Reconstructing Java Objects from Virtual Machine Memory Dumps

Florentin Wieser
University of Passau
wieser14@stud.uni-passau.de

ABSTRACT

Interest in forensic analysis of RAM memory recently grew. Said memory contains the latest accessed data which in addition often is unencrypted. As data like variable values in Java programs may be of interest for forensic experts, the volatile memory firstly has to be persisted by taking snapshots. In this work a memory dump is created on a Linux VM while a custom Java program is running. Further, approaches to acquire memory dumps remotely from a second VM on the same hypervisor are presented. Methods used for local memory acquisition are the debugger *GDB* and a system's ability to create crash dumps. The remote dumping approaches failed as no further processable dump files could be generated.
In the second step, the reconstruction of Java objects is done on the basis of a previously generated core dump file. First, the dump file is converted to the *hprof* binary file format which contains the objects in the heap. Afterwards, the actual variable values could be extracted with Oracle's *JHAT* heap analyzing tool.

1 INTRODUCTION

Digital forensics is defined as the preservation, collection, identification, examination, and analysis of evidence derived from digital sources [13].

One traditional sub field of digital forensics is storage forensics. It focuses on the recovery and analysis of file fragments or whole files from persistent storage media. Over the past decade however, interest in the field of memory forensics or more specifically the analysis of volatile memory grew [1]. The examination of a digital device's RAM is interesting, as it contains information connected to recent activity — for example active processes, open network connections, and fragments of program data [1]. Latter seems attractive, as useful content (e.g. actual chat messages and not only metadata) could be retrieved from applications like Java programs.

This information can be important in criminal investigations or the assessment of breaches. In the first scenario, the focus is on the extraction of evidence for legal proceedings. In breach assessment, memory analysis could reveal traces which could yield details about the intruder.

Main memory data is considered volatile, meaning that the data is lost once the power is cut. One way to persist the data is by taking snapshots of the target device's memory. These so-called core dumps allow further analysis even after cord cutting.

As malware has the possibility to hide itself from the system once it gained kernel-level privileges [3], digital forensic experts aim for higher level memory access. One option is the usage of virtual machines (VMs) — the hypervisor has full access to the memory of its guest systems. In this paper, the method of acquiring memory dumps (of the target VM) from a monitoring VM on the same hypervisor is referred to as remote memory acquisition. Remote hence refers to the fact that the acquisition is performed by the monitoring VM and the target VM is unaffected from the acquisition process.

The next step after the acquisition of a memory dump is the analysis of the gained data. In this paper I will try to reconstruct Java objects from memory dumps. More specifically, I try to recover the names and assigned values of variables defined in the class scope. If the recovery is successful, forensic experts can use the approach to get insight into the state of an application.

Two main goals are set: taking a memory snapshots from a remote VM and the reconstruction of Java objects from this snapshot. During reconstruction, also any possible impact from data types or modifiers should be assessed.

Chapter 2 starts with theoretical problems that arise when obtaining memory dumps and presents a taxonomy for acquisition techniques. In section 3 the experimental setup is explained and approaches for the acquisition of RAM images are presented and conducted. The next chapter 4 covers the actual reconstruction of Java objects. Afterwards, memory analysis approaches found in related work are presented in section 5. Chapter 6 compares these approaches to the approach presented in this paper. Finally, section 7 summarizes this paper and shows restrictions as well as further research questions.

2 BACKGROUND

This chapter will introduce basic problems in memory forensics (2.1) and present a taxonomy for persisting techniques (2.2).

2.1 Memory Forensics - State of the Art

Andrew Case and Golden G. Richard III present in their paper 'Memory Forensics — The path forward' current issues for memory acquisition and analysis [1].

According to the authors, the main issues for memory acquisition are page smearing, non-resident pages and dependence on operating system (OS) versions.

Page smearing is the inconsistency between page tables and the corresponding pages of memory. It occurs when the memory changes in the period between acquiring the tables and acquiring the data. The only remedy incorporated by acquisition tools is the attempt to quickly generate the dumps.

Non-resident pages describes the usage of swap-files or similar concepts by the OS. These allow to provide more RAM than physically available by storing pages to other places — often the local file system. Memory acquisition tools are challenged with these (often large) files, as long acquisition times increase page smearing.

