Chapter 1

Encountering Malware Analysis

Charles Nicholas
Robert Joyce
Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD, USA

1.1 Introduction

With the exponential growth of the malware industry in recent years has come a similar growth in the amount of malware and related data that can be collected. Individual malware specimens can be analyzed, and textbooks (such as [Sikorski and Honig (2012)]) cover this topic. Since Fall of 2012 our university has used Sikorski's book **Practical Malware Analysis**, hereafter often abbreviated PMA, to teach our combined graduate/undergraduate Malware Analysis course. PMA remains a comprehensive introduction to malware analysis, and the concepts, techniques, and labs it provides continue to be relevant. We still regard it as the best available textbook for this subject, although other books on this subject have appeared since its publication, and several features of PMA have become outdated as time has passed:

- Several malware analysis tools in PMA have fallen out of favor, including ApateDNS, INetSim, and OllyDbg, and some new tools need to be addressed.
- Several versions of Windows have come and gone since PMA was published. PMA focuses on Windows XP, which is now used by

only a small fraction of all Windows users.[1,2]
- The malware ecosystem has changed drastically. Attacker tradecraft has evolved, ransomware and cryptocurrency miners have become cybercriminal staples, and malware is increasingly targeting Android and Linux devices.

It is our purpose in this chapter to present a primer on malware analysis, supplementing PMA with malware analysis concepts, resources, and tools that have emerged in the years since its publication. We include several tools that increase analyst productivity via automation or artificial intelligence. The intended audience is the undergraduate or graduate student of cybersecurity or computer science. We assume working knowledge of a high-level language such as C or C++, up to the level of an undergraduate course in data structures. Knowledge of assembly language is quite helpful, but not necessary for the concepts discussed in this introductory chapter. We further assume that the reader is familiar with Windows at the user level. Knowledge of the internals of Windows can be acquired as needed. Basic knowledge of computer networks is assumed, and again additional knowledge can be acquired as needed.

1.2 A Safe Environment for Malware Analysis

To begin, we need to discuss the malware analysis environment, which we define as a hardware platform, a suite of software tools, and other resources of use to the malware analyst. A rudimentary malware analysis environment can be built from the items below:

- A virtualization environment such as VirtualBox or VMWare.
- A virtual machine (VM) with an appropriate operating system for analyzing a particular type of malware.
- A suite of malware analysis tools installed on the VM.
- One or more files to analyze.

Access to a virtualization environment is critical to malware analysis. We begin by explaining some terminology. The host (or base layer) refers to whatever operating system is running on "bare metal". The virtualization

[1]Throughout this volume, we use footnotes to lead the reader to web sites, rather than placing links in the main body of the text, or the references.
[2]`https://www.bleepingcomputer.com/news/microsoft/its-windows-xps-20th-birthday-and-way-too-many-still-use-it/`

environment, whether it be VirtualBox, VMWare, or perhaps some other hypervisor, runs on top of that base layer. The virtualization environment can run several guests, or virtual machines, at the same time if there are sufficient compute resources available.

Running malware specimens inside of a virtual machine safeguards the host from any damaging behavior. Furthermore, most virtual environments offer snapshot capabilities. A snapshot captures the state of a virtual machine in such a way that the VM can be restored to that state in a few seconds. Snapshots are essential for malware analysis, so that after running a malware specimen, the VM can be reverted back to a "clean" state with ease.

A free version of VMWare is used in PMA, and there may be times when VMWare is the best choice. However, since PMA came out, VirtualBox[3] has become widely used, and we likewise use it throughout this chapter. VirtualBox is available for Windows and Linux, but support for Apple Silicon devices may be limited. We will not review the steps required to install VirtualBox on the reader's computer, but the VirtualBox website has ample documentation, and helpful tutorials and demos are available on the Internet.[4]

Once VirtualBox has been installed, the next step is to install a virtual machine. The reader is free to fashion their own malware analysis environment, and there are advantages in doing so. Like PMA, we focus on malware that targets a modern Windows computer; a different analysis environment would be needed to analyze malware that targets other platforms. For this, we strongly recommend setting up a Virtual Machine with a licensed version of Windows 10[5] or higher and installing FLARE-VM[6] on it. Using FLARE-VM, analysts no longer have to individually locate, download, and install every malware analysis tool they plan to use. FLARE-VM provides access to an extensive suite of malware analysis tools, and manages the installation of any tools selected by the user. In our opinion, it is the most streamlined way to create a Windows malware analysis environment.

The reader who attempts to use the various tools available to them will soon realize that a powerful computer is of great help when doing malware analysis. More memory is better than less. Additional monitors make a surprising difference, given that the analyst may need to have a web browser, a

[3] `https://www.virtualbox.org/`
[4] `https://www.virtualbox.org/manual/topics/installation.html`
[5] `https://www.microsoft.com/en-us/software-download/windows10`
[6] `https://github.com/mandiant/flare-vm`

disassembler, a debugger, a text editor, and several other tools open at the same time. An external hard drive may be useful for storing large Virtual Machines and collections of malware specimens. The environment we just described might represent a significant expense, especially on a student's budget. Labs provided by their department or university may provide such an environment, but many students may not have access to such labs. The last section of this chapter describes how to create a malware analysis environment in the cloud, which a student may be able to set up and use from even a modest laptop.

1.3 Accessing Malware Specimens to Analyze

There is no shortage of malware specimens to analyze. The collection of labs[7] provided in PMA is of great help to the novice student, especially since detailed solutions to those labs are given in the back of that book. Samplepedia[8] provides malware analysis labs of varying difficulty, and each has a written or video solution. Vx Underground[9] and SecRepo[10] are massive repositories of malware specimens.

For the purpose of demonstrating several malware analysis tools and concepts, the remainder of this chapter will feature a file named EncounteringLab.exe. The program is not malicious, but it performs potentially-unwanted behaviors that are frequently exhibited by malware. As we will learn by analyzing it, EncounteringLab.exe copies itself to another location, makes a system configuration change so that it runs on startup, creates a mutex (a shared resource object), and makes a DNS request. It also uses a few anti-analysis techniques. If the reader wishes to follow along, EncounteringLab.exe is available for download as an encrypted 7zip file.[11] The password to the 7zip file is `infected`. It is best practice to treat EncounteringLab.exe as if it were live malware – this file, and all such files, should only be downloaded and executed in a safe malware analysis environment.

[7] `https://github.com/mikesiko/PracticalMalwareAnalysis-Labs`
[8] `https://samplepedia.cc/`
[9] `https://www.vx-underground.org/`
[10] `https://www.secrepo.com/`
[11] `https://drive.google.com/file/d/1k18axSmxSbM_lZySiadtbP1mwKLxFOVS`

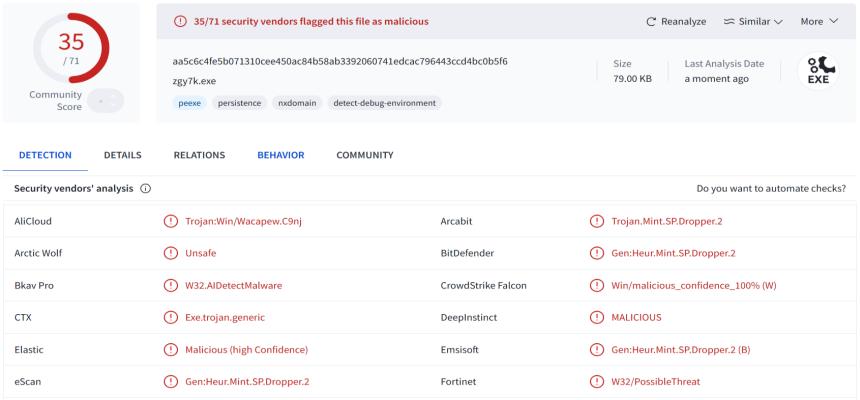


Fig. 1.1: Detection results showing that 35 of 71 anti-virus products in VirusTotal detect EncounteringLab.exe as malware.

1.4 Malware at a Glance Using VirusTotal

The quantity and variety of tools available for the malware analyst may seem intimidating at first. Some tools focus on aspects of the malware specimen as a file, while others focus on aspects of the specimen's execution. This is a quick description of the difference between static and dynamic malware analysis, and each has its place in the malware analyst's toolbox. Some tools are used to take a "quick-and-dirty" look at a file, while others are used to fully understand a file inside and out. Some tools (whether static or dynamic) are run locally, while others are hosted on various websites.

In some instances, the analysis objective is simply to answer the question "Is this file malicious?" For this, we introduce VirusTotal,[12] a web-based malware analysis platform that happens to have both static and dynamic analysis capabilities. VirusTotal can scan suspected files or URLs with a large collection of security products, making it an excellent way to make quick determinations about a suspect file. Each security product then outputs whether it believes the input to be malicious or benign. Some may take a while to return a decision, while others scan files or URLs quickly. VirusTotal accepts submissions of any file type, including executable binaries, scripts, and documents. The set of security products used to scan a file depends on its file type.

[12] `https://www.virustotal.com/gui/home/upload`

Most anti-virus products in the VirusTotal ensemble scan files using a combination of signatures and heuristics. Signatures attempt to match patterns of bytes known to appear in a particular type of malware, while heuristics look for file attributes or behaviors that are associated with maliciousness. Anti-virus products are increasingly incorporating (or in some cases, exclusively employing) machine learning to identify malware.

If a file or URL is flagged by enough security products, it's typically safe to assume that it is malicious. Likewise, when no security products flag a file or URL, it's most likely benign. However, be aware that security products can make mistakes and newly-emerging malware may evade detection. The analyst's interpretation of these detection results depends on the situation and the user's tolerance for risk.

EncounteringLab.exe is not malware, but in many aspects it certainly looks like it. The program is not one that any normal user would want to run. When uploaded to VirusTotal, EncounteringLab.exe appears suspicious enough to be flagged by 35 of the 71 security products that scanned it. Some of the detection results are shown in Figure 1.1.

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY

Basic properties ⓘ

MD5	4efd3fc5a389a9514bcad1e314075645
SHA-1	a8cd372705d3a808506343fe3a21d41b12118ad5
SHA-256	aa5c6c4fe5b071310cee450ac84b58ab3392060741edcac796443ccd4bc0b5f6

Fig. 1.2: The MD5, SHA-1, and SHA-256 hashes EncounteringLab.exe shown in VirusTotal.

Files known to VirusTotal can be queried by their hash digests, which are presumably unique numeric values generated from a file's contents. Malware reports share the hashes of malicious files so that others can check if their networks have been compromised by those files. Clicking the "Details" tab in VirusTotal shows the file's MD5, SHA-1, and SHA-256 hashes, among others.

It must be noted that some malware analysts work in sensitive environments where online analysis tools like VirusTotal should not be used. Uploading a file, or even one of its hash values, to some external web site is proof that you, the malware analyst, have access to that file. An attacker becoming aware of this could ruin an investigation.

