator”



Tutorial - Dynamic Analysis

- Go to C:\code\concrete_executor
- Run > python run.py
- No need to worry for some copy failed error messages in the output, It's for other purposes. As long as you can see trace file copied over to the shared folder, the tracer program should be running correctly

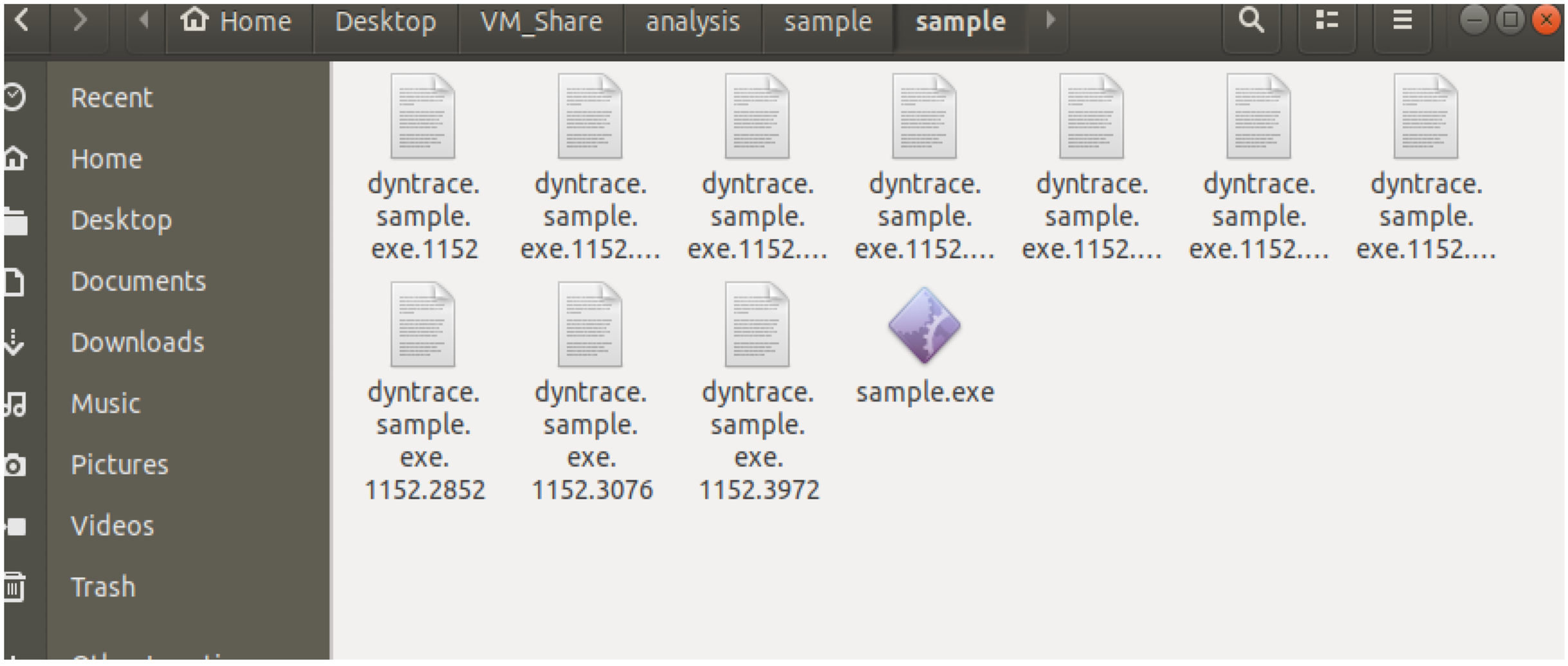


Tutorial - Dynamic Analysis

- After the script finished running, it wrote “DONE” back in the mw.analysis file.
- If you want to re-analyze the same task, Delete “DONE”, go back to WIN7 VM and execute python run.py in C:\code\concrete_executor

Tutorial - Dynamic Analysis

- The trace file will be copied back to the shared folder where the malware sample sits



