Security and Performance Analysis of Custom Memory Allocators

by

Tiffany Tang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author .....
Department of Electrical Engineering and Computer Science
June 10, 2019

Certified by .....
Howard E. Shrobe
Principal Research Scientist and Associate Director of Security, CSAIL
Thesis Supervisor

Certified by .....
Hamed Okhravi
Senior Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Albert R. Meyer,
Chairman, Masters of Engineering Thesis Committee

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Security and Performance Analysis of Custom Memory Allocators

by

Tiffany Tang

Submitted to the Department of Electrical Engineering and Computer Science
on June 10, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Computer programmers use custom memory allocators as an alternative to built-in or general-purpose memory allocators with the intent to improve performance and minimize human error. However, it is difficult to achieve both memory safety and performance gains on custom memory allocators. In this thesis, we study the relationship between memory safety and custom allocators. We analyze three popular servers, Apache, Nginx, and Appweb, and show that while the performance benefits might exist in the unprotected version of the server, as soon as partial or full memory safety is enforced, the picture becomes much more complex. Based on the target, using a custom memory allocator might be faster, about the same, or slower than the system memory allocator. Another caveat is that custom memory allocation can only be protected partially (at the allocation granularity) without manual modification. In addition, custom memory allocators may also introduce additional vulnerabilities to an application (e.g., OpenSSL Heartbleed). We thus conclude that using custom memory allocators is very nuanced, and that the challenges they pose may outweigh the small performance gains in the unprotected mode in many cases. Our findings suggest that developers must carefully consider the trade-offs and caveats of using a custom memory allocator before deploying it in their project.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist and Associate Director of Security, CSAIL

Thesis Supervisor: Hamed Okhravi
Title: Senior Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I am forever in-debt to my advisor, Dr. Hamed Okhravi, for his guidance and mentorship throughout the years. He has provided me with invaluable advice on how to think as both a person and a computer scientist in difficult projects, which has served as a basis for how I manage many situations in work and in life: by breaking hard problems into smaller, more manageable ones and differentiating between urgent and important. His patience in reading through my convoluted emails and drafts is something I am very grateful for. In addition, I'd like to thank Howard Shrobe and Stelios Sidiroglou-Douskos, who were equally as patient with me throughout my time at MIT.

Next, I would like to thank my friends. Without them and their continuous encouragement and countless hours of proofreading for grammatical errors and oddly phrased sentences, I would surely not have made it this far: Michelle Wong, Alex Tran, and Ulziibayar Otgonbaatar.

Finally, I would like to express gratitude to my parents, who have provided me with unconditional trust and love throughout the years and never failed to provide support.

Contents

List of Figures

List of Code Listings

Chapter 1

Introduction

By definition, memory corruption occurs when memory is modified illegally. The memory contract between the program and operating system is violated, and as a result, malicious actions may occur. We use the term "may" because an instance of memory corruption does not necessarily lead to a security exploit.

First, we must consider whether the corrupted memory can affect the programs intended behavior. There are instances in which the corrupted memory is either contained within a section of the program that does not impact the program in a significant way, such as the cases in which the overwritten memory is no longer used by the program or the overwritten memory gets written over by the correct value before the program uses the memory. For a program to run correctly, only a subset of values in the data execution path must be correct. These bits are referred to as Architecturally Correct Execution (ACE). The rest are un-Ace. In a Spec2K benchmark test, it was shown that 46% of bits in a program on average are ACE [1]. Therefore, it is possible for a memory corruption to be benign.

On the other hand, memory corruption could also be malicious and result in a successful exploit. The majority of memory corruption bugs are caused by developer error and the use of a memory unsafe programming language. Consider code example 1.1, which is written in C. This example was taken from *Smashing the Stack for Fun and Profit*[2], which demonstrates how to spawn a shell from an unsuspecting user's computer by taking advantage of a simple buffer overflow.

```
1 void copyArray(char bad[100])
2 {
3     char buf[10];
4     strcpy(buf, bad);
5 }
```

Code Listing 1.1: Buffer Overflow Example

`copyArray` is a function that takes in one parameter called `bad`, which is an array of type `char` and of size 100. `bad` acts as a string. `copyArray` copies the data located at the address of `bad` into the address at `buf`. `buf` is a `char` array of size 10 and is significantly smaller in size than `bad`. The function, `strcpy`, does not check to see if `buf` has enough space to copy all of the elements in `bad`. With no space to write all the characters from `bad` into the `buf` array, the program writes to the memory after `buf` in the stack.

At this point, unexpected behavior may occur. The memory contract between the operating system and the program when `buf` was allocated was that `buf` would only consist of 10 `char`s. The operating system only gave `buf` enough space for 10 characters, so the memory that gets overwritten by the other 90 characters after the `buf` array does not belong to `buf`. `buf`'s data could even be overwritten at a later point, as `buf` has no control over what happens to memory that was not given to it. If the memory overwritten is ACE and is later used by the program, then the attacker may be able to trick the program into executing malicious code through something like return-oriented programming (which we will expand upon at a later section) [17]. This allows an attacker to remotely hijack the program execution and take malicious actions afterwards.

The implied solution to preventing memory corruption is to use memory safe languages like Rust, Go, or Java. If we had used Java to write `copyArray` instead, then we would have received a out-of-bounds error when attempting to copy the contents of `bad` over to `buf`. However, unsafe low-level programming languages such as C and C++ remain popular and are still the preferred programming languages in many development environments today.

Memory unsafe languages require the developer to manage memory themselves.

By doing so, the developer gains significant freedom and in many cases, can increase performance by optimizing memory management for their specific application. Programming languages that perform automatic memory management such as Java are often frowned upon when discussing performance and optimizations. As such, the ability to manage memory becomes a double-edged sword.

Even though numerous safeguards have been developed over the years to ensure memory safety, most of these defenses are often discarded for performance reasons or lack of compatibility with the existing code. Most programmers tend to avoid anything with a high performance overhead.

According to Milo Martin, one of the authors of Softbounds [3], the endowment effect [30] affects the response of programmers when it comes to making decisions between having more performance or security. The endowment effect is an attachment to the object that the person already owns and the persons reluctance to part with it. Thus, there is a status quo bias. The question generally asked amongst programmers when deciding to implement a certain security defense is how much performance must I sacrifice for this defense? If the number is say, 50%, it is highly unlikely that the defense will be used. However, consider if the question was rephrased to how much security must I sacrifice for this performance gain? If the number is also, say 50%, the programmer might think differently before trading security for performance.

Unfortunately, it has been an ongoing trend to trade security for performance. More emphasis is placed on improving the performance of a program rather than the security. Due to this, a number of security exploits have arised such as the OpenSSL Heartbleed bug [5] and the BadUSB bug [6]. The number of exploitable features only grow, especially with the development of the internet of things, which has been shown to neglect security because of consumer demand for better performance and usability [4].

In this thesis, we demonstrate this trend of choosing performance over security with custom memory allocators. Custom memory allocators achieve performance improvements, but are poorly designed for security. One of the main goals of custom memory allocators is to achieve better performance than the system memory

allocator. However, recent research [9] shows that the system memory allocator outperformed most known custom memory allocators. Only a specific type of custom memory allocator, the pool/region allocator, seemed to achieve any sort of performance gain. Even then, we show that the challenges created by custom memory allocators may out-weight the small performance gain offered. Thus, developers may want to reconsider the use of custom memory allocators.

To better understand custom memory allocators, we study three major servers: Apache, Nginx, and Appweb. Each three of these major servers use a custom implementation of a pool allocator. First we evaluate their unprotected performance with and without the custom allocator by manually removing the custom allocator and replacing it with the system memory allocator. This required code changes to all three servers. Then, we enable partial protection for the custom memory allocator, which only enforces spatial safety at the granularity of allocations, and evaluate the performance of the three servers by measuring the latency for a number of HTTP requests. Finally, we manually modify these servers to enforce full spatial safety by adding Intel MPX functions to log all custom allocators. Like before, we evaluate with and without a custom allocator.

The studies resulted in mixed results. In some cases, performance gains we expect to see exist. However, in other situations, the performance is actually worse. For example, when Apache has full protection, the system allocator performs worse than the custom allocator. Given that the level of granularity is higher in the case of the custom allocator, one would expect the custom memory allocator to perform slower. When we consider the other problems caused by using custom allocators, such as vulnerabilities and granularity limitations, our findings show that developers must carefully consider the implications before using a custom allocator.

The rest of the thesis is as follows. In chapter 2, we review existing memory corruption literature and a number of available defenses. In chapter 3, we discuss Intel MPX, a software and hardware defense that enforces spatial memory safety. In chapter 4, we examine the general memory allocator and commonly used types of custom memory allocators. Then we list the advantages and disadvantages of each

custom memory allocator explored. In chapter 5, we evaluate the security of custom memory allocators. In chapter 6, we analyze custom memory allocator-aware memory safety and its implications. We conclude in chapter 7.

Chapter 2

Background

In this chapter, we examine memory corruption in more detail. Then we review modern defenses, namely enforcement-based and randomization-based defenses.

2.1 Definition

Memory corruption bus can be categorized into two major classes: spatial memory bugs and temporal memory bugs. To better understand these bugs, here we look at how each class works by using a demonstrative example.

A spatial memory bug occurs when an attempt to dereference memory that is outside the bounds of the allocated object. Traditionally speaking, this is the cause of buffer overflows. To better illustrate this, consider a character array of size 255. The developer makes an error while writing his code and mistakenly allows 256 characters to be written into the array. The array only has enough space to store 255 characters. Since the program is forced to write all 256 characters into the array, the last character is written to the memory location right after the array. This provides a good opportunity for an attacker to exploit.

The second case deals with use-after-free bugs, which occurs when the developer uses memory that has been returned to the operating system already. A common example of a use-after-free bug is a dangling pointer. A dangling pointer occurs when memory is allocated for a pointer and then freed, but the pointer is still pointing to

the same memory address. Though the memory is returned to the operating system, it can still be reused if the pointer has a reference to the freed memory address. This may be disastrous if the memory has been reassigned to another pointer, as the developer expects the newer pointer to be the only pointer that has access to the memory address.

In both these examples, the leading cause is developer error due to the complexity of having to manage memory. The goal of achieving security is seemingly at conflict with achieving good performance. New attack vectors arise when the developer is allowed to manage memory. Note that Java is also susceptible to memory corruption bugs and many have been disclosed to the public [58]. Though Java is memory safe, the JRE (Java Runtime Environment) is written in C++ .

2.2 Memory Corruption Attacks

In this section, we present the history of exploit techniques that take advantage of memory corruption.

2.2.1 Code Injection

We start with the traditional buffer overflow exploits [2, 14], a technique that writes more data to a buffer than the buffer can hold and forces the memory contents to spill over to adjacent memory locations. The attacker might then be able to change the control flow of the program. A program consists of an ordering of computer instructions that get executed. When the program is exploited and the control flow changes as a result, the program is executing a path that would not get executed under normal circumstances. To fully understand why overwriting memory allows the attacker to change program control flow, imagine that we have a stack.

The call stack grows from the bottom-up, as indicated by the arrow right of the figure. The stack is used to store metadata about the function being executed by the program. Say the program is currently within a function call (figure 2-1). The object of interest is the return address, which is the instruction address that the program

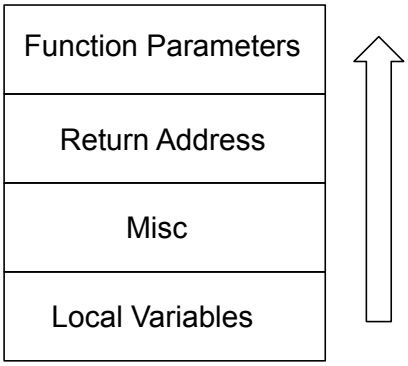

Figure 2-1: Function Call Stack Layout

will return to once the program finishes executing the current function. Generally, the return address points to the succeeding instruction in the previous function that called this function. In order words, the return address points to the next instruction the program will execute after finishing this function. If the attacker can modify the return address, then the attacker can change the program control flow by forcing the program to return to a different location.

```
1  void function(char *str) {
2     char buffer[16];
3     strcpy(buffer,str);
4  }
5  void main() {
6    char large_string[256];
7    int i;
8    for( i = 0; i < 255; i++)
9      large_string[i] = 'A';
10   function(large_string);
11 }
```

Code Listing 2.1: Buffer Overflow Example 2 taken from [2]

The code excerpt in Listing 2.1 contains a bug that will cause a buffer overflow. In this example, the program first initializes `large_string`, which is a `char` array of size 256 and sets all elements in the array to be the character A. Once that is completed, `large_string` is copied over to `buffer`, an array of size 16. `buffer` is not able to hold all 256 characters, but the function `strcpy` (as opposed to the

safer version, `strncpy`) does not perform any checks regarding the size difference and will allow the action to occur. As the program copies all 256 characters from `large_string` to `buffer`, some of the As will spill over to adjacent memory.

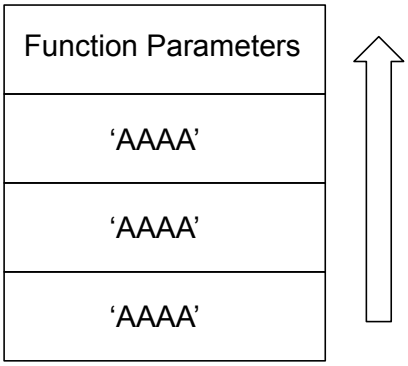

Figure 2-2: Function Call Stack Layout After An Overflow

In the aftermath of the overflow is shown in the call stack. Since the return address is higher up in the call stack than the local variables, the return address has been overwritten with the As. By doing a simple mathematical calculation (or plain guess) based on where the local variable `buffer` is relative to the function's return address, the attacker can find out exactly where in `large_string` to put the relevant information to change the return address to a set of instructions that the attacker wants to execute.

However, where can the attacker find the exact set of instructions to execute? Do they even exist? The attacker can insert program instructions onto the stack using a buffer overflow, calculate the address at which these inserted program instructions are located, and then set the return address in the function call stack to be at that address. At the time of simple code injection attacks [2], the stack was executable (gcc today marks the stack as unexecutable except for certain situations), so this exploit could be pulled off easily.

