Baggy bounds with LLVM

Anton Anastasov Chirantan Ekbote Travis Hance

6.858 Project Final Report

1 Introduction

Buffer overflows are a well-known security problem; a simple buffer-overflow bug can often lead to a security vulnerability letting an adversary execute arbitrary code. There are many approaches attempting to mitigate this danger. One approach is baggy bounds checking, which we discussed in class. Baggy bounds checking was first described in [1], whose authors implemented it using the Phoenix compiler framework. Our project was to implement baggy bounds checking using the LLVM compiler infrastructure. LLVM provides an *intermediate representation* (IR) that can be targeted as a back-end for any high-level language. LLVM also provides a framework for developers to write *passes* (typically optimizations although that is not the primary use for us) which transform LLVM IR. Finally, the LLVM code can be compiled down to machine code for a specified architecture. We use the LLVM pass framework to write a transformation which inserts baggy bounds instrumentation into code.

For this project we focused on 32-bit architecture and used the 6.858 Linux VM image.

2 Implementation

2.1 LLVM passes

We instrument a program with baggy bounds checking by applying several transformation passes to its IR:

- baggy-globals - Insert code to initialize global state variables like the `baggy_size_table`
- baggy-save-locals - Ensure that objects allocated on the stack are aligned to `SLOT_SIZE` boundaries and insert code to update their entries in the `baggy_size_table`
- baggy-rewriter - Rewrite calls to unsafe libc functions (like `strcat` and `strcpy`) to use baggy bounds aware wrappers for those functions instead
- baggy-pointers - Instrument all pointer arithmetic to use baggy bounds checking and insert a branch to the slow path function if the bounds check fails. Additionally, perform bounds checking before LLVM intrinsics like `llvm.memset`

Listing 1: A simple implementation of memset

```
void *memset(void *ptr, int value, size_t num)
{
  size_t i;
  unsigned char c = (unsigned char)value;
  unsigned char *buf = (unsigned char *)ptr;

  for (i = 0; i < num; i++) {
    buf[i] = c;
  }

  return ptr;
}
```

Listing 2: LLVM IR generated for memset

```
define i8* @memset(i8* %ptr, i32 %value, i32 %num) {
entry:
  %ptr.addr = alloca i8*, align 4
  %value.addr = alloca i32, align 4
  %num.addr = alloca i32, align 4
  %i = alloca i32, align 4
  %c = alloca i8, align 1
  %buf = alloca i8*, align 4
  store i8* %ptr, i8** %ptr.addr, align 4
  store i32 %value, i32* %value.addr, align 4
  store i32 %num, i32* %num.addr, align 4
  %0 = load i32* %value.addr, align 4
  %conv = trunc i32 %0 to i8
  store i8 %conv, i8* %c, align 1
  %1 = load i8** %ptr.addr, align 4
  store i8* %1, i8** %buf, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %2 = load i32* %i, align 4
  %3 = load i32* %num.addr, align 4
  %cmp = icmp ult i32 %2, %3
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %4 = load i8* %c, align 1
  %5 = load i32* %i, align 4
  %6 = load i8** %buf, align 4
  %arrayidx = getelementptr inbounds i8* %6, i32 %5
  store i8 %4, i8* %arrayidx, align 1
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %7 = load i32* %i, align 4
  %inc = add i32 %7, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %8 = load i8** %ptr.addr, align 4
  ret i8* %8
}
```

To understand how our transformation passes work, consider Listing 1, which shows a very simple implementation of the `memset` function. Listing 2 shows the LLVM IR that is generated for the `memset` function. This IR is then fed to our transformation passes which insert the code

necessary to perform baggy bounds checking. A key feature of the LLVM IR is the `getelementptr` instruction, which can be seen on line 30 of Listing 2. All pointer arithmetic and array accesses use this instruction to get a pointer the requested location of memory. This useful feature made it much easier for us to find and properly instrument all the pointer arithmetic in a program. Figure 1 shows the control flow graph (CFG) of the `memset` function before and after performing baggy bounds instrumentation.

