EXPLORING APPLICATION MEMORY

BY

KEVIN ANDREW LARSON

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Roy Campbell
Increasingly complex malware continues to evade detection, stealing information, taking systems offline, and disrupting functionality of many computer systems. Traditional techniques have not adequately protected systems from attackers, and the most commonly used detection techniques overlook the contents of memory.

Modern systems contain a wealth of information in the contents of memory, but making use of that information is anything but trivial. There are a number of challenges related to both the acquisition and analysis of a system’s memory. Many forensic situations could involve machines in hostile environments, and many acquisition techniques result in artifacts, which reduce the fidelity of the image and hinder the analysis phase. Although the kernel memory space has come a long way in being mapped, the state of application memory has largely been unexplored.

We have created a toolset that extracts the application’s context from the structure of pointers in a sample of that application’s memory. This context allows us to perform statistical analysis, visualize the structure of memory, and provides a new way to train classifiers.
To those who told me I couldn’t
ACKNOWLEDGMENTS

I want to thank my advisor, Professor Roy Campbell, for inspiration, encouragement, support, and most importantly, looking past my undergrad GPA and believing in me. Ellick Chan, Mirko Montanari, and Wucherl Yoo for the mentorship and collaborations in my early years of research. Murph Finnicum as a friend, roommate, and colleague over nearly a decade at the university. My family, for being there when times were tough and helping to make undergrad possible. Anthony Carusiello, Dan Cullina, Phil Garret, Dan Kiss, Lisa Pearson, Aaron Raiff, Darcy Ross, Joe Teslik, and Will Dietz for being incredible friends and making my life in Champaign-Urbana something I’ll never forget. Illini Swing Society, B.U.Z.Z. club, and the Climbing Club for providing amazing friends and color to my life. Renee Dumond, Bradley Smith, Andrea Walker Belcher, Lindey Pelch, and Cassie Stoa for living with my shenanigans in Grad school. And last, but not least, everyone else who supported and encouraged me on my journey.
# TABLE OF CONTENTS

LIST OF FIGURES .......................................................... vii

CHAPTER 1 INTRODUCTION ............................................. 1

CHAPTER 2 BACKGROUND .................................................. 4
  2.1 Forensics ................................................................. 4
  2.2 Forensic Process ....................................................... 4
  2.3 Types of Digital Forensics ........................................... 5

CHAPTER 3 CHALLENGES IN MEMORY FORENSICS ................. 7
  3.1 Acquisition .............................................................. 7
  3.2 Analysis ................................................................. 10

CHAPTER 4 A SURVEY OF MEMORY ..................................... 11
  4.1 Cafegrind ................................................................. 11
  4.2 Life Cycle of Data ..................................................... 12
  4.3 Results ................................................................. 13

CHAPTER 5 MEMORY ACQUISITION .................................... 16
  5.1 Forenscope .............................................................. 16
  5.2 Inception ............................................................... 17
  5.3 Virtual Machines ..................................................... 17
  5.4 Inception + Agent ..................................................... 18
  5.5 Hardware Support .................................................... 18
  5.6 Custom Hardware ..................................................... 18

CHAPTER 6 VISUALIZATION .............................................. 19
  6.1 Acquiring Memory ..................................................... 19
  6.2 Application Heap Extraction ....................................... 19
  6.3 Heap Analysis ........................................................ 20

CHAPTER 7 VISUALIZATION OF CHROMIUM ......................... 25
  7.1 Initial Chromium Evaluation ....................................... 25
  7.2 Visual Analysis of Chromium Tab State ......................... 26
  7.3 Quantitative Analysis of Chromium Tab State ................. 26
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The lifecycle of data</td>
<td>12</td>
</tr>
<tr>
<td>4.2</td>
<td>Firefox: Object age histogram</td>
<td>13</td>
</tr>
<tr>
<td>4.3</td>
<td>Firefox: Freed age vs size</td>
<td>14</td>
</tr>
<tr>
<td>4.4</td>
<td>Firefox: Freed age vs Age</td>
<td>15</td>
</tr>
<tr>
<td>6.1</td>
<td>Subset of the pointer structure of initd</td>
<td>22</td>
</tr>
<tr>
<td>6.2</td>
<td>Pointer structure of rsyslogd</td>
<td>23</td>
</tr>
<tr>
<td>6.3</td>
<td>Pointer structure of udev</td>
<td>24</td>
</tr>
<tr>
<td>7.1</td>
<td>Chromium with one tab open</td>
<td>27</td>
</tr>
<tr>
<td>7.2</td>
<td>Chromium with 25 tabs open</td>
<td>28</td>
</tr>
<tr>
<td>7.3</td>
<td>Some data for various numbers of tabs</td>
<td>28</td>
</tr>
</tbody>
</table>
In June 2010, the company VirusBlokAda [1] discovered a piece of malware which later came to be known as Stuxnet [9]. In addition to being one of the largest and most complex pieces of malware ever written, it was also one of the first discovered piece of malware to specifically target industrial control systems. While it was initially believed to have been active for only a few months, it was later found that variants had been active for over a year. If not for an error introduced in an update, which accidentally caused it to spread outside of the intended systems, it likely could have gone undetected for a significantly longer period of time. Stuxnet demonstrated for the first time that there were entities interested in writing extremely complex and sophisticated malware, and in the light of these sophisticated rootkits, current tools and techniques simply were not up to par.

