Picking up the trash: Exploiting generational GC for memory analysis

Adam Pridgen a, *, Simson Garfinkel b, Dan S. Wallach a

a Rice University, Houston, TX, USA
b George Mason University, Fairfax, VA, USA

A R T I C L E  I N F O
Article history:
Received 26 January 2017
Accepted 26 January 2017

Keywords:
Memory forensics
Malware analysis
Java
HotSpot JVM
Managed runtimes

A B S T R A C T
Memory analysis is slowly moving up the software stack. Early analysis efforts focused on core OS structures and services. As this field evolves, more information becomes accessible because analysis tools can build on foundational frameworks like Volatility and Rekall. This paper demonstrates and establishes memory analysis techniques for managed runtimes, namely the HotSpot Java Virtual Machine (JVM). We exploit the fact that residual artifacts remain in the JVM’s heap to create basic timelines, reconstruct objects, and extract contextual information. These artifacts exist because the JVM copies objects from one place to another during garbage collection and fails to overwrite old data in a timely manner. This work focuses on the HotSpot JVM, but it can be generalized to other managed run-times like Microsoft .Net or Google’s V8 JavaScript Engine.

2017 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

Memory analysis can yield important information when performing forensic analysis as a part of incident response, but it can also be extremely tedious. Several factors hinder memory forensics. First, an analyst requires tools or some understanding about how to extract and interpret the data structures supporting the program. Second, these data structures might be incomplete, overwritten or missing. Finally, the amount of data extracted from memory and its creation order can be impossible to know for certain.

Standard memory analysis frameworks like Rekall and Volatility focus on recovering forensic information from OS structures and services. Conversely, when dealing with a garbage-collected/managed runtime memory system, the interpretation of recovered memory objects depends not on the host machine’s architecture or operating system, but on the particularities of the managed runtime implementation. As more applications are written in language like Java, Python, or Microsoft’s .Net languages, using garbage collection to manage their memory, threads, and other system state, it becomes increasingly important for forensics tools to address these systems.

Furthermore, attackers are increasingly crafting exploits for code running within managed runtimes, delivering code-injection attacks against a variety of services. Forensic tools must then connect low-level kernel state with high-level object state to present a coherent picture of the attacker at work.

This research builds on our previous work exploring JVM data retention through the observation and measurement of latent artifacts in the heap (Pridgen et al., 2017). Here we present JVM tools that we built to rapidly analyze an obfuscated malware. Next, we demonstrate that evidence of sockets and other important artifacts can be recovered from residual data in the Java heap, even if they are not present in the operating system. We focus on the HotSpot Java Virtual Machines because it has been widely adopted within the enterprise and is a vector of current attacks against a number of industries.

The remainder of this paper is organized as follows: “Prior work” Section focuses on related work and past research, and “HotSpot memory background” Section discusses how memory is organized in the HotSpot JVM. “Approach” Section talks about the process of recovering Java objects and low-level object pointers (e.g. Original Object Pointers (OOPs)) from the HotSpot JVM. “Evaluation” Section shows how this can be applied to memory forensics and malware analysis. “Future work” Section expands on future needs for this project. “Conclusions” Section concludes.

Prior work

A large body of work has established usable techniques for copying memory, including Halderman et al. (2009)’s “cold-boot attack,” direct memory access (DMA), FireWire, JTAG, and specially constructed interface cards can perform DMA; VoMel and Freiling...
survey such techniques for acquiring main memory in computers running Microsoft Windows.

The two most common forensic frameworks for decoding memory dumps are the Volatility Memory Forensics Framework (Walters, 2007) and Rekall Memory Forensics Framework (Cohen, 2014). These frameworks are written in Python and implement plugins for specific functions such as listing valid processes and open network connections. Separately, researchers have demonstrated special-purpose memory analysis tools for rendering the pieces of documents that remain in an application’s memory (Saltaformaggio et al., 2014), recovering Android GUIs from apps (Saltaformaggio et al., 2015a), and recovering photographic images that were shown in the view finder, even if they were never written to storage (Saltaformaggio et al., 2015b).