Dependency on OS versions refers to the various operating systems available. On Linux, memory acquisition faces the problem of different distributions and versions. Each subversion of a kernel

needs a particularly compiled memory acquisition driver. The broad spectrum of versions and the possibility to compile custom kernels make it hard to create a central kernel module database.

While Windows only has few different (kernel) versions, Microsoft recently started to make major changes to their OS. One modification addressed the layout of the *hibernation file*. This file is created when a system is put to stand-by mode and contains the current RAM content in order to allow a quicker reboot. In the past, these files simply were used by forensic experts. Further, Microsoft started to incorporate mechanisms preventing malware from affecting critical components. This also influences the work of digital forensics as they must access exactly these components.

Besides the acquisition difficulties, the authors also named issues on the specific applications running on a system. There are only few pre-built tools to extract information from common applications like notepad or some chat-clients. All further analysis of applications like browsers, web servers and databases must be done by manual, unstructured analysis.

Unstructured analysis hereby refers to tools like *strings*, *grep*, and hex editors which are used on raw byte streams [1]. In comparison, structured analysis relies on information about underlying data structures and allows a more targeted analysis by accessing known fields.

2.2 Taxonomy of Memory Acquisition Techniques

The first step in the analysis of running Java programs is to persist the target's main memory. Latzo et al. described various techniques in their 'Taxonomy and Survey of Forensic Memory Acquisition Techniques' [3]. Techniques are differentiated by the following three dimensions.

- **Access Hierarchy Level** describes the level of the highest privileged code executed by the technique upon acquiring the memory. Levels range from the least privileged *User Level* over the *Kernel Level* and the *Hypervisor Level* to the highest privileged *Asynchronous Device Level*.
- **Pre- or Post-Incident Deployment** describes the time of installing the acquisition tool compared to the time of the incident. *Post-Incident Deployment* often reduces the integrity as installations alter the memory.
- **Non-Terminating or Terminating Acquisition** describes whether the target program is terminated during the acquisition.

Tools utilized in this work will also be classified by these dimensions.

3 SETUP, GENERATION AND PROCESSING OF CORE DUMPS

The following section describes firstly the used tools for memory acquisition. After a description of the VM setup, the Java program from which I want to reconstruct objects is presented. Finally, different approaches for core dump generation are assessed and performed.

3.1 Used Memory Acquisition Approaches

In this paper, approaches for memory acquisition on the *Kernel Level* or the *Hypervisor Level* are considered. Particularly the tools *LibVMI* and *GDB* (which are categorized as *pre-incident* and *non-terminating*) are used. Another employed approach is the leveraging of system crash dumps — a *terminating* method. Usage of crash dumps may be considered *pre-incident* as the system already includes this feature. Having said that, some systems may be configured to disable crash dumps. In this case, the approach is considered *post-incident* as the system needs to be reconfigured before memory acquisition.

One straight-forward way to access memory in a test setup is to use VMs. The host has full access to the guest system and hence also can dump the memory. However, there is a semantic gap between the introspecting VM and the target VM. Latzo et al. [3] explain this gap as follows: "In contrast to acquisition software running within the kernel, tools that operate on the *Hypervisor Level* generally do not have any contextual information on how to interpret a guest process's address space". The gap therefore refers to the difference between the binary representation and the meaning of the data to the OS.

In order to overcome this gap, the *Hypervisor Level* tool *LibVMI* is used. It performs low-level operations like mapping kernel symbols to virtual addresses and allows higher-level Virtual Machine Introspection (VMI) applications to be built [5].

Another approach to generate memory dumps is the use of debuggers. One wide-spread debugger is the *GNU Project Debugger* (*GDB*) which allows to generate core dumps without terminating the target program. Further, it is able to operate as a pre- and post-incident acquisition tool: either *GDB* starts the debuggee or it attaches to an already running process. In this paper, I will use the post-incident variant and attach to a running process.

3.2 Setup of VMs

The practical experiments were conducted on Linux machines. More specific, the basic setup consists of two VMs, one introspecting VM running Debian 8 (512MB of RAM) and one target VM running Ubuntu 16.04 (256 MB of RAM). Virtualization of the VMs is realized with the hypervisor *Xen*. Both machines are linked with *LibVMI*. This utility allows the introspecting VM to access the memory of the target VM with the high-level VMI framework *Volatility 2.5*.

