Dangless: Safe Dangling Pointer Errors

Gábor Kozár

June 29, 2019

Contents

Abstract

Manual memory management required in programming languages like C and C++ has its advantages, but comes at a cost in complexity, frequently leading to bugs and security vulnerabilities. One such example is temporal memory errors, whereby an object or memory region is accessed after it has been deallocated. The pointer through which this access occurs is said to be dangling.

Our solution, Dangless, protects against such bugs by ensuring that any references through dangling pointers are caught immediately. This is done by maintaining a unique virtual alias for each individual allocation. We do this efficiently by running the process in a light-weight virtual environment, where the allocator can directly modify the page tables.

We have evaluated performance on the SPEC2006 benchmarking suite, and on a subset of the benchmarks have found a geometric mean of 3.5% runtime performance overhead and 406% memory overhead. This makes this solution very efficient in performance - comparable to other state-of-the-art solutions - but the high memory overhead limits its usability in practice.

Chapter 1

Introduction

TODO: Maybe reorg: Motivation to go into Background instead?

1.1 Motivation

Developing software is difficult. Developing software that is bug-free is all but impossible. Programming languages like C and C++ (the so-called "low-level" programming languages [1]) offer a great deal of control to the programmer, allowing them to write small and efficient computer programs. However, they also require the programmer to take a great deal of care with their development: the same level of control that allows extremely efficient software to be built also places a large burden on the developer, as making mistakes has steeper consequences than in high-level, safer programming languages (such as Java or C#).

Typically, low-level programming languages require the programmers to manage memory manually, while higher-level languages generally include a garbage collector (GC) that frees the programmer from this burden. Managing memory manually means that objects whose lifetime is dynamic (not tied to a particular program scope) have to be *allocated* as well as *deallocated* (freed) explicitly.

In C, such memory allocation typically occurs using the `malloc()` or `calloc()` functions. These allocate a region inside the *heap memory* of the application, reserving it for use, and returning a pointer (typically, an

[1] "low-level" is traditionally used to indicate that the level of abstraction used by these programming languages is relatively close to that of the hardware

untyped `void *`) to it. On the x86 and x86-64 architectures, which we will mainly concern ourselves with in this thesis, a pointer is just a linear memory address: a number representing the index of the first byte of the pointed region in the main memory. This makes pointer arithmetic, such as accessing `numbers[4]` in a `int *numbers` very easy and efficient to perform: just load `sizeof(int)` bytes from the memory address `(uintptr_t)numbers + 4 * sizeof(int)`. Conversely, after we are done with using a given memory region, we can and should deallocate it using the `free()` function. This marks the memory region as no longer in use, and potentially reusable – a characteristic that forms the basis of this thesis.

In C++, memory allocation typically happens using the `new` or `new[]` operators, and deallocation using the `delete` or `delete[]` operators. However, these behave exactly like C's `malloc()` and `free()` in all ways that are important from a memory management point of view. I should note that in modern C++, the use of such memory management is discouraged and generally unnecessary since *smart pointers* – wrappers around pointers that automate the lifecycle management of the pointed memory region, conceptually similarly to a garbage collector – were introduced. However, a lot of applications are still being developed and maintained that do not make use of such features.

Making mistakes with manual memory management is very easy. Allocating but not freeing a memory region even after it's no longer used is called a *memory leak* and it increases the memory usage of the application, often in an unbounded manner, potentially until a crash occurs due to insufficient memory. Attempting to deallocate an object twice – *double free* – causes the memory allocator to attempt to access accounting data stored typically alongside the user data in the since-freed region, often leading to seemingly nonsensical behaviour or a crash, given that the region may have been re-used. Accessing an offset that falls outside the memory region reserved for the object – *out-of-bounds access*, sometimes also called *buffer overflow* or *buffer underflow* – can lead to reading unexpected data or overwriting an unrelated object, again often causing hard-to-understand bugs and crashes. One example for this would be attempting to write to `numbers[5]` when only enough space to hold 5 elements was allocated, e.g. using `int *numbers = malloc(5 * sizeof(int))` (recall that in C and related languages, indexes start at 0, so the first item is located at index 0, the second at index 1, and so on). Finally, accessing a memory region that has been deallocated – *use after free* – is similarly problematic. This

generally occurs when a pointer is not cleaned up along with the object it referenced, leaving it dangling; often also called a *dangling pointer*.