Viega (2001) identified that memory is not securely deallocated in not only C, C++, but also in systems with managed runtimes, such as Java, and Python, potentially allowing sensitive information to be recovered. Chow et al. (2004, 2005) showed that Unix operating systems and standard libraries failed to sanitize deallocated memory; attackers could exploit this issue to recover latent secrets from common applications like Apache and OpenSSH. Li (2015) shows that sensitive information from the Python runtime is easily retrievable, but some Java has similar issues. Forensic analysts can potentially recover copies of Java objects long after the active objects have been garbage collected and overwritten. Such objects might contain sensitive data related to service and user accounts, financial data, or network artifacts left behind by attackers. Forensic analysts can use such objects for establishing an event timeline, looking for evidence related to compromises, understanding the behavior of malicious software, and enumerating compromised data. Pridgen et al. (2017) showed that the JVM fails to overwrite garbage-collected objects, potentially allowing the recovery of TLS secrets long after the TLS connection has been terminated.

Many researchers have demonstrated techniques for recovering usable latent secrets from dump files or system memory. For example, Harrison and Xu (2007) identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux kernel bug. Halderman et al. (2009) showed that AES encryption keys can be readily detected in RAM from their key schedule. Case (2011) presented an approach for analyzing the contents of the Dalvik virtual machine. Similar attacks are possible against Android smartphones, allowing for the recovery of disk encryption keys (Müller and Spreitzenbarth, 2013) and Dalvik VM memory structures (Hilgers et al., 2014).

This article is predicated on the assumption that an attacker or forensic examiner has somehow found a way to capture an unencrypted system memory image; based on our survey and direct experience, we believe that this threat is credible.

HotSpot memory background

The HotSpot JVM implements several different garbage collectors, but all use generational copying to improve memory management performance.

In generational copying, new objects are created in the Eden space. The Eden space is further partitioned into thread local allocation buffers (TLAB), allowing for low-cost memory allocation with minimal locking in multi-threaded applications. The generational hypothesis states that most objects die young; indeed, most Java objects die quickly, but some age and survive GC, and are migrated from Eden to the survivor spaces, which together with the Eden Space are called the young generation. Objects are eventually copied from the young generation to the tenured generation. A typical Java heap memory layout contains many such sections (Fig. 1a).

Because of its focus on performance, the JVM does not clear the contents of memory when an object is moved from one space to another (Sun Microsystems, 2006). Stale data will eventually be overwritten as memory is reused, but these overwrites may also never happen.

In newer HotSpot JVM’s, an alternative Garbage First Garbage Collector (G1GC) uses a partitioned heap space (Fig. 1b), allowing parallel garbage collection during incremental collection. During an incremental collection, G1GC identifies regions with the most garbage and copies the objects into a new region, allowing it to reclaim those regions (Detlefs et al., 2004).

Java runtime performance typically improves when the JVM is given additional RAM, as less memory pressure results in more flexibility for the garbage collector. This decreased pressure also results in latent secrets remaining longer in RAM, improving the chances of recovering sensitive information—a boon for forensic analysis.

Furthermore, the HotSpot JVM uses a region-based memory allocator to manage the sharing of large blocks of memory between the garbage collector and native C libraries. This creates the additional possibility that a garbage collector, finished with a region, might release it to the region allocator, which could then reuse the memory without first zeroing it.

Java objects are variably sized. The invariant part of the object structure includes a mark header, object metadata, and class information. The variable portion contains object’s non-static fields. Raw pointers otherwise known as original object pointers (OOPs) refer to the address of the Java object in process memory, which lies in the heap.

The mark header usually starts the structure and includes the hash identifier of the object, thread ownership information, and metadata (e.g. age and liveness) used by the GC. This mark header is typically followed by a pointer to a type definition (which HotSpot calls a Klass). The type pointer defines each offset necessary to access fields of the object in the heap. If the object fields are primitive values, then these values are written directly into that memory location. If the field is a reference, then the field value is an OOP pointing to the object.
Array objects have a slightly different structure. In addition to the mark header and type information, the object also contains metadata defining dimensionality and the number of elements in the array. The size of an array object also depends on the type (e.g., Byte[], char[], int[]). The Byte[] is an array of OOPs, while the char[] and int[] are arrays of 2- and 4-byte values, respectively.

Fig. 2 depicts the heap memory layout of a String, which is actually an array of two Object references, each pointing to a String. The first element contains a reference to a value of type char[5]. The values for the char[] are inline. Needless to say, these basic object header structures are kept very simple, because they will be widely repeated in memory.

**Approach**

Our memory analysis approach focuses on both the virtual machine and the managed memory. Our analysis components rely on a simple overlay system for data structure interpretation and a simple system for accessing memory using the process’s virtual addressing scheme. Our analysis framework, RecOOP, is written in Python and can be used with an interactive environment like IPython or as a library like Rekall.

