Present but unreachable: Reducing persistent latent secrets in HotSpot JVM

Adam Pridgen
Rice University
dso@rice.edu

Simson L. Garfinkel
George Mason University
simsong@acm.org

Dan S. Wallach
Rice University
dwallach@cs.rice.edu

Abstract

Applications that manage sensitive secrets, including cryptographic keys, are typically engineered to overwrite the secrets in memory once they're no longer necessary, offering an important defense against forensic attacks against the computer. In a modern garbage-collected memory system, however, live objects will be copied and compacted into new memory pages, with the user program being unable to reach and zero out obsolete copies in old memory pages that have not yet been reused. This paper considers this problem in the HotSpot JVM, the default JVM used by the Oracle and OpenJDK Java platforms. We analyze the SerialGC and Garbage First Garbage Collector (G1GC) implementations, showing that sensitive data such as TLS keys are easily extracted from the garbage. To mitigate this issue, we implemented techniques to sanitize older heap pages and we measure the performance impact–sometimes good, sometimes unacceptable. We also discuss how future garbage collectors might be designed from scratch with efficient heap sanitation in mind.

1. Introduction

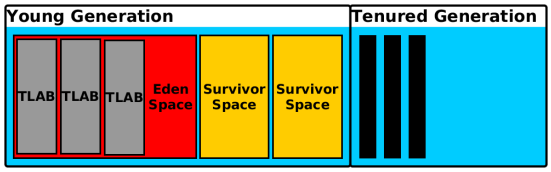
Managed memory runtime environments like Java eliminate many kinds of programming errors that can become security vulnerabilities. For example, Java programs are not vulnerable to buffer overflow attacks, making Java (and other "safe" languages) attractive for building security-critical software. Meanwhile, techniques such as just-in-time compilation, hot-spot optimization, and parallel garbage collection, have largely eliminated the performance penalty of using managed runtime environments. These features and a rich set of standard libraries have led to a broad adoption of Java and other such languages.

However, the HotSpot JVM introduces risk when dealing with sensitive data [1]. Our research shows that the HotSpot JVM allows session identifiers, passwords, and TLS 1.2 session keys to remain in the JVM process memory after the corresponding Java objects have been garbage collected. Furthermore, because Java provides no direct access to the underlying memory, developers cannot explicitly sanitize their sensitive data once it is no longer needed. An attacker, on the other hand, might still be able to gain access to the raw memory through many means, such as a hypervisor bypass attack, access to a swap or hibernation file, or from another process running on the same physical machine. Of course, we hope that traditional system security mechanisms can keep an attacker away from this data, but for the cases in which an attacker can gain access to the JVM's process memory, limiting the time that sensitive data remains in memory provides defense-in-depth security coverage.

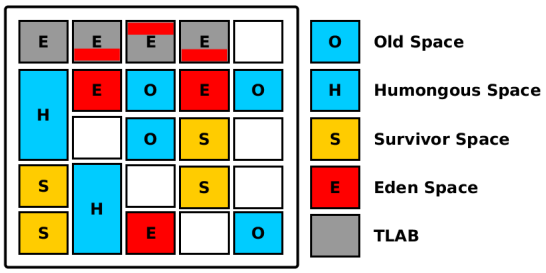
Automated memory management falls into two categories: *reference counting*, which is used in Python and Swift, and *tracing garbage collection*, which is used by Java and many other language systems. Tracing garbage collection measures reachability from a set of *root objects* to every object in memory. Objects that can no longer be reached are considered *garbage*. Garbage collectors are typically *lazy*; there is a gap between when an object becomes unreachable and when the garbage collector reuses that memory. GC can also *re-arrange* the managed heap to help improve collector performance and reduce pause times. This rearrangement inherently involves *copying* objects, which can leave behind multiple "old" copies.

In this paper we demonstrate confidentiality failures due to a semantic gap between the language that programmers use, the language implementation, and the underlying execution environment, echoing similar findings in other areas (e.g. [2, 3]). Specifically, we establish the volume of secrets that an unsanitized heap can expose, using a TLS web client atop Oracle's HotSpot JVM, driven by a synthetic load under different levels of memory contention. We capture whole system memory images and then use binary string searches to find TLS keys and other sensitive data. We then make changes to the TLS code and the garbage collector in attempt to eliminate most of these secrets. We find that zeroization adds a significant workload to the JVM; Section 6 provides future direction on how to design JVM systems that do not sacrifice performance for improved confidentiality.

URI: http://hdl.handle.net/10125/41887
ISBN: 978-0-9981331-0-2



(a). Typical SerialGC Generational Heap



(b). Typical G1GC Generational Heap

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

2. HotSpot Memory Background

The HotSpot JVM uses *generational copying* to improve memory management performance. This scheme allocates new objects to a *young generation* and later *promotes* them to a *tenured generation* if they are still reachable. HotSpot implements several garbage collectors, all of which follow variations of this strategy.

