Dangless: Safe Dangling Pointer Errors

Gábor Kozár

July 24, 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

1.1 Memory errors

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 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

[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

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 safe alternatives to pointers have been written and are being maintained, and these applications remain affected by memory errors. Significant amount of

[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.

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.

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 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

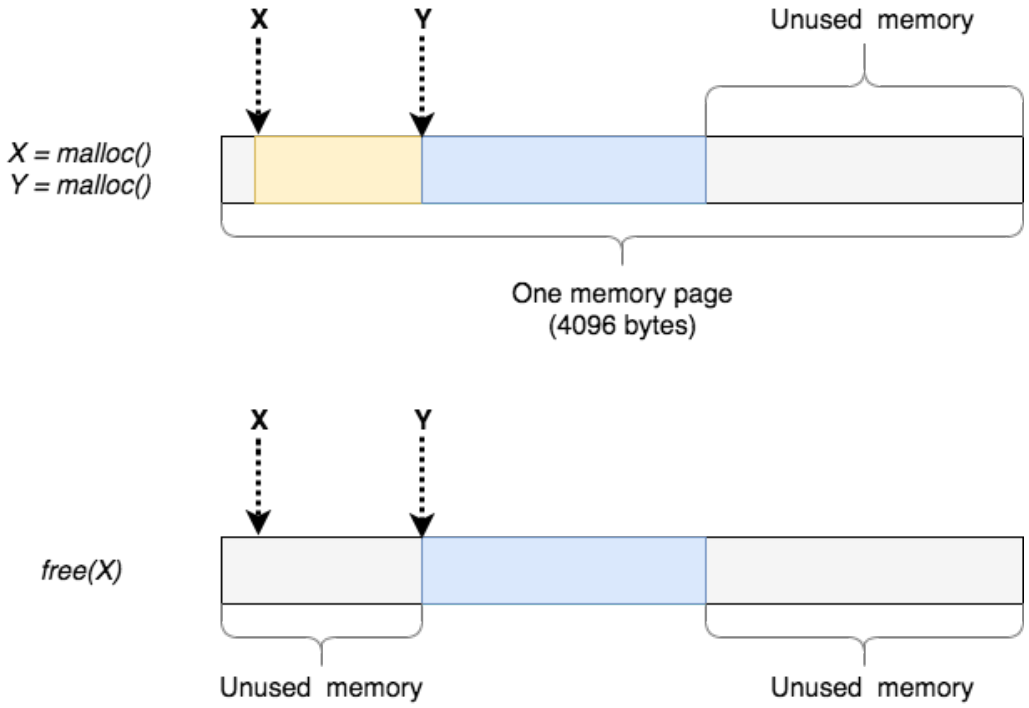

Figure 1.1: Memory layout of two small allocations. X and Y are pointers, referencing their corresponding memory regions. X becomes dangling

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 An example

Let's look at a less trivial example. This is a simplistic codebase, written in C++ of an in-memory messaging system. Each `User` has an inbox

and outbox, `Mailbox` objects, which wrap an `std::vector<Message *>`. `Message` objects are allocated on the heap, referenced by plain pointers, allowing the sender and recipient mailboxes to just both retain a pointer to the same message – a memory optimization. Each message object keeps track of who has deleted it, and when both the sender and receiver have done so, the message can be safely deallocated.

```
1  struct Message {
2      const std::string mContent;
3
4      bool mHasSenderDeleted = false;
5      bool mHasRecipientDeleted = false;
6
7      explicit Message(std::string content)
8          :   mContent{std::move(content)}
9      {}
10
11     void OnDeleted() {
12         if (mHasSenderDeleted && mHasRecipientDeleted)
13             delete this;
14     }
15 };
16
17 struct Mailbox {
18     std::vector<Message*> mMessages;
19
20     void AddMessage(Message* msg) {
21         mMessages.push_back(msg);
22     }
23
24     void DeleteMessage(Message* msg) {
25         mMessages.erase(std::find(mMessages.begin(), ↩
               mMessages.end(), msg));
26         msg->OnDeleted();
27     }
28 };
29
30 struct User {
31     Mailbox mInbox;
32     Mailbox mOutbox;
```

```
33
34      void SendMessage(User& recipient, std::string ↩
           content) {
35          Message* msg = new Message{std::move(content)};
36
37          mOutbox.AddMessage(msg);
38          recipient.mInbox.AddMessage(msg);
39      }
40
41      void DeleteReceivedMessage(Message* msg) {
42          msg->mHasRecipientDeleted = true;
43          mInbox.DeleteMessage(msg);
44      }
45
46      void DeleteSentMessage(Message* msg) {
47          msg->mHasSenderDeleted = true;
48          mOutbox.DeleteMessage(msg);
49      }
50  };
```