2.2.2 return-to-libc

The return-to-libc [14, 16] exploit was developed afterwards. It is the first example of a code reuse attack, in which the existing code in the program is used to change

control flow as opposed to shellcode injected by the attacker. As the name implies, the attacker takes advantage of the presence of the `libc` library in a return-to-libc attack. `libc` is the C standard library, which all C programs are guaranteed to have. The `libc` functions and source code are readily available on the internet for anyone to access. There are thousands of lines of code in the `libc` library, which makes it a good tool for exploits. In this example, we assume that the attacker has found a way to overflow the stack through a buffer overflow and has the ability to modify the return address of the current function to an address of his choosing.

```
1 (gdb) p printf
```

Code Listing 2.2: Finding Address of libc function

If we do not (or cannot) use the stack to execute instructions, we must use existing instructions in the program, but how do we guarantee that an instruction we need will exist? We know that all C programs contain the `libc` library, which is composed of many functions. In most cases, having access to these functions is sufficient in carrying out an exploit. All that is left is to find out where the `libc` functions we need are in memory, which is a simple task. The attacker can find the address to a function in `libc` by running the program in a debugger like `gdb` and printing out the address. Alternatively, the attacker can use object dump. The location of `libc` functions are relative to each other so once the attackers find the starting address of `libc` in the program, they know all the addresses in libc.

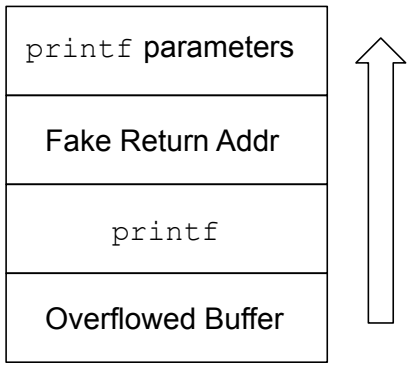

Figure 2-3: Function Call Stack Layout after return-to-libc attack

For example, suppose the attacker wants to execute `printf`, even though the program never calls `printf`. The attacker must first find the address of the `printf` function in `libc`. If he overwrites the return address in the call stack of the current function to point to `printf`, then the program will jump to `printf`. However, that will still not achieve the goal of calling `printf`. As we noticed, all functions have their own call stack. Since the program was never meant to call `printf`, the program does not have a proper call stack for `printf`. The attacker will have to modify the stack to mimic the function call stack of the `printf` function first before "returning" to `printf`.

The attacker needs to set the return address of the current function's call stack to point to the memory address of the `printf` function. In addition, he needs to set a fake return address for the call stack of `printf`, as the program expects all function call stacks to have a return address. The fake return address can be bogus or if the attacker wishes to chain his return-to-libc attack, can be the address of the next function the attacker wants to call. Then the attacker needs to set up the function parameters of the `printf` function, if any exist.

2.2.3 Return-Oriented Programming

Return-Oriented Programming, or ROP, is a technique that was developed around 2005 [17, 18] and has more freedom than return-to-libc attacks. It is fine-grained code reuse and Turning-complete [17]. In ROP, the hacker reuses machine code (as opposed to just existing functions) to carry out his exploit.

In this type of exploit, the attacker builds gadgets, which contain lines of

```
1 pop $eax;
2 pop $ebx;
3 ret;
```

Code Listing 2.3: ROP Gadget Example

existing machine code. ROP is called "return-oriented programming" because in the original exploit, the `RET` return instruction was used to jump around in code. As a

result, gadgets often ended with the `RET` instruction. Similar to how `libc` functions are discovered by the attacker, the attacker can find the address of any instruction using the same techniques. Then, using these lines of instructions, the attacker can make gadgets. Since there are so many lines of instructions available (whether it be in the program code or `libc`, we can guarantee that the gadgets made by the attacker are Turning complete.

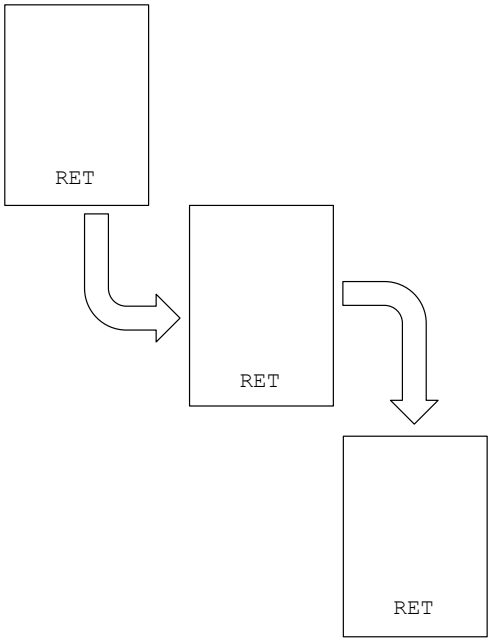

Figure 2-4: ROP Attack

In a ROP exploit (figure 2-4, the attacker jumps to his first gadget. After the gadget has finished running, the gadget "returns" to the next gadget until all gadgets have been executed. Since the first paper, there have been numerous papers published showing that it is not necessary for gadgets to end with a `RET` instruction to carry out a successful attack [31, 32, 33]. For example, a `JMP` instruction could be used in place of a `RET` instruction, as both instructions function similarly enough.

2.3 Enforcement-based Defenses

In this section, we examine enforcement-based defenses. Enforcement-based defenses take a proactive approach in preventing exploits from occurring.

2.3.1 Non-Executable Memory

Non-executable memory was first developed on OpenBSD in 2003 [15] as W^X (write exclusive-or execute). It attempts to solve the problem of code injections during buffer overflows. With non-executable memory, memory can either be writable or executable, but not both. The pages (unit of memory) in memory are marked with an extra bit called the NX bit that determines whether or not the page is writable or executable. Recall in the code injection example, we overflowed the stack with shellcode. Even though the stack's intended purpose was to store information regarding the variables and return address of the function, we were able to execute instructions located on the stack. By marking the stack as non-executable (since it is writable), the shellcode will not be able to run correctly.

Most popular operating systems offer support for the NX bit. It is known as Data-Execution Prevention (abbreviated as DEP) [14, 36] on Windows.

In response to non-executable memory, code reuse attacks were developed. Code reuse attacks counteract non-executable memory by not having to inject shellcode. Instead, because the attack reuses pre-existing code in memory (that is marked as executable), the attack will not be prevented by non-executable memory. In certain cases, some attackers find a way to disable non-executable memory on the operating system (usually with a code reuse attack), inject shellcode, and then complete their attack [35]. Nonetheless, having non-executable memory is a good deterrent and is already automatically enabled in most operating systems today.

2.3.2 Control-Flow Integrity

Control-flow integrity (CFI) is a recently developed defense that was first mentioned in 2005 [24]. Since then, many versions of CFI have been developed, including coarse-grained and fine-grained CFI. CFI was developed in an attempt to counteract code reuse attacks. The main principle of CFI is to maintain correct control flow for a program. CFI must be able to build accurate static or dynamic control flow graphs for the program that it is protecting. The control flow graphs contain the possible

paths that a program might follow. Whenever the program takes an illegal path not marked on the control flow graph, it means that an attacker may possibly be redirecting control flow.

However, in practice, due to the number of possible paths and unknown variables (such as cast types) before runtime, control flow graphs are often very complex and in certain cases impossible to generate beforehand [18].

Fine-grained CFI has a high performance overhead, and as stated before, the control flow graphs do not have 100% accuracy. Attempts have been made with coarse-grained CFI to improve performance numbers, but a number of papers have shown that it is very easy to bypass coarse-grained CFI due to the way coarse-grained CFI attempts to simplify the control flow graph [37, 38, 39]. More recently, it has been shown that even with correct control flow graphs, it might be possible to launch an exploit.

2.3.3 Complete Memory Safety

Complete memory safety is a very powerful technique that prevents almost all memory corruption attacks [14]. This technique relies on enforcing temporal and spatial safety by ensuring that the bounds of objects and pointers are correct. The only obvious disadvantage is the performance overhead that comes with this defense. Intel MPX [7], which will be discussed in a later chapter, is an example of this defense.

2.4 Randomization-based Defenses

In this section, we study randomization-based defenses. Randomization-based defenses base its defense on information hiding through randomization.

2.4.1 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) relies on randomizing the address space of a program [40, 41, 42]. The commonly randomized areas are the stack,

heap, and libraries. In the example for the return-to-libc attack, `printf` becomes randomized with ASLR, meaning that the next time the program runs, the address of `printf` would be at a different memory location. By randomizing the layout of the process, ASLR makes it difficult for the attacker to guess where everything in `libc` is located. There is a higher chance that the attacker will redirect the program to a faulty address and cause the program to crash before anything bad occurs.

Combining non-executable memory and ASLR is very common in operating systems and increases security with a relatively low overhead compared to other defenses.

2.4.2 Code Randomization

Code Randomization is actually as the name implies [43, 44]. It randomizes code through a series of transformations to make it harder for the attacker to know the memory layout of the code. Generally, the techniques involving code randomization are the re-ordering of instructions, changing which register contains the return value of a function (usually in the x86 instruction set, this register is eax), randomizing register allocations, inserting nonsensical instructions (NOP), inserting basic-blocks, and randomizing the stack locations used to save and restore register values.

2.4.3 Instruction Set Randomization

Instruction Set Randomization (abbreviated as ISR) randomizes the instruction set that the computer is using [45, 46, 47]. This method relies on hiding the instruction set from the attacker. Thus, the later the computer decides on a instruction set, the better. However, it does not prevent return-to-libc attacks, which do not rely on knowing the instruction set [46].

Chapter 3

Intel MPX (Memory Protection Extensions)

In this chapter, we review Intel MPX, a base and bounds checker than enforces spatial safety at the software, hardware, and operating system level. We also examine similar approaches.

3.1 Definition

To understand how Intel MPX functions, we need to understand what base and bounds mean. Base refers to the start of the memory location of an object or pointer. Bounds is the size of the object or pointer in question. For clarity, we demonstrate this with an example below.

This code snippet allocates an array of size 1024. In this case, the base is the

```
1  char *buf = malloc(1024);
```

Code Listing 3.1: Simple Malloc Example

memory location at the start of the array. The bounds is the size of the array, which is 1024 in this example. The memory location at the end of the array is obtained by adding the bounds to the base. Note that since the index starts at 0, the bounds is actually 1023.

If the program tries to access a memory location not recorded as a "base",

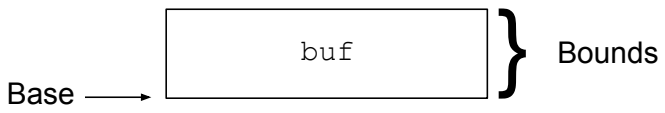

Figure 3-1: Base and Bounds

Intel MPX will report a bounds violation error.

3.2 Intel MPX

3.2.1 Design

Intel MPX is a pointer-based solution to enforcing memory safety. Intel Pointer Checker is the software-based predecessor of Intel MPX. Both defenses function similarly, though Intel Pointer Checker is meant to be used as a diagnostic tool due to its high overhead. Figure 3-2 demonstrates the change in performance once Intel Pointer Checker is enabled. Instead, Intel MPX is recommended, which Intel claims has much faster run-times.

Traditionally, the pointer-based approach modifies the pointers themselves to maintain base and bounds information. Instead of storing the metadata in each pointer, Intel MPX records the base and bounds information of every pointer allocated into a separate bounds table. The bounds table takes a hierarchical approach, similar to how memory is organized. A bounds directory is used to organized bounds tables, which in turn, record the base and bounds information for individual pointers. On a pointer dereference, the table is consulted and a check is performed to ensure that the operation is legal, meaning the pointer still maintains correct base and bounds. The bounds table resolves the compatibility issues that traditional pointer-based approaches struggle with as the pointers in this scheme are left untouched.

Fundamentally, Intel MPX is easy to use. The requirements are to possess a fifth generation or later Intel processor and to have kernel support in the operating

system for Intel MPX. Either the gcc or icc compiler can be used. If using gcc, then gcc 5.0 or later is required. At compile time, the developer must specify the correct configuration flags, at which point Intel MPX will be enabled for the program. Some of the possible values for the flags are:

- `-fcheck-pointer-bounds`: tells the compiler to compile with Intel MPX enabled
- `-fchkp-check-read`: every time a read to memory occurs, checks that the memory read is within bounds
- `-fchkp-check-write`: every time a write to memory occurs, check that the memory being written to is within bounds
- `-fchkp-store-bounds`: every time a write to memory occurs, stores the base and bounds
- `-fchkp-narrow-bounds`: use field bounds instead of full bounds (for example, a struct in C contain multiple fields)
- `-fchkp-first-field-has-own-bounds`: the first element in the struct has its own bounds; otherwise, the first element in the struct has the same bounds as the whole struct

3.2.2 Implementation

Bounds Table

As stated earlier, Intel MPX achieves spatial safety by checking the base and bounds of allocated memory. Four new bound registers were introduced to the Intel architecture and are exclusively used by Intel MPX. Given the limited number of registers available for storing base and bounds, we will need more than just registers to store bounds information. The base and bounds are stored in a bounds table instead and swapped out to the bounds registers as needed. Two new assembly instructions `BNDLDX` and `BNDSTX` were introduced, which correspond to bounds load and bounds

store respectively. `BNDLDX` loads an entry from the bounds table to a bounds register. `BNDSTX` stores an entry into the bounds table.

The bounds table is a two-level radix tree, where the index is the virtual address of all the pointers in the bounds table. A bounds directory is used to look-up a particular bounds table. Each entry in the bounds table consists of four elements (the size of pointers): (1) lower bounds, (2) upper bound, (3) check pointer value, and (4) unused.

Performing Checks

Intel MPX instruments code during compile time in order to perform its base and bounds checking during run-time. There are five ways in which a pointer may be dereferenced [50].

