Towards Patching Memory Leak Bugs in Off-The-Shelf Software

Domenico Cotroneo, Roberto Natella
Consorzio Interuniversitario Nazionale per l'Informatica (CINI)
Università degli Studi di Napoli Federico II, Italy
{cotroneo, roberto.natella}@unina.it

***Abstract*—Static and dynamic analysis techniques for bug detection have significantly improved in the last decades, and are today implemented in industry-strength tools and routinely applied by developers. Nevertheless, it is still difficult to deal with bugs located in OTS software, since developers lack the source code and/or knowledge about their internals to fix these bugs. In this paper, we propose an approach for fixing memory leak bugs in OTS software, that leverages dynamic binary analysis tools to find bugs, and binary code rewriting to patch them. Patching will allow to rejuvenate OTS-based software less frequently, thus further improving the availability of applications using this approach. Future work will implement this approach in a prototype, and validate it on real memory leaks found in complex software.**

Keywords*—*Memory leaks; Off-The-Shelf software; dynamic analysis; static analysis; binary analysis; software aging; software rejuvenation

I. Introduction

There are software bugs that are very difficult to find and to fix. This is the case, for instance, of memory management and synchronization bugs, that surface as software failures only after a long period of execution. As pointed out by Huang et al. in their seminal work [1], the complexity of these bugs, and tight time and budget constraints, hinder developers in finding and fixing them before a software release. This problem is even worse when the bugs are located in reused and Off-The-Shelf (OTS) software, for which developers do not have access to the source code: in this case, it is not possible for developers to fix the root cause of a bug.

To deal with these bugs, developers and researchers have been devising practical fault tolerance approaches to mask them. For instance, *software rejuvenation* can prevent failures by proactively cleaning (e.g., by restarting) the software at a convenient time [1]. This kind of approaches work around many tricky bugs without fixing them, at the cost of small period of downtime during recovery and rejuvenation [2], [3].

Today, after several decades, we need to consider that bug detection techniques significantly improved since early studies on fault tolerance and software rejuvenation. Several techniques have been developed for static and dynamic code analysis, that are able to discover classes of subtle bugs such as race conditions, deadlocks, memory leaks, integer and memory overflows [4], [5], [6]. These techniques are now regularly adopted by developers, and are implemented by industry-strength tools (both commercial and open-source) [7], [8], [5] and included in development toolchains (for instance, modern compilers includes rich set of advanced static and dynamic analysis techniques for early bug detection) [9], [10]. Some of these techniques can even be applied on binary code.

As a result, developers are now able to detect and fix tricky bugs that were previously out of reach to them. Nevertheless, it is still difficult to deal with bugs located in OTS software. Even if developers are able to pinpoint these bugs with automated tools, they lack the source code and/or knowledge about software internals needed to fix these bugs [11]. Thus, for this case, even if the bugs are known, software rejuvenation and fault tolerance still represent the sole viable paths.

In this paper, we propose an approach for fixing memory leak bugs in OTS software. The approach aims to patch OTS software automatically, and without requiring source code or other knowledge about its internals. The idea is to leverage dynamic binary analysis tools to find memory leaks, and binary code rewriting to inject memory management code for preventing the leaks. Reducing memory leaks by patching OTS code will allow to rejuvenate software less frequently, thus further improving the availability of applications using this technique. The goal of this paper is to present the ideas behind the approach, and discuss the challenges that need to be faced. Future work will implement this approach in a prototype, and validate it on real memory leaks found in complex software.

This paper is structured as follows: Section II briefly describes dynamic binary analysis for detecting memory leaks; Section III discusses our proposed idea for patching OTS code; Section IV discusses related work; Section V concludes with directions for future work.

II. Dynamic binary analysis

There are several tools for static and dynamic analysis of binary code, aimed at testing, security and reverse engineering purposes. In this section, we focus on Valgrind [5], a dynamic binary analysis framework that is widely used by developers for detecting and debugging memory corruption and concurrency issues. For instance, MySQL developers adopt Valgrind in their automated regression tests, in order to timely detect memory leaks and other memory-related issues [12].