The noteworthy lines have been highlighted. While this design is error-prone, as we will see, it does work correctly and does not – in its current form – represent a vulnerability.

However, code evolves over time as bugs are fixed and new features are added, often by a different developer than the original authors. Sometimes these programmers understand the codebase less, or have less experience with programming or the technologies used, and can easily make mistakes. Especially with a language like C and C++, mistakes are extremely easy to make, and sometimes hard to notice, let alone debug.

Consider now that another programmer comes along and has to implement a feature to allow forwarding messages. His deadline is in an hour, perhaps there is a presentation scheduled with a big client, and this feature was simply forgotten about until now. This programmer adds a simple function as a quick hack to get message forwarding to work, and schedules some time for next month to revisit the feature and implement it properly. This function is added:

```
1  struct User {
2  // ...
3
```

```
4       void ForwardMessage(User& recipient, Message* ↩
            msg) {
5           // TODO: do this properly later
6           recipient.mInbox.AddMessage(msg);
7       }
8
9   // ...
10  };
```

He didn't understand how the simplistic reference counting of the `Message` objects work, and a quick test showed that this feature seems to work reasonably well. His attention was quickly drawn away by another tasks and this code won't be revisited for a while.

The problem shows itself when a message that was forwarded gets destroyed. While the code correctly ensures that the message is removed from the both the sender and the recipient's mailbox before it can be destroyed, but any potential forwardees were not taken into account. Consider now the following chain of events:

```
1   Message* funnyMessage = bob.SendMessage(alice, "Hey, ↩
        look at this funny gif: <image>");
2   bob.DeleteSentMessage(funnyMessage);
3
4   alice.SendMessage(cecile, "Haha, look at this funny ↩
        gif!");
5   alice.ForwardMessage(cecile, funnyMessage);
6
7   alice.SendMessage(bob, "HAHA that's pretty awesome");
8   alice.DeleteReceivedMessage(funnyMessage);
```

Bob sends a message to Alice, who forwards it to Cecile. Both Bob and Alice delete the message, causing the object to be destroyed, while Cecile's inbox still retains a pointer to it: a dangling pointer! What will happen if Cecile looks at her inbox? The application will attempt to dereference the dangling pointer, with unpredictable results.

What if there's another message, containing some sensitive information, is sent directly afterwards, potentially between completely unrelated users?

```
1   bob.SendMessage(daniel, "My PIN code is 6666");
```

Depending on the memory allocator, it's entirely possible that the memory referenced by `funnyMessage` before is now reused for the new, secret

message. In this case, Cecile's inbox now contains a message not intended for her, containing sensitive information.

```
1 for (const Message* msg : cecile.mInbox.mMessages) {
2     std::cerr << msg->mContent << "\n";
3 }
```

The following output is produced when compiled with a recent version of GCC and run:

```
Haha, look at this funny gif!
My PIN code is 6666
```

This is an example of how dangling pointer errors can pose a security vulnerability, even without the active efforts of an attacker.

It's worth noting that using a correct implementation of reference-counting, such as with the standard `std::shared_ptr` (since C++11), this problem could have been avoided. However, while smart pointers go a long way towards making dynamic memory safer and more convenient to use, they do have limitations even in the current C++ version (C++17 as of the time of writing). For instance, it's common to use plain pointers to represent a non-owning, optional reference to memory owned by another object, such as an `std::unique_ptr`, enabling dangling pointer errors to occur.

1.4 Overview

TODO: Better title? Move it somewhere else? I don't think the introduction is the right place for this.