Fig. 3 depicts the process we use to recover objects from managed memory. Currently, our RecOOP analysis focuses on recovering HotSpot JVM OOPs from ×86 architectures. Adding support for 64-bit machines would only require minor modifications to address the OOP encoding. We similarly expect that our work would generalize to support other managed runtime environments such as those used by Mono, .Net, or JavaScript.

We implemented our analysis for the Linux and the Windows operating systems. We have successfully tested RecOOP against 32-bit versions of Ubuntu, Windows XP SP3, Windows 7, and Windows 8 with a Java heap size of 2GiB. Overlays are structural templates used to interpret raw memory as a program data structure. Only 8 out of 150 C++ overlays require different padding to achieve the correct memory layout on the different OSs. We believe these differences are due to compilers, which tend to vary field padding in the structures.

**Process reconstruction**

RecOOP analysis begins with process reconstruction. If the process’s memory has not already been extracted from a RAM image, RecOOP will dump it using the Volatility Framework. Volatility will identify the target process by name or PID and enumerate all the physical memory page frames, which are then ordered according to the process’s virtual address space. Finally, the memory is saved to file for future analysis or for use with other tools such as Radare (Pancake, 2015).

**Extract loaded classes**

Program portability in managed runtimes is accomplished through several key systems that resolve, link, and sometimes compile the program being loaded and executed. Internally, there is a loader and type system used to find and load specific classes or types, and then store these types for future reference. The loader will look inside the application or loading path to find the correct library.

When the required types, classes, and code are loaded into the virtual memory, symbols for each of these artifacts are created. Once completed, the runtime then links together the code for each of the classes for the given types. Linking ensures that all class dependencies for inter-class method calls and field access are loaded. Linking also optimizes method calls in classes that implement an interface. For example, if class Boingo implements Boingo, the links to the Boingo methods need to be created to reduce any performance penalties when Boingo is treated as a Boingo. After linking and loading, the original class file defining the Java types and code are transformed into machine optimized structures. These transformations are with their symbolic references in a central location.

The HotSpot JVM stores requisite information in three different hash tables: a SystemDictionary, a SymbolTable, and a StringTable. The SystemDictionary contains all the loaded type information (e.g., Java classes). The SymbolTable contains all the loaded symbols for classes, methods, fields, and enumerable types. Finally, the StringTable contains all the constant strings or strings that exist for long periods of time. Generally, only the types required for linking are resolved and loaded into the runtime; this proves useful when with dealing obfuscated JAR files, because forensic or malware analysis need only focus on the loaded class files and types.

Our JVM analysis engine first looks for the symbol table and then the system dictionary. The symbol table is a good place to...
begin, both because it’s structurally simple and because those strings will be helpful to us later.

These data structures are located by scanning for invariant values (0x00004e2b or 0x000003f1) in the C++ _table_size field of the structure. When these values are found, _entry_size and _number_of_entries are used for an initial sanity check. The _entry_size is the size in bytes for each value entry (e.g. 1 entry = 0 × 000C bytes). We place an upper bound on _number_of_entries that starts at 100 K entries but can be adjusted if necessary. Table 1 shows the memory layout of a system dictionary that we want to apply these constraints to.

When these constraints are met, the engine attempts to parse a subset of the hash table entries. The system dictionary has a pointer to these entries (e.g. HashTableBuckets* _buckets); the internal array that forms the spine of the hashtable.

The engine iterates over this array and tries to follow the bucket entries to the target value using a memory overlay. If the entries can be interpreted as the expected values, we accept it as valid. For example, the C++ type Symbol is a vtable, followed by some metadata, size, and the symbol string. As a heuristic, if the length of this string exceeds a manually-chosen threshold, the entry is considered invalid.

Table 2 shows some entries from a valid dictionary found by the JVM analysis engine. Most Klass structures should have symbol names appearing in the symbol table, so when we parse candidate dictionaries, the dictionary entries (e.g. Klass *) names are checked to see if they are known symbols. If a majority of these symbols are found, we accept the candidate. The product of this analysis yields the low-level memory layout of each Java class, methods, and other meta-data like the Java constant pool. (Note: since the JVM supports class unloading, unloaded Klass names may not be present in the SymbolTable, even when dead objects of those types are still in heap pages waiting to be reused.)

This extraction technique is OS agnostic and easily automated.

Alternative approach: the JVM tool interface. Prior to developing these techniques, we attempted to use symbol structures intended for the JVM tool interface (JVMTI). These structures were found using string pointers to the symbol names, the structure size, static values in the fields, and the location relative to the JVM library’s base offset. When the JVM is started, these structures are filled with the appropriate runtime data structures (e.g. SystemDictionary, SymbolTable). We also used an optimization technique to find the best fit memory locations based on the locations of other recovered structures, the order of the structures, and invariant values that should be present in the structure.

