An Experimental and Industrial Experience: Avoiding Denial of Service via Memory Profiling *

Saeed Abu-Nimeh, Suku Nair, and Marco Marchetti
HACNet Lab
Southern Methodist University
Dallas, TX 75275
{sabunime, nair, marco}@engr.smu.edu

Abstract

Poor memory management leads to memory leaks, which cause significant performance degradation and failure of software. If ignored, such leaks can potentially cause security breaches and holes in applications. The present study shows that memory leaks can be exploited to cause Denial of Service (DoS) of applications. The ultimate goal of this study is to introduce a security profiling technique that can be used to identify security holes in software. We instrument memory leaks in a Java applet using an open source memory profiler based on Java Virtual Machine Profiler Interface (JVMPI). The results show that it is crucial to perform memory profiling prior to application deployment in order to avoid DoS and vulnerability exploits.

Keywords

Denial of Service, JVM, JVMPI, memory profiling, memory leak, garbage collection.

1. Introduction

In spite of garbage collection (GC) presence, memory management plays a vital role in software development. GC does not (and in general cannot) collect all the garbage produced by the program [28]. GC has some benefits, such as: eliminating some memory management bugs, such as dangling pointers and improving modularity [30]. Many new programming languages such as: Java [18], Modula-3 [26], and SML [25] require garbage collection and use automatic storage reclamation [22]. In older languages such as C and C++, GC is not a part of the language definition; however programmers use GC for reclaiming memory or for leak detection [21]. Several garbage collection algorithms are used by the Java Virtual Machine (JVM). First, mark and sweep algorithm. This algorithm involves two phases: marking and sweeping. The marking phase marks reachable objects through either a depth- or breadth-first search on the object reference graph. Second, generational collectors. In this algorithm collectors divide objects into groups based on their age. GC for young-generation objects involves minimal marking effort. On the other hand, GC in old generations could need full marking. Third, reference counting algorithm. This algorithm associates each object in the reference graph with a reference count, which records the number of objects that can access the object. Once the reference count reaches zero, the corresponding object is garbage. Fourth, copy collectors. Here collectors trace garbage but don’t mark it. Once a copying collector identifies live objects, it copies those objects to another memory space to achieve memory compaction, thus eliminating memory fragmentation. The original memory space becomes free automatically. After compaction, memory allocation is very fast. Fifth, mark-compact garbage collection. This involves marking and compacting. The marking phase is similar to that of the mark-and-sweep algorithm. Once the marking phase finishes, live objects are moved side by side in the memory. Both mark-compact and copying algorithms achieve heap compaction through copying. However, copying garbage collection has low memory utilization. Sixth, stop-the-world garbage collector. The collector suspends all program threads when garbage collector threads are running. This suspension is required to prevent the program threads from modifying the object reference graph. Although this pause results in no progress in the program execution, it is useful in synchronizing program execution with garbage collection. Stop-the-world garbage collection is not the only garbage collector to induce a garbage collection pause.
Other algorithms such as the copying collector must stop the program threads during copying. Moreover, the mark-compact algorithm requires suspending program threads in its marking and copying phases [24].

An ideal garbage collector or leak detector would identify all heap-allocated objects that are not dynamically live. A dynamically-live heap object is one that will be used in the future of the computation and that can be reached by subsequent pointers that will be de-referenced in the future of the computation [23]. The ideal garbage collector must exactly identify what memory locations contain dynamically-live pointers in order to retain only dynamically-live objects. Yet, a real garbage collector or leak detector can not know what pointers will be de-referenced in the future; thus it may use compiler support to identify an approximation to dynamically-live pointers [22].