1. return value from a function call: both the pointer and its bounds are returned
2. load from memory: bounds are loaded from the bounds table
3. function argument: bounds are passed along with the pointer
4. object address: the object address acts as the low bound and its size is used to compute the upper bound
5. field address: bounds are narrowed for a field address

We demonstrate bounds checking with an example.

```
1 int function (void **p)
2 {
3         int *ptr = (int *)(*p);
4         return *ptr;
5 }
```

Code Listing 3.2: Bounds Checking Example [50]

In Code Listing 3.2, the base and bounds information of `p` are retrieved from the bounds table at line 3. The actual check is performed when the pointer is returned at line 4.

```
1    int buf[100];
2     int foo (int i)
3     {
4         int *p = buf;
5         return p[i];
6     }
```

Code Listing 3.3: Bounds Checking Example 2 [50]

In Code Listing 3.3, when `p` is assigned to `buf`, it receives the same base and bounds of the array. In this case, because `buf` is an `int` array, the base and bounds are `[buf, buf + 399]`. (It is 399 rather than 99 because each element in the array is 4 bytes). During line 5, a check is performed to ensure that we are still within the base and bounds of the array.

Bounds Narrowing

```
1  struct S1
2     {
3       void *field1; //size is 4
4       void *field2; //size is 4
5     };
```

Code Listing 3.4: Bounds Narrowing [50]

If bounds narrowing is enabled (which is not turned on by default), Intel MPX will attempt to get a better estimate of the individual fields in a struct. However, the fields in a struct has their bounds narrowed only when the fields are referenced. In this example, `field1` has base and bounds as `[S1, 3]` and `field2` has base and bounds of `[field2, 3]`, `S1` has base and bounds of `[S1, 7]`.

3.2.3 Performance Overhead and Compatibility

In this section, we look at the performance overhead of Intel MPX and possible compatibility issues that may arise.

As mentioned earlier, an entry in the bounds table consists of four pointers. If a program allocates memory for a pointer, the program also needs to allocate memory

for the bounds table entry, which would be four times the size of the pointer. If the program uses a lot of pointers, the total memory consumed can quickly add up (4x). Additionally, given that we are fetching data from memory, there could be a performance impact caused by address lookup and cache usage [59].

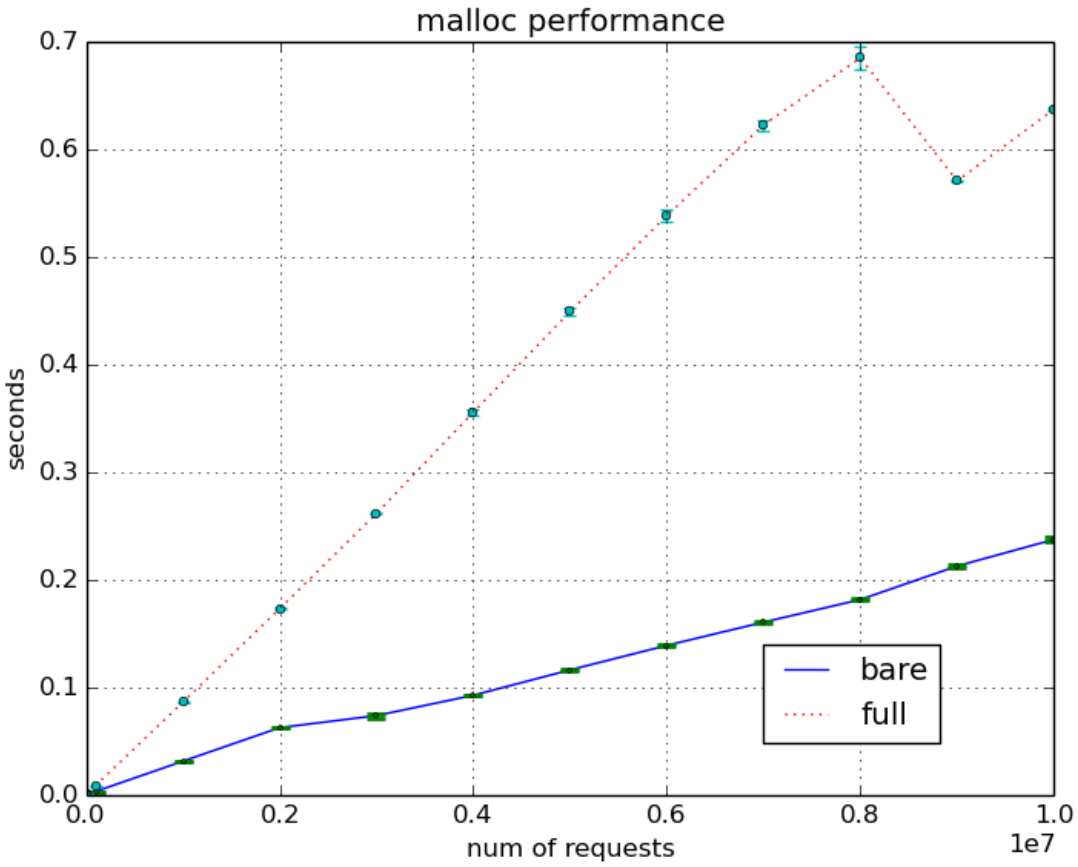

Figure 3-2: malloc performance when compiled with and without Intel's Pointer Checker

To get an accurate estimation about the performance overhead of Intel's Pointer Checker (the software-only solution), we performed a test with `malloc`, which is the default allocator for C. For the experiment, we continuously called `malloc` and `free`. The `num of requests` in the graph indicate the number of times `malloc` was called. `bare` indicates that the code was compiled without Intel's Pointer Checker. `full` indicates that the code was compiled with Intel's Pointer Checker. The times were

recorded using C's time library with the `clock` function. The testing was done on a personal laptop and on a virtual machine running Ubuntu 14.04. For this experiment, Intel's compiler `icc` was used.

From what is shown, we can see that when the code is compiled with Intel's Pointer Checker, there is a 3.5x performance overhead on average.

Compatibility-wise, Intel's Pointer Checker is compatible with most operating systems with the exception of Mac OS. It is software-based and consists of wrapper code for the memory management functions. The program needs to be compiled with icc and specific flags that enable Pointer Checker. Intel's Pointer Checker itself transforms the code during compile time to include base and bounds check.

3.3 Related Work

Numerous defenses [3, 48, 49] have emerged to safeguard against memory corruption attacks by tracking pointer usage and ensuring pointers are manipulated correctly. We highlight some of the more popular approaches.

1. Object-Based Approach

2. Pointer Based Approach

3.3.1 Object-Based Approach

For the object-based approach [25, 26], the entire object's base and bounds are recorded and checked. This approach is favorable because it is easily compatible with the existing system. The memory layout does not need to be modified in order to adopt this approach. Secondly, for this scheme, pointers are always mapped to an object. While compatibility might be a reason someone chooses to adopt this scheme, there are many reasons not to use this approach. One important reason is that this approach does not entirely enforce memory safety due to the number of edge cases that arise. The Softbounds paper [3] notes an important case, in which pointers to

the same object are treated as the same. In their example, they use:

```
1  struct person{
2    int age;
3    int height;
4   };
```

Code Listing 3.5: Simple struct Example

In Code Listing 3.5, the pointer to person and the pointer to age, the first parameter in the structure, both point to the same memory location. Therefore, according to the object-based approach, the pointer to age would have the same base and bounds information as the pointer to person. This is incorrect behavior. The pointer to age should have a smaller bounds than the entire struct. Since this approach is object-based, it would not be able to narrow the bounds like Intel MPX can.

In addition to edge cases such as the previous example, there is a significant overhead with the object-based approach due to the complexity of storing the base and bounds information of the pointers mapped to objects. For the reasons pointed out, the object-based approach is not very favorable. It does not enforce complete memory safety and incurs high performance overhead.

Valgrind

Valgrind is an example of an object-based approach and widely used in industry to detect memory bugs such as use-after-free, memory leaks, and reading and writing to invalid memory locations. Memcheck is included in Valgrind's source code. Valgrind often introduces a huge drop to performance due to how it instruments the code. Therefore, Valgrind is usually used as a diagnostic tool to detect mistakes made by the developer before shipping out the final product.

Whenever memory has been freed, Valgrind will record the address of that piece of memory in a separate area. If a pointer tries to dereference an already freed memory address, Valgrind will report an error to the developer. Sometimes, this might fail if

another pointer is reassigned the memry and the previous pointer's reference has not been removed [14].

For memory leaks, Valgrind will report the number of memory blocks that are "maybe lost" and "definitely lost" and print out the stack trace that lead to the issue. Valgrind will also report illegal read and writes to bad memory addresses. As it is an object-based approach, Valgrind does not offer complete temporal safety.

3.3.2 Pointer-Based Approach

The second is the traditional pointer-based approach [20], which involves recording base and bounds information for each individual pointer, rather than each object. In the traditional scheme, the pointer is modified to include this information. It solves many of the concerns that arise with the object-based approach. The edge cases, such as the aforementioned case where two pointers pointing to the same memory location are treated with the same base and bounds information, is solved. However, the most obvious disadvantage with this scheme is that it requires the pointers to be modified and changes the layout of the program. Compatibility becomes a major concern for this approach.

Softbounds+CETS

There are also similar safeguards, such as Softbounds+CETS. Softbounds itself is another example of a base and bounds checker. Like Intel MPX, Softbounds also records base and bounds information of individual pointers. Instead of a bounds table, Softbounds uses a shadow table. Softbounds transforms the code with an LLVM (LLVM is a compiler) pass to record a base and bounds or check a base and bounds depending on the function or method. It contains a lot of wrapper code for the memory management functions. It bares a lot of similarities to Intel MPX, with the exception that it is software only. However, the use-after-free checking for CETS [8] is much lighter weight than Intel's Pointer Checker according to the CETS authors.

Chapter 4

Custom Memory Allocators

In this chapter, we discuss memory management and memory allocators. Then we expand on the topic with custom memory allocators including popular implementations, such as Apache's and nginx's custom pool allocator. We assume the reader has basic knowledge of memory in operating systems.

4.1 Background

Memory management is a crucial function for any computer. It involves handing off memory from the operating system to the individual processes. Memory is a resource required by all programs to run properly. When a process wants memory, the process simply has to ask the operating system. The operating system will hand off memory to the process that asks for it and keep track of who has what memory. It is similar to how a library functions. Everyone can borrow memory, but only one process can borrow a particular block of memory. Once the block of memory is checked out by the process, no other process can check it out until the memory has been returned to the operating system. Processes also cannot hand off memory to each other (i.e., once process A terminates, it cannot give the memory it borrowed to process B). Only the operating system can lend out memory. Unlike a library, there is no hard deadline that the process will have to return the memory by. However, the operating system can at any point reclaim the borrowed memory (such as in the case of a restart or a

forceful program termination).

Manual memory management is performed when the developer makes a conscious decision on when and how much to ask the operating system for memory. This is done through allocators. With allocators, developers are given the freedom to allocate and free memory at an given moment in the program. Though the concept of manual memory management is relative straightforward, dealing with allocators can often be frustrating. It is easy to make an error, such as forgetting to free a pointer, which would cause a memory leak to occur. If the program was running for a long period of time and the memory leak occurred every ten minutes or so, for example, then the leakage could add up.

The function of an allocator is to provide available memory to the developer when asked, to inform the developer that no free memory is available at this moment, or to take back memory when the developer no longer leads it. For performance and space efficiency (avoiding fragmentation), the allocator generally maintains a free list. The free list is most commonly a singly-linked list because elements in a linked list do not have to be adjacent in memory. It is very rarely that freed memory from the program returns to the heap. Most of the time, the freed memory is shuffled into the free list, where it may be reused at a later point by the program. Depending on the design, the free list may or may not be initially empty. The size of each free memory block in the list may vary depending on the allocator and on the sizes that the developer chooses.

Figure 4-1 depicts a free list after a number of allocations and frees have been made. When a request for memory is made to the operating system, the free list is checked to see if there are any memory blocks available. There are many implementations of how the free list performs this check. If there are no memory blocks available in the free list, memory is then obtained from the heap, which contains dynamic memory that is available for the developer to take. Otherwise, a free memory block of the appropriate size is removed from the free list and given.

C and C++ comes with a pre-packaged allocator, which performs dynamic memory allocation. Dynamic memory allocation differs from static and automatic

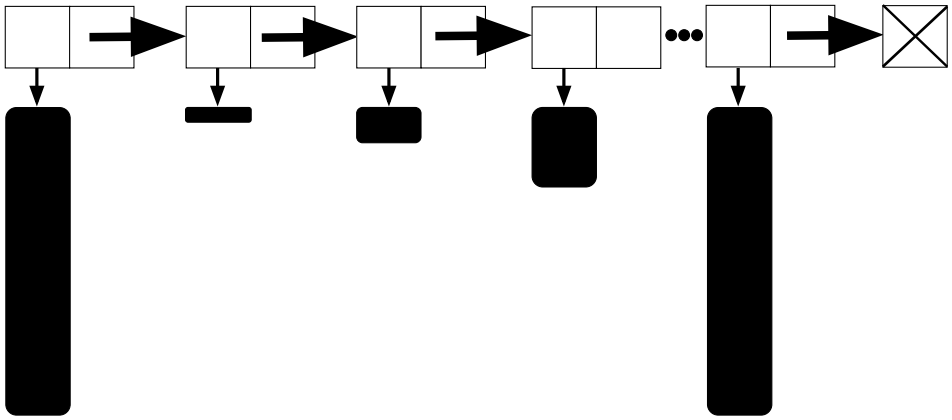

Figure 4-1: free list with free memory blocks

memory allocation. In static memory allocation, the variable persists for the entire lifetime of the program. In automatic memory allocation, the variable persists for the duration of the function. In dynamic memory allocator, the variable persists for as long as the developer wishes or until the end of the program. Dynamic memory allocation allows the developer more control in allocating and freeing memory, but also allows for more mistakes to happen in between.

In general, the heap is used whenever there is a request for dynamic memory allocations. When the developer wants to get more memory, the developer takes it from the heap and when the developer needs to free an object, the developer returns the memory to the heap. As discussed in chapter two, failure to properly obtain or free memory can cause a multitude of problems. As one might expect, the two common mistakes are when the developer fails to free memory properly after it has been used and when the developer fails to check if memory was properly allocated. Below, we discuss reasons why these mistakes sometime occur.