The young generation is partitioned into an *Eden space* where objects are created (and where most die), and two *survivor spaces* that hold objects that are copied out of the Eden space. The Eden space is further partitioned into *thread local allocation buffers* (TLAB), where allocation is performed using a "bump-the-pointer" technique to minimize the number of locks required for multi-threaded applications. As objects age and survive GC, they are migrated from Eden to the survivor spaces and tenured if the objects surpass an age threshold. 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 [5]. Stale data may be overwritten as memory is reused, but these overwrites may never happen. Figure 1**a** shows a typical Java heap memory layout for this implementation.

HotSpot's Garbage First Garbage Collector (G1GC) uses a partitioned heap space (Figure 1**b**), allowing parallel garbage collection during incremental collection prior to the full stop-the-world garbage collection. Furthermore, G1GC does not need to collect an entire generation before a collection cycle completes. During an incremental collection, G1GC identifies regions with the most garbage so that a collection cycle reclaims the most optimal regions to achieve its soft real-time goals. G1GC tries to collect regions with the most garbage, which is done by copying objects into a new region and reclaiming the previous one [6].

Most garbage collectors can take advantage of additional RAM, gaining additional performance when faced with less pressure to compact live objects and reuse memory. We find that JVMs configured with larger heaps generally allow latent secrets to remain longer in RAM, improving their chance of recovery. It might seem tempting to solve the problem of latent secrets by limiting the heap size to compel more memory reuse, but this tactic decreases performance.

Below the garbage collector are HotSpot's memory *regions*: large blocks of memory which might be used by the garbage collector or might be otherwise used by native 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 it without first zeroing it.

3. Prior Work

In 2001 Viega identified that memory is not securly deallocated in C, C++, Java, and Python runtimes [1]. Chow *et al.* 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. The authors implemented proper sanitization in the Unix operating systems with roughly a 1% impact on performance [2, 7]. However, Chow *et al.*'s techniques cannot address the latent secrets found in the HotSpot, because the JVM uses it's own memory management primitives. Additional work has been done to help reduce latent secrets due to shared program variables with static analysis [8]; Anikeev *et al.* [9] proposed introducing keywords into managed languages hinting at how to securely manage object instances.

Anikeev *et al.* [10] study the problem of latent secrets in an Android runtime that uses the Dalvik VM (DVM) and attempt to solve this problem by altering the GC implementation. Their work mentions negative performance impacts due to sanitization, but they do not demonstrate how well latent secrets are eliminated from the DVM heap. We investigate similar questions with two different GC implementations and measure both the performance impact and the effectiveness of eliminating latent secrets.

CleanOS [11] is the most effective solution for eliminating the *clear-text* presence of latent secrets in a VM runtime (the DVM). The researchers extended the Android SDK to allow programmers to explicitly tag some objects as *sensitive data objects* (SDOs) and developed

a new GC, the *evict-idle GC* (eiGC), that properly sanitizes SDOs. Beyond programmer tagging, SDOs can be implicitly tagged as the result of taint analysis. CleanOS uses TaintDroid [12] to help identify sensitive data that are sourced from SDOs (e.g. data from TLS sockets). The eiGC protects these objects by encrypting *idle* object data with an escrowed cloud application key. When the object is idle long enough, this key is securely deallocated. If the object is still in use by the application, the eiGC can fetch the key from the cloud application and decrypt the data. This approach relies on a third party or an additional application server to manage keys in a secure manner, which is not ideal in certain situations.

The process of extracting latent secrets from dump files or system memory seems challenging, but many researchers have found the task to be quite surmountable. For example, Harrison and Xu identified RSA cryptosystem parameters in unallocated memory that had been inadvertently written to untrusted external storage as the result of a Linux kernel bug [13]. Halderman *et al.* showed that AES encryption keys can be readily detected in RAM from their key schedule [14]. Case presented an approach for analyzing the contents of the Dalvik virtual machine [15]. Similar attacks are possible against Android smartphones, allowing for the recovery of disk encryption keys [16] and Dalvik VM memory structures [17]. Jin *et al.* used symbolic execution and intra-procedural analysis to accurately extract the composition of type data generated by C++ programs [18].

Finally, there are a variety of memory disclosure attacks and techniques. The most straightforward technique uses one process to read the memory of another process utilizing a suitable device driver or kernel module (e.g. */dev/mem* or the */proc/nnn/mem* devices). Because such devices are commonly exploited by malware, many operating systems no longer include devices for reading the memory of other processes. However, Stüttgen and Cohen developing an approach for safely loading a pre-compiled kernel modules into memory on running Linux systems [19]; their approach is now used by the Rekall Memory Forensics Framework [20].

Halderman *et al.* developed the "cold-boot attack" in which the DRAM memory from the target computer is physically chilled and then transferred to a computer that is known not to wipe memory on boot [14]. It is also possible to physically read the contents of a computer's memory using hardware that provides direct memory access (DMA). Consumer firewire interfaces, JTAG interfaces, and specially constructed interface cards can perform DMA; VöMel and Freiling survey such techniques for acquiring main memory in computers running Microsoft Windows [21]. Consequently, the threat of an attacker conducting a memory disclosure attack is significant, justifying efforts to mitigate these attacks.

We note that this class of attack may apply in a variety of different devices. Smartphones and laptops may be physically stolen or otherwise captured, giving a motivated attacker physical access to the device. Cloud services may migrate virtual machines from physical system to system, allowing for a variety of attacks while the VM is migrating, or accessing the system's memory from a potentially compromised hypervisor.