The sophistication of Stuxnet was immense. Unlike traditional malware, it was designed to only affect very specific system configurations. The goal of Stuxnet’s developers appears to be to damage the Iranian nuclear program, and the end target of Stuxnet appears to have been to damage the uranium centrifuges. In order to achieve these goals, many layers of software were required, including the first ever PLC rootkit. Stuxnet would infect the Step7 software to inject malicious code onto the PLC’s controlling the centrifuges.

This attack is even more impressive when one considers that much of the infrastructure targeted by Stuxnet was in an isolated network, which was not connected to the Internet. Despite being in a controlled environment, the attack was successful in propagating through the Iranian systems, heavily relying on the use of USB thumb drives. If an attack could flourish in such a tightly controlled environment, one could imagine the vulnerabilities that would exist in a more conventional environment.

The majority of popular anti-malware tools are online, that is they execute in the environment of a running system. While this has traditionally been
sufficient, advancements on the malware side, such as rootkits have significantly reduced the effectiveness of online-anti-malware tools. Modern malware, even examples far less sophisticated than Stuxnet, have become very good at hiding their identity. Sophisticated anti-anti-malware techniques have allowed malware to evade or disable conventional malware techniques.

While the majority of forensic use cases are likely on systems controlled by those doing forensics, forensic techniques are also useful in hostile contexts. In the case where the FBI or another agency were trying to get information from an adversarial system, many of the underlying assumptions a conventional forensic analyst would make would be invalid.

Recent works, such as Forenscope [13] have allowed for malware analysis outside of the context of a running system. While this circumvents the anti-anti-malware techniques employed by advanced malware, Forenscope does disable a systems functionality for the duration it runs. The authors addressed this by demonstrating that one of the ideal uses of Forenscope would be to capture memory dumps instead of doing analysis on the spot.

The ability to split the forensic process to separate acquisition and analysis was not new. Traditional techniques involved other means to capture a memory dump. Use of the unix tool dd, actually seemed to be the most common practice, but unlike Forenscope, dd could be subverted by malware and also left artifacts in the dumped memory. There are many unique challenges related to the memory acquisition process, which are discussed in this thesis.

With access to a memory dump, the obvious next step is to analyze the kernel and applications running at the time of the dump. Traditionally, memory dump analysis was a tedious manual process, relying only on the aid of simple tools that searched for strings in the memory dump. In time, tools began to become much more comprehensive in aiding this process. While memory dump analysis has begun to do a thorough job analyzing the contents of memory related to the kernel [6], little has been done to explore the application space.

Sophisticated malware, such as Stuxnet, can include a number of rootkits, many of which exist outside the kernel space. As the current state of the art is lacking in this domain, we developed a tool which can extract some basic application context from the memory dump. This context contains a numerical analysis of the pointers in memory and allows for visualization of
those pointers structure. If this context were to be mined and stored, memory
dumps could be checked against the stored copies from trusted copies of the
applications in order to search malicious modifications. For example, the
complex modifications Stuxnet made to the Step7 software would certainly
have changed the application memory structure significantly. This type of
infection would be an ideal to identify with our tool.

Building off of volatility, which provides process ids, and memory offsets
to the start of heap and stack data, we have created a tool to analyze and
visualize application data in the heap. It provides a variety of numbers
related to the pointer structure and pages in addition to the visualization.
The visualization shows the actual connections between each pointer, which
provides an intuitive way to reason about the structures of the heap.

We run our tool on the Chromium web browser and observe considerable
differences in the memory objects found in dumps of Chromium in different
states.
CHAPTER 2
BACKGROUND

2.1 Forensics

The term forensics originates from the Latin word forensis, which means of or before the forum. Forensics has predated computers by hundreds of years, but many of the ideals remain the same. Physical processes, evidence collection, analysis, etc all have corollaries in digital forensics.

Digital forensics is modern extension of the much broader field of forensics. The term digital forensics has been used in a wide variety of contexts between legal, technical, and common audiences, but for this paper we will use this definition: Tools and techniques to recover, preserve, and examine digital evidence on or transmitted by digital devices.

2.2 Forensic Process

While there are published standards, such as the Electronic Crime Scene Investigation Guide [2] by the National Institute of Justice, it is important to consider that digital forensics is used in a variety of instances beyond that of crime. That being said, the Electronic Crime Scene Investigation Guide provides a good baseline for which we can compare. It outlines the following four main stages of the forensic process:

- Collection (referred to as acquisition) - The first stage of the forensic process. The goal is to find, document, and collect or duplicate as much evidence as possible.

- Examination - Discover the relevant items from those collection, documenting their origin and significance. Digital evidence often contains incredible quantities of information, and a critical part of examination
is the discovery of the relevant pieces of information, which is often akin to finding a needle in a haystack.

- Analysis - Finding the significance of the evidence uncovered in examination and its probative value to the case.

- Reporting - Documentation of not only the points of evidence, but the process and validity of the other stages of the forensic process.

There are other important parts of the forensic process as well. The continued development of tools is invaluable to aiding the stages listed above. It is also important to note that outside of the legal process, different focuses will change how the forensic process is applied. In the case of a company responding to an attack, they are likely be much more concerned about minimizing damage, repairing the affected systems, and preventing further incidents instead of focusing on attribution and validity of their methods. As a result, the methodology can differ considerably between these situations. Also, in practice, many tools can span both the examination and the analysis phases, both finding and analyzing data. There are many other topics of digital forensics, related to these processes, the legal system, scientific approaches, etc, but these are outside the scope of this thesis.