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, writeable, but not executable, while the memory pages hosting the application code can be made readable, executable, but not writeable, 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

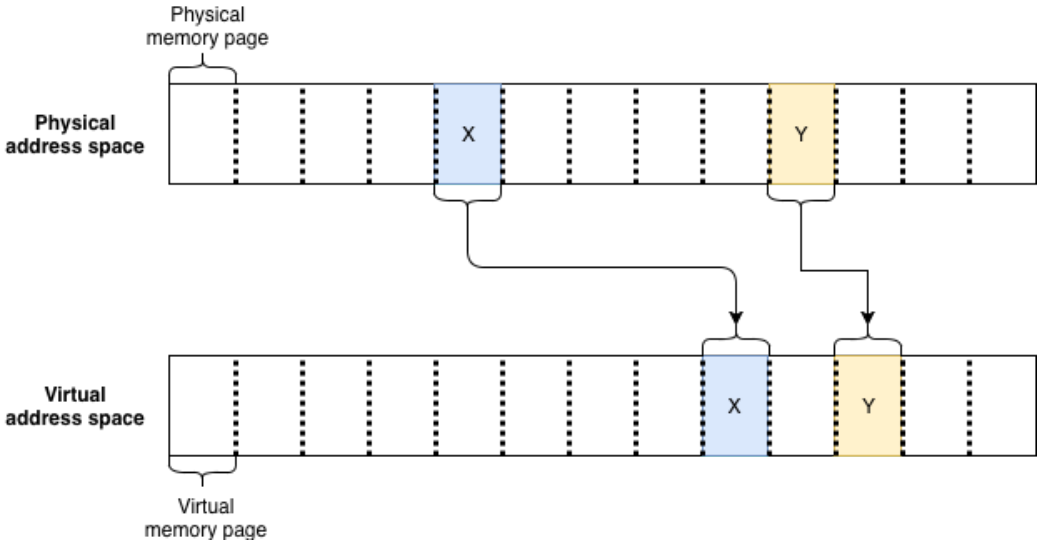

Figure 2.1: Mapping two physical memory pages X and Y to virtual memory

- De-duplicating 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.

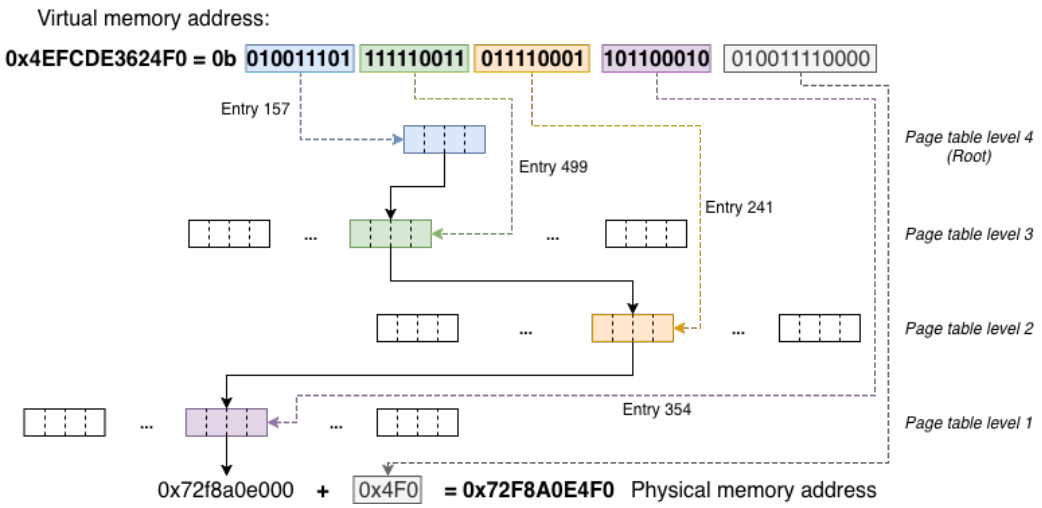

Figure 2.2: Translating a virtual memory address to physical using the page tables

The bits of the virtual memory address identify the page table entry to follow during address resolution. For 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, as well as a series of bits that represent the different access permissions, such as writeable and executable. 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.)