Unfortunately, this approach has many obstacles. First, it was overly complex; the identification of strings and reverse-mapping them to pointers was time intensive, and the specific strings and structures were not always present in memory. Second, every version of Java requires a new set of constraints because the location of the JVMTI symbols and data structures change. Thus, this approach could not generalize across multiple platforms and JVM versions. Finally, since these structures and JVMTI symbols were not in use, the OS paged the sections of the process’s virtual memory out to make space for other relevant data. Most memory analysis protocols do not consider the OS swap or page files; thus, if RAM was dumped near the start time of the JVM process, recovery of the dictionary, symbol table, and string table addresses were likely, but after a few minutes the chance of success dissipated.

Identifying managed memory

Knowing the location of managed memory helps with object enumeration and with sanity checking whether or not the results are valid. However, automatically and correctly identifying these segments is difficult. We err on the side of caution and
enumerate all possible locations. To prevent erroneous object identification, we perform type checking on every object’s non-primitive field references.

Potential managed memory areas are found by looking for an abundance of type-pointers (e.g. Klass*). Every object is required to have a defined type. Consequently, areas with a large number of type-pointers are likely to contain objects. The exceptions to this rule are places where class metadata or compiler interface data structures are located. Most of the class metadata is known, so these addresses are filtered out. For other areas of memory, we rely on our type checking to remove invalid entries.

We isolate managed memory boundaries by first ignoring all memory regions less than 256 KiB, since this is less than the smallest default heap space. Second, we only consider pages with more than 10 type-pointers, and then we smooth variations using a moving average. We only consider areas with more then 10 type-pointers in at least 32 consecutive pages (e.g. 32 \( \times 4096 \) B = 64 KiB). This algorithm might need adjustment for G1GC, because G1GC uses humongous memory regions (e.g. large allocations exceeding multiple MiB) for large objects.

This analysis only establishes boundaries where objects might be clustered. We avoid identifying memory regions as a particular generational space (e.g. Eden, survivor, etc.), as this could lead to misclassifications. For example, the JVM may expand or contract a heap space depending on application activity. Using our analysis, we might misclassify two segments of memory as different spaces, when they were part of the same region at some point in time. A consequence of this misclassification might also lead us to miss regions where Java objects are present.

The JVM is very good at measuring performance and logging events, so if identifying generations is necessary, it still might be possible to do so without using type-pointers. If at least one GC event has occurred, the JVM logs information about all the heap spaces internally. These log strings contain the named heap space, start and end addresses, and other information. These messages can be found by searching a strings dump using regular expressions like “space.*used|Metaspace.*used.” This expression will reveal most, if not all, of the managed memory spaces used by the Java heap.

To demonstrate these procedures, we analyze data from the Adwind malware analysis case study in the next section. Table 3 shows the most relevant information with an emphasis on the heap space and memory region. The color of the heap space is also reflected in Table 4, which shows the results of the type-pointer clustering. Using type-pointers narrows the number of memory regions that need to be scanned from 431 to 11, and it isolates the areas where objects might exist. The sparklines show several memory chunks that contain some of these regions, most obviously in the 0xa47f0000-0xa4c0f000, 0xa4c0f000-0xa4d50000, and 0xa9d50000-0xaaa0000 memory chunks. If we want more granular heap information, additional memory analysis is required.