2.3 Types of Digital Forensics

Digital forensics spans a large number of different types of data. Traditionally, digital forensics was performed by shutting down and collecting digital devices. This process is often referred to as dead box forensics. This refers to the fact the system isn't running while (parts of) collection, examination, and analysis are being performed. With the evidence contained in network traffic and encrypted drives being unavailable to deadbox forensics, modern trends have been favoring live forensics. In live forensics, machines are kept online in order to collect this otherwise unavailable evidence. Live forensics allows for network information to be recorded, intercepted, and explored in other fashions, the recovery of data on mounted encrypted drives, volatile memory extraction, and other evidence which would be lost doing deadbox forensics.
2.3.1 Disk Forensics

Disk forensics is the classic deadbox forensic technique. It is quite simple to search unencrypted disks for a variety of information. The structures the disk and operating system use to organize data can be leveraged to quickly find files, metadata, and other forensic information. More exotic information is also available, such as deleted files, paged memory, hibernation files, and more, but these types of information can't be expected. While disk forensics was a great start to digital forensics, additional types of data are important to paint a complete picture of the situation.

Network Forensics

Network forensics is in short, the collection and analysis of network traffic. This includes both wired and wireless communications and due to the transient nature of network traffic, is short lived and volatile. Unlike disk forensics, network forensic information typically isn't available after an incident, so acquisition has to be proactive instead of reactive. Many types of malware use the network for propagation, and detection via intrusion detection systems analyzing network traffic is often how new malware are discovered.

2.3.2 Memory Forensics

Memory forensics is the branch of digital forensics exploring the contents of a machine's volatile memory (typically DRAM and SRAM). There is a wealth of information in volatile memory, ranging from modifications to kernel data structures, to network sockets, to encryption keys which could unlock otherwise useless disks.

While many aspects of memory forensics are similar to all branches of digital forensics, there are a variety of factors which make it a difficult and interesting challenge to do.
CHAPTER 3

CHALLENGES IN MEMORY FORENSICS

There are many unique situations, all which warrant the consideration of volatile memory forensics. Many cases pose unique challenges to both the acquisition and analysis phases of memory forensics.

3.1 Acquisition

3.1.1 Memory Quality

The quality of a memory dump is one of the simplest ways to reason about the quality of an acquisition technique. A perfect memory dump would be identical to the entirety of memory at one point in time. While this ideal is attainable, it is often not fully realized due to many factors in acquisition. The two main factors of memory quality are taint and blurriness.

Taint

When an application, an agent, a buffer, or any component of an acquisition tool uses memory of the machine, that memory use disrupts the capture of a perfect quality memory image. One of the most common (albeit generally pretty crude) ways to capture volatile memory is to use the unix application dd to do a block by block copy of memory from /dev/mem. The use of dd is very popular due to the fact that it is extremely simple to use, captures a decent fidelity image of memory, and is already installed on many systems. The tool provides a great example of taint in a forensic system. It requires memory for the application itself (although quite small), but the memory used for page caching (used to write to disk) can add up to a considerable amount.
Blurriness

Many tools capture memory in a running system. An ideal dump of memory would be from one instance of time. Unfortunately, without the ability to pause the system, the machine will continue to run, so blocks of the dump will be from different time intervals. For example, a tool (such as dd) copies memory 4mb at a time each second to a thumb drive, so the first 4mb are from a time of 0 sec, the second 4mb are from a time of 1 sec, third... 2sec, etc. While many of these blocks will be identical across the time intervals, many blocks will also change. As a result of this pointers will (incorrectly) point to invalid locations, data spanning blocks will be incoherent or lost, and data that was moved to a block that had already been copied wouldn't be found in the dump. The more actively the system is writing to memory, the more significant the effects of blurriness.

3.1.2 Hostile Context

Many examples of malware include components which either passively or actively hide their existence. This includes simple obfuscation such as modification of applications which display files or running programs as well as more complex mechanisms such as hooking into system calls to extreme examples, utilizing firmware and external hardware to aid in obfuscation. Again, dd poses a great example of a tool, which despite its common use, has issues in a variety of circumstances. In regards to malicious context, the obvious shortcoming of dd is that a trivial (and malicious) modification of dd could hide all traces of malware in volatile memory regardless of the size and complexity of the malware. Many techniques like this are still quite suitable for the majority of memory forensics, but the forensic analyst needs to be aware of the possible shortcomings of their tools.

3.1.3 Lack of Permissions

Memory forensics is critical in very different situations, such as trying to do forensics on a hostile machine. If the FBI or a comparable entity finds a locked machine, they are faced with a dilemma. Traditionally, the accepted practice was to shut down the machine and scan drives offline, but
recent trends show increasing trends towards encrypting drives, and presumably people targeted by such an organization would be attempting to protect themselves. Without an encryption key, these drives have effectively random data and are useless for any sort of investigation. In cases where the drive was mounted, the encryption key would be somewhere in volatile memory. Although the lack of permissions also complicates many of the simpler memory acquisition techniques (dd included), many of the more advanced techniques can circumvent this issue.

3.1.4 Internal Compromises

When an attack on an organization occurs from the inside (the attacker will be referred to as the insider), there are a unique set of challenges to the forensic acquisition process. In a similar case to lacking permissions, the machines used by the insider may have been locked down or tampered by leveraging legitimate permissions that he or she may have had. Additionally, the insider could install malicious software to hide their existence, similar to the situation mentioned in Hostile context. Some tools that might be effective in one of the situations will be ineffective when both of the challenges are present together.