Valgrind is a *framework* for supporting several types of code analysis at the binary level. This framework consists of the Valgrind core, and a set of *tool plug-ins* (each implementing a specific type of analysis) based on the Valgrind core. The most well-known Valgrind tool is *MemCheck*, which is aimed at detecting memory-management problems in C and C++ software [13], [14]; other Valgrind tools include *Cachegrind*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void print_hello(void) {

  char * string = (char *) malloc(sizeof(char)*12);

  strcpy(string, "Hello World");

  printf("%s\n", string);

  return 0;
}
...
```

```
HEAP SUMMARY:
    in use at exit: 12 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 12 bytes allocated

12 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x400555: print_hello (test.c:7)
   by 0x400585: main (test.c:18)

LEAK SUMMARY:
   definitely lost: 12 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

Fig. 2. An example of trivial memory leak bug, and report from Valgrind/MemCheck. Please note that source-code locations in the report are provided for readers' convenience, but no source-code information can be assumed to be available for OTS software.

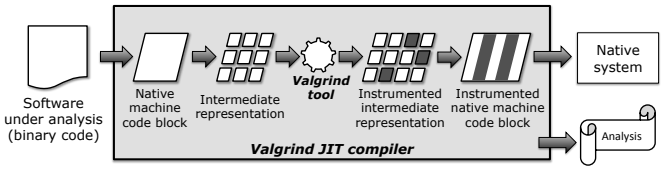

Fig. 1. Overview of dynamic binary analysis in Valgrind.

(a cache profiler), *Helgrind* (a data race detector), and *Massif* (a heap profiler).

In Valgrind, the software under analysis is interpreted by a JIT compiler, in a similar way to a JVM interpreting Java bytecode (Fig. 1). The JIT compiler dynamically translates native code (e.g., x86 machine instructions in a binary program) into an *intermediate representation* (IR), which is an architecture-independent language with simple (RISC-like) instructions and amenable to instrumentation an analysis. IR code fragments are then passed to a tool plug-in (such as MemCheck), which injects additional IR instructions in the code fragment for tracing and analysis purposes. Finally, the instrumented IR code fragment is translated into native code and executed by the CPU. Compared to other binary instrumentation tools, such as Pin [15], which instrument the native code without intermediate translations, the dynamic translation performed by Valgrind enables more powerful types of analysis, such as shadow-tracking every memory bit (it is thus denoted by its authors as a "heavyweight" binary instrumentation tool [5]). However, this flexibility incurs in a overhead: for instance, programs executed under MemCheck run about 10–30x slower than normal.

The MemCheck tool plug-in uses this dynamic binary instrumentation mechanisms to trace operations that allocate and de-allocate heap memory, as well as memory accesses made by a program, in order to detect memory leaks (i.e., blocks of heap memory that are not referenced by any pointer, and are thus unreachable and leaked) and read accesses to uninitialized memory. MemCheck injects tracing instructions after each instruction that accesses memory, and after each instruction that allocates or de-allocates heap memory. For instance, the injected instructions record the location of every live heap block in a hash table; with this information, MemCheck can detect bad or repeated frees of heap blocks, and memory leaks [14]. Fig. 2 shows an example of report produced by MemCheck for a trivial memory leak: in this example, a memory leak occurs at the end of the `print_hello()` function, in which the pointer to the string allocated in heap memory is lost and is not more reachable by the program. Using information collected by the instrumented program, MemCheck provides a summary about the amount of leaked memory, and the location in the program where leaked blocks were allocated.

Finally, it is important to note that while this kind of analysis (and, in general, static/dynamic analysis) cannot find every possible memory leak bug in a program (in the case of MemCheck, the leak must occur during execution to be pinpointed), it proved to be useful at identifying several subtle bugs in practice, even in mature and complex software [12].

III. Binary Patching Approach

Our approach is aimed at fixing memory leak bugs in binary code and in an automated way, that is, by avoiding human involvement and the need for source code and other information about software internals. The idea (see also Fig. 3) is to collect feedback from program analysis tools about memory leaks, and to inject additional memory management code where necessary into a binary program, to force the deallocation of a leaked memory area once the last pointer to that area is invalidated.