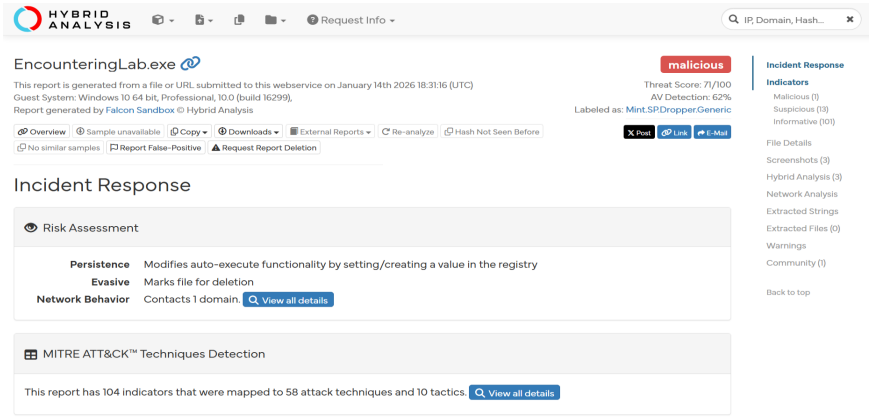


Fig. 1.3: The Hybrid-Analysis sandbox analysis for EncounteringLab.exe.

1.5 Automating Malware Analysis Using Sandboxes

Once it is known that a file is malware, the next question to be asked might be "What malicious things can this file do?" The simplest way to attempt to answer this question is to run the malicious file in a sandbox. A sandbox is a virtual environment for safely running programs and monitoring the behaviors they exhibit. A sandbox may be hosted as a web service or on an analyst's own virtual machine. VirusTotal supports several in-house and external sandboxes.[13] Cuckoo Sandbox is perhaps the most well-known sandbox software that can be hosted locally.[14] In recent years, PANDA (Platform for Architecture-Neutral Dynamic Analysis) has also gained a following as a locally-hosted whole-system dynamic analysis engine.[15]

Hybrid-Analysis[16] is a notable example of a web-based sandbox. When we submit EncounteringLab.exe to the Hybrid-Analysis sandbox, we receive the report shown in Figure 1.3, as well as a longer PDF file with more details. Among other actions that the malware performs, Hybrid-Analysis shows that the malware writes to a file and creates a registry key, as shown in Figure 1.4. First, the malware writes a new file to `%WINDIR%\SysWOW64\Encount- eringLab.exe`.

[13] `https://docs.virustotal.com/docs/in-house-sandboxes`
[14] `https://cuckoosandbox.org/`
[15] `https://github.com/panda-re/panda`
[16] `https://www.hybrid-analysis.com/`

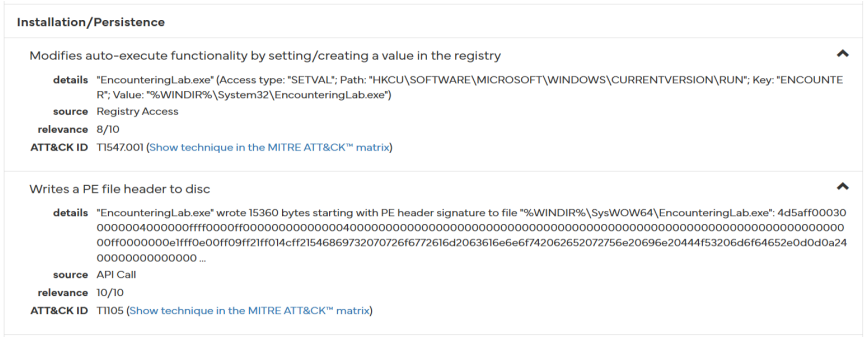


Fig. 1.4: Hybrid-Analysis shows that EncounteringLab.exe writes to a file in the `%WINDIR%\SysWOW64\` folder and makes that file persistent.

`%WINDIR%` is an environment variable that typically has the value `C:\Windows\`. Further analysis will show that the malware is copying itself to the `SysWOW64\` folder. Then, the registry key `HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Encounter` is created and assigned the value `%WINDIR%\SysWOW64\EncounteringLab.exe`. Malware often interacts with the Windows Registry to make configuration changes, and there are several registry keys that grant a program the ability to run automatically when a computer boots up. This is known as a "persistence mechanism". Of the many persistence mechanisms malware is known to use on Windows, the `Run` key used by EncounteringLab.exe is perhaps the most common.

As another example of online sandbox services, we include ANY.RUN. Like Hybrid-Analysis, ANY.RUN supports analysis of suspicious URLs and files of all kinds.[17] With an appropriate license, ANY.RUN will also make the submitted specimens and the corresponding reports available to other users, with the submitters' consent. The submitted specimens include many files found to be benign, as well as specimens that are suspicious, and others that determined to be malicious. The number of specimens available is on the order of hundreds of thousands.

[17] `https://app.any.run/`. For a time, ANY.RUN made enterprise-level access to their site available to the authors for research and educational use, including the preparation of this section.



Fig. 1.5: An excerpt from the archive of reports generated by ANY.RUN.

```
C:\Users\Student>strings -n 8 EncounteringLab.exe

Strings v2.54 - Search for ANSI and Unicode strings in binary images.
Copyright (C) 1999-2021 Mark Russinovich
Sysinternals - www.sysinternals.com

!This program cannot be run in DOS mode.
...
explorer.exe
Global\EncounteringLab_Mutex
C:\Windows\System32\EncounteringLab.exe
C:\Windows\System32\EncounteringLab.exe
Software\Microsoft\Windows\CurrentVersion\Run
C:\Windows\System32\EncounteringLab.exe
C:\Windows\System32\EncounteringLab.exe
Encounter
ae503qhyrxzeor5n7hu7jtey07kchmsco3qlr2ph564jvltw6.com
```

Fig. 1.6: The output of `strings -n 8 EncounteringLab.exe`.

1.6 Utilities for Basic Static Analysis

Although automation can accelerate malware analysis, it has limits. Malware might detect that it is running in a virtual environment, causing a sandbox run to fail. It may exhibit certain behaviors only under specific conditions, such as receiving commands from an attacker via a network connection. In such cases, the analyst will have to get their hands dirty. This often begins by investigating the file's contents and attributes, a stage referred to as basic static analysis. This process heavily depends on the type of file being analyzed, and the scope of this chapter is limited to Win-

dows malware in the Portable Executable (PE) format, a file format that includes EXEs, DLLs, and device drivers.

One of the first steps to be taken in basic static analysis is to scan the suspect binary for readable strings. A non-obfuscated binary, that is, one where the executable has not been packed, encrypted, or in some other way made difficult to read, should have several such strings. The first string in most PE files will be the `This program cannot be run in DOS mode.` message, which is an error message to be printed if the file is run on DOS rather than Windows. Other common strings are other error messages, the names of imported functions, and the names of DLLs. One should be particularly interested in any strings that were defined in the malware's source code and that suggest behaviors the malware might take – for example, if one encounters the string `C:\Windows\System32\evilfile.exe`, they should check whether the malware creates or interacts with that file during later analysis. Other types of strings to pay particular attention to include IP addresses, URLs, user agent names, registry keys, and mutexes. Analysts should also be on the lookout for unique strings that could be used to signature the malware or assist in attribution.

The `strings` command in UNIX, and the utility of the same name in the Sysinternals Suite for Windows, can be used for this. It is often useful to limit the output to strings of at least a certain length using a command such `strings -n 8 EncounteringLab.exe`. This is because printable characters may occur next to each other by chance, causing short "junk" strings. The `strings` output in Figure 1.6 reveals the mutex `Global\EncounteringLab_Mutex`, the file path `C:\Windows\System32\En- counteringLab.exe`, the partial registry key `Software\Microsoft\Window- s\CurrentVersion\Run`, and the random-looking domain name beginning with `ae503`.

The `floss` command-line utility will produce a list of readable strings, in addition to searching for certain types of obfuscated strings. `floss` will, for example, identify strings that are encoded with a simple cipher or constructed on the stack at runtime. Its command-line arguments are very similar to those of `strings`. As shown in Figure 1.7 there are five "stack strings" found by `floss` but not by `strings`. The first, `SeDebugPrivilege`, is a Windows privilege that allows a program to read and modify the memory of any other process. The second, `kernel32.dll`, is the name of a legitimate dynamic-link library file that contains numerous functions for interacting with the Windows operating system via the Windows API. Following that, we see the names of three Windows API functions that appear

```
C:\Users\Student>floss -n 8 EncounteringLab.exe
...

  FLOSS STACK STRINGS (5)

SeDebugPrivilege
kernel32.dll
VirtualAllocEx
WriteProcessMemory
CreateRemoteThread
```

Fig. 1.7: The output of `floss -n 8 EncounteringLab.exe`.

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder
1185C	WS2_32.dll	3	FALSE	125EC	0	0
11870	ADVAPI32.dll	6	FALSE	124AC	0	0
11884	KERNEL32.dll	72	FALSE	124C8	0	0

KERNEL32.dll [72 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder
D024	IsDebuggerPresent	-	1268A	1268A	-
D028	CloseHandle	-	1269E	1269E	-
D02C	GetLastError	-	126AC	126AC	-
D030	CreateMutexA	-	126BC	126BC	-
D034	GetCurrentProcess	-	126CC	126CC	-
D038	FreeLibrary	-	126E0	126E0	-
D03C	GetModuleFileNameA	-	126EE	126EE	-
D040	GetProcAddress	-	12704	12704	-
D044	LoadLibraryA	-	12716	12716	-
D048	CopyFileA	-	12726	12726	-
D04C	CreateToolhelp32Snapshot	-	12732	12732	-
D050	Process32First	-	1274E	1274E	-
D054	Process32Next	-	12760	12760	-

Fig. 1.8: A few of the imports from EncounteringLab.exe's IAT.

within kernel32.dll – we will return to these shortly.

Scrutinizing the functions that an executable file imports is another important aspect of basic static analysis, especially those functions that enable the malware to interact with the victim's computer (e.g. Windows API functions). For PE files, imported functions are listed in the Import Address Table (IAT). Seeing particular functions, or combinations of functions, in a file's IAT can point to specific behaviors the file might perform. For example, the combination of `VirtualAllocEx`, `WriteProcessMemory`,

and `CreateRemoteThread` (in addition to the `SeDebugPrivilege` string from Figure 1.7) indicates that EncounteringLab.exe may try to execute code in the address space of another process. This technique is called process injection.

There are several tools that can display the functions in a PE file's IAT, including PEStudio and Detect It Easy. Here, we use PE-Bear, which shows that EncounteringLab.exe imports 81 functions from three DLL files. One notable inclusion is `IsDebuggerPresent`, which EncounteringLab.exe may use to check whether it is being debugged. The combination of `ToolHelp32- Snapshot`, `Process32First`, and `Process32Next` would allow EncounteringLab.exe to inspect which programs are currently running. Other imports suggest that it can interact with the filesystem and registry, make network connections, and create mutexes. We note that `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` do not appear in the IAT of EncounteringLab.exe. Several techniques can prevent functions from appearing in the IAT, and we will revisit how EncounteringLab.exe achieves this later.