This article is predicated on the assumption that an attacker 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.

4. Measuring Latent Secrets

Here we discuss the infrastructure and software used to measure latent secrets in the HotSpot JVM. For simplicity, we used *black-box* analysis. Our Java client application repeatedly made TLS connections to our instrumented web server, creating an abundance of latent secrets in the heap. On the web server, our modified OpenSSL library recorded each session's pre-master secret (PMS) and master secret (MKB). We then searched a memory dump of our Java client's Linux virtual machine for all of the previously logged secrets from every TLS session.

We ran these experiments on a small cluster of PCs running Linux KVM; the number of simultaneous virtual machines were limited to avoid resource contention and prevent measurement discrepancies. Each experiment consisted of a pair of x64 Ubuntu 14.04 LTS VMs: our synthetic client and a TLS webserver using a modified OpenSSL library. The web server used NGINX and TLS 1.2 to serve several static web pages. The web server VMs utilized four logical cores and 2 GiB of RAM—enough to ensure that server performance wasn't the bottleneck during the experiments. The VMs that ran the Java clients were configured with 20 GiB of RAM and 4-CPUs when using the SerialGC and 8-CPUs when using G1GC. VMs running the synthetic client were rebooted at the conclusion of each experiment, allowing us to restart each run from a similar starting point.

4.1. Synthetic Client Functionality

Our synthetic Java client is a *multi-threaded*, *configurable* TLS web client. The client implements several parameters that manipulate the memory pressure exerted on the heap, the number of concurrent threads, the maximum number of HTTPS requests, and the lifetime of a thread sending the web requests. These parameters

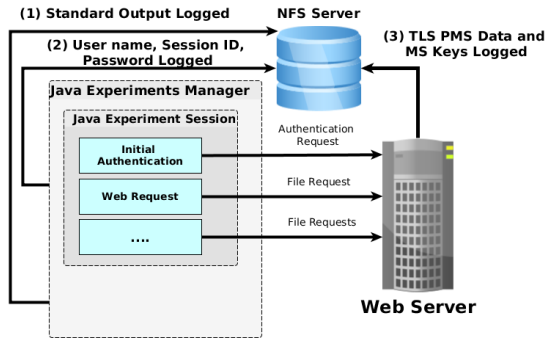


Figure 2. Functional overview of our synthetic web client.

provide the ability to model basic transactions for applications such as a thick- or web-service client.

For this paper we choose two specific configurations. Both configurations allowed up to 192 concurrent TLS connections that were active for at least 96 seconds. They differed in amount of heap memory allowed (e.g. memory allocated from the JVM in the form of objects) and by the number of requests allowed per thread. The high memory pressure (HMP) experiment allowed allocations to consume up to 80% of the JVM's managed memory, and the low memory pressure (LMP) experiments allowed a maximum allocation of 20% from the JVM's managed memory.

Figure 2 shows the two main components of the synthetic client. The Java Experiment Manager (JEM) managed all experimental sessions (threads). The Java Experimental Sessions (JES) implemented the web client functionality in a Java thread. A Python script started the experiment with parameters that defined the behavior of the synthetic client, the IP address of the server, and where to store log files containing events and other data.

The JEM was responsible for managing the number of JESs and enforcing the experimental behavior and garbage collection parameters. The JEM controlled the number of concurrent JESs, JES allocation behavior, JES HTTPS requests, and the overall lifetime of the JES thread. Parameters controlling the garbage collection defined the frequency of collection, when to start collecting, and whether or not to pause JESs after the first GC. Our experiments also allowed us to vary the TLS library in use (Oracle vs. BouncyCastle) and whether to use the Apache HttpClient or a TLS on top of a basic Java socket.

We implemented and use three different TLS web clients in the JES. The most basic TLS client was the "Socket TLS Client". This type of client opened a TLS socket to the remote server, sent a raw HTTP request as a formatted string, received data, and closed the socket. The second client ("Apache TLS Client") used the Apache "HTTPComponents" library to create an `HttpClient`, which then connected to the remote host. Most of the internal HTTP mechanics were abstracted away, simplifying the entire retrieval task; this abstraction removed sensitive data like usernames and passwords from our control. The final client ("BoucyCastle TLS Client") was a variant of the Apache TLS Client that uses the BouncyCastle cryptography library instead of Oracle's cryptography library. This option allowed us to measure whether the TLS implementation, itself, can contribute to the volume of latent secrets.

Each implementation made every effort to remove excess references and prepare the connecting object for a future collection. In the Socket TLS Client, we close the `Socket` and set our references to it to `null` as soon as possible. The Apache `HttpClient` does not have an explicit close or shutdown API, so only references to the Apache TLS Client and BouncyCastle TLS Client be set to `null`, and we hoped its internals don't maintain references to sensitive data. We also note that the Apache `HttpClient` uses an `HttpClientConnectionManager` to manage client connections. This manager may choose to maintain open connections to the remote hosts. Such socket reuse makes reconnecting to an old peer much faster, avoiding the overhead of rebuilding a TLS connection, but may also contribute to the build-up of key material in memory longer.