3.1.5 Critical Nature of Systems

There are many systems that simply cannot afford downtime. While one could use tools such as dd to capture memory, they would be faced with issues with both memory quality and hostile context. This facet of volatile memory forensics imposes a unique set of challenges for tools in that they have to minimize or eliminate downtime altogether. While pausing a system (either by taking control or using a hardware/software technique to suspend normal execution) is an option for many circumstances, pausing long enough for even a memory dump can be too much in a critical systems environment.
3.1.6 Presence or Absence of Infrastructure

In a system where forensic analysis is desired, there are many techniques that can be applied beforehand to simplify the situation. While obviously not without tradeoffs, these techniques can be used to increase image quality, avoid hostile contexts, and reduce downtimes in cases of critical systems. There exist a spectrum of options, ranging from choices of hardware, to special software implementations, to the creation and use of specialized hardware to aid in forensic efforts (although trivial to reason about, none appears to be publicly available).

3.2 Analysis

In a completely different domain than acquisition, analysis also poses a unique set of challenges. The context of how the dump is captured, the operating system, and many other factors complicate what is already a difficult challenge. While many tools have emerged to simplify this process, they all have their shortcomings and leave something to be desired.
In order to understand what one can expect to find in memory, it is useful to be able to quantify the lifespan of objects in memory. Secondly, this lifespan and how often objects are written to give us a quantitative understanding of how significant the effects of blurriness might be.

4.1 Cafegrind

We created a tool called Cafegrind[14], which is created to quantify many of the characteristics of memory. We ran Cafegrind on a variety of applications.

4.1.1 Design

Cafegrind is an extension to Valgrind[19], a C and C++ memory debugging tool. It has a virtualized processor and already incorporates debugging symbols of the programs it executes. Cafegrind extends Valgrind’s handling of debugging symbols to infer the types of data structures for every memory access. We collect the following data for each access.

1. Type - The type of an object
2. Object Size - The size of an object
3. Age - The length of time an allocation lasts before it is deallocated
4. FreedAge - The length of time a deallocated structure lasts before it is clobbered by a subsequent allocation
5. Reads - Number of reads performed
6. Writes - Number of writes performed
7. Allocation Size - Size of the allocation including slack
4.2 Life Cycle of Data

The life-cycle of data in a program is shown in Figure 1. First, memory is allocated by using a function such as malloc() or new and it is then initialized by a function such as memset(), C++ constructor or memory pool constructor. Once the base object is ready, its fields are populated with information and the data structure is accessed and modified as the program runs. Once the data structure is no longer needed, it is freed and its memory returns to a pool for reallocation. Throughout this process, memory locations can be overwritten by modification, initialization and reallocation. However, the process of relinquishing memory does not always clear the latent contents of the data structure. In many cases, data is only partially destroyed as reuse of a memory area does not always completely overwrite old data. This partial destruction process is one of the underlying principles behind volatile memory forensic analysis and is useful in uncovering freed data. Cafegrind uses empirical methods to track how much data can be recovered from memory dumps that contain both active and freed data.

Figure 4.1: The lifecycle of data
4.3 Results

We ran the popular Firefox web browser in CafeGrind to observe how long objects lasted in memory. This gives us a probabilistic idea of what data types we would find in a memory dump with Firefox and an idea how significant the effects of blurriness might be.

Figure 4.2: Firefox: Object age histogram

Figure 4.2 shows a histogram of the distribution of object ages. Many of the objects allocated by Firefox have a long lifespan. This is likely to be the case because Firefox uses a custom allocator and smart pointers.

Figure 4.3 shows how long freed objects last in memory before they are ultimately reallocated and clobbered. There seem to be three distinct clusters representing long-term, medium-term and short-term reallocations. This behavior is reflective of how the memory allocator redistributes memory. Smaller allocations are more frequent and therefore, the longevity of their data is also shorter because these smaller memory pools are heavily used. Larger allocations tend to be more rare and thus latent data has a longer life expectancy in these pools. However, if the system is running low in mem-
ory, larger pools can be split and reallocated to service requests for smaller allocations.

Figure 4.4 interpolates all this data into one image.

More details can be found in the Cafegrind paper[14] and much more detailed quantitative analysis can be found in Ellick Chan’s thesis[12]
Figure 4.4: Firefox: Freed age vs Age
CHAPTER 5
MEMORY ACQUISITION

In order to do memory forensics, first and foremost, one needs a source of memory dumps. In practice, acquisition of memory dumps often seem to be done in a very ad-hoc fashion. Usage of the UNIX tool dd in conjunction with a portable medium (such as a thumb drive) or netcat seems to be commonplace. Although these memory dumps are not of perfect quality due to the memory footprint of the dd tool and temporal effects of attempting to make a copy of an actively changing system, we suspect in practice, due to the quantity of memory in modern systems, the differences between these memory dumps and the perfect case will be small. That being said, many tools exist offering a variety of advantages over dd.

5.1 Forenscope

Forenscope [13] offers a very unique set of tradeoffs for memory acquisition (as well as a variety of other functionality). Forenscope uses a technique called memory remanence to get the machine running in a forensically stable state (changes aren’t made to disk or memory) in order to acquire dumps or perform analysis. The principle behind memory remanence is that memory is basically a bank of capacitors which can retain their charge for a period of time after being powered off. The duration of remanence depends on a variety of factors including the type and density of the RAM and the temperature of the chips.