1.7 Basic Dynamic Analysis on a Virtual Machine

At this point, basic static analysis has given us an understanding of EncounteringLab.exe's capabilities, and running it in the Hybrid-Analysis sandbox has revealed some of its behaviors. Using a sandbox is one form of basic dynamic analysis; the other is to run malware on a VM and manually investigate its behavior. There are a host of tools for performing basic dynamic analysis on a VM. Several tools recommended in the Practical Malware Analysis textbook remain relevant at the time of writing:

- Process Monitor, which captures Windows events related to the filesystem, registry, processes, and threads.
- Process Explorer, which is can be used to list and inspect all currently-running processes.
- Regshot, which identifies changes to the filesystem and registry after running a malware specimen.
- Netcat, which lists information about all current network connections.
- Wireshark, which captures and visualizes network traffic.

Before running malware in a virtual machine, it is important to take a



Fig. 1.9: Making a HTTP request while FakeNet-NG is running. The FakeNet-NG HTTPListener intercepts our request and makes this response.

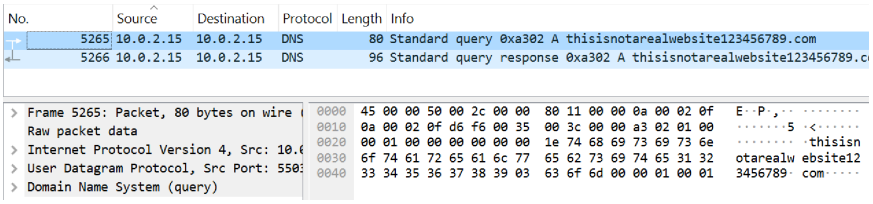


Fig. 1.10: A DNS request from EncounteringLab.exe and the DNS response by FakeNet-NG's DNSListener shown in WireShark.

snapshot of the VM. This allows the VM to be restored to a clean state after it has been infected. It is also crucial to ensure that the VM cannot make external network connections. For this, we want to highlight FakeNet-NG,[18] a tool that was released after the publication of PMA. Fakenet-NG spoofs network connectivity on a VM, tricking a malware specimen into believing it is connected to the Internet, when it is, in fact, running in an isolated environment. Fakenet-NG redirects external network traffic to a set of listeners for common network protocols.

Figure 1.9 shows an example of this network redirection. If we try navigating to a webpage, FakeNet-NG's HTTPListener intercepts our web browser's HTTP request and sends a fake response back. This keeps malware running just a bit longer - without FakeNet-NG, a large portion of

[18] `https://github.com/mandiant/flare-fakenet-ng`

malware specimens would simply exit when they discover that they cannot reach the Internet.

Continuing our analysis of EncounteringLab.exe, we take a snapshot of the analysis VM and run FakeNet-NG. We can confirm that FakeNet-NG is running properly by attempting to make a ping request or by navigating to a webpage in a browser. At this point, we could also run any basic dynamic analysis tools with which we wish to monitor the malware's behavior. Finally, we execute the malware.

FakeNet-NG saves all captured network traffic to .pcap files. With FLARE-VM, these are saved in the `fakenet_logs\` folder on the VM's desktop by default. Double-clicking the .pcap file for the current FakeNet-NG session opens WireShark. Filtering for DNS packets shows that EncounteringLab.exe tried to make a DNS request to the domain `ae503qhyrxzeor5n7- hu7jtey07kchmsco3qlr2ph564jvltw6.com`. We leave investigating EncounteringLab.exe's other behaviors using basic dynamic analysis tools as an exercise for the reader.

1.8 Utilities for Advanced Static Analysis

For a long time, the most important tool available to the malware analysts was the Interactive Disassembler, or IDA. IDA is a commercial product with many useful reverse engineering features. The student malware analyst may be glad to learn that a freeware version of IDA is available.[19] The free local installation of IDA supports only disassembly, but its cloud version offers free decompilation of some architectures.

In addition to IDA, we must mention Binary Ninja,[20] which is functionally similar to IDA and (as we shall see) Ghidra. In addition to having a similar user interface as the former tools, Binary Ninja can also be used from the command line, and from inside Jupyter Notebooks, which is an intriguing vehicle for simultaneously performing analysis and writing a report.

1.8.1 *Using Ghidra*

The release of Ghidra was one of the most important events in the malware analysis field in recent years. Ghidra is a free, open-source tool that can disassemble, decompile, and debug programs. One can download Ghidra

[19] `https://hex-rays.com/ida-free/`
[20] `https://binary.ninja/`

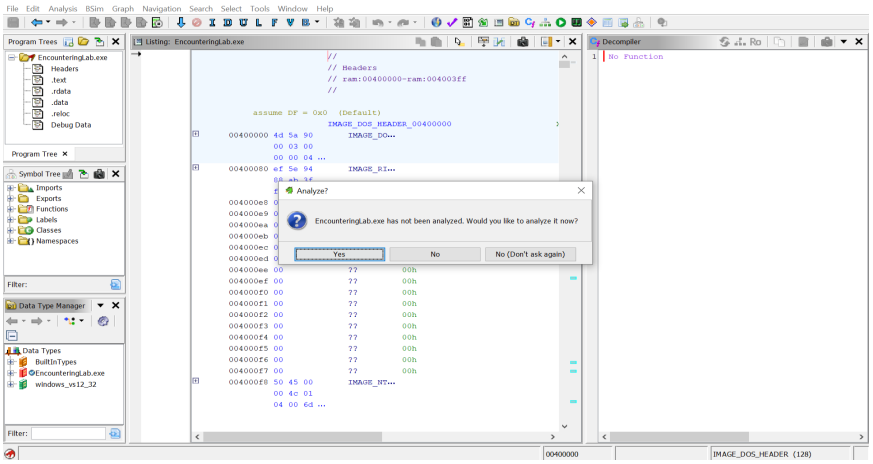


Fig. 1.11: Importing EncounteringLab.exe in Ghidra.

from its web site.[21] The reader may wish to view any of the several Ghidra tutorials available on the Internet, especially YouTube. Serious users of Ghidra may want to acquire the Ghidra Book [Eagle and Nance (2020)]. Our purpose here is not to duplicate those efforts, so we focus on the Ghidra decompiler. Decompiled C code is much easier to interpret than assembly language, and if we can figure out what is going on in some portion of a binary by looking at C code (even if it doesn't exactly match the original source code) then that's an advantage. We still have the option of drilling into the assembly code if needed.

To begin analyzing a file in Ghidra, create a new project using "File" -> "New Project" and import the file using "File" -> "Import File". As shown in Figure 1.11, Ghidra then prompts the user to begin auto-analysis and select which analysis options to apply. In addition to the default analysis options, we recommend selecting "WindowsPE Propagate External Parameters".

Once a PE file has been analyzed, Ghidra places the Listing and Decompilation views at the file's entry point. Although it is possible to reverse engineer a file from start to finish, this is often wasteful. A large portion of functions in a PE file are often boilerplate added by the compiler or functions from statically linked libraries. To avoid wasting analysis time, a

[21] `https://ghidra-sre.org/`

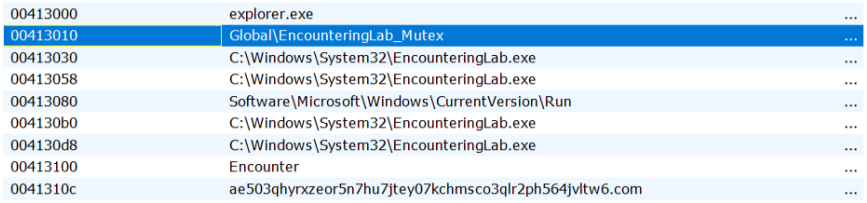


Fig. 1.12: The Defined Strings view in Ghidra. `Global\EncounteringLab_Mutex` appears at the address 0x00413010.

better approach is to begin with functions that contain notable strings or Windows API functions identified during prior analysis.

Importing and analyzing EncounteringLab.exe in Ghidra and then navigating to "Window" -> "Defined Strings" displays the file's strings and the addresses at which they appear. Some of the more notable strings are shown in Figure 1.12. One could begin with any of these, but we will start by investigating `Global\EncounteringLab_Mutex`, which is defined at the address 0x00413010. Double-clicking this address in the Defined Strings view will update each of Ghidra's other views to show that address. Alternatively, "Navigation" -> "Go To" will achieve the same result.

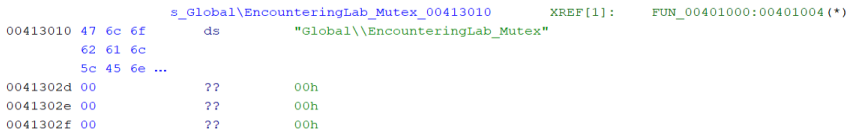


Fig. 1.13: Information about `Global\EncounteringLab_Mutex` in Ghidra's Listing view, including an XREF at 0x00401004.

Switching to the Listing view, shown in Figure 1.13, there is one "XREF" to `Global\EncounteringLab_Mutex` at the address 0x00401004. An XREF, or cross-reference, denotes where one part of the program references another part of the program. Here, the XREF is the result of the program using the string. Navigating to the address 0x00401044 updates the Listing and Decompilation views in Ghidra to show FUN_401000.

FUN_401000, displayed in Figure 1.14, calls three Windows API functions: `CreateMutexA`, `GetLastError`, and `CloseHandle`. When coming across an unfamiliar function, a great first step is to read its documenta-

```
undefined4 FUN_00401000(void)

{
  HANDLE hObject;
  DWORD DVar1;
  undefined4 uVar2;

  hObject = CreateMutexA((LPSECURITY_ATTRIBUTES)0x0,1,s_Global\EncounteringLab_Mutex_00413010);
  DVar1 = GetLastError();
  if (DVar1 == 0xb7) {
    if (hObject != (HANDLE)0x0) {
      CloseHandle(hObject);
    }
    uVar2 = 0;
  }
  else {
    uVar2 = 1;
  }
  return uVar2;
}
```

Fig. 1.14: The Ghidra Decompilation view for FUN_401000.

tion. For example, according to the Microsoft Learn documentation for `CreateMutexA`,[22] its third parameter is a pointer to a string containing the name of the mutex to create. Investigating the arguments passed to `CreateMutexA` in EncounteringLab.exe shows that this function attempts to create the mutex `Global\EncounteringLab_Mutex`. The documentation also states that `CreateMutexA` returns a handle to the created mutex object, or NULL if the function call failed. If the mutex already existed before `CreateMutexA` was called, the error `ERROR_ALREADY_EXISTS` is set. The next line in FUN_401000 is a call to `GetLastError`, which returns the most recently set error code. This is stored in a local variable automatically named `DVar1`.

It is good practice to rename variables and function names in Ghidra throughout the course of analysis. To do this, right-click the item and select "Rename Variable" or "Rename Function". A descriptive new name for `DVar1` would be `DLastError`.

After the call to `CreateMutexA` in FUN_401000, an if statement checks if `DLastError` is equal to 0xb7. This happens to be the hexadecimal value of `ERROR_ALREADY_EXISTS`.[23] If the mutex `Global\EncounteringLab_Mutex` already exists, FUN_401000 attempts to close the handle to that mutex

[22] `https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa`

[23] `https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-`

```
undefined4 CheckForMutex(void)

{
  HANDLE hObject;
  DWORD DLastError;
  undefined4 uReturnValue;

  hObject = CreateMutexA((LPSECURITY_ATTRIBUTES)0x0,1,s_Global\EncounteringLab_Mutex_00413010);
  DLastError = GetLastError();
  if (DLastError == 0xb7) {
    if (hObject != (HANDLE)0x0) {
      CloseHandle(hObject);
    }
    uReturnValue = 0;
  }
  else {
    uReturnValue = 1;
  }
  return uReturnValue;
}
```

Fig. 1.15: The Ghidra Decompilation view for FUN_401000 after renaming the function and two local variables.

and sets the value of another local variable, `uVar2`, to 0. The function returns uVar2, so an appropriate new name might be `uReturnValue`. If the mutex did not already exist, the function sets `uReturnValue` to 1. Finally, the function returns `uReturnValue`.

