Practical Bounds Checking Using an Alignment-based Metadata Management Scheme

Joost Heitbrink
Vrije Universiteit Amsterdam
j.m.heitbrink@student.vu.nl

Abstract

With attacks on servers becoming increasingly sophisticated and exploits becoming increasingly harder to discover, software hardening solutions have appeared that attempt to prevent buffer overflow attacks through checking memory allocation bounds.

These hardening solutions all need some kind of metadata management in order to efficiently and correctly check if memory accesses are legal or not. All of these bounds checking applications have their own version of a metadata management scheme to efficiently store and retrieve metadata.

Haller et al. [6] present a new metadata management scheme called METAlloc that is not custom made to one specific bounds checking framework. Instead, it is intended to be an efficient metadata management framework that efficiently takes care of metadata storage and retrieval, allowing the authors of software hardening suites to devote more time on creating safer applications and checking methods instead of having to reinvent the weel each time.

In this paper, we attempt to implement an existing bounds checking solution, presented by Akritidis et al. [3] using METAlloc as its metadata management scheme instead of the one presented by the authors, and measure if METAlloc improves the overhead reported by Baggy Baggy bounds checking.

We find that this is not the case, with our solution having an overhead of [overhead]. We determine this degraded perfomance to be a result of conflicting design philosophies and thightly coupled designs making Baggy Bounds unsuitable to a generic metadata manager such as METAlloc.

We suggest to improve METAlloc by implementing more than one specific allocator to allow for more flexibility, and caution against implementing a replacement for metadata management schemes without ascertaining that its design goals and assumptions do not conflict with the replacement solution.

1 Introduction

With the increasing networking of systems all over the world, attacks on remote servers have become not only more prevalent, but also show an increasing level of sophistication in both execution and exploits used.

With many servers employing in-memory databases such as Redis [2] and Memcached [1], increasing amounts of, often sensitive, data is stored in-memory. The heap thereby becomes a prime target for hackers of all shapes and sizes, since sensitive data is often worth a pretty penny.

Most software deployed on high-performance servers are written in low-level languages such as C or C++, where their immense power and unrestricted access to the heap greatly improves performance. However, one cannot be certain without additional tooling that memory accesses are actually legal within the program, and are not controlled by an attacker.

If bugs or oversights exist that enable attacker controlled access to the heap, they can have tremendous impact if they are not discovered, allowing attack-

ers to either (or in the worst case, both) read from or write to arbitrary locations in memory at their leisure. An example of one such bug which had a severe impact on a significant portion of the internet as a whole was the OpenSSL HeartBleed vulnerability, discovered by [4] Durumeric et al.

In short, the bug allowed an attacker to read arbitrary portions of the affected server's memory contents, potentially containing passwords, banking records, patient records or other equally sensitive data.

While stringent software development practices might catch the most egregious bugs in this category, only one bug has to slip through the cracks to allow an attacker to wreak untold havoc. Clearly, just trying to catch bugs during development is insufficient. This is where software security hardening solutions come in to play. These hardening solutions come in many shapes and sizes, but all try to preserve the integrity of the heap in some form. We will examine these software hardening solutions at a high level, and present our on version of bounds checking afterwards.

2 Background

In order to explain our design, we first need to elaborate on the principles of bounds checking, Baggy bounds checking in particular and METAlloc.

2.1 Bounds Checking

Bounds checking can be implemented in many different ways, [cite examples here or in related work?] but they all share the same principle. When an object is allocated, either on the stack or the heap, the memory allocator will attempt to reserve the requested amount of space, and return a pointer to the beginning of the requested part of memory. This returned pointer is what is commonly called the "base pointer" or "base address".

The base pointer plus the size is the upper limit of the allocation, any access beyond that point is prohibited, but not enforced except for unallocated memory. The region of memory between the base

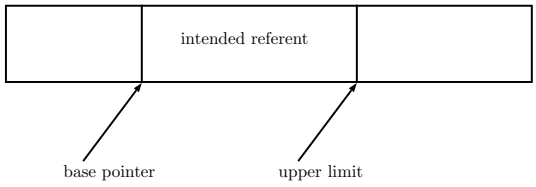

Figure 1: Boundaries of an allocated object, called the Intended Referent.

pointer and its upper limit is what Jones and Kelly [7] call the "intended referent". The pointer that is returned by the memory allocator should only point to anywhere between the boundaries of the intended referent, and this is what bound checking attempts to enforce.