One possible consequence of memory corruption is a memory leak. A memory leak occurs when memory becomes allocated, used in the program, and the developer forgets to free it after use. Since the memory never becomes freed, it will not be used again. We illustrate this with a simple example, where the developer forgets to free

memory after allocating.

In Code Listing 4.1, we first obtain memory from the heap by calling `malloc(size)`,

```
1 void makeArray(){
2   char *arr;
3   arr = (char*) malloc(1024);
4   .. //do stuff
5 }
```

Code Listing 4.1: Memory Leak Example

where `size` is the amount of memory the developer wants in bytes. However, in this example, the memory allocated for `arr` is never freed. The heap has not been informed that this piece of memory has been freed and thus, nothing else will be allowed to use it. Memory is wasted.

The proper way, instead, would be to free the memory such as in Code Listing 4.2

In this example, we add an additional line, which calls `free` after the variable is no

```
1 void makeArray(){
2   char *arr;
3   arr = (char*) malloc(1024);
4       //do stuff
5   free(arr);
6 }
```

Code Listing 4.2: Freeing Memory Example

longer used. `free` marks the memory as no longer being in use and allows it to be reused.

Another common case is when the developer fails to check if memory was successfully allocated. If memory is not successfully allocated, a null pointer is returned instead when `malloc` is called.

We also illustrate this with an example below. These two examples are not the only scenarios in which an error may occur when dealing with manual memory management. These errors occur very easily. Most of these errors are resolved once the program exits. Once the program terminates, all memory is returned to the operating

```
1  void makeArray(){
2   char *arr;
3   arr = (char*) malloc(1024);
4   if ( arr == NULL){
5     // failure to obtain memory
6     // do stuff to fix problem and/or inform program
7   }
8     //do stuff
9   free(arr);
10 }
```

Code Listing 4.3: Checking `malloc` Example

system. Unfortunately, an attack may have already occurred by then.

The general-purpose allocator is implemented in several ways. One of the two most common ways is via a naive implementation with sbrk or with Doug Leas `malloc` [12], which stores pre-allocated chunks of memory into sorted bins of varying powers of two for faster performance. When searching for memory in the free list, the allocator looks for a best fit match, meaning it tries to find a memory block that is most similar in size to the size that has been requested by the developer. The C allocator uses the implementation by Doug Lea, whose implementation is very efficient in terms of performance.

4.2 Custom Memory Allocators

Most developers who wish for an improvement in performance may choose to implement their own custom memory allocators. This section details a number of popular custom memory allocators.

4.2.1 Per Class Allocators

Per-class allocators are relatively straightforward [51]. These allocators use the prepackaged `malloc` provided by the C/C++ libraries (abbreviated as libc). However, as the name suggests, per-class allocators maintain several free lists per class [9]. In the general case, the free list contains different-sized memory chunks. The allocator must

iterate through the free list to find a block of the appropriate size, which takes time. However, in per-class allocators, each free list contains memory blocks of identical sizes because there are more than one free list. In per-class allocators, the burden of having to iterate through the free list is eliminated. If there is a free memory block in the list, the block can immediately be returned because it will be of the appropriate size.

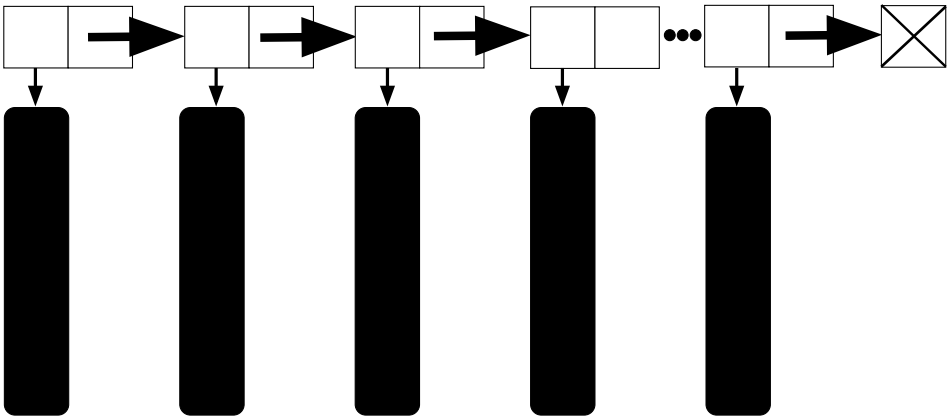

Figure 4-2: A free list with a fixed memory block size

One of the bottlenecks is the time it takes for the allocator to iterate through the free list. There are several optimizations that deal with this. Some free lists allow the developer to know the size of each memory block according to index, but in the worst case scenario, the list must be checked at `[index, MAX_INDEX)`. Some optimizations that developer use to eliminate this worse case scenario is by first checking index and if there are no free memory blocks at index, jump to `MAX_INDEX` instead. However, this is not efficient in terms of space. By making each free list per class, the per-class allocator essentially gets rid of this problem because all memory blocks are of the appropriate size. Note that this adds complexity and according to [9], the trade-off between the increased complexity and the performance of the per-class allocator is negligible and sometimes, the per-class allocator performs worse than the general-purpose allocator.

Another possible advantage of this method is that less fragmentation may occur. Fragmentation is a tricky problem and occurs when memory blocks of different sizes are returned to the free pool.

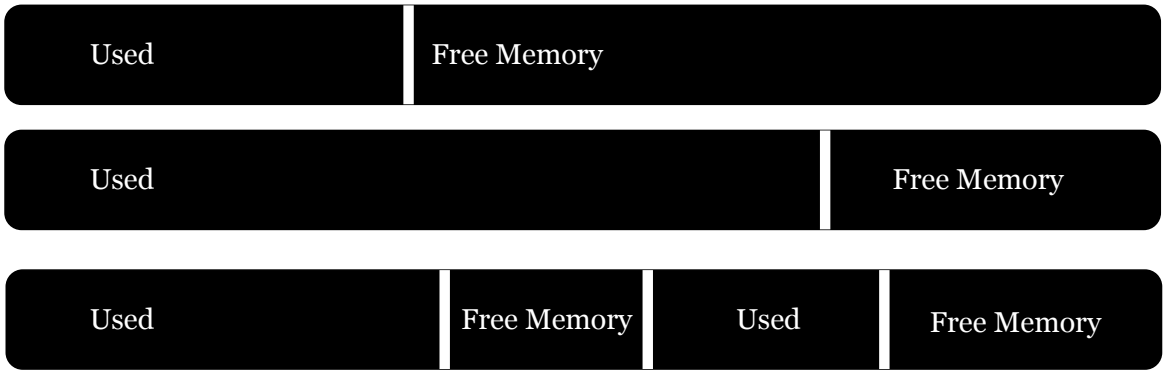

Figure 4-3: Memory Fragmentation

Eventually, as shown in the figure above, the free memory available becomes fragmented and becomes interleaved with used memory. The developer is not able to use all of the available free memory depicted in the figure above for one single allocation because the free memory is no longer adjacent. Thus, per-class allocators may help alleviate this effect because all requests are of the same size.

4.2.2 Region and Pool Allocators

Region allocators allocate huge chunks of free memory beforehand. When requests for memory are made, memory is taken from the pre-allocated chunks of memory, not from the heap. This reduces overhead as it reduces the amount of communication that occurs between the process and the operating system. Region and pool are used interchangeably.

4.2.3 Custom Pattern

Custom pattern allocators are optimized for specific pieces of code. The program is analyzed during run-time and the patterns in which memory is allocated is observed and recorded to see if there are specific trends in which memory is allocated. If the memory patterns of the code are known beforehand, then optimizations can be done knowing that pattern. For example, for a specific program, it may be possible that the program allocates mostly for arrays of size 1024 and 4096. In this case, the developer knows that it may be wiser to allocate free memory blocks of only size 1024 and 4096.

4.2.4 Apache Pool Allocator

In this section, we explain how the Apache pool allocator is implemented. We first introduce the three main structs of the Apache pool allocator which are `memnode`, `allocator`, and `pool`. Then we follow with an example on how pools are used, and then explain why Apache chose to adopt the pool allocator. For more reference, refer to `apr_pools.c` and `apr_pools.h`.

Memnode

The most basic component of the Apache pool allocator is the `apr_memnode_t`, which is used by both the `allocator` and `pool` structs. `apr_memnode_t` is a singly-linked list. It is the most basic component of Apache's memory management, as it contains metadata on the memory owned by Apache.

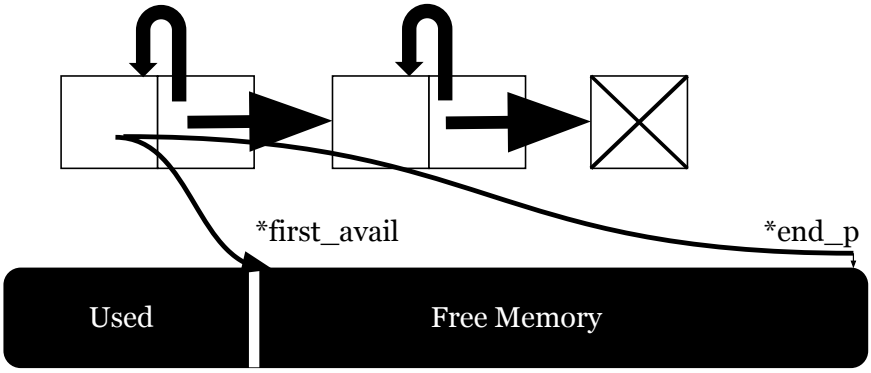

Figure 4-4: A free list with a fixed memory block size

Each `apr_memnode_t` points to the next `apr_memnode_t`, unless it is the last `apr_memnode_t`, which points to `NULL`. Each `apr_memnode_t` also contains a reference to itself. All `apr_memnode_t` nodes point to a specific block of memory that they have been assigned. Additionally, there is a pointer to the memory address of the remaining available memory in the memory block, `first_avail`, as well as a pointer to the end of the free memory block, `end_p`. `index` records the size of the entire memory block. `first_index` records the amount of free memory still available. To get to the start of the entire memory block, we can use `end_p` and `size`. For reference, the code source for `apr_memnode_t` is provided.

```
1 struct apr_memnode_t {
2       apr_memnode_t *next;    //ptr to next node
3       apr_memnode_t **ref;  //ptr to self
4       apr_uint32_t   index;
5       apr_uint32_t   free_index;
6       char          *first_avail;
7       char          *endp;
8 };
```

Code Listing 4.4: apr_memnode_t

Allocator

The main purpose of the `apr_allocator_t` struct is to maintain a free list of memory blocks and the bookkeeping variables for that list. The free list is implemented with

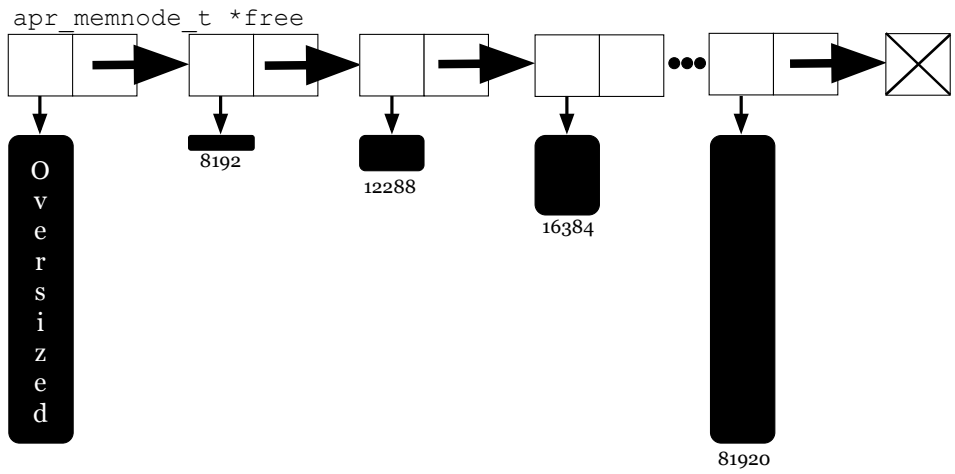

Figure 4-5: allocator

apr_memnode_t elements. It contains MAX_INDEX slots, and the size of each memory block is based on where it is in the list.

$$\text{size} = (i + 1) * \texttt{BOUNDARY\_SIZE}$$

i is the index number and BOUNDARY_SIZE is a set variable. By default, Apache sets BOUNDARY_SIZE to 4096. The exception to this rule is slot 0, which contains memory block sizes that are greater than the maximum memory block size at the last slot. Slot 0 houses the over-sized memory blocks.

The bookkeeping variables are mainly there to keep track of the number

```
1 struct apr_allocator_t {
2     apr_uint32_t          max_index;
3     apr_uint32_t          max_free_index;
4     apr_uint32_t          current_free_index;
5     apr_pool_t           *owner;
6     apr_memnode_t        *free[MAX_INDEX];
7 };
```

Code Listing 4.5: apr_allocator_t

of free blocks and the current index of the list. Memory blocks are not allocated beforehand. The index of the free list increases as needed. Whenever a request is sent to the allocator to obtain a free memory block, the following steps occur. First, the allocator checks to see if the size is at least MIN_ALLOC. If not, the size will increase

to be `MIN_ALLOC`. Then, the allocator does a walk around the linked list in search of a memory block that is either the same size or greater than the requested size. If no block of that size is available, the allocator allocates a new memory block and adds it to the list. Lastly, the allocator keeps a record of the current pool that owns the allocator.

Pool

```
1 struct apr_pool_t {
2     apr_pool_t            *parent;
3     apr_pool_t            *child;
4     apr_pool_t            *sibling;
5     apr_pool_t           **ref;
6     apr_allocator_t       *allocator;
7 };
```

Code Listing 4.6: apr_pool_t; Note: Some parameters have been removed for simplicity.