In summary, FUN_401000 returns 0 if the mutex `Global\EncounteringLab_Mutex` already exists, and 1 if it does not. A descriptive new name for FUN_401000 would be `CheckForMutex`. Figure 1.15 shows FUN_401000 after it and two of its local variables have been renamed.

An appropriate next step for analyzing EncounteringLab.exe is to investigate how the `CheckForMutex` function is used within the file. To do this, XREFs are once again helpful. The Listing view shows that `CheckForMutex` has three XREFs, though only one occurs due to other code in EncounteringLab.exe calling the function. This XREF occurs at the address 0x4018f3 inside of `FUN_004018f0`, shown in Figure 1.16. At the top of `FUN_004018f0` is a call to `CheckForMutex`, which stores its return value in `iVar2`. If `iVar2` is 0 (which occurs when `Global\EncounteringLab_Mutex` already exists), the local variable `uVar3` is set to 1 and then the function returns `uVar3`.

The remainder of `FUN_004018f0` has calls to seven other functions, and only executes if the mutex did not already exist prior to `CheckForMutex` being called. This is a common behavioral pattern in malware that wishes to avoid re-infecting a computer. First, the malware checks if a unique mutex exists on the computer it plans to infect. If it does, the malware determines

```
undefined4 FUN_004018f0(void)

{
  bool bVar1;
  int iVar2;
  undefined4 uVar3;
  undefined3 extraout_var;
  undefined3 extraout_var_00;

  iVar2 = CheckForMutex();
  if (iVar2 == 0) {
    uVar3 = 1;
  }
  else {
    FUN_00401050();
    FUN_004010d0();
    bVar1 = FUN_00401160();
    if (CONCAT31(extraout_var,bVar1) == 0) {
      uVar3 = 1;
    }
    else {
      bVar1 = FUN_00401200();
      if (CONCAT31(extraout_var_00,bVar1) == 0) {
        uVar3 = 1;
      }
      else {
        FUN_004013f0(s_explorer.exe_00413000);
        FUN_00401220();
        FUN_004014b0();
        uVar3 = 0;
      }
    }
  }
  return uVar3;
}
```

Fig. 1.16: The contents of FUN_004018f0 in Ghidra's decompilation view.

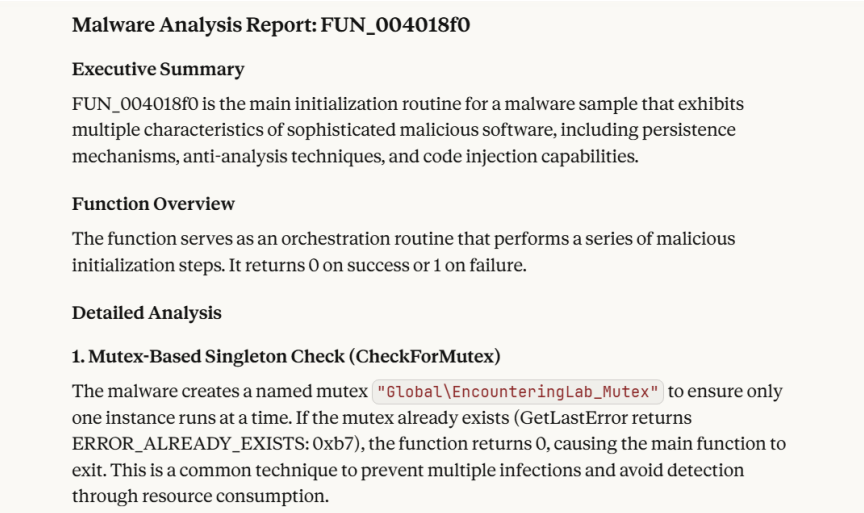
that it has already infected that computer and exits. Otherwise, it creates the mutex and performs its malicious behavior.

1.8.2 *Automating Reverse Engineering with GhidraMCP*

Machine learning is being increasingly utilized to automate malware analysis tasks. Large language model (LLM) copilots show immense promise for streamlining the reverse engineering process. One such tool is XRefer,[24] a plugin for the professional version of IDA. XRefer uses Google Gemini to automatically analyze, identify, and group the functions in an executable file.

We will continue our analysis of EncounteringLab.exe using GhidraMCP,[25] a plugin that allows Ghidra to communicate with a LLM

[24] `https://github.com/mandiant/xrefer`
[25] `https://github.com/LaurieWired/GhidraMCP`



Malware Analysis Report: FUN_004018f0

Executive Summary

FUN_004018f0 is the main initialization routine for a malware sample that exhibits multiple characteristics of sophisticated malicious software, including persistence mechanisms, anti-analysis techniques, and code injection capabilities.

Function Overview

The function serves as an orchestration routine that performs a series of malicious initialization steps. It returns 0 on success or 1 on failure.

Detailed Analysis

1. Mutex-Based Singleton Check (CheckForMutex)

The malware creates a named mutex `"Global\EncounteringLab_Mutex"` to ensure only one instance runs at a time. If the mutex already exists (GetLastError returns ERROR_ALREADY_EXISTS: 0xb7), the function returns 0, causing the main function to exit. This is a common technique to prevent multiple infections and avoid detection through resource consumption.

Fig. 1.17: The beginning of a report about FUN_004018f0 in EncounteringLab.exe, created by interfacing Claude with Ghidra using GhidraMCP. It correctly identifies the mutex check we found in our own analysis.

via the Model Context Protocol (MCP). GhidraMCP allows a connected LLM to interact with a program being reverse engineered in Ghidra, and it provides nearly two dozen built-in commands. The GhidraMCP installation page has instructions for connecting the plugin with Claude Desktop, so we will use it as well. Specifically, the malware analysis tasks performed below were performed using Claude Opus 4.5.

Claude was instructed to reverse engineer FUN_004018f0 (the function we began analyzing at the end of Section 1.8.1) using the following prompt: "In Ghidra, analyze FUN_004018f0 and the functions it calls. Output a report about this code's functionality." The beginning of its response is shown in Figure 1.17. After this, we prompt Claude one more time: "In Ghidra, rename all functions and variables in FUN_004018f0. Give each a descriptive new name." Using GhidraMCP, Claude was able to modify the contents of FUN_004018f0 in Ghidra, making the function much easier to understand. The modified function is shown in Figure 1.18.

Claude's report contained detailed and (mostly) correct analysis of each function while drastically reducing the time it would have taken to reverse engineer them. XRefer, GhidraMCP, and similar tools have massive poten-

```
undefined4 MalwareInitialization(void)

{
  bool checkResult;
  int mutexCheckResult;
  undefined4 returnValue;
  undefined3 extraout_var;
  undefined3 extraout_var_00;

  mutexCheckResult = CheckForMutex();
  if (mutexCheckResult == 0) {
    returnValue = 1;
  }
  else {
    InstallToSystem32();
    CreateRegistryPersistence();
    checkResult = CheckC2ServerReachable();
    if (CONCAT31(extraout_var,checkResult) == 0) {
      returnValue = 1;
    }
    else {
      checkResult = CheckDebuggerPresent();
      if (CONCAT31(extraout_var_00,checkResult) == 0) {
        returnValue = 1;
      }
      else {
        FindExplorerProcess(s_explorer.exe_00413000);
        ObtainDebugPrivilege();
        LoadProcessInjectionAPIs();
        returnValue = 0;
      }
    }
  }
  return returnValue;
}
```

Fig. 1.18: The contents of FUN_004018f0 in Ghidra's decompilation view, after Claude has renamed each variable and function using GhidraMCP.

tial in aiding malware analysts. At the same time, it is important to keep in mind that these tools are not infallible, and malware analysts must be diligent about how these tools are applied and how their outputs are interpreted. We encourage the reader to reverse engineer these functions themselves and compare their own analysis to that of a LLM. Below, we provide brief summaries of each remaining function called inside FUN_004018f0.

- **FUN_00401050**. This function uses `GetModuleFileNameA` and `str- icmp` to check whether its current path is `C:\Windows\System32\Enc- ounteringLab.exe`. If not, it copies itself to that location. It is common for malware to copy itself to a system folder, where it may blend in with other legitimate programs.
- **FUN_004010d0**. This function uses `RegOpenKeyExA` to get a

handle to the registry key `Software\Microsoft\Windows\CurrentVersion-\Run`. It creates the subkey `Encounter` with the data entry `C:\Windows\System32\EncounteringLab.exe` using `RegSetValueExA`. Then, it closes the handle to the registry key. This makes the copy of EncounteringLab.exe persistent.

- **FUN_00401160**. This function makes a DNS request to `ae503qhyr- xzeor5n7hu7jtey07kchmsco3qlr2ph564jvltw6.com` using `gethost- byname`. If the DNS request is successful, the function returns 0. If it fails, it returns 1. Claude's report speculates that this function "likely serves as a kill-switch or verification that the C2 server is reachable." It ultimately renames the function `CheckC2ServerReachable`, when its true purpose is the former. This function is an imitation of the WannaCry ransomware's "kill switch domain" anti-analysis check.[26] It makes a DNS request to a domain that is unregistered at the time of writing. If EncounteringLab.exe is run while a tool like FakeNet-NG is answering all DNS requests, it determines that it is being analyzed and the rest of FUN_004018f0 is skipped.
- **FUN_00401200**. This function checks whether EncounteringLab is running in a debugger using `IsDebuggerPresent`. It returns 0 if the program is being debugged and 1 if it is not. If the function returns 0, the rest of FUN_004018f0 does not run.
- **FUN_004013f0**. This function takes a pointer to a string as an argument. FUN_004013f0 calls `CreateToolhelp32Snapshot`, which creates a list of processes that are running at that moment. The function then calls `Process32First` and `Process32Next` in a loop over those processes. It compares the name of each process to the string argument using the case-insensitive compare function `stricmp`. If any process' name matches that string, the function returns 1. Otherwise, it returns 0. Searching through processes is a common behavior in malware, and one that may have a multitude of purposes. Malware may be checking if it is running in a VM, searching for the presence of a security product, or searching for another process to inject into. When called in FUN_004018f0, its parameter is `explorer.exe`, which manages the Windows File

[26] `https://www.checkpoint.com/cyber-hub/threat-prevention/ransomware/wannacry-ransomware/`

Explorer and Windows GUI, and is a common target of process injection. Similar functions often return a handle for the process the malware plans to inject into, rather than a boolean value.