Tutorial - Dynamic Analysis

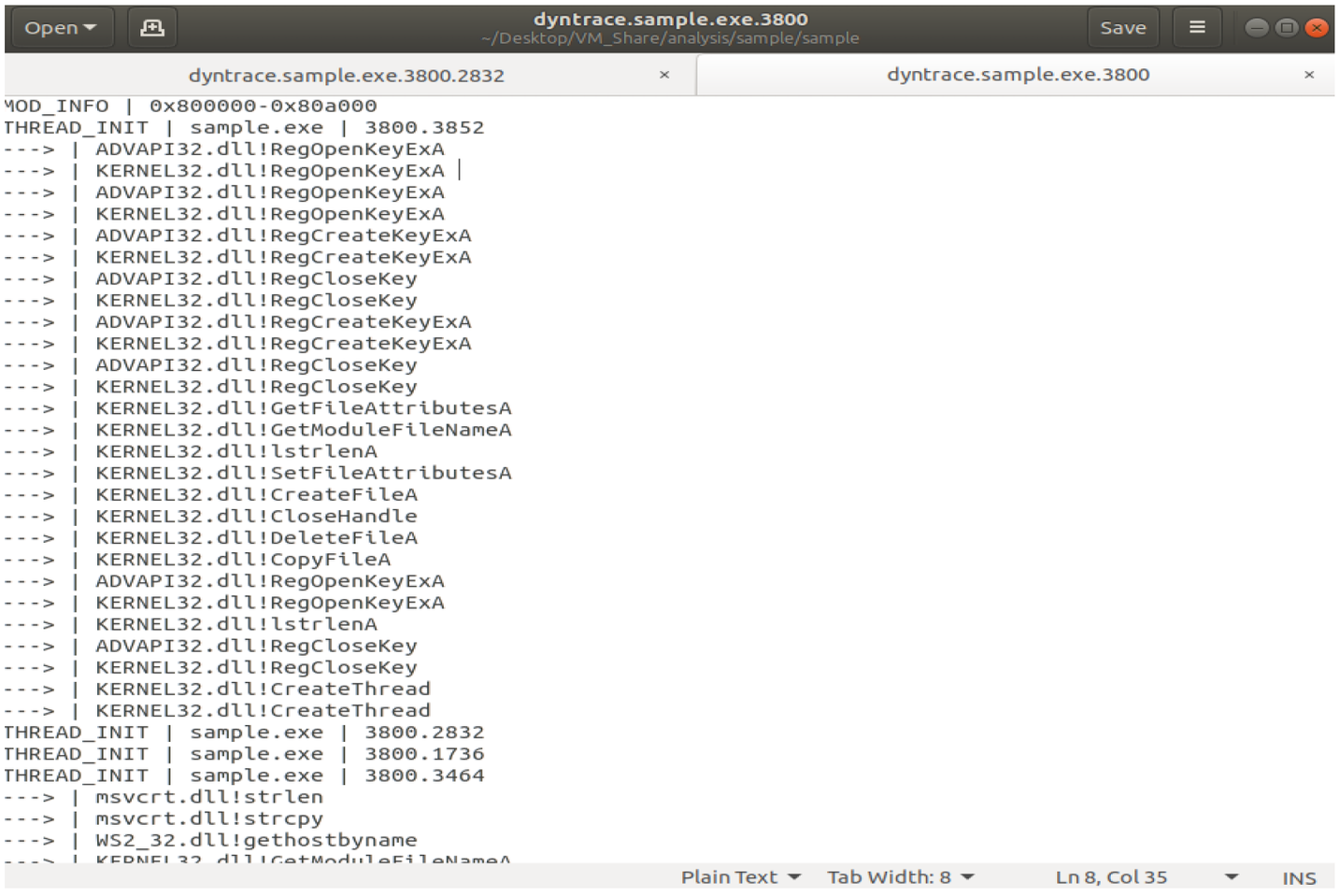
- Now that you obtained the trace file, you need to analyze the malware traces to find out how to trigger the behaviors

- There will be some break-points set by Dynamorio around certain behavior. Check out the code block that contains the actual BRK address in Ghidra and find a way to lure the malware into your desired path.

Tutorial - Dynamic Analysis

- Analyze the trace files to see the default behavior of malware sample.

- while checking the first trace file, we found that after sample.exe thread inits, it invoked few kernel operations to manage the key and file and then created three more threads



Tutorial - Dynamic Analysis

- As you can see, after threads have been created, the malware called GetModuleFileNameA and started tons of ReadFile operations

- It should be an indication that this malware is trying to find/read from certain file/directory/socket in the system!

