Secure garbage collection: Preventing malicious data harvesting from deallocated Java objects inside the Dalvik VM

Maxim Anikeev, Felix C. Freiling, Johannes Götzfried, Tilo Müller

1. Introduction

Today, most people, including both private users and company employees, store highly sensitive data such as email passwords and access credentials on their mobile devices. Contrary to stationary PCs, mobile devices are continuously at risk to get physically lost or stolen. Although the hard disks of mobile devices can be encrypted, main memory cannot be encrypted due to performance issues and technical constraints of the hardware. With physical access to a switched-on device, an adversary can read out everything that is left in main memory, e.g., by so-called cold boot attacks (Halderman et al., 2008), even though a device is encrypted and locked. In 2013, it was shown that the class of cold boot attacks is also effective against Android-driven smartphones and tablet PCs (Müller and Spreitzenbarth, 2013).

1.1. Motivation

We consider an Android email app that stores an IMAP password on the encrypted hard disk, or prompts the user for it each time the app is launched. This app must be able to wipe data reliably from RAM in response to screen lock events to protect the password from cold boot attacks. Unfortunately, with the current Java runtime environment in Android, called the Dalvik VM, it is impossible to reliably remove data, as we have proven by several experiments. In low-level languages like C, deallocation is the responsibility of the application level programmer and it is up to the programmer to decide whether...
an allocated data array should be freed, or whether it should be shredded and zeroed out before. Even though secure deallocation is not provided by OS environments, C software designers are able to override default deallocation routines. But when it comes to languages with embedded garbage collection like Java, secure deallocation becomes impossible. Neither the time of data freeing can be predicted, nor the fact whether data shredding routines affect a certain memory location. Internally, Java can hold several copies of the same instance of a class, as objects may get cloned upon write operations, such that from a Java application programmer's point of view, it is not always easy to track whether overwritten and freed objects are still in RAM or not.

1.2. Related work

The intention to reduce lifetime of confidential data in main memory is a general practice in secure software development (Howard and LeBlanc, 2004). During the past decade, several research papers have been published on the topic of secure deallocation. These papers were not motivated by the threat of cold boot attacks, but by traditional threats like the reuse of uncleared pages in different protection domains or data leakage between processes and virtual machines.

Chow et al. (2004) traced a password typed into a web form on its journey through a system. They discovered copies in a variety of kernel, window manager, and application buffers, as well as copies in the user heap. Even if programmers do their best to minimize data lifetime, their efforts are often futile as the fate of memory is out of their control. A process has no control over kernel buffers, window manager buffers, or even application memory in the event that a program crashes.

Chow et al. (2004) came to conclusion that the problem of secure data deallocation is actually solved in only several products, which were designed by IT-security experts. These products include cryptographic software and password managers. Outside this narrow segment, the situation is not satisfactory. Therefore, they proposed a set of measures they call secure deallocation (Chow et al., 2005). Their approach is based on the idea to introduce specially designed operating system modules that monitor all memory management operations. The modules keep track of various memory regions and erase the data immediately after it became useless, or at least within a short predictable time afterwards. Even though this approach potentially reduces unjustifiably long data lifetime in deallocated memory, it does not help the software developer to define the earliest possible moment in which deallocation should be applied.

Tang et al. (2012) were the first to investigate the problem of secure deallocation within mobile Android devices. Contrary to our approach, Tang et al. identify and track sensitive data in RAM, encrypt it with a secret key and evict that key to a cloud storage when the data is not in active use. They implemented this process, which they call idle eviction of sensitive data, in an Android-based operating system named CleanOS. While CleanOS addresses a similar problem to our approach, it relies on cloud computing and thus, on an active network connection.

1.3. Contributions and overview

We study how secure deallocation can be automatically enforced by the Android runtime system when deallocation is supported at the programming level. The idea is to give the application programmer different methods to declare sensitive data and to define a usage scope, outside of which main memory of sensitive variables is securely erased. As a proof-of-concept, we introduce our deallocation method inside the Dalvik VM. We argue that our proposed approach decreases the risk of sensitive data exposure of mobile apps considerably, if applied properly by the programmer when the software is designed.

More precisely, our contributions are threefold:

• Design (Section 3): We present an outlook of possible strategies to facilitate software development with the awareness of confidential memory leaks, and we give design recommendations for new deallocation concepts.

• Implementation (Section 4): Following our design concepts, we present a proof-of-concept implementation of secure deallocation for the Dalvik VM. We provide this implementation as an open-source patch.

• Evaluation (Section 5): Finally, we present an evaluation of our Dalvik patch regarding its security and performance behavior. We prove the effectiveness of our approach with respect to secure deallocation, and show that the performance overhead is negligible for most practical apps.