- **FUN_00401220**. At the beginning of this function, the string `SeDe- bugPrivilege` is created by individually copying each character to a location on the stack. Ghidra's Decompilation view abstracts this by defining a function named `builtin_memcpy`, but the behavior is visible in the Listing view's disassembly. As discussed in Section 1.6, this causes `SeDebugPrivilege` to not appear in the output of `strings`, but it is still found by `floss`. As we mentioned earlier, `SeDebugPrivilege` is a Windows privilege that allows a program to read and modify the memory of another process. The function uses a combination of `OpenProcessToken`, `LookupPrivilegeValueA`, and `AdjustTokenPriv- ileges` to grant itself this privilege. When running as a normal user, malware needs this privilege to perform process injection.
- **FUN_004014b0**. This function begins by constructing four more stack strings: `kernel32.dll`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. It then calls `LoadLibraryA` with the parameter `kernel32.dll`, causing `kernel32.dll` to be loaded into EncounteringLab.exe's address space. Next, `FUN_004014b0` calls `Get- ProcAddress` three times, getting pointers to the functions `Virtual-AllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` in `ker- nel32.dll`. The combination of `LoadLibraryA` and `GetProcAddress` allows access to any function from any DLL on the computer. This technique is called runtime linking, and it allows EncounteringLab.exe to keep`VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemote- Thread` from appearing in its IAT. These three functions can be used to reserve memory in another process' address space, write code and/or data to that memory, and start a thread inside of another process. Their presence is rare in typical executable files, and would likely cause a program to get flagged by a security product due to the program's ability to perform process injection.

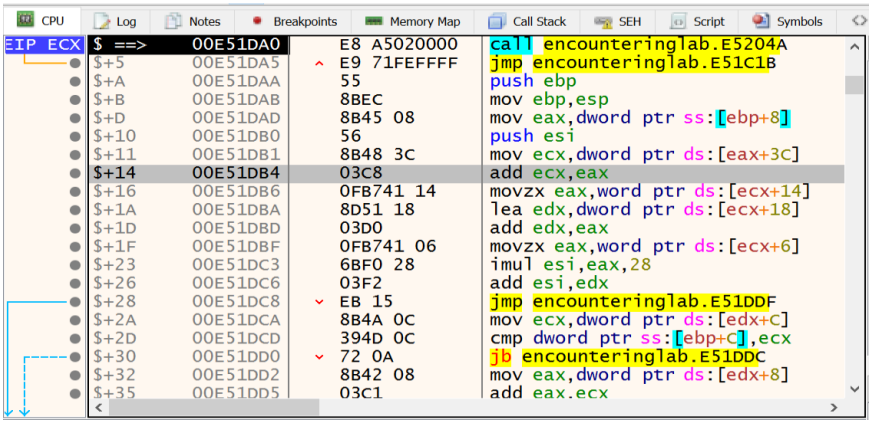


Fig. 1.19: Execution paused at the entry point of EncounteringLab.exe in x32dbg's CPU view.

1.9 Analyzing Malware with a Debugger

The fact that malware is often complex and attempts to hide its behavior should not come as a surprise to the reader. Frequently, analysts are faced with code that is difficult to understand or whose behavior cannot be determined without first running the program. Often, the best approach is to use advanced dynamic analysis – running the program in a debugger and investigating exactly how the registers, stack, and other memory are affected. Debuggers also allow an analyst to control a program's execution, forcing execution to take a code path it may not have ordinarily followed.

We assume the reader is familiar with debuggers from other courses in programming. We recommend x32dbg and x64dbg, open-source debuggers for both both 32- and 64-bit applications respectively. EncounteringLab.exe was compiled for a 32-bit architecture, so we make use of x32dbg in this chapter, but both debuggers are functionally identical. Be cautious when debugging malware – like basic static analysis, it should be done inside of a safe environment with external network connectivity disabled. It is good practice to take a snapshot prior to debugging malware and restoring to a clean state afterwards.

Upon opening EncounteringLab.exe in x32dbg, execution is shown paused in `ntdll.dll`. Like similar debuggers, x32dbg begins execution outside of the debugged program and attempts to place a one-time break-

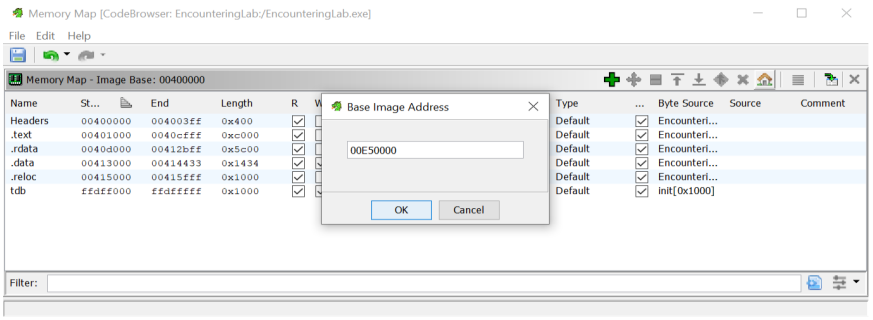


Fig. 1.20: The assembly contents of FUN_00401DA0, which was relocated to 0x00E51200.

point at its entry point. Selecting "View" -> "Breakpoints" shows that there is indeed a breakpoint placed at 0x00E51DA0. Clicking "Debug" -> "Run" (or Run icon) once causes execution to reach this breakpoint and pause. Figure 1.19 shows execution paused at the entry point.

Comparing the program's entry point in x32dbg to the entry point reported by any static analysis tool would reveal a discrepancy: The entry point is located at 0x00E51DA0 in x32dbg, but the entry point is listed as 0x00401DA0 in EncounteringLab.exe's PE header. The reader who follows along may see the entry point in x32dbg at a different address than either of the above. This is due to relocation, which presents an issue if trying to debug a program while simultaneously referencing a tool such as Ghidra, since addresses between the two will not match. The issue must be resolved in the disassembler/decompiler tool.

The image base of the program can be found in x32dbg using "View" -> "Memory Map". In our debugging session, the entry for EncounteringLab.exe lists the address of the image base as 0x00E50000. Ghidra allows the image base to be changed by navigating to "Window" -> "Memory Map" and clicking the "Set Image Base" icon that appears as a house (see Figure 1.20). Setting the image base in Ghidra to match that of x32dbg allows addresses to match again.

The function previously at 0x00401200 in Ghidra has been re-mapped to the address 0x00E51200. This address can be navigated to in x32dbg by right-clicking the upper-left pane of the CPU view, selecting "Go to" -> "Expression", and entering 0x00E51200. To set a breakpoint at 0x00E51200, right click the address and select "Breakpoint" -> "Toggle".

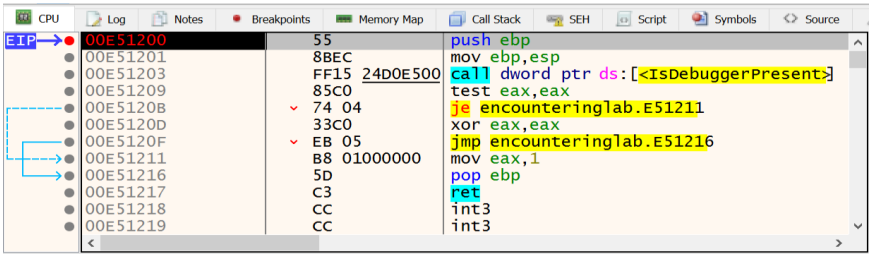


Fig. 1.21: The assembly contents of FUN_00401DA0, which was relocated to 0x00E51200.

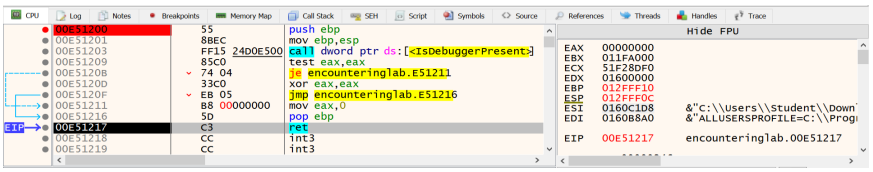


Fig. 1.22: The assembly contents of FUN_00401DA0 after being patched to ignore the debugger check.

Finally, selecting the "Run" icon allows execution to reach 0x00E51200.

Be cautious when attempting to run code to a breakpoint. Malware may be able to take several code paths, and execution will not necessarily take the code path that reaches a given breakpoint. However, in this instance, the program's execution does pause at the desired address.

Figure 1.21 shows the assembly code of the function at 0x00E51200. Recall that this function checks whether the program is being debugged by calling `IsDebuggerPresent`. The function returns 0 if a debugger was detected and 1 otherwise. In assembly, the RAX register stores the function's return value, and at the address 0x00E51211 we see assembly instruction `MOV EAX, 1`. This instruction executes if `IsDebuggerPresent` detects that the program is being debugged, and would cause EncounteringLab.exe to exit early. One way to circumvent this is to patch the assembly instruction in a way that forces the function to return 0 regardless of whether a debugger is in use.

The assembly instruction `MOV EAX, 1` can be patched by right-clicking it in x32dbg, selecting "Assemble", and entering `MOV EAX, 0`. EncounteringLab.exe can be permanently patched using "File" -> "Patch file". The

patched function is shown in Figure 1.22. Stepping to the "RET" instruction shows that the EAX register is 0, even though the program is being debugged. This nullifies the debugger check, and we are free to continue debugging the program further. The reader is encouraged to try patching other parts of EncounteringLab.exe themselves, including the function containing the "kill switch" DNS request.

1.10 Using the Cloud for Malware Analysis

Before concluding this chapter, it seems appropriate to admit that tools we've described so far, including disassemblers, debuggers, not to mention virtualization environments running different operating systems, may put quite a load on an ordinary laptop. Indeed, there may be several reasons to use a cloud service as a platform for malware analysis:

(1) The user's own laptop (or desktop) computer may not have enough CPU power, or memory, to run several VMs at once.
(2) The user's computer may not support virtualization, and therefore not be able to run VMWware or VirtualBox at all.
(3) The user may want to analyze malware in the cloud in order to conform to policy that forbids the presence of malware on their computer, even if virtualization is used.
(4) The user may want to have access to the same malware analysis environment from a variety of physical locations, such as home, work, or when on travel.

Cloud service providers, or CSPs, offer their customers access to a variety of virtual machines. This variety may include a choice of operating systems, number and type of CPUs, memory, disk storage, and software configurations. Selecting the right configuration is mostly a trade-off between capability and budget. Many CSPs offer a free tier of services, and charge more depending on the size of the virtual machine, and the level of access desired. For the student malware analyst, free or at most "pay as you go" is probably the best choice.

Before describing how to set up a Windows desktop suitable for malware analysis using a particular cloud service, we need to discuss nested virtualization. Nested virtualization can be described as the ability to run a virtual machine inside another virtual machine. If one acquires a Windows or Linux desktop running in the cloud, that desktop is actually a VM running under the control of a hypervisor (such as Xen), and it's the

hypervisor that is running on real, bare metal hardware. Cloud service providers vary in the extent to which, and at what cost, they make nested virtualization available.

Whatever CSP we use, we intend to connect to the cloud platform using a virtual network controller, or VNC. Several VNC servers are freely available on the Internet. The Remote Desktop Protocol, or RDP, would be an option were we to connect to a Windows VM from a Windows PC. For the sake of licensing if no other reason, Linux VMs seem to be the more common CSP offering. In what follows, we describe the creation of an Ubuntu VM, and how to connect to it using VNC. We caution the reader that details in what follows may change at any time.