CERT/CC [7] defines Denial of Service (DoS) attack as an explicit attempt by attackers to prevent legitimate users of a service from using that service. According to CSI/FBI Computer Crime and Security Survey [12] the percent of DoS attacks was 17% of total attacks in 2004. As shown in Figure 1, the percentage of attacks decreased compared to previous years. Figure 2 shows that DoS attacks caused losses estimated by $26 million in 2004 compared to approximately $66 million total annual losses in 2003. Although there was a decrease in the percentage of DoS attacks in 2004, the number of DoS attacks has increased during the first six months of 2005 by more than 679% compared to the last six months of 2004. On average, the number of DoS attacks grew from 119 to 927 per day, over the previous reporting period. Figure 3 shows the increase in number of attacks per day from Jan 1, 2004 to June 16, 2005 [29].

This paper examines the effect of memory leaks on security vulnerabilities. The experiments show that memory leaks in Java applications, cause a huge JVM memory waste. If such applications are web-based systems, these leaks can be exploited to cause DoS of the server and application.

Memory leaks can be identified by memory profiling of applications. Memory profiling is the process of characterizing a program’s memory behavior by observing and recording its response to specific input sets, and then using the relevant aspects of the behavior to guide memory optimizations [31]. In our experiments, we use an open source memory profiler to identify memory leaks in a Java applet.

Our results demonstrate the benefit of memory profiling prior to deployment stages that can identify memory leaks specifically in web-based applications. This helps in avoiding DoS and preventing other security exploits in the future. The rest of this paper is organized as follows: Section 2 discusses related and previous work. Section 3 presents the experimental set up, the test environment and the tools used. In section 4 we show our experimental results. Section 5 presents a discussion of the experimental results and how to avoid DoS using memory profiling. In section 6 we draw conclusions and motivate for future work.
2. Related Work

In this section, we discuss two research areas that are related to our study: memory profiling and memory leak detection.

Calder et al. [14] proposed a data object placement algorithm, called Cache-Conscious Data Placement CCDP. In CCDP a profile is generated listing all data objects encountered during execution along with information of data objects’ reference count, size, and other life-time information. Rubin et al. [27] proposed an efficient profile-analysis framework for data layout optimizations. They built a memory profiler that collects a data-object trace. A unique global identifier object ID assigned to an elementary piece of data, which can be a field of a record, a global variable or any other single-access granularity objects, is used along with the raw address as a pair to identify memory references.

In [15] Chilimbi proposed an efficient representation of the memory access stream for the purpose of quantifying and exploiting data reference locality. He proposed an address abstraction profile which abstracts the address to a name or an identifier. This identifier pair does not have a size or a type associated with it and it does not distinguish the individual fields within a heap object.

Wu et al. [31] proposed a profile with emphasis on object-relative translation, decomposition, and compression in the context of a wider set of optimizations. They defined objects as a part of the hierarchy of (group, object, offset) tuples, which yields an explicit view of the data structures, the fields, and their common properties. Also, their scheme factored out the raw addresses from the representation, which made identifying patterns easier and the compression rates higher.

Memory leak detection can be divided into three approaches. First, static analysis tools such as [20]. Heine et al. [20] proposed a static analysis tool that can automatically find memory leaks and deletions of dangling pointers in large C and C++ applications. They provided a way to identify leaking sites before running the program and before the program is deployed. Their technique did not cause any runtime overhead, yet lack of dynamic information led to false positives and partial leak detection.

Second, dynamic analysis tools, such as: Quest jProbe Memory Debugger [1] which pinpoints memory leaks in Java code and reduces application memory usage. It provides a visual of memory usage, object allocations and garbage collections. It tracks down all references between objects and measures the memory impact of a memory leak or code change. Also, there are Borland Optimizet Profiler [2] and Jprofiler [10]. These tools take a snapshot of the heap and allow the interactive analysis of memory consumption. In addition, they provide statistics about the amount of memory allocated and object sizes in runtime environments.