Reconstruction should be performed on a Java program running on the target machine. The underlying platform is OpenJDK version 1.8.0_222.

As none of the profiles pre-delivered with *Volatiltiy* matched the Ubuntu version of the target VM, a custom profile had to be created by following the instructions in section 3.4.1. Assuming a profile created this way is named *customProfile*, it is passed to *Volatility* with the parameter `--profile customProfile`.

3.3 Analyzed Java Program

As the aim of this paper is the reconstruction of Java objects, a suitable program with known structure is implemented and then run on the target VM. Aspects considered during reconstruction are each classes variable names and the assigned values. This work focuses on variables defined in the member variable scope.

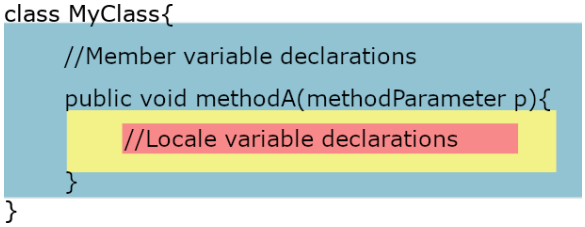

Figure 1: Definition of scopes inside a Java class

3.3.1 Reminder Variable Scope. As the scope of a variable influences the ability to reconstruct it, a short definition is presented in figure 1 which is based on the Java tutorial [11]. Three scopes are depicted: member variable scope, method parameter scope, and local variable scope, but only the member variable scope is examined in this work.

3.3.2 Structure of the Program. The primary objects to be reconstructed are of class *TestObject*. All variables are defined in the member variable scope. As different data types may yield different results, the following variants are investigated.

- **Access Level Modifiers**: public, private, protected
- **Primitive Data Types**: string, int, char, boolean, byte, double
- **Keyword final**
- **Arrays**: one- & multidimensional arrays
- **Objects**: other non-primitive instances

The only defined method is the constructor which assigns actual values to all variables. Instantiation is conducted in a Client class. The Client class only creates two arrays: a String array and a two-dimensional int array and passes these as parameters to the created *TestObject*.

In order to recognize values found in the memory, each variable is filled with an individual value. Figure 2 depicts the relation between both classes. As the example program would terminate directly after the variable initialization, it would be impossible to generate a memory snapshot at the right moment. To allow snapshot generation just after the initialization, the program waits for an arbitrary console input before finishing execution.

3.4 Generation of Core Dumps

In this section, different ways to acquire core dumps are presented. First, the possibility to generate dumps remotely is assessed. As this approach fails, two ways to generate local dumps are presented.

3.4.1 Remote Dump Generation with Volatility. As the VMs were already set up with *LibVMI*, the first attempt to generate a core dump is with the framework *Volatility*. It is a cross-platform, modular, and extensible framework for volatile memory forensics. *Volatility* allows the extraction of information about processes, process memory, networking status, and much more from either memory dumps or a *LibVMI* connection [12].

The framework uses so-called profiles to locate and parse critical data from core dumps. A profile is basically a zip-file containing information on the kernel's data structures and debug symbols [12]. As mentioned earlier, there exist many variations of kernels

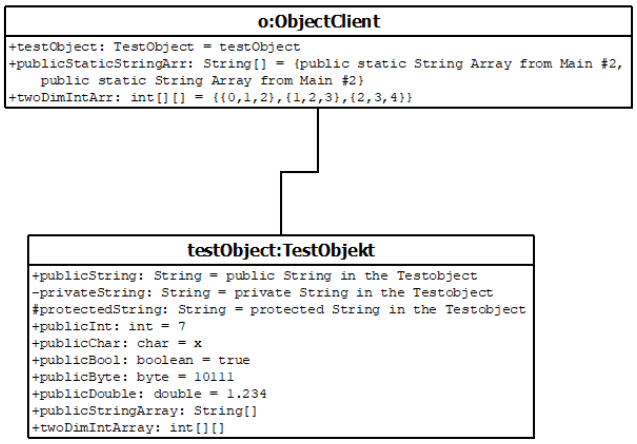

Figure 2: Variable values of the instantiated program

incorporated in Linux operating systems. Therefore, it may be necessary to manually build such a profile for the target system. The creation consists of the following three[1] steps [7].