1.10.1 *Google Compute Engine*

The Google Cloud Platform (GCP), and the Google Compute Engine in particular, is the example we use. The place to start is the GCP Console, accessed at `https://console.cloud.google.com/`, as shown in Figure 1.23

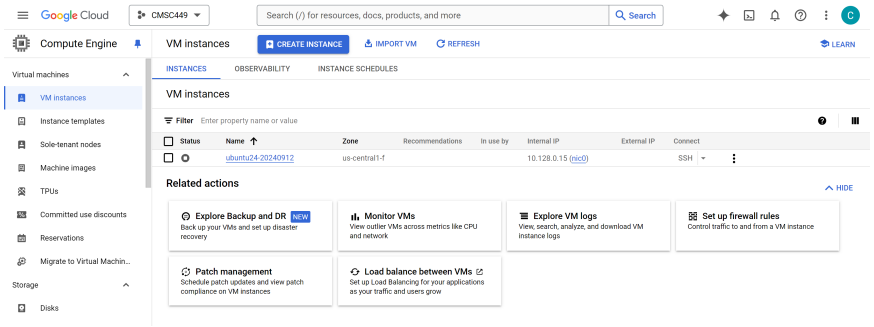


Fig. 1.23: Google Compute Engine Console, showing a single VM instance. Note the "CREATE INSTANCE" button highlighted in blue.

The GCP console may look different at your institution. We again caution the reader that systems change, and what worked for us as we were preparing this chapter may not work for you. Details large and small may need to be updated.

From the Console, use the "Create Instance" button to bring up a menu, allowing you to create a new VM instance, with many possible configurations. For malware analysis, which may require running one or two virtual machines, as well as the host operating system, choose a configuration with

at least 16GB of RAM. Each of those VMs will require a couple of processor cores at least, so choose a configuration with at least eight cores. Adjust the size of the boot disk to at least perhaps 200 GB, since OVA files, VMs, and snapshots can use a lot of disk space!

Referring back to the matter of nested virtualization: The GCP documentation suggests that a chip such as the Intel Haswell, or better, is required. In the example in the figures, the machine type we've selected is "n2-highcpu-16", running on an Intel "Cascade Lake" or better, which meets GCP's requirement. See Figure 1.24

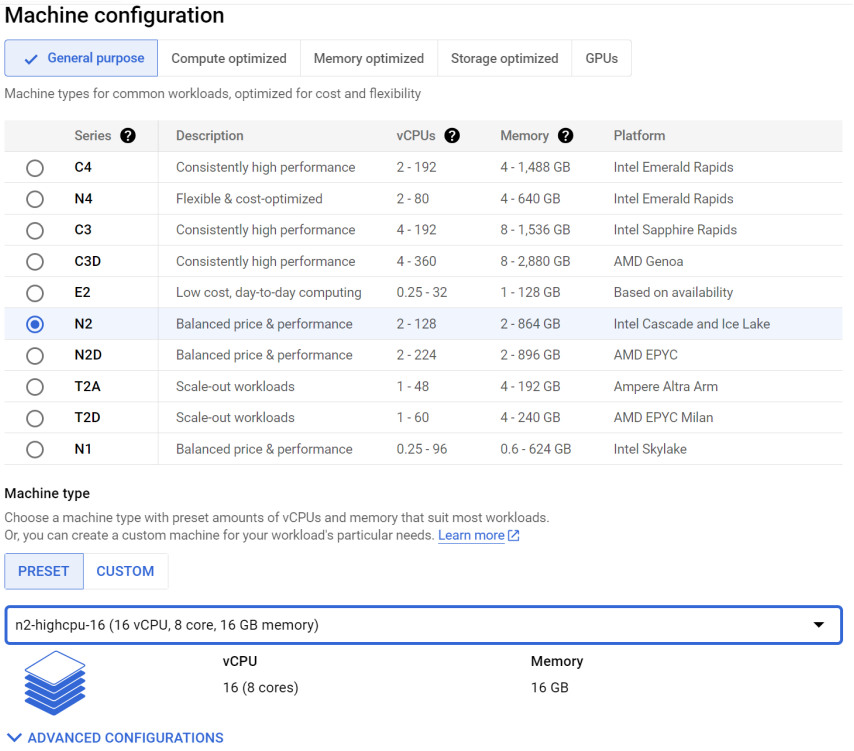


Fig. 1.24: Machine configuration menu. We need a machine that has several cores, sufficient RAM, and support for nested virtualization.

We will need more than the default disk configuration, so scroll down to the Boot disk area. Select "CHANGE", and note the various Boot

disk options. We've selected the Ubuntu version 24.04, and a size of 200 GB, as shown in Figure 1.25. Press "SELECT" to return to the Machine Configuration Menu, and press the "CREATE" button at the bottom.

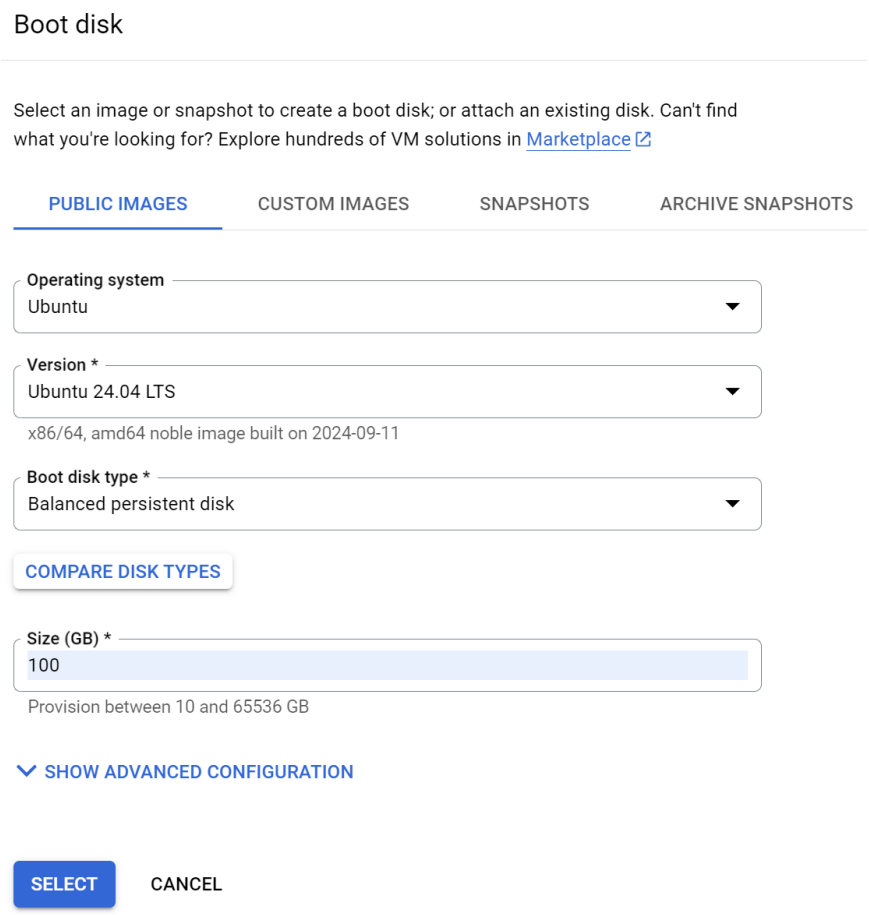


Fig. 1.25: Selecting the OS and boot disk capacity.

After a few moments, an updated list of VM instances appears, as shown in Figure 1.26. The default name for a VM is based on its creation timestamp. The green circle with a check mark indicates that the new VM is already running. Take note of its "External IP", which will change each

time the instance is started.

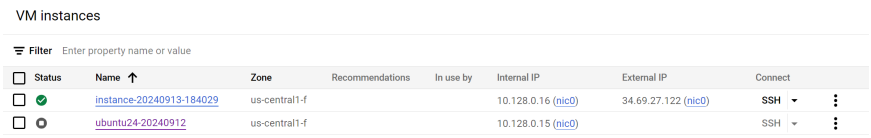


Fig. 1.26: Listing of VM instances, with the VM we just created at the top.

We can start using this VM by connecting to it with ssh. Press "SSH" on the far right. By default, an ssh client built into the browser is used, and we see no reason to change that. The first action is an exchange of ssh keys, and no password is needed - or allowed. The default user has sudo access. After a moment, a new shell appears, as shown in Figure 1.27.

To see that our shell has sudo access, see if the command `sudo whoami` replies with "root". Now, running a shell is a good start, but we need to do some additional setup to use this VM as a remote desktop. From among several options, we've chosen to use the xfce4 desktop, a well-known and quite suitable window manager. To install xfce4, we first need to install the tasksel utility, as shown below. Installing tasksel make take a few moments, but it makes installing xfce4 much easier.

```
sudo apt-get update && sudo apt-get upgrade
sudo apt install tasksel -y
sudo tasksel
```

Invoke tasksel as shown above, select the Xfce option, tab down to the bottom, and press "Okay", as shown in Figure 1.28

Once xfce is installed, we'll install a VNC server as shown below. This too will take a few minutes. Once all this is done, we can invoke the vnc-server. This last command will prompt for a password. Use the command "lsof -i :5901" to see that the Xtightvnc process is listening to port 5901. If that succeeds, kill the server because some additional configuration work needs to be done. The process is summarized here:

```
sudo apt install tightvncserver
sudo apt-get install autocutsel
sudo apt-get install dbus-x11
```

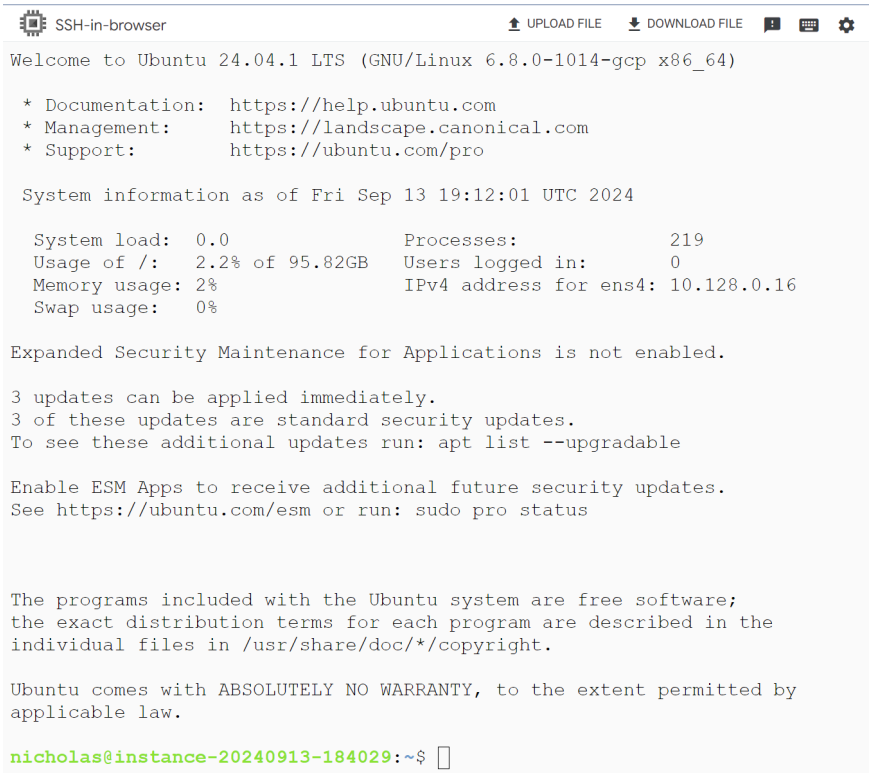


Fig. 1.27: A new SSH session, using the in-browser client. Note the icons for uploading and downloading files.

```
vncserver
(prompts and confirms a password)
lsof -i :5901
(see if the VNC server is listening on port 5901
vncserver -kill :1
```