Furthermore, in Section 6, we conclude with an outlook about current limitations of our approach and possible future enhancements.

2. Background

2.1. Cold boot attacks

Cold booting a device means to briefly cycle power off and on, without allowing the OS to shut down properly. In contrast to common beliefs, after cold booting a device, main memory contents are not lost immediately because RAM chips exhibit a behavior which is called the remanence effect (Gutmann, 2001; Skorobogatov, 2005). The remanence effect says that RAM contents fade away over time rather than disappearing all at once, and that they fade more slowly at lower temperatures. That is, the colder RAM chips are, the longer the memory contents persist and hence, cold boot attacks are typically more practical when RAM chips have been cooled down.

In 2008, Halderman et al. (2008) have shown that cold boot attacks can be exploited to break software-based full disk encryption since the disk encryption key is stored in clear in RAM. In 2013, Müller and Spreitzenbarth (2013) have shown that Android based smartphones are vulnerable to the same kind of attack, such that adversaries with physical access to a locked phone can recover all of its data which is present in RAM.

2.2. Memory management in the Dalvik VM

The Dalvik VM (DVM) is a register-based Java virtual machine developed by Google for it’s Android operating system.
Basically, the Dalvik VM is an interpreter for Android apps that are written in Java and compiled into Dalvik-compatible bytecode. Dalvik-compatible bytecode is distributed as DEX files (Dalvik executables) which are sets of Java class files. The memory management of the Dalvik VM is similar to what is known from the JVM, meaning that data on the heap is not freed explicitly by the programmer but gets automatically managed by the garbage collector (GC).

Interestingly, most data types in Java are allocated on the heap because everything other than primitive data types are objects in Java, which are created via new, including String objects. That is, unlike low-level languages such as C, which hold null-terminated character sequences oftentimes on the stack, Java strings are always placed on the heap. If an Android application stores a password, this password is placed on the heap by the Dalvik VM.

The DVM garbage collector, which manages the Dalvik heap, runs in a separate thread that is periodically woken up, and can additionally be called from an application level programmer by the System.gc() command. However, calling System.gc() is only a hint for the runtime environment and does not give an explicit order. Furthermore, even though the GC runs and frees unused objects, objects are not securely deallocated. The GC does not overwrite heap memory with zeros at the time of deallocation, but only at the time of reallocation.

Technically, the GC manages objects on the heap by a classic mark-and-sweep algorithm, meaning that all objects are traversed, beginning from a root set, and marked if they are reachable. Objects that are unreachable, i.e., objects that are not marked at the end of the GC routine, are swept. As the DVM itself is implemented in C++, the heap management is based on the dlmalloc algorithm by Lea; variants of this algorithm are also used inside GNU C Library on UNIX systems. The sweep step of the GC is simply implemented by a call to free(), i.e., no explicit data shredding or zeroing is enforced.

3. Design considerations for secure deallocation

Developers of application software often ignore the necessity to remove allocated data from memory in due time. Even low-level programmers in C, who try to selectively implement secure deallocation for certain variables, must be aware of the fact that compiler optimizers can ignore memory shredding instructions because they identify them as useless for the implemented algorithm. Consider the same problem for languages with automatic memory management like Java. The characteristic feature of automatic memory management is the inability to predict when exactly the data is deallocated. As a consequence, methods for secure deallocation must be provided as an explicit but easy-to-use feature of the programming language or runtime environment.

Generally speaking, all data in programs can be classified as static memory and dynamic memory variables. The variables are declared in their visibility scope. Static variables are first declared, then initialized and handled through either reading or writing for indefinite number of times, and finally recycled when program flow leaves their visibility scope. Dynamic variables are normally allocated before initialization and should be explicitly deallocated before their visibility scope is lost. The operation of accessing a variable the last time before deallocation is particularly important because afterwards the value of this variable is no longer important anymore, though poses a threat of exposing sensitive data. That is why it is crucial to correctly identify the usage scope of variables and to erase their values as soon as possible.

In programming languages with explicit memory management like C, developers should have a chance to use a syntactic construct for declaring a subset of sensitive data objects. Without that, all the processed data has to be treated as potentially sensitive, which will complicate writing the source code greatly. An example of syntax for declaring sensitive variables can look like the following hypothetical extension for C, as proposed by Anikeev and Freiling (2013):

```c
sensitive char a[SIZE];
```

The keyword sensitive here indicates that variable a contains confidential information during its lifetime, hence requires secure deallocation when leaving the usage scope of this variable. This concept could easily be implemented as a compiler level extension, but it lacks support for scenarios where a variable should be deallocated before its visibility scope is left.