(1) Create the file *module.dwarf* by executing the tool *dwarfdump* shipped with *Volatility* on the target system. *DWARF* files help to build a logical connection between an output binary and the actual source code
(2) Locate the *System.map* file which commonly is located in the targets */boot*-directory. It contains the name and addresses of a kernel's static data structures
(3) Zip both files and move the resulting archive in the *Volatility* installation folder

Actual acquisition on Linux systems is then done in three steps:

(1) Mount the VMI file system of the target on the monitor VM

```
vmifs name targetVM /mnt
```

(2) Use *Volatility* to find the PID of the Java process

```
volatility -f /mnt/mem linux_pslist --profile
    ↪ customProfile
```

(3) Dump the memory range of the process to the disk

```
volatility -f /mnt/mem linux_dump_map -p PID --dump
    ↪ -dir Dumps --profile customProfile
```

The result of this operation is the data of all memory segments connected to the given process in *.vma* files. Additional information on these files could be extracted with the command `linux_dump_map`. An extract from this command can be found in figure 3.

[1]Detailed instructions can be found on the Volatiltiy wiki [12]

Start	End	Flags	Path
7FFECDDDF000	7FFECDDE1000	r-x	[vdso]
7FFECDD81000	7FFECDDA2000	rw-	[stack]
194E000	196F000	rw-	[heap]
7F25484FC000	7F254866E000	r-x	/usr/lib/...
⋮	⋮	⋮	⋮

Figure 3: Information given by `linux_dump_map`

As there isn't any *Volatility* plugin designed to reconstruct Java objects, analysis has to be done without *Volatility*. The tools used to further process the core dump accept one coherent core file. A core file is basically an ELF file with a specific type flag [2] which contains memory segments. Therefore, attempts to combine the segments generated by `linux_dump_map` into one combined file were conducted:

- *MakeELF* [3] is a Python library to parse, modify and create ELF binaries [6]. At first glance this tool seemed very promising, but unluckily it doesn't support the *ELFCLASS64* and hence also the used 64 bit VMs.
- *PyELFtools* also is a Python library with comparable functionality. However, the intended use of this toolset is rather the analysis and parsing of already existing ELF files than the creation from scratch.

As the artificial composing of the ELF-file with the named tools failed, a different approach of remote memory acquisition was attempted.

3.4.2 Remote Dump Generation with VMIDBG & generate-core-file. The approach uses the tool *VMIDBG* [4]. It also builds on top of *LibVMI* and allows debuggers to remotely access the VMs memory. The fundamental idea is to use *GDB's* remote debugging functionality. *VMIDBG* creates a stub on the monitoring VM and forwards calls to *LibVMI* which in turn passes them to the target system. In order to access information on the target, one has to start *GDB* on the monitoring VM and then use the command `target extended-remote localhost:2159`.

In theory, the subsequent step would be to use the *GDB* utility `generate-core-file` to create a core dump. Unfortunately, this solution doesn't work as the generated files contain no data and *GDB* shows the error message *Command not implemented for this target*.

Another applicable *GDB* command is `dump memory`. It allows dumping specific memory segments. This aproach isn't useful either as the aim is to generate one coherent core dump file. Additionally, the resulting files only contain zeroes.

As the approaches to acquire memory from remote machines weren't successful, I decided to generate a core dump locally on the target machine.

3.4.3 Local Dump Generation with GDB. The first tool used was the already mentioned debugger *GDB*. It attaches to an already running process and then allows to dump the respective memory with the command `generate-core-file`. The resulting file is pretty large (approx. 2 GB), but suitable for further processing as shown in section 4.1. Acquisitions performed by this method require *GDB* installed and are *non-terminating*.

3.4.4 Local Generation of Crash Dumps. Another method to locally generate a memory dump is to leverage the system's ability to create crash-dumps. All relevant desktop OS today include this feature [3]. When a program crashes and the size limit for core-files is set right (e.g. via *ulimit*), a core dump will be created.