In relatively modern systems (Forenscope was tested on a 00’s system) memory remanence is long enough for memory to survive a reboot. As a result of this, a forensic investigator can power a machine off and back on again, boot of an external media (CD drive, USB key, etc), and load a minimal forensic OS which resides entirely in conventional memory (the first
640kB, which is unused in the operation of modern OSes). At this point, the machine is running in a state where it cannot be affected by malware, won’t taint extended memory (everything past the 640kB of conventional memory), and can capture memory without worrying about blurriness.

Forenscope then is able to reconstruct the state of the machine and restore operation. This is due to the fact that the stack and running processes are stored in memory, registers can be restored, and

5.2 Inception

Inception [3] is a bus-based memory hacking tool. It operates over Firewire, Thunderbolt, PCI/PCIe, and other similar interfaces. These interfaces use buses, which allow for direct memory access. Inception is a tool that basically just presents a simple interface to these buses, which allows one to trivially read or write to memory while a system is operational. The logical extension of this is to use one of these interfaces to simply capture the contents of memory. Blurriness is a concern due to the fact the system is operating during the dump.

5.3 Virtual Machines

There are numerous virtual machine(VM) implementations. Popular solutions include QEMU[4], Xen[11], and Virtualbox[5]. Virtual machines have the advantage that the can work from outside the context of the guest OS, can pause the VM so there are no issues with blurriness, and are able to circumvent the issue of taint. Additionally, the VM can simply be resumed and operation can continue. Virtual machines still have drawbacks, for example if the host OS or virtual machine monitor are infected, the dump can definitely be subverted and the tradeoffs associated with stopping execution are still present.
5.4 Inception + Agent

In addition to the ability to read from memory, the buses allow one to write to memory as well. Using inception to write the binary of a small program to memory (preferably in conventional memory to minimize taint to relevant sections), one can exit the normal context of the system. This would pause the system as to prevent blurriness, prevent malicious applications from subverting the dump, and could restore execution using the resuscitation mechanisms from Forenscope.

5.5 Hardware Support

Wang et al [21] created a system leveraging a network card with custom firmware and System Management Mode (SMM) to acquire the contents of memory. The network card can simply use the PCI interface to access the contents of memory. SMM offers quite a few advantages over just using DMA. SMM code can be locked so it isn’t accessible my malware on the computer and beyond that, it can also find the values of registers. Wang also creates an online memory analysis using SMM and a serial port to provide an interface for GDB.

5.6 Custom Hardware

While Wang’s work provided quite a few advantages over software based methods, implementing some sort of custom hardware solution could do much more for the problem. If a hardware solution were created to capture memory, it could be done securely and without taint, blurriness, or interrupting the system. A simple implementation could be having a fully duplicated set of RAM which mirrors the active RAM. When a dump is needed, the main RAM would continue to operate as normal, but the mirrored ram would hold the value from when it was prompted for the dump.
6.1 Acquiring Memory

In order to analyze the contents of memory, first and foremost, one needs a source of memory dumps. Obviously, we had quite a number of options available to us to capture memory. To test and run our tools, we wanted the highest quality dumps possible, so minimizing taint and blurriness was paramount. Options like Forenscope[13] were considered, but the use of these tools would be too time consuming or complex to capture the quantity of memory dumps we required. In the end, we chose the software VirtualBox [5] to virtualize our system and collect memory dumps. Like most virtualization setups, it allows for a variety of useful functionalities, including snapshots, stop/start/pause/resume, memory capture (specifically referenced as a "core dump"), and more. Functionality like snapshotting will be particularly useful when we want to explore how different specific application states affect our model of the application. VirtualBox was selected over a handful of other virtualization options due to is compatibility, ease of memory capture, and ease of use.

6.2 Application Heap Extraction

Now that we have access to high quality memory dumps, the next step is to extract information specific to the applications. The three largest and most relevant sources of information about an application are the heap, stack, and code segment. Beyond this, there is a variety of application context managed by the operating system, but we don’t expect this to be robust enough to contribute much to our model. It does however provide useful functionality
for finding and identifying our targets (the heap, stack, and code segment) in memory.

6.2.1 Volatility

The Volatility Framework is a set of tools used to extract various forms of information from memory images (or samples of memory images). It’s strength lies heavily in the fact that it is able to support memory dumps from an incredible variety of systems. It has profiles created for many flavors of Windows and Mac OSX and a number of Linux distributions as well. Additional Linux distributions are also easily supported as the profile is made up from configuration files and files created from building the kernel.

6.2.2 Heap and Virtual Memory

We created a profile for our target virtual machine and use Volatility to extract the pages from the heap related to our application. Some applications had multiple processes, many of which had heaps of their own; however, one of those heaps was often significantly larger, so we used only the largest heaps in these cases.

6.3 Heap Analysis

Once we have an application’s heap memory specifically isolated and mapped, we need to analyze it to make useful observations. The first thing to acknowledge is that the heap is going to have noise. Due to the nature of how malloc works, there will be pieces of freed data, malloc padding, and worse, uninitialized data in the pages we capture that make up the heap. While freed memory can be invaluable for finding specific data fragments, it provides an obstacle for someone using heap data for identification.

