Digital Investigation 20 (2017) S20–S28

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 Europe — Proceedings of the Fourth Annual DFRWS Europe

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

ARTICLE INFO

Article history:
Received 26 January 2017
Accepted 26 January 2017

Keywords:
Memory forensics
Malware analysis
Java
HotSpot JVM
Managed runtimes

ABSTRACT

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.

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 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; VöMel and Freiling

---

* Corresponding author.
E-mail address: dso@rice.edu (A. Pridgen).

http://dx.doi.org/10.1016/j.diin.2017.01.002

(2011) 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, and 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.

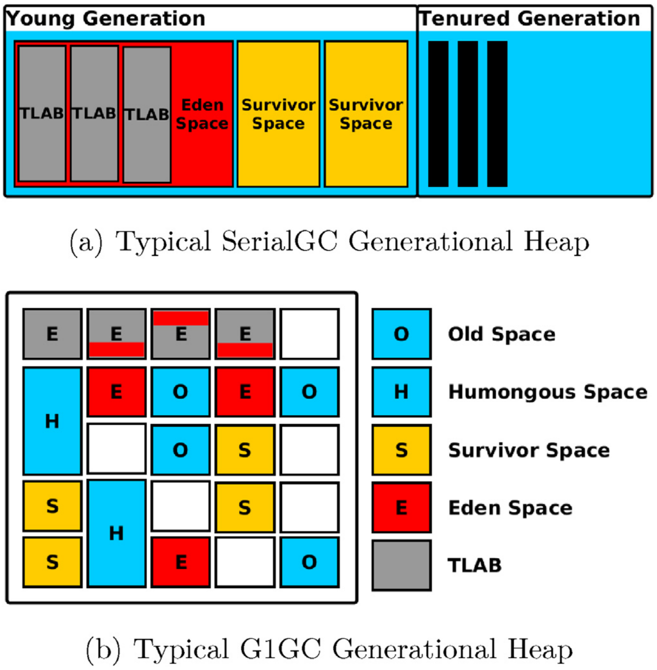
Many researchers have demonstrated techniques for recovering usable latent secrets from dump files or system memory. For example, Harrison and Xu (2007) identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux 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).



(a) Typical SerialGC Generational Heap

(b) Typical G1GC Generational Heap

Fig. 1. Heap memory layouts used by the HotSpot JVM.

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[]` vs. `char[]` vs. `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[2]`, 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 inlined. 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.

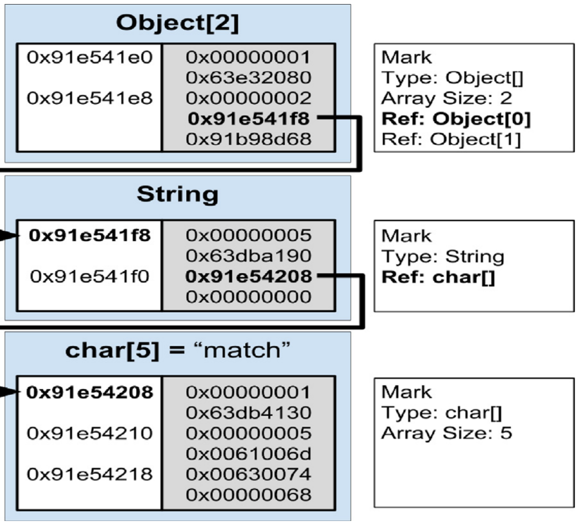


Fig. 2. This diagram shows the Java heap memory layout when examining OOPs. Here we show a `java/lang/String` referenced from `java/lang/String[2]`.