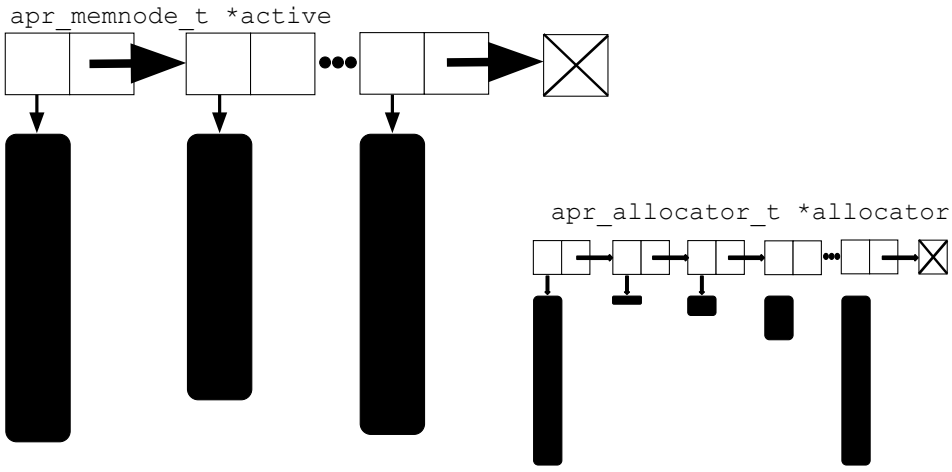

Figure 4-6: pool

The `apr_pool_t` struct is the main component of the Apache pool allocator, and the struct that is used by the developer to obtain memory. To put it simply, the pool contains two linked lists, one containing the active memory blocks that are currently in use, and a free list of non-active memory blocks, which is maintained by

the `apr_allocator_t` struct. We have discussed the free list in the previous section. We will focus on the active list in this section. When a request for memory is first made, the active list, rather than the free list is first checked. The active list is checked if the current memory block has enough memory available to fulfil the request. If there is not enough memory, then the request will be passed to the free list. Otherwise, the pool will use the memory block in the current active `apr_memnode_t` that it is on, pass the `first_avail` pointer to the developer, and then move the `first_avail` pointer down accordingly.

After a request for memory, we can see that the pointer `first_avail` has moved right from its initial position to indicate that memory has been allocated to the developer and is no longer free.

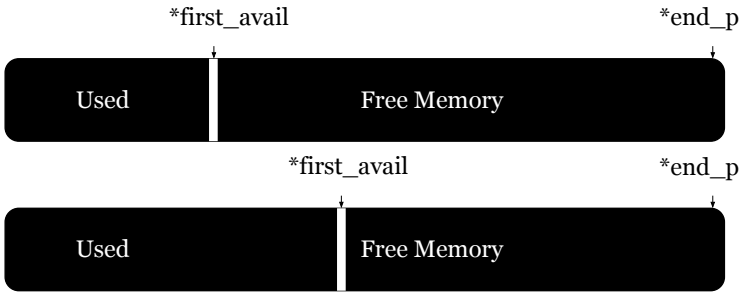

Figure 4-7: pool

Example

In this section, we will use an example to demonstrate how the Apache pool allocator can be used.

```
1    apr_pool_t *pool
2    apr_pool_create(&pool, NULL);
3    char *buf;
4    int *buf2;
5    buf = apr_palloc(pool, SIZE);
6    buf2 = apr_palloc(pool, SIZE2);
7    apr_pool_destroy(pool);
```

Code Listing 4.7: Apache pool allocator example

In line 2 of Code Listing 4.7, we initialize the pool. In lines 5 and 6, we allocate from the pool. Since `buf2` is allocated right after `buf`, the two arrays are adjacent in memory. Finally, when we want to free the memory, we simply destroy the entire pool. There is no need to individually free everything that was allocated.

Utility

The Apache pool allocator is used for several main reasons. First, the code is fairly straightforward and easy for most developers to understand and use. Second, there is a performance increase. Finally, it prevents memory leaks from occurring.

As shown, the Apache pool allocator is straightforward and easy to use. Unlike the regular `malloc`, the developer is not required to free all the variables that he initialized; therefore, that burden is lifted off the developer and only a call to destroy the pool at the end of the function is required. Pools may also be given a lifetime, in which the pool is automatically destroyed once a certain amount of time has passed. In Apache, there is a parent pool that all other pools are children of. Once the parent pool is destroyed, the children are also destroyed. Thus, once the program ends, the parent pool is destroyed and all memory is returned to the heap.

Secondly, the use of pool allocators allows for a performance increase. To understand this, we must understand the difference between general allocators and pool allocators. Unlike general allocators, which repeatedly ask the operating system for memory blocks when needed, when a request for memory is made for pool allocators, the request is fulfilled by the process rather than the operating system. This is possible because the pool allocator asks for large chunks of memory at a time from the operating system, which can be used for many requests. Because Apache is managing its own allocations, it has a performance increase.

Finally, the use of pool allocators means that there won't be any memory leaks. The pools themselves have a set lifetime. At the end of their lifetime, the pools are destroyed and the memory that they previously owned is freed. Developers do not have to free each individual variable for which they allocated memory. This lifts the burden of potentially causing a memory leak off the developer's hands.

4.2.5 nginx Pool Allocator

In this section, we examine the pool allocator of nginx, another popular HTTP server, and how it works. The nginx pool allocator and the Apache pool allocator are actually pretty similar. For reference, refer to `ngx_palloc.c` and `ngx_palloc.h`.

Being both pool allocators, the design of the nginx pool allocator is almost identical to the Apache pool allocator. Like the Apache pool allocator, the nginx pool allocator asks the operating system for large chunks of memory and then manages subsequent requests locally by using its pre-allocated chunks of memory.

The nginx pool allocator also manages its free list similarly, in that the free list houses a number of free memory nodes. The nginx pool allocator uses the first element in its free list like the Apache pool allocator. The first element is used to store over-sized memory blocks. However, the difference is that the sizes of the free memory blocks in the linked list is based on an amount specified by the user. Aside from the first element, the rest of the linked list contains blocks of sizes specified by the user. Memory is aligned by an internal number when allocated.

Below, we show the two main structs of the nginx pool allocator.

```
1 typedef struct {
2     u_char               *last;
3     u_char               *end;
4     ngx_pool_t           *next;
5     ngx_uint_t            failed;
6 } ngx_pool_data_t;
```

Code Listing 4.8: ngx_pool_data_t struct

The `ngx_pool_data_t` serves as the most basic of components, the singly linked list. Like the Apache `apr_memnode_t`, it contains a reference to the next node. `last`, `end`, and `failed` will be explained in a later example.

The `ngx_pool_t` functions similarly to Apaches `apr_allocator_t` and `apr_pool_t`, but is much simpler. `d` is the free list used to manage available memory blocks. `max` is the maximum size a memory block in anywhere that isnt the over-sized list. `current` is the current pool that we are obtaining memory from. `large` is used to store over-sized memory nodes.

```
1  struct ngx_pool_s {
2      ngx_pool_data_t       d;
3      size_t                max;
4      ngx_pool_t           *current;
5      ngx_pool_large_t     *large;
6  };
```

Code Listing 4.9: ngx_pool_s struct

Next, we show how a pool is created.

```
1  ngx_pool_t *ngx_create_pool(size_t size, ngx_log_t *log)
2  {
3      ngx_pool_t  *p;
4
5      p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
6      if (p == NULL) {
7          return NULL;
8      }
9      p->d.last = (u_char *) p + sizeof(ngx_pool_t);
10     p->d.end = (u_char *) p + size;
11     p->d.next = NULL;
12     p->d.failed = 0;
13
14     size = size - sizeof(ngx_pool_t);
15     p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size :
           NGX_MAX_ALLOC_FROM_POOL;
16     p->current = p;
17     return p;
18 }
```

Code Listing 4.10: ngx_pool create pool

The nginx pool allocator uses `ngx_memalign`, which is a function that calls `posix_memalign` and then logs the action. The parameters of the `ngx_pool_t` are then set. last refers to the starting memory location in which there is free memory. end refers to the end of the free memory block. `failed` is used during the allocation process, in which the current pool is searched for available memory. Once failed reaches four, the next memory pool is looked at.

If the size is larger than page size, then it is placed in the over-sized node. Otherwise, the allocator checks if a pre-allocated chunk fits the criteria. If not, then it will communicate with the operating system to get memory.

Example

We demonstrate how the nginx pool allocator works with a simple example.

```
1 ngx_pool_t * pool = ngx_create_pool(4096);
2 char *p = (char *)ngx_palloc(pool, 1024);
3 char *c = (char *)ngx_palloc(pool, 1024);
```

Code Listing 4.11: ngx_pool create pool

Like the Apache pool allocator, the code to initialize the pool and to allocate memory is straightforward. The difference is that the `ngx_create_pool` function takes in a parameter, size. size is used to specify the size of the pool. To allocate from the pool, the developer only needs to call `ngx_palloc` and provide two parameters. The first is the pool the developer wishes to allocate from and the second is the size of the object. In the case that we use all 4096 bytes, then the pool allocator will retrieve a new memory block, where the size will be equal to the amount specified by the developer.

```
1  void * ngx_palloc(ngx_pool_t *pool, size_t size)
2  {
3      uint8_t      *m;
4      ngx_pool_t  *p;
5      if (size <= pool->max) {
6          p = pool->current;
7          do {
8              m = (uint8_t *) ngx_align_ptr(p->d.last, NGX_ALIGNMENT);
9
10             if ((size_t) (p->d.end - m) >= size) {
11                 p->d.last = m + size;
12                 return m;
13             }
14             p = p->d.next;
15
16         } while (p);
17         return ngx_palloc_block(pool, size);
18     }
19     return ngx_palloc_large(pool, size);
20 }
```

Code Listing 4.12: ngx pool allocator usage example

As shown in the method ngx_palloc, if the current memory block does not have enough space to store the new request, then a new block is allocated.

Utility

The nginx pool allocator is used for reasons similar to the Apache pool allocator. Like the Apache pool allocator, it is easy to learn and use. There is also a performance increase, making it highly favorable in the eyes of many developers. Like the Apache pool allocator, we will take advantage of the the design. More specifically, we exploit the fact that that the pool allocator allocates big memory chunks at a time.

Unlike the Apache pool allocator, the nginx pool allocator allows the developer the freedom to choose the initial pool size. We ran a test to see how the initial pool size affect performance.

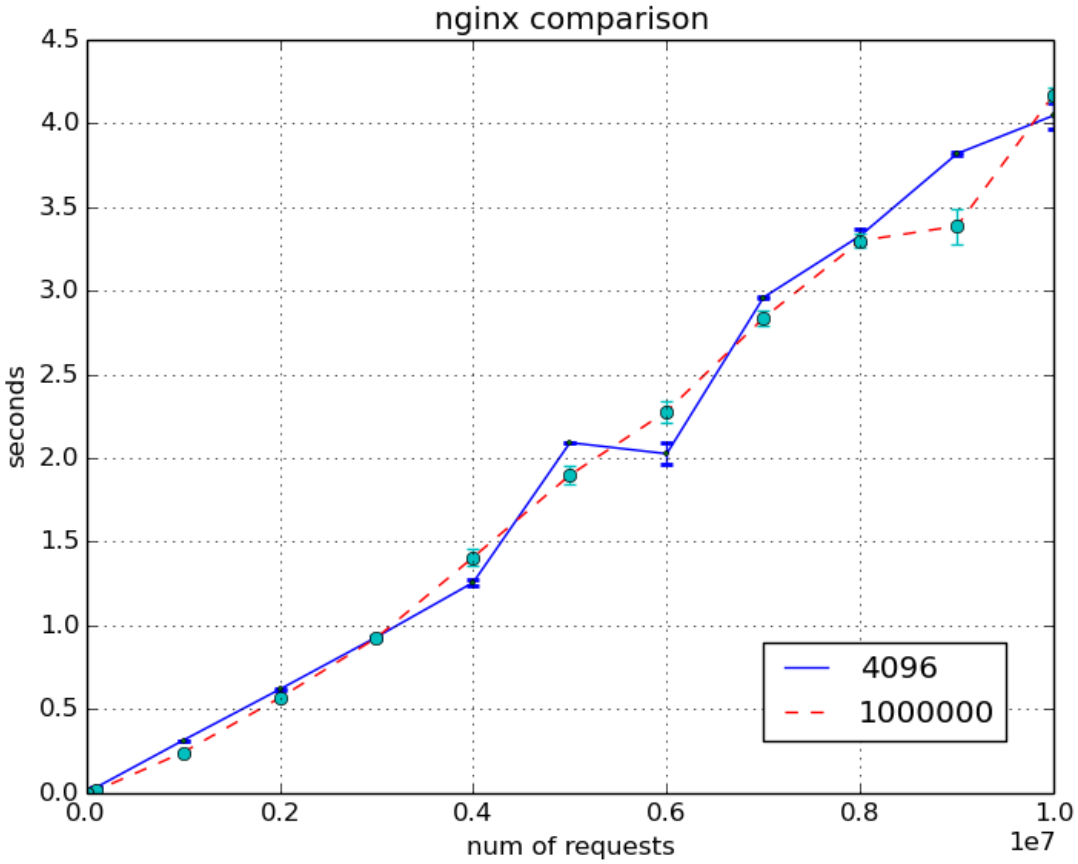

Figure 4-8: nginx Pool Comparison

For this test, we had two initial pool sizes, which are 4096 and 1000000. Each instance was subjected to a number of allocation requests before the pool was finally destroyed at the end. From what can be seen in the graph, the performance of both instances are similar. This is due to the fact that after the initial pool is used, the nginx pool allocator allocates block sizes set by the developer. It does not allocate larger chunks. In essence, the nginx pool allocator behaves like `malloc` after the memory in the initial pool has run out.

Chapter 5

Security Evaluation of Custom Allocators

5.1 Previous Work

We now present the history of memory allocator corruption exploits.

5.1.1 Doug Lea Allocator

In Smashing the Heap for Fun and Profit [52], named after the infamous Smashing the the Stack for Fun and Profit [2], MaXX demonstrates one of the first-ever exploits against the general-purpose allocator. The exploit takes advantage of a heap overflow through an adjacent heap corruption. In addition, MaXX demostrates unlinking and frontlinking techniques for corrupting meta-data.