Besides the instability and general mayhem that such memory errors routinely cause, they can also leave the application vulnerable to attacks, for instance by enabling unchecked reads or writes to sensitive data, sometimes even allowing an attacker to hijack the control flow of the application, execute almost arbitrary instructions, and in essence take over the application.

It should be clear by now why modern, "high-level" programming languages restrict or completely prohibit the direct use of pointers, often by making it impossible and unnecessary to manage memory manually. In such languages, the programmer can perform allocations only by creating objects (preventing another class of bugs relating to the use of uninitialized memory), and leaving their deallocation to the runtime environment, commonly its component called the garbage collector (GC). The GC will periodically analyse the memory of the application, and upon finding objects that are no longer referenced, marks them for reclaiming, and eventually deallocating them automatically. This, of course, comes at a cost in performance, often one that's unpredictable as the GC is controlled by the runtime environment as opposed to the user code. (It's worth noting that this scheme doesn't protect against all possibilities of memory errors; for instance, leaks are still both possible and common.)

A notable exception is the Rust programming language, which, while does allow pointers, heavily restricts how they can be used, preventing any code that could potentially be unsafe. It does so using static (compile-time) checking using their so-called *borrow checker*. However, realizing that in doing so it also disallows some valid uses of pointers, it also provides an escape hatch, allowing code sections to be marked as *unsafe* and go unchecked. (For example, it's not possible to implement a linked list in safe Rust, and even the built-in `vec` type is written using unsafe code.) Another programming language that follows a similar pattern is C#: normally used as a high-level, managed language employing a GC, it also allows pointers to be used directly in code marked as unsafe [2].

Still, applications written in languages like C or (older) C++ with no

[2]Usage of raw pointers in an otherwise managed environment comes with caveats; for instance, the memory is often compacted after GC passes with the surviving objects moved next to each other to reduce fragmentation. Such relocation is not possible if there are raw pointers in play; therefore, programmers are required to mark the pointers they use as *pinned* using the `fixed()` construct.

safe alternatives to pointers have been written and are being maintained, and these applications remain affected by memory errors. Significant amount of research has been and continues to be conducted in this topic, as such applications are often high-value targets for attackers: operating system kernels, device drivers, web servers, anti-virus programs are commonly developed using these technologies.

This thesis is focused specifically on dangling pointer errors, a class of memory issues defending against which has traditionally been difficult and inefficient. **TODO: structure? here or later?**

1.2 Dangling pointers

A *dangling pointer* is a pointer which outlives the memory region it references. Subsequent accesses to the pointer usually lead to unwanted, confusing behaviour.

In the very best-case scenario, the memory access fails, and the application is killed by the operating system kernel; for example on Unix systems by sending it a signal like `SIGSEGV`, leading to the well-known "Segmentation fault" error and a groan from the programmer. This is useful (and often highly underrated by programmers), because it clearly indicates a bug, and the responsible memory address is readily available, greatly helping with debugging.

Unfortunately, in the majority of cases in practice, the memory access will not fail. The reason for this is that most modern architectures in widespread use (such as x86 and ARM) handle memory on the granularity of *pages*, where a single page is usually 4096 bytes (4 kilobytes). From the point of view of the hardware, and so the kernel, a page is either in use or is not; pages are treated as a unit and are never split up. Of course, typical memory allocations tend to be significantly smaller than this, and it would be wasteful to dedicate an entire page of memory to just hold for instance 200 bytes of user data.