(1) Allow all processes to create core dumps with unlimited size
```
ulimit -Sc unlimited
```
(2) Start the target Java program
(3) Terminate the Java process with the `kill -4` command

This approach presumes the target application isn't running before the size limit is set right — either by the system's configuration or by *ulimit*. Either way, the generated files are rather small (around 25 MB) and are applicable for further processing.

3.4.5 Comparison. Both local approaches yield usable core dumps. However, the file sizes differ significantly. When comparing the resulting ELF-files, the size difference could be explained by the number of sections included. The file generated by *GDB* contains over 100 section headers while the file resulted by triggering a crash dump contains none. Anyways, the information in the sections doesn't affect the ability of further processing. As in this work the core dump generation is artificial and the size parameter easily can be set before starting the program, I decided to use the *ulimit* approach as the resulting files are small and therefore easier to process.

4 RECONSTRUCTION OF JAVA OBJECTS

In this chapter the previously generated core dumps are transformed into the *hprof* file format. Afterwards, different tools for analyzing the transformed files are presented. Finally, a script is developed to automate the object reconstruction.

4.1 Further Processing of Core Dumps

This subsection shortly describes the *hprof* file format. Based on this format, analysis is done with graphical tools in a first step and subsequently with a Java tool called *JHAT*.

4.1.1 Conversion of Acquired Dumps. As only the Java heap of a memory dump is interesting for object reconstruction, the first step is to extract the heap from the core dump file. This is possible, as the heap is only a friction of a core dump (see memory sections in figure 3). The tool used for this conversion is *JMAP* [5], a utility shipped with the JDK which prints heap memory details for core files. Oracle's documentation states that this tool is unsupported and experimental which reflected on pointless error messages like 'Can't attach to core file'. But after installing the package *OpenJDK-debug* the tool worked fine.

The result of the conversion is a file in the *hprof binary format*. It contains allocated objects in the heap and its intended purpose is to track down and isolate performance problems involving memory usage [8]. However, the actual data is also contained in the file which allows object reconstruction.

4.1.2 Graphical Tools - MAT & VisualVM. Both tools allow analyzing *hprof*-files. *Eclipse Memory Analyzer* (*MAT*) allows to explore the object graph, find memory leaks, and offers a search engine which processes Open Query Language (OQL) statements [2]. *VisualVM* further allows loading application snapshots or core dumps generated by VM hypervisor software. In my opinion, *VisualVM* is the more intuitive program and it also seems to find more relations between objects than *MAT*.

[2] https://www.sco.com/developers/gabi/2000-07-17/ch4.eheader.html
[3] https://github.com/v3l0c1r4pt0r/makeelf
[4] https://github.com/Zentific/vmidbg
[5] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jmap.html

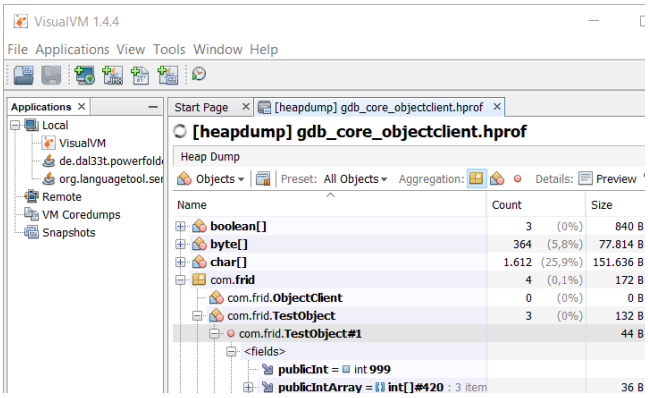

Figure 4: Information provided by *VisualVM* on a *hprof*-file

Figure 5: Webserver of the *JHAT* utility with a query result

The screenshot in figure 4 shows the main window of the tool. All classes which were found in the *hprof*-file are listed, for example `boolean`, `byte` or the custom class `TestObject`. When expanding the class, the instances, references, and fields including the actual values are shown. Further, information about the number of elements and their size is shown.

Still, these graphical tools only allow manual searching of objects and no automatic processing of files. Hence, these tools are useful to get a first insight on the target application. For a more automated reconstruction, further tools are needed.

4.1.3 JHAT. Another tool suitable for browsing allocated objects in *hprof*-files is the *Java Heap Analysis Tool* (*JHAT*)[6]. Like *JMAP*, it is shipped with the JDK and is also considered experimental. It parses a *hprof*-file and then launches a webserver on which one could browse the object topology found in a heap snapshot.