MaXX takes advantage of the fact that the Doug Lea allocator places meta-data within the memory block itself. Therefore, by overwriting parts of the heap in a particular manner, the attacker is able to change the meta-data of the memory blocks. Changing the meta-block data leads to unlinking and frontlinking techniques, which are used to carry out the exploit.

The unlink technique takes advantage of the linked list structure used to store free memory blocks.

```
1 #define unlink( P, BK, FD ) {            \
2  [1] BK = P->bk;                         \
3  [2] FD = P->fd;                         \
4  [3] FD->bk = BK;                        \
5  [4] BK->fd = FD;                        \
6 }
```

Code Listing 5.1: unlink, taken from [52]

The call to `free` essentially is a call to `unlink`. When `unlink` occurs, meta-data that is adjacent to the memory chunk is looked at. With a heap overflow, the attacker is able to overwrite the meta-data with data of their choosing to execute the `unlink` technique.

The frontlink technique is much more difficult to achieve and as been noted by the author has never been executed in the wild. Frontlinking occurs when there is an attempt to insert a memory block into the free list. The exploit also tries to overwrite meta-data to trick the computer into processing a pointer.

5.1.2 jemalloc per-class allocator

`jemalloc` is a per-class allocator that focuses on performance. The exploits for this allocator take advantage of adjacent heap corruption. For this allocator, the attacker first sets up the heap. As a per-class allocator, the free list contains memory chunks of the same size. The attacker first allocates many times to insert malicious data and then frees every other memory chunk so that the heap looks like:

Figure 5-1: Adjacent Heap Corruption

Chapter 6

Custom Allocator-Aware Memory Safety

Generally, developers use custom memory allocators to improve performance or to make memory management for languages like C and C++ easier. In this chapter, we focus on the performance benefits of custom memory allocators along with the trade-offs that come with using them.

In chapter 4, we defined custom memory allocators and the types of custom memory allocators. We concluded that the presence of a custom memory allocator reduces the performance overhead of having to repeatedly ask the operating system for memory and by extension, improves the application's performance. However, as with all designs, there are trade-offs to consider. In the case of custom memory allocators, the performance benefits of using custom memory allocators become unclear when we add security defenses to the equation. For instance, consider the case where we are not using any custom memory allocators and are using the system memory allocator. In the following code example, we allocate memory by calling system `malloc`.

```
1 int *arr;
2 arr = (int*) malloc(sizeof(int) * 5); // int array of size 5
3 free(arr);
```

Code Listing 6.1: Simple Malloc

If we use Intel MPX to enforce memory safety for an application, we know that we must establish the base and bounds for the memory allocated by the operating system for the application at some point. But how does this occur? For Linux and GCC, the memory management functions in libc such as `malloc, calloc, realloc, memcpy, free` become modified by MPX wrapper functions. When the application makes a call to `malloc`, the application is first makes a call to a MPX wrapper function that will record the base and bounds for the memory being allocated before finally calling `malloc`.

In this example where we allocate an int array of size five, the base will be the address of `arr` and the bounds will be 39. The bounds is 39 because the array has five `int`s and each int has eight bytes, totalling to a size of 40 bytes. However, since the index starts at 0, the bounds is 39. When `free` is later called, the base and bounds information for the array will be removed from the bounds table. Finding the base and bounds is relatively straightforward in the case of the system allocator, but what happens when we add custom memory allocators to the equation? Consider the following example, where the application is using a pool allocator to manage its memory allocations. We define `pool_alloc` to be the malloc function of a generic pool allocator.

```
1 int *arr;
2 arr = (int*) pool_alloc(sizeof(int) * 5); // int array of size 5
```

Code Listing 6.2: Simple Malloc Using a Pool Allocator

In this case, assume the application would have asked the operating for a large piece of memory using system `malloc` beforehand making calls to `pool_alloc`. When the application needs memory, it will check if its custom memory allocator has memory available for use. If the custom memory allocator has memory available, the application will ask the custom memory allocator for memory rather than the operating system. However, when we use Intel MPX to try and enforce memory safety for our application with the pool allocator, Intel MPX is unaware of the fact that the application is managing its own memory allocations. Therefore, Intel MPX will only

enforce the base and bounds set by the system memory allocator rather than the custom memory allocator. Memory safety is only enforced on the large piece of memory given by the system to the allocator. The individual bits of memory allocated by the pool allocator through `pool_alloc` are not protected by Intel MPX. This means that if `arr` goes out of bounds in this example, Intel MPX will not catch the memory violation.

6.1 Partial and Complete Custom Allocator-Aware Memory Safety

6.1.1 Partial Custom Allocator-Aware Memory Safety

In this section, we introduce the concept of partial and complete custom allocator-aware memory safety. Partial custom allocator-aware memory safety is when the application is only aware of the base and bounds set for memory allocated by the system. This is the scenario where we compile an application using a custom memory allocator with Intel MPX. If the application ever attempts to use memory that is not allocated to it by the system, such as attempting to read or write to a memory address that does not belong to it, then Intel MPX will report a memory violation.

Even without Intel MPX to guard the base and bounds of memory allocated by the system allocator, the MMU (memory management unit) has ways to defend against similar situations. An application does not have information about the memory addresses of pages not given to it, so if an application attempts to read or write to memory outside of its process map, a page fault will occur. Therefore, despite achieving only **partial** custom allocator-aware memory safety, there are still benefits to using it.

Namely, the operating system can enforce base and bounds for every application and ensure that all of the applications are only using the memory given to them. If the developer only cares about achieving memory safety at the granularity of the application level, this is a good solution. Partial custom allocator-aware memory

safety is easy to implement in that it rarely requires manual modification of the code and in most cases, the developer will not have to do anything to achieve partial custom allocator-aware memory safety.

6.1.2 Complete Custom Allocator-Aware Memory Safety

If we want to achieve finer-grained memory safety, we need to use complete custom allocator-aware memory safety rather than just partial custom allocator-aware memory safety. As the name suggests, complete custom allocator-aware memory safety is when the application is aware of the base and bounds for every piece of memory allocated, even the pieces allocated by the custom memory allocator. In the code example above where a pool allocator is being used to manage memory for the application, it means knowing the base and bounds of the array allocated by `pool_alloc`.

By default (as we have learned), just compiling the application with something like Intel MPX will not give us that awareness in the application. These base and bounds must be explicitly set for the custom allocator. In the case of the system allocator, the developers of GCC had to devote some work into writing the previously mentioned wrapper functions for the libc memory management functions to allow awareness of the base and bounds of allocations performed by the system. Below, we display an actual example from the Intel MPX wrapper functions for libc.

```
1 #include "mpxrt/mpxrt.h"
2
3 void *
4 __mpx_wrapper_malloc (size_t size)
5 {
6   void *p = (void *)malloc (size);
7   if (!p) return __bnd_null_ptr_bounds (p);
8   return __bnd_set_ptr_bounds (p, size);
9 }
```

Code Listing 6.3: GCC's wrapper function for malloc

In the Intel MPX wrapper function for `malloc`, we need mark the base and bounds explicitly using `__bnd_set_ptr_bounds`, which is part of the API provided by Intel to enable Intel MPX at the software level. Every time the base and bounds

changes, such as when `realloc` (which resizes a previously allocated piece of memory) is called, then we need to mark the new base and bounds.

For applications with custom memory allocators, the way to achieve complete memory safety often requires the developer to manually modify the application's code. For example, using Intel MPX, the developer will need to modify the memory management functions in their application's custom memory allocator code.

The wrapper function might consist of only one to two additional lines of code. However, even such a small change might not be be feasible to do in a production setting (as we will soon see.)

6.2 Performance and Security Implications

In this section, we report our findings on the performance implications of enforcing memory safety protection on applications using a custom memory allocator. We used Intel MPX as our choice of defense to carry out the experiment. In order to achieve any sort of custom allocator-aware memory safety with Intel MPX, we must add an additional number of checks to the application to enforce the base and bounds of the memory used.

There are two obvious consequences from this action. The first is that the amount of space required will increase to store this new metadata. Space refers to the amount of memory used to store the base and bounds in the bounds table and bounds directory along with the registers added to support the extra Intel MPX operations. Secondly, we expected that there will be an increase in performance overhead due to the fact that we are now doing more work by setting and checking the base and bounds information. While we obtained data to backup the first observation, it was not as clear-cut for the second observation.

6.2.1 Experiment Methodology

We performed the experiment on three different applications with custom memory allocators. The first was the Apache HTTP server, an open-source project that

uses a pool allocator for memory management. The second was the nginx web server, another open-source project that uses a pool allocator. The third and final application we used was appweb, a proprietary web server that also uses a pool allocator. While all three applications use pool allocators, there were significant enough differences in their implementations that the performance varied differently depending on how strict the base and bounds were being enforced for the memory allocations being done. For a more detailed discussion on how the custom memory allocators work for Apache and nginx, please refer to chapter 4.

Since all three applications are implementations of web servers, the most important metric is the time it takes for the web server to respond to a client request. In our experiment, we measure the time it takes for the web server to answer a client's request, given changes to memory management and safety.

We used ApacheBench to measure the performance of four scenarios for all three applications. ApacheBench is a bench-marking tool that sends a number of user requests concurrently (this number is controlled by the tester) to a specified HTTP web page. In our experiment, we send a large amount of user requests to an HTTP web page that is being managed by the application web server and measure the amount of time it takes for the web server to respond to the user.

The size of the web page was consistent throughout the experiment. Originally, the HTTP web page we used was a small file that only consisted of the words "Hello World", but we found that having such a small file made it hard to interpret results. Therefore, we increased the size of the HTTP web page to 360kb.

The experiment was performed on a standard laptop running with an Intel Core i7 6700HQ processor (Skylake generation) @ 2.60GHz with 12GB RAM on Linux Ubuntu 16.04.4 LTS. Each of the three chosen applications were subjected to performance test under five different situations to gather data on how the usage of Intel MPX and custom memory allocators impacted performance.

The configurations are Custom Allocator/No Protection, Custom Allocator/-Partial Protection, Custom Allocator/Full Protection, System Allocator/No Protection, and System Allocator/Full Protection. "Custom Allocator/No Protection"

means that the application was compiled using GCC without Intel MPX enabled. No base and bounds information is generated about memory allocated from either the system allocator or the custom memory allocator. As this is the default way to compile the application (without Intel MPX and without modifications done to the application's source code), the data from this scenario served as the baseline for our later experiments.

"Custom Allocator/Partial Protection" refers to the partial custom memory-aware memory safety that we defined in the previous section. The application is compiled using GCC with Intel MPX enabled so we are aware of the base and bounds of the allocations performed by the system allocator. As the application code does not get modified, we do not know the base and bounds of the allocations performed by the custom memory allocator.

"Custom Allocator/Full Protection" refers to the complete custom memory-aware safety that we also defined in the previous section. The application code is modified, such that the memory management functions of the custom memory allocator make calls to the Intel MPX API. Then the application is compiled with Intel MPX. We know the base and bounds of all allocations done by both the system allocator and the custom memory allocator.

"System Allocator/No Protection" and "System Allocator/Full Protection" required huge modification of all three application's source code. It involved replacing the custom allocator with the system allocator. In other words, rather than call `pool_alloc`, we modified the applications to call `malloc` instead. When `pool_free` is called, `free` is called instead. The implication behind this is that the application gets the overhead of having to contact the system every time it wants to request or discard a piece of memory. In return, for using the system allocator, the application no longer has the overhead of managing its memory. In most of these cases, that means not having to manage a free list. In "System Allocator/No Protection", the application is using only the system allocator and not compiled with Intel MPX. In "System Allocator/Full Protection", the application is using only the system allocator and is compiled with Intel MPX. These two tests were added to further understand

the implications of using custom memory allocators. By choosing to use the system allocator only, we wanted to see if there would be a huge impact to performance when the application was compiled with and without Intel MPX.

6.2.2 Apache Results

Setup

We briefly describe the setup for Apache. We used Apache version 2.2.19. The configure commands to compile Apache with and without Intel MPX respectively are:

```
./configure --prefix="/usr/local/apache"
```

```
./configure --prefix="/usr/local/apachewithMPX" CFLAGS="-mmpx
-fcheck-pointer-bounds -lmpx" LDFLAGS="-lmpxwrappers -lmpx"
```

The "no protection" form of Apache is compiled regularly and without any additional options. For the versions compiled with Apache, we need to add additional information so that GCC knows to compile with Intel MPX. The libc libraries used were `libmpx.so.0` and `libmpxwrappers.so.0`. Setting the compiler flag `-fcheck-pointer-bounds` tells GCC to record and validate the base and bounds of allocated memory. For a comprehensive list of compiler options offered by Intel MPX, refer to the Intel Software Developer's Manual. Note that the lastest version of GCC has stopped support for Intel MPX. GCC 9 removes support for Intel MPX altogether.

Both figure 6-1 and 6-2 record the same data, but display the data differently. Each data point is based on the number of client HTTP get requests sent to the web server and the time it took for the web server to respond. The x-axis increment by 100, starting at 10 client requests and ending at 910 requests. Both nginx and appweb follow the same rules for data-gathering, so we will only explain this in the Apache section of results.