Therefore, all memory allocator implementations used in practice do split up pages, and will readily place two allocations on the same page, typically even directly next to each other (not counting any meta-data). After the deallocation of one of the objects, the page as a whole still remains in use, and so the hardware will not fault on subsequent accesses to it, regardless of the offset; see Figure 1.1. Notably, even if a memory page holds no live

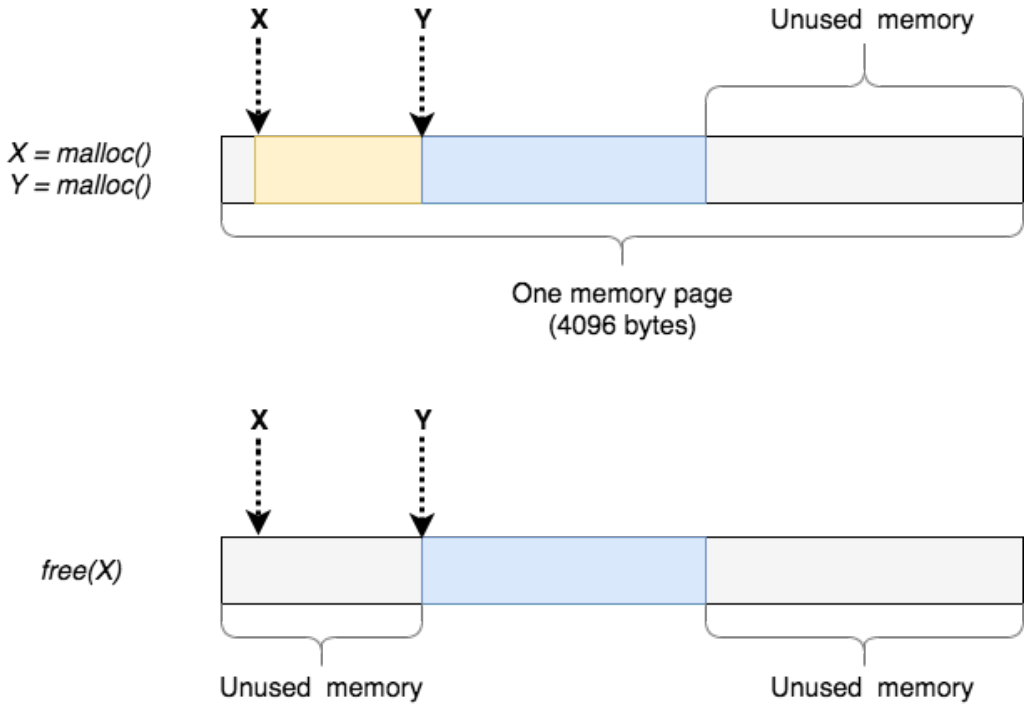

Figure 1.1: Memory layout of two small allocations. X and Y are pointers, referencing their corresponding memory regions. X becomes dangling

objects, it's often still not returned to the system; the memory allocator retains it as an optimization, expecting more allocations in the future. (This is because the memory allocators being discussed run in user-space, so in order to gain access to memory pages to use, they have to perform a system call such as `mmap()` or `brk()`, which is costly.)

If a page is known to be unused by the hardware and kernel, then accessing it will trigger a page fault in the kernel, which will generally terminate the application, leading to the best-case scenario described earlier. This is a far more manageable problem than the alternative, because the error is clear, even if in practice it's often difficult to discover the underlying reason, given that time may have passed between the deallocation and attempted access, and so the code executing at the time of access may not have any relation to the code that was responsible for the deallocation. Furthermore, this scenario doesn't generally pose a security problem, as a crashed application is difficult to exploit. Therefore, I will generally ignore this scenario for the remainder

of this thesis.

The effect of an unchecked access through a dangling pointer depends on whether or not the referenced memory region has been reused since the time of deallocation. If it hasn't, the data read often will still be valid, and execution may continue without anyone the wiser, masking the bug until a modification in the code or one of the libraries leads to a change in the memory allocation pattern. Otherwise, the data read or overwritten will almost always be a source of unexpected behaviour. One typical case is *type confusion*: the value read or written will be treated as a different type than the value stored there. A string value can be for instance accessed as an integer, causing the characters that make up the string to be interpreted as bytes in an integer, essentially leading to the same behaviour as this code snippet:

```
1 char *s = "foobar";
2 int i = *(int *)s;
```