One feature of *JHAT* is the ability to parse OQL queries, which will be used in the next subsection. Figure 5 shows the webserver started by *JHAT* and the result of executing an OQL statement.

[6]https://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html

With the tool *JHAT*, it is possible to retrieve the variable names defined in a class as well as the static fields. Also, once could query class instances and extract the corresponding values.

4.2 Browse Java Heap with OQL

As stated, *JHAT* offers the ability to process OQL statements. OQL is similar to SQL but bases on the JavaScript expression language. It further allows accessing an object's fields with the natural syntax `obj.field_name` and `array[index]`. [7]

Before being able to recreate object instances, the respective class structure must be examined. To achieve this, *JHAT* provides the query-method `heap.classes()`. It returns an enumeration of all class names found in the heap — including platform classes which aren't of any interest. To select the classes of interest, manual filtering is conducted.

Based on the fully qualified class name, one could find further information about its content with the query `select heap.findClass("com.foo.Example")`. By extending the query, one could determine the following properties:

- **fields**: An array containing the member variables:
  - **name**: variable name
  - **signature**: the signature of the variable (e.g. *I* for Integer, or *L* for all kinds of arrays - including strings)
- **statics**: name, value pairs for static variables
- **name**: name of the class

In the next step, the actual instances of found classes are retrieved with the query `select objectid(s) from com.foo.Example s`. Each of the previously found fields for the corresponding class can then be queried. Before accessing the fields, it is necessary to determine the data type. If the target is a primitive type, the assigned value can be accessed directly. If it is an array, each field recursively has to be accessed. When the target is a non-primitive object, only the instance ID can be obtained.

4.3 Automating Object Reconstruction

In order to automate this command sequence, I decided to build a small Python script which sends queries to the webserver and processes the responses. One example method to determine the value of a variable in an object instance can be found in figure 6. The check on the length of the field yields the difference between a singular variable and a respective array. Unfortunately, this distinction cannot be done with the signature property, as it indicates all kinds of arrays (including Strings) with *L*. The recursive call allows traversing multidimensional arrays.

To generate an output, *JHAT* must be running with the desired *hprof*-file. Then a call of the script will produce an output similar to the excerpt in figure 7.

As can be seen, all instances and most of the respective variables including the actual values could be retrieved. Only the byte-variable couldn't be detected. *JHAT* seems to be unable to receive the value as a *java.lang.reflect.InvocationTargetException* is thrown upon accessing this field.

[7]http://cr.openjdk.java.net/~sundar/8022483/webrev.01/raw_files/new/src/share/classes/com/sun/tools/hat/resources/oqlhelp.html

```
def parseVariable(objID, var):
    result = []
    arr = []
    length = parseTable(getHTML("heap.findObject(" +
        ↪ objID + ")." + var[0] + ".length"))
    if(length[0] == "null"): #String or Integer
        result = parseTable(getHTML("heap.findObject(" +
            ↪ objID + ")." + var[0] + ".toString()"))
        print(str(var[0]) + "␣has␣value␣" + str(result
            ↪ [0]))
    else: #actual Array with values
        for i in range(int(length[0])):
            result = parseVariable(objID, [var[0] + "[" +
                ↪ str(i) + "]", "L"])

    return result
```

Figure 6: Method to query data on variables from the *JHAT* webserver

```
$ python ParseJHAT.py
...
Class com.frid.OtherObject - Instance 0xf8c61470
a has value 8
b has value 9
...
Class com.frid.TestObject - Instance 0xf8c5f668
publicString : "Public␣string␣in␣the␣testobject"
privateString : "Private␣string␣in␣the␣testobject"
protectedString : "Protected␣string␣in␣the␣testobject"
publicStringArray[0] : "Public␣static␣string␣array␣from␣
    ↪ main␣#1"
twoDimIntArray[0][0] : "0"
twoDimIntArray[0][1] : "1"
...
twoDimIntArray[2][1] : "3"
twoDimIntArray[2][2] : "4"
publicInt : "7"
publicChar : "x"
publicBool : "true"
Cant decode Variable publicByte
publicDouble : "1.234"
publicOtherObject : "com.frid.OtherObject@0xf8c61470"
publicFinalString : "Public␣final␣string␣preset␣inside␣
    ↪ TestObject"
```

Figure 7: Call & output extract of the script ParseJHAT.py

When parsing non-primitive objects like *publicOtherObject*, only the class name and ID of the object is given. However, the instance of this object is parsed as well in the first lines of figure 7.