In addition to the transformation passes, we also provide a static library that implements various functionality that is necessary for baggy bounds checking:

- Initialization code to perform tasks like moving the stack to the bottom half of memory and updating all stack references
- Start files that call our initialization code very early in the program's execution
- A userspace page fault handler that allocates pages for the size table as needed
- The slow path function responsible for handling pointers that may be out of bounds
- Baggy bounds aware wrappers to unsafe libc functions
- A baggy bounds aware memory allocator

2.2 Memory allocator

By design, baggy bounds checking relies on a buddy allocator that rounds up dynamic allocations to the nearest next power of two. Storing the size of a memory allocation in `baggy_size_table` requires 5 bits on a 32-bit architecture, because it's sufficient to store the binary logarithm of the allocation size. Similarly to [1], we chose a `SLOT_SIZE` of 16 bytes, and have one byte in `baggy_size_table` per slot. Our buddy allocator follows the one in the paper closely, but has one distinct feature. We wanted to implement fast and resource-efficient dynamic memory allocator. Achieving both properties requires coalescing free buddy blocks of size 2^p into a single free block of size 2^{p+1}, which guarantees good external memory fragmentation.

However, this design change necessitates keeping track of one additional bit per memory allocation that denotes if the block is free or not. Since storing the binary logarithm of the allocation size requires only 5 bits, we use one of the three unused bits in each entry of `baggy_size_table` to signify if the corresponding slot is free or not without modifying the memory overhead of baggy bounds checking.

2.3 Build procedure

We tested our pass with some test programs in C++ with the clang front-end to compile the source code to LLVM. In principle, we could have used any language with an LLVM front-end. See Figure 2. For a C++ file `program.cpp`, we use clang to compile to the intermediate `program.ll`. We then apply our own pass (e.g., using the LLVM tool `opt`) to construct another LLVM file `program.bg.ll`. Here, we use the `".bg"` substring to indicate that it has the special baggy bounds arithmetic. Finally, we can use an LLVM back-end for the target architecture to construct a standard object file `program.bg.o`. Then we can link that with our library `baggylib.a` to get an ELF executable `program.bg.out`. The final linking step provides our special memory allocator.