This code compiles and runs successfully. What will be the value of `i`? Of course, we are deep into undefined behaviour territory here, meaning that the programming language promises nothing. In practice, on x86-64 architectures where the `int` C type is 4 bytes long (`sizeof(int)== 4`), the result will typically be `1651470182`, or `0x626f6f66` in hexadecimal. This makes sense: the string `"foobar"` (including the null terminator) is represented by the byte sequence `0x66 0x6f 0x6f 0x62 0x61 0x72 0x00`. Interpreting it as an `int` means reading the first 4 bytes (`0x66 0x6f 0x6f 0x62`) and assembling it into a multi-byte integer according to the endianness of the processor. My laptop has an Intel CPU in it, which is little endian, meaning that the bytes of an integral type are stored as least significant byte first (this is `0x62`), followed by bytes of increasing significance; simply put, bytes are interpreted in "reverse order".

Of course, type confusion doesn't have to occur in order for invalid behaviour to occur. For instance, overwriting an Unix file descriptor with the number of characters in a text will typically result in an invalid file descriptor; or consider a buffer's length overwritten by the age of the user; or an integer representing the next free index in an array overwritten by the length of a file in bytes. Once memory corruption occurs, sanity flees.

1.3 Overview

TODO: Better title? Dangless is a drop-in replacement memory allocator that provides a custom implementation of the standard C memory management functions `malloc()`, `calloc()`, `realloc()`, `free()` and a few others. It aims to solve the problems that dangling pointers lead to by guaranteeing that any access through a dangling pointer will fail, and the application will terminate. It relies on the underlying memory allocator to perform the actual allocation and deallocation. In principle, because Dangless makes no assumptions on the nature or behaviour of the underlying allocator – referred to as "system allocator" by Dangless –, it should be usable on top of even non-standard implementations such as Google tcmalloc.

Catching dangling pointer accesses is ensured by *permanently* marking memory regions as no longer in use upon deallocation (e.g. a call to `free()`). Therefore, during the lifetime of the application, in principle, no other memory will be allocated in such a way that it would visibly alias a previously used location. Of course, the physical memory available is very limited even on modern systems, and not re-using is hopeless. The trick then, is to leave the management of physical memory to the system allocator, and change how the physical allocations are mapped to virtual memory: the address space that user applications interact with. Virtual memory is plentiful: on the x86-64 architecture, pointers are 64-bit long, which in theory means 2^{64} bytes of addressable memory. In practice however, on all current processors that use this architecture, only 48 bits are used, which limits the size of the address space we can work with to 2^{48} bytes, or 256 terrabytes. That's also not unlimited, but as it turns out, in practice it almost is.

Normally, the difference between physical and virtual memory is entirely hidden from the user code, and is dealt with only by the operating system kernel. This allows the overwhelming majority of users and developers - even programmers working with lower-level languages such as C++ - to work and develop software without ever being aware of the difference, while enjoying the benefits of it. The Linux kernel does provide some system calls that allow the virtual memory to be manipulated, notably `mprotect()` which is used to manage access permissions (readable, writeable, executable) of memory regions. This is useful for example when developing just-in-time (JIT) compilers such as the ones employed by browsers to run JavaScript code. Another example is `mremap()`, which allows a memory region to be moved almost for free, an ability that makes it useful for garbage collectors for instance. Of

course, `mmap()` and `munmap()` also primarily work by manipulating virtual memory mappings.

These system calls are sufficient to implement the functionality of Dangless, with some caveats – in fact, this is exactly how Oscar operates **TODO: reference**. The biggest issue is that of performance: system calls are expensive compared to normal memory allocations, and in this scheme, for every single memory allocation, at least one extra system call would be required. The costs and possibilities for optimizations are explored in depth by the Oscar paper.

The solution Dangless uses is a technology called Dune **TODO: reference**: a Linux kernel module and library that provides a lightweight virtualization layer based on Intel VT-x. Using Dune, a normal Linux application can choose to enter a virtualized environment where it has ring-0 privileges, allowing it to efficiently and directly manipulate virtual memory mappings and the interrupt descriptor table, while retaining the ability to perform system calls on the host kernel using the `vmcall` instruction. In principle, an application running in Dune mode has the best of both worlds; normal Linux libraries and executables are able to run in Dune mode without any modifications, while gaining access to ring-0 features when beneficial. While there is an overhead associated with running in a virtualized environment, especially when performing `vmcall`s, in practice this turns out to be negligible for most applications.