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 Foo implements Boingo`, the links to the `Boingo` methods need to be created to reduce any performance penalties when `Foo` 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

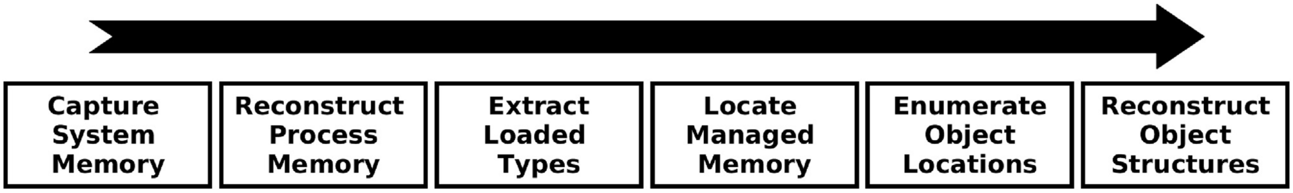


Fig. 3. An overview of the steps that RecOOP takes to extract and recover managed memory objects for forensic analysis.

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 threshhold, 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

Table 1
The memory layout of a JVM `SystemDictionary` captured from an embedded Windows 7 OS instance.

Memory		Data Structure Interpretation of the `SystemDictionary::_dictionary`
Address	Values	
`0x00e5b928`	**0x00004e2b**	[struct_field: int table_size]
`0x00e5b92c`	`0x00e5cd50`	[struct_field: SymbolTableBucket* buckets]
`0x00e5b930`	`0x00000000`	[struct_field: SymbolTableEntry* free_list]
`0x00e5b934`	`0x14283408`	[struct_field: char* first_free_entry]
`0x00e5b938`	`0x14283be0`	[struct_field: char* end_block]
`0x00e5b93c`	**0x0000000c**	[struct_field: int entry_size]
`0x00e5b940`	**0x00002f4d**	[struct_field: int number_of_entries]

Table 2
A memory dump showing the offsets and values embedded Windows 7 OS instance.

Memory			Data Structure Interpretation	
Offset	Address	Values	Value information	`SystemDictionary::_dictionary` entry values
`0x00`	`0x142d61b4`	`0x0b32967d`		[struct_field: int _hash]
	`0x142d61b8`	`0x00000000`		[struct_field: DictionaryEntry* _next]
	`0x142d61bc`	**0x13fdc908**	Klass*: java/nio/channels/ByteChannel	[struct_field: Klass* _literal]
	`0x142d61c0`	`0x00000000`		
	`0x142d61c4`	`0x00e5cd18`	[→ oop class_loader]	
`0x14`	`0x142d61c8`	`0x257f6796`		[struct_field: int _hash]
	`0x142d61cc`	`0x00000000`		[struct_field: DictionaryEntry* _next]
	`0x142d61d0`	**0x13fdccb8**	Klass*: java/nio/channels/SeekableByteChannel	[struct_field: Klass* _literal]
	`0x142d61d4`	`0x00000000`		
	`0x142d61d8`	`0x00e5cd18`	[→ oop class_loader]	
	`0x142d61dc`	`0x55f713ed`		[struct_field: int _hash]
`0x28`	`0x142d61e0`	0x00f357f8		[struct_field: DictionaryEntry* _next]
	`0x142d61e4`	**0x13fdcf58**	Klass*: java/nio/channels/GatheringByteChannel	[struct_field: Klass* _literal]
	`0x142d61e8`	`0x00000000`		
	`0x142d61ec`	`0x00e5cd18`	[→ oop class_loader]	

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. 32×4096 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, tenured, 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.

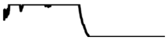
The JVM is very good about measuring performance and logging events, so if identifying generations is necessary, it still might be possible to do so without using type-pointers. If at least one GC event has occurred, the JVM logs information about all the heap spaces internally. These log strings contain 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 `0xa47ff000-0xa4c0f000`, `0xa4cd0000-0xa4d50000`, and `0xa9d50000-0xaa000000` memory chunks. If we want more granular heap information, additional memory analysis is required.

Table 3
The regular expression “space.*used” used in conjunction with ffastrings to determine the eden, survivor, and tenure generation spaces. Note *[...]* signifies omitted message content.

GC Log Message		
Generational space		Start and end of the space
`eden space`	*[...]* used	[`0xa4800000`, [...] `0xa4c50000`)
`from space`	*[...]* used	[`0xa4c50000`, [...] `0xa4cd0000`)
`to space`	*[...]* used	[`0xa4cd0000`, [...] `0xa4d50000`)
`the space`	*[...]* used	[`0xa9d50000`, [...] `0xaa800000`)

Table 4
Number of pointers found in address ranges with more than 10 unique “type-pointer” occurences found on addressable word boundaries in a version of the Adwind malware. The *red*, *yellow*, and *black* lines correspond to the eden, survivor, and tenure space, respectively. Table 3 shows how these locations are found.