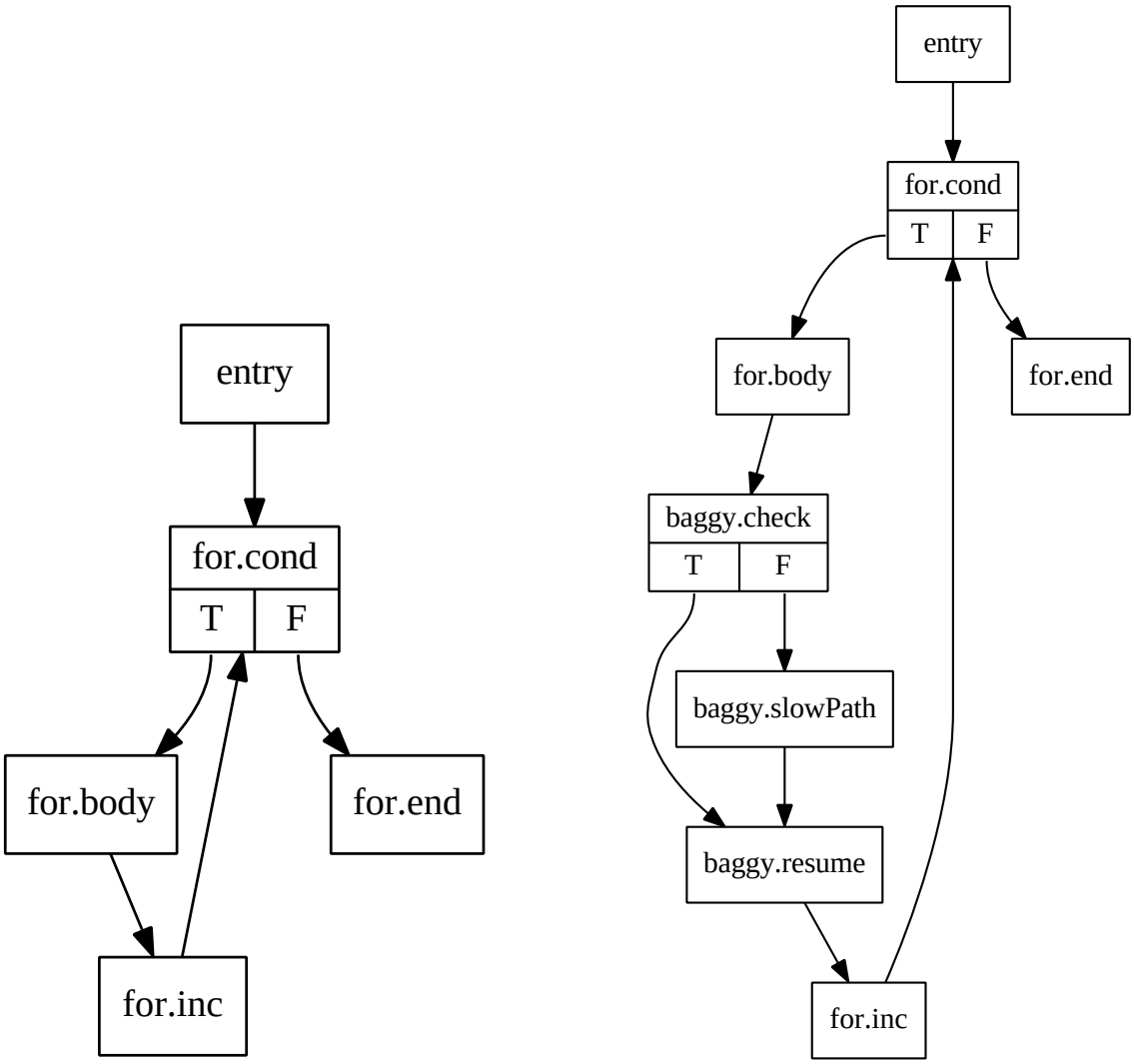

Figure 1: Control flow graph for the `memset` function before (left) and after (right) baggy bounds instrumentation

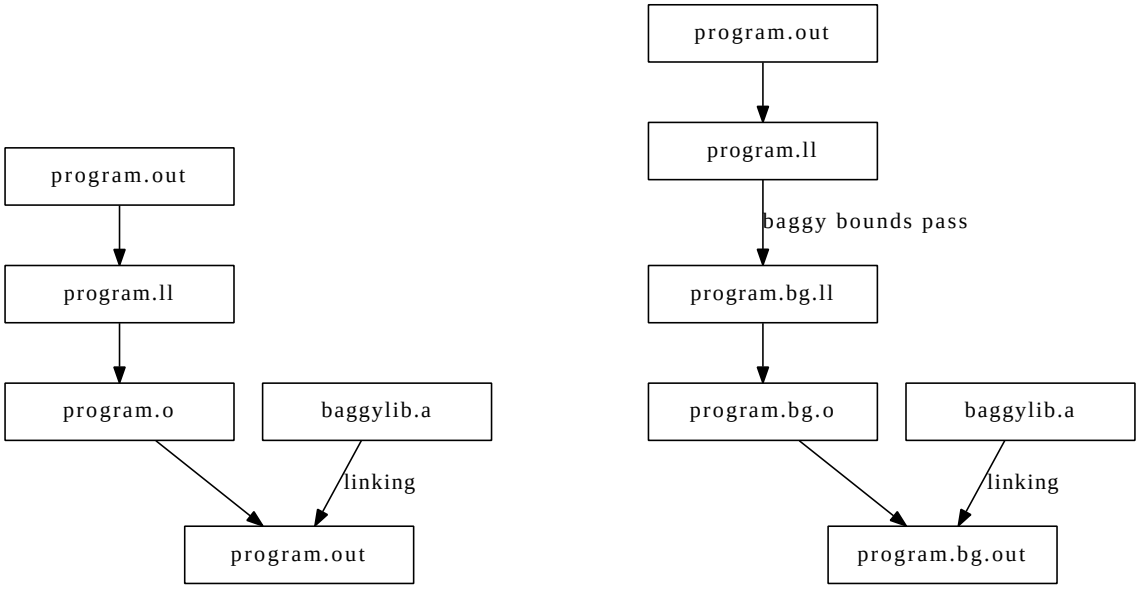

Figure 2: The left shows a "standard" build which links against the `baggylib.a` archive to use our memory allocator but does not use the special baggy bounds arithmetic. The right shows the build process with our baggy bounds pass in the middle.

3 Application: Zoobar webserver

To verify that our instrumented code properly defends against attacks, we compiled the unmodified zoobar web server from lab 1 with baggy bounds checking and then attempted to use the exploits we developed in lab 1 on the server. The following table summarizes our findings:

Exploit	Error Type	Allowed code execution
2a	SIGSEGV	No
2b	LIBC	No
3	LIBC	No
4a	SIGSEGV	No
4b	LIBC	No

None of the exploits were able to execute arbitrary code on the web server. Instead one of three possible bounds checking errors was returned by the server in every exploit. The SIGSEGV error type indicates that a pointer to out-of-bounds memory was de-referenced while the LIBC error type indicates that an out-of-bounds access was detected in the wrapper for a libc function. The third error type, SLOWPATH, which was not triggered by our exploits indicates that a computed pointer was more than `SLOT_SIZE / 2` out-of-bounds.