Chapter 2

Background

2.1 Virtual memory

Virtual memory is an abstraction over the physical memory available to the hardware. It's an abstraction that is typically transparent to both the applications and developers, meaning that they do not have to be aware of it, while enjoying the significant benefits. This is enabled by the hardware and operating system kernel working together in the background.

From a security and stability point of view, the biggest benefit that virtual memory provides is address space isolation: each process executes as if it was the only one running, with all of the memory visible to it belonging either to itself or the kernel. This means that a malicious or misbehaving application cannot directly access the memory of any other process, to either deliberately or due to a programming error expose secrets of the other application (such as passwords or private keys) or destabilize it by corrupting its memory.

An additional security feature is the ability to specify permission flags on individual memory pages: they can be independently made readable, writeable, and executable. For instance, all memory containing application data can be marked as readable and writeable, but not executable, while the memory pages hosting the application code can be made readable and executable, but not writeable, severely limiting the capabilities of attackers.

Furthermore, virtual memory allows the kernel to optimize physical memory usage by:

- Compressing or swapping out (writing to hard disk) rarely used memory pages (regions) to reduce memory usage

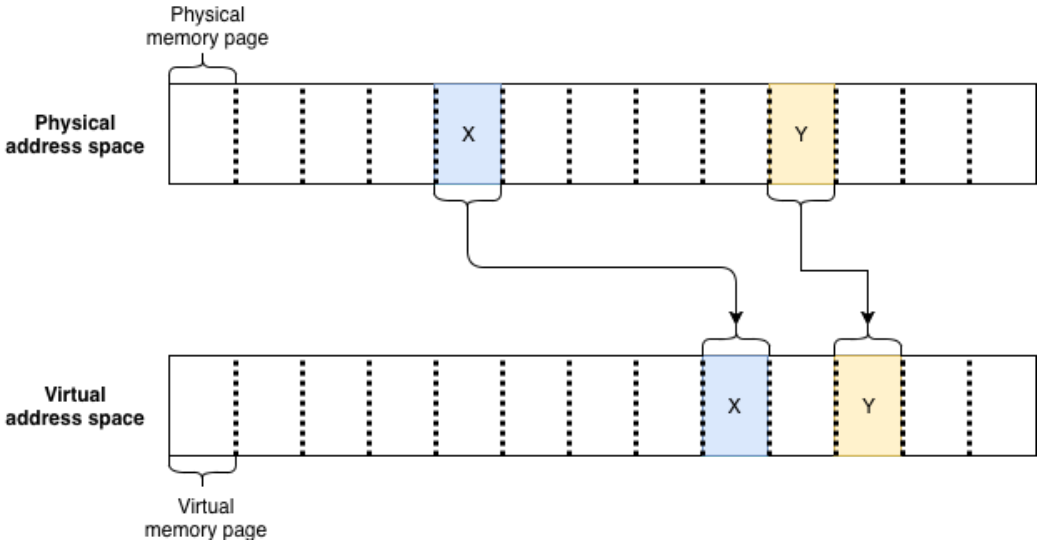

Figure 2.1: Mapping two physical memory pages X and Y to virtual memory

- Deduplicating identical memory pages, such as those resulting from commonly used static or shared libraries
- Lazily allocating memory pages requested by the application

Virtual memory works by creating an artificial (virtual) address space for each process, and then mapping the appropriate regions of it to the backing physical memory. A pointer will reference a location in virtual memory, and upon access, is resolved (typically by the hardware) into a physical memory address. The granularity of the mapping is referred to as a memory page, and is typically 4096 bytes (4 kilobytes) in size. (See Figure 2.1)

This mapping is encoded in a data structure called the *page table*. This is built up and managed by the kernel: as the application allocates and frees memory, virtual memory mappings have be created and destroyed. The representation of the page table varies depending on the architecture, but on x86-64, it can be represented as a tree, with each node an array of 512 page table entries of 8 bytes each making up a 4096 byte page table page. The root of this tree is where all virtual memory address resolution begins, and it identifies the address space. The leaf nodes are the physical memory pages that contain the application's own data. The bits of the virtual memory address identify the page table entry to follow during address resolution. For