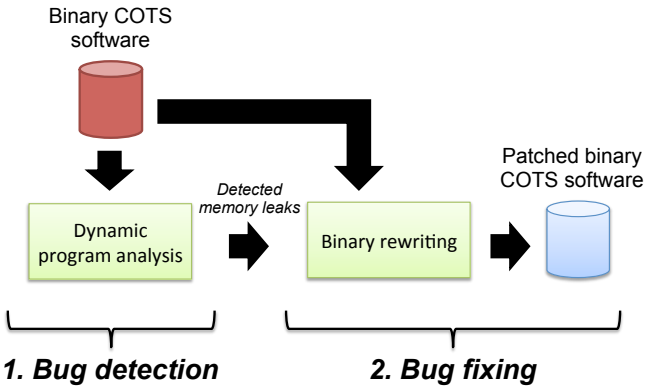

Fig. 3. Overview of the proposed binary patching approach.

To be applicable to OTS software, the approach should only rely on results from binary analysis (such as MemCheck). Therefore, for each detected memory leak, we will only require: (i) the code instruction that allocated a leaked area, and (ii) the last code instruction after which the the memory

leak occurs. This information can be used to (i) insert code right after the allocation of the heap area (*allocation site*) that records the address of the area, and (ii) insert code right before the loss of the pointer to the heap area (*loss site*) that forces the de-allocation of the area.

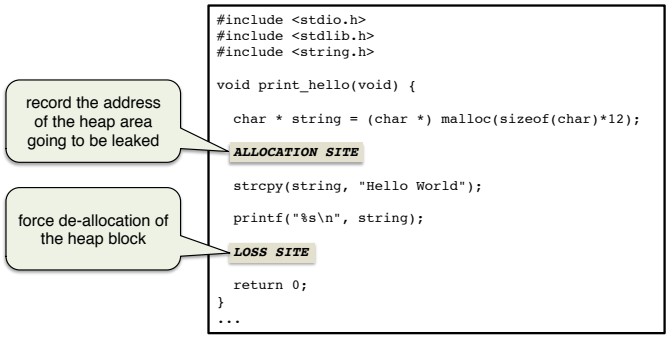

Fig. 4. An example of binary code patching through the injection of memory management code.

For instance (see Fig. 4), the loss site can be the end of a function, in which the heap area is pointed by a local pointer variable, and the pointer is lost when the function ends. Another case is when the pointer is stored in a data structure in heap memory, and this data structure is deallocated without deallocating the area pointed by this pointer (for instance, a C++ class that does not deallocate a heap area in its destructor); the loss site is the code location in which the data structure (e.g., the object of the C++ class) is deallocated. Finally, a third case is when a pointer, either in local, heap, or global memory, is overwritten with a new value without deallocating the heap area pointed by the old value; in this case, the loss site is before the instruction that overwrites the pointer. All these types of loss sites can be detected by a dynamic memory analyzer such as MemCheck, which tracks the life of a pointer to heap memory from the allocation site to the loss site in order to detect memory leaks.

The injected code must strive for a very low overhead, since it is meant to be executed in production, in order to avoid memory leaks at run-time and thus reduce software rejuvenation. Since our memory management code will be significantly simpler than tools such as MemCheck, existing "lightweight" binary rewriting techniques, that inject machine instructions either dynamically (such as in Pin [15]) or statically (such as in PSI [16]), will suffice for our purposes. These techniques can be adopted to introduce small portions of memory management code into allocation and loss sites of memory leak bugs in the program. We expect that the overhead of this code will be low since: (i) In most cases, only a small minority of heap areas are leaked, and only leaked heap areas need to be tracked and freed by injected code. (ii) Injected code will be executed only when the the memory area is *allocated* and when its pointer is *loss*; memory management code will run seldom, and *will not be involved* during the program reads/writes to the leaked memory areas. We also expect that this approach will lead to significantly lower overhead than using an automatic, full-fledged garbage collector [17], [18], since our memory management code will focus on specific heap areas that we know to be affected by leaks: as mentioned above, this information will be provided us by a memory leak analyzer such as MemCheck.

Special care is needed to deal with heap areas that are pointed by more than one pointer. In such cases, the heap area must not be forcefully deallocated when there are still valid pointers to the area, otherwise the program will not be able to access to the area even if it is still reachable. It is thus necessary to deallocate the area only when it is actually leaked, that is, all pointers to it have been removed (e.g., overwritten with a new value or de-allocated). To do so, the approach will need to inject memory management code for each pointer, respectively when each pointer is initialized and is removed, using a *reference counting* mechanism: when a pointer is removed, a counter associated to the heap area is decremented, and when the counter reaches zero, the heap area is deallocated. To implement this mechanism, the memory leak analysis must be extended to provide information about all pointers to the leaked heap area (in particular, the code location in which these pointers are initialized and removed). These pointers can be found by means of static analysis [19]. Even if static analysis has intrinsic limitations (e.g., in some cases, it may not be able to track inter-procedural propagation of pointers through function pointers), it will still allow to fix those bugs for which propagation of pointers can be precisely tracked. The degree of effectiveness of this approach, however, has to be validated by applying it on memory leak bugs actually experienced by users and developers.