### Table 3

<table>
<thead>
<tr>
<th>GC Log Message</th>
<th>Start and end of the space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generational space</td>
<td>[...] used</td>
</tr>
<tr>
<td>eden space</td>
<td>[0xa4d880000, [...] 0xa4d500000)</td>
</tr>
<tr>
<td>from space</td>
<td>[0xa4c0f0000, [...] 0xa4c0d8000)</td>
</tr>
<tr>
<td>to space</td>
<td>[0xa4c0d8000, [...] 0xa4d500000)</td>
</tr>
<tr>
<td>the space</td>
<td>[0xa9d500000, [...] 0xaaa00000)</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>Address range</th>
<th>Type pointers</th>
<th>Unique pointers</th>
<th>Pointer occurrences per page (Y-axis: 0–64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8a12e000-0x9a3bf000</td>
<td>1353</td>
<td>265</td>
<td>0x8a13e000-0x9a3bf000</td>
</tr>
<tr>
<td>0x8a81e000-0x9a47f000</td>
<td>2735</td>
<td>331</td>
<td>0x8a94e000-0x9a47f000</td>
</tr>
<tr>
<td>0x8a360000-0x9a3692000</td>
<td>362</td>
<td>360</td>
<td>0x8a400000-0x9a477f9000</td>
</tr>
<tr>
<td>0x8a47f000-0x9a4c0f000</td>
<td>13261</td>
<td>266</td>
<td>0x8a4c9f000-0x9a4c92000</td>
</tr>
<tr>
<td>0x8a4c5000-0x9a4c92000</td>
<td>129</td>
<td>28</td>
<td>0x8a4d1d000-0x9a4df8000</td>
</tr>
<tr>
<td>0x8a4d1c000-0x9a4d50000</td>
<td>1121</td>
<td>79</td>
<td>0x8a9d5000-0x9a9d50000</td>
</tr>
<tr>
<td>0xb6936000-0xb6996000</td>
<td>28810</td>
<td>661</td>
<td>0xcb693600-0xcb6996000</td>
</tr>
<tr>
<td>0x0c001000-0x0c7f8f000</td>
<td>11085</td>
<td>1211</td>
<td>0x0c001000-0x0c7f8f000</td>
</tr>
</tbody>
</table>

### Enumerate and extract objects

The location of type-pointers is used to help object enumeration and extraction. Enumeration for objects like threads, sockets, and files happens automatically, but RecOOP permits enumerating specific types of objects any time after the managed memory is identified. In the previous phase, all the addresses to type-pointers were found and saved. These addresses are used to extract objects if they fall in a managed memory boundary.

Object extraction happens in several phases. First, we check that the address adheres to the basic object structure. Next, we use the loaded type information to determine the size of the object and locate its references. Then, each non-primitive field is parsed recursively, repeating these steps. The field references are checked to see whether the value is null or the given type of the field. Note, we also track all the potential classes a reference could be due to polymorphism. After the fields have all been parsed and extracted, all the values in the fields are updated, and the process completes.

Values are set after all the referenced objects are enumerated to avoid an uncontrolled recursion.

**Java threads** (e.g. java.lang.Thread) are enumerated and extracted first. During this process, the native structures implementing the thread are also identified and mined for information. After the initial identification, each thread is checked for validity and fields holding pertinent information are analyzed, most importantly settop, the thread’s native address. We use this field to find the linked list containing all the threads, and we iterate over it to identify any missing threads from the Java heap. If any are found, we repeat the object analysis for the missing thread.

**Buffers and streams** are investigated next because they are typically used to manage IO between the program, the JVM, and the operating system. Given the ubiquitous nature of these objects, there are a number of base and abstract classes (e.g. java.io.InputStream) that are used to create the different IO classes like java.io.BufferedWriter. We were challenged by...
the polymorphism and the number of types an object might implement. For example, determining if a SocketInputStream is used by a java.io.BufferedInputStream requires identifying the java.io.BufferedInputStream that wraps the SocketInputStream. To deal with this issue, we perform multiple scans for the different IO implementations and create a basic link table. This link table helps cut through obscure object relationships to map IO classes with buffers and other data. Generally, we found that the only IO classes containing buffered data were either used by a buffered IO class or the class maintained its own buffer, as was the case with InetAddressInputStream.

Native buffers are used to marshal IO data in and out of the JVM. Classes used for the functionality appear to implement the DirectByteBuffer interface, which permits direct memory access. We have only found the implementations MappedByteBuffer, NativeBuffer, and HeapByteBuffer in the source code. Data in these buffers is captured, but it is volatile and may not be useful.

File information is collected from objects using the java.io.FileDescriptor or java.io.File type. For the most part, the filename or path are the only useful information found in this object type. If there is a reference from an IO object like a FileChannelImpl or FileInputStream, we might be able to determine whether or not the file is open. If buffering is not used by the IO stream, identifying any attributable data is difficult.

JAR files and entries contain information related to loaded files and might reveal sensitive information by way of compressed streams. JAR files typically hold all the program resources and class files for a library or program. Class files are decompressed and loaded from JAR files as a ZipEntry object. Usually, decompression and loading happen in lockstep, so any data related to the process may dissipate very quickly. When raw compressed data is present in memory, a zip library may be able to decompress it. We have been able to successfully recover JAR filenames, named entries, and decompressed entries. If parts of the JAR file are present in memory, we read the low-level zip file structure, dump the resident data, and investigate the result as a zip file.

Socket objects can reveal connections well after the artifacts disappear in the OS. In particular, the IP address along with the port number is useful. We also attempt to associate the socket with any identified streams and data buffers.

