Using Memory Forensics to Analyze Programming Language Runtimes

Modhuparna Manna

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Information Security Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/5737

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
USING MEMORY FORENSICS TO ANALYZE PROGRAMMING LANGUAGE RUNTIMES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Division of Computer Science and Engineering

by

Modhuparna Manna

B.Tech., Heritage Institute of Technology, India 2014
May 2022
This thesis is dedicated to my mom and dad.
Acknowledgments

I am extremely indebted to my advisor, Professor Golden Richard for guiding me through all the ups and downs of my Ph.D. life. Golden helped me lay my foundations in cybersecurity and supported me right from the beginning when I was completely new to this field. I gained a lot from him, not only in terms of knowledge, but also in the form of leadership, freedom, and support. He was very generous with his ideas and showed me how to think like a researcher. Golden’s optimism and enthusiasm motivated me during the times when I was not making good progress in my research. Working with him is clearly the best part of my Ph.D. research life.

I will like to thank my co-advisor Andrew Case for guiding me in my research. Andrew’s research approach is very unique and I gained a lot in terms of problem-solving techniques from him. I will also like to thank Aisha Ali-Gombe for her constant inspiration during my research.

I would take this opportunity to thank my Committee Members, Professor Nash, Professor Carver, and Professor Zhang for their valuable feedback. Special thanks to Professor Karki for his immense support. I would also like to thank Professor Kundu and Professor Busch for their guidance. I thank all my professors in LSU Computer Science Department, LSU Business School, and LSU Department of Education who have shared their knowledge with me. I will also like to thank Professor Baggili and the entire team at the University of New Haven for supporting me during my final semester.

My colleagues and my lab mates constitute a big chunk of my research life. It’s a pleasure to be a part of the Applied Cybersecurity Lab at the Center for Computation and Technology (CCT) at LSU. I want to thank all my lab mates for supporting me
throughout my research. I would also like to thank Elsa Hahne for motivating me and providing feedback on key projects.

I thank Professor Partha Basuchowdhury as I would not have reached here without his guidance. I also thank Professor Nirman Ganguly and Professor Pratyusa Dash for believing in me from the beginning of my Ph.D. life. I thank all my professors at Heritage Institute of Technology, Kolkata, and my teachers from Brightlands and Delhi Public School, R.K. Puram. I would also like to thank all my colleagues at Capgemini. Special thanks to my mentors at Dutt and Verma Coaching Institute, Dehradun for their inspiration.

I would like to extend my heartfelt thanks to all my friends who supported me during the tough times of my life. Although the list is very long, I thank each one of my friends who have prayed for my success. Finally, I would like to thank my family. I immensely thank my grandmother for shaping me into the person I am. I do not have enough words to thank my parents for the sacrifices they made for me. I am so thankful to the Almighty for the people in my life who have influenced me and nurtured me, and I hope to pay it forward.
# Table of Contents

Acknowledgments .......................................................... iv  
List of Tables ............................................................... viii  
List of Figures ............................................................. ix  
Abstract ........................................................................... xi  

Chapter 1. Introduction ...................................................... 1  
  1.1. Memory Forensics .................................................... 1  
  1.2. Memory Acquisition ................................................ 2  
  1.3. Memory Forensic Frameworks .................................... 3  
  1.4. Operating Systems Overview .................................... 4  
  1.5. Motivation .............................................................. 4  
  1.6. Thesis Contributions ............................................... 5  
  1.7. Thesis Outline ....................................................... 6  

Chapter 2. Related Works .................................................. 7  
  2.1. Memory Forensics .................................................... 7  
  2.2. Page Queues .......................................................... 8  
  2.3. Malware Memory Analysis ....................................... 9  
  2.4. Userland Runtime Analysis ..................................... 10  

Chapter 3. macOS Page Queues ........................................... 15  
  3.1. Data Structures Used .............................................. 15  
  3.2. \texttt{vm\_page\_lookup} and \texttt{vm\_page\_buckets} ........... 18  
  3.3. Memory Queues and Queue States ......................... 21  
  3.4. Memory Analysis of Page Queues ............................ 24  
  3.5. Testing Environment and Data Set ......................... 27  
  3.6. Page State Statistics ............................................ 28  
  3.7. Address Space Evaluation .................................... 29  

Chapter 4. Analysis of the Objective-C and Swift Runtimes ........ 37  
  4.1. Objective-C Memory Analysis .................................. 37  
  4.2. Swift Memory Analysis .......................................... 45  
  4.3. Malware Testbed Creation ..................................... 49  
  4.4. Analysis Capabilities and Evaluation ..................... 52  

Chapter 5. .NET Runtime .................................................. 58  
  5.1. .NET Malware ...................................................... 58  
  5.2. .NET Architecture ............................................... 63  
  5.3. .NET Applications ............................................... 64  
  5.4. .NET Under the Hood ............................................ 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5. .NET Runtime Analysis</td>
<td>70</td>
</tr>
<tr>
<td>5.6. .NET Plugin Implementation</td>
<td>84</td>
</tr>
<tr>
<td>Chapter 6. Conclusions</td>
<td>91</td>
</tr>
<tr>
<td>Appendix: Copyright Information</td>
<td>93</td>
</tr>
<tr>
<td>Bibliography</td>
<td>97</td>
</tr>
<tr>
<td>Vita</td>
<td>106</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Memory Samples Used for Testing ........................................... 28
3.2 Physical Page Queue Distribution ........................................... 29
3.3 Analysis of Stacks and Heaps Pages ....................................... 31
3.4 Number of Processes Recovered ............................................ 34
3.5 Number of Processes Recovered ............................................ 35
3.6 Runtime of mac_procdump in Seconds ..................................... 36
4.1 List of Functions and Methods Alerted by Default ..................... 50
4.2 List of Classes Alerted by Default ......................................... 53
5.1 List of Metadata Tables ....................................................... 68
List of Figures

3.1 vm_fault_internal ........................................... 16
3.2 vm_map_lookup_locked ...................................... 17
3.3 vm_page_lookup ............................................. 18
3.4 vm_map_object ............................................. 25
4.1 Objective-C Object ........................................ 39
4.2 Code Snippet for Finding Class Names ..................... 40
4.3 class_rw_t Data Structure .................................. 41
4.4 class_ro_t Data Structure .................................. 41
4.5 Code Snippet for Finding Instance Variables .............. 42
4.6 ivar_t Data Structure ....................................... 44
4.7 method_t Data Structure .................................... 45
4.8 Code Snippet for Finding Methods ......................... 45
4.9 mac_analyze_classes Detecting Realtime-Spy .............. 54
4.10 mac_analyze_variables Detecting Ventir .................. 55
4.11 mac_analyze_code Detecting Mac Loader .................. 56
5.1 Covenant Dashboard ....................................... 60
5.2 Screenshot Taken by Covenant ............................. 60
5.3 Keylogging Done by Covenant ............................... 61
5.4 Manifest File ............................................... 67
5.5 Module Data Structure ..................................... 69
5.6 Source Code for Example Program ......................... 73
5.7 Code Output Snippet ....................................... 76
5.8  Method Output Snippet

5.9  Field Output Snippet

5.10 Output of dotnet_assemblies Plugin

5.11 Output of dotnet_classes Plugin

5.12 Output of dotnet_fields Plugin

5.13 Output of dotnet_remote_server Plugin

5.14 Source Code for Memory-only Remote Server

5.15 Source Code for Memory-only Loader

5.16 Source Code for Memory-only Malware Output
Abstract

The continued increase in the use of computer systems in recent times has led to a significant rise in the capabilities of malware and attacker tool-kits that target different operating systems and their users. Over the last several years, cybersecurity threat reports have documented numerous instances of users that were targeted by governments, intelligence agencies, and criminal groups, and the result was that the victims ended up having highly sophisticated malware installed on their systems. Unfortunately, the rise of these threats has not been met with equal research and development of defensive mechanisms that can detect and analyze such malware. Though newer techniques such as memory forensics have been incorporated in digital investigations, there is still a huge gap in automated analysis for such frameworks. Consequently, inexperienced investigators have been left with little chance of detecting the malware’s presence and even for experienced investigators, detection is still difficult in many circumstances and requires significant manual investigation for a chance at success. This thesis documents our research efforts to close this gap through the development of novel memory forensic capabilities aimed at detecting advanced, real-world malware that targets popular operating systems such as macOS and Windows. This research is driven through analysis of numerous malware samples that were used as part of espionage and criminal attack campaigns and the research objective is to automate the detection of such malware through memory forensic techniques. The research includes the study of kernel and runtime source code of macOS and Windows operating systems to analyze the various components used to store information about application malware. The end results are new memory forensics techniques that can be leveraged by investigators of all skill levels to detect userland malware in an automated, scalable,
and flexible manner. These techniques have been implemented and tested within the open source Volatility memory forensics framework.
Chapter 1. Introduction

This chapter provides a brief overview of memory forensics, which is the primary defensive strategy considered in this thesis. We discuss the details of memory acquisition techniques and memory analysis frameworks that are used in this research. Userland runtimes of different operating systems are also described, as they play an important role in the research. Finally, we provide the motivation, contributions, and outline for our thesis.

1.1. Memory Forensics

Memory forensics is the process of examination of volatile memory (RAM) for artifacts related to a digital investigation. Memory forensics has become mainstream in recent years because it allows recovery of a wide variety of artifacts that are never written to the file system and are therefore not available when performing traditional file system forensics.

Memory forensics mainly relies on finding information from the volatile data stored in the physical memory of a digital device. Volatile data is temporarily stored in the physical memory while the computer is running and gets lost as soon as the computer is powered off. Earlier, during a digital investigation, the computer system was usually shut down, and only its hard drive was analyzed. In recent days, digital investigators are advised to take a memory dump of the system before shutting it down before removing its hard drive. Much important information related to clipboard data, browsing history, chat messages, open network connections, the presence of malware, and so on can be retrieved from a memory dump which may not be able to be retrieved through traditional filesystem forensics, where only the data written to files are analyzed.
Today, many new trends and techniques have come up which may only be solved through memory forensics. Currently, there is a rise in fileless and memory-only malware [1] [2]. Fileless malware does not create any new files but modifies pre-existing files and system configuration (e.g., the Windows registry), and is therefore very difficult to detect. Furthermore, memory-only malware resides only in RAM, which renders most traditional digital forensics techniques ineffective. Memory forensics is probably the only solution to detect and analyze such malware. Following proper memory forensics procedures is therefore crucial in order to analyze and reveal information from such malware.

1.2. Memory Acquisition

The first step of performing memory analysis is to obtain a memory dump for the digital device. A memory dump, also known as memory snapshot, is the capture of the current state of the system’s memory. The process of acquiring a memory snapshot is known as memory acquisition. Ideally, a special software that dumps the physical memory of a device is required to perform memory acquisition. We have several acquisition tools depending on the operating system, e.g., MAGNET RAM, MDD, Process Hacker, Winen, FTK, WinPmem, MANDIANT Memoryze, WindowsSCOPE are some of tools used for Windows memory acquisition [3]. For Linux we have tools like LiME, Linux Memory Grabber, while for Mac we may use Goldfish, Mac Memory Reader or OSXPMem for the purpose of acquiring memory. [3]. In our research work, we use macOS and Windows virtual machines and take the RAM dump using the memory snapshot feature provided by VMWare Fusion.

Memory forensics involve the reproduction of the data structures and algorithms of
an operating system, and does not depend on the live system and its API to determine the
system state. Therefore, after a memory dump has been obtained, the snapshot is taken
offline and investigated by forensic experts. To analyze memory samples, an investigator
can use one of several available memory analysis frameworks. In the next section, we dis-
cuss some of these frameworks.

1.3. Memory Forensic Frameworks

With the rise of memory forensics, several memory analysis frameworks are now
available which are responsible for parsing and presenting the acquired raw data in a
meaningful way. The success of a memory forensics investigation relies heavily on the
sophistication and ease of use of the memory forensics tools provided by such frameworks.
By leveraging modern memory analysis frameworks, investigators can deeply explore
the address space of the kernel and each process as well as historical data left behind in
de-allocated pages. Current malware often uses highly advanced camouflaging techniques,
including the use of memory-only payloads and heavily obfuscated and encrypted code
and data. In recent times, malicious insiders have resorted to the use of encrypted file
system and network traffic. All these reasons combined necessitate memory forensic tools
that can parse a wide range of subsystems in both kernel and userland memory to fully
reconstruct system state.

Volatility [12] is one such widely used memory forensics framework, which is con-
sidered an industry standard tool in the field of incident response and malware analysis.
Written primarily in Python, Volatility has plugins to analyze Windows, Linux, and ma-
cOS memory dumps. In this thesis, we document our efforts to create new memory foren-
sic tools for the Volatility framework and expand the capabilities of the existing Volatility plugins.

1.4. Operating Systems Overview

All the modern operating systems such as macOS, Linux, and Windows primarily consist of two parts: a non-privileged part for user programs known as the userland, and a privileged part for the low level programs that interact with the hardware, known as the kernel. The kernel may be considered as the inner core of the operating system and mainly consists of software that is crucial for the operating system to function. The kernel typically executes with requires higher security measures and has a higher privilege level. The userland, on the other hand, interacts with the user and usually has fewer security measures and lower privilege levels. If malware authors gain kernel level access, they can control the entire operating system. Although the kernel is usually protected from user space, the userland may still use provided APIs, for example Core Audio and Core Video APIs, QuickTime, etc. which may be abused by malware to record microphone audio, access web camera streams, capture keystrokes, and covertly gain screenshots of sensitive data. In most cases, if suitable vulnerabilities are found in the userland runtime, it becomes an easy task for malware authors to create user level malware. Several userland runtime environments that have been exploited by malware authors include the Android runtime, Windows subsystem GUI runtime, and macOS runtime.

1.5. Motivation

We noticed that in the past decade, research was focused on the data structures of the kernel. This focus was driven by the need to detect kernel level malware as well
as the fact that the data structures describing process data, networking artifacts, file system information, and many other common artifacts are stored within the kernel’s address space. While the results of past research have led to significant kernel analysis capabilities, there is a severe lack of subsystem and runtime analysis capabilities in userland. Given the trend of all major operating system vendors towards strict code signing requirements for 3rd-party kernel drivers, there have been significant developments in userland malware, as malware authors shift to an “easier” target. Unfortunately, defensive research has not kept up with this trend, as existing memory analysis techniques generally only provide broad extraction of userland data, such as extracting an entire process’ address space or specific regions of a process. While these capabilities are certainly useful, they do not provide the structured analysis of activity that is needed to fully analyze running applications and code within process memory. We therefore tried to bridge this research gap in two most widely used userland subsystems: the Objective-C and Swift runtimes of macOS, and the .NET runtime of Windows. To advance the state of the art and to provide generic memory analysis capabilities against these runtimes, we examined a wide variety of runtime data structures and related APIs.

1.6. Thesis Contributions

In this research, we also developed Volatility plugins that automate the extraction of the structured runtime data for macOS and .NET runtime analysis. We also deeply analyzed a number of macOS and C# malware samples that abuse these runtimes. These plugins are configurable and support operations such as searching for known-bad classes and data, extracting all data matching a given pattern, and recognizing code patterns that
are often abused for malicious purposes. The inclusion of these capabilities into memory forensics workflows will allow even novice investigators to automatically detect sophisticated real-world malware.

During our research on the macOS userland runtimes, we realized that a huge number of memory pages in the macOS were being ignored by the existing memory forensic tools. Since the accuracy of our results from the userland macOS runtime research depended on how well the page handling mechanisms were modeled, we decided to first study the macOS page queues subsystem. We wrote Volatility plugins to reconstruct data from these queues allowing a significant number of memory pages to be analyzed that are currently ignored by memory forensics tools. This is a supporting but also very important contribution.

1.7. Thesis Outline

The remainder of the dissertation is structured as follows. In Chapter 2, we consider closely related work. Our efforts to analyze the macOS page queues are documented in Chapter 3. The analysis of the Objective-C and Swift runtimes of the macOS subsystem has been described in Chapter 4. In Chapter 5, we discuss the .NET runtime of the Windows subsystem. Finally, we give our conclusions and sketch some future work in Chapter 6.
Chapter 2. Related Works

Although our research on finding the page queues and userland runtime analysis for the Objective-C, Swift, and the .NET runtime is largely based on previously unexplored methods, we would like to mention some of the past and ongoing research work related to memory forensics, page queues, Windows runtimes, and macOS runtimes.