5 RELATED WORK

In this section some results of related research are presented. Chapter 5.1 compares the ability to reproduce Java artifacts on different OS. Afterwards, two publications investigating memory forensic approaches on Android systems are summarized in chapter 5.2 and chapter 5.3.

5.1 Exploring RAM Artifacts of Java Programs

Al-Sharif et al. investigated in the paper 'Live Forensics of Software Attacks on Cyber Physical Systems' which remnants could be found in RAM dumps of VMs running Java programs [10]. One major aspect of their work was to test the same Java program on the operating systems Windows, Mac OS and Linux Fedora in different scenarios. These scenarios included stopping the program, terminating the JVM and explicitly invoking the garbage collection.

The experiment performs unstructured analysis on bit-by-bit copies of the memory. Examination is elaborated by searching the dumps for string literals that also occur in the source code of the running program. Besides the source code, no knowledge about the OS nor the used Java version is assumed.

The authors found that variable values are more likely to be found in the Windows dumps than in the ones generated on UNIX-based operating systems like Linux or Mac OS. As the same Java versions were used, this effect is attributed to the different implementation of the JRE (including the JVM) on each platform. Further, they discovered that objects could even be restored after garbage collection was explicitly invoked or the software was stopped. This shows that the main memory may contain important information for forensic experts, even when the action of interest happened earlier in time.

5.2 Analysis of Android Dumps

The student research paper 'Android Memory Dump Analysis' describes the acquisition, the conversion, and memory dump analysis of popular Android applications [4]. Acquisition and conversion of the dump file is specific to the Android system and therefore will be skipped here. The analysis was conducted using the program *Eclipse Memory Analyzer* (*MAT*) which analyzes binary formatted heap dumps (*hprof*-files). Its feature to process OQL statements was used by Leppert to find all instances of the class *String* in an application's dump and look for regular expressions/strings of interest. This analysis resulted in the extraction of usernames and email-addresses of the Facebook app and the inbox content of the GoogleMail application. The increasing spread of smartphones and the finding of sensitive data on them stresses the importance of including memory forensics in investigations.

5.3 Live Memory Forensics with Volatility

Macht built his diploma thesis 'Live Memory Forensics on Android with Volatility' on top of the work done by Leppert [4] [7]. His aim is to step away from the unstructured analysis of text-literals to a structured approach. The basis for his work is the *Volatility* framework.

In the further thesis, Macht creates *Volatility* plugins for the analysis of Android applications. The plugin functionality encompass for example the reading of emails from the *K9-Mail* application or the accessing of chat conversations from *WhatsApp*. In comparison to the results by Leppert, Machts outcomes are highly structured and allow further insights than the simple String results provided by the earlier work.

6 EVALUATION

The approach using *hprof*-files and *JHAT* to recreate Java objects was shown to be successful. Variable names and values of almost all types could be retrieved.

When comparing the presented approach with unstructured analysis, the reconstruction seems more promising than the unstructured analysis as additional information (the name of the variable containing the value) is available. This fact allows a quicker manual browsing of the extracted values — especially when a program is repeatedly assessed and the important variable names are already known.

However, it doesn't unveil the complete underlying structure like the usage of *Volatility* plugins does. When applying the presented approach, the meaning of the results still have to be assessed manually. While this manual work is a downside, the presented approach is way more generic. While a new *Volatility* plugin may be created for each version of the target application, the presented approach barely is impacted by the underlying structure and could easily be transferred to new applications.

7 CONCLUSION

Concluding, one could record that the reconstruction of Java objects on *hprof*-files can be achieved with little effort. *JHAT* allows to access all objects of interest — either manually or with automated requests to the webserver. Testing different variable characteristics shows, that the access modifiers don't affect the reconstruction ability. The only data type with limited results is the byte data type.