At this point it was important to discern the structure of the memory we were trying to observe. We explored the memory space to find all the pointers and looked for structures made up of multiple pointers. We looked for circular links, long chains, and other patterns indicating common structures. We also tracked invalid pointers (pointers which point to data outside of the
stack) and misaligned pointers. There are a handful of things one wants to take into consideration however. We acknowledge that there could be some memory within the pages mapped to our processes which may be irrelevant to the application we are targeting, but we believe this should not impose a significant enough difference to affect our ability to classify various aspects of applications. Additionally, we were concerned that there would be lots of pointers pointing between the stack and heap, but fortunately that didn’t seem to be too much of an issue in practice.

6.3.1 Pointer Visualization

While trying to understand what these structures looked like, we envisioned various chains, loops, etc of pointers; however, we were not confident in how these chains were connected or how many existed. We needed to create meta-structures to represent the structures the pointers made in memory, but we weren’t sure of the best way to do this. These meta-structures would be used to train machine learning tools to later identify the states of an application, but we did not have a good idea of what characteristics were most important. The idea to visualize the structure came up and became a good intermediate point to do some evaluation. The fundamental idea is simple, each green dot represents a pointer. If the dot isn’t connected to something, that pointer is pointing either to itself or to something that is not a pointer. If the dot is connected to another dot, the pointer is pointing to the pointer represented by the second dot. Linked list structures appear as long lines of singly connected dots. Circularly linked lists or other looped structures will create loops. Balls are also common (lots of dots connected to a single dot) and represent when many pointers point to a single pointer. More complex structures will utilize multiple combinations of these shapes.

While developing our tools, we looked at a handful of applications as well as developing our own. We wrote a simple application that utilized linked lists and behaved deterministically. The specific numbers are not interesting, but it behaves exactly as expected and the numbers and visualizations match up with the intended behavior. Some of the more interesting ones were initd, rsyslogd, and udev, which we will discuss briefly below.
6.3.2 Example Application: initd

Initd is the first application started by the kernel at boot. All other applications are children of initd and it is utilized in a wide variety of functionality including the handling of orphaned processes. We captured the memory image from our virtual machine shortly after opening an application. We used volatility to extract all the pages related to initd. In total, we had 85 pages, which accounts for 348,160 bytes, or 348kb. We identified 21,643 pointers, which take up 173,144 bytes of data, or about 49.7% of the total memory used by initd’s pages. Of these pointers, 17,852 pointers (or about 82% of all pointers) have cyclical dependencies, and account for about 41% of the total memory space of initd. 1622 pointers (7.5%) point to invalid pages (likely references to the stack or code segment) or are not properly aligned. 3549 pointers, (16%) are involved in longer chains of pointers. Some pointers were repeated. A visualization of these pointers can be seen in figure 6.1.

Initd had many pointers, so we can only see a subset of those in the visualization. We can see many small chains of two or three pointers, but there are also a good handful of longer chains as well as many loops.

Figure 6.1: Subset of the pointer structure of initd
6.3.3 Example Application: rsyslogd

Rsystlogd is an application that centralizes the systems logging. It can take a variety of inputs, such as TCP, file, klog, and others, and output to shell, various database formats, various file abstractions, and more. Figure 6.2 shows the pointer structure of rsyslogd.

The visualization of rsyslogd offers a nice contrast to that of initd. While initd had many small structures, it seems the majority of pointers in rsyslogd are involved in one very large structure. Rsyslogd also has a good number of ball pointer structures, something absent from initd. Loops are absent in rsyslogd however.

Figure 6.2: Pointer structure of rsyslogd
6.3.4 Example Application: udev

Udev is the device manager for Linux. It handles /dev as well as adding/removing of devices and loading firmware. Figure 6.3 shows the pointer structure of udev.

Like rsyslogd, udev lacks loops, but like initd, it has a lot of small structures. It does have an interesting double-ball structure, but lacks any additional balls.

Figure 6.3: Pointer structure of udev
Chromium is a web browser developed by Google, which is the open source version of the popular Chrome web browser (it lacks a few of the media and other functionalities of chrome). As of March 2014, usage of Chrome accounted for approximately 43% of worldwide usage of web browsers, which makes it the most popular browser in the world. As we had mentioned previously, many applications have multiple processes. Chrome is no exception to this, and has 7 unique PIDs, of which 5 have heaps. One of the processes’ heaps usually has considerably more pages than the other processes with heaps. Our analysis focuses on only the heap from those PIDs. As our numbers will show, there are enough differences in the memory objects, that just this one heap is enough to show differences between several memory dumps of Chromium in a variety of states.

7.1 Initial Chromium Evaluation

Our initial tests of Chromium involved starting the browser, opening a variety of tabs, and taking a memory snapshot. The analysis with Volatility yielded 5,311 pages, for a total of 21,244kb of data. This includes 68,290 pointers, which have the following characteristics:

- pointers not found in the collection after few traversals: 61
- pointers in a linked list with data at the: 5,959
- pointers which have data immediately: 33,629
- pointers which are not 8 byte aligned: 2,006
- pointers in a linked list and pointing to non-aligned pointer: 78
7.2 Visual Analysis of Chromium Tab State

After our initial visualization, we proceeded to take quite a number of memory dumps outlining a variety of behaviors in Chromium. We experimented with a variety of tabs, windows, general web, and video content.

For this evaluation, we simply opened the browser and took a dump without doing anything. We opened incrementally more tabs (1-5, 10, 25) and took a memory dump after each instance. None of these tabs had any content open, as we just wanted to see if we could quantify and/or visualize the different number of tabs between the snapshots. In Figure 7.1 we have simply opened Chromium and taken a memory dump. Figure 7.2 shows Chromium after 25 tabs have been opened.