Child process information is collected from the ProcessBuilder and OS Process implementation class. The ProcessBuilder class is the typical way to start a process. This class takes a command string or an array of strings in addition to any object for redirecting IO. Once the process is started, an OS-specific implementation of a Process object is created. Unfortunately, when GC happens, the string objects used for the command string are likely to be overwritten. These overwrites can prevent identifying the command by name.

The Process object remains in the heap for a significant period of time. In our experiments, even though GC happened several times, all Process objects were still recoverable. Additionally, the IO buffers stdout, stdin, and stderr retained some data. Even though the information used in the original instantiation of the process dissipated, we could use the process output to identify some of the processes.

One of the benefits of a managed runtime for forensic analysis is event ordering. In the HotSpot JVM, memory is allocated from the heap or TLABS directly after the last allocation; this sequential allocation is a fundamental property of a wide variety of garbage collection strategies. Because the TLABS are thread-local, objects allocated sequentially by a thread will likely be adjacent in memory, regardless of memory allocation activity by other threads.

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Thread information extracted from the HotSpot JVM executing the Adwind malware on Linux.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread identifier</td>
<td>Native address</td>
</tr>
<tr>
<td>1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>1</td>
<td>0x86907000</td>
</tr>
<tr>
<td>1</td>
<td>0x8695f800</td>
</tr>
<tr>
<td>1</td>
<td>0x8695f800</td>
</tr>
<tr>
<td>3</td>
<td>0x86f61000</td>
</tr>
<tr>
<td>3</td>
<td>0x86f61000</td>
</tr>
<tr>
<td>4</td>
<td>0x86f70000</td>
</tr>
<tr>
<td>4</td>
<td>0x86f70000</td>
</tr>
<tr>
<td>5</td>
<td>0x86f7f800</td>
</tr>
<tr>
<td>5</td>
<td>0x86f78000</td>
</tr>
<tr>
<td>6</td>
<td>0x86f92c00</td>
</tr>
<tr>
<td>6</td>
<td>0x86f92c00</td>
</tr>
<tr>
<td>7</td>
<td>0x86f3a00c</td>
</tr>
<tr>
<td>7</td>
<td>0x86f3a00c</td>
</tr>
<tr>
<td>8</td>
<td>0x86f000000</td>
</tr>
<tr>
<td>9</td>
<td>0x86f3100c</td>
</tr>
<tr>
<td>9</td>
<td>0x86f3100c</td>
</tr>
<tr>
<td>10</td>
<td>0x86f36f800</td>
</tr>
<tr>
<td>10</td>
<td>0x86f36f800</td>
</tr>
<tr>
<td>11</td>
<td>0x86f924c00</td>
</tr>
<tr>
<td>12</td>
<td>0x86f907000</td>
</tr>
<tr>
<td>14</td>
<td>0x86f350000</td>
</tr>
</tbody>
</table>

This ordering lends itself well to timelines and trying to determine the relationships between events.

Evaluation

Three case studies demonstrate RecOOPs ability to extract information from a Java runtime. In each case, only a memory image is available for analysis. Traditionally, understanding Java malware beyond sandboxing and behavioral analysis requires two things: the JAR file and a decompilation tool such as CFR or JD-GUI. The analyst decompiles the JAR file, modifies the code, recompiles, and runs the code in an IDE such as Eclipse (Chen and Chen, 2006; Proebsting and Watterson, 1997; Cimato et al., 2005). Obfuscation tricks can be very effective at blocking these efforts (Chan and Yang, 2004; Low, 1998; Schlumberger et al., 2012), and if the malware removes itself from the disk, extraordinary efforts are required to recover the original file, which may not be feasible. In this section, we show that Java processes contain copious amounts of information which lends itself to static forensic analysis.

<table>
<thead>
<tr>
<th>Table 6</th>
<th>Compressed class (gray), password (red), and other files found in the Java heap.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zip Index</td>
<td>Name</td>
</tr>
<tr>
<td>0xa6f4f66</td>
<td>181</td>
</tr>
<tr>
<td>0xa6f91202</td>
<td>170</td>
</tr>
<tr>
<td>0xa6f24c20</td>
<td>15</td>
</tr>
<tr>
<td>0xa6f2c590</td>
<td>170</td>
</tr>
<tr>
<td>0xa6f2c510</td>
<td>477</td>
</tr>
<tr>
<td>0xa6f8c50</td>
<td>819</td>
</tr>
<tr>
<td>0xa6f8e28</td>
<td>157</td>
</tr>
<tr>
<td>0xa6f8e600</td>
<td>157</td>
</tr>
<tr>
<td>0xa6f8e648</td>
<td>157</td>
</tr>
</tbody>
</table>
Blackbox malware analysis and reverse engineering

We found an old version of the Adwind trojan on Malwr.org1 and performed a Java centric analysis. We ran the malware on both Linux and Windows XP SP3 VMs and found that the malware appears to behave a little differently on Linux. Both versions of Java produce a similar thread listing (Table 5). However, the program behaviors diverge because the backdoor must dump a native library that is used by Java for snooping and keystroke logging.