Based on the results in figure 6-1, "Custom Allocator/No Protection" Apache has the fastest run-time, followed by "System Allocator/No Protection" Apache,

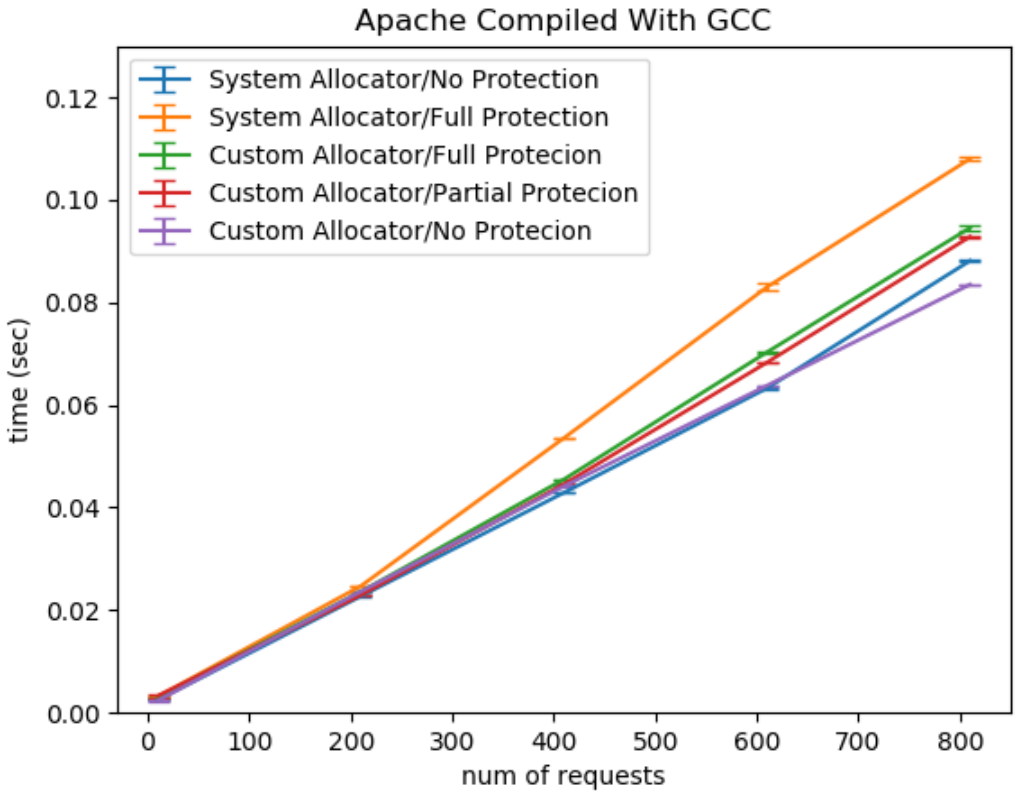

Figure 6-1: apache performance

"Custom Allocator/Partial Protection" Apache, "Custom Allocator/Full Protection" Apache, and finally "System Allocator/Full Protection" Apache. It seems intuitive for why "Custom Allocator/No Protection" Apache is the fastest out of all five scenarios. "Custom Allocator/No Protection" Apache does not have the overhead added by Intel MPX nor does it have to regularly talk to the system allocator for memory management."Custom Allocator/No Protection" Apache has faster results than "System Allocator/No Protection" Apache, which has the increased overhead of having to communicate with the system allocator for all memory allocations and frees. After "System Allocator/No Protection" Apache, the next fastest scenario for is "Custom Allocator/Partial Protection" Apache. This makes sense because the application is starting to become aware of the base and bounds for one of the allocations, but not all of them.

We noticed that for certain data points in 6-2, "System Allocator/No Pro-

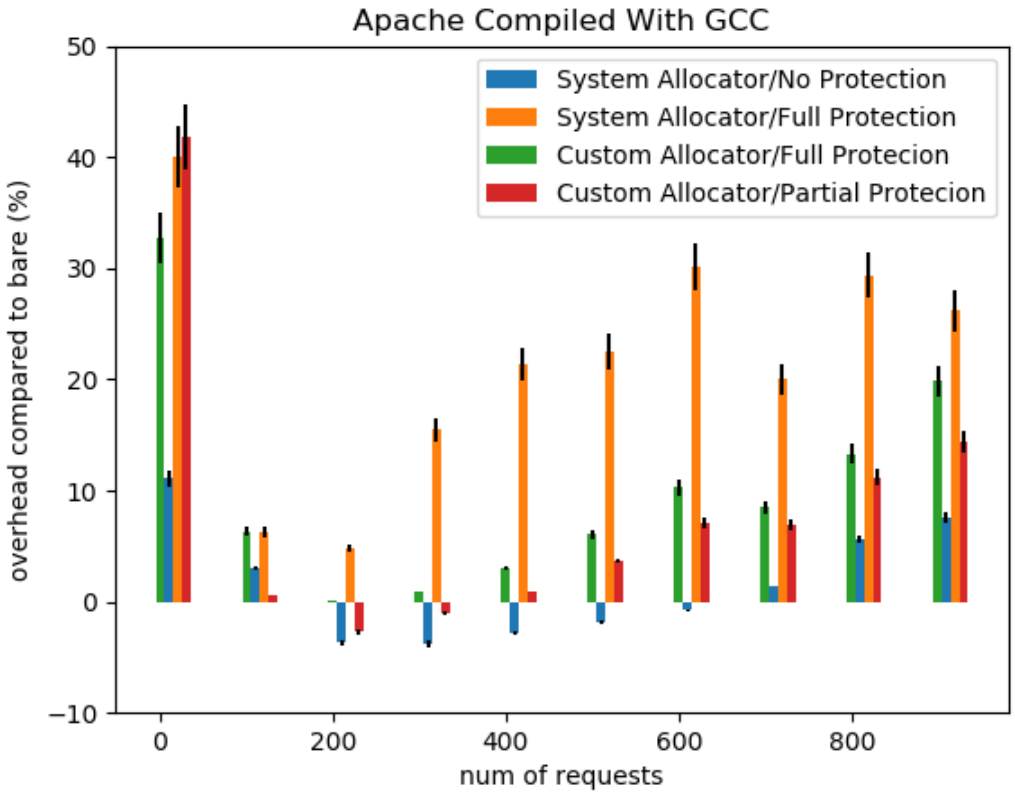

Figure 6-2: apache performance

tection" Apache and "Custom Allocator/Partial Protection" Apache performed *better* than "Custom Allocator/No Protection". Considering the observations we just made, it does not make sense that "System Allocator/No Protection" and "Custom Allocator/Partial Protection", with the additional overhead, would be able to beat "Custom Allocator/No Protection" Apache in performance. However, consider the case for "System Allocator/No Protection". Recall that the Apache pool allocator maintains a free list of all the memory blocks that was freed by the application previously. The free list is a linked list of lists, which contain memory blocks. As more and more allocations are done, the lists in the free list will grow longer and it will take time to iterate through these lists (as this is a linked list, and list traversals take linear time). When the memory used by the custom allocator is returned to the system, the custom allocator will have to remove all the references to the memory blocks in the linked lists. Depending on how and when these frees are being done, it

is possible to have an impact on performance. These were the reasons we attributed as to why "System Allocator/No Protection" was sometimes better than "Custom Allocator/No Protection".

In the points where "Custom Allocator/Partial Protection" Apache performed better than "Custom Allocator/No Protection" Apache, "Custom Allocator/Partial Protection" Apache's increase in performance was never more than 5%. This indicates that the performance gain might be caused by variance in the data results or there was indeed an advantage to compiling with Intel MPX when the number of requests is in the range of 200-400.

The next fastest scenario was "Custom Allocator/Full Protection" Apache, which makes sense based on the results we have seen so far. "Custom Allocator/Full Protection" Apache has to generate and record the base and bounds of all memory allocations, unlike the previous three results.

Finally, the slowest scenario was "System Allocator/Full Protection" Apache, which is not only using Intel MPX, but the system allocator for all allocations. The overhead of "System Allocator/Full Protection" Apache compared to the second slowest "Custom Allocator/Full Protection" Apache was significant in that the difference was greater than "Custom Allocator/Full Protection" and the third slowest, "Custom Allocator/Partial Protection". This reflects the fact that custom allocators do help in improving performance, if only slightly.

6.2.3 nginx Results

Setup

We briefly describe the setup for nginx. The version of nginx used was 1.12.2. The configure commands to compile nginx with and without Intel MPX respectively are:

```
./configure --prefix="/usr/nginx" --without-http_rewrite_module
./configure --prefix="/usr/nginxwithMPX" --with-cc-opt="-mmpx
-fcheck-pointer-bounds -lmpx" --with-ld-opt="-lmpxwrappers -lmpx"
--without-http_rewrite_module
```

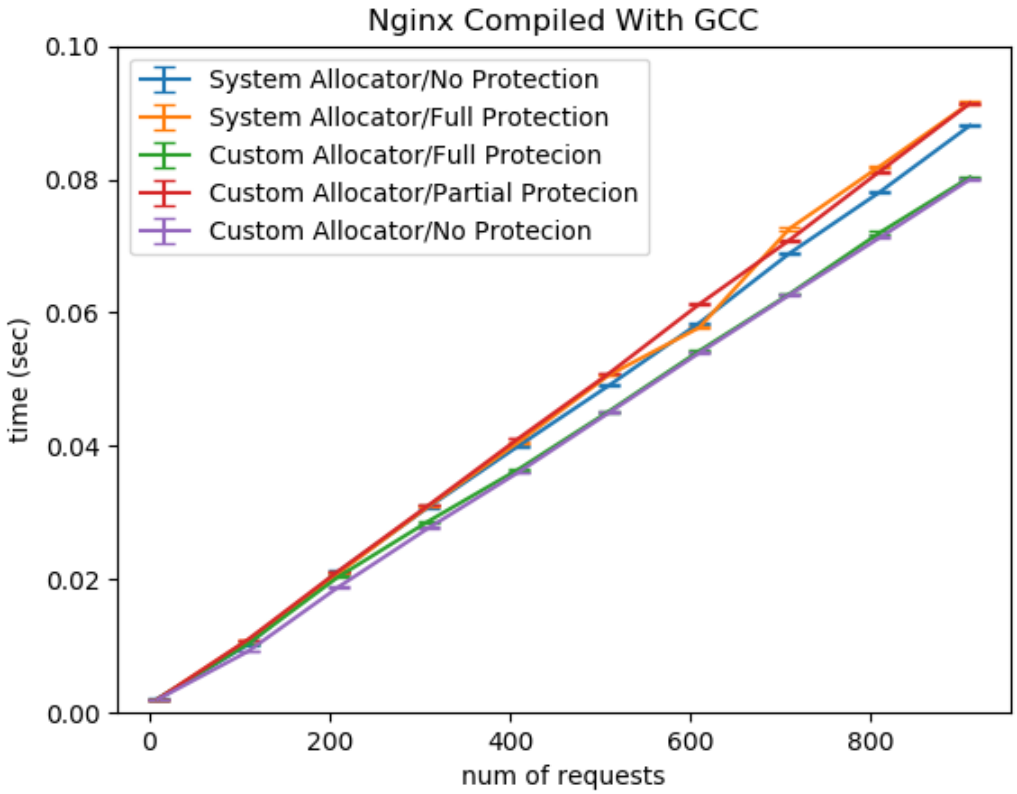

Figure 6-3: nginx performance

Due to the differences in implementation of pool allocators between Apache and nginx, the results for nginx are slightly different. "Custom Allocator/No Protection" nginx and "Custom Allocator/Partial Protection" nginx roughly have the same performance measurements. For Apache, there was a bigger difference between "Custom Allocator/No Protection" and "Custom Allocator/Partial Protection". The discrepancy is due to the fact that nginx's pool allocator tends to allocate either small or large blocks of memory from the system allocator. In both "Custom Allocator/No Protection" and "Custom Allocator/Partial Protection", allocations done by the system allocator are rare. The small blocks that do get allocated tend to get used for the client HTTP get requests, but are never reused by the custom allocator. When large blocks get freed, the blocks are returned to the system allocator and not added to the free list. Therefore, it makes sense that "Custom Allocator/No Protection" and "Custom Allocator/Partial Protection" exhibit the same behavior. The number

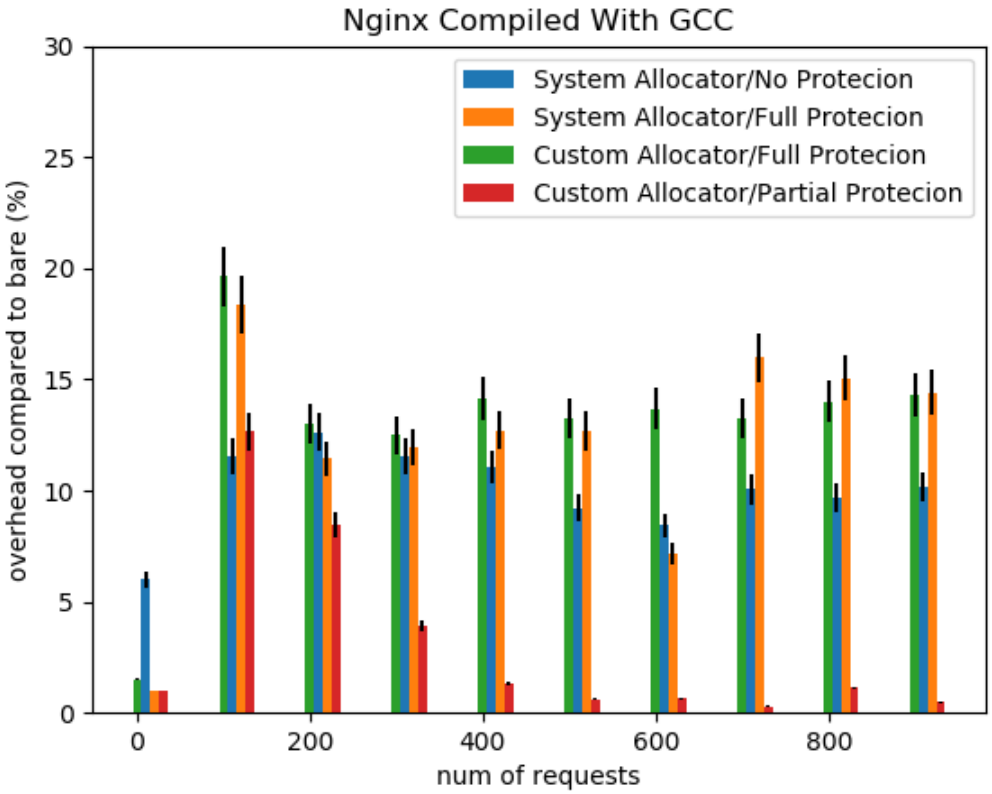

Figure 6-4: nginx performance

of calls to `malloc` is insignificant enough that enabling Intel MPX does not impact run-time.