4.2. Memory and data analysis

Data analysis and extraction happened in three distinct phases. After an experiment, the resulting memory, TLS session data, and web client logs containing sensitive HTTP parameters such as the username and password are queued for analysis. First the analysis process scans the memory dump for latent secrets (e.g. PMS and MKBs) using `jbgrep`. This scan is conducted using two perspectives of the memory dump. The first perspective is the *raw* memory dump, which reveals all the latent secrets along with a count for each one found. The second reconstructs the process memory using virtual memory mapping, which details where the latent secret exists in the Java process (e.g., which generational heap and the address in the Java process).

After the latent secrets are identified and counted, a post-processing step enumerates every HTTP request for each JES and pairs these requests using the TLS session data and a monotonically increasing timestamp. Although we are unable to pair the exact TLS session to the corresponding web request, such granular knowledge is not necessary to create an approximate timeline showing live objects versus latent garbage in the heap. All

Heap Size (MiB)	# of TLS Sessions	Socket TLS Client Recovered Keys		Apache TLS Client Recovered Keys	
		#	%	#	%
512	5000	489	9%	286	5%
1024	5000	1059	21%	499	9%
2048	10000	1845	18%	929	9%
4096	10000	3177	31%	1608	16%
8192	15000	4786	31%	3008	20%
16384	30000	9058	30%	5354	17%

Table 1. The average percentage of recoverable TLS sessions from HMP clients using the SerialGC on the Oracle HotSpot JVM.

JVM Version	Keys recovered after GC: Bouncy Castle TLS Client	Apache TLS Client	Sockets TLS Client
Low Memory Pressure (LMP) Results			
Oracle JVM	1542 ± 92	2972 ± 81	1084 ± 84
Modified JVM	341 ± 55	827 ± 30	304 ± 117
Modified JVM/JCE	364 ± 102	848 ± 44	371 ± 89
High Memory Pressure (HMP) Results			
Oracle JVM	1671 ± 86	3052 ± 60	1202 ± 86
Modified JVM	406 ± 87	944 ± 78	371 ± 94
Modified JVM/JCE	375 ± 103	1010 ± 55	387 ± 56

Table 2. The number of unique TLS keys that are recoverable after garbage collection.

the dead PMS and MKB data are identified, and the results are stored.

5. Removing Latent Secrets

Figure 3 combines the results of several experiments. The preliminary experiments (shown in black) use the Oracle HotSpot JVM to examine the retention of latent secrets in the heap. These experiments use the selected GC and run with a varied heap size between 512MiB – 16GiB. The number of TLS session also vary to establish a reasonable baseline of retained latent secrets in each heap size. We collected 20 samples per memory collection (Figures 3**a** – 3**d**) to help us identify any potential variance in our measuremnts. We see the obvious outcome where the number of recoverable TLS keys increases with heap size, doubly in some cases.

Table 1 shows a sample of recoverable unique keys from two control experiments. Specifically, the table focuses on the Socket and Apache TLS Clients using HMP parameters using SerialGC. The Apache TLS Client has fewer recoverable keys than Socket TLS Client, apparently attributable to the larger memory footprint of each `HttpClient`. The Socket TLS Client requires less heap memory per connection because it only requires IO buffers and a reference to the OS socket. This means the Socket TLS Client client can make more connections before GC happens.

Thus, the JVM process is a viable target for memory disclosure attacks. For each TLS key recovered in our control experiments, there are roughly 1-2 copies of the pre-master secret (PMS) data and 3-4 copies of fully intact master key blocks (MKB), i.e., TLS session keys. Multiple copies of key data are the result of extraneous copies and excessive references to these copies. When our results refer to "unique keys," we note that an MKB can be derived from a PMS, so if we find both, we'll only count them as one "unique key."

Where are these key copies coming from? Inspection of the OpenJDK Java JDK source code reveals that local variable references are not zeroed then set to `null` and *cloned* `byte[]` values are not zeroed when they are no longer needed, so the latent data stays in memory until the memory gets reused. And, because of the generational structure of the GC, there may be additional copies of older keys.