4 Performance and benchmarks

We tested the performance of our implementation on a set of memory intensive benchmark programs. In particular, we implemented matrix multiplication of square matrices, three variations of

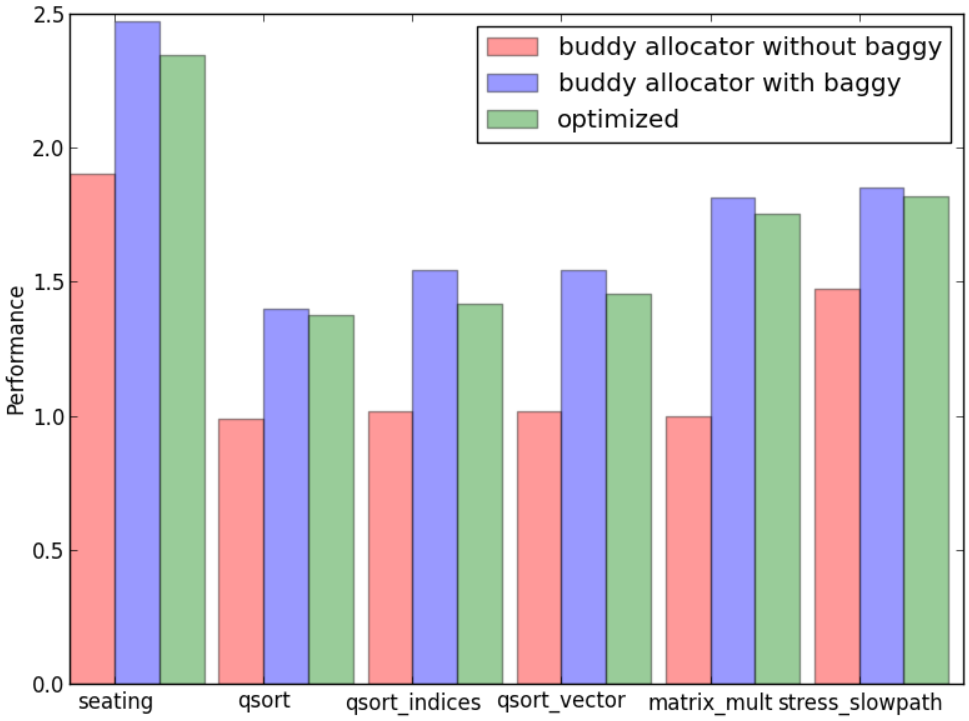

Figure 3: Execution time for a set of benchmark programs using the buddy allocator and baggy bounds, normalized by the execution time using the system allocator without instrumentation.

quick sort with pointer implementation, indices implementation and STL's vector-based implementation, a program that makes updates and queries to a dynamic tree, and a program that results in many calls to `baggy.slowPath`. Our findings can be seen in Figure 3. We observe a performance degrade from 50% up to 2.5x, which is comparable to [1].

4.1 Optimizations

Since LLVM provides a framework and libraries for writing compiler optimizations, there is a very large space of possibilities for static analyses and optimizations. For example, if we can *prove* that a pointer will only be accessed safely, then we do not have to instrument its arithmetic operations with the baggy bounds arithmetic.

As proof-of-concept, we implemented one very simple optimization. Note that in some cases, there is no need to save the pointer sizes in the `baggy_size_table`. For example, we do not have to load the size of an allocation from the baggy size table more than once for the same pointer (or for pointers derived from that pointer). Standard common subexpression elimination (CSE) can eliminated redundant memory loads in some cases, but we can do even better with the additional knowledge we have about the semantics of the `baggy_size_table`.

Furthermore, if a function allocates an array on the stack but never passes the a pointer to it (or any pointer derived from it) to another function, it does not have to save the size in the `baggy_size_table` at all. Some careful static analysis is required to determine when this is safe,

but it is mostly a matter of following the uses of the the variable and looking for "dangerous" ones (e.g., stores and function calls).

We implemented more sophisticated version of the `baggy-pointers` and `baggy-save-locals` passes which use these ideas to avoid unnecessary loads and stores. The improvements are small but noticeable, as can be seen in Figure 3.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. *Proceedings of the 18th conference on USENIX security symposium*, pages 51-66, 2009.