There is a bigger overhead when we transition between "Custom Allocator/No Protection" and "System Allocator/No Protection." It shows that the nginx custom memory allocator `does` improve performance.

The slowest two are "Custom Allocator/Full Protection" nginx and "System Allocator/Full Protection" nginx. The two have similar run-times. This makes sense because of symmetry, given that "Custom Allocator/No Protection" nginx and "System Allocator/No Protection" nginx have similar run-times as well.

6.2.4 appweb Results

Setup

We briefly describe the setup for appweb. We used appweb 7.0.3. Note that appweb requires the use of an in-house "make" program if we want to specify compiler or linker flags. The version of appweb used was The configure commands to compile appweb with Intel MPX respectively is:

`CFLAGS="-mmpx -fcheck-pointer-bounds -lmpx"`
`LDFLAGS="-lmpxwrappers -lmpx -L/usr/lib/x86/`$_6 4 - linux - gnu - mmpx - fcheck-pointer-bounds"meconfigure--prefixbase = "/usr/local/fullappweb"--static$

For the compiler and linker flags to be recognized, both flags have to be before "me configure". Otherwise, the in-house "make" program will ignore the flags.

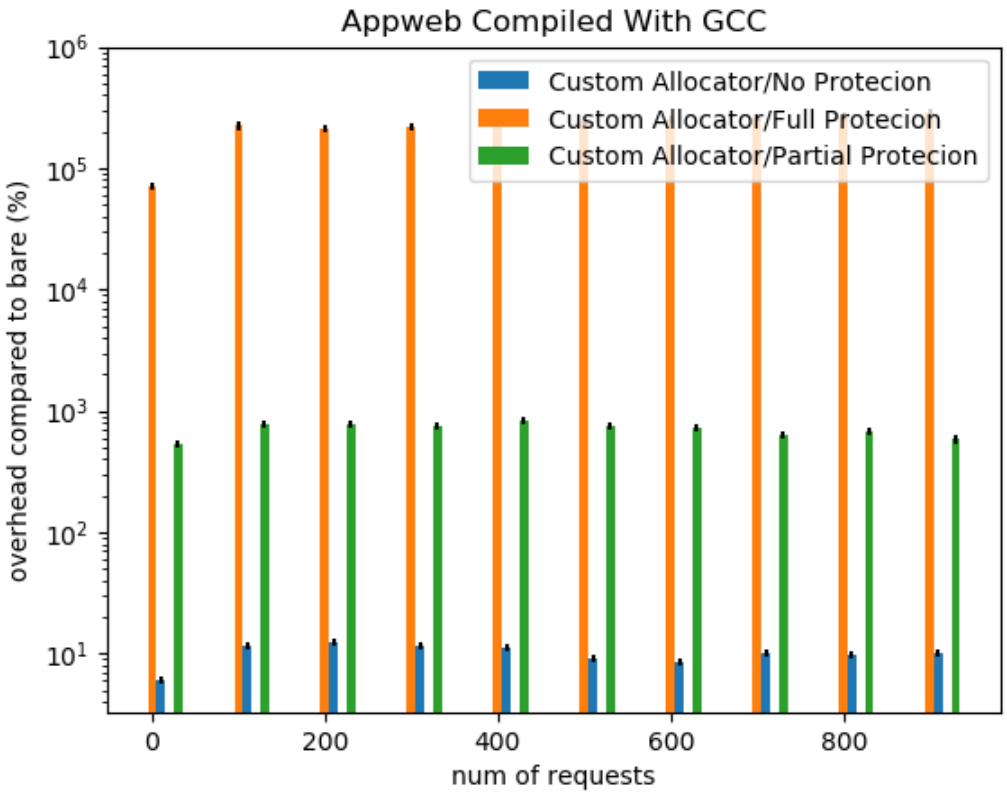

Figure 6-5: nginx performance

We only have a bar graph with a log y-axis for appweb (with three different scenarios), due to the huge performance overhead that prevent all the data from being displayed in a regular line graph. "Custom Allocator/No Protection" appweb has the best performance. The run-times get exponentially worse than the base. "Custom Allocator/Partial Protection" appweb is exponentially worse (almost 1000% times slower) and "Custom Allocator/Full Protection" is 10000% times slower. Considering even an 15% increase in performance overhead can often invalidate a security mechanism, even "Custom Allocator/Partial Protection" appweb is unusable. In certain cases like appweb, using Intel MPX can have devastating effects.

6.2.5 Analysis of Results

In all three applications, the changes caused by enabled Intel MPX varied depending on how strongly memory safety was enforced and what type of allocator was used. Therefore, even though custom allocators help by increasing performance, they tend to increase the complexity of the situation when memory protection is required. First, we have to understand the implementation of the custom allocator. For example, we need to know how memory is being managed by the custom memory allocator and the frequencies of memory requests the system and custom allocator get. We also need to know the latency caused by Intel MPX for all scenarios, such as "Custom Allocator/Partial Protection" and "Custom Allocator/Full Protection", but that knowledge would be incomplete without knowing the latency caused by Intel MPX when the application is using the system allocator only. With so many factors at hand, using something like Intel MPX with custom allocators becomes a much more complicated situation and we have to wonder if custom allocators are really all that beneficial.

Chapter 7

Conclusion and Future Work

Obtaining either good performance or security is difficult. Trying to achieve both is very difficult. We demonstrate this with custom memory allocators. We show that in most cases, efficiency and correctness are hard to achieve due to the complexity that comes with manual memory management. More specifically, we know that custom memory allocators are not the most practical of tools. Only a certain subset of custom memory allocators yield decent results. For the majority of custom memory allocators, there are many cases in which the general-purpose allocator or to be more specific, the Doug Lea memory allocator outperforms the custom memory allocator [9]. In certain cases, such as the pattern memory allocator, which relies on obtaining real-life data about the memory allocation patterns of a program, it may not always be possible. Even if it is possible to examine the program at depth, it may be hard to find any relevant patterns to exploit for performance.

In Chapter 4, we list a number of popular custom memory allocators and explain the advantages and disadvantages they provide to the programmer. Only the pool/region memory allocator proves to be advantageous in terms of memory leak prevention, minimized communication between the kernel and the process, and a decrease in run-time. We show that despite the advantages provided by the pool/region memory allocator, in hindsight, security considerations make those advantages more questionable. In Chapter 5, we show that there has been a history of exploits made on custom memory allocators, and that security is still very much a concern even if

memory becomes easier to manage.

The idea of custom memory allocators was created from the desire of programmers to improve performance and compatibility for specific programs. However, it is clear that while custom memory allocators have been shown to be good in dealing with certain bugs that are often common in the C programming language, such as memory leaks, custom memory allocators are bad in terms of safety. Rather than use custom memory allocators purely for performance, programmers should look to a more viable solution that does not undermine safety for performance. There have been custom memory allocators made that target safety, such as Cling. More efforts should be placed on these allocators.

In addition, ideas such as minimized communication between the kernel and process are not exclusive to the pool allocator. In general, it is good practice, especially if performance is an issue. The Doug Lea allocator also made attempts to minimize communication. The Doug Lea allocator was created with the goal to act as a general-purpose allocator. It does not try to improve performance for one specific type of program, but for all programs. It achieves this and outperforms much of the custom memory allocators. Therefore, while performance is still an important concern, perhaps there can be effort on fixing the C programming language such that memory leaks are harder to be created by the programmer.

By showing the trade-off between performance and security in custom memory allocators, we hope that this research will allow programmers to realize that more attempts should be made to find a better solution that does not undermine security.

Bibliography

[1] Emer J., Reliable Architectures, *MIT 6.823 Lecture*, MA, June 2015.

[2] Aleph One, A Smashing the stack for fun and profit, *Phrack*, 7(49), November 1996.

[3] Nagarakatte, S., Zhao J., Martin, M., Softbound: Highly compatible and complete spatial memory safety for C,. *SIGPLAN*, June 2009.

[4] "FTC Internet of Things Workshop," *Federal Trade Commission*, 19 November 2013.

[5] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, December 3 2013.

[6] CVE-2014-4115. Available from MITRE, CVE-ID CVE-2014-4115, June 12 2014.

[7] Intel MPX Support in the GCC Compiler. https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler, 2015. Accessed: 2015-07-06.

[8] Nagarakatte, S., Zhao J., Martin, M., CETS: compiler enforced temporal safety for C. *ISMM*, ACM, 2010.

[9] Berger, E., Zorn, B., McKinley, K., Reconsidering Custom Memory Allocation. *OOPSLA*, 2002.

[10] Apache Foundation. Apache Web Server. http//www.apache.org

[11] Free Software Foundation. GCC Home Page. http://gcc.gnu.org

[12] Doug Lea. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

[13] nginx community. nginx Web Server. nginx.org/en/

[14] Szekeres, L., Payer, M., Wei, T., Song, D., Sok: Eternal war in memory. *IEE Symposium on Security and Privacy*, 2013.

[15] OpenBSD. Openbsd 3.3, 2003.

[16] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P., On the expressiveness of return-into-libc attacks. *RAID*, 2011.

[17] Shacham, H., The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *ACM*, 2007.

[18] Soderstrom, E., Analysis of return oriented programming and countermeasures. Masters Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2014.

[19] Nagarakatte, S., Zdancewic, S., Martin, M., Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. *ISCA* 2012.

[20] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J., Region-Based Memory Management in Cyclone. In PLDI 2002

[21] Berger, E., Hertz, M., Automatic vs. Explicit Memory Management: Settling the Performance Debate. *OOPSLA*, 2004.

[22] Johnstone, M., Wilson, P., The memory fragmentation problem: Solved? In International Symposium on Memory Management, 1998.

[23] Kreinin, Y., Why Custom Memory allocators/pools are hard. http://yosefk.com/blog/why-custom-allocatorspools-are-hard.html

[24] Abadi, M., Budiu, M., Erlingsson, J., Ligatti, J., Control-flow integrity. In Proceedings of the 12th ACM conference on Computer and communications security, *ACM*, 2005.

[25] Eigler. F., Mudflap: Pointer Use Checking for C/C++. In GCC Developers Summit, 2003.

[26] Dhurjati, D., Adve, V., Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In Proceeding of the 28th International Conference on Software Engineering, May 2006.

[27] Austin, T., Breach, S., Sohi, G., Efficient Detection of All Pointer and Array Access Errors. In Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation, *SIGPLAN*, 1994.

[28] Regehr, J., Memory Safe C/C++: Time to Flip the Switch. http://blog.regehr.org/archives/939, 2013. Accessed: 2015-08-12.

[29] El-Sherei, S., Return-to-libc. https://www.exploit-db.com/docs/28553.pdf

[30] Kahneman, D., Knetsch, J., Thaler, R., Anomalies: The Endowment Effect, Loss Aversion, and Status Quo Bias. *The Journal of Economic Perspectives*, 1991.

[31] Liang, Z., Bletsch, T., Jiang, X., Free, V., Jump-Oriented Programming: A New Class of Code-Reuse Attack. *ASIACSS*, 2011.

[32] Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D., Hacking Blind. In *IEEE Symposium on Security and Privacy*, 2014.

[33] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M., Return-Oriented Programming without Returns . *CCS*, 2010

[34] PaX Team., PaX Non-executable Stack (NX): http://pax.grsecurity.net/docs/noexec.txt

[35] Schwartz, E., The Dangers of Unrandomized Code. *USENIX*, 2011.

[36] Andersen, S., Abella, V., Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies,

Data Execution Prevention, Microsoft TechNet Library, September 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[37] Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G., Out Of Control: Overcoming Control-Flow Integrity. *IEEE Symposium on Security and Privacy*, 2014.

[38] Davi, L., Sadeghi, A., Lehmann, D., Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. *USENIX*, 2014.

[39] Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., Negro, M., Qunaibit, M., Sadeghi, A. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. *CCS*, 2015.

[40] PaX ASLR Documentation. http://pax.grsecurity.net/docs/aslr.txt.

[41] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D., On the Effectiveness of Address-Space Randomization. *CCS*, 2004.

[42] Snow, K., Monrose, F., Davi, L., Dmitrienko, A., Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[43] Pappas, V., Polychronakis, M., Keromytis, A., Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In ACM, 2012.

[44] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A., Brunthaler, S., Franz, M., Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy*, 2015.

[45] Kc, G., Keromytis, A., Prevelakis, V., Countering Code-Injection Attacks With Instruction-Set Randomization. In *CCS*, 2003.

[46] Sovarel, A., Evans, D., Paul, N., Wheres the FEEB? The Effectiveness of Instruction Set Randomization. In *ACM*, 2005.

[47] Barrantes, G., Ackley, D., Palmer, T., Zovi, D., Forrest, S., Stefanovic, D., Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM*, 2005.

[48] Ganesh, K., Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses. In *Intel Parallel Universe Magazine*, 2012.

[49] Nagarakatte, S., Marin, M., Zdancewic, S., WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *CGO*, 2014.

[50] Intel Pointer Checker. https://software.intel.com/en-us/node/522702

[51] jemalloc. http://www.canonware.com/jemalloc/

[52] MaXX, Vudo - An object superstitiously believed to embody magical powers, *Phrack*, November 2001. Accessed: 2015-09-16.

[53] argp, huku, Pseudomonarchia jemallocum, *Phrack*, April 2012. Accessed: 2015-09-16.

[54] argp, huku, The Art of Exploitation: Exploiting VLC, a jemalloc case study, *Phrack*, April 2012. Accessed: 2015-09-16.

[55] Karmitas, C., Argyroudis, P., Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap, In *Blackhat*, 2012.

[56] Akritidis, P., Cling: A Memory Allocator to Mitigate Dangling Pointers. In *IEEE Symposium on Security and Privacy*, 2010.

[57] Oleksenko, O., Kuvaiskii, D., Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2018*

[58] Drake, J. Exploiting Memory Corruption Vulnerabilities in the Java Runtime. In *Black Hat Abu Dhab*, 2011.

[59] Hansen, D. Intel Memory Protection Extensions (Intel MPX) for Linux. 2016.