Hence, a practical solution for secure deallocation is not as easy as declaring a variable as sensitive. It is the responsibility of the software designer to correctly specify the usage scope of sensitive variables, which might be different from their visibility scope. If the usage scope of a variable is managed incorrectly, the system is either unnecessarily vulnerable to memory leakages, or it is causing memory access errors. One way to overcome this difficulty is to add the postfix last_use to the programming language, in addition to the prefix sensitive, indicating that a variable is not used anymore. However, this concept is inconvenient and prone to errors like double free and null pointer dereferences, known from dynamically allocated variables.

For managed programming languages like Java, the situation is different. The GC manages the usage scope of all variables automatically and programmers do not have to manually declare the end of usage of dynamic variables. This fact can be exploited to introduce a flexible and easy-to-use concept for secure deallocation. We propose a novel scheme for secure deallocation in Java that reminds of critical sections in concurrent programming: With System.enableSecureDeallocation() and System.disableSecureDeallocation() a programmer can enter and leave, respectively, a section in which all allocated objects are securely deallocated when they become unused. This concept is highly motivated by the fact that library calls oftentimes allocate internal data objects and thus, library calls often duplicate sensitive data within objects that cannot be controlled by the programmer (as the programmer has no reference to these objects). Calling third party libraries inside a secure deallocation section, that means after calling enableSecureDeallocation(), the runtime environments is aware of all newly allocated objects and securely deallocates them as soon as they are sorted out by the mark-and-sweep algorithm. Additionally, disableSecureDeallocation() calls the
garbage collector implicitly, and manually calling System.gc() assuredly starts the garbage collector.

4. Implementation of secure deallocation inside the Dalvik VM

Note that secure deallocation cannot be implemented on a Java basis because zeroing out allocated memory before deleting a reference to it is not reliably in Java. First, some object types are immutable, like String objects, and just cannot be zeroed out. Second, objects may recursively refer to other objects, such that the definition of secure deallocation of an object becomes unclear. Does secure deallocation mean to recursively shred all referenced objects, or does it mean to shred only pointers to other objects? Consider a custom string object that has a pointer to a byte array holding the actual payload. Shredding only the string object instance would not shred the string, but on the other hand, generally zeroing out all objects in a recursive fashion may have disastrous consequences, e.g., in linked lists.

The only reasonable way to implement secure deallocation for Android apps is to patch the Dalvik VM, particularly its GC routines. As stated in Section 2.2, the DVM is available as an open source project, implemented in C++, such that we were able to modify and recompile its source code. Our DVM patch is a neat modification of only 100 lines of code. Technically, this modification follows the design concept outlined in Section 3, meaning that we introduce calls into the DVM for enableSecureDeallocation() and disableSecureDeallocation(). Whether secure deallocation is enabled or disabled is stored inside the global state of the DVM, called gDvm.enableSecureDeallocation(). By default, this boolean state is false and must be set to true by the programmer by calling enableSecureDeallocation(). If gDvm.enableSecureDeallocation is set to true, the GC routine zeroes out freed heap chunks with a call to memset() before calling free(). The size of each heap chunk can be determined via mspace_usable_size().

Note that calling disableSecureDeallocation() does not only set the global state to false, but that it first calls System.gc() to clean up before leaving a critical section. Basically, an entire Android app can just be declared as critical section within our DVM. Due to the large software stack of Android, with many internal created by third party libraries. To prove the effectiveness of our approach, we ran several tests observing memory dumps of Android processes inside the emulator at runtime. Basically, we examined two test apps, one allocating/deallocating memory with certain strings locally, and one sending sensitive data via network, thus calling third party libraries. Running these apps inside the Android emulator, rather than on a real device, has the advantage that memory dumps can easily be acquired.

We started each app several times in two configurations, with and without secure deallocation sections. In all test runs we observed the same result, namely that we were always able to identify the secret string in main memory without secure deallocation, but that we were never able to identify the string in the process memory after it was securely deallocated. If the string reference was only removed, but not securely deallocating with our method, the string could still be traced. In the network setup, we could even identify each string several times in RAM without secure deallocation. Only with secure deallocation enabled, we were never able to find any sensitive data in the process memory after the corresponding object references were removed.

5. Security and performance analysis

5.1. Security analysis

The advantage to implement secure deallocation by declaring critical sections is that implicit memory, which is allocated by third party libraries, is covered as well. Consider an email app that stores an IMAP password in a String object controlled by the programmer. Whenever the programmer passes this String object to a network library for user authentication with a server, the network library creates network packets and thus it copies the password. When implementing secure deallocation by directives like sensitive, as proposed in Section 3, implicit memory from third party libraries cannot be securely erased without recompiling the libraries. Programmers have no reference to dynamic variables which are internally created by third party libraries.