Third, automatic dynamic tools, which provide a list of leaked objects and leak sites at the end of the program run, such as: LeakTracer [13] and Zorn et al. [32] who instrumented the allocation functions to capture immortal objects. However, Hastings et al. [19] instrumented the allocation functions, and used a mark and sweep garbage collection approach [8] to capture unreachable objects. In addition, Dion and Monier [9] built a tool Third Degree that performs memory access checks and memory leak detection of C, C++, and Fortran programs at run-time. The tool is used to determine if any memory locations are accessed when not properly allocated or initialized. Insure++ [3] an automated runtime application-testing tool that detects memory leaks. During compilation, Insure++ reads and analyzes the source code. It then inserts test and analysis functions around every source code line. Insure++ builds a database of all program elements. At runtime, Insure++ checks each data value and memory reference against its database to verify consistency and correctness.

Hirzel et al. [21, 22] showed that the quality of the results of a reachability-based leak detector depends on the type and liveness accuracy of its reachability traversal. Chilimbi et al. [16] proposed SWAT an automatic dynamic memory leak detection tool. SWAT traces program allocations/frees and constructs a heap model using an adaptive profiling infrastructure to monitor loads/stores of objects with low overhead. The tool captures leaks based on object accesses. They assessed the liveness of objects by observing accesses to those objects.

Ding et al. [17] proposed phase boundaries as the gates to monitor and control the memory usage. They presented three techniques: memory usage monitoring, object lifetime classification, and preventive memory management. They used phase-level patterns to predict the trend of the program’s memory demand, identify and control memory leaks, and also to improve the efficiency of garbage collection.

Most of the related work focused on memory profiling and memory leak detection as a solution to memory management, memory footprint and the memory performance. Yet, this study focuses on using these techniques as a security tool that is used in DoS avoidance and other security flows prevention.

3. Experimental Setup

In this section we discuss the application description, the tools used, and the hardware specification.

We use an open source Java Memory Profiler (JMP) version 0.47 [4] to perform our experiments. JMP is a profiler for Java that can be used to trace objects usage and method timings. It uses the JVMPI interface to gather sta-
tistics and interact with the JVM. It shows the classes in memory with summary information for number of instances and total bytes of used memory for all the instances. It also, performs heap analysis and has the ability to show which objects own (have references to) all the objects of a specified class [4].

We use Java 2 Platform, Standard Edition (J2SE) version 5.0 update 03 with Java Runtime Environment (JRE) version 1.5.0_03. We test on two browsers: Internet Explorer 6.0.2800 and Firefox 1.0.4. All experiments are performed using a single client PC running dual operating systems: Windows XP Professional (service pack 2), and Free BSD version 5.4 with Pentium 4 processor, 2.8 GHz, 1 GB of RAM, and 40 GB H.D.D. The client hardware has 33 MHz PCI 32-bit buses and one 10/100 MB Ethernet adapter.

We test a forecast system Forecast developed at Dallas Semiconductor [5]. In Table 1 we summarize the system platform and database system specifications.

Forecast is a web-based intranet application, which is used by Business Unit Finance, Business Unit Managers, and Sales Managers. The managers are responsible for entering and maintaining forecast data for their products. The Business Unit Finance Analysts analyze and review the data on weekly and quarterly basis with the managers, by running analytical and summary reports to help assess the business. The application consists of two main parts: a web server and a web-based interface on the client side. The interface contains an applet that is used to access and maintain forecast data in the database. The applet performs Excel [11] sheet-like functionality. Data is loaded into the applet based on user’s access to certain part numbers. All parts are displayed from business units down to detail product lines. Additional filtering can be applied by part’s rank or special discrepancy definition.

The applet contains two major components, a navigation tree and a data grid. The data grid is a third party library. The name of it is anonymized to protect the privacy of the producing company. We refer to it as F1 data grid. At startup data is loaded from the database to populate the navigation tree with enterprise, divisions, business units, product lines, and detail product lines that are accessible based on username.