2.1. Memory Forensics

In the last fifteen years, there has been a substantial number of publications in the field of memory forensics. The 2005 DFRWS challenge [4] is widely considered to have been the catalyst for the initial research in this area. This challenge required development of tools that could parse a Windows memory sample in a structured manner to recover key artifacts. Such tools did not exist at all at the time and the challenge led to several novel research efforts [5, 6]. In the following years, many new capabilities for Windows analysis were added, such as analysis of VAD trees [7], the registry in memory [8], and cached files [9]. Since then, there have been dozens of new capabilities added that are all now a part of standard Windows systems investigations. Structured analysis of Linux systems largely began with efforts to solve the 2008 DFRWS challenge [10] and has seen considerable growth since then. Mainstream analysis of macOS samples began in 2013 [11] with the release of Volatility 2.3. Volatility [12] is the mostly wide used framework in the field and provides significant analysis capabilities for Windows, Linux, and macOS memory samples. The Art of Memory Forensics [13], a book written by four of the core Volatility developers - Michael Ligh, Andrew Case, Jamie Levy, and AAron Walters, documents the modern memory forensic techniques related to macOS, Linux, and Windows...
operating systems.

2.2. Page Queues

There have been many research efforts to decipher the operating system-specific encoding of invalid pages so that they can be located and incorporated into analysis. In 2007, several papers were published that addressed a variety of these sources for Windows, including the recovery of invalid pages from paging files, transition pages, and prototype pages [6] [14] [15]. Paging files, also referred to as swap files, are used to store excess pages of memory that have not been recently or frequently accessed to free physical memory pages for more actively used data. Publications by Stimson in 2008 [16] and Iqbal in 2009 [17] deeply explored the incorporation of paging files into memory analysis. In a blog post describing the incorporation of paging files into the Rekall memory forensics framework (Google, 2016), Cohen describes how invalid pages that are stored within arbitrary, non-paging files in the filesystem can be recovered during address translation [19]. This is very useful, as the on-demand loading strategy of operating systems only loads in pages from files on disk into memory when they are accessed (“on demand”) or when they are part of a read-ahead set for applications that access many portions of a file. To evade Linux malware that maps regions with the PROT_NONE protection flag, Volatility specifically checks for pages marked as not present but that have the 8th bit in the page table entry set, which signifies a “global” page. The full explanation is given in a blog post [20], and the end result is that these pages are no longer hidden from Volatility even though they are not marked as present within the hardware state. The addition of compressed paging stores has also driven considerable interest in the address translation mechanisms of
operating systems. These stores are used to hold compressed pages inside of memory to avoid the performance penalties of writing them out to disk. The power of modern CPUs, combined with the performance of modern compression algorithms, means that a considerable number of pages can be efficiently compressed and decompressed from one physical page inside the store. Case and Richard analyzed these stores on Linux and macOS in an award-winning 2014 DFRWS paper [21]. Microsoft did not add support for compressed stores until Windows 10. In 2019, FireEye released a three-part blogpost [22] [23] [24] on Windows 10 compressed stores. The author of the blogpost, Omar Sardar, along with Dimiter Andonov, presented their work on “Finding Evil in Windows 10 Compressed Memory” at BlackHat 2019 [25]. This research demonstrated how to retrieve a compressed page using the structures and algorithms in Windows 10 and that forensic artifacts (including reflectively loaded malware) could remain undiscovered if their techniques were not integrated in Volatility and Rekall memory forensic tools.

2.3. Malware Memory Analysis

The Volatility framework provides a suite of plugins to analyze the GUI subsystem of Windows systems such as wndscan, windows, wintree, messagehooks, etc. [13]. This allows detection of malware that leverages related APIs, such as for keystroke logging, monitoring of USB device insertion, and code injection. It also allows reconstruction of user activity through mapping per-session data, reconstruction of screen views (screenshots), and the recovery of values from input forms and menus. In 2013, Carbone in his malware memory analysis series used these plugins to analyze various memory images provided by Prolaco and SpyEye [26], and several malware samples such as the Stuxnet worm [27] and
Tigger Trojan horse [28]. In 2014, Carbone examined Linux memory malware such as the Jynx2 Rootkit using the Linux plugins provided by Volatility. In 2014, Wardle [29] discussed various macOS malware samples. Although this work did not involve any memory forensic efforts, the persistence mechanisms of various macOS malware were looked at thoroughly and the paper also helped debunk the common misconception that “Macs don’t get infected with malware”.

2.4. Userland Runtime Analysis

In the past few years, there has been some research related to analysis of various runtime environments. We mention some of those efforts related to our research.

2.4.1. Android Runtime

There have been several efforts to analyze the data structures and effects of the Android application runtime. In [31], the Android runtime was analyzed to recover the classes loaded by an application as well as the metadata of those classes (instance variables, methods, etc.). In [30], the Android runtime was studied even further and algorithms were presented that allowed for location of class instances and decoding of variable values. In [32], a system was presented for recovery of heap metadata and objects, which led to recovery of a significant amount of application data.

2.4.2. Objective-C and Swift Runtime

There are several papers and blog posts related to the Objective-C runtime. In 2009, Neil Archibald (popularly known as nemo) published a paper in the Phrack journal which discusses the Objective-C runtime data structures in detail [36]. In this article, the author describes how the Objective-C classes and methods are represented in memory. In
2016, Archibald published another paper which elaborates on how tagged pointers work in Objective-C [37]. The data structures used in our research are the same as that described by nemo, however, the purpose of our research is completely different from the paper published in the Phrack Journal, as we utilize these data structures to write plugins for Volatility and to provide memory forensics capabilities. Andrew F. Hay’s thesis published in 2012 broadly talks about macOS memory forensics [38]. The author explains the memory acquisition and analysis techniques of Mac memory images with Volatility. The author provides some test cases to demonstrate the forensic techniques. The book, *Hacking and Securing iOS Applications*, by Jonathan Zdziarski, gives an insight into the Objective-C runtime as well [39]. The Objective-C runtime source code, which is now open-source [34], provides detailed documentation on Mac memory management and how the Objective-C runtime works [35]. In 2016, Case and Richard published a paper that detailed Objective-C memory analysis [41]. The purpose of this effort was to develop Volatility plugins that would detect the Crisis malware [42] within memory samples. To accomplish this, select portions of the Objective-C runtime were analyzed to determine the address of class method handlers. This allowed detection of method swizzling as well as malware that registered keyloggers through the use of the *addGlobalMonitorForEventsMatchingMask* or *addLocalMonitorForEventsMatchingMask* functions. Though novel, this research is now largely outdated as it only supported analysis on macOS versions up to 10.9, which stopped being supported in 2016. Due to the change in macOS versions, the algorithm presented in this research for enumerating the loaded classes inside a process is no longer valid. The detection algorithms utilized by the plugins are also very limited in their scope, and leave much of the runtime information unexplored. For example, these algorithms do
not fully enumerate all methods associated with an object instance and class methods are only examined on two criteria. The first is based on the method implementation’s memory region, and it is examined for being mapped by a file in a suspicious directory or being in a non-file backed region. The second criteria is if the region is hosting a handler for one of the previously mentioned event monitors. The use of these event monitors is not typical in current macOS malware and the referenced work is unable to detect more popular modern methods. This research also did not provide algorithms for decoding variable values, instead concentrating solely on enumerating names. This left behind strings and other data that may be useful when analyzing malware and investigating malicious insiders.

There are also some recent blog posts on Objective-C and Swift data structures. Ash has a series of Friday Q&A discussion articles where he primarily discusses the Objective-C runtime[40]. These articles compiled in three book volumes discuss how the Objective-C and Swift data structures are arranged in memory. Ash also talks about method dispatch techniques and tagged pointers in the Objective-C runtime.

2.4.3. .NET Runtime

The bulk of information about the .NET architecture and runtime has been articulated in Microsoft documentation [86]. There are also several books, papers and blog posts related to the .NET runtime. Some of these have been mentioned below.

In 2000, Richter described the details of the garbage collection process in .NET in his two-part article [88]. Richter explains how resources are allocated and managed, then gives a detailed step-by-step description of how the garbage collection algorithm works. He also discussed ways to clean up properly after a resource’s memory has been freed. Richter

There are several other books that give a description of .NET Framework including the CLR internals, the .NET architecture, .NET programming, and other features of .NET. One of the most widely used books is *The Book of the Runtime* [92] which gives a vivid description of the .NET internals. The .NET Framework Essentials [93] by Thai and Lam is another book that provides an overview of the .NET Core Language Runtime, the .NET programming core features and languages, and the .NET architecture. Pro .NET Performance [94] describes ways to measure performance in .NET applications and discusses how to make these applications faster. The author Sasha Goldshtein has also presented his work in the .NET DeveloperDays conference and his work is now available in the form of training materials [98].

Some of the most important resources for .NET runtime can be found online in the form of blog posts. One of the seminal blog posts is on CLR runtime internals by Hanu Kommalapati and Tom Christian [99]. This article is a part of the Microsoft documentation and discusses the .NET Framework 1.1. Although there has been several changes to the source code during the version changes from .NET Framework 1.1 to .NET Framework 4.0., this documentation can still be considered as a valuable resource. Simon Cooper has a four-part series of articles on the anatomy of .NET assembly [100][101][102][103]. In
these articles, Cooper gives a detailed description of the .NET metadata tables including the TypeDef, methodDef, and FieldDef tables. The different heaps including the metadata heap, String Heap, User String Heap, GUID Heap, and Blob Heap are discussed in detail. Cooper describes the techniques to identify these streams and tables using the hex dump of an executable. A comprehensive collection of research papers by different authors on various topics related to the .NET language, runtime and compiler source code has been assembled by Matt Warren [104]. Some of the research papers in this collection has been described here. Sedgewick [105] describes a new variant of left leaning red-black trees that uses one-fourth less code for common implementations while meeting the needs of the .NET design goals. Herlihy [106] presented a new class of resizable sequential and concurrent hash map algorithms known as the hopscotch algorithms to be adapted by the .NET coreclr source. Kennedy [107] in their research paper discussed the design and implementation of generics for the .NET Common Language Runtime. Generics are specific types that are to-be-specified-later and are instantiated when needed. Pizlo et al. in their papers [108][109] discuss the garbage collection technique which is considered as one of the most attractive features of .NET.

A comprehensive list of .Net Performance Books, Courses, Trainings, Conference Talks, Blogs and Most Inspiring Open Source Contributors can be found in this github resource compiled by Adam Sitnik [110]. A curated list of .NET blog posts has been recommended by Thomas Ardal[111]. This list includes blog posts by authors like Scott Hanselman, Michael Crump, Troy Hunt, Derek Comartin, Damien, Mads Kristensen, Iris Clas- son, Steve Sanderson, Ben Foster, and Mark Rendle.
Chapter 3. macOS Page Queues

Data is stored and retrieved by operating systems in fixed sized blocks known as pages. To enable memory analysis, memory forensics frameworks such as Volatility need to translate the unordered set of physical pages in a memory sample into the structured and ordered virtual address spaces in which the operating system and all running applications operate. The process of associating virtual address spaces with their corresponding physical pages is known as address translation. Each hardware architecture defines the mechanism by which the hardware state describes whether a page in virtual memory is present in physical memory.

Recovery of pages that are present in physical memory is trivial as the hardware state encodes the offset in the memory sample. For pages that are not marked as present in physical memory (i.e., "invalid" pages), the operating system defines the hardware state to encode the actual location of the page, such as in a paging file, a file on disk, in a compressed data store, or elsewhere. The ability to decipher these invalid states and recover the associated pages is a critical component of a memory forensics framework. If an investigator relies solely on pages marked as present in physical memory, a substantial number of pages is left behind and negatively impacts results produced during analysis.

3.1. Data Structures Used

In this section, we describe the page queue data structures relevant to our research.

---

15
3.1.1. The page_handler

The *vm_fault_internal* function is defined in (https://github.com/apple/darwin-xnu/blob/main/osfmk/vm/vm_fault.c). The structure of the *vm_fault_internal* is shown in Figure 3.1.

```c
static kern_return_t vm_fault_internal(
    vm_map_t map,
    vm_map_offset_t vaddr,
    vm_prot_t caller_prot,
    boolean_t change_wiring,
    vm_tag_t wire_tag,
    int interruptible,
    pmap_t pmap,
    vm_map_offset_t pmap_addr,
    ppnum_t *physpage_p);
```

Figure 3.1. *vm_fault_internal*

The *vm_fault_internal* function performs the bulk of the operating system’s page fault handling duties. The function *vm_fault_internal* begins by locating the memory map associated with the virtual address that caused the fault. This is done with the help of the *vm_map_lookup_locked*. *vm_map_lookup_locked* is defined in (https://github.com/apple/darwin-xnu/blob/main/osfmk/vm/vm_map.c). The structure of the *vm_map_lookup_locked* is shown in Figure 3.2.

The *vm_map_lookup_locked* returns the _vm_map data structure associated with the faulting address and the values of the *vm_object* and *vm_object_offset* values. The _vm_map data structure in macOS is the equivalent of the VAD structure in Windows

16
Figure 3.2. vm_map_lookup_locked

kern_return_t vm_map_lookup_locked(
    vm_map_t *var_map,
    vm_map_offset_t vaddr,
    vm_prot_t fault_type,
    int object_lock_type,
    vm_map_version_t *out_version,
    vm_object_t *object,
    vm_object_offset_t *offset,
    vm_prot_t *out_prot,
    boolean_t *wired,
    vm_object_fault_info_t fault_info,
    vm_map_t *real_map,
    bool *contended)

It holds metadata about a memory region of a process, such as the starting and ending address, permissions, and mapped file, if any. The _vm_map data structure is defined in (https://github.com/apple/darwin-xnu/blob/main/osfmk/vm/vm_map.c). The _vm_object data structure is defined in (https://github.com/apple/darwin-xnu/blob/main/osfmk/vm/vm_object.h) tracks the physical pages of a particular object and defines how to find the associated pages. For pages in physical memory, there is a list of pointers to _vm_page data structures (defined in https://github.com/apple/darwin-xnu/blob/main/osfmk/vm/vm_page.h), one for each page. For non-resident pages, the object tracks the pager responsible for gathering the pages. Pagers exist for data stored in locations such as the compressed store or files on disk. The _vm_object_offset is an integer value that holds a particular page’s offset into its host object.
3.2. vm_page_lookup and vm_page_buckets

Once the map, object, and offset are found, the page fault handler then uses the \texttt{vm_page_lookup} function. The structure of the \texttt{vm_page_lookup} is as shown in Figure 3.3.

\begin{verbatim}
vm_page_t vm_page_lookup (vm_object_t object,
vm_object_offset_t offset)
\end{verbatim}

Figure 3.3. \texttt{vm_page_lookup}

The page fault handler passes the object and offset to the \texttt{vm_page_lookup} to find the associated \texttt{vm_page}. The \texttt{vm_page_lookup} first checks the per-object cache to locate the object/offset pair, and if that fails it then relies on the use of \texttt{vm_page_buckets}. Although we get some information about the object from the look-aside cache, for a complete recovery of data associated with an object, we must investigate the \texttt{vm_page_buckets}.

The \texttt{vm_page_buckets} is defined in (https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/osfmk/vm/vm_resident.c).

The \texttt{vm_page_buckets} is a hash table, defined in source file \texttt{vm_resident.c}, that maps object/offset pairs to their corresponding \texttt{vm_page} structures. This hash table has an entry for each physical page currently tracked by the operating system, and it plays a similar role to that of the page frame number (PFN) database of Windows [43] and the \texttt{mem_map} data structure of Linux [44].

Through our analysis of the page fault handler, we discovered that by analyzing \texttt{vm_page_buckets}, a memory forensics framework can derive the physical offset of any vir-
tual address, even when the virtual address is marked as not present. The only exceptions to this are when the virtual address truly has no backing physical page or when the page is in an error state.

3.2.1. vm_page and page states

In macOS, each physical page is associated with a vm_page structure, which tracks its associated object and offset, its current state, its PFN, and a few other pieces of metadata. The physical offset of a page can be calculated by multiplying its PFN by 4096 (size of a hardware memory page on modern Intel processors). Before accessing a page, its state must first be examined. Scrutiny of vm_fault_internal, as well as the vm_fault_page helper function, illustrated that several problematic page states are skipped when the operating system attempts to recover a non-present page. The page states that are skipped are unusual, error, busy, absent, fictitious, and restart. These states and others are enumerated in the kernel source. Here is a list of such states:

- busy: page is in transit
- wanted: something is waiting for page
- tabled: page is in VP table
- hashed: page is in vm_page_buckets[] + the bucket lock
- fictitious: physical page doesn’t exist
- clustered: page is not the faulted page
• pmapped: page has at some time been entered into a pmap (the memory map for a process)

• xpmapped: page has been entered with execute permission

• wpmapped: page has been entered at some point into a pmap for write

• free_when_done: page is to be freed once cleaning is completed

• absent: data has been requested, but is not yet available

• error: data manager was unable to provide data due to error

• dirty: page must be cleaned

• cleaning: page clean has begun

• precious: page is precious; data must be returned even if clean

• overwriting: request to unlock has been made without having data

• restart: page was pushed higher in shadow chain by copy_call-related pager

• unusual: page is absent, error, restart or page locked

• validated: page was checked

• tainted: page is tainted

• nx: page is nx
• reusable: page can be reused

• written_by_kernel: page was written by kernel

For pages that are not in one or more of these problematic states, the particular memory queue that the page is on is then checked for suitability of recovery.