Adwind Obfuscated Java Malware on Linux Loaded Classes = 1626			
Address range	Type pointers	Unique pointers	Pointer occurrences per page (Y-axis: 0–64)
`0xa32de000–0xa3355000`	1353	265	
`0xa33ce000–0xa349d000`	2735	331	
`0xa349e000–0xa34f5000`	609	122	
`0xa3600000–0xa3692000`	362	360	
`0xa40b0000–0xa4779000`	11926	1229	
`0xa47ff000–0xa4c0f000`	13261	266	
`0xa4c50000–0xa4c92000`	129	28	
`0xa4cd0000–0xa4d50000`	1121	79	
`0xa9d50000–0xaa000000`	28810	661	
`0xb6936000–0xb6996000`	427	413	
`0xc0001000–0xf7bfe000`	11085	1211	

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 `eetop`, 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.BufferedReader`. 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 `InflaterInputStream`.

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 zlib 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 remote and local port are extracted, if the object is still intact. 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, then objects allocated sequentially by a thread will likely be adjacent in memory, regardless of memory allocation activity by other threads. This ordering lends itself well to timelining and trying to determine the relationships between events.

Table 5
Thread information extracted from the HotSpot JVM executing the Adwind malware on Linux.

Thread identifier	Native address	Heap address	Thread name
1	0x00000000	0xa9e91020	Main
1	0xb6907000	0xa4d3d050	Main
2	0xb695f800	0xa4cd4e10	Reference Handler
2	0xb695f800	0xa9e90c58	Reference Handler
3	0xb6961000	0xa9e90ab0	Finalizer
3	0xb6961000	0xa4cd4c68	Finalizer
4	0xb697e000	0xa9e90938	Signal Dispatcher
4	0xb697e000	0xa4cd4af0	Signal Dispatcher
5	0xb697f800	0xa9e907c0	C1 CompilerThread0
5	0xb697f800	0xa4cd4978	C1 CompilerThread0
6	0xb6982c00	0xa4cd4800	Service Thread
6	0xb6982c00	0xa9e90648	Service Thread
7	0xa360ac00	0xa9e904a8	Java2D Disposer
7	0xa360ac00	0xa4cd4660	Java2D Disposer
8	0x00000000	0xa9e204f8	XToolkt-Shutdown-Thread
9	0xa361dc00	0xa4cd4318	AWT-XAWT
9	0xa361dc00	0xa9e90160	AWT-XAWT
10	0xa36f5800	0xa9e8fef8	Thread-0
10	0xa36f5800	0xa4cd15d8	Thread-0
11	0x09e24c00	0xa49f2720	Thread-1
13	0xb6907000	0xa49f9e58	DestroyJavaVM
14	0x09e35000	0xa49f5a78	pool-1-thread-1

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 6
Compressed class (*gray*), password (*red*), and other files found in the Java heap.

Zip Inflater Address	Size	Decompressed Data
0xa48f46e0	181	\xca\xfe\xba\xbe ...
0xa48ff850	186	\xca\xfe\xba\xbe ...
0xa4901720	433	\xca\xfe\xba\xbe ...
0xa491fcd8	170	Manifest-Version: 1.
0xa4920228	170	Manifest-Version: 1.
0xa4924c20	15	plugins._008_
0xa492c590	170	Manifest-Version: 1.
0xa492cc10	477	\xca\xfe\xba\xbe ...
0xa9d87160	10	vooXN3UW
0xa9e88c50	819	\xca\xfe\xba\xbe ...
0xa9e89600	152	\xca\xfe\xba\xbe ...
0xa9e89f30	607	\xca\xfe\xba\xbe ...
0xa9e8a9c8	117	\xca\xfe\xba\xbe ...
0xa9e8ad70	727	\xca\xfe\xba\xbe ...
0xa9e8b700	1324	\xca\xfe\xba\xbe ...
0xa9e8c440	1008	\xca\xfe\xba\xbe ...
0xa9e8cf50	157	Manifest-Version: 1.
0xa9e8d2e8	157	Manifest-Version: 1.
0xa9e8d680	157	Manifest-Version: 1.
0xa9e8da58	157	Manifest-Version: 1.