To check whether a pointer is pointing at its intended referent, we therefore need to know at least two things:

1. The base pointer of the object.
2. The upper limit of the object.

Note that this information need not all be stored. It is possible to deduce some information from other metadata, given the right assumptions. It depends on the exact implementation of a bounds checking program what kind of metadata is stored and what type of information is deduced from the metadata or not.

After looking up and (if needed) transforming the metadata into the base pointer and the upper limit, checking if a given pointer points inside or outside the object is trivial.

However, pointer arithmetic can let a pointer, which originally pointed to object A, point to object B, which is not its intended referent. Thus, the bounds checking application needs to distinguish between a pointer which points to its intended referent, and a pointer which was made to point to that object.

The general solution to this issue involves either marking or otherwise tracking pointers through a data structure who no longer point to their intended referent. The main point of the solution is to make sure that all pointers are unique in some way, since they are effectively used as keys that map to metadata.

Failing to preserve the uiniqueness of the keys may result in pointers being falsely judged to be legal while they actually point far beyond their intended referent.

2.1.1 Baggy Bounds checking

Baggy Bounds checking is presented by Akriditis et al. [3] as a fast, efficient bounds checking solution. By tailoring the address space layout to utilize the behavior of a buddy allocator to full effect, they present a bounds checking model that only needs one byte of metadata to compute the base pointer and upper limit.

This is achieved through rounding up all allocations to the nearest power of two and aligning all allocations to powers of two. This allows for fast and predictable retrieval of base pointers when the size is known, at the cost of increased memory overhead due to the mandatatory rounding of allocations to nearest powers of two. We will attempt to implement this form of bounds checking using a new metadata management scheme, described below.

2.2 METAlloc

Haller et al. [6] present a new metadata management scheme that is optimised for fast metadata storage and retrieval, with additional options present for variable or fixed metadata sizes, allowing for either increased performance or flexibility.

METAlloc achieves a high performance on metadata storage and lookup by building up a "meta-page-table" which serves as a translation unit from pointers to a metadata pointer, which points to a table entry that contains the actual metadata.

The primary performance gain is that, since the metadata pointers and metadata are stored separately, the data locality for the stored object is very high, allowing for fast access with simple offset computation, which minimizes memory access that generally slows down traditional metadata management schemes using data structures such as splay trees or red-black trees[cite relevant papers].

Our research question will thus be: *Can we replace the metadata allocator that Baggy Bounds Checking uses with METAlloc and improve the reported overhead of Baggy Bounds Checking?*

3 Design

3.1 Overview

The design of Baggy Bounds checking using METAlloc can be roughly divided into 5 parts:

1. The code analysis component. This component performs static analysis on the source code and implements high-level optimizations. This stage also replaces certain operations and instructions with instrumented versions of them.

2. METAlloc itself, responsible for the actual metadata allocation, management and retrieval.

3. The instrumentation library. This library contains all functions and instruction necessary to replace certain operations, mainly those pertaining pointer manipulation and memory allocation.

4. The crash library. This library contains functions relating to the process of handling any unrecoverable out-of-bounds violation. This component is modular, meaning the user can decide for themselves if a violation terminates the program or logs an error of some sort, and anything in between.

5. Allocation hooks. These hooks execute whenever a function of the malloc family is called, and ensure that metadata is set and properly updated everytime these functions are called.

When a program is instrumented through our software hardening suite, The code analysis component will analyze the source code and identify memory allocation calls, pointer arithmetic and pointer comparisons. These operations are instrumented to facilitate bounds checking.

Stack allocations are instrumented in a similar way to the memory allocator hooks. We round off any allocation, both on the heap and the stack, to the nearest power of two, allowing any size to be saved

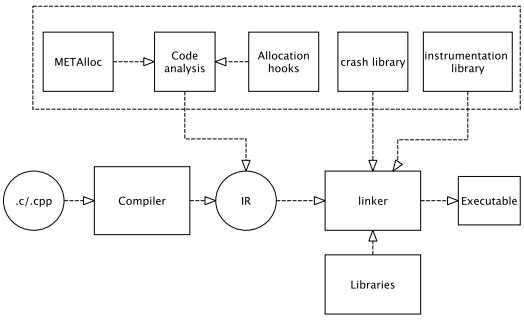

Figure 2: The instrumentation process, with our contribution in the dashed box.

as the binary logarithm, reducing the metadata size to one byte.