3.3. Memory Queues and Queue States

The queue that a page is currently on is tracked inside of its `vm_page` structure.

The queues used by the macOS kernel to track pages are:

• Wired: pages that cannot be swapped. Similar to the non-paged pool of Windows.

• Compressed: pages that are being used by the compressed swap subsystem. They must be decompressed before use.

• Free: several queues hold free pages.

• Throttled: previously busy pages eligible to be made active.

• Page Out: pages prepared to be paged out.

• Active: pages active in memory.

• Inactive External: queue of anonymous (non-file backed) pages.

• Inactive Internal: queue of file-backed pages.

• Inactive Cleaned: pages that were previously dirty but have since been written to a
backing store.

- Speculative: read-ahead pages from objects recently accessed.

- Secluded: memory reserved for the Camera application.

A set of macros and constants is used to associate pages with these queues. Below are some of the constants that are used:

- `VM_PAGE_NOT_ON_Q` (constant 0): page is not present on any queue, nor is it wired, mainly a transient state

- `VM_PAGE_IS_WIRED` (constant 1): page is currently wired

- `VM_PAGE_USED_BY_COMPRESSOR` (constant 2): page is in use by the compressor to hold compressed data

- `VM_PAGE_ON_FREE_Q` (constant 3): page is on the main free queue

- `VM_PAGE_ON_FREE_LOCAL_Q` (constant 4): page is on one of the per-CPU free queues

- `VM_PAGE_ON_FREE_LOPAGE_Q` (constant 5): page is on the lopage pool free list

- `VM_PAGE_ON_THROTTLED_Q` (constant 6): page is on the throttled queue, anonymous pages are stashed here when not paging

- `VM_PAGE_ON_PAGEOUT_Q` (constant 7): page is on one of the page out queues
(internal/external) awaiting processing

- **VM_PAGE_ON_SPECULATIVE_Q** (constant 8): page is on one of the speculative queues

- **VM_PAGE_ON_ACTIVE_LOCAL_Q** (constant 9): page has recently been created and is being held in one of the per-CPU local queues

- **VM_PAGE_ON_ACTIVE_Q** (constant 10): page is in global active queue

- **VM_PAGE_ON_INACTIVE_INTERNAL_Q** (constant 11): page is on the inactive internal queue i.e. anonymous queue

- **VM_PAGE_ON_INACTIVE_EXTERNAL_Q** (constant 12): page in on the inactive external queue i.e. file backed queue

- **VM_PAGE_ON_INACTIVE_CLEANED_Q** (constant 13): page has been cleaned to a backing file and is ready to be stolen

- **VM_PAGE_ON_SECLUDED_Q** (constant 14): page is on secluded queue

By comparing a page’s state to its current queue, the kernel can decide if it is eligible to be retrieved during a page fault. Along with the previously mentioned macOS kernel functions, we also studied the `memory_object_lock_page` (defined in https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/osfmk/vm/memory_object.c) and `vm_pageout_scan` (defined in https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/osfmk/vm/vm_pageout...
3.4. Memory Analysis of Page Queues

After studying the kernel source related to the page fault handler and the paging subsystem, we were able to develop a memory forensics algorithm that successfully recovers pages that are still in memory but marked as being non-present in the hardware page state. This algorithm involves the following steps. First, for a given virtual address in an address space, it must be determined if there is a corresponding `vm.page` structure. Second, if the corresponding structure exists, it must be checked for eligibility to be recovered based on its state and current memory queue. Finally, its physical offset must be extracted so that the corresponding data can be used during analysis.

3.4.1. Finding `vm.page` structure

We achieved this step using two different methods.

Method I

This method was mostly implemented to aid our understanding of the page queues and to provide statistical analysis. In this method, we enumerate every page tracked by the `vm.page_buckets` and report each page’s metadata. We follow the following steps. First, we create an array of `vm.page_bucket_t` typed elements and with an array size determined by the `vm.page_bucket_count` global variable. Each bucket of the array references a list of `vm.page` instances. Next, for each instance eligible for recovery, the plugin prints the `vm.page` address, its Page Frame Number, the address of the page’s corresponding object, its offset in the object, and which queue the page is stored in.
Method II

In this method, we replicate the macOS page fault handler’s ability to locate \textit{vm\_page} instances for specific non-present virtual addresses in specific contexts. This method includes the following steps. First, we locate the map data structure \textit{vm\_map\_entry} instance corresponding to the given virtual address with the help of the Volatility \textit{get\_proc\_maps} function. Then, we discover the object corresponding to the map and the map’s offset into that object.

We mentioned that in the second step of this method, we need to extract the object and its offset. Extracting the offset is straightforward as it is simply an integer field in the map data structure. However, extracting the object requires the following steps. First, we access the object structure. The pointer to this structure inside of the map is embedded inside a union, as a map can represent either an actual object or a \textit{submap}. The object structure is shown in Figure 3.4.

```c
typedef union vm\_map\_object {
    vm\_object\_t vmo\_object;
    vm\_map\_t vmo\_submap;
} vm\_map\_object\_t;
```

Figure 3.4. \textit{vm\_map object}

We then check for maps that are sub-maps by examining the \textit{is\_sub\_map} member and bailing out if it is set. For actual objects, after getting the initial object structure, it must then be checked to see if it has a shadow object [45]. Shadow objects are created when an object is copied, such as during a process fork. Changes (writes) to the object
are propagated to the shadow object by other objects that shadow it. To use the correct object structure, our code uses the shadow object of map objects when they are present.

After finding the map corresponding to a given virtual address and extracting the correct offset and object, our plugin is able to calculate an index into the `vm_page_buckets` hash table. This is the same operation performed by the macOS page fault handler and its use of the `vm_page_hash` macro, defined in (https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/osfmk/vm/vm_resident.c). By indexing the table directly, we can quickly narrow our search to just a single bucket list of `vm_page` instances. In our testing, these lists are generally very small, usually less than ten elements. To determine the matching `vm_page` instance, we use the same algorithm as the page fault handler, which matches the extracted object and offset with those embedded in the `vm_page`. Through use of this algorithm, we are able to quickly determine which if any `vm_page` instance references our given virtual address and context.

Once a `vm_page` instance is found, we must then verify that the data at its referenced page frame number is suitable for recovering. The suitability of an instance is first determined by examining its unusual, error, absent, and fictitious state members. Although the macOS handler also skips the busy and restart states, these are states our code can work around. As mentioned previously, if the page is in any of the error states then the corresponding physical page must be ignored. After examining its state, the queue on which the page resides must be examined. Our code considers pages in the wired, throttled, page out, active, inactive internal, inactive external, internal cleaned, speculative, and secluded states to be suitable for recovery. All of these pages that are still in memory, but are marked as non-present in the particular address space being examined. We skip
pages in the compressed state as previous work incorporated recovery of these [46] and furthermore, the next release of Volatility will support transparent RAM decompression. We also skip pages on any the free queues as we do not currently believe that these pages are in a suitable state for recovery based on our reading of the kernel source code.

3.5. Testing Environment and Data Set

To test our algorithms, we generated a corpus of memory samples with diverse operating system version and hardware configurations. In this section we describe the memory samples generated as well as the system used to test our Volatility integration.

3.5.1. Memory Samples

Table 3.1 lists the memory samples generated for testing our integration. Using VMware, we created 2GB and 4GB samples covering every version of macOS from the latest Catalina all the way back to Mavericks, which is the same version coverage provided by Volatility itself. The use of small memory sizes was simply for expedience and to illustrate that pages are present in the queues for a variety of versions of macOS. Obviously the number of pages for the 2GB systems, which were booted and then immediately analyzed, will be small, but pages are still present in the queues. The 64GB and 128GB samples were taken from a MacBook Pro and a Mac Pro. These are the daily use systems of two members of the research team and were used to provide real world data sets.

Memory snapshots of the virtual machines were acquired using VMWare’s built in facilities [47]. Memory snapshots of the physical systems were acquired using Surge Collect Pro [48]. All of the virtual machine systems were acquired within an hour of being booted. The MacBook Pro had an up-time of 41 days when acquired and the Mac Pro 30
days. If systems with 64GB and 128GB had been recently rebooted, the number of pages present in the queues would be much smaller, mirroring the 2GB VM cases more closely. The point is that pages are definitely present in the queues across a wide variety of system configurations and system up-times and therefore it is clearly necessary to introduce measures to ensure this data is acquired.

### Table 3.1. Memory Samples Used for Testing

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Version</th>
<th>Build</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mavericks</td>
<td>10.9.5</td>
<td>13F1712</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>10.10.5</td>
<td>14F1021</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>ElCapitan</td>
<td>10.11.4</td>
<td>15E65</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Sierra</td>
<td>10.12.5</td>
<td>16F73</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>HighSierra</td>
<td>10.13.5</td>
<td>17F77</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Mojave</td>
<td>10.14.5</td>
<td>18F132</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Catalina</td>
<td>10.15.2</td>
<td>19C57</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Mojave</td>
<td>10.14.4</td>
<td>18E226</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>Catalina</td>
<td>10.15.1</td>
<td>19B88</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Catalina</td>
<td>10.15.2</td>
<td>19C57</td>
<td>64</td>
</tr>
</tbody>
</table>

### 3.6. Page State Statistics

Before leveraging `vm_page_buckets` to aid in address translation, we first wanted to understand the state and purpose of pages that are actively tracked by the buckets. We used Method I to locate the `vm_page` structure mentioned earlier to determine the relationship between RAM size and the number of tracked pages, how many pages are in a state suitable for use in address translation, and the distribution of pages across the various queues. Table 3.2 lists the distribution of page states for each sample in our data set. The number of pages in the file cache are separated from the other recoverable pages as their exact purpose is known just from being in that queue. Unfortunately, identification of pages in the file cache is only possible starting with Sierra so the samples from older
versions have the file cached marked as N/A. Pages in the file cache queue always correspond to portions of cached files from disk, regardless of whether any processes are actively mapping the file. The pages that are in a recoverable state, but not in the file cache, are counted in the Other column. This includes pages belonging to anonymous memory, stacks, heaps, and other non-file backed regions. Only a relatively small number of pages were kept in the various free page queues, which constitute the Invalid column count. Analysis of the distribution showed that a significant percentage of physical pages in each sample were tracked in the page buckets and directly recoverable through examination of the corresponding vm_page instance.

Table 3.2. Physical Page Queue Distribution

<table>
<thead>
<tr>
<th>#</th>
<th>File</th>
<th>Other</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/A</td>
<td>229,591</td>
<td>364</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>396,138</td>
<td>1,568</td>
</tr>
<tr>
<td>3</td>
<td>N/A</td>
<td>400,618</td>
<td>1,001</td>
</tr>
<tr>
<td>4</td>
<td>34,674</td>
<td>214,979</td>
<td>693</td>
</tr>
<tr>
<td>5</td>
<td>37,581</td>
<td>446,832</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>67,574</td>
<td>387,421</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>27,874</td>
<td>460,329</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>88,104</td>
<td>686,018</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>361,541</td>
<td>1,976,958</td>
<td>4,386</td>
</tr>
<tr>
<td>10</td>
<td>747,269</td>
<td>14,138,592</td>
<td>103,475</td>
</tr>
</tbody>
</table>

3.7. Address Space Evaluation

After understanding the distribution of pages across physical memory, we then wanted to test and leverage our newly developed address space to examine process memory. We started with examination of process memory regions critical to memory analysis. This was performed to determine precisely how much of previously unavailable critical process data could be recovered through analysis of the queues. Our first experiment was
the examination of the stacks and heaps. In the next experiment, we examined the contents of the process executables.

3.7.1. Recovery of Stack and Heap Data

The process runtime stack holds metadata and data related to function calls. Depending on the compiler, programming language used, and the calling convention, this information can include return values, return addresses, function parameters, and local variables to functions. Analysis of stack data has a long history in memory forensics to uncover call stacks, parameters passed to sensitive APIs, and more. A process’ heaps are what hold the dynamically generated data processed by the application and its associated libraries, frequently including artifacts such as keystrokes, images viewed or downloaded, HTTP requests and responses, copy/paste buffers, and other valuable forensic artifacts.

Due to the forensic value of the contents of stacks and heaps, we wanted to determine how many pages belonging to these regions could be recovered through the use of the queues. To start this analysis, we walked each process’ set of memory mappings and then filtered for only those related to stacks and heaps and only those of 50MB or less. The 50MB restriction is to eliminate issues with data smear. It then attempted to translate the virtual address of each page within these regions and kept a count of how many pages translated to a physical offset and how many did not. This plugin was run twice against every sample in our data set. The first run was with an unaltered version of Volatility, and the second was with a version that had our new address space installed. Table 3.3 documents the result of this experiment. As illustrated, a substantial number of critical pages are made available through analysis of the queues.
Table 3.3. Analysis of Stacks and Heaps Pages

<table>
<thead>
<tr>
<th>#</th>
<th>Total</th>
<th>Valid without AS</th>
<th>Valid with AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>231,508</td>
<td>8,248</td>
<td>11,565</td>
</tr>
<tr>
<td>2</td>
<td>452,008</td>
<td>50,024</td>
<td>54,849</td>
</tr>
<tr>
<td>3</td>
<td>442,254</td>
<td>61,366</td>
<td>66,679</td>
</tr>
<tr>
<td>4</td>
<td>588,739</td>
<td>59,538</td>
<td>76,055</td>
</tr>
<tr>
<td>5</td>
<td>518,504</td>
<td>66,299</td>
<td>68,248</td>
</tr>
<tr>
<td>6</td>
<td>220,230</td>
<td>24,458</td>
<td>25,533</td>
</tr>
<tr>
<td>7</td>
<td>776,664</td>
<td>69,889</td>
<td>79,377</td>
</tr>
<tr>
<td>8</td>
<td>2,999,337</td>
<td>106,712</td>
<td>220,213</td>
</tr>
<tr>
<td>9</td>
<td>11,216,197</td>
<td>49,750</td>
<td>479,436</td>
</tr>
<tr>
<td>10</td>
<td>17,411,435</td>
<td>17,604</td>
<td>383,827</td>
</tr>
</tbody>
</table>

3.7.2. Process Executable Recovery

Our second test involved attempting to recover the process executable for every running process in the data set of memory samples. This is a very common operation during analysis as recovering the executable from process memory provides several extremely beneficial advantages, but there are also several impediments to successful recovery.

Benefits and Impediments

The benefits to process executable recovery include, but are not limited to, the following:

**Defeating Packers**

Upon execution, unpackers will usually de-crypt/de-compress/de-obfuscate strings and code from disk into their plain-text forms in memory. Recovery of these plain text versions saves substantial reverse engineering time and effort [49].

**Recovering Injected Code**

Code injecting malware replaces legitimate code in the address space of running processes with malicious alternatives. These changes to the code are not reflected back to
the file on the file system and require memory forensics to detect. Common examples of such techniques are API hooks [13] and process hollowing [50].

**Recovery of Command and Control Data**

Malware will often dynamically load configuration data from a remote command and control server. These configuration options can include a list of servers to exfiltrate information to, commands to run on the local system, and more. Such data is not available in the executable on the file system, but is available in the address spaces of actively executing malware.

These benefits cannot be fully realized when any of the impediments that hinder the ability to fully recover the executable or to recover it at all are present. The largest issue is that, to fully and properly recover an executable, its metadata must be available. This metadata encodes the layout and size of all of the file’s sections and tells the memory forensics framework how to reconstruct the executable. The majority of this metadata is in the file’s header, which will be mapped into the first page of the executable’s memory region.

Unfortunately, since the header is never or rarely referenced after initial process loading, it is often a target of the swap manager since it will be not have been recently accessed. This will then remove the header’s page from the process’ address space, and with the header gone the memory forensics framework will produce an empty file when attempting to recover the executable.

Even when the header is present, analysis is still hindered in situations where many of the executable’s pages are not present. Particularly in situations where an analyst wants to perform full static analysis of an executable from memory, it is very important to
recover as much original data as possible as most reverse engineering tools are intolerant of corrupted executables.