It's important to realize that it is possible to map a physical page into multiple virtual pages, as well as to have unmapped virtual pages. Attempting to access a virtual page (by dereferencing a pointer to it) that is not mapped – i.e. not backed by a physical memory page – will cause a *page fault* and execution to trap inside the kernel. The kernel then can decide what to do – for instance if it determines that the memory access was in error (an *access violation*), it passes the fault on to the process which usually

terminates it. On Linux this is done by raising the `SIGSEGV` signal (segmentation violation or segmentation fault) in the process, normally aborting its execution.

Other types of access violation, such as attempting to write a non-writeable page – a page on which writing was disallowed by setting the corresponding bit in its page table entry to 0 – or attempting to execute a non-executable page – a page which has its no-execute bit set, a new addition in the x86-64 architecture over x86-32 – will also trigger a page fault in the kernel the same way.

This mechanism also allows the kernel to perform memory optimizations. These are important, because (physical) memory is often a scarce resource. For example, a common scenario is that multiple running processes use the same shared library. The shared library is a single binary file on disk that is loaded into memory by multiple processes, meaning that naively the same data would be loaded into memory multiple times, taking up precious resources for no gain. The kernel can instead load the shared library into physical memory only once and then map this region into the virtual address space of each user application. Other de-duplication opportunities include static libraries that are commonly linked into applications (such as the C standard library), or the same binary executing in multiple processes.

In addition to memory de-duplication, the kernel can also choose to compress or even swap out rarely used memory. In these cases, the kernel marks the relevant page table entries as invalid, causing the hardware to trigger a page fault in the kernel if they are accessed. Upon access, instead of sending a signal to the process, the kernel restores the data by decompressing it or reading it from disk, after which it will resume the process which can continue without even being aware of what happened. Modern kernels include a large number of similar optimizations. **TODO: Could probably find some citations for this**

2.2 Prior art

TODO: Oscar, etc.

2.3 Dune: light-weight process virtualization

TODO: What is Dune, which version of Dune is used and why, and the patches applied TODO: Important: memory layout, how is virtual memory mapped by default, since we make extensive use of that TODO: Also write about how multi-threading is not currently supported by Dune, and how one could go about supporting it using the vmcall-hooks. TODO: Also write about the brk() bug with exceeding 4 GB leading to EPT violations; see vmcall-hooks.c

TODO: This should probably be earlier in the chapter (or maybe in Background?) TODO: Also have to write about the requirements of Dangless and how to build it

2.4 Rumprun

TODO: Development started on rumprun, why did I end up dropping it?

Chapter 3

Dangless - Implementation

3.1 API overview

Dangless is a Linux static library `libdangless.a` that can be linked to any application during build time. **TODO: Probably should write about how to build it and how to link it to existing applications.** 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 [5], 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 Performing an allocation

Whenever Dangless is asked to allocate some memory via a call to `dangless_malloc()`, `dangless_calloc()`, or `dangless_realloc()`, a number of steps have to happen: physical memory has to be allocated, virtual memory has to be allocated, and the mapping created. Most of the process is the same regardless of the exact function called. The only exception is `dangless_realloc()`, which I will detail later.

3.2.1 Allocating physical memory

The first step Dangelss has to perform is to acquire the 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()` [4]. 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 [8].

Finally, when `sysmalloc()`, etc. returns, we have a completed physical memory allocation. However, what is returned to us is a virtual memory address. We could perform a pagetable walk to find the corresponding physical memory address, but this is unnecessary, as the mapping provided by Dune is very simple, so it's sufficient to use Dune's `dune_va_to_pa()` function from `libdune/dune.h` that is far cheaper computationally than a page-table walk.