Since this malware uses obfuscation, we explore the process for any latent buffers containing compressed data. Table 6 shows that files can either reveal information like passwords or contain unobfuscated class files. Table 7 shows several recovered class interfaces, and Listing 1 shows a high level prototype of a recovered class file created by Radare. The extra/CLM.pass reveals a password field, so enumerating the object and its field in the heap reveals the string value (vooXN3UW). Finding this value in a strings dump would be difficult.

Malware proxy

To demonstrate the effectiveness of socket analysis, we wrote a program that simulates the basic capabilities of Java malware. In this case, an infection has been detected in the network, and an investigation of the system reveals malware acting as a network proxy. This proxy allows an external attacker to communicate with hosts on the internal network. Normally, the investigator may not find out what information moved in and out of the network. However, Java's memory model allows the socket connections and buffered data to persist indefinitely.

We ran the simulation for five minutes, sending commands instructing the agents to "do something evil." Table 8 shows the recovered socket data. Since the buffered stream and subordinate objects were never collected or overwritten, most of the attackers commands remained intact. However, we found that some of the structural information of the messages was lost, because the proxy used DataInputStream to read the message length and command

---

1 Windows analysis from Malwr.org: [https://malwr.com/analysis/NGUwYjM1OGI4MGE4NDZkYjg5ZGVyMGJ1YTYMyN2RIMDU/](https://malwr.com/analysis/NGUwYjM1OGI4MGE4NDZkYjg5ZGVyMGJ1YTYMyN2RIMDU/)
The plugin activates, it uses administrator credentials to upload and activate the plugin. When uploaded to a dotCMS server. In this case the attacker leverages around an attacker exploiting the fact that dynamic plugins can be would behave in a post-compromise setting. The scenario centers Scripted intrusion

Listing 1. Rader2 java prototypes command reveals a custom loader in the decompressed class data.

decompressed class data.

Table 9

<table>
<thead>
<tr>
<th>Event Log from Extracted Process Builder and Process Information</th>
<th>Address</th>
<th>Process Builder Command</th>
<th>PID</th>
<th>Buffered Data</th>
</tr>
</thead>
</table>
| 0x91e8728e | subo cat /etc/subo | 1242 | g|start This file MUST be edited w | [...
| 0x91e67ee8 | unanne w | 1245 | Linux java-workx2-00:3:19:0-1 | [...
| 0x91e67e88 | id | 1247 | java | [...
| 0x91e67f00 | id | 1248 | java adm droid sudo dipp plugl | [...
| 0x91e68388 | subo cat /etc/passed | 1250 | root:0:360:root:/root/tas | [...
| 0x91e68540 | subo cat /etc/shadow/ | 1252 | root:16676:0:99999:7:1668:7::1 | [...
| 0x91e68a00 | subo netatap | 1273 | total 10312|d|nrwxrwxr | [...
| 0x91e68a80 | le-all | 1273 | total 10312|d|nrwxrwxr | [...
| 0x91e68e88 | sysfs | 1298 | sysfs on /sys type sysfs (rw,n | [...
| 0x91f5e7cc | cat /etc/bash_history | 1300 | history | [...
| 0x91f5e838 | cat /home/java/bash_history | 1307 | ifconfig sudo add-apt-repo | [...
| 0x92029a78 | ps -ef | 1322 | adding home/java/shub | [...
| 0x9202b760 | sig-r | 1323 | adding home/java/shub | [...
| 0x9207b760 | ps -ef | 1336 | total 48|d|nrwxr | [...
| 0x9208aa88 | cat /home/bakup.dat | 1337 | PKA|d|nrwxr | [...
| 0x920b7718 | le-all | 1338 | root root /root/bin/bash | [...
| 0x920b5d48 | dd bsdev/var | 1338 | root root /root/bin/bash | [...
| 0x920d8d00 | sig-r | 1344 | adding var/log | [...
| 0x920e5c50 | ps -ef | 1345 | adding var/log | [...