We note that these issues related to recovering process executables affect analysis of all operating systems and not just macOS.

**Executable Headers and the Queues**

Our goal in testing process executable recovery was two-fold. First, we wanted to test whether incorporation of the queues could allow for recovery of the executable header for more processes than traditional translation provided. Second, we wanted to test how much more data inside of executables we could recover by processing the queues. To recover process executables, we relied on the existing `mac_procdump` Volatility plugin, which examines Mach-O files in memory and uses the metadata to properly reconstruct each section. We then ran the plugin twice as with `mac_walk_stack_heap`, once in an unaltered version of Volatility and once in a version with our address space installed.

The results of this experiment showed that there were 1,893 active processes in total across our data set. Without our address space being active, 749 of these were recovered and 1,144 were written as empty files. This produced 419MB of data. With our address space active, 1,366 executables were recovered and only 527 were written as empty files. This produced 943MB of data in total. Table 3.4 lists the number of processes recovered with and without our address space active for each sample.

As shown in the table, a substantial number of previously unavailable executable headers become available with the use of our new queue-aware address space. This is exemplified in samples 9-10 as they were acquired on long running systems with substan-
Table 3.4. Number of Processes Recovered

<table>
<thead>
<tr>
<th>#</th>
<th>Active Processes</th>
<th>Without AS</th>
<th>By AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>85</td>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>90</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>103</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>116</td>
<td>115</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>56</td>
<td>53</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>116</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>240</td>
<td>127</td>
<td>104</td>
</tr>
<tr>
<td>9</td>
<td>466</td>
<td>12</td>
<td>238</td>
</tr>
<tr>
<td>10</td>
<td>554</td>
<td>4</td>
<td>259</td>
</tr>
</tbody>
</table>

Initially more processes and real-world work loads, all of which contribute to headers being made not-present over time by the swap manager. These results show that our developed address space has the potential for significant real-world benefits during investigations.

**Executable Data Recovery with the Queues**

As mentioned previously, it is not just the headers of executables that can be made not present, but also any other portion of the executables as well. To maintain alignment, memory forensics will fill these missing pages with zeroes (NULL bytes). This can have severe, negative effects on the reverse engineering process as well as other types of forensics analysis. To further test the usefulness of our address space, we calculated how many pages of data belonging to executables were successfully recovered by *mac_procdump* in both runs.

The column marked as ‘Total’ in Table 3.5 lists the total number of pages in executables that *mac_procdump* tried to access but could not recover as it did not have our address space installed. The column marked as ‘Recovered’ lists the number of pages in executables that were recovered by *mac_procdump* once it got our address space installed.
The ‘Recovered %’ lists the percentages of the pages recovered with the help of our address space.

Table 3.5. Number of Processes Recovered

<table>
<thead>
<tr>
<th>#</th>
<th>Total</th>
<th>Recovered</th>
<th>Recovered %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,679</td>
<td>908</td>
<td>24.68</td>
</tr>
<tr>
<td>2</td>
<td>11,325</td>
<td>3,260</td>
<td>28.79</td>
</tr>
<tr>
<td>3</td>
<td>12,251</td>
<td>2,584</td>
<td>21.09</td>
</tr>
<tr>
<td>4</td>
<td>12,181</td>
<td>4,064</td>
<td>33.36</td>
</tr>
<tr>
<td>5</td>
<td>13,857</td>
<td>4,958</td>
<td>35.78</td>
</tr>
<tr>
<td>6</td>
<td>6,340</td>
<td>481</td>
<td>7.58</td>
</tr>
<tr>
<td>7</td>
<td>15,522</td>
<td>1,895</td>
<td>12.20</td>
</tr>
<tr>
<td>8</td>
<td>20,238</td>
<td>3,647</td>
<td>18.02</td>
</tr>
<tr>
<td>9</td>
<td>11,613</td>
<td>271</td>
<td>2.33</td>
</tr>
<tr>
<td>10</td>
<td>102</td>
<td>66</td>
<td>64.71</td>
</tr>
</tbody>
</table>

As shown in the results, our address space provided the ability to recover a substantial number of pages belonging to process executables that otherwise would have been missed. Through use of our address space, analysts will be provided with much richer and complete data sets and artifacts to perform their investigations.

Performance Impact

To assess the performance impact of our address space on analysis, we timed the execution of the previously described mac_procdump-based analysis as well as monitored memory usage during the plugin’s execution. The memory usage is based on the caching the address space does to avoid repeated calculations related to enumerating process mappings and object/offset calculations and lookups. Even on the large samples, these caches combined for only a few MBs of extra memory usage. Table 3.6 lists the runtime for mac_procdump with and without the address space installed.

Of course this minimal performance hit is far outweighed by the forensic informa-
Table 3.6. Runtime of `mac_procdump` in Seconds

<table>
<thead>
<tr>
<th>#</th>
<th>Runtime Without AS</th>
<th>Runtime with AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.25</td>
<td>26.24</td>
</tr>
<tr>
<td>2</td>
<td>31.14</td>
<td>40.54</td>
</tr>
<tr>
<td>3</td>
<td>38.88</td>
<td>40.46</td>
</tr>
<tr>
<td>4</td>
<td>32.99</td>
<td>42.18</td>
</tr>
<tr>
<td>5</td>
<td>32.02</td>
<td>40.74</td>
</tr>
<tr>
<td>6</td>
<td>22.89</td>
<td>30.27</td>
</tr>
<tr>
<td>7</td>
<td>23.94</td>
<td>31.47</td>
</tr>
<tr>
<td>8</td>
<td>27.88</td>
<td>61.40</td>
</tr>
<tr>
<td>9</td>
<td>21.13</td>
<td>277.02</td>
</tr>
<tr>
<td>10</td>
<td>19.01</td>
<td>549.46</td>
</tr>
</tbody>
</table>

The only samples that experienced substantial runtime spikes were samples #9 and #10. As shown previously in Table 3.4, sample #9 is the sample for which the address space recovered a further 238 processes compared to the non-queue aware Volatility. As a result of these 238 extra processes, the amount of data produced by the plugin for that sample grew from 46MB to 234MB. For sample #10, a further 259 processes were recovered, and the amount of data recovered went from 444KB (kilobytes) to 291MB. The few extra minutes these invocations take to run are well worth the substantial amount of extra forensic data.
Chapter 4. Analysis of the Objective-C and Swift Runtimes

Once we were able to reconstruct data from the page queues, we could find a significant number of memory pages that were earlier being ignored by memory forensics tools but could now be analyzed. We then focused our attention to find macOS runtime information in memory. In this chapter, we document our research effort to study the Objective-C and Swift runtimes of macOS. These runtimes power a number of common macOS applications and are also the programming language of choice for a variety of malware samples and toolkits. Given the significant market share of Macs in both enterprise and home environments [51] as well as the targeting of Mac users by nation-state backed threat actors [52, 53, 54, 55, 56, 57], it is necessary for analysts to have full visibility of these runtimes.

4.1. Objective-C Memory Analysis

Objective-C is the primary userland runtime provided by Apple for development on its platforms [58]. It features a C-like syntax, but with features that allow for quicker and simpler development. Part of this feature set is a wide range of APIs that allow for direct interaction with numerous subsystems, including those that manage hardware devices and interface with users and user-driven activity. Given its prevalence on macOS systems along with its history of abuse by malware, being able to deeply analyze the runtime is necessary for complete investigations.

Providing full analysis of this runtime required several artifact recovery capabilities to be developed, including enumeration of the following:

• All loaded classes within a process

• The location of instances of each class

• The instance variables of each class

• The type-specific values of all instance variables

• The methods of each class

Through recovery of this substantial data set on a per-process basis, we were able to develop a variety of plugins that allow for full structured analysis of Objective-C.

4.1.1. Enumerating Loaded Classes

Prior to macOS version 10.10, each loaded class was tracked by the realized_class_hash global variable. Starting with macOS 10.10 and continuing through the latest version, which is 10.15.7 at the time of writing, classes were instead tracked by the gdb_objc_realized_classes global variable. The gdb_objc_realized_classes is of type NXMapT- able and is defined here (https://github.com/opensource-apple/objc4/blob/cd5e62a5597ea7a31dccef089317abb3a661c154/runtime/objc-runtime-new.mm#L910).

Enumeration of this variable allows recovery of the objc_class structure used to represent each class in memory. Inside this structure, there are references to the definition of each class’ variables and methods.

To find the per-class information, the data pointer of the class must be decoded. In Objective-C versions prior to macOS 10.15, this was accomplished by first following the bits member of the class. This value was then masked with the FAST_DATA_MASK
value of 0x7fffffffe0 for 64-bit systems. This then provided the address of the class_rw_t data structure, which provides references to the class_ro_t data structure along with other class values. class_ro_t can then be used to recover the base methods, name, instance variables, and properties of the class. Figure 4.2 shows the code snippet where we locate the obj_class using Volatility’s pre-defined Object() method. The Object() method uses the address offset and the address space to locate a particular data structure. We then find the bits member and mask the value with the FASTDATA_MASK which we have initialized earlier (not shown in the code). The offset thus obtained helps to locate the class_rw_t structure. In Figure 4.3, we see that the class_rw_t structure contains a data structure of type class_ro_t. If we look at the structure of the class_ro_t, we see the name field which can be used to recover the names of the classes. Starting with macOS 10.15, the class_rw_t structure was changed so that the data pointer could point to either the class_ro_t data structure directly or to a class_rw_ext_t structure that contained the ro pointer. This change was encapsulated into a PointerUnion whose flag value determined which data structure the pointer referenced. This change breaks code that only understands the runtime before 10.15.

4.1.2. Locating Class Instances

The objc_object structure is used to represent each instance of a class. Figure 4.1 shows the layout of the objc_object data structure. To locate each instance of a particular
class, the heap region(s) of the process must be scanned, since Objective-C simply allocates the space for an object instance through `malloc` or `calloc` and does not specifically track each instance afterwards.

Luckily, we can use the first member of `objc_object`, `isa`, as both a scanning marker as well as for context. This is possible as the `isa` variable points to the `objc_class` that that defines the object. By first enumerating all classes as described previously, we can then scan the heap memory and look for any addresses that reference a loaded class.

Since our recovery algorithm relies on scanning, our plugins not only find the current set of object instances, but also find deallocated instances. This provides significant analysis capabilities as long-running processes can have a substantial number of deallocated instances still left within the heap regions. Special care must be taken when parsing these instances though as pointers to variable data and method references can lead to buffers that have since been overwritten. This leads to a similar effect as memory-
smearing [59], so our code validates each address and data type before processing it or using it as a result.

![Figure 4.3. class_rw_t Data Structure](image)

![Figure 4.4. class_ro_t Data Structure](image)

### 4.1.3. Parsing Instance Variables

Both instance variables and class methods are declared using the `entsize_list_tt` template. From the programmatic perspective, this template provides a generic list interface, but it is actually backed by an array. To implement this, the template stores the current count of elements, the size of each element, and a pointer to the beginning of the array of elements. This allows trivial enumeration in our plugins as Volatility has built-in
support for arrays.

The set of instance variables for a class are stored as the `ivar_list_t` type, which uses the `entsize_list_tt` template. Each instance variable is tracked by an `ivar_t` structure, which stores the offset, name, and type of the variable. The offset specifies how many bytes away from the beginning of an `objc_object` instance is the value of a variable stored. The name and type are stored as strings. Figure 4.5 shows the Python code for locating the `ivar_list_t` data structure. We access each member of the `ivar_list_t` array using a for loop. The structure of the `ivar_t` data structure is shown in Figure 4.6. The code in Figure 4.5 shows how we read the name and type fields of the `ivar_t` data structure.

### 4.1.4. Decoding Variable Types

To accurately gather the value of a variable, the type string must be interpreted. These type representations are documented online [60], and our Volatility additions currently support parsing the encoded value of the following types:

- Single byte characters
- Signed and unsigned short integers

---

```python
ivar_list = obj.Object("entsize_list_tt<ivar_t, ivar_list_t, 0", offset = bits.ro.ivars, vm = proc_as)
if ivar_list.size>0 and ivar_list.count<024:
    ivars = obj.Object(theType="Array", targetType="ivar_t", offset = ivar_list.first.v(), vm=proc_as, count=ivar_list.count)
for ivar in ivars:
    ivar_name = proc_as.read(ivar.name, 32)
    if not ivar_name:
        continue
    idx = ivar_name.find(\"\x80\")
    ivar_name = ivar_name[idx]
    ivar_type = proc_as.read(ivar.type, 32)
    if not ivar_type:
        continue
    idx = ivar_type.find(\"\x80\")
    ivar_type = ivar_type[idx]
    yield ivar_name, ivar_type
```

Figure 4.5. Code Snippet for Finding Instance Variables
- Signed and unsigned four byte integers
- Signed and unsigned eight byte longs
- Floats and doubles
- Character (string) pointers
- Boolean values
- Class pointers
- Type pointers
- Arrays of supported types
  - `NSString`, `NSInteger`, `NSURL`, `NSArray`

Recovering the direct integer types is straightforward as they are simply stored as their little-endian value. Floats and doubles are stored such that the `f` and `d` type-specifiers to Python’s `struct.unpack` can convert them. Pointers to classes are recursively parsed.

The NS types require separate parsing as they encode the data into separate, undocumented data structures. Binary analysis was required to understand how to recover variables of these types. Decoding variables of type `NSString` requires special care as it can be stored in numerous forms. In one encoding, the string is simply stored starting at the address of the data structure. In a second encoding, the string begins at offset sixteen (16) of the data structure, and in a third encoding, a pointer to a character string is
stored at offset sixteen. Our code correctly recovers the target string in all three encodings that we uncovered during our analysis.

The `NSURL` type was discovered to have a `NSString` that holds the URL at offset 0x18 of its data structure. `NSInteger` references the integer value directly. `NSArray` stores the size of the array at offset 4 and then the elements are listed contiguously after. The type of each element can be determined by following the `isa` pointer for each element. For non-class elements, such as `NSInteger`, the value can be directly interpreted.

![Figure 4.6. ivar_t Data Structure](image)

**4.1.5. Enumerating Class Methods**

The set of methods for a class are stored as the `method_list_t` type, which also uses the `entsize_list_tt` template. Figure 4.8 shows how we access the `method_list_t` and locate each member of the array. Each method is represented by a `method_t` data structure. Figure 4.7 shows the layout of the `method_t` which tracks the method’s name `sel`, parameter specification `types`, and implementation pointer `imp`. As described in Section 4.4.3, our plugins automatically examine the code of each method to look for calls to often-abused methods and functions.
4.2. Swift Memory Analysis

4.2.1. Swift and Objective-C Interoperability

Swift supports two modes of operations. The first is standalone and, in this mode, the runtime does not rely on any Objective-C infrastructure nor does it interact with the Objective-C runtime. The second mode, which is used on all Apple platforms, enables interoperability between Objective-C and Swift. This allows programs written in the two languages to natively interact with each other and share APIs and data structures [61, 62].

Since the target of our research was macOS systems, all samples that we analyzed had interoperability enabled. This allowed us to re-use some of our existing algorithms for enumerating data, but it also meant that we had to deeply study the Swift-specific parts of the runtime as well as the precise mechanisms that enable the interoperability.
4.2.2. Swift Metadata

A significant difference between Swift and Objective-C is that Swift encodes all information related to classes, variables, methods, and types into metadata records [63]. This required us to develop Volatility code that could parse the metadata records and derive the encoded values. It also required our plugins to be aware of when they are dealing with purely Objective-C classes versus a Swift class. These specifics are discussed in detail in the following sections.

Further adding to this complexity, the layout of the metadata structures has changed substantially between releases of Swift [64]. It was not until Swift 5.1, which was released in September 2019, that module stability was added [65]. From our understanding of reading related code and documents, this should prevent the addition of future metadata changes that are not backwards compatible.

4.2.3. Enumerating Loaded Classes

When runtime interoperability is enabled, Swift registers all loaded classes with the Objective-C runtime. This allows us to enumerate Objective-C and Swift classes using the same algorithm as described in Section 4.1.1 for Objective-C.

To determine if a loaded class is a Swift class, we check its data value address for either of the `FAST_IS_SWIFT_LEGACY` or `FAST_IS_SWIFT_STABLE` bits being set. These bits are set by the runtime when Swift classes are registered with the Objective-C runtime.
4.2.4. Locating Class Instances

As described in the Swift Type Metadata documentation [63], Swift class metadata records are also valid *objc_class* structures. When interoperability is enabled, the *isa* pointer serves the same purpose as in pure Objective-C, which is to reference the defining class of the instance. This allows us to follow the *isa* pointer as described in Section 4.1.2 to determine an instance’s type.