Table 7
Extracted class data for Adwind's plugin interface and survey functionality.

Offset	0 1	2 3	4 5	6 7	8 9	A B	C D	E F	0123456789ABCDEF
0x00000000	café	babe	0000	0032	000d	0700	0b07	000c	……2……
0x00000010	0100	0a61	6464	4172	6368	6976	6f01	0011	…addArchivo…
0x00000020	284c	6a61	7661	2f69	6f2f	4669	6c65	3b29	(Ljava/io/File;)
0x00000030	5601	000d	6361	7267	6172	506c	7567	696e	V…cargarPlugin
0x00000040	7301	0003	2829	5a01	000a	6765	7450	6c75	s…()Z…getPlu
0x00000050	6769	6e73	0100	1928	295b	4c70	6c75	6769	gins…()[Lplugi
0x00000060	6e73	2f41	6477	696e	6453	6572	7665	723b	ns/AdwindServer;
0x00000070	0100	0a53	6f75	7263	6546	696c	6501	0015	…SourceFile…
0x00000080	496e	7465	7266	6163	6550	6c75	6769	6e73	InterfacePlugins
0x00000090	2e6a	6176	6101	0018	706c	7567	696e	732f	.java…plugins/
0x000000a0	496e	7465	7266	6163	6550	6c75	6769	6e73	InterfacePlugins
0x00000000	café	babe	0000	0032	000a	0700	0807	0009	……2……
0x00000010	0100	0e67	6574	496e	666f	726d	6163	696f	…getInformacio
0x00000020	6e01	0014	2829	4c6a	6176	612f	6c61	6e67	n…()Ljava/lang
0x00000030	2f53	7472	696e	673b	0100	0d67	6574	4d61	/String;…getMa
0x00000040	6341	6464	7265	7373	0100	0a53	6f75	7263	cAddress…Sourc
0x00000050	6546	696c	6501	0012	696e	7465	7266	6163	eFile…interfac
0x00000060	6549	6e66	6f2e	6a61	7661	0100	166f	7063	eInfo.java…opc
0x00000070	696f	6e65	732f	696e	7465	7266	6163	6549	iones/interfaceI
0x00000080	6e66	6f01	0010	6a61	7661	2f6c	616e	672f	nfo…java/lang/

Blackbox malware analysis and reverse engineering

We found an old version of the Adwind trojan on Malwr.org[1] 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, 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 be able to 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

Table 8
Recovered socket data (colored by the proxied connection) shows how the heap address forms a communication timeline.

Obj. Address	Remote Connection		In/Out	Data (Up to 30 Bytes)
0x91c779b8	10.18.120.18	48002	⇒	Do something evil-48002!
0x91c7ead0	10.18.120.18	48003	⇒	Do something evil-48003!
0x91c85b70	10.18.120.18	48002	⇐	s3cr3t_d4t3_48002-00000000s3cr
0x91c938d8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDItMDAw
0x91c980d0	10.18.120.18	48003	⇐	s3cr3t_d4t3_48003-00000000s3cr
0x91ca5cb8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDMtMDAw
0x91cbfef0	10.18.120.18	48004	⇒	Do something evil-48004!
0x91cc7008	10.18.120.18	48005	⇒	Do something evil-48005!
0x91ccdee8	10.18.120.18	48004	⇐	s3cr3t_d4t3_48004-00000000s3cr
0x91cdbad0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDQtMDAw
0x91ce02c8	10.18.120.18	48005	⇐	s3cr3t_d4t3_48005-00000000s3cr
0x91cedeb0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDUtMDAw