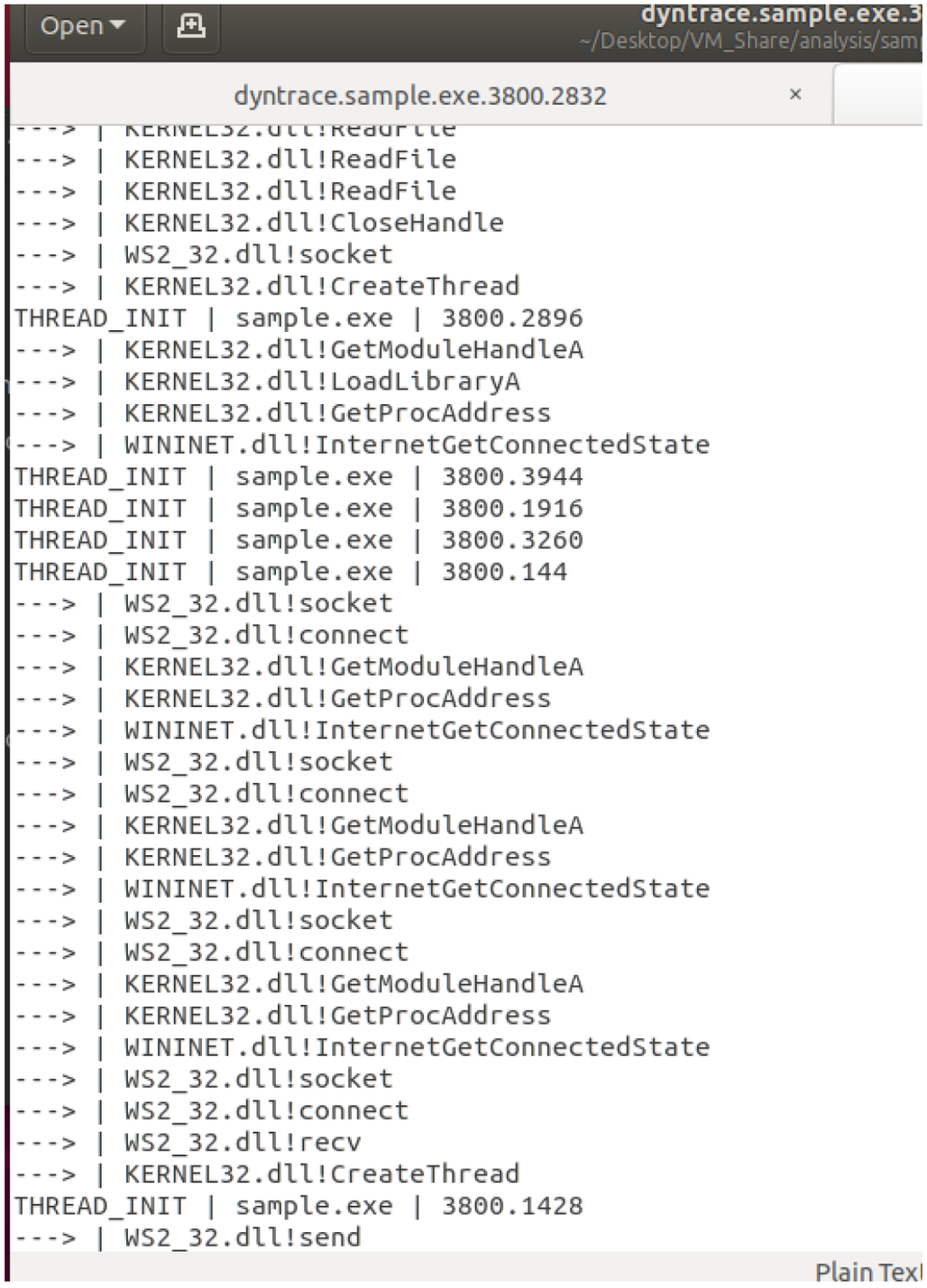
- Details for win32 api calls please see: https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile

```
---> | msvcrt.dll!strlen
---> | msvcrt.dll!strcpy
---> | WS2_32.dll!gethostbyname
---> | KERNEL32.dll!GetModuleFileNameA
---> | KERNEL32.dll!CreateFileA
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
---> | KERNEL32.dll!ReadFile
```

Tutorial - Dynamic Analysis

- At the end of first trace file, the malware creates 4 socket connections. After recv returns, one more thread is created and it then issued a send.

- Em, Interesting internet connection found here. Maybe it's worth digging into in the other trace files for break points to see what they are really doing.