To store the Swift-specific information, the Swift runtime a uses *objc_class* compatible metadata record named *TargetClassMetadata*. The beginning bytes of this record match the *objc_class* values and the Swift-specific information is added at the end.

4.2.5. Parsing Instance Variables

Because Swift class metadata records are also valid *objc_class* structures, our initial thought was to recover instance variable information using the technique described in Section 4.1.3. This approach did not work, however, because Swift does not propagate the type information for variables into their *ivar_t* structures, instead only propagating the name and offset fields.

To recover the types of the variables, which is required to properly parse instances of them, we had to instead rely on the Swift metadata record. Each *TargetClassMetadata* class holds a pointer to the *nominal type descriptor* for the class it represents. This descriptor is of type *TargetClassDescriptor*, and it holds information about the location of the methods, variables, superclass, storage size, and more of the target class.

By examining the class descriptor, we can determine the location of the metadata records used for reflection on the class. These records, of type *TargetTypeContextDescrip-
tor, hold the name and type of each variable of the class. By parsing these records, we can gather the type information that is purposely set to null in the *ivar_t* structures of Swift classes. This completes the needed set of type, name, and instance offset for each variable of a class.

We note that we could also recover the offset of each variable by parsing the *field offset vector* of the *TargetTypeContextDescriptor*, but this is unnecessary as we already gather it from the *ivar_t*. This would be required though to support Swift in non-interoperability mode on non-Apple platforms.

### 4.2.6. Decoding Variable Types

Unlike Objective-C, which stores the type string directly in the type member, Swift mangles all type names [66]. This prevented direct use of our existing Objective-C variable parsing code. Instead, we implemented a basic version of Swift’s demangler to support the mostly common used types. This allows parsing the specific values most often encountered during our research. We are currently investigating support for full name demangling in our plugins, but this will be a non-trivial task as the name mangling schema is quite complex and the format changed multiple times between Swift versions. More complete name demangling is left as future work.

### 4.2.7. Enumerating Class Methods

Swift provides a number of mechanisms for classes to declare methods [67]. For the purposes of our research, we divide the many options into two categories, based on whether or not the method information is propagated to the Objective-C runtime. For methods that are propagated, we use the technique described in 4.1.5 to recover the class
method information.

To recover methods that stay solely within the Swift runtime, we must start by inspecting the vtable information provided in the class metadata. This metadata is represented by a TargetVTableDescriptorHeader structure and stores the offset and size of the vtable. Inside the vtable is a TargetMethodDescriptor structure for every method of the class. Unfortunately, this method descriptor only stores the function pointer (implementation address in Swift parlance) of the described method.

In order to recover the method name and argument types, we then have to then cross-reference the function pointer with the symbol table of the analyzed executable. Volatility provides native support for parsing the symbols of Mach-O objects, which is the executable file format for Apple devices. Once a matching symbol is found, we then have the mangled representation, which encodes the function name and parameter types. We note that this is the same approach as described in [68].

By combining recovery of Objective-C methods and Swift methods, we are able to recover all methods of a class.

4.3. Malware Testbed Creation

To ensure that our algorithms were capable of detecting a wide variety of real-world malware, we gathered a diverse set of samples that have been discovered in the wild and that were used in real attacks. Our research of which samples would be useful came from studying macOS malware analysis resources, such as Objective-See [69] and ESET reports [70], searches through threat intelligence sources, and through online searches of reported macOS malware infections and outbreaks.
Table 4.1. List of Functions and Methods Alerted by Default

<table>
<thead>
<tr>
<th>Name</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>popen</td>
<td>Allows running executables</td>
</tr>
<tr>
<td>NSTask::launch</td>
<td>Allows running executables</td>
</tr>
<tr>
<td>if_nametoindex</td>
<td>Used in gathering MAC addresses</td>
</tr>
<tr>
<td>NSCreateObjectFileImageFromMemory</td>
<td>Used for memory-only payload execution</td>
</tr>
<tr>
<td>CGEventTapCreate</td>
<td>Used for keylogging</td>
</tr>
<tr>
<td>CGEventTapEnable</td>
<td>Used for keylogging</td>
</tr>
<tr>
<td>IOHIDManagerRegisterInputValueCallback</td>
<td>Used for keylogging</td>
</tr>
<tr>
<td>CGWindowListCreateImage</td>
<td>Used in generating screenshots</td>
</tr>
<tr>
<td>CGContextClipToRect</td>
<td>Used in generating screenshots</td>
</tr>
<tr>
<td>UIGraphicsGetCurrentContext</td>
<td>Used in generating screenshots</td>
</tr>
<tr>
<td>method_exchangeImplementations</td>
<td>Used for method swizzling (hooking)</td>
</tr>
<tr>
<td>NSWorkspace:: runningApplications</td>
<td>Used to enumerate running processes</td>
</tr>
</tbody>
</table>

For each malware sample, we documented its capabilities and the APIs, classes, and subsystems that it used to accomplish its malicious actions. This was accomplished through a mix of studying existing research as well as performing static and dynamic binary analysis. The results of this effort was a deep understanding of the system features that macOS malware abuses. We then used this knowledge to allow our plugins to generically detect this behaviour in running Objective-C and/or Swift applications.

We document our gathered malware dataset in the remainder of this section and then showcase our new plugins against them in the following section.

4.3.1. Crisis

As discussed in [41], Crisis was a highly sophisticated malware program used to spy on users, including recording of audio and web cameras, stealing browser data, and taking screenshots. It also deployed anti-forensics techniques to hide from live analysis.
4.3.2. EvilQuest

EvilQuest is a multipurpose malware that combines ransomware features along with data exfiltration and keylogging [71] [72].

FinSpy

FinSpy is a commercial surveillance tool that has been used against journalists and political activists throughout many countries in the world [73]. In late 2020, Amnesty International found a copy of this tool for macOS and published analysis of it [74] [75]. The results showed that the tool is capable of keylogging, recording of system audio, web camera recording, screen recording, sensitive file exfiltration, and more.

4.3.3. Komplex

Komplex is a trojan used by APT28, which is the Russian-backed APT group believed to be responsible for the 2016 Democratic National Committee hack as well as attacks against many other high-profile espionage targets [76]. Komplex provides the operator with information on running processes, the currently logged on user, and the victim system itself. It also allows the operator to write to any file on the file system or to execute any command.

4.3.4. MacDownloader

MacDownloader is the malware used by an APT group believed to be based out of Iran [77]. This malware gathers the list of running processes, installed applications, and saved browser data. It also prompts the user with a fake username and password dialogue to steal plain-text credentials. Finally, it gathers the victim’s keychain files, which can be later decrypted using the stolen account credentials.
4.3.5. MacLoader

MacLoader is a trojan used by an APT group believed to be sponsored by North Korea. It has the ability to execute memory-only payloads that are delivered by an operator [78].

Realtime-Spy

Realtime-Spy is a commercial spyware suite that allows logging of all user activity, such as applications executed, keystrokes, emails, and websites visited, as well as capturing screenshots [79]. It came to notoriety when it was leveraged in a phishing campaign against users of the Exodus cryptocurrency software [79].

4.3.6. Ventir

Ventir is a Trojan found in the wild that provides operators with the ability to enable keylogging, downloading and uploading files, and commands execution [80] [81].

4.3.7. XAgentmacOS

XAgent is another trojan used by APT28, but it provides many more capabilities than Komplex [82]. Beyond the capabilities of Komplex, it also allows taking screenshots, stealing Firefox passwords, a remote shell, uploading files over FTP, and reading files from the file system.

4.4. Analysis Capabilities and Evaluation

Our goal in developing new Volatility plugins was to provide both automated detection of malware in a generic manner as well as empower investigators to add new capabilities specific to their investigations. In this section, we document our suite along with their algorithms.
<table>
<thead>
<tr>
<th>Name</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSPasteboard</td>
<td>Allows clipboard monitoring</td>
</tr>
<tr>
<td>AVAudioRecorder</td>
<td>Used to access microphone audio</td>
</tr>
<tr>
<td>AVCaptureDeviceInput</td>
<td>Used to monitor web cameras</td>
</tr>
<tr>
<td>NSEvent</td>
<td>Used to monitor the keyboard, mouse, and other devices</td>
</tr>
<tr>
<td>NSAppleScript</td>
<td>Allows running scripts and gathering data</td>
</tr>
</tbody>
</table>

### 4.4.1. Analyzing Loaded Classes

**Plugin Algorithm**

To detect the use of APIs and classes that are often abused for malicious purposes, we developed the `mac_analyze_classes` Volatility plugin. This plugin uses previously described algorithms to enumerate all Objective-C and Swift classes loaded in a process and then checks if any are in the known-bad list of classes. The plugin comes with a default list of known-bad classes as described in Table 4.2. This list can also be trivially extended by the analyst specifying a file name containing a list of classes with the plugin’s `-class-list` option. This plugin provides an extremely powerful capability as malware must use runtime-provided APIs to access system hardware, gather system state, and to spawn other tasks, and the plugin successfully detects processes using such APIs.

**Plugin Output**

Figure 4.9 shows `mac_analyze_classes` against a sample infected with RealTime-Spy (RTS). As can be seen, the use of NSAppleScript by RTS is automatically detected without requiring hardcoding of RTS-specific indicators.
4.4.2. Analyzing Instance Variables

Plugin Algorithm

The second plugin we developed was mac_analyze_variables. This plugin works by first using previously described algorithms to record the name, type, and offset of each instance variable of every loaded class in a process. It then finds each instance of every class and decodes the variables that are of supported instance types. By default, it will report the name and value of every instance variable of every class. This output can then be stored in a file and searched by the investigator for patterns of interest and/or indicators of compromise. The ability to automatically find all strings, which can include file paths, URLs, command and control configuration data, and more, is a powerful capability that the plugin provides.

Besides simply extracting all variables, the plugin also supports two modes of operation. The first is through –patterns, which specifies the path of file containing a list of search terms of interest. This can be used to filter output to only variables pertaining to a particular investigation or set of IOCs. The second is through –variables, which allows investigators to specify comma-separated lists identifying class names and members names of interest. The plugin will then only report the presence and decoded contents of variables matching the list. This allows IOCs of known-bad classes and variables to be automatically found and reported in samples. Such a capability is particularly useful when an investigator wants to scan many samples in parallel to determine which, if any, are in-
fected. Given the size and scope of modern investigations, these scalable approaches are required.

**Plugin Output**

While analyzing the Ventir malware, we noticed that there was a variable of type `NSString` named `path_res` inside of its `UITimer` class. This variable caught our attention as `mac.analyze.variables` showed that it contained a path inside of our lab user account’s home folder. We investigated further and saw this path was the location in which the malware stores its executables and configuration data. Having this information, we could then make a simple comma separated file containing “UITimer, path_res” and use this file’s path as the argument to `–variables`. Figure 4.10 shows the output of the plugin when ran against a Ventir infected memory sample and our given configuration file. As can be seen, the decoded value is automatically reported as well as the malicious process’ information.

![Figure 4.10. mac.analyze.variables Detecting Ventir](image)

### 4.4.3. Analyzing Functions and Class Methods

Beyond analyzing just the loaded classes and the current values of the instance variables, we also wanted to provide deep analysis of the code running inside of Objective-C and Swift processes. During investigation of our malware analysis dataset, we realized that this meant we needed to analyze both the methods associated with each class as well as the globally accessible methods not associated with a particular class. To meet this criteria, we developed the `mac.analyze.code` plugin, which provides automated analysis
of code inside of Objective-C and Swift processes and reports usage of functions that are known to be abused by malware.

4.4.4. Suspicious APIs

During analysis of our malware dataset, we discovered many runtime-provided APIs that allow for a wide range of system abuse. Most of these were methods not associated with a particular Objective-C or Swift class, but instead are lower level APIs also available to code in other programming languages, such as C and C++. We did however find several methods of Objective-C classes that allowed for abuse. To avoid creating false positives in mac_analyze_classes by alerting on any usage of these classes, we instead incorporated detecting only uses of the methods that are known to be abused. Table 4.1 lists the methods that mac_analyze_code considers suspicious and automatically alerts to when found. As with the other plugins, this list can trivially be extended by analysts through text files.

![Table 4.1: Suspicious APIs](image)

| Process | PID | Class               | Method                           | Suspicious
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>macloader</td>
<td>1596</td>
<td>Authentication Controller</td>
<td>getRunningProcessList</td>
<td>NSWorkspace:runningApplications</td>
</tr>
</tbody>
</table>

Figure 4.11. mac_analyze_code Detecting Mac Loader

Enumerating Functions to be Analyzed

mac_analyze_classes finds the functions and methods to analyze through two sources. First, the methods of all loaded Objective-C and Swift classes are enumerated using the previously described algorithms. Next, the implementation address pointer for each method is recorded. The second source used is the symbol table of the executable being analyzed. This will contain the address of all functions, not just those that are methods of a class. This was determined to be necessary as we found several instances of
malware that used global functions as part of its malicious code flow.

**Code Analysis Algorithm**

The plugin analyzes code through use of the distorm [83] disassembly library. This library provides the plugin with parsable instructions for each instruction of a function and any sub-functions that are also called. When the plugin encounters a `CALL` instruction, which is executed to call a function in a program, it compares the address being called to the addresses of known suspicious functions. The mapping of known suspicious functions to their runtime addresses in a particular process is built by enumerating the symbol table of all libraries mapped into the process. Through this comparison, the plugin is able to report which processes are hosting code that calls suspicious functions as well as context of the calling function.

**Plugin Output**

Figure 4.11 shows the output of `macanalyze_classes` against a memory sample infected with MacLoader. In the output, it can be seen that the `getRunningProcessList` method of the malware’s `AuthenticationController` class calls the suspicious function `NSWorkspace::runningApplications`. This immediately tells the investigator of the process hosting potentially malicious code as well as where to begin any reverse engineering efforts.
Chapter 5. .NET Runtime

In this chapter, we extend our research on memory forensics of programming language runtime environments to analyze the .NET runtime for the Windows operating system. According to recent reports, “Microsoft’s Windows is the most widely used computer operating system in the world with around 70 percent of the market share during the first half of 2021” [112]. Microsoft has added several security features including the Microsoft Defender Antivirus, which protects against potentially dangerous apps, files, websites, and downloads [84]. Though these features provide a measurable amount of protection, they are not adequate compared to the sophistication of the latest .NET malware. This then requires the development of new defensive techniques that can detect and analyze .NET malware. In the next section, we discuss the rising trends in .NET malware, which should concern Windows users and malware defense researchers alike.

5.1. .NET Malware

The .NET languages are extremely suitable for developing and camouflaging malware. For example, malware can easily include a VBA Script or C# executable and pivot to the victim machine unnoticed with the help of some common applications such as Word, Excel, Powershell, etc. Windows has a huge number of essential applications which are written in .NET languages like C#, F#, and VB.NET. Notably, Powershell which is considered one of the key Windows applications, is coded in C#. Other Windows applications that are developed in C# include Windows Installer, Microsoft Visual Studio, and KeePass (a highly popular password manager). Since many such applications are crucial for the operation of the system, they are usually not flagged by antivirus and
malicious authors can easily hide malicious .NET scripts in these applications. Once transferred to the victim machine, the malicious payloads can perform activities like keylogging, installing persistent backdoors, lateral movement, monitoring of web cameras and microphones, and more. These payloads will also often connect to a command and control server, commonly abbreviated as C&C or C2. These servers allow attackers to control victim systems in real time.

An example of these powerful attack frameworks is Cobalt Strike. It is used by governments, businesses, and red team members of organizations to analyze the risk associated with an attack and evaluate post exploitation techniques. Cobalt Strike helps to establish a C&C connection in order to perform malicious activities.

Similar to Cobalt Strike, Covenant is another C&C framework that develops a reverse connection between the attacker and the victim machine. A Covenant attack usually has the following steps. First, the Covenant Listeners are created. Listeners configure the addresses to which the reverse connections report. Next, the Covenant Launchers are used to stage the attack. Launchers generate and download binaries and scripts to help Covenant transfer the attack payload to a victim machine. These payloads are transferred to the victim machine in a way that the users or antivirus installed in the system do not generally detect. The transfer methods can be as simple as sending phishing mails containing the malicious scripts. The Grunts are the implants on the target machine. Once a complete setup is in place, tasks such as keystroke logging, taking screenshots, opening command shells, etc. can be performed by Covenant. Figure 5.1 shows the Covenant dashboard once all the connections have been made for a particular session.