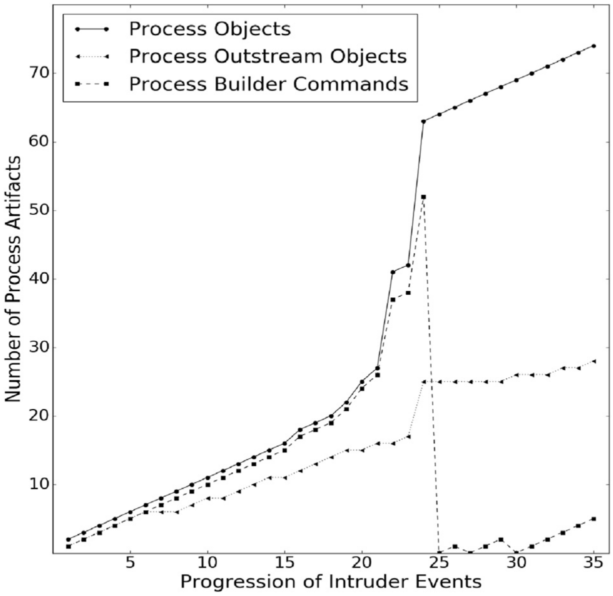


Fig. 4. Number of recoverable process artifacts present in the heap throughout the scripted attack.

---

[1] Windows analysis from Malwr.org: https://malwr.com/analysis/NGUwYjM1OGI4MGE4NDZkYjg5ZGVhMGU4YTMyN2RlMDU/.

```
//  r2 −c ’java prototypes a’ \
//   ae05bdff4d_1324_a9e8b700.class
import java.lang.ClassLoader;
import java.lang.String;
import java.util.zip.GZIPInputStream;
import extra.CLM;
import extra.CLM;
import extra.Constante;

class extra/CLM { // @0x0000
  // Fields defined in the class
  public static String pass;
  private static final extra.CLM instancia;

  // Methods defined in the class
  private void <init> ();
  public static extra.CLM getInstance ();
  public java.lang.Class findClass (String);
  private byte[] loadClassData (String);
  public static byte[] descomprime (byte[]);
  static void <clinit> ();
}
```

Listing 1. Radare2 `java prototypes` command reveals a custom loader in the decompressed class data.

Table 9
At *t* = 21, process artifacts are used to create an event log.

Event Log from Extracted Process Builder and Process Information

Address	Process Builder Command	PID	Buffered Data
0x91cb7258	sudo cat /etc/sudoers	1242	#\n# This file MUST be edited w
0x91e1c6f8	uname -ar	1245	Linux java-workx32-00 3.19.0-1
0x91e2bd98	id -un	1247	java\n
0x91e3aff0	id -nG	1248	java adm cdrom sudo dip plugde
0x91e4a598	sudo cat /etc/passwd	1250	root:x:0:0:root:/root:/bin/bas
0x91eb1240	sudo cat /etc/shadow	1252	root:!:16678:0:99999:7:::\ndaem
0x91ec2da0	sudo netstat -ap		
0x91eefea8	ls -all	1273	total 10312\ndrwxrwxr-x 6 java
0x91f556f0	ls -all		
0x91f66498	sudo -S nmap [...]	1275	\nStarting Nmap 6.47 (http://n
0x91f7e400	sudo lsof		
0x91f8d810	mount -v	1298	sysfs on /sys type sysfs (rw,n
0x91ff7ce0	cat /root/.bash_history	1301	history | grep pg\n history | gr
0x92014d20	cat /home/java/.bash_history	1307	ifconfig\nsudo add-apt-reposito
0x9203ae78	ps -ef		
0x920623c0	zip -r /tmp/backup.dat.zip	1322	adding: home/java/.ssh/ (sto
0x920720a0	ps -ef		
0x92099530	ls -all /tmp/	1328	total 44\ndrwxrwxrwt 9 root ro
0x920aab68	cat /tmp/backup.dat.zip	1333	PK\x03\x04\n\xdbL\x1fG\x0f\x1c
0x920cb718	ls -all /tmp/backup.dat.zip	1338	-rw–r–r– 1 root root 1617 Au
0x920db348	dd if=/dev/urandom of=		
0x920ead40	zip -r		
0x921601c8	ps -ef		
0x922c5dd8	ps -ef		
0x922ed348	zip -r /tmp/logs.zip /var/log/	1354	adding: var/log/ (stored 0%)
0x92305c50	ps -ef		

directly off the wire. While the message may not be intact, a forensic analyst could examine the class and method metadata and try to assemble a data flow graph, which could help recreate the message structure.