Next, forecast data for parts from the first business unit node are loaded into the lower part of the F1 grid. Afterward the actual data points for the first part loaded are loaded into the top part of the F1 grid. The user is allowed to modify forecast units for forecasted parts only. Cells are highlighted based on variance of change, red for greater than 25% and yellow if less than or equal to 25%. Upon completion of modifications the user can save updates to the database at which time the highlighting is removed for all updated cells. The user can traverse the navigation tree to load groups of parts based on business unit, product line, or detail product line. A dropdown box is located on the display to allow further filtering by rank or discrepancy.

The applet has a timer thread running to keep track of the last time the database has been accessed. After a predetermined time the database connection is closed automatically. Users can re-login by using menu option to enter password. Particularly, we perform memory profiling for the applet, as it is the performance bottleneck in the application. We perform the tests using the default JVM memory allocation pool parameters, and then we modify the parameters to be able to examine more observations.

### Table 1. Forecast application specifications.

<table>
<thead>
<tr>
<th>System Platform</th>
<th>Operating System</th>
<th>Application Server</th>
<th>Programming Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Sun Solaris 9</td>
<td>Oracle 10g AS</td>
<td>J2EE application</td>
</tr>
<tr>
<td>Database System</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Sun Solaris 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBMS</td>
<td>Oracle 10g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDBC Connections</td>
<td>J2EE 1.4 complaint, Oracle JDBC API supplied with JDeveloper 4.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In Figure 4, we profile the applet using the default JVM parameters [6]. We initiate the applet twice. After the second initiation the JVM crashes giving "Out of memory Error: Java heap space". In Figure 5, we modify the JVM parameters before loading the applet. We change the default JVM parameters to be able to initiate more instances of the applet. In Figure 6, we initiate the applet and observe the memory leak in the applet. We test the identified class in a lightweight applet. We initiate the applet and observe the memory usage.

4. Experimental Results

In this section, we present our experiments of testing an applet in Forecast application. We perform memory profiling of the applet using JMP. The goal of these experiments is to define memory leaks that exist in the applet and cause performance degradation of the application. In Figure 4, we profile the applet using the default JVM parameters [6]. We initiate the applet twice. After the second initiation the JVM crashes giving "Out of memory Error: Java heap space". In Figure 5, we modify the JVM parameters before loading the applet. We change the default JVM parameters to be able to initiate more instances of the applet. In Figure 6, we isolate one of the objects that we expect is causing a memory leak in the applet. We test the identified class in a lightweight applet. We initiate the applet and observe the memory usage.
ory allocation of the object. We initiate the light weight applet for a second time. As shown in Figure 7, the heap space is decreasing (pink or light area) and the allocated used space (by all objects including the identified object) is increasing (blue or dark area). In Figure 8, the size of the allocated space increases and the available heap space decreases when the applet is initiated for the third time. GC is automatically called by JVM. As shown in Figure 9, unused objects are cleaned, however there is a small amount of memory still allocated to the identified object.

5. Discussion

The previous experiments explore memory profiling to identify memory leaks in an applet in the Forecast application. The results show that multiple initiations of the studied applet lead to high CPU usage and cause the JVM to crash. In Figure 3, we use the default JVM parameters "-Xms" and "-Xmx". "-Xms" specifies the initial size, in bytes, of the memory allocation pool and the default value is 2 MB. "-Xmx" specifies the maximum size, in bytes, of the memory allocation pool and the default value is 64 MB [6]. In Figure 4, the total area of the heap is represented by the light color (pink) and the area that is used of the heap (by all applet classes) is represented by the dark area (green). When the applet is initiated, the used area increases till it reaches a peak where the classes of the applet are loaded and an automatic garbage collection is forced by the JVM. Thus, the used area stabilizes till the second initiation occurs. Now, the used area increases till the size of applet objects exceeds the total heap space. As a result, the JVM crashes giving "Out of Memory Error: Java heap space". After the JVM crashes, the used area decreases and stays stable.