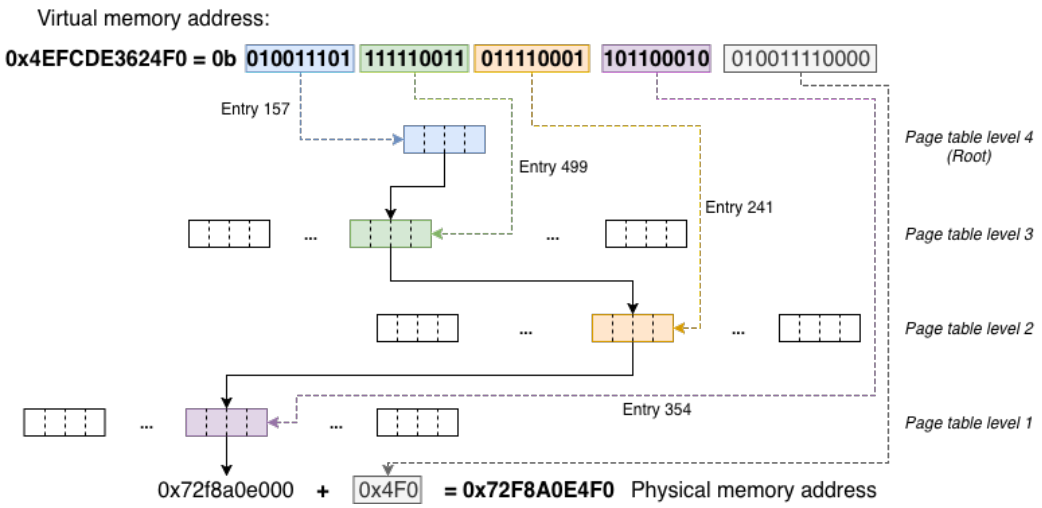

Figure 2.2: Translating a virtual memory address to physical using the page tables

each level of page tables, 9 bits are required to encode an index into the array of 512 entries. Each entry contains the physical memory address of the next page to traverse during the address resolution. Finally, the least-significant 12 bits are used to address into the application's physical page (which is 4096 bytes) itself and so require no translation. (See Figure 2.2)

On x86-64, there are currently 4 levels of page tables, using $4*9+12 = 48$ out of the 64 available bits in the memory addresses, and limiting the size of the address space to 2^{48} bytes or 256 terabytes. (The size of addressable space per page table level, in reverse resolution order being: $512 * 4$ kilobytes $= 2$ megabytes; $512 * 2$ megabytes $= 1$ gigabyte; 512 gigabytes; 256 terabytes.)

Note that it is possible to map a physical page into multiple virtual pages, and it is also possible to have unmapped virtual pages. Attempting to access a virtual page that is not mapped (i.e. not backed by a physical memory page) will cause execution to trap inside the kernel, which will then usually interpret it as segmentation fault and kill the process. (This is also how lazy memory allocation and swapping out works: instead of killing the process, the kernel goes ahead and allocates the physical memory it promised, or swaps the page back in from disk.)

Thanks to Dune, Dangless has direct access to the page tables, and manipulates them using custom functions (see *include/dangless/virtmem.h* and

src/virtmem.c). Upon memory allocation, Dangless forwards the allocation call to the system allocator and then reserves a new virtual page, mapping it to the same physical address as the a virtual page returned by the system allocator. This is called virtual aliasing. During deallocation, besides forwarding the call to the system allocator, Dangless unmaps the virtual alias page and overwrites the page table entry with a custom value, enabling it to recognize a would-be access through a dangling pointer.

2.2 Prior art

TODO: Oscar, etc.