For our two tab visualizations, it is interesting how definitively there appears to be a visual difference between these two instances. In the one tab instance, there are a handful of large balls in a sea of linear structures; however, in the twenty five tab instance, there are many small balls (we’re assuming there would be 25 if some were not cropped). It seems that the number of balls is definitely proportional to the number of tabs open.

7.3 Quantitative Analysis of Chromium Tab State

For these tests, we will look at memory dumps taken for Chromium when 2, 3, 4, and 10 tabs (with no content) are open. As you can see in Figure 7.3, we have a good mix of both types of pointers that have consistent numbers across varying amounts of tabs as well as types of pointers that scale with the number of tabs.

First, looking at total pointers, its interesting to observe that each tab correlates approximately 10k additional total pointers. The pointers just going to data increase proportionally, but do not show such a simple numerical increase. Its interesting to see that all the numbers related to circles and partial circles do not seem to be affected though.
Figure 7.1: Chromium with one tab open
Figure 7.2: Chromium with 25 tabs open

Figure 7.3: Some data for various numbers of tabs
There are many extensions to consider for our work. The original vision for the project involved training classifiers to identify when a program is behaving outside of normal parameters, akin to anomaly detection. Anomaly detection techniques are typically far more costly than signature detection in terms of computing resources, and our work is no exception to this. Just extracting the pointer structure from one instance of Chrome takes several hours.

One important thing to note is while our tools are implemented for a Linux environment, there are no parts of our technique that should not work in a Windows or Mac OSX environment. Volatility has very good support for those systems, and especially considering the flexibility of python, our tool chain should be adaptable with minimal effort.

One potential concern would be the classification accuracy of our tool and its ability to discern small changes in an application. We strongly believe that with sufficient training, changes should be evident in both smaller programs (where the change would obviously have a larger effect) and larger programs (where it would likely be easier to hide). We believe using the structure of the objects in memory should provide more granularity than just the data structures, and Cozzie et al. [16] was quite successful isolating the storm malware even though it was only through observing changes in services.exe.

While the computational costs of our tool are large, we think the rewards would be worthwhile. There are many attacks that go undetected for significant periods of time. Stuxnet, despite being one of the largest instances of malware ever discovered, existed for nearly a year undetected [8]. Flame, the largest known instance of malware, went undetected several times longer [10]. The important thing to consider is that both Stuxnet and Flame had infected systems for many months or years before actually damaging any systems. If a weekly test had caught one of them early on, the (suspected
damage to a thousand centrifuges could have been averted.

Another thing to consider is that the cost of training a classifier is very high, especially so if version changes of the application exhibit significant differences in the heap structures. It may require full duplication for every version of the application, which could admittedly be quite expensive. Fortunately, most Linux systems are trending towards larger, more monolithic releases.

We feel our tools would be particularly useful in a variety of specialized environments. Ideal targets would be environments that have lots of headless machines or embedded systems, such as the power grid, where machines typically only run a few versions of a small set of software. The computation costs of profiling the software running on these machines would be low due to the smaller set of appropriate hardware. Additionally, user installations and modifications would be much rarer, so false positives which would stem from upgrades and configuration changes would also be much less of a concern. Machines (even user workstations) that adhere to strict policy would also be good targets. The strict policy would likely simplify the breadth of applications and versions which require training, making the cost of implementing our techniques lower.

There are many things to consider moving forward. While we found good results just from the heap, it should be obvious that much more information can be obtained by also examining the stack and code segment.

8.1 Stack Analysis

The stack is likely to differ from the heap in a variety of ways. The stack frame will be set up and torn down many times, and as a result, the data here is likely to be much more volatile. While the heap mostly contains structures that are some degree of pervasive, the stack is likely to contain data that is much shorter lived. The stack will also provide additional context to that data. As one would walk down through frames of the stack, one could observe what structures in the stack, heap, and code segment are tied to each stack frame. The various frames of the stack could be used to create some sort of structure that would provide additional insight into the data and pointers found in the stack and heap.
8.2 Code Segment Analysis

There is no shortage of data in the code segment. Every application has a number of libraries. Sometimes these libraries are built statically when the code is compiled. The result of this is that every application using these libraries will have its own copies of those libraries. The other possibility is that the libraries are built dynamically, which would result in a variety of different applications sharing some of the pages containing these libraries. We could potentially model what some of these libraries look like and identify what libraries and even possibly what versions of libraries various applications are using. While obviously not always present, when available, debug information could provide quite a bit of insight as well.

Beyond this, with insight into the instruction pointer and other registers (which should be stored in-memory for context switches and therefore available in memory), one could conceivably play or rewind a small bit of execution (obvious barriers include i/o, interrupts, etc).

8.3 Further Heap Analysis

We have found some basic structure from tracing the pointers in the heap, but there is still more we could do. There are plenty of statistics which could be used for creating a stronger model context.

In addition to structural pointer analysis and visualization, there is more insight into the form of data in the heap. For the non-pointer data in the heap, the number of consecutive non-pointer blocks of memory could indicate useful information. There are also techniques which try and infer what type of data various binary data is, but it has traditionally been less than ideally reliable in most cases, with the exception being the identification of strings.