Figure 5.2 shows a screenshot of a victim Windows machine taken by the attacking
Figure 5.1. Covenant Dashboard

Figure 5.2. Screenshot Taken by Covenant
machine running Covenant. Figure 5.3 shows the output of the key logged within 30 seconds. Frameworks like Cobalt Strike and Covenant may include libraries such as Sharp-

![Keylogging Example](image)

Figure 5.3. Keylogging Done by Covenant

Sploit. This is post exploitation written in C# and can provide features like Mimikatz (used for credential theft), shell code execution and so on.

Although tools such as Covenant and Cobalt Strike are intended to help in the analysis and exploitation of system vulnerabilities to secure them, the features they offer are also extremely useful to malware authors. Features similar to the ones provided by these offensive security tools are being used extensively by the malware families known as RATs (Remote Access Trojan). These are remote tools with administrative privileges which can perform malicious activities. In the last couple of years, new many samples of RAT were seen that relied partially or exclusively on .NET features. One such malware sample is Dark Crystal Remote Access Trojan (“dcRAT”). Besides downloading malicious JavaScript encoded files and malicious screen saver files, this Trojan downloads the exe-
cutable files which help the malware figure out if it is being analyzed on a virtual machine. dcRAT tries to establish a connection between the attacker and the victim by opening a URL in Google Chrome. dcRAT also downloads malicious cookies to authenticate itself with the victim’s Google account [113] [114]

The Royal Road v7 Backdoor is another example of a .NET RAT which targeted Mongolian authorities. The malware sample was propagated through a document about COVID19 infections in Russia. Royal Road performed malicious activities, such as allowing the transfer and execution of arbitrary executable files [115].

Another new era of malware that has emerged recently is known as fileless malware. Malware in this category does not write anything to non-volatile storage. Instead, this type of malware misuses the tools and executables that are already present in the system to initiate the attack. Fileless malware may include documents, scripts, and wrappers, many of which are written easily in .NET languages. These payloads can be executed on the fly using .NET objects. Some variants of fileless malware may be embedded in the system or browser. Such malware may be considered fileless as they are embedded even though they might have some traces on the system. However, some variants of fileless malware may be “living off the land” or completely memory-only. Such malware may use the wrapper executable of another process to execute the malware payload [116]. In some cases, the process which the malware uses to evade detection may actually be a legitimate process. Some examples of .NET fileless malware include CactusTorch, UrSnif, etc. According to reports [117], there was a 900% rise in fileless malware in the year 2020.

It is evident that there is an urgent need to develop defensive security tools, which will help detect and analyze .NET malware. We propose memory forensic techniques that
can be used to investigate malicious .NET applications and in the following sections, we discuss the .NET architecture and the layout of the .NET data structures to help understand how these tools are designed. We further explain how these tools detect artifacts within the process memory .NET applications. Finally we demonstrate how our tools can be leveraged to detect and understand malware with the help of a proof-of-concept C# malware.

5.2. .NET Architecture

The .NET Architecture consists of two major components, namely the Common Language Runtime (CLR) and the Class Library.

5.2.1. The Common Language Runtime (CLR)

CLR is the primary platform that is responsible for core tasks such as managing memory, thread execution, code execution, code safety verification, compilation, and other system services in the .NET framework. CLR facilitates memory management with the help of the automatic garbage collection process. Garbage collection is the process of removal of objects that are no longer required to be used. Automatic memory management helps resolves memory leaks and pinpoints invalid memory references in applications. CLR supports Just-In-Time compilation (JIT) which executes computer code “on the fly”; i.e., the code is compiled while it is being executed, at runtime. CLR increases performance by keeping frequently used objects together, thereby reducing memory fragmentation. The Common Type System (CTS) is the part of the CLR that is responsible for the type and code verification. The code managed by the CLR have different trust levels which determines what permissions they will have.
5.2.2. The Class Library

The class library is well integrated with the CLR and provides programmers with a huge collection of methods to be used in their own code. The class library contains object-oriented methods to assist with string management, data collection, database activity, and file access. The programmers may choose from the following three types of class libraries: platform-specific, portable, and .NET Standard libraries[118]. The platform specific libraries work on a single .NET implementation and have access to APIs specific only to that implementation. The portable libraries work on multiple specified platforms. The .NET Standard libraries provide the best of both portable and platform specific. They provide the functionality of a platform-specific library while working on multiple platforms. All these three types of libraries are supported on Mono which is a cross platform.

5.3. .NET Applications

The .NET applications can be broadly classified into three categories, namely Winforms, ASP.Net, and ADO.Net. Winforms are used to develop client-based form applications, such as Notepad. ASP.Net is used for developing web-based applications. ADO.Net is used to access data and data services from a database server. These applications may be written in any of the programming languages that are supported by the .NET runtime. The primary languages that are supported by .NET include C#, F#, and Visual Basic. Each language has a separate language compiler to first convert these programming languages into a Common Intermediate Language (CIL). The assembly using the CIL can execute on any system with the .NET Runtime installed. CIL is converted to the native machine instructions using the JIT compiler.
5.4. .NET Under the Hood

In this section, we discuss the CLR internals including several key data structures that constitute the CLR.

5.4.1. Application Domains

The first step for the CLR before executing any of the managed code is to create three application domains, namely the System Domain, the Shared Domain, and the named Application Domain. The System Domain creates and initializes the default Application Domain. It is also responsible for loading the system library *mscorlib.dll* in the Shared Domain. One of the key goals of the System Domain is to keep track of all the domains in a process and helps in loading and unloading of these domains. The Shared Domain has fundamental types from the System namespace such as Objects, Strings, Enums, Value Types, Arrays, and Delegates. Although most of the user code is loaded in the Default Domain, some of the shared user code may also loaded in the Shared Domain. The Shared domain contains an assembly map that can be used as a lookup table for managing shared dependencies of assemblies being loaded into the Default Domain. The Default Domain is an instance of the Application Domain that contains all the user created application code. Usually only a single domain is created by an application. Each Application Domain contains the following: Security Descriptor, High-Frequency Heap, Low-Frequency Heap, Stub Heap, Handle Table, Large Object Heap Handle Table, Interface Vtable Map Manager, and Assembly Cache.
5.4.2. Loader Heaps

The artifacts accessed by the CLR are loaded in the Loader Heaps. The more frequently accessed artifacts are allocated on a High Frequency Heap, e.g., `MethodTables`, `MethodDesc`, `FieldDesc`, and `Interface Maps`. The less frequently accessed data structures, such as `EEClass` and `ClassLoader` get allocated on a Low Frequency Heap. The Stub Heap has stubs to facilitate Code Access Security (CAS), Component Object Model (COM), and P/Invoke. CAS helps to prevent code which is not trusted from performing privileged actions. The COM wrapper calls allow calls to COM objects from managed code. P/Invoke allows access to structs, callbacks, and functions in unmanaged libraries from managed code. CLR stores the objects with sizes greater than 85,000 bytes on the Large Object Heap (LOH) while the objects with sizes smaller than 85,000 bytes are stored on the GCHeap. This process of separating large objects from small objects help in optimization and garbage collection. The LOH is collected only on full garbage collections, whereas the GCHeap is collected on all garbage collections.

5.4.3. Metadata Streams

CLR has five streams that contain most of the runtime specific data. First is the Metadata stream which has predefined contents and structure. It stores all the information on the types, methods, fields, properties and events in the assembly. Second is the Strings heap where all the namespace, types, and member names are stored. Types are the fundamental unit of programming in .NET and can be implicitly generated or created by the programmer. The User String Heap stores all the strings used in code directly. All the strings a user embeds in the source code end up in here. This stream is only refer-
enced from method bodies. The Globally Unique Identifier Heap (GUID Heap) exclusively stores GUIDs used throughout the assembly. GUIDs are 16 byte unique integers assigned to the .NET objects so that they can be identified using these integers. The Blob heap is primarily used for storing pure binary data such as method signatures, generic instances, etc.

5.4.4. Metadata Tables

The metadata information in the .NET runtime is organised into 45 metadata tables. The metadata tables are shown in Table 5.4.4.

Out of these 45 tables, the most important ones are the TypeDef table, MethodDef table, and the FieldDef table. The TypeDef table contains the information about the classes. The MethodDef table contains the information about the methods whereas the FieldDef table contains the information about the fields.

5.4.5. Assembly

Assemblies are the building blocks of a .NET Application. Each assembly has the manifest file that contains the metadata for the types and resources. Without the manifest file, the CIL code cannot be executed. Each assembly may contain one or more modules. We wrote a plugin to find the pointer to the Manifest file of an assembly. This is the address of the Manifest file. The output of our plugin is shown in Figure 5.4.

```
modhu@bluefish volatility % python vol.py --plugins=/Users/modhu/Desktop/Winplugins -f/Users/modhu/Desktop/Local\ VM/Windows10x64_19041.vmwarevm/Windows10x64_19041-Snapshot5.vmem --profile=Win10x64_19041 showmanifestfile
Volatility Foundation Volatility Framework 2.6.1
(1927907902496, [CType Assembly] @ 0x1C0E0440420)
('ManifestFile Address', <CType pointer to [0x1C0E043FFA0]>)
```

Figure 5.4. Manifest File
<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Module</td>
</tr>
<tr>
<td>01</td>
<td>TypeRef</td>
</tr>
<tr>
<td>02</td>
<td>TypeDef</td>
</tr>
<tr>
<td>04</td>
<td>Field</td>
</tr>
<tr>
<td>06</td>
<td>MethodDef</td>
</tr>
<tr>
<td>08</td>
<td>Param</td>
</tr>
<tr>
<td>09</td>
<td>InterfaceImpl</td>
</tr>
<tr>
<td>10</td>
<td>MemberRef</td>
</tr>
<tr>
<td>11</td>
<td>Constant</td>
</tr>
<tr>
<td>12</td>
<td>CustomAttribute</td>
</tr>
<tr>
<td>13</td>
<td>FieldMarshal</td>
</tr>
<tr>
<td>14</td>
<td>DeclSecurity</td>
</tr>
<tr>
<td>15</td>
<td>ClassLayout</td>
</tr>
<tr>
<td>16</td>
<td>FieldLayout</td>
</tr>
<tr>
<td>17</td>
<td>StandAloneSig</td>
</tr>
<tr>
<td>18</td>
<td>EventMap</td>
</tr>
<tr>
<td>20</td>
<td>Event</td>
</tr>
<tr>
<td>21</td>
<td>PropertyMap</td>
</tr>
<tr>
<td>23</td>
<td>Property</td>
</tr>
<tr>
<td>24</td>
<td>MethodSemantics</td>
</tr>
<tr>
<td>25</td>
<td>MethodImpl</td>
</tr>
<tr>
<td>26</td>
<td>ModuleRef</td>
</tr>
<tr>
<td>27</td>
<td>TypeSpec</td>
</tr>
<tr>
<td>28</td>
<td>ImplMap</td>
</tr>
<tr>
<td>29</td>
<td>FieldRVA</td>
</tr>
<tr>
<td>32</td>
<td>Assembly</td>
</tr>
<tr>
<td>33</td>
<td>AssemblyProcessor</td>
</tr>
<tr>
<td>34</td>
<td>AssemblyOS</td>
</tr>
<tr>
<td>35</td>
<td>AssemblyRef</td>
</tr>
<tr>
<td>36</td>
<td>AssemblyRefProcessor</td>
</tr>
<tr>
<td>37</td>
<td>AssemblyRefOS</td>
</tr>
<tr>
<td>38</td>
<td>File</td>
</tr>
<tr>
<td>39</td>
<td>ExportedType</td>
</tr>
<tr>
<td>40</td>
<td>ManifestResource</td>
</tr>
<tr>
<td>41</td>
<td>NestedClass</td>
</tr>
<tr>
<td>42</td>
<td>GenericParam</td>
</tr>
<tr>
<td>44</td>
<td>GenericParamConstraint</td>
</tr>
</tbody>
</table>
5.4.6. Module

Modules are the building blocks of an assembly. They contain the classes, methods, and fields. We wrote a plugin to find the name and some of the pointers from a module. In Figure 5.5, the *Simple name* is the pointer to the module’s name. We dereference the name pointer to get the module name. The *file* is the pointer to the PEFile. The *Assembly* is a back pointer to the assembly which loads the module. The *TypeDefToMethodTableMap* is the pointer to the lookup map which gives the *MethodTable* addresses. This process has been explained later in Section 5.5.3. The *m_MethodDefToDescMap* is the pointer to the lookup map which gives the *MethodDesc* addresses. This process has been explained in Section 5.5.4. The *m_FieldDefToDescMap* is the pointer to the lookup map which gives the *FieldDesc* addresses. This process has been explained in Section 5.5.5.

```
modhubitional volatility % python vol.py --plugins=/Users/modhu/Desktop/Winplugins -f/Users/modhu/Desktop/Local\ VM/Windows10x64_19041.vmwarevm/Windows10x64_19041-Snapshot5.vmem --profile=Win10x64_19041 showmodule Volatility Foundation Volatility Framework 2.6.1 (140711049820120, [CType Module] @ 0x7FF9D823F7D8) ('simple name', <String pointer to [0x10E1CB265D]>) ('Module Name', 'AreaTest') ('file', <CType pointer to [0x10E043FFA0]>) ('Assembly', <CType pointer to [0x10E044B20]>) ('TypeDefToMethodTableMap', [CType mTypeDefToMethodTableMap] @ 0x7FF9D823F8F0) ('m_MethodDefToDescMap', [CType mMethodDefToDescMap] @ 0x7FF9D823F930) ('m_FieldDefToDescMap', [CType mFieldDefToDescMap] @ 0x7FF9D823F950)
```

Figure 5.5. Module Data Structure

5.4.7. MethodTable

In a .NET application, each class has a MethodTable. All the object instances of the same class will point to the same MethodTable. The MethodTable also contains the number of interfaces implemented, the interface map for method dispatch, the number of slots in the method table, and a table of slots that point to the implementations.
5.4.8. EEClass

One of the key data structure of the Common Language Runtime is the EEClass. The EEClass is created by the ClassLoader from the metadata before the MethodTable is laid out. The MethodTable contains a pointer to the EEClass. Less frequently accessed information like names, fields, and offsets needed by Just-in-Time (JIT) compile functions are loaded in EEClass, whereas frequently accessed information required at runtime like Vtable slots and Garbage Collection information are loaded in the MethodTable. There is one EEClass for each type (interface, class, abstract class, array, and struct) loaded into an AppDomain.

Each EEClass is a node of a tree tracked by the execution engine. The Common Language Runtime uses this tree to navigate through the EEClass structures for purposes including class loading, MethodTable layout, type verification, and type casting. The child-parent relationship between EEClasses is established based on the inheritance hierarchy, whereas parent-child relationships are established based on the combination of inheritance hierarchy and class loading sequence [87].

5.5. .NET Runtime Analysis

The Volatility Framework does provide a suite of memory analysis tools to analyze the Windows operating system, but currently none of these plugins can provide information related to a particular application written in .NET languages such as C#. Given the use of C# in many Windows applications and the fact that it is often used in memory-only malware, its extremely crucial to deeply analyze the .NET runtime for forensic investigations. To offer full analysis of a .NET application we retrieved the following informa-
• The Default Application Domain

• The Domain Assembly and the Module

• The classes within a .NET process

• The methods associated with the process

• The fields present within a process

• The values contained by the fields

In the rest of the section we describe how we find the required information. To understand the layout of the .NET runtime data structures, we did a manual review of the .NET runtime source code [85]. This was a very time-consuming process, but it was necessary to develop the deep familiarity with the data structures required to develop new investigative techniques for .NET. As the CLR contains an enormous amount of data structures, it was impossible to analyze each of them manually from the source code. We therefore retrieved the information from the coreclr PDB file. PDB files contain the types and symbol offsets for data structures defined by an application. For modules for which Microsoft chooses to support full debugging, the corresponding PDB files are hosted on the Microsoft Symbol Server.

Our analysis focused on coreclr.dll, as it is the dynamic linked library containing all the code and data related to the CLR. We used the DLL file located in the %PROGRAM_FILES% as an input for the dotnet-symbol tool to download the coreclr PDB file
containing all the debug symbols from the Microsoft symbol server. To parse the coreclr PDB file into a human readable form, we used the llvm-pdbutil. We downloaded the source code for llvm and installed llvm-pdbutil using Visual Studio. A detailed process for this conversion can be found in [95]. We used the pretty command along with the types sub-command. The complete list of the commands and the sub-commands that can be used to parse PDB files can be found here in [96]. We initially used a simple, non-malicious program to test our plugins. This is a .NET application used to calculate the area of a rectangle.