Chapter 3

Dangless - Implementation

3.1 API overview

Dangless is a Linux static library that can be linked to any application during build time. It defines a set of functions for allocating and deallocating memory:

```
1 // sources/include/dangless/dangless_malloc.h
2
3 void *dangless_malloc(size_t sz) ↩
     __attribute__((malloc));
4 void *dangless_calloc(size_t num, size_t size) ↩
     __attribute__((malloc));
5 void *dangless_realloc(void *p, size_t new_size);
6 int dangless_posix_memalign(void **pp, size_t align, ↩
     size_t size);
7 void dangless_free(void *p);
```

These functions have the exact same signature and behaviour as their standard counterparts `malloc()`, `calloc()`, and `free()`. In fact, because the GNU C Library defines these standard functions as weak symbols [2], Dangless provides an option (`CONFIG_OVERRIDE_SYMBOLS`) to override the symbols with its own implementation, enabling the user code to perform memory management without even being aware that it's using Dangless in the background.

Besides the above functions, Dangless defines a few more functions, out of which the following two are important.

```
1 void dangless_init(void);
```

First, `dangless_init()` initializes Dangless, and has to be called before any memory management is performed that Dangless should protect. The most important thing that this function does is initialize and enter Dune by calling `dune_init()` and `dune_enter()`. Dangless relies on Dune to be able to manipulate the page tables. Afterwards, we register our own pagefault handler with Dune, which enables us to detect when a memory access has failed due to the protection that Dangless offers.

By default, Dangless automatically registers this function in the `.preinit_array` section of the binary, causing it to be called automatically before any user-defined constructors or the `main()` entry point. This can be disabled via the `CONFIG_REGISTER_PREINIT` option.

It's important to note that heap memory allocation can and does happen *before* `dangless_init()` is called, for example as part of the glibc runtime initialization. This case needs to be handled, so all of the `dangless_` functions simply pass the call through to the underlying (system) allocator without doing anything else if they are called before `dangless_init()`.

```
1 int dangless_dedicate_vmem(void *start, void *end);
```

In order for Dangless to work, it requires exclusive use of some virtual memory to remap user allocations into. This region has to be large, as each `dangless_malloc()` call will use up at least one page from it, and currently this virtual memory is never re-used because we lack a mechanism (such as a garbage collector) to be reasonably certain that a given virtual memory page is no longer referenced. This function can be used to make virtual memory regions available to Dangless for this purpose.

Since users of Dangless will typically not know or want to make this decision themselves, we provide the option `CONFIG_AUTO_DEDICATE_MAX_PML4ES` which allows Dangless to take ownership of one or more unused PML4 pagetable entries which can each map 512 gigabytes of memory. This occurs at most once when a `dangless_malloc()` or similar call is made, but Dangless does not have sufficient virtual memory available to it to protect the call.

This solution is very simplistic, and a smarter way to take ownership of virtual memory is decidedly possible. For instance, any time we require more virtual memory, we could scan the page tables and take ownership of some amount of currently-unused page table entries. Some implementation effort would need to be made to make sure this doesn't conflict with Dune's virtual memory allocation. However, this is very easy in the used Dune

version, as Dune's page allocator (as defined in `libdune/dune.h`) uses maximum `MAX_PAGES = (1 << 20)` pages (i.e. 4 GB of memory) starting at `PAGEBASE = 0x200000000`. Any memory outside of this that is not used to hold application or kernel code or data is available for use by Dangless uncontested.

3.2 Allocating physical memory

Whenever Dangless is asked to allocate some memory via a call to `dangless_malloc()`, `dangless_calloc()`, or `dangless_realloc()`, it has to first acquire some physical memory it can use to satisfy the allocation. (It does not currently defer allocating physical memory like kernels typically do, although in principle it could.) Since the goal of Dangless is only to provide security benefits, Dangless has no strategy of physical memory management unlike normal implementations. In fact, the way this is done ultimately does not matter for Dangless' purposes. Due to these reasons, Dangless delegates the responsibility of actually performing (physical) memory allocation to the memory allocator that was in place before Dangless "hijacked" the memory management function symbols.

Specifically, it uses `dlsym(RTLD_NEXT, "malloc")` to determine the address of the original `malloc()`, etc. functions. Then it simply calls these functions whenever it needs physical memory allocation done: primarily when the user code requests an allocation, but sometimes also for internal purposes, such as for keeping track of available virtual memory regions.