8.4 Further Language and Application Targets

Before diving into the behaviors of Chrome, we only looked at a few other basic applications to test out the functionality of our tool. We would have loved to explore several more applications, in particular applications that do their own memory management, such as Firefox and or applications written
in languages with managed memory, such as Java. It would be particularly interesting to see how those structures compare to applications written in C/C++. The variation between languages would be interesting to observe. It is our suspicion that the very large clusters seen in Chrome and Midori are due to C++’s implementation of vtables. We wouldn’t expect non object oriented languages to exhibit this behavior.

8.5 Malware Identification

The original goal of our tool was to use applications memory to classify the normal behavior of the application. The output of our tools (and some of the other suggested future works) will be a map of the structures in the applications. This component would use these maps to train a set of classifiers. The classifiers would be used periodically to detect if applications running on a system fall outside the normal functionality of that application and sound the necessary alerts if it is in violation. This is an extremely complex task as there are a variety of factors that can change the memory profile of an application.

8.5.1 Malware Changes

In order to do their malicious activity, malware has to modify the infected application in some way. This could include loading additional libraries or adding hooks to the program API or functions. For example, Stuxnet infects the Step 7 executable by injecting it with a malicious .dll (dynamic linked library) and modifies the necessary Windows files to ensure it persists through a reboot [9]. A memory dump of the Step 7 executable would certainly contain traces of these modifications.

8.5.2 Benign Changes

There are many legitimate reasons an application could have a slightly or even significantly changed memory profile. While small patches might only result in slight changes, a new version of the program could have considerably different characteristics, and many programs now incorporate regular updates
for security. Many applications also include options to extend functionality
with plugins or add-ons, all of which would result in an expanded memory
profile.

8.5.3 Classification of Changes

An elaborate tool would be able to avoid false positives for most benign
changes. Version changes, updates, and the installations of plugins or add-
ones are all predictable. An automated tool could trivially be configured to
profile the application after the benign changes were implemented.

8.5.4 Virtual Machine Introspection

As the popularity of the cloud continues to expand, the use of virtualized
environments continues to grow as well. This is an ideal environment for
memory forensics. On one level, this infrastructure makes it simple to cap-
ture memory dumps, but beyond this, there is even more available. Virtual
machine introspection builds off a lot of the technology underneath the vir-
tual machines and allows for fine grained access to memory. This could allow
for applications’ memory to be sampled individually without ever taking the
system offline. It truly is an exciting time for memory forensics.
Traditional malware techniques are typically based on signature detection. These techniques involve the concept of a signature, some sort of identifying trait, code, or behavior which is tied to a specific instance of malware. Conventional anti-virus software is typically just large lists of signatures and processes to iterate through the files on the system looking for these signatures. Anomaly detection is less common, and while signature detection tries to identify malware, anomaly detection does something of the inverse. Instead of identifying something, it instead tries to create an alert when something isn’t as expected.

Without even taking the effectiveness of signature detection into account in an ideal situation, there are two major flaws working against it. For signature detection to work, there are two fundamental requirements which must be met. First and foremost, the signature of the malware must be known. Typically, an antivirus company or security firm (like VirusBlokAda in the case of Stuxnet [1]) identifies a new piece of malware and reports it. The developers of the various signature detection based tools then have to create signatures and the users of the software need to download updates containing those new signatures. In the case of malware that has not yet been identified, unless it recycles code from other known viruses (sophisticated malware would not do this), signature detection would not be able to detect it. In addition to the inability to identify malware for which signatures have not been created, signature detection tools need to be run in an environment where they can see any infected files. Sophisticated malware often disrupts the functionality of signature detection tools, either by the use of rootkits or by disrupting the detection processes directly. Some work has been done to scan systems from a safe context [18], but the majority of tools do not do this.

In the realm of more advanced detection techniques, Cozzie et al. [16],
actually use very similar techniques to what we are envisioning. They created a tool named Laika, which also extracts data from memory dumps in order to classify applications, but focuses on reconstructing the various data structures that the application uses. We’re working to go beyond this and explore the shape and structure of instances of these data structures during execution. While our techniques are computationally much more expensive, we believe this is a necessary cost to find previously unknown malware. Additionally, Laika is constructed as a signature detection based system, where our tool intends to be closer to anomaly detection.

Anomaly detection is based on the principal that there is some sort of normal behavior. If various thresholds are crossed, it can reasonably be assumed that something has been changed. This approach is often used to discover the presence of attacks or malware. In practice, use of anomaly detection seems to be focused on network based information [20]. Our work hopes to instead look for anomalies in the memory objects of applications.

Our work looks at the structure of how data is used in the heap. This is similar to shape detection, a technique for estimating the shape of data structures, which had been explored in compilers and model checkers [15, 17]. Corbett [15] explores these techniques to reduce state models for model checkers which examine concurrency bugs in parallelized Java applications. Ghiya et al. [17] explore the detection of trees, linked lists, graphs, and other structures for use in compilers.
CHAPTER 10

CONCLUSION

Given the fact that modern day antivirus techniques are struggling with sophisticated malware, new techniques are required to continue to protect systems from infection. Few techniques focus on finding infections in applications, and the application context we create will definitely aid in the identification of unknown viruses like Step 7.

Regular signature based scans have become the norm, but that does not account for viruses that have not been identified. Exploratory periodic scans of critical infrastructure could have saved considerable time and resources in cases like Stuxnet. Our tools create a new insight into the context of an application’s memory and will help push state of the art to simplify identification in the future.
REFERENCES