Scripted intrusion

We created a script that models how a *smash-and-grab* attacker would behave in a post-compromise setting. The scenario centers around an attacker exploiting the fact that dynamic plugins can be uploaded to a dotCMS server.[2] In this case the attacker leverages administrator credentials to upload and activate the plugin. When the plugin activates, it uses `wget` to retrieve and start the attacker's backdoor. The attacker uses a script to execute a series of steps using the malware. After each step in this script, we take a memory snapshot of the virtual machine and perform the JVM analysis. The 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.

Table 10
Interesting process output recovered at t = 35.

Address	PID	Buffered output
0x67020b20	1275	\nStarting Nmap 6.47 (http://n
0x67020c10	1403	total 176584\ndrwxrwxrwt 9 roo
0x6702de70	1273	total 10312\ndrwxrwxr-x 6 java
0x6702eb78	1252	root:!:16678:0:99999:7:::\ndaem
0x67092350	1250	root:x:0:0:root:/root:/bin/bas
0x67092b88	1248	java adm cdrom sudo dip plugde
0x670c0720	1245	Linux java-workx32-00 3.19.0–1
0x670c0880	1242	#\n# This file MUST be edited w
0x670c0f18	1354	adding: var/log/ (stored 0%)
0x670c13d8	1338	-rw–r–r– 1 root root 1617 Au
0x670c15d0	1333	PK\x03\x04\n\xdbL\x1fG\x0f\x1c
0x6711d068	1328	total 44\ndrwxrwxrwt 9 root ro
0x6711d718	1322	adding: home/java/.ssh/ (sto
0x6714c8b0	1307	ifconfig\nsudo add-apt-reposito
0x6714cc10	1301	history | grep pg\nhistory | gr
0x6714ceb8	1298	sysfs on /sys type sysfs (rw,n

Table 11
Call metadata for a selection of the Loader's methods at t = 35, revealing a large number of IO operations.

Address	Calls	Method name
0x63fdb6f8	256	Loader getLoaderInstance(…)
0x63fdb908	73	byte[] b64Decode(…)
0x63fdce98	256	integer sendSocketData(…)
0x63fdd038	256	integer sendSocketData(…)
0x63fdd670	1	void addClientHandlerSocket(…)
0x63fdd718	256	void stdout(…)
0x63fdd850	256	void logEvent(…)
0x63fddbd8	73	integer getPid(…)
0x63fddd50	73	integer startProcess(…)
0x63fddf68	256	java.lang.String readProcessStdout(…)
0x63fdd9e8	1	void main(…)
0x63fde1d8	1	integer access$100(…)
0x63fde168	2	java.lang.String access$000(…)
0x63fdb670	1	void start(…);

This evaluation concentrates on the created processes, and how much information can gleaned from their Java artifacts. This script starts 73 processes that execute OS 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_psxview` are used to try and find artifacts. Only one process (`sudo lsof`) 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

Case, A., 2011. Memory Analysis of the Dalvik (Android) Virtual Machine. URL: http://dfir.org/research/android-memory-analysis.pdf.

Chan, J.-T., Yang, W., 2004. Advanced obfuscation techniques for Java bytecode. J. Syst. Softw. 71.

Chen, K., Chen, J.-B., 2006. On instrumenting obfuscated Java bytecode with aspects. In: Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems SESS '06.

Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M. (2004). Understanding data lifetime via whole system simulation. In: Proceedings of the 13th Conference on USENIX Security Symposium – Volume 13 SSYM'04. URL: http://dl.acm.org/citation.cfm?id=1251375.1251397.

Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M. (2005). Shredding your garbage: reducing data lifetime through secure deallocation. In: Proceedings of the 14th Conference on USENIX Security Symposium – Volume 14 SSYM'05. URL: http://dl.acm.org/citation.cfm?id=1251398.1251420.