All pointer arithmetic is intercepted and replaced with a function that first checks if the computation would mean the pointer would point past its intended referent. If it does, we do not immediately invoke the crash library, since the C standard allows pointer to be out of bounds legally, as long as they are not dereferenced.

Instead, we mark the offending pointer in such a way that if it is dereferenced while out of bounds, the program will invoke the crash library. This marking also means the instrumentation watches for pointers which were out of bounds, but come back into bounds. When a previously marked pointer comes back into the bounds of the intended referent, the instrumentation will remove the marking from the pointer, ensuring it can be safely dereferenced again.

Tracking out of bounds pointers cannot go on indefinitely however, since the marking capabilities of the library can only guarantee the uniqueness of a pointer up to a certain point, meaning that if an out of bound pointer is incremented or decremented past this limit, the crash library is invoked.

If a pointer that is out of bounds does get dereferenced while within the marking boundary, but outside the bounds of the intended referent, the program will also invoke the crash library, which will take action that defaults to aborting the application, but this behaviour can be modified to any action the application designer deems optimal. The action taken can take any form, from simply logging the error to aborting the program entirely.

This enables systems that cannot be turned off to notify of a bounds violation, and let personnel controlling the system decide the best course of action, instead of making dangerous assumptions itself.

3.2 Bounds Checking Design

We will now examine the design of each component in more detail.

3.2.1 Code Analysis Module and instrumentation library

The code analysis module will analyze the source code to best optimize metadata placement and management. The module performs some checks to make sure no redundant metadata is set, and if a pointer can be proven to always be in bounds it will refrain from instrumenting it. After a first round of optimizations, we are left with a set of pointers who cannot be guaranteed to be in bounds at any point in time. All pointer arithmetic involving these pointers is instrumented and if at any point the pointer goes out of bounds, we mark it to prevent it from being dereferenced, but allow it to still be used in pointer arithmetic.

Because we allow marked pointers to be used as long as they are not dereferenced, we also need to instrument the comparison instructions to make sure that pointer comparison compares the actual pointer instead of the marked version. The instrumentation simply performs the comparison, but first unmarks any affected pointers, before marking the pointers again.

3.2.2 Allocation hooks and crash library

The allocation hooks are built in directly to the memory allocator, allowing any form of memory allocation call to be instrumented directly, and also allowing for metadata management when functions like realloc are called. We also round up any allocation size to the nearest power of two, and zero the remaining padding of the allocation bounds.

The free function is also instrumented to free any metadata and perform any other necessary actions. Again, these hooks, apart from their default behavior, are fully customizable to the needs of the application being instrumented.

The crash library is similarly customizable, with bounds violations defaulting to the process calling the `exit(2)` library call, but it can instead be extended to log violations to a file, use notification software to broadcast an alarm message system wide or through networks, allowing the application programmer to best pick and choose the behavior for bounds violations. The crash library's extensibility is only limited by the fact that it must be programmed in pure C, but apart from that, there is no restriction imposed on what should happen at the moment of a bounds violation.

3.3 Implementation

Our implementation is made using the LLVM compiler framework, since METAlloc requires it for metadata management. We implement the code analysis module as a single LLVM pass that will utilize the LLVM API to instrument our code. This does mean that we have to adjust to some LLVM specific paradigms, the most important being the lack of direct pointer arithmetic. Instead of pointer arithmetic, LLVM uses a unique instruction called the GetElementPtr instruction to perform address computation and an approximation of pointer arithmetic [8]. It is this instruction that we replace with our own version when instrumenting pointer arithmetic. An important note is that although METAlloc attempts to get out of your way as much as possible, it cannot be fully separated from other components in the system, the main component being the memory allocator, which is the tcmalloc [5] allocator.

While this might seem like a trivial detail, we need to be mindful of the fact that tcmalloc has a specific approach on memory managment, meaning that any application making assumptions about the memory allocator should be aware of this.

For the pointer marking mechanism, we restrict ourselves to 64 bit architectures. Doing this allows us to use the last 12 bits that are unused on 64 bit pointers to store information. In additions, we need to restrict programs to the lower half of the address space. On most 64 bit architectures, this is already the case.