Now use a text editor such as `nano` to modify the file `.vnc/xstartup` as shown:

```
#!/bin/sh
xrdb $HOME/.Xresources
startxfce4 &
```

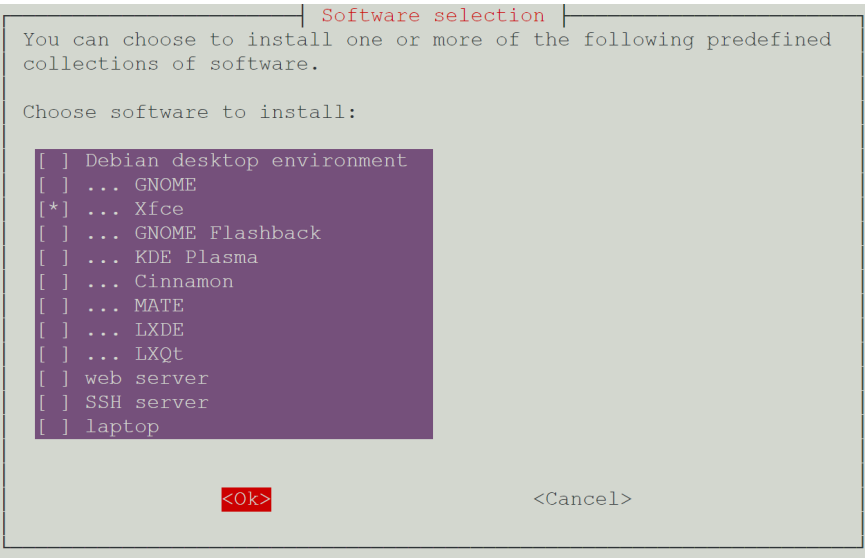


Fig. 1.28: Use tasksel to install the Xfce window manager.

Issue the command `vncserver` and, having done so, verify that our VNC Server is listening on port 5901 using the command `lsof -i :5901`. Feel free to specify a window size, e.g. `vncserver --geometry 1600x1200`. we should now be able to connect to our new Ubuntu VM using a VNC client of our choice. We happen to prefer Remote Ripple, from the same people who wrote Tight VNC. Scale the VNC window as you wish. See Figures 1.29, 1.30, and 1.31

In the GCE version of Ubuntu, VirtualBox is not installed by default. Instructions for installing VirtualBox are available on the web, and there are several ways of doing it, so we omit such instructions here. The instructions will likely describe adding an appropriate repository, adding security keys, and running `sudo apt-get update` followed by `sudo apt-get install virtualbox`. To actually run VirtualBox, however, it turns out that nested virtualization needs to be explicitly enabled for the Ubuntu instance using the Cloud Shell. Stop the newly created Ubuntu instance, and give it time to shut down gracefully. (Feel free to refer to the GCP documentation.[27]) Invoke the Google Cloud shell, by tapping the `">_"` icon at the upper right,

[27] `https://cloud.google.com/compute/docs/instances/nested-virtualization/`

```
nicholas@instance-20240913-184029:~/.vnc$ cat xstartup
#!/bin/sh

xrdb "$HOME/.Xresources"
startxfce4 &
nicholas@instance-20240913-184029:~/.vnc$ vncserver

New 'X' desktop is instance-20240913-184029.us-central1-f.c.cmsc449.inter
nal:1

Starting applications specified in /home/nicholas/.vnc/xstartup
Log file is /home/nicholas/.vnc/instance-20240913-184029.us-central1-f.c.
cmsc449.internal:1.log

nicholas@instance-20240913-184029:~/.vnc$
```

Fig. 1.29: Start the VNC server.

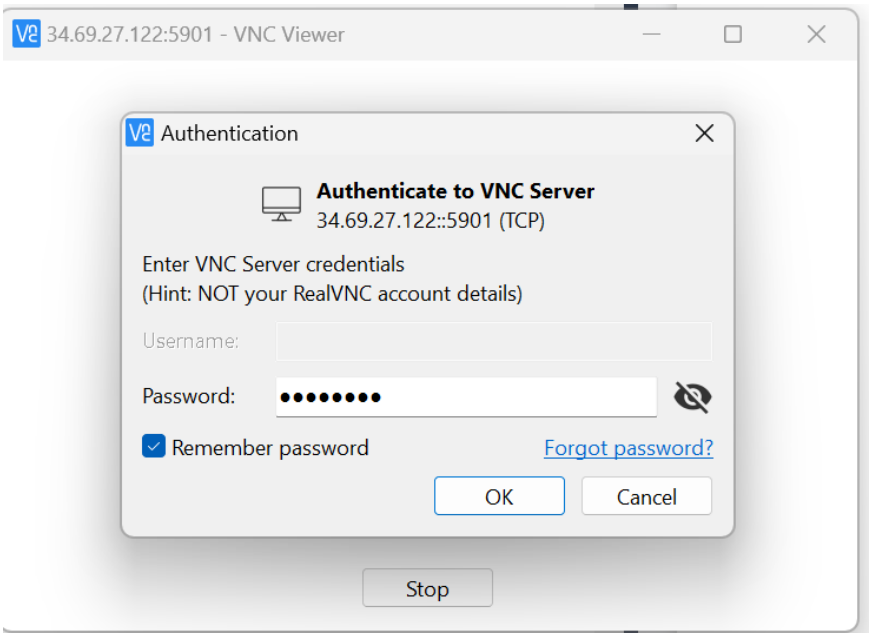


Fig. 1.30: Enter a password to connect the VNC session.

indicated in Figure 1.32.

The new Cloud Shell session may take a moment to start, but you'll soon see a shell prompt. Have the shell write a configuration file to "yaml.txt",

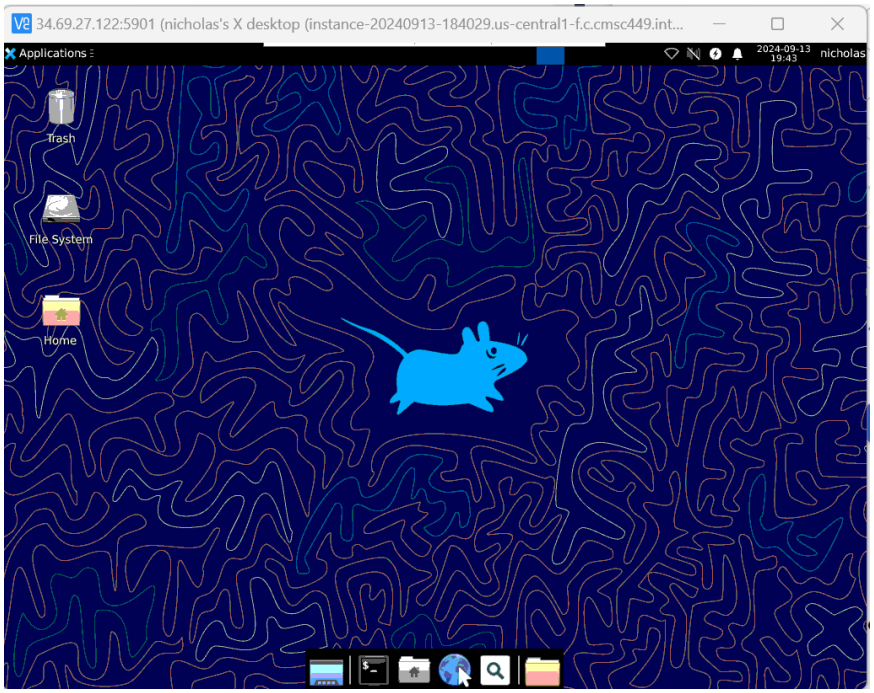


Fig. 1.31: We've connected to a new Ubuntu VM!

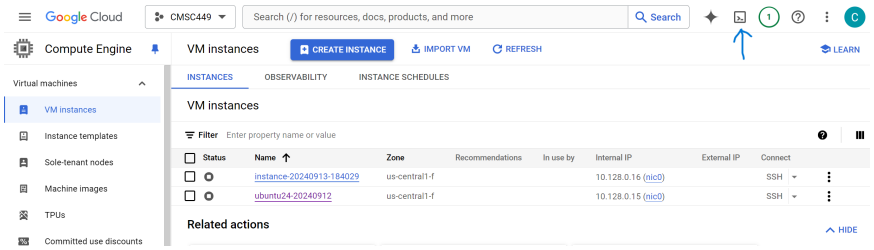


Fig. 1.32: The icon used to activate the cloud shell.

and then edit that file to include the parameter setting to enable nested virtualization, as shown below.

```
nicholas@cloudshell:~ (cmsc449)$ gcloud compute instances \
export instance-20240913-184029 --destination=yaml.txt \
```

```
--zone="us-central1-f"
Exported [instance-20240913-184029] to 'yaml.txt'.
nicholas@cloudshell:~ (cmsc449)$ nano yaml.txt
nicholas@cloudshell:~ (cmsc449)$ tail -3 yaml.txt
advancedMachineFeatures:
  enableNestedVirtualization: true
```

Now, upload the edited configuration file.

```
nicholas@cloudshell:~ (cmsc449)$ gcloud compute instances \
update-from-file instance-20240913-184029 --source=yaml.txt \
--most-disruptive-allowed-action=RESTART --zone="us-central1-
f"
Updated [.../us-central1-f/instances/instance-20240913-
184029].
nicholas@cloudshell:~ (cmsc449)$
```

We then start the Ubuntu instance, connect to the machine using ssh, and confirm that the command `grep -cw vmx /proc/cpuinfo` returns something non-zero. (As we mentioned earlier, the GCP documentation says to make sure the VM is running on an Intel Haswell or later.)

1.10.2 *Connecting to the New GCE Instance*

Once the virtual machine has been created as described above, we need to install VirtualBox and start running virtual machines. It's usually a good idea to do an update and upgrade, especially if the VM has not been used for a while. Issue the command `vncserver` after the updates, and use your VNC Client to access the VM. If you've forgotten the password for your VNC Server, remove the file `.vnc/passwd` and restart vncserver. It may prompt for a new password.

Before doing much more, we need to get a web browser working. Ubuntu comes with Firefox, which can be launched from the Applications menu under Internet, or by creating a desktop launcher. If Firefox doesn't seem to come up right away, use the command "xhost +localhost" from a shell and try again.