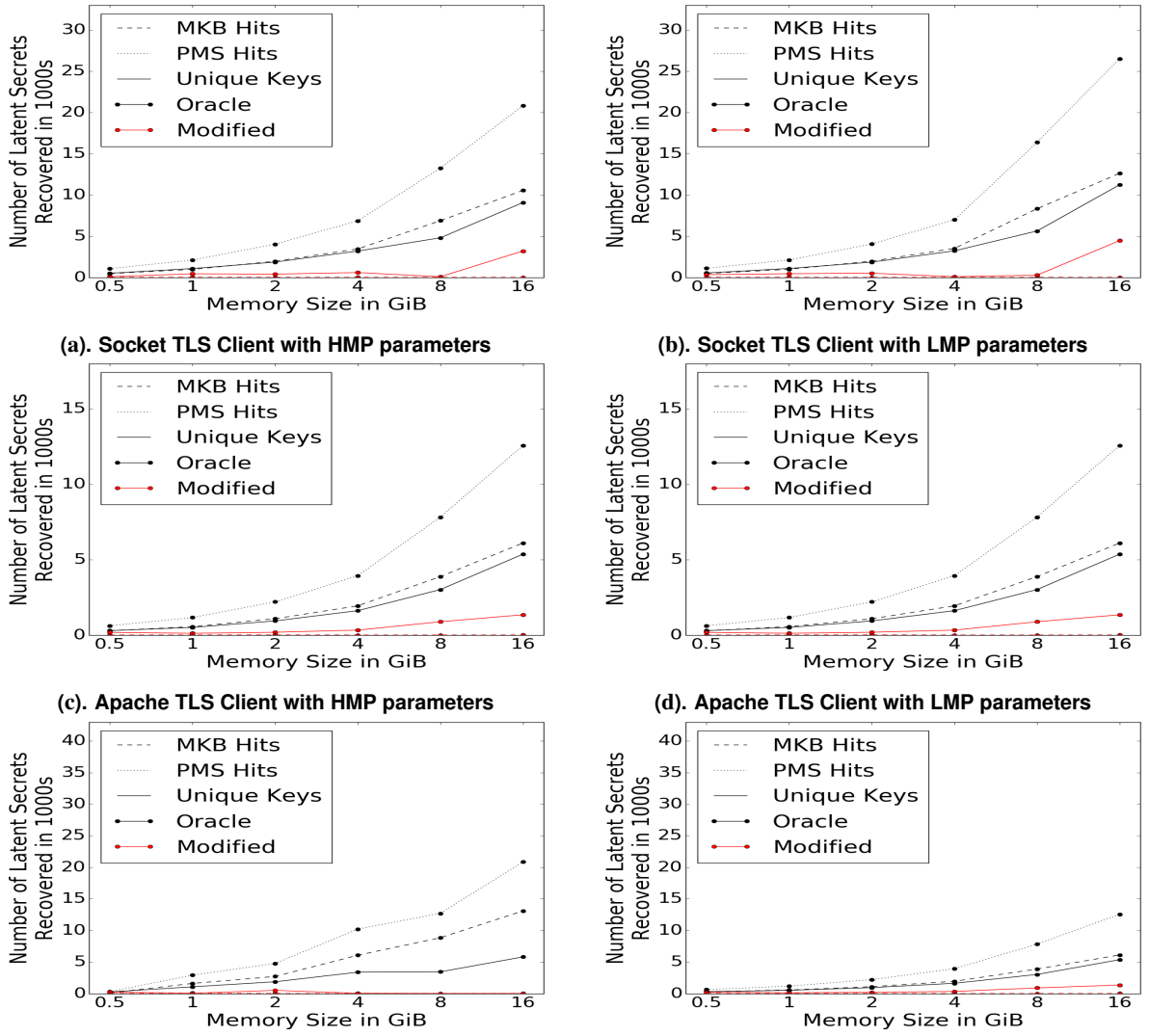
It is clear that latent secrets are a concern. When key material from thousands of closed connections sticks around in memory, it significantly increases the risk that encryption keys might be compromised.

Consequently, we devised two approaches to address this issue, both requiring changes to the OpenJDK source code. First, we attempted to patch the Java Cryptography Engine (JCE) and Java Secure Sockets Extensions (JSSE). After manually auditing the code, we took steps to ensure classes perform explicit sanitization on local variables containing secrets, and explicit calls are added to ensure key data is overwritten when TLS sessions and sockets close. Our second approach focused on modifying the JVM internals. Specifically, we added code to zero memory as it was de-allocated, and to zero all unused heap spaces after each GC-cycle.

5.1. Adding Sanitization to the JVM

We modified the OpenJDK HotSpot JVM source code to implement a *global* sanitization solution in the heap. For simplicity, we choose to modify the SerialGC and G1GC implementations—the current and future default *server* garbage collectors for the HotSpot JVM. First, we focused our efforts on cleansing the young generation in the SerialGC, and then we tackled the problem in the tenured generation. The following approach generalizes nicely to both collectors. The emphasis of the approach forces sanitization on the internal memory structures of the JVM and managed heap during and after the garbage collection cycle happens.

Since generational GC partitions the heap, the algo-



(a). Socket TLS Client with HMP parameters

(b). Socket TLS Client with LMP parameters

(c). Apache TLS Client with HMP parameters

(d). Apache TLS Client with LMP parameters

(e). Socket TLS Client with HMP parameters (G1GC)

(f). Apache TLS Client with HMP parameters (G1GC)

Figure 3. These plots compare the results for the Socket TLS Client the Apache TLS Client. The lines show how many latent secrets can be removed from memory by sanitizing the heap space after garbage collection. High- and Low-pressure applications are also shown. Figures 3a-3a use the modified SerialGC and Figures 3e and 3f use the modified G1GC.

rithms and policies used to collect each generation can vary. For example, the SerialGC young generation uses copy-collection while the tenured generation generally relies on mark-and-sweep followed by compaction. The G1GC employs similar techniques to decide how and when incremental or full collection happen. In both cases, we tagged along with the sweep or incremental phases, zeroing out regions of the memory corresponding to dead objects. When compaction happens, we similarly zero out the original objects, one by one, after they're relocated. (Unfortunately, Hotspot's SerialGC doesn't ever do a giant copy-compaction during collection in the tenured space with a from-space and a to-space, so there's never a huge block of memory we can blindly zero out.)