One remaining issue is the generation of suitable core files from remote machines. The approach using *Volatiltiy* fails as the output consists of separate dumps of memory regions — *JMAP* requires one continuous dump file. Attempting to attach *GDB* to the remote machine also failed as the dump command is unsupported. Feasible solutions to acquire core dumps were conducted on the target machine itself: *GDB* and crash dumps (using *ulimit*) both produced appropriate dumps.

Another challenge is the conversion of the generated core dump to the *hprof* file format. The problematic issue is the dependency on the tool *JMAP* which transforms core dumps into the desired file format. This tool is considered experimental and lacks proper documentation which complicates the use of it. Despite these uncertainties, the presented approach produced useful results.

7.1 Restrictions

Results presented in this paper underlay some restrictions. Analysis was conducted on machines running Java 8. On later versions the basic proceeding should be applicable, but parts like the referenced Java tools may change. In Java 9 for example, Oracle merged the analysis tools into the *jhsdb* utility. This merger may require changes to the shown approach.

Also the Java program under investigation is simply structured. More complex programs which are in use on productive systems may deliver lots of variables which results in extensive manual work.

7.2 Further Research Questions

As this paper failed to actually reconstruct Java applications on remote machines, this should be the primary focus of further research. Especially the combination of the numerous files generated by *Volatility* into one coherent ELF-file seems pursuable. Besides the already mentioned tools, the *LIEF*-library [8] and *PWNtools* [9] incorporate some kind of ability to work with ELF-files. Due to time constraints, these weren't assessed in this paper.

Another possible research topic is the impact of garbage collection (GC) on the ability to reconstruct objects. GC improves runtime performance by sweeping memory of dead objects when memory is exhausted. Because GC is only invoked once memory is exhausted and not when objects die, its possible for forensic experts to restore latent information [9]. Research should assess the extent to which object reconstruction can be applied on dead objects.

REFERENCES

[1] A. Case and G. G. Richard III. "Memory forensics: The path forward." In: *Digital Investigation* (Jan. 2017). DOI: 10.1016/j.diin.2016.12.004.

[2] Eclipse Documentation. *Memory Analyzer Tool*. accessed on 07.12.2019. URL: https://help.eclipse.org/2019-09/index.jsp?topic=/org.eclipse.mat.ui.help/welcome.html.

[3] T. Latzo, R. Palutke, and F. Freiling. "A universal taxonomy and survey of forensic memory acquisition techniques." In: *Digital Investigation* 28 (2019), pp. 56–69. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2019.01.001.

[4] S. Leppert. "Android Memory Dump Analysis." Student Research Paper. Friedrich-Alexander University Erlangen-Nuremberg, May 2012.

[5] LibVMI. *Introduction to LibVMI*. accessed on 05.12.2019. URL: http://libvmi.com/docs/gcode-intro.html.

[6] K. Lorenc. *MakeELF Readme*. accessed on 08.12.2019. URL: https://github.com/v3l0c1r4pt0r/makeelf.

[7] H. Macht. "Live Memory Forensics on Android with Volatility." Diploma Thesis. Friedrich-Alexander University Erlangen-Nuremberg, Jan. 2013.

[8] Oracle. *HPROF: A Heap/CPU Profiling Tool*. accessed on 05.03.2020. URL: https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

[9] A. T. Pridgen. "Exploiting Generational Garbage Collection: Using Data Remnants to Improve Memory Analysis and Digital Forensics." Thesis. Rice University, Jan. 2017.

[10] Z. A. Al-Sharif, M. I. Al-Saleh, L. M. Alawneh, Y. I. Jararweh, and B. Gupta. "Live forensics of software attacks on cyber–physical systems." In: *Future Generation Computer Systems* (July 2018). ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.07.028.

[11] Sun Microsystems Inc. *The Java Tutorial - Scope*. accessed on 08.12.2019. URL: https://www.math.uni-hamburg.de/doc/java/tutorial/java/nutsandbolts/scope.html.

[12] The Volatiliy Foundation. *Volatility Wiki*. accessed on 07.12.2019. URL: https://github.com/volatilityfoundation/volatility/wiki/.

[13] D. F. R. Workshop. *A Road Map for Digital Forensics Research*. 2001.

[8] https://lief.quarkslab.com/

[9] https://github.com/Gallopsled/pwntools