IV. Related work

Binary code rewriting has been proposed in past studies for making OTS software more robust against external threats and to improve its security, and we discuss here some relevant examples of binary rewriting for security purposes. SecondWrite [20] is a binary rewriting tool aimed at retrofitting security checks in binary COTS software. It has been used to prevent attacks that affect the control flow, by inserting "stack canaries", by eliminating base pointers, by adding code for checking return addresses and indirect function calls. Several other tools [21], [22] adopt binary code rewriting to *randomize* elements of a program and to make it more difficult to attack, such as randomizing the ordering of code blocks and data, the layout of stack and heap memory, and the allocation of variables to registers. In [23], automatic black-box testing and code instrumentation are adopted to improve the robustness of web services, by discovering inputs that are not gracefully handled, and by filtering these inputs through protective wrappers, that are introduced in Java bytecode using an aspect-oriented programming framework. Finally, binary rewriting has been applied to protect against untrusted binary programs obtained from external sources: in Reins [24], COTS x86 binaries are rewritten to prevent damages to the file system or network, and corruptions of trusted modules, by redirecting system API calls to a policy-enforcement library and by protecting it from attacks that affect the control flow. All these security applications are not meant to fix known bugs into binary software, but to make software more robust against external attacks, and to sandbox untrusted applications.

Binary rewriting has also been applied to perform "live updates" of software, in order to apply patches in production without restarting neither applications nor the OS [25], [26]. Live updates differs from our work, since we do not expect the availability of the source code, and do not require developers

to provide bug-fixes, as we take advantage of dynamic binary analysis to automatically discover and fix memory leak bugs.

V. Conclusion and future work

In this paper, we discussed an approach for fixing bugs in OTS software, using dynamic binary analysis and rewriting techniques. This approach has the potential to reduce the impact of memory leaks caused by OTS software, in an automated way and with a small overhead. In future work, we will implement this approach in a prototype, by extending existing static and dynamic analysis tools such as Valgrind. Moreover, we will apply the approach on complex software with known memory leak bugs, in order to evaluate the overhead introduced by the approach and its ability to deal with these bugs. Finally, we will investigate how to extend the approach to fix other types of bugs, such as race conditions and deadlock bugs.

Acknowledgment

This work has been partially supported by the TENACE PRIN Project (n. 20103P34XC) and by the SVEVIA PON Project (PON02 00485 3487758) funded by the Italian Ministry of Education, University and Research.

References

[1] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE, 1995, pp. 381–390.

[2] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 235–248.

[3] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[4] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[6] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 760–770.

[7] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240–253, 2006.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[9] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 2009, pp. 62–71.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[11] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, 2010.

[12] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 178–187.

[13] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *USENIX Annual Technical Conference*, 2005, pp. 17–30.

[14] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 65–74.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[16] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2014, pp. 129–140.

[17] B. Willard and O. Frieder, "Autonomous garbage collection: resolving memory leaks in long-running server applications," *Computer Communications*, vol. 23, no. 10, pp. 887–900, 2000.

[18] T. Tsai, K. Vaidyanathan, and K. Gross, "Low-overhead run-time memory leak detection and recovery," in *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*. IEEE, 2006, pp. 329–340.

[19] Y. Park and B. Goldberg, "Static analysis for optimizing reference counting," *Information processing letters*, vol. 55, no. 4, pp. 229–234, 1995.

[20] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in cots software with binary rewriting," in *Future Challenges in Security and Privacy for Academia and Industry*. Springer, 2011, pp. 154–172.

[21] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[22] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *Security Privacy, IEEE*, vol. 12, no. 2, pp. 28–35, Mar 2014.

[23] N. Laranjeiro, M. Vieira, and H. Madeira, "A Technique for Deploying Robust Web Services," *IEEE Trans. Services Comput.*, vol. 7, no. 1, pp. 68–81, 2014.

[24] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 299–308.

[25] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 187–198.

[26] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 279–292, 2013.