The implementation of Dangless cannot handle the system allocator returning a (guest) virtual memory region that is backed by non-contiguous (guest) physical memory. This should not normally be a problem, unless Dangless is used together with code that implements the system calls used by memory allocators (usually `brk()` and `mmap()`). Note that it does not matter whether the host physical memory is contiguous or not: any `mmap()` (or `brk()` for that matter) allocation is mapped into the guest memory contiguously. **TODO: I think that's the case though, but I'm not totally sure. Check this? Note that the implementation can be fixed if necessary.**

3.2.2 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 singly linked-list of `vp_span` objects each representing a free span of virtual memory, ordered by their end address:

```
1  struct vp_span {
2      vaddr_t start;
3      vaddr_t end;
4
5      LIST_ENTRY(vp_span) freelist;
6  };
7
8  struct vp_freelist {
9      LIST_HEAD(, vp_span) items;
10 };
```

(The NetBSD `queue.h` [7] v1.68 is used for the linked list handling macros.)

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 (by adjusting `start`), and the span is deleted if it is now empty. 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-understood problem.

3.2.3 Remapping

TODO: Use diagrams for this, such as the ones used in the presentation

Now that Dangless has the physical memory address and a brand new virtual memory address, all that is left to do is mapping the virtual memory to the physical memory by modify the guest page tables. In a normal Linux userland application, this would not be possible to do directly or cheaply without implementing a Linux kernel module. Dune makes it possible for us to do this, as inside the virtualized environment, we have ring 0 privileges, so we can read control register 3 (`cr3`) containing the (guest) physical address of the page table root. Thanks to the `dune-ix-guestppages.patch` patch to Dune, the host memory pages used to hold the guest's page tables are mapped into the guest virtual memory, allowing us to manipulate them during runtime from inside the guest.

Should the physical memory allocation fail, the machine is out of memory, and Dangless can do little but pass on this failure to the caller.

We have a different situation however if it's the virtual memory allocation that 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 mechanism 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.

3.2.4 Deallocations

When deallocating some memory using `dangless_free()`, the challenge is detecting whether the given pointer was successfully remapped previously, and if so, obtaining the original virtual address that can be passed to `sysfree()`. This is necessary if we don't want to make assumptions about the underlying allocator's implementation details. Typically, memory allocators will not behave correctly if a different virtual memory address is used for deallocation than the one returned during allocation, even if both map to the same physical memory address.

To obtain the original virtual memory address, we make use of Dune's simple memory layout. First, we perform a page walk on the virtual memory address to obtain the corresponding physical memory address. Then we can compare `dune_va_to_pa(ptr)` (where `ptr` is the potentially-remapped memory address we're trying to free) to the resulting physical address. If they match, then `ptr` was mapped into virtual memory by Dune itself, meaning that it's not a memory address assigned by Dangless. This is possible, on allocations that were performed before Dangless finished initializing. In this case, we can simply call `sysfree()` to perform the deallocation.

Otherwise, we have to determine what Dune-mapped virtual memory address belongs to the obtained physical memory address. In other words, given `PA`, we have to determine `VA` such that `dune_va_to_pa(VA)= PA`. Once again, this is something that Dune's memory layout makes easy to do, as these conversions are performed using simple arithmetic. Finally, we can call `sysfree()` to perform the physical memory deallocation.

What is left to do is invalidating the page table entries for the remapped virtual memory address, to make sure that any dangling pointer access will fail. Locating the relevant page table entries is done by performing a page-table walk down to the 4K pages. The relevant page table entries are then overwritten by an entry that does not have the *present* bit set. Dangless uses the 64-bit value `0xDEAD00` for this purpose, to make the invalidated entries easily identifiable. We then flush the TLB (Translation Lookaside Buffer; used to cache the results of pagewalks for performance) to force the CPU to check the PTE should an access occur.

In order to determine how many of the page table entries we need to invalidate, we have to know how many 4K pages did the allocation span. Recall that Dangless places each allocation on virtual memory pages that will never be used for anything else. `malloc_usable_size()` is used to get the size of the allocation in bytes. (Of course, this has to be done before calling `sysfree()`.) Note though that determining the number of spanned pages is not as easy as it might seem at the first glance, because the allocation can start anywhere within a memory page. This means that, for instance a 512-byte region can span 1 or 2 pages depending on where it begins within the first page (Figure 3.2.4).