To overcome the "Out of Memory" problem, we increase the initial and the maximum sizes of the memory allocation pool in Java. We modify the parameters in the Java plugin to "-Xms256m" and "-Xmx256m". The initial size of the memory allocation pool is changed to 256 MB. Likewise, the maximum size of the memory allocation pool is changed to 256 MB. Interestingly, we could launch the applet more than two times. As shown in Figure 5, the applet is successfully launched for four times before the JVM halts. Every time the applet loads all classes, the JVM automatically garbage collects all unusable objects till the next initiation occurs, where the used area increases again. When the forth iteration occurs, the size of applet objects exceeds the maximum heap size, hence the JVM stops responding.

To force GC after every initiation, we close the applet window yet keeping the process running. In addition, we manually call GC from Java console. Interestingly, the applet does not release allocated memory. The amount of memory used does not decrease, even after we call GC. After the next initiation, the new amount of memory allocated by the applet builds up on the previous amount of the last initiation. Figure 10 shows the amount of memory consumed after each initiation. Obviously, there exists a memory leak in the applet, otherwise the amount of memory allocated by each initiation should have been cleared after the applet is terminated and the objects are garbage collected. In order to identify the leak, we track the objects that have the highest memory consumption. We find that the object com.flj.ss.gw has the most memory allocated among all objects. To support our observations, we find that the size of the object gw
increases gradually after each applet initiation and the object is not affected by GC when GC is forced. Figure 11 shows how the size of gw object increases after every initiation of the applet. To clearly identify that the memory leak is caused by gw object, we isolate gw class in a lightweight applet and perform memory profiling for that applet. In Figure 6, Figure 7, and Figure 8 multiple instances of the lightweight applet are run. The allocated memory keeps increasing till GC is called automatically by the JVM as shown in Figure 9. Although GC is called, the size of allocated memory by gw object stays the same. Figure 12 shows the amount of memory allocated by gw object.

5.1. DoS Scenario

The previous results show that there is a memory leak in Forecast applet caused by com.f1j.ss.gw object. If multiple users try to access a web application that contains an applet similar to the one studied in the previous experiments, this might cause a memory stall (refer to Figure 10) in the production server leading to DoS in the application and the server. This shows the important role of application infrastructure and the application design in the security of the application. A small amount of 23 KB memory leak (see Figure 12), when deployed on the production server, can result in huge 50 MB memory waste (see Figure 10).

6. Conclusion and future work

The study presented an analysis of memory profiling to detect memory leaks in JVM. We tested a web-based application developed at Dallas Semiconductor. We profiled an applet, which is built on a third party library to perform an excel sheet functionality. The applet performed extensive calculations depending on algorithms that retrieved data from several database servers to generate forecasts of business units, product lines, and detail product lines. The results showed the role of memory profilers in identifying memory leaks in software. Hence, we identified a memory leak in one of the objects in Forecast application applet. We isolated the identified class and tested it using a lightweight applet. There was a memory leak of approximately 23 KB in one object of the third party library used in the applet. After using the library in the applet the leak grew to 50 MB of memory waste. This showed how a small amount of memory leak can result in a huge memory waste.
causing DoS of applications and servers. The observations confirmed the vital role of software infrastructure and design in system security.

As a result, we recommend using memory profilers as an early alarm against DoS. It would be useful for companies and businesses to perform memory profiling for their web applications prior to the deployment phase. Therefore, they will be able to determine and define all memory leaks that might occur in an application during earlier stages of the design phase. Fixing memory leaks plays a major role in avoiding DoS after the application is on the production server.

The results motivate future work to develop new profilers that provide alarms of potential DoS vulnerability in an application. Memory profilers and leak detectors are well-known tools to experiment memory management, memory footprint, and memory performance, yet not in DoS avoidance. Future work will focus on memory profilers as DoS detection tools. In general, future work will explore the need and use of memory profilers as security profilers.

Acknowledgement

This work was completed on an internship at Dallas Semiconductor, Dallas, TX. We would like to thank the management and staff at the MIS department for this opportunity.

References