Zeroing individual objects, or arrays, as the sweep phase or copy phase figures out that they're garbage, seemed like a relatively efficient change to make to the garage collection, since the memory in question was just recently touched, so it should already be in the CPU's cache. Unfortunately, this strategy required us to understand all of the specific tricks that the garbage collector uses, so we know when it's truly safe to write zeros into memory.

Notably, we encountered cases where *invalid* dummy objects were placed in the heap. Without knowing this fact, we would check pointers and class types using internal APIs, and these checks caused segmentation

faults in the JVM. After some investigation, we discovered that this issue was the result of a hack to make the heap appear to be contiguous during collection, which is a precondition to make GC work correctly. Recall, each TLAB is a small partition of the eden space, so the JVM fills the empty spaces with dummy objects during GC or when a TLAB is invalidated. We resolved this issue by ensuring `Klass` pointers (i.e., pointers to the C++ representation of a Java class) fall inside the Java metaspace, where all Java meta-objects (e.g. classes, methods, etc.) reside, prior to overwriting.

We also had a variety of other minor issues. For example, dealing with primitive Java types (like `byte` arrays) versus class types (like `Byte` arrays) required specific logic.

After modifying the garbage collector, we still found latent secrets that survived, and they were outside of the managed heap. Recall that the JVM maintains memory blocks that are explicitly allocated and freed. Latent secrets were getting copied there as well. We addressed the problem by sanitizing all internal memory deallocations (similar to [2]). Leveraging the JVM's native memory tracking (NMT) for this task. Typically, NMT is used to track internal memory allocations to help with profiling, diagnostics, and debugging. For our purposes, we used NMT to identify the size of each allocation, and then we zero the buffer before the memory is returned to allocation pool.

5.2. Sanitization Effectiveness

Our second set of experiments focus on assessing our JCE/JSSE modifications, and we also explore whether Oracle's cryptography library might be responsible for retaining additional latent secrets. Three different Java runtime configurations use the SerialGC: Oracle HotSpot JVM, modified OpenJDK HotSpot JVM and modified OpenJDK HotSpot JVM cryptography libraries; our three different TLS clients run in these environments. The JVM heap size is fixed at 4GiB, and 10 memory dumps are collected for each experimental configuration. To ensure that sanitization is working properly, the JEM pauses the JES threads and performs an explicit GC to invoke the added sanitization steps before dumping the VM's system memory. The Socket TLS Client experiment averages 11.9K TLS sessions, and the Apache TLS Client and Apache TLS Client with BouncyCastle experiments average 16.6K and 16.8K respectively.

Table 2 shows the average results from this set of experiments. The modified JVMs exhibit a significant drop in the number of latent secrets present. However, this massive reduction is mostly attributable to the sanitization added after collecting the young generation. Since sanitization of the heap generations depends largely on allocation failures, the tenured generation needs more collection activity to trigger the removal of latent secrets, which we realized after analyzing the GC logs.

We also see that the JCE and JSSE modifications modestly increased the number of latent secrets in the heap. We're not entirely sure why this occurred. It's possible that our code modifications created side-effects on the JIT compiler. For example, compiler could have optimized out our zeroing code or perhaps the resulting code maintains unnecessary references to otherwise dead objects. These negative findings reinforce the importance of support from not only the garbage collector but also the underlying JVM. Pure application-level zeroing of data will never adequately address the problem.

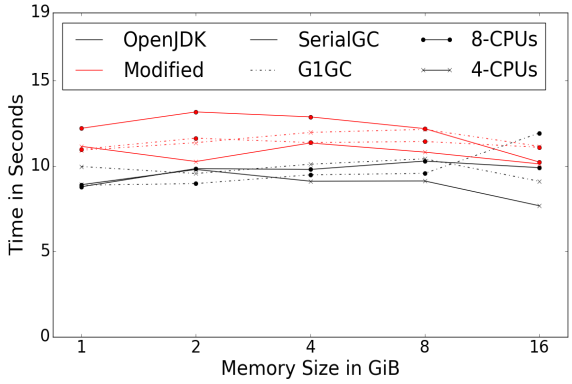
Both the "BouncyCastle" configuration and the "Apache" configuration use the same Apache HTTP client library, so the only significant difference is that the "Apache" configuration is using the Oracle TLS library. Why is the BouncyCastle version so much better? A manual inspection of the BouncyCastle code shows that the authors make fewer copies of key material. That said, the "Socket TLS Client" experiment drops the Apache HTTP client library and directly drives the Oracle TLS libraries. This gives up the performance and concurrency features of the Apache library, but has the fewest latent secrets. These results suggest that complex interactions between libraries and networking layers can have unforeseen increases in the volume of latent secrets.

The third and fourth experiments recreate the conditions from the initial assessments to evalutate the overall reduction of latent secrets. The JVM heap size varies from 512 MiB – 16 GiB, the number of sessions vary between 5K – 30K, and the host system uses 4-CPUs (SerialGC) or 8-CPUs (G1GC). In the third experiment, the focus is on both the HMP and LMP configurations of the Apache and Socket TLS Client, and the fourth experiment only looks at HMP configurations.