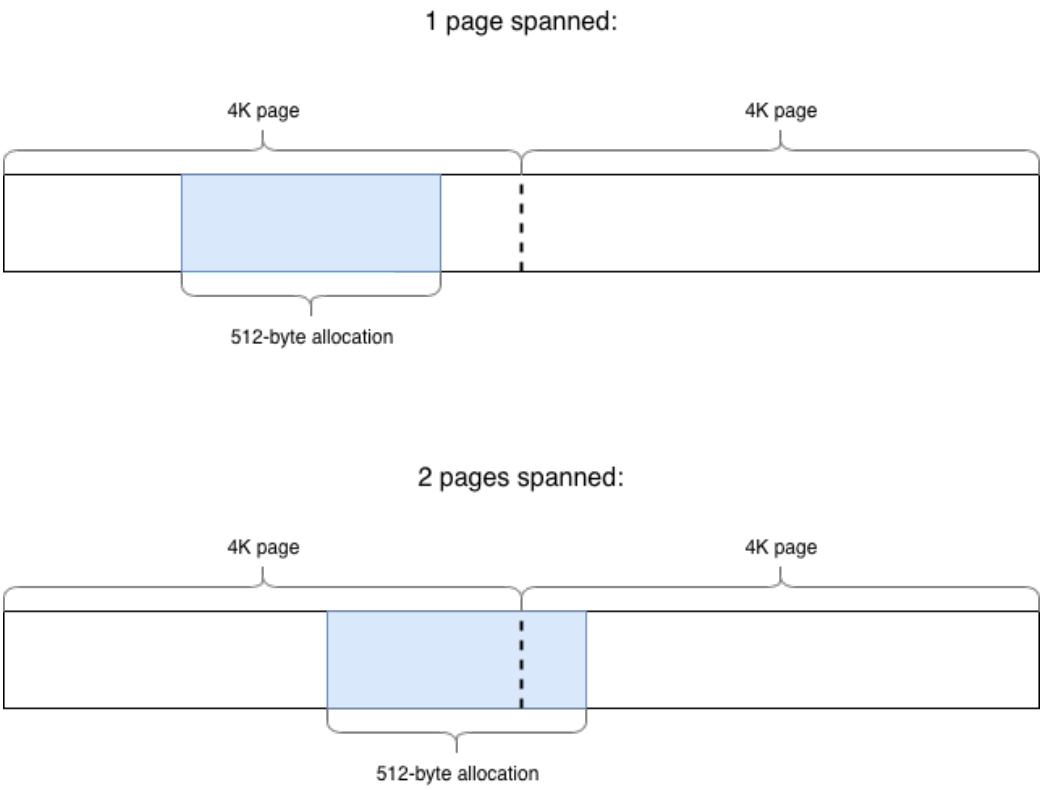

Figure 3.1: A memory allocation can span different number of pages depending on where it begins

3.2.5 Handling realloc

TODO: Is this interesting to write about? It's just a software engineering problem, nothing theoretically interesting about it

3.3 Fixing up vmcalls

3.3.1 The problem

One of the goals of Dune is to remain as simple as possible, and not re-implement most functionality of operating system kernels when not absolutely necessary. This means that most system calls performed the application running inside Dune are not actually handled by Dune itself, but rather are passed on to the host kernel via the `vmcall` instruction. (`vmcall` is identical to `syscall`, except it exits the virtual environment first.) This includes tasks as common as I/O operations (such as `printf()` or `fopen()`), or even memory management (such as `mmap()`).

This presents a challenge for Dangless, which can be efficient because it can directly manipulate the page tables inside the virtual environment (guest machine) itself, without having to manipulate the host's page tables (which would only be possible via a system call such as `mprotect()`). However, any virtual memory addresses returned from Dangless will only be valid inside the guest environment, meaning that attempting to pass such a memory address to the host kernel when performing a system call (vmcall) will fail with `EINVAL`.