When a pointer is increast past its intended referent's upper limit, we set the highest bit of the pointer. Similarly, if the pointer gets decremented past its intended referent's base address, we set the second highest bit. If the pointer comes back into the intended referent, we clear any set bits.

Our instrumentation responsible for handling the pointer arithmetic and comparison ensures marked pointers are first returned to their unmarked form before any checks or comparisons are performed, and if necessary, revert the pointers to their marked state again after the operation.

4 Results and Evaluation

We tested our implementation using the SPEC2006 benchmark suite, running 4 uninstrumented instances and 4 instrumented instances. The results are shown in the table below.

We have measured an overhead of [overhead] times. Interesting spikes are [spikes].

4.1 Evaluation

We suspect the major difference in performance is due to the fact that Baggy Bounds makes a significant amount of assumptions about the layout of the address space of the program. So much, in fact, that we believe that METAlloc's allocator and design directly oppose the assumptions made by Baggy Bounds. We see this in [bench] ...

5 Related Work

Apart from live object tracking that we have just shown, are many other techniques, ranging from tracking *all* memory allocations, to address space obfuscation and an unnameable number in between.

For example [9] proposes to instrument memory with so called `shadow memory` that map safely accessible sections to this shadow memory, allowing in-

strumentation to check if a memory access is legal. It then assigns guard zones between the valid memory sections to prevent pointers from going out of bounds. These guard zones contain a specific "poison value", allowing the instrumentation to watch out for those zones and take action when they encounter these pointer values.

6 Future Work

Since our project did not perform as well as expected, the biggest takeaway message should be that even though METAlloc has the potential to be really efficient with metadata management, it still brings with it a few limitations.

Application writers should be mindful of their own assumptions or optimiziations regarding the address space layout and underlying compiler infrastructure. Even though we eventually want to arrive on a totally independent metadata management solution, the current compiler landscape prohibits certain easy extensions or modifications.

As witnessed with our implementation of Baggy Bounds checking using METAlloc, the assumptions Baggy Bounds makes essentially mean that the metadata allocator and the resulting checking library are so tightly coupled to each other that they can, on the implementation side, no longer be seen as two separate components.

This might not be obvious at first, since it is also easier to explain a system consisting of many small modules with a clear role then a system consisting of a few monolithic components. However, on the implementation side, components might be merged for performance reasons or because their separation only made sense for explaining their concept better. In doing so, METAlloc might be unfit for this purpose because of the lessened modularity in design of the application.

Care should be taken, as we have proven that although METAlloc makes great strides in metadata management performance, it is not a "drop-in" replacement for all projects using some form of metadata.

Future research could focus on implementing different allocators for METAlloc, allowing a greater realm of flexibility for applications needing a specific allocator for either address space layout or compatibility reasons.

If METAlloc can achieve a greater level of modularity itself, application authors need not be as considerate to METAlloc's assumptions and ecosystem needs as it needs to be now, making METAlloc a true drop-in replacement for custom made metadata allocators.

7 Conclusion

In this paper, we implemented a form of Baggy Bounds checking using the metadata management scheme METAlloc to see if it could improve the reported overhead of Baggy Bounds checking. This was not the case, and the reported overhead was [times] higher than the reported overhead from the Baggy Bounds reference implementation.

We deterimined this performance overhead was due to the design of Baggy Bounds checking having such a tight coupling with its metadata management scheme that any deviation from it means incurring significant overhead costs. With METAlloc not sharing the same design principles, the chief component being the lack of a buddy allocator, we found that METAlloc was actively "fighting" the way Baggy Bounds was designed.

We therefore conclude that METAlloc cannot serve as a generic replacement for metadata management schemes without first ascertaining how tightly coupled the metadata management schemes are with their runtime checking schemes, and what kind of assumptions the implementation makes over the metadata management scheme.

We recommend METAlloc to, if possible, implement support for different memory allocators, allowing METAlloc to be tailored to the specific needs of each application as necessary.

References

[1] Memcached: distributed memory object caching system. `https://memcached.org/`. Accessed: 2016-07-12.

[2] Redis: in-memory data structure store. `http://redis.io/`. Accessed: 2016-07-12.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[4] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[5] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2009.

[6] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. Metalloc: efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security*, page 5. ACM, 2016.

[7] R. W. Jones and P. H. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 13–26. Linköping University Electronic Press, 1997.

[8] C. A. Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.