5.2. Performance analysis

In 2012, Oh et al. (2012) benchmarked the performance of the standard Dalvik VM in comparison to Oracle’s HotSpot JVM. Since the DVM employs register-based bytecode rather than stack-based bytecode, its interpretation performance is slightly better. The JIT performance of the DVM, however, is worse than that of the JVM due to less optimizations and short traces (Oh et al., 2012).

In the work at hand, we compare the performance of memory allocation in the standard DVM to that of our patched DVM. Due to the large software stack of Android, with many side effects like multithreading of an app and the garbage collector, OS scheduling, and possibly the underlying hardware or emulator, we rely on average values over five test runs per measurement. In each test run, we allocate a variable number of bytes in an MB to 1 GB in 25 MB steps, iteratively in 1 MB chunks where each MB is allocated in its own critical section. That is, for our patched DVM variant, each MB is allocated and securely deallocated (i.e., first overwritten with zeros and then freed) whereas for the standard DVM, each MB is allocated and then deallocated only spontaneously by the GC. To prevent compiler optimizations from eliminating these memory allocations, we additionally operate on the allocated objects in a simple manner, e.g., by computing and returning mean values over large array objects.

We were first astonished when we ran our tests on the Android emulator and constantly received higher
performance results for the patched DVM than the unmodified DVM. As it turned out, this behavior is emulator-specific and does not show up on real hardware. An explanation is that the emulator presumably keeps track of zeroed pages, such that explicitly freed pages can be re-allocated faster. On real hardware, however, the Android OS has no possibility to detect whether a page was filled with zeros before. On real hardware, we measure a performance drawback for secure memory allocations of about factor 1.5 (see Fig. 1). We ran our main tests on a Galaxy Nexus device from Samsung and confirmed our results by running similar tests on an LG Nexus 4 and an Asus Nexus 7. As also proven by our tests, the software stack of Android is compatible with our patched DVM. After we replaced the original DVM by our own variant, we were still able to boot the phone, receive calls, browse the web, and more.

As shown in Fig. 1, the average de-/allocation time for secure memory in the patched DVM is about 1.5 times higher than the allocation time for insecure memory in the standard DVM. While this is a notable performance drawback, everyday Android apps do not suffer to such an extent, because a normal Android app does not allocate and deallocate hundreds of MB of secure memory in a row. Programmers are advised to declare secure memory within a critical section with `enableSecureDeallocation()` and `disableSecureDeallocation()`, respectively.

6. Conclusions

Sensitive data, such as passwords, private information, and confidential documents, often remain in random access memory for indefinite time, which increases the risk of exposure. To defeat the leakage of confidential data on Android systems, e.g. due to cold boot attacks, user mode applications must be provided with a method to securely deallocate memory after usage. To this end, we have proposed a novel approach for secure deallocation in the Dalvik VM based on critical sections that can be entered and left with `enableSecureDeallocation()` and `disableSecureDeallocation()`, respectively.

6.1. Limitations

We identify two notable limitations of our implementation: The first concerns OS caching effects, like file system buffers and keyboard buffers, which cannot be counteracted on the Dalvik level. To rule out these effects, the OS kernel of Android must additionally be patched, as outlined by Chow et al. (2005). The second limitation concerns shared memory which is not zeroed by our approach because this type of memory is always marked as in use by the mark-and-sweep algorithm. This corner case could be counteracted by a special treatment of sensitive shared memory areas, but so far we advise the programmer not to store sensitive data in shared memory areas.

6.2. Future work

Further research in the direction of secure deallocation can be conducted by the implementation of our approach in more programming systems. For example, we plan to patch the designated successor of the DVM, called ART, that was recently introduced with Android 4.4. Moreover, we are planning to patch open-source Android apps, like email clients, to benefit from our secure deallocation approach.
Afterwards, extended security and performance tests are planned to confirm our results.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89) and by the joint “Mikhail Lomonosov” program of German Academic Exchange Service (DAAD) and the Ministry for Education and Science of the Russian Federation (A/10/76208).

REFERENCES


学霸图书馆

www.xuebalib.com

本文献由“学霸图书馆-文献云下载”收集自网络，仅供学习交流使用。

学霸图书馆（www.xuebalib.com）是一个“整合众多图书馆数据库资源，提供一站式文献检索和下载服务”的24小时在线不限IP图书馆。

图书馆致力于便利、促进学习与科研，提供最强文献下载服务。

图书馆导航：

图书馆首页 文献云下载 图书馆入口 外文数据库大全 疑难文献辅助工具