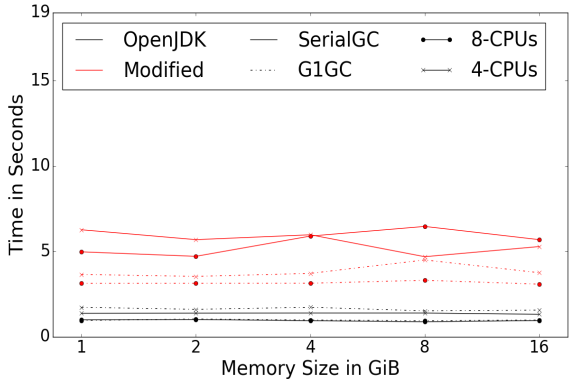
Figures 3**a**-3**d** presents a progression towards eliminating latent secrets in the heap using the SerialGC. In some circumstances the volume of latent secrets stays small regardless of heap size, while in other circumstances the volume of latent secrets starts small, but with very large heaps it grows significantly. We believe this is a consequence of the tenuring process. The GC may not collect seemingly dead objects in the tenured generation because extraneous references to those objects are not dead yet. Additionally, if memory pressure is inadequate then this generation may never be collected.

Figures 3**a** and 3**c** demonstrate that this tenuring effect on objects is mitigated, and most, if not all, of the la-

tent secrets are eliminated. The G1GC offers improved performance and sanitization because it uses smaller heap regions to store objects. These partitions help facilitate parallel GC, and the smaller regions can lessen the amount of memory that needs to be zeroed during each GC, especially in the tenured generation. After some initial experiments, we found that *two* explicit GC calls are necessary to achieve *full* heap sanitization; the SerialGC required *four* explicit calls to fully zero the heap. The first explicit GC triggers an incremental collection, and the second forces a full collection because it interrupts the incremental collection.



(a). `tradebeans` - DayTrader Benchmarks



(b). `lusearch` - Text Searching Benchmark

Figure 4. The benchmarks show that our modifications have significant effects on the GC performance. The unmodified OpenJDK (black) benchmarks are the baseline.

5.3. Benchmarking

Overall, the findings from each of the experiments demonstrate the difficulty of eliminating sensitive data in a managed runtime. Now, we want to determine how our modifications affect the JVM's performance. We use the Dacapo benchmark suite, version 9.12 [22] to measure these impacts. This benchmark framework uses several real world applications to measure runtime performance, but we only show two for brevity: `lusearch` and `tradebeans`. The `lusearch` benchmark performs a number of searches over a textual corpus using `lucene`, a text searching engine. The `tradebeans` benchmark is a DayTrader application that interacts with an Apache Geronimo backend and `h2`.

Each benchmark executes 50 times with the default workload parameter, *pre-iteration* GC disabled and four workload threads. The benchmarks run on an unmodified OpenJDK HotSpot JVM and the modified JVM with variable heap sizes (e.g. 1 GiB–16 GiB) and CPUS (either 4 or 8). The unmodified JVM is built on the same machine using the same build settings of our modified JVM; we take this precaution to eliminate any build or code optimization variables that might influence the benchmark results, which cannot otherwise be done with Oracle's HotSpot JVM.

Figure 4 shows the benchmark results, which are not surprisrising. The G1GC with 8-CPUs incurred performance penalties from sanitization: 200% in `lusearch` and 21% in `tradebeans`. We discuss how future GC implementations can address the performance issue in §6.

6. Discussion and Future Work

Cleansing latent secrets from managed memory is a challenging problem, and application or runtime demands are going to dictate how these challenges are addressed. We have seen that parallel collectors like the G1GC offer some relief for these issues, but redesigning the JVM to provide for proper sanitization seems to be a better alternative in terms of performance. CleanOS exemplifies how these modifications to both the managed heap and the software VM imprpove security [11]. This extension to Android encrypts sensitive data objects in place, and then when the object is no longer needed or it's idle, the key is securely deallocated. However, when considering server-side changes, re-engineering the JVM should also consider dealing with sensitive native IO operations, shared variables in applications [8], and incorporating explicit *data lifetimes* into the programming language [7, 9].

A feasible strategy explicitly segregates sensitive data into its own *monitored space* that identifies and removes latent secrets promptly. Developers need the ability to define data lifetimes when writing their code. Java currently has meta-data tags, known as *annotations*, that help with the compile-, build-, and runtime operations. *Data lifetime* annotations could help the JVM handle, store, and sanitize these data items without impacting other code.

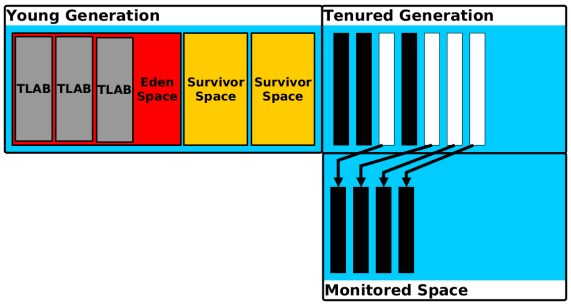


Figure 5. A generational heap layout concept that uses a monitored space where *sensitive* data is stored and maintained.