5.5.1. Finding the Application Domain

We find the Application Domain (AppDomain) by locating the AppDomain::m_pTheAppDomain variable, whose address can be obtained from the PDB file. On locating the AppDomain, we can further find its name using the m_friendlyName field.

5.5.2. The Domain Assembly and Module containing user created data

The assemblies can be accessed by traversing the linked list m_Assemblies. Each element of the linked list is a struct of Type Assembly. The m_pAssembly field provides a pointer to the Domain Assembly.

We locate the manifest file in the Domain Assembly with the help of the m_pManifestFile variable. We can trace the name of the Domain Assembly from the m_identity member of the m_pManifestFile structure. The m_identity structure contains the m_path field which contains the name of the Domain Assembly.

The Domain Assembly may contain one or more modules. Most of the important classes and structs are stored in the master module. We locate the master module with
Figure 5.6. Source Code for Example Program

```csharp
using System;
namespace AreaTest
{
    class Program
    {
        int rect_width;
        int rect_length;
        String rect_owner;
        int rectangle_area;
        public Program()
        {
            rect_width = 5;
            rect_length = 47;
            rect_owner = "Modhuparna Manna";
        }
        public Program(int prov_width, int prov_len, String prov_owner)
        {
            rect_width = prov_width;
            rect_length = prov_len;
            rect_owner = prov_owner;
        }
        public void find_rect_area()
        {
            rectangle_area = rect_width * rect_length;
        }
        public void display_area()
        {
            Console.WriteLine("The area of the rectangle is: " + rectangle_area);
            Console.WriteLine("The owner of the rectangle is:" + rect_owner);
        }
        public static void Main()
        {
            Program r1 = new Program();
            r1.find_rect_area();
            r1.display_area();
            Program r2 = new Program(4, 17, "Selfish Giant");
            r2.find_rect_area();
            r2.display_area();
            Console.WriteLine("Starting sleep time: ");
            System.Threading.Thread.Sleep(2000000000);
        }
    }
}
the help of the \textit{m\textunderscore pModule} field. The \textit{m\textunderscore pSimpleName} gives the name of the module.

5.5.3. The classes within a .NET process

To thoroughly investigate .NET runtime applications, it is crucial that we enumerate the classes within a .NET module. This can be achieved by inspecting the following key data structures:

1. The \textit{MethodTable} for each class
2. The \textit{TypeDef} Metadata table holding class information

**Inspecting the MethodTable for each class**

We discussed about the \textit{MethodTable} structure in section 5.4.7. To get the \textit{MethodTable} of the classes we use \textit{LookupMaps}. From the \textit{Module} data structure, we find the address of the \textit{TypeDefToMethodTableMap}. The \textit{LookupMaps} contain three primary fields. First is the \textit{pNext} pointer that helps to traverse the list of \textit{MethodTables} by pointing to the next element. Second is the \textit{pTable} pointer that points to the list of \textit{MethodTable} addresses. Third is the \textit{dwCount} which is the maximum number of entries that can be present in the list of relative virtual address offsets. With the help of these addresses, we are able to find the \textit{MethodTable} structures for the classes.

In the \textit{MethodTable} structure, we find the field \textit{m\textunderscore wToken}. This token contains a unsigned short integer value which when masked with the METHODTABLE\_TOKEN\_OVERFLOW value, which is 0xffff, gives the index to find a particular record in the metadata database in memory. The masking is done to make sure that the token value does not overflow. In Table 5.4.4, the third entry (marked with 02) is the \textit{TypeDef} metadata table. This metadata table holds the information about the classes. To enumerate
the classes present in the process, we need to extract the `TypeDef` table.

**Inspecting the TypeDef Metadata table holding class information**

There are two structures that hold information on metadata tables. First is the `m_TableDefs` structure. This is an array of length 45 and holds the metadata for the 45 Metadata tables in the metadata database in memory. Second is the `m_Tables` structure. This array is also of length 45 in order to accommodate the information in the 45 Metadata tables. `m_Tables` contains the actual data for the tables.

These two aforementioned table arrays are parallel arrays that map into each other, e.g., index 01 of `m_TableDefs` will hold the metadata information on the `TypeRef` table, while index 01 of `m_Tables` will hold the actual data on the `TypeRef` table. Similarly, index 02 of `m_TableDefs` will hold the metadata information on the `TypeDef` table, while index 02 of `m_Tables` will hold the actual data on the `TypeRef` table, and so on.

The `TypeDef` table which holds the class information has the following structure. Each class forms a record (row) in the `TypeDef` table e.g., if there are four classes in a given process, the `TypeDef` metadata table will consist of four records, one for each class. There are seven fields (columns) in the `TypeDef` table. These fields are defined in the `TypeDefRec` structure. Out of these columns, we are interested in the second column as it contains the offsets to the string table for the names of the classes.

**Extracting the class names using the MethodTable and TypeDef MetaData table**

All the strings across the `AppDomain` are stored in one location - the String Pool in the String Heap. The Core Language Runtime heavily follows an optimization technique known as *string interning*, where only a single entry of a String are stored. String
Interning saves memory by having only a single instance of the string for a given literal across all the application domains. The first time a string literal is encountered, an entry for the string is made into the String Table. The next time the same string literal is encountered, no new entry is created. Instead, there is a reference to the previous entry.

The strings in the interned String Pool are sorted in a particular manner in order to be retrieved easily. To retrieve the actual strings for the class names we first need the segment in which the string is located. This segment information is available from the m_pSegData array. Once we locate the segment in which the string is located, we need to find the index to the first character of the null terminated string in order to retrieve the string. The remainder of this subsection describes how this goal was accomplished. Please refer to the Figure 5.7 for the output values for our memory sample running the Area program.

```plaintext
LiteWeightStgdb 1927908641088
md = 1c0e04f4940
pool = 0x1c0e04f4e68
segdata = 0x1c0e1cb23c8
token = 2
TableDefs = 1c0e04f4a20
cbRec = 0xe
coldef_ptr = 7ffa37f6c9f1
coldef = 0x7ffa37f6c9f4
pdata = 0x1c0e1cb224e
segdata = 0x1c0e1cb23c8 record = 0x1c0e1cb225c m_col = 4
String index before = 29501f4
string_index = 0x1f4
Class: Program
```

Figure 5.7. Code Output Snippet
To find the segment of the String Pool where the class name is located, we must first find the \textit{m\_LiteWeightStgdb} structure. In Figure 5.7, this is the ‘LiteWeightStgdb’ address. This structure points to the String Heap. To locate the string database, we find the Module address using the technique discussed in Section 5.5.2. Each module has a pointer to the \textit{PEFile} which contains the \textit{m\_pMDImport} field which offsets into the required string database structure. In Figure 5.7, the \textit{m\_pMDImport} is the ‘md’ address. Once we locate the String Heap, we can access the String Pool which is divided into segments of a specific segment size. In Figure 5.7, we see the String Pool as the ‘pool’. We can find the segment data using the \textit{m\_pSegData} field and the segment size using the \textit{m\_cbSegSize} field of the String Pool data structure.

On obtaining the segment data and segment size, we proceed to find the string index which is an offset to the first character of the string. The string index is calculated from the address of the record and the address of the field in the \textit{TypeDef} table. The address of the record is calculated using the following:

\[
\text{record\_address} = B + R \times t
\]

Here, B is the table base address of the \textit{TypeDef} Table. R is the value of \textit{m\_cbRec} obtained from the \textit{m\_TableDefs} array, and t is the row index of the particular class.

The base address B of the \textit{TypeDef} table is calculated from the \textit{m\_Tables} array mentioned earlier in this subsection. Each element of the \textit{m\_Tables} array is a \textit{TableRO} structure. To find the class information, we look at the \textit{TableRO} structure at index 2 of the \textit{m\_Tables} array as the \textit{TypeDef} table is the third table in the \textit{m\_Tables} array list. The \textit{m\_pData} field of the \textit{TableRO} structure gives the base address of the \textit{TypeDef} table.

The number of records R in the \textit{TypeDef} table is calculated from the \textit{m\_TableDefs}
array. Each element of the _m_TableDefs_ array is a _CMiniTableDef_ structure. We find the _CMiniTableDef_ structure at index 2 of the _m_TableDefs_ array as the _m_TableDefs_ array maps into the _m_Tables_ array and the _TypeDef_ table is again the third table in the _m_TableDefs_ array list. The _CMiniTableDef_ structure has a _m_cbRec_ field which is an unsigned integer value. In Figure 5.7, it is shown as the ‘cbRec’ value.

The index _t_ is the index of the class which is 1 in our case.

The address of the field is calculated with the help of the _CMiniTableDef_ structure. Along with the record information, the _CMiniTableDef_ structure also has Column information. The column is specified by the _m_pColDefs_ field which is a data structure of type _CMiniColDef_. The Name is the second column of the _TypeDef_ table. From the _CMiniColDef_, we find the _m_oColumn_ which specifies the location of the string index. The _m_oColumn_ is the pointer to an integer that holds the offset of the _m_pSegData_ array.

On adding the record address with the address of the _m_oColumn_ we get the string index. In Figure 5.7, the _m_oColumn_ is shown as the ‘m_col’ value. This string index has to be masked according to the location of the string. The string is masked with 0xffffffff. If it is greater than the segment size calculated earlier, it is further masked with 0xffff. The value of the mask is determined by the _m_iStringsMask_. Adding the masked string index to the segment base address gives the pointer to the first character of the string. We can read the string characters until we find the null terminator to obtain the string.

5.5.4. The methods associated with the process

To enumerate the method names present in a process, we need to analyze the following key data structures:
To get the Method descriptors for the methods within an application, we again use the `LookupMaps`. From the `Module` data structure, we find the address of the `MethodDefToDescMap`. The structure of the `LookupMaps` has been explained in section 5.4.7. The `dwCount` gives us the count of Method Descriptors associated with the `MethodDef` table. The `pTable` pointer is used to point to the first method descriptor address, while `pNext` is used to iterate through the method descriptor address array. Adding these offsets to their location base address gives the actual `MethodDesc` pointers. We can thus find the `MethodDesc` objects at these pointer locations.

The `MethodDesc` structure contains the `m_wSlotNumber`. This unsigned short integer value which when masked with `0xfc00` gives the index to find a particular `MethodDef` record in the metadata database in memory. If we look at the the Metadata tables
mentioned in Section 5.4.4, the fifth table (marked with 06) is the MethodDef table. This metadata table holds the information about the methods. To enumerate the methods present in the process, we need to extract the MethodDef table.

The structure of the MethodDef table is as follows. Each method will form a record (row) in the MethodDef table e.g., if there are 10 methods in a given process, the MethodDef metadata table will consist of 10 records, one for each method. There are 8 fields (columns) in the MethodDef table. These fields are defined in the MethodRec structure. Out of these columns, we are interested in the fourth column as it contains the offsets to the string table for the names of the methods.

To get the actual method name strings, we can use a technique similar to finding the class name strings. The segment information can be found from the String Pool in the same manner as described in section. The calculation of the address of the records and the columns follows the same technique we used while retrieving the class strings. The base address of the MethodDef table is extracted from index 5 of the m_Tables array. The record information can be obtained from m_cbRec field of the CMiniTableDef structure located at the index 5 of the m_TableDefs array list. We get the index information from the slot numbers calculated from the MethodDesc structure. On multiplying the index with the m_cbRec integer value, we obtain an offset which when added to the base address of the MethodDef table gives the record information. The field information is calculated from the CMiniColDef structure. Since the Name field is the fourth column in the MethodRec, the pointer to the base of this column is obtained by adding the m_pColDefs object offset to the size of the CMiniColDef multiplied by 3 as there are 3 columns before the required Name column. Once we find the string index by adding the record address to the column...
address given by $m_oColumn$, we can mask it according to the segment size and find the offset to the first character of the null terminated method name string.

5.5.5. Finding the fields present within a process

To enumerate the field names present in a process, we need to analyze the following key data structures:

1. FieldDesc structure

2. FieldDef Metadata Table

![Field Output Snippet](image)

Figure 5.9. Field Output Snippet

To obtain the Field descriptors for the fields within a process, we implement a technique similar to the one we used to identify the method descriptors. From the Module data structure, we find the address of the FieldDefToDescMap which is a LookupMap similar to the MethodDefToDescMap. The structure of the LookupMaps has been explained in section 5.4.7. The $dwCount$ of the FieldDefToDesc gives us the count of Field Descriptors
associated with the FieldDef table. The pTable pointer is used to point to the first field descriptor offset, while pNext is used to iterate through the field descriptor address array. With the help of these FieldDesc pointers, we can find the FieldDesc objects at these pointer locations.

The FieldDesc structure contains the m_mb. This unsigned short integer value which gives the index to find a particular FieldDef record in the metadata database in memory. If we look at the the Metadata tables mentioned in 5.4.4, the fourth table (marked with 04) is the FieldDef table. This metadata table holds the information about the fields. To enumerate the fields present in the process, we need to extract the FieldDef table.

The structure of the FieldDef table is as follows. Each field will form a record (row) in the FieldDef table e.g., if there are 20 fields in a given process, the FieldDef metadata table will consist of 20 records, one for each field. There are 5 fields (columns) in the FieldDef table. These fields are defined in the FieldRec structure. Out of these columns, we are interested in the second column as it contains the offsets to the string table for the names of the fields.

To get the actual field name strings, we can use a technique similar to finding method name strings. The segment information can be found from the String Pool in the same manner as described previously. The calculation of the address of the records and the columns follows the same technique we used while retrieving the class strings. The base address of the FieldDef table is extracted from index 4 of the m_Tables array. The record information can be obtained from m_cbRec field of the CMiniTableDef structure located at the index 4 of the m_TableDefs array list. We get the index information from
the \textit{m_mb} calculated from the \textit{FieldDesc} structure. On multiplying the index with the \textit{m_cbRec} integer value, we obtain an offset which when added to the base address of the \textit{FieldDef} table gives the record information. The field information is calculated from the \textit{CMiniColDef} structure. Since the Name field is the second column in the \textit{MethodRec}, the pointer to the base of this column is obtained by adding the \textit{m_pColDefs} object offset to the size of the \textit{CMiniColDef} as there is one column before the required Name column. Once we find the string index by adding the record address to the column address given by \textit{m_oColumn}, we can mask it according to the segment size and find the offset to the first character of the null terminated field name string.

\textbf{5.5.6. Finding the values present in the fields}

We have already established the fact that each class has a \textit{MethodTable}. We can find the \textit{MethodTable} address using the technique mentioned in 5.5.3. All the instances of a class will have a pointer \textit{m_pMethTab} that points back to its \textit{MethodTable}. To find the values present in the fields, we need to utilize Volatility’s capability to search the heap regions. As explained in Section 5.4.2 all objects are allocated on the heap. We can therefore scan the heap for all the instances of a class. The subsequent pointers after the \textit{m_pMethTab} belong to fields of the class. Depending on the type of the field we will either get a value or a reference, To find the type of the field we look into the Blob Pool in the Blob Heap. The Blob Heap contains only binary data values. We calculate the blob pointer using the same technique as the string pointer (in 5.5.3). Instead of the \textit{m_iStringMask}, we use the \textit{textitm_iBlobMask}. The first byte pointed by the blob pointer is an encoded integer. We are interested in the next byte which represents the signature
of the field. We look up this byte value in the CorElementType table, eg., a 32 bit integer will be represented by 0x8 in the Blob Pool whereas a string will be represented by 0xe. Depending on the type, we find the values of the respective fields, eg., in case of an integer the subsequent field pointer will directly contain the value while in case of a string the subsequent field pointer will contain another pointer to the first character of the null terminated string.

5.6. .NET Plugin Implementation

Following the detailed analysis discussed in Section 5.5, we were able to create appropriate Volatility plugins. The dotnet_assemblies is used to find information about the assemblies present in the process. The dotnet_classes plugin enumerates the classes in a given process. The dotnet_methods finds the methods in the given class. The dotnet_fields finds the fields in the given class. The dotnet_remote_server plugin is used to find the values of the fields related to our POC command and control server application.

We test our plugins against a memory image running our POC (Proof Of Concept) malware sample. A proof of concept malware does not cause any harm to the system but is often used to demonstrate the capabilities that real-world malware possesses against a system. First, we analyze the memory sample using the dotnet_assemblies plugin. This give us all the loaded assemblies. The output of the plugin is shown in Figure 5.10. We find that most of them are the usual Windows system dynamic linked libraries found in the ‘Program files’ sub-directory. However, the c2app.dll file looks suspicious as it is loaded from the User directory and not the System directory. At this point, we do not know whether it is malicious or not. However, we are interested in investigating
this assembly because of its unusual location and uncommon name. Next, we use our dot-