There is a caveat to using `dlsym()`: when `dlsym()` it is first called on a thread, it allocates a thread-specific buffer for holding a `struct dl_action_result` object using `calloc()` [1]. This means that without special handling for this case, execution can easily get into an infinite loop:

1. User calls `malloc()`, which is a strong alias of `dangless_malloc()`
2. `dangless_malloc()` defers the physical memory allocation to the underlying allocator by calling `sysmalloc()`
3. `sysmalloc()` does not yet have the address of the original `malloc()` function, so it calls `dlsym()` to get it
4. `dlsym()` notices it's running on this thread for the first time, so it calls `calloc()` to allocate a buffer

5. `calloc()` is a strong alias of `dangless_calloc()`, which calls `syscalloc()` to allocate physical memory

6. `syscalloc()` does not yet have the address of the original `calloc()` function, so it calls `dlsym()`

7. Repeat steps 4-6 forever...

To get around this, `syscalloc()` uses a static buffer of `CONFIG_CALLOC_SPECIAL_BUFSIZE` size for the very first allocation. This allows `dlsym()` to complete and populate the addresses of the original allocation functions, which are used normally for all subsequent calls. The same approach was used by other projects that implement their own memory allocator replacements [3].

Finally, when `sysmalloc()`, etc. returns, we have a completed physical memory allocation. However, what is returned to us is a virtual memory address. This is translated to a physical memory address by simple arithmetic provided by the `dune_va_to_pa()` (from `libdune/dune.h`), made possible by Dune's predictable implementation of the `mmap()` system call.

As a limitation, the current implementation of Dangless cannot handle the system allocator returning a virtual memory region that is backed by non-contiguous physical memory. **TODO: Is this a big deal? We could fix it**

3.3 Allocating virtual memory

Given a physical memory address of the user allocation, Dangless needs to allocate the same amount of virtual memory pages from the regions dedicated to it. Furthermore, we need to guarantee that these virtual memory addresses are only be used for exactly one allocation, and are never reused. For this purpose, Dangless employs a simple freelist-based span allocator. A freelist is simply a linked-list of `vp_span` objects, where each object represents a free span of virtual memory. When virtual memory is needed, the freelist is walked until a `vp_span` object representing a region of sufficient size is found. When one is found, the allocated space is removed from the beginning of the span, and the span is deleted if it is now empty. The beginning address of the span becomes the allocation result. If no such span is found, the allocation fails.

Note that in the current, simple implementation of the virtual memory allocator there is only a single freelist, which is sufficient because we do not ever re-use any virtual memory. If we were to add a garbage collector-like solution, then this approach would likely lead to significant fragmentation with a negative performance impact on each allocation. In this situation, a possible enhancement would be to have several independent freelists of different page sizes, similar to common memory allocator designs. Other improvements are also possible: memory allocation is a well-studied problem.

3.4 Remapping

TODO: Integrate this: If no such span is found, the allocation fails. I have already talked about Dangless' ability to automatically acquire virtual memory for its allocator by scanning the page table and taking unused PML4 entries for its own use. If even despite this Dangless does not have sufficient virtual memory to protect the user allocation by remapping it, then Dangless gives up. If the `CONFIG_ALLOW_SYSMALLOC_FALLBACK` option is enabled, then Dangless simply forwards all later memory management function calls to the system allocator, to ensure that the user application keeps functioning. Otherwise, it exits the application.

Bibliography

[1] GNU C library source - dlsym() calls calloc(). https://github.com/lattera/glibc/blob/59ba27a63ada3f46b71ec99a314dfac5a38ad6d2/dlfcn/dlerror.c#L141. Accessed 2019-06-29.

[2] GNU C library source - weak definitions of memory management functions. https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#__calloc. Accessed 2019-06-29.

[3] The universal elite game trainer for cli (linux game trainer research project) - hooking memory allocation functions. https://github.com/ugtrain/ugtrain/blob/d9519a15f4363cee48bb3c270f6050181c5aa305/src/linuxhooking/memhooks.c#L153. Accessed 2019-06-29.