Figure 5 shows a hypothetical heap with an additional monitored space. In the monitored space, memory might be explicitly reference counted, allowing for immediate sanitization when an object dies. Furthermore, these objects could have explicit "destroy" APIs, so applications can explicitly kill them or an executive task can reclaim the object. Functionally, references from the main heap to the monitored space would act like *weak references*, making it clear to the application author that sensitive, monitored data could disappear at any time, and must be checked explicitly before each use. Such a strategy sounds straightforward, but it would have a variety of problems. For example, sensitive data can touch multiple layers of the protocol stack. Zero-copy IO techniques (e.g., IO-Lite [23]) could help with this, but require the entire stack to be engineered around a particular buffer management strategy. Changes like this would break existing APIs and require re-engineering of libraries.

7. Conclusion

Java and the HotSpot JVM will likely be around for decades to come. This runtime offers a rich set of development tools and libraries that help engineers construct and deploy useful software. However, servers and services are susceptible to a number of attacks through a variety of vectors, so there is no guarantee that the system where the software executes will remain free from compromise. Attackers evolve quickly, and they will realize that the JVM does not effectively sanitize internal or Java heap memory. This lack of sanitization can compromise sensitive data and lead to unforeseen impacts and consequences. We have taken a proactive approach to this problem by measuring its existence and developing several strategies to help mitigate the problem.

Problems with managed runtime environments like Java and the JVM are well known, but they are not well understood. Our research provides several fundamental elements. We establish the heaps capacity to retain latent secrets. Furthermore, we show that as heaps increase in size the number of latent secrets also increases. Cryptographic libraries should protect sensitive data such as keys, but we find that Oracle's JCE implementation of TLS 1.2 does not attempt to eliminate key data.

Given the lack of sanitization in the Java heap, we demonstrate several approaches that reduce the accumulation of sensitive data. Used together, the number of TLS keys are reduced dramatically. To accomplish this feat, we first modify the JVM to zero unused heap space in the young generation. Second, the tenured generation is also wiped when the dead objects or live objects are encountered during the mark-sweep-compact collection algorithm. We also zero unused heap space after the garbage collection, showed these approaches work for both the SerialGC and the G1GC.

We define how to improve performance of garbage collection implementations while keeping data security in mind. Our proposed design modifies the overall structure of the heap, carving out a segment specifically for sensitive data. The design also exploits Java annotations, which can be used to inform the runtime about how to properly handle specific types of data. This design keeps execution and runtime efficiency in mind while allowing for the timely and effective sanitization of data.

8. Acknowledgments

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

9. References

[1] J. Viega, "Protecting sensitive data in memory," 2001.

[2] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2004.

[3] V. DSilva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE CS Security and Privacy Workshop*, (San Jose, CA), 2015.

[4] P. Stirparo, I. N. Fovino, and I. Kounelis, "Data-

in-use leakages from android memory - test and analysis," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pp. 701–708, IEEE, 2013.

[5] Sun Microsystems, "Memory management in the java hotspottm virtual machine," Apr. 2006.

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th international symposium on Memory management*, pp. 37–48, ACM, 2004.

[7] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2005.

[8] K. Gondi, A. P. Sistla, and V. Venkatakrishnan, "Deics: Data erasure in concurrent software," in *Secure IT Systems*, pp. 42–58, Springer, 2014.

[9] M. Anikeev and F. Freiling, "Preventing malicious data harvesting from deallocated memory areas," in *Proceedings of the 6th International Conference on Security of Information and Networks*, pp. 448–449, ACM, 2013.

[10] M. Anikeev, F. C. Freiling, J. Götzfried, and T. Müller, "Secure garbage collection: Preventing malicious data harvesting from deallocated java objects inside the dalvik vm," *Journal of Information Security and Applications*, vol. 22, pp. 81–86, 2015.

[11] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "Cleanos: limiting mobile data exposure with idle eviction," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 77–91, 2012.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, osdi vancouver," 2010.

[13] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pp. 137–143, IEEE, 2007.

[14] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, pp. 91–98, May 2009.

[15] A. Case, "Memory analysis of the dalvik (android) virtual machine," in *Source Seattle*, Dec. 2011.

[16] T. Müller and M. Spreitzenbarth, "Frost: Forensic recovery of scrambled telephones," in *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, (Berlin, Heidelberg), pp. 373–388, Springer-Verlag, 2013.

[17] C. Hilgers, H. Macht, T. Müller, and M. Spreitzenbarth, "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, (Washington, DC, USA), pp. 62–75, IEEE Computer Society, 2014.

[18] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering c++ objects from binaries using inter-procedural data-flow analysis," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, p. 1, ACM, 2014.

[19] J. Stttgen and M. Cohen, "Robust linux memory acquisition with minimal target impact," *Digital Investigation*, vol. 11, Supplement 1, pp. S112 – S119, 2014. Proceedings of the First Annual DFRWS Europe.

[20] M. Cohen, "Rekall memory forensics framework," in *DFIR Prague 2014*, SANS DFIR, 2014.

[21] S. VöMel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digit. Investig.*, vol. 8, pp. 3–22, July 2011.

[22] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, pp. 169–190, ACM, 2006.

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Iolite: A unified i/o buffering and caching system," in *Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, (New Orleans, LA), Feb. 1999.