Table 10

<table>
<thead>
<tr>
<th>Interesting process output recovered at t = 35.</th>
<th>Address</th>
<th>PID</th>
<th>Buffered output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x96700200</td>
<td>1275</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 0x967002c0 | 1403 | total 175684|d|nrwxrwxr | [...
| 0x967002e0 | 1273 | total 10312|d|nrwxrwxr | [...
| 0x967002f0 | 1252 | root:/16676:0:99999:7:1668:7::1 | [...
| 0x96700350 | 1250 | root://0:root:/root/bin/bash | [...
| 0x967003b8 | 1248 | java adm droid sudo dipp plugl | [...
| 0x96700720 | 1245 | Linux java-workx2-00:3:19:0-1 | [...
| 0x96700c80 | 1242 | # This file MUST be edited w | [...
| 0x96700e18 | 1354 | adding var/log | [...
| 0x96700e38 | 1338 | rsrv-1 root root /root/bin/bash | [...
| 0x96700e70 | 1333 | PKA|d|nrwxr | [...
| 0x96700ee0 | 1228 | total 44|d|nrwxrwxr | [...
| 0x96701d78 | 1222 | adding home/java/shub | [...
| 0x96716c80 | 1307 | ifconfig sudo add-apt-repo | [...
| 0x96714c10 | 1301 | history | [...
| 0x96714eb8 | 1298 | sysfs on java type sysfs | [...

implant relies on ProcessBuilder to execute system commands outside of the Java environment and has functions that allow the attacker to proxy and communicate with other systems, read and write files, download files, and interact with the OS.

This evaluation concentrates on the created processes, and how much information can be gleaned from their Java artifacts. This script starts 73 processes that execute system commands (e.g., ls, ps, etc.)

Fig. 4 shows how much command history is retained in the heap over time. Three garbage collection cycles are observable at t = 25, t = 27, and t = 30. Malware data exfiltration triggers the GC events.

The process objects accumulate and remain in memory even after several garbage collections. These processes also retain buffered data, which can be used to infer the commands executed on the system. Table 10 shows a sample of these buffered processes at the end of the experiment. We can see that an analyst could infer what 11 out of the 16 listed processes were doing. To verify this information, we refer back to t = 24 before the garbage collection wiped out most of the command history. Table 9 shows how the ProcessBuilder objects and the Process data buffers can be used to assemble an event log.

We also evaluate recovery of process artifacts from Volatility. For each memory image, the linux_psview and linux_psview-x are used to try and find artifacts. Only one process (sudo lsodf) could be accurately identified. This process runs from t = 10 through the end of the experiment. No other process artifacts could be recovered. We believe they were unrecoverable due to OS activity and memory volatility, because the relevant data structures were overwritten at some point after process termination and memory

---

2 http://dotcms.com/.
deallocation. This shows the limits Volatility and other similar frameworks, which do not currently account for runtime artifacts. The HotSpot JVM produces telemetric data to help improve performance. MethodCounters are initialized on a method’s first call, and it tracks the number of calls, which can be useful for malware analysis (Table 11). For example, if malware uses a HashMap to map specific commands to specific functions, understanding the malware’s behavior is very challenging. The telemetric information helps discern relevant functions from noise or potential obfuscation.

Future work

Future work should address several key challenges. First, our analysis tools can use a significant amount of memory while processing a malware memory image. Overhead results from creating environmental objects and values in addition to annotations which help support our analyses. The resulting memory consumption becomes an issue when memory image sizes are multiple gigabytes or when there are 100,000 or more individual objects.

Furthermore there is a semantic gap between some objects that prevents directly finding links between these objects without deeper analysis. At runtime, relevant information about an object can be determined with an API call. When the memory is analyzed in a static manner, those API calls are not available (even though the information that they use is typically in memory). This was the case with JAR file entry names and the compressed data from the entry in our experiment: A symbolic execution for VM bytecode (e.g. CLR, HotSpot, etc.) would help to eliminate this problem.

Conclusions

RecOOP is a memory analysis framework that helps generalize digital forensics of managed runtimes. We developed an implementation focused on the HotSpot JVM for Java 8. We also showed that the framework is practical for digital forensics and malware analysis, complementing other such tools.

Acknowledgments

Special thanks to the The Cover of Night, LLC for providing the hardware and infrastructure necessary for performing our experiments. This research was supported in part by NSF grants CNS-1409401 and CNS-1314492 and a National Physical Science Consortium Fellowship.

References


Li, Y., 2015. Where in Your Ram is “python san_diego.py”? URL: https://www.youtube.com/watch?v=sMIoxcXe-00.