Tutorial - Dynamic Analysis

- The rest of the trace files contain a more detailed version of traces that have breakpoint address set both before and after the kernel operations you found in the first trace file. Now it’s the time, you should go back to static analysis to check the break point address in the disassembler to see what the malware is doing at that time.

Open ▾ dyntrace.sample.exe.3800.1428 ~/Desktop/VM_Share/analysis/sample/sample Save

```
THREAD_INIT | sample.exe
BRK | 0x80783e
BRK | 0x807852
BRK | 0x807866
BRK | 0x80786f
BRK | 0x807877
BRK | 0x8077d8
BRK | 0x80789b
BRK | 0x8078a8
---> | WS2_32.dll!send | 0:0x1ac ;1:8® ;2:2 ;3:0 ;
```

Tutorial - Dynamic Analysis

- You can also use Ghidra to find and analyze all the breakpoint address to see the actual trace where malware execute through

- If necessary, find the code block and the corresponding checking instruction that is preventing the malware from going further to the code blocks containing c2-command (you should already know the **target function** that contains the dispatching logic from static analysis and the **triggering command** that you need to feed into the malware to activate the behavior)

Tutorial - Dynamic Analysis

- How can you wrap the data to define what its value is?

- Add following two functions in C: \code\concrete_executor\tracer\libcall_handler.cpp with their declaration in .h file:

```
static void
wrap_pre_target(void *wrapcxt, OUT void **user_data) {
   char *buf = (char *) drwrap_get_arg(wrapcxt, 0);
   strcpy(buf, "[Symbolic Execution Generated String]" );
}

static void
monitor_target_function(void *drcontext) {
   app_pc tgt_function = (app_pc) 0xADDRESS;
   drwrap_wrap_ex(tgt_function, wrap_pre_target, NULL, NULL, 0);
}
```

- Read more about how to wrap functions in Dynamorio using this link

Tutorial - Dynamic Analysis

- Check the given skeleton code for more insight on how to feed malware the correct input
- In order to compile your edited code, execute `build.bat` in the Windows VM
- Once you build the code, reset the Dynamorio task by deleting the "DONE" in mw.analysis and run >python run.py in C:\code\concrete_executor
- Dynamorio will give back new traces.
- Analyze the traces with Ghidra to see if any more barriers exist that prevent malware going to the target function
- Record any meaningful behaviors in the Excel sheet along the way. Explore all behavior for malware1.exe and malware2.exe

Tutorial - Dynamic Analysis

- For malware2:
  - In libcall_handler.cpp
    - Modify wrap_pre_target(void *wrapcxt, OUT void **user_data)
    - monitor_target_function(void *drcontext)
    - Inside wrap_pre_lib(),
      (hook the internet api call to emulate attackers command)
  - In tracer.cpp
    - You are encouraged to add code in instr_should_instrument() and event_app_instruction() to manipulate malware instructions to pass or bypass certain condition check.
    - Helper link: http://dynamorio.org/docs/API_BT.html

Summary

- You are given three malware samples:
  - malware1.exe (mydoom1.exe)
  - malware2.exe (win33.exe)
  - malware3.exe (unknown.exe)
- We suggest you analyze in the order as following:
  - malware1.exe, malware2.exe, malware3.exe
- We also provided you with angr skeleton script sample_inputs.py in ubuntu VM to better analyze the behavior, you can check out the code and add more to it
- For malware3.exe, do whatever you can to analyze it. Manual reverse engineering may be a good approach to earn points. Fill in the excel sheet with function descriptions and category checking.
- None of the tools provided will promise to work on this one. Just try your best.

Additional Help for Malware 3

As you already knew/will know that Malware 3 is obfuscated by the malware author, which is most of the case in real day life. One of the most important thing for Malware Analyst is to recover the real payload of malware.

Considering your life and happiness, we deobfuscated the first two malware(free lunch here:>) and left the third intact malware sample that we collected from the network for you to learn some deobfuscation.

Obfuscation:

https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/

https://www.vadesecure.com/en/malware-analysis-understanding-code-obfuscation-techniques/

Deobfuscation:

https://www.youtube.com/watch?v=4VBVMKdY-yg

Resources

- Ghidra:
  - https://vimeo.com/335158460 (1h thorough introduction video)
  - Other tutorials available online
- Dynamorio
  - http://dynamorio.org/tutorial.html
  - https://github.com/DynamoRIO/drmemory/tree/master/drltrace
  - https://github.com/mxmssh/drltrace
- Angr
  - https://docs.angr.io/core-concepts