1.10.3 *Installing VirtualBox and Virtual Machines*

Much of the rest of the setup for VirtualBox can be done from a shell and a browser running on the Ubuntu instance. Instructions for installing

VirtualBox on Ubuntu are available on the Internet, so you need not follow the instructions given here. From a shell, enter "sudo apt install virtualbox". There will be several lines of output, and then the shell prompt returns. To see if it worked, try the command "VirtualBoxVM –help" to see version information and the command line options. Use the command "virtualbox" in the usual way.

Since we have a browser, you can visit whatever web site you wish to find OVA files available for download. We have OVA files for Kali Linux and Windows Server, among others. Download the OVA files, use the VirtualBox File menu to import the new appliance(s), and see if they work!

We now have the ability to run virtual machines, using VirtualBox, but those machines will not be able to access the Internet until they can acquire IP addresses. So, from a shell on the Ubuntu instance, issue the command `VBoxManage list vms` to see the names of the VMs, and then enter `VBoxManage modifyvm aNewVM --natdnsproxy1 on` where "aNewVM" is replaced with that VM's name in VirtualBox. This will allow the guest operating systems to go out to the Internet for updates, and web browsers running on the guest will be able to reach the Internet.

Some helpful hints:

- Before starting a newly-imported VM, check the settings. Both Kali and Windows Server will run with 2 gigs of RAM, but 4 gigs is better.
- When interacting with Windows, ctrl-alt-del is used a lot. Don't do this with your physical keyboard - use the VirtualBox menu Input; Keyboard; Insert Ctrl-Alt-Del instead.
- If you find a virtual machine taking control of your mouse, pressing Right-Ctrl may help.
- When booting Kali Linux, and if it hangs, check the Settings. Under System;Processor, check both Extended features.
- VirtualBox usually selected reasonable defaults. For example, set paravirtualization to default, unless experience indicates otherwise. Linux prefers KVM, while Windows prefers Hyper-V.
- We found it convenient to use the Workspaces provided by the Ubuntu desktop. For example, run a shell in one workspace, Firefox in another, Kali in the third, and the Windows Server OS in the fourth.
- When building a new VM, whether from an ISO image or an OVA file, it's probably better to build the new VM on an ordinary (non-cloud) desktop. Let it do updates there. Once the new system is updated and stable, create an OVA file and move that up to the cloud, where it can

be unpacked and used.

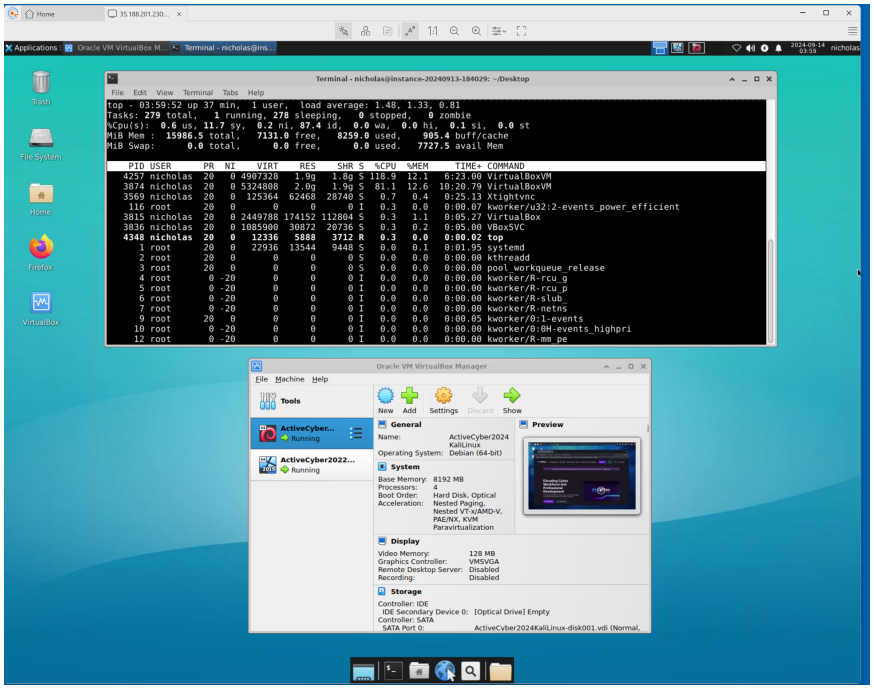


Fig. 1.33: Running a shell and the VirtualBox GUI on our Ubuntu instance

1.10.4 *Optional Setup: Disks in the Cloud*

Given the size of OVA files, and that we may need several of them, we might want to add a standard disk, which we need to format. The GCP console has links to instructions. In some earlier work of this sort, we set up the disk /dev/sdb to be mounted at /mount/disks/disk1, with fstab configured to mount it automatically at boot time. (In a situation where several students in the same class may need read-only access to a set of ISO or OVA files, the disk can be shared between virtual machine instances belonging to the same GCP project.)

Once the various guest systems are updated and stable, it's not a bad idea to export them as VirtualBox appliances, that is, as OVA files. Like-

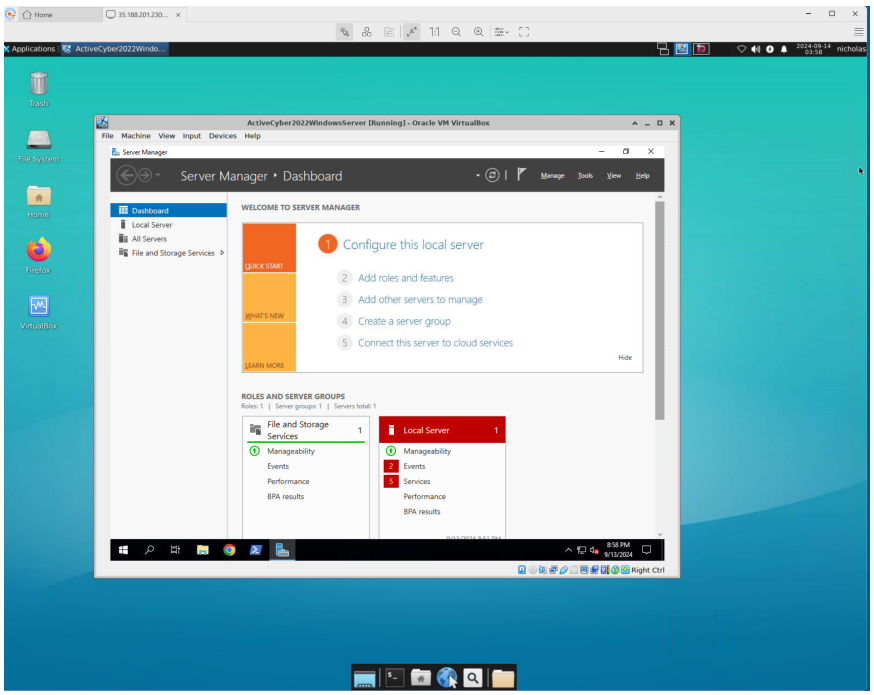


Fig. 1.34: Running Windows Server as a guest on the Ubuntu instance

wise, once the GCP instance is able is able to support VNC connections and run VirtualBox successfully, creating an image of that VM instance as a backup is appropriate.

The process of starting a VM, connecting to it with VNC, and using VirtualBox to create virtual machines can be summarized as follows:

- Go to the GCP Compute Engine VM Instances page
- Select the VM instance to be started, and start it
- Note the external IP address `nnn.nnn.nnn.nnn`
- Use SSH (right menu) to connect to that VM once it has started
- From that shell, run `vncserver`
- Use a VNC client to connect to `nnn.nnn.nnn.nnn:5901`
- When the remote desktop appears, zoom to your level of comfort
- Invoke VirtualBox from the new desktop
- If creating or importing a new VM into VirtualBox, issue the VBoxManage modifyvm command as shown above.

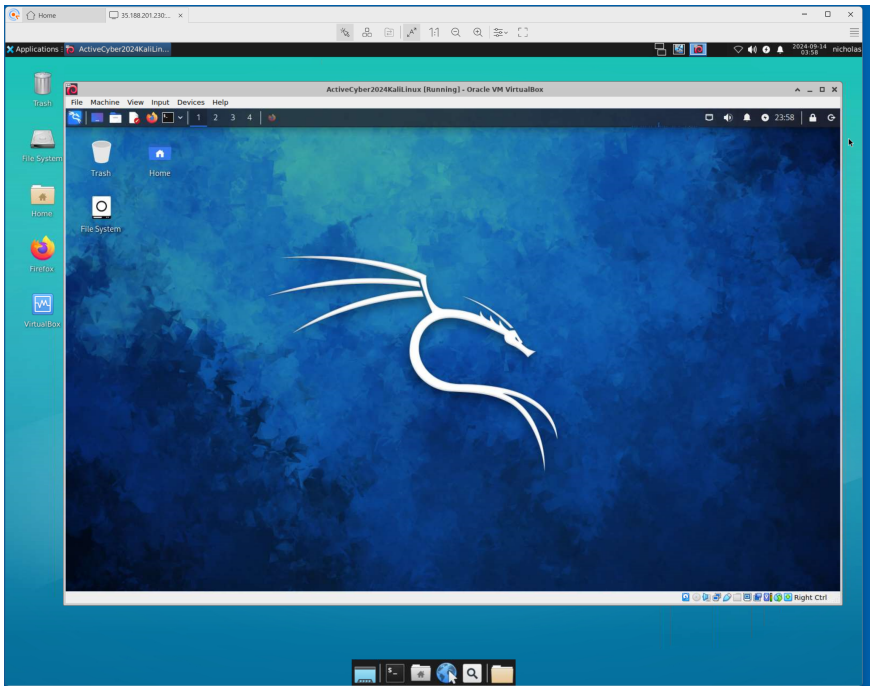


Fig. 1.35: Running Kali as a guest on the Ubuntu instance

- Keep the CPU and memory configuration minimal, e.g. one CPU and four GBs of RAM. More can be added later, once the new VirtualBox VM is stable
- Network should be NAT, cable connected
- Install Guest Additions as desired
- Do whatever you need to do...take snapshots as desired
- Shut VirtualBox down cleanly
- Close the VNC client session
- If desired, run `vncserver -kill :1` to stop that server
- **MAKE SURE** to shutdown the VM instance on the Instances page - or it will cost more money!

1.11 Conclusion and Lessons Learned

In this work we have tried to summarize the tools and techniques that have become part of malware analysis over the last several years. We have made

some observations of a more general nature during that same time:

- Malware analysis is a field where details matter. But when it comes to instruction sets, system calls, and vulnerabilities, it's much easier to look things up as needed rather than to try to remember it all.
- The tools we use, such as Ghidra and x64dbg, are powerful and complicated. They can be intimidating for the beginner, and the cure for that is experience.
- Since the publication of PMA, relevant blogs and social media are among the best ways to keep up with the latest and greatest developments and trends.

Given this chapter's place in this volume, we obviously appreciate the importance of machine learning, now and in the future, in the field of malware analysis. The anti-virus vendors have long been making use of machine learning when applied to malware detection. Analyst-assistant tools will reshape the traditional reverse engineering process and only continue to improve. The malware analyst's time is valuable, and tools that use ML to assist in triage, classification, or in-depth analysis will continue to gain importance.

Bibliography

Eagle, C. and Nance, K. (2020). *The Ghidra Book: The Definitive Guide* (no starch press).

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis* (no starch press).