```
$ python vol.py -f data.lime --profile=Win10x64_19041 dotnet_classes
Volatility Foundation Volatility Framework 2.6
Class Address    Class Name
-----------------  -------------------------
0x00007ffcc64827950 System.AppContext
0x00007ffcc64765d10 System.Array
0x00007ffcc64892758 System.CLRCfg
0x00007ffcc6476f6f0 System.Delegate
0x00007ffcc647618a0 System.Enum
0x00007ffcc648413a0 System.Environment
0x00007ffcc64824170 System.Exception
0x00007ffcc6482ff18 System.GC
0x00007ffcc6489c720 System.Math
0x00007ffcc6476fac0 System.MulticastDelegate
0x00007ffcc64760ae8 System.Object
0x00007ffcc647644c0 System.RuntimeType
  <snip>
0x00007ffcc64862228 c2app.remote_server
0x00007ffcc64841b78 c2app.Program
  <snip>
```

Figure 5.11. Output of dotnet_classes Plugin

`dotnet_classes` plugin to look at the classes present in the process. The output of this plugin is shown in 5.11. We are particularly interested in the classes of the `c2app` assembly DLL.

We find that `c2app` has two classes, namely remote_server and Program. This arouses our suspicion that this DLL has something to do with a remote server. We now suspect that ‘C2’ stands for ‘Command and Control’. There is nothing to confirm our suspicions except to dig deeper and find the fields associated with these classes. We apply our `dotnet_fields`
plugin to the memory sample. The output of our plugin is shown in Figure 5.12. We now

```
$ python vol.py -f data.lime --profile=Win10x64_19041 dotnet_fields
Volatility Foundation Volatility Framework 2.6
Class Name                   Field Address Name          Offset   Token
<snip>
System.String                0x7ffcc64820688 _stringLength 0x08   0x0243
System.String                0x7ffcc64820690 _firstChar 0x0c   0x0244
<snip>
System.Int16                 0x7ffcc64769730 m_value   0x08   0x03ff
<snip>
c2app.remote_server         0x7ffcc64862170 server_url 0x08   0x0001
```

Figure 5.12. Output of dotnet_fields Plugin

see that the application has something to do with a server url and a server port. We are curious to see the values to which these fields are set. We use the `dotnet_remote_server`

```
$ python vol.py -f data.lime --profile=Win10x64_19041 dotnet_remote_server
Volatility Foundation Volatility Framework 2.6
URL                      Port
---------------------------------------------------------------------
https://amaonaws.com/image.gif 17442
```

Figure 5.13. Output of dotnet_remote_server Plugin

plugin to extract the values of the fields. We find that the port is set to 17442. We find that our POC malware sets the url to `https://amaonaws.com/image.gif`. The POC malware uses an incorrect spelling of Amazon AWS. This is a commonly used malicious techniques, where domains registered for use with malware closely mirror real ones.

5.6.1. Detecting Memory-Only .NET Malware

During our analysis we realized that our plugins are capable of detecting memory-only malware, as well. The prime difference between on-disk malware and memory-only malware is that the textitm_path field of the `PEFile` data structure of on-disk malware is
a valid string whereas the textitm_path field of the PEFile data structure of memory-only malware is an empty string. Even for the memory-only malware, we can extract the executable from memory to disk, to then support binary analysis. To get the base address of the PE of the memory-only malware we look inside the PEImage structure. The PEImage structure contains an array called m_pLayouts, where each element has the structure PEImageLayout. We find the member m_base from the first element of the array which reveals the base address. We then extract the memory-only malware onto the disk using an existing Volatility plugin: dlldump. Once the memory-only malware is extracted to the disk, all of the capabilities to inspect the .NET data are the same.

Figure 5.14 shows the source code of our proof-of-concept memory only malware. We see that the remote server part of this malware has no contents in main and all its values are initialized with the help of constructors. The loader for this malware is shown in Figure 5.15. Although in this case we read from a file, in a real malware scenario, the stream of bytes can be easily populated from a web browser keeping no trace of the malware on disk. We use the Load function to load the data in memory. We find the type of the assembly data using the method GetType and use the Activator class we create an object. Figure 5.16 shows the output from our proof-of-concept memory-only malware.

The analysis of the POC malware and POC memory-only malware helps to understand how our plugins help to provide an insight into the activities of .NET malware. From our analysis, we can conclude that our POC is involved in malicious network activities. In the beginning of this section, we discussed the different types of .NET malware and attack frameworks. Our next goal will be to analyze the different attack frameworks and real-life .NET malware. We will also create a database of malicious methods so that
using System;

namespace c2_as_dll
{
    class remote_server
    {
        public string server_url { get; set; }
    
        public short server_port { get; set; }
    
        public remote_server(string url, short port)
        {
            this.server_url = url;
            this.server_port = port;
        }
    
    public static void Main()
    {
    
    }

    }
}

Figure 5.14. Source Code for Memory-only Remote Server
using System;
using System.IO;
using System.Reflection;
using System.Threading;

namespace loadtest
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            byte[] data;
            Assembly asm;

            data = File.ReadAllBytes(args[0]);
            asm = Assembly.Load(data);
            Type t = asm.GetType("c2_as_dll.remote_server");

            object[] b = new object[2];

            string url = "https://memoryonly.com";
            short port = 11225;
            b[0] = url;
            b[1] = port;

            var o = Activator.CreateInstance(t, b);

            Console.WriteLine("o = " + o + " | " + data.ToString());

            while(true)
            {
                Thread.Sleep(1000000000);
            }
        }
    }
}
we can compare if the application being analyzed uses any of the methods used by other malware. It will be interesting to see how our plugins work against the latest “living off the land” malware.

Figure 5.16. Source Code for Memory-only Malware Output
Chapter 6. Conclusions

Memory forensics will continue to be a critical technique in the fields of digital forensics, incident response, and malware analysis. Given the rise of memory-only malware and exploits across all platforms, there is a strong need for memory forensics to recover as much data as possible from application samples written in various programming languages. In this thesis, we have presented novel memory forensic methods for automated analysis of modern runtime environments namely the Objective-C and Swift runtime, and the .NET runtime.

We began by documenting our efforts to incorporate analysis of the macOS page queues into memory forensics. We chose Volatility as our research framework given its popularity in the industry as well as academia. The result of our work is the seamless integration of page queue analysis into all existing and future macOS plugins. As demonstrated in the results of analysis of stacks, heaps, and process executables, a substantial number of pages that were previously inaccessible are now made readily available. This will allow investigators to gather a much richer and more complete set of data and artifacts to perform thorough analysis.

We have described our analysis of the Objective-C and Swift runtimes to find detailed information related to classes, methods, and instance variables, which is crucial for memory analysis of macOS applications. We also deeply analyzed the .NET runtime to find information in memory for applications written in .NET languages like C#. We described how modern malware can abuse .NET’s ubiquity to camouflage scripts and have them executed on the fly. We thoroughly analyzed the .NET runtime data structures to help create memory tools that can investigate .NET malware. The tools developed as a
part of this effort allow for investigators of all skill levels to detect malware and other sus-
picious code running on a system in a straightforward manner. These plugins also allow
for automated extraction of variable data of applications, which can assist in investigations
beyond malware, such as insider threats. The plugins are all easily extensible through ba-
sic text files to support filtering of output and incorporation of IOC (Indicator of Compro-
mise) data.

Given the popularity of macOS and Windows operating systems in the corporate
environment as well as the continued rise in home environments, deeper inspection of
these userland runtimes is now a required feature of memory analysis frameworks. Our
work is a crucial step towards finding vulnerable features provided by these widely used
language runtimes which can be used by userland malware to gain control over the host
machine. In the future, we will continue our fight to keep up with the sophistication of
modern malware and create more powerful tools to help detect and analyze malware
through memory forensics. We have developed this capability for the Volatility framework,
and our goal is to have these capabilities integrated directly into the framework. This will
allow the capabilities to be usable by the entire digital forensics and incident response
community. We strongly believe that our work is a step forward in memory forensic
research and a step closer to making the world safer from cyber attacks.
Appendix: Copyright Information

Publishing Information for the macOS Page Queues Paper on which Chapter 3 is based

Forensic Science International: Digital Investigation
Volume 33, Supplement, July 2020, 301004

DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA

Memory Analysis of macOS Page Queues

Andrew Case a, b, Ryan D. Maggio b, c, Modhuparna Manna b, c, Golden G. Richard III b, c, d, e

https://doi.org/10.1016/j.fsidi.2020.301004
Under a Creative Commons license
Publishing Information for Modern macOS Userland Runtime on which Chapter 4 is based

**Modern macOS userland runtime analysis**
Corresponding author: Ms. Modhuparna Manna
E-mail address: mmanna3@lsu.edu
Journal: Forensic Science International: Digital Investigation
Article number: 301221
Our reference: FSIDI_301221
PII: S2666-2817(21)00129-3

**Your Status**
- I am one author signing on behalf of all co-authors of the manuscript
- I am not a US Government employee but some of my co-authors are
- I am signing on behalf of the corresponding author.
  - Name/Job title/Company: Modhuparna Manna, Graduate Assistant, Louisiana State University
  - E-mail address: modhupamanna10@gmail.com

**Assignment of Copyright**
I hereby assign to Elsevier Ltd the copyright in the manuscript identified above (where Crown Copyright is asserted, authors agree to grant an exclusive publishing and distribution license) and any tables, illustrations or other material submitted for publication as part of the manuscript (the "Article"). This assignment of rights means that I have granted to Elsevier Ltd, the exclusive right to publish and reproduce the Article, or any part of the Article, in print, electronic and all other media (whether now known or later developed), in any form, in all languages, throughout the world, for the full term of copyright, and the right to license others to do the same, effective when the Article is accepted for publication. This includes the right to enforce the rights granted hereunder against third parties.

**Supplemental Materials**
"Supplemental Materials" shall mean materials published as a supplemental part of the Article, including but not limited to graphical, illustrative, video and audio material.

With respect to any Supplemental Materials that I submit, Elsevier Ltd shall have a perpetual worldwide, non-exclusive right and license to publish, extract, reformat, adapt, build upon, index, redistribute, link to and otherwise use all or any part of the Supplemental Materials in all forms and media (whether now known or later developed), and to permit others to do so.

**Research Data**
"Research Data" shall mean the result of observations or experimentation that validate research findings and that are published separate to the Article, which can include but are not limited to raw data, processed data, software, algorithms, protocols, and methods.

With respect to any Research Data that I wish to make accessible on a site or through a service of Elsevier Ltd, Elsevier Ltd shall have a perpetual worldwide, non-exclusive right and license to publish, extract, reformat, adapt, build upon, index, redistribute, link to and otherwise use all or any part of the Research Data in all forms and media (whether now known or later developed) and to permit others to do so. Where I have selected a specific end user license under which the Research Data is to be made available on a site or through a service, the publisher shall apply that end user license to the Research Data on that site or service.
Reversion of rights

Articles may sometimes be accepted for publication but later rejected in the publication process, even in some cases after public posting in "Articles in Press" form, in which case all rights will revert to the author (see https://www.elsevier.com/about/our-business/policies/article-withdrawal).

Revisions and Addenda

I understand that no revisions, additional terms or addenda to this Journal Publishing Agreement can be accepted without Elsevier Ltd's express written consent. I understand that this Journal Publishing Agreement supersedes any previous agreements I have entered into with Elsevier Ltd in relation to the Article from the date hereof.

Author Rights for Scholarly Purposes

I understand that I retain or am hereby granted (without the need to obtain further permission) the Author Rights (see description below), and that no rights in patents, trademarks or other intellectual property rights are transferred to Elsevier Ltd. The Author Rights include the right to use the Preprint, Accepted Manuscript and the Published Journal Article for Personal Use and Internal Institutional Use. They also include the right to use these different versions of the Article for Scholarly Sharing purposes, which include sharing:

- the Preprint on any website or repository at any time;
- the Accepted Manuscript on certain websites and usually after an embargo period;
- the Published Journal Article only privately on certain websites, unless otherwise agreed by Elsevier Ltd.

In the case of the Accepted Manuscript and the Published Journal Article the Author Rights exclude Commercial Use (unless expressly agreed in writing by Elsevier Ltd), other than use by the author in a subsequent compilation of the author's works or to extend the Article to book length form or re-use by the author of portions or excerpts in other works (with full acknowledgment of the original publication of the Article).

Author Representations / Ethics and Disclosure / Sanctions

I affirm the Author Representations noted below, and confirm that I have reviewed and complied with the relevant Instructions to Authors, Ethics in Publishing policy, Declarations of Interest disclosure and Information for authors from countries affected by sanctions (Iran, Cuba, or Syria). Please note that some journals may require that all co-authors sign and submit Declarations of Interest disclosure forms. I am also aware of the publisher's policies with respect to retractions and withdrawal (https://www.elsevier.com/about/our-business/policies/article-withdrawal).

For further information see the publishing ethics page at https://www.elsevier.com/about/our-business/policies/publishing-ethics and the journal home page. For further information on sanctions, see https://www.elsevier.com/about/our-business/policies/trade-sanctions

Author representations

- The Article I have submitted to the journal for review is original, has been written by the stated authors and has not been previously published.
- The Article was not submitted for review to another journal while under review by this journal and will not be submitted to any other journal.
- The Article and the Supplemental Materials do not infringe any copyright, violate any other intellectual property, privacy or other rights of any person or entity, or contain any libellous or other unlawful matter.
- I have obtained written permission from copyright owners for any excerpts from copyrighted works that are included and have credited the sources in the Article or the Supplemental Materials.
- Except as expressly set out in this Journal Publishing Agreement, the Article is not subject to any prior rights or licenses.
- If I and/or any of my co-authors reside in Iran, Cuba, or Syria, the Article has been prepared in a personal, academic or research capacity and not as an official representative or otherwise on behalf of the relevant government or institution.
- If I am using any personal details or images of patients, research subjects or other individuals, I have obtained all consents required by applicable law and complied with the publisher's policies relating to the use of such images or personal information. See https://www.elsevier.com/about/our-business/policies/patient-consent for further information.
Any software contained in the Supplemental Materials is free from viruses, contaminants or worms.

If the Article or any of the Supplemental Materials were prepared jointly with other authors, I have informed the co-author(s) of the terms of this Journal Publishing Agreement and that I am signing on their behalf as their agent, and I am authorized to do so.

Governing Law and Jurisdiction

This Agreement will be governed by and construed in accordance with the laws of the country or state of Elsevier Ltd ("the Governing State"), without regard to conflict of law principles, and the parties irrevocably consent to the exclusive jurisdiction of the courts of the Governing State.

For information on the publisher’s copyright and access policies, please see http://www.elsevier.com/copyright.

For more information about the definitions relating to this agreement click here.

I have read and agree to the terms of the Journal Publishing Agreement.

3 July 2021
Bibliography


into the digital investigation process, Black Hat DC, 2007.


[69] P. Wardle, See, https://objective-see.com/, 2018


[92] Microsoft, https://github.com/dotnet/coreclr/tree/master/Documentation/bo...


[94] Sasha Goldshtein, Pluralsight, https://www.pluralsight.com/authors/sasha-goldshtein?aid=701j0000001heIoAAI&amp;promo=&amp;utm_source=non_branded&amp;utm_medium=digital_paid_search_google&amp;utm_campaign=US_Dynamic&amp;utm_content=&amp;cq_cmp=175953558&amp;gclid=CjwKCAjwq9mLBhB2EiwAuYdMtc15kJ0-chqU20EYt7aSGZLBUC7ppQuWzCgkJSwHHz3i8-3&n22aVrSBoCDWQA2Vd_BwE&amp;exp=3.

18, 2021.


Vita

Modhuparna Manna received her B.Tech. in Computer Science from Heritage Institute of Technology, Kolkata, WB, India, 2014. After obtaining her undergraduate degree, she worked in a Multi-National Corporation named Capgemini. She joined Louisiana State University, Baton Rouge in Spring, 2017 to pursue a Ph.D. in Computer Science under Professor Golden G. Richard III. Her areas of expertise include: Application security, Memory forensics, Reverse engineering, Digital forensics, Malware and vulnerability analysis, and Operating systems. She has publications in conferences including DFRWS, AAFS, and Digital Investigation journal. She has been awarded student scholarships to attend conferences like BlackHat and WiCyS. She has experience in teaching Reverse Engineering, Operating Systems, Java, MATLAB, Data Structures and Algorithms, and CS Teaching Methods, as a teaching assistant. She has developed a cybersecurity curriculum for high school students. She joined University of New Haven, Connecticut in Fall 2021 as a Visiting Assistant Professor.