Cimato, S., De Santis, A., Petrillo, U.F., 2005. Overcoming the obfuscation of Java programs by identifier renaming. J. Syst. Softw. 78.

Cohen, M., 2014. Rekall memory forensics framework. In: DFIR Prague 2014. SANS DFIR. URL: https://digital-forensics.sans.org/summit-archives/dfirprague14/Rekall_Memory_Forensics_Michael_Cohen.pdf.

Detlefs, D., Flood, C., Heller, S., Printezis, T., 2004. Garbage-first garbage collection. In: Proceedings of the 4th International Symposium on Memory Management. ACM.

Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W., 2009. Lest we remember: cold-boot attacks on encryption keys. Commun. ACM 52. URL: http://doi.acm.org/10.1145/1506409.1506429. http://dx.doi.org/10.1145/1506409.1506429.

Harrison, K., Xu, S. (2007). Protecting cryptographic keys from memory disclosure attacks. In: Dependable Systems and Networks, 2007. DSN'07. IEEE International Conference on 37th Annual IEEE/IFIP..

Hilgers, C., Macht, H., Müller, T., Spreitzenbarth, M. (2014). Post-mortem memory analysis of cold-booted Android devices. In: Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics IMF'14. URL: http://dx.doi.org/10.1109/IMF.2014.8. http://dx.doi.org/10.1109/IMF.2014.8.

Li, Y., 2015. Where in Your Ram is "*python san_diego.py*"? URL: https://www.youtube.com/watch?v=tMKXcc2-xO8.

Low, D., 1998. Protecting Java code via code obfuscation. Crossroads 4.

Müller, T., Spreitzenbarth, M. (2013). FROST: Forensic recovery of scrambled telephones. In: Proceedings of the 11th International Conference on Applied Cryptography and Network Security ACNS'13. URL: http://dx.doi.org/10.1007/978-3-642-38980-1_23. http://dx.doi.org/doi:10.1007/978-3-642-38980-1_23.

Pancake, 2015. Radare2: a Unix-Like Reverse Engineering Framework and Commandline Tools. URL: http://www.radare.org/.

Pridgen, A., Garfinkel, S., Wallach, D.S. (2017). Present but unreachable: Reducing persistent latent secrets in HotSpot JVM. In: IEEE International Conference on System Science (HICSS), 2017 50th Hawaii.

Proebsting, T.A., Watterson, S.A., 1997. Krakatoa: decompilation in java (does bytecode reveal source?). In: COOTS.

Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D. (2015a). GUITAR: Piecing together android app guis from memory images. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security CCS'15. URL: http://doi.acm.org/10.1145/2810103.2813650. http://dx.doi.org/10.1145/2810103.2813650.

Saltaformaggio, B., Bhatia, R., Gu, Z., Zhang, X., Xu, D. (2015b). VCR: App-agnostic recovery of photographic evidence from Android device memory images. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security CCS'15. URL: http://doi.acm.org/10.1145/2810103.2813720. http://dx.doi.org/doi:10.1145/2810103.2813720.

Saltaformaggio, B., Gu, Z., Zhang, X., Xu, D. (2014). DSCRETE: Automatic rendering of forensic information from memory images via application logic reuse. In: Proceedings of the 23rd USENIX conference on Security Symposium. USENIX Association.

Schlumberger, J., Kruegel, C., Vigna, G. (2012). Jarhead analysis and detection of malicious Java applets. In: Proceedings of the 28th Annual Computer Security Applications Conference ACSAC'12.

Sun Microsystems, 2006. Memory Management in the Java HotSpot Virtual Machine. URL: http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

Viega, J., 2001. Protecting Sensitive Data in Memory. URL: http://www.ibm.com/developerworks/library/s-data.html?n-s-311.

VöMel, S., Freiling, F.C., 2011. A survey of main memory acquisition and analysis techniques for the Windows operating system. Digit. Investig. 8. URL: http://dx.doi.org/10.1016/j.diin.2011.06.002. http://dx.doi.org/doi:10.1016/j.diin.2011.06.002.

Walters, A., 2007. The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. URL: https://www.volatilesystems.com/default/volatility.