To demonstrate the problem, first consider the following:

```
1 puts("Hello world!\n");
```

The `puts()` standard library function used here is a comparatively simple wrapper around the `write()` system call, and unlike `printf()`, does not perform any formatting or other manipulation of the string [6]. Recall that in C, strings are represented by null-terminated character arrays, and are typically passed to functions as `const char*`: a (virtual) memory address pointing to the first character of the string. Since in this example, the argument to `puts()` is a string literal, the string data is stored in the executable's data section (such as `.rodata`) without any dynamic memory allocation that would go through Dangless. The result is that the pointer

passed to the `write()` vmcall references virtual memory that is mapped by the host machine, so the call will succeed.

```
1  void determineAnswer(char* buffer) {
2      strcpy(buffer, "Fourty-two");
3  }
4
5  char* answerBuffer = malloc(64 * sizeof(char));
6  determineAnswer(answerBuffer);
7
8  puts("The answer is: ");
9  puts(answerBuffer);
10 puts("\n");
11
12 free(answerBuffer);
```

I hope that the problem is now clear: we're passing a `malloc()`-d buffer to `puts()`. Since `malloc()` refers to `dangless_malloc()`, it will perform virtual memory remapping inside the guest system, yielding a pointer value in `answer` that is valid inside the guest machine, but not in the host machine. (Of course, the data is present in the host physical memory, but is not mapped to host virtual memory.) The `write()` system call when executed by the host, is not going to be amused by this fact, and is going to return the error code `EINVAL`, indicating an invalid argument.

In this case by knowing how Dangless works it is easy to spot the problem. However, often dynamic memory allocation is performed and the resulting pointer is passed to a system call in a way that's not immediately obvious from the user code, such as when done internally by the standard library implementation. The simplest example is probably `printf()`, which can call `malloc()` in some circumstances [3] [1]. I've also already mentioned how `dlsym()` when running for the first time will call `calloc()`.

3.3.2 Intercepting vmcalls

In order to fix this problem, Dangless needs to intercept any system calls that are about to be forwarded to the host kernel. This capability is not provided by default in Dune, so I've implemented it (`dune-ix-vmcallhooks.patch`), allowing Dangless to register a pre- and post-hook function that will be called before and after a `vmcall` instruction is performed by Dune, respectively. In the pre-hook, Dangless can access and modify the system call number, any of

the arguments, and even the return address. Similarly, the post-hook exposes the syscall return code.

3.3.3 Determining which arguments to rewrite

The next problem is how to determine what the system call arguments are, and which of them can possibly reference a Dangless-remapped pointer. Note that it is not sufficient to find the pointers among the arguments themselves, as pointers can be nested: the arguments can be pointers to arrays or structures which in turn contain pointers – sometimes, after a few layers of indirection. Examples include the `readv()` and `writev()` system calls, which are used by GNU implementation of the `<iostream>` standard C++ header such as when writing to `std::cout`, i.e. `stdout`:

```
1 ssize_t readv (int fd, const struct iovec *v, int n);
2 ssize_t writev(int fd, const struct iovec *v, int n);
3
4 struct iovec {
5     void  *iov_base; /* Starting address */
6     size_t iov_len;    /* Number of bytes */
7 };
```

When handling either of these functions, not just the `const struct iovec *v` pointer has to be fixed, but also the `void *iov_base` pointer inside the pointer `struct iovec`.

Another example is the functions `execve()` and `execveat()`:

```
1 int execve(const char *filename, char *const argv[], char *const envp[]);
2 int execveat(int dirfd, const char *pathname, char *const argv[], char *const envp[], int flags);
```

Besides the `const char *filename` simple pointer, the parameters `char *const argv[]` and `char *const envp[]` are both a null-terminated array of pointers, in which every entry has to be fixed.

Finally, pointers to more complicated structures are also sometimes passed as system call arguments:

```
1 ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
2
3 struct msghdr {
```

```
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    size_t        msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags on received message */
};
```

So, we need some way to determine which arguments of the system call can be a pointer, and find any nested pointers. Essentially, what we need is to be able to tell for a system call number what arguments it takes and what type they are. For this purpose, I have created a Python module `linux-syscallmd` (source on GitHub [2]) that parses the Linux kernel header file `include/linux/syscalls.h` and exposes it to user code. The Dangless script at `scripts/gen_syscall_param_fixup_flags.py` then uses this information to generate a file containing C code that can be `#include`-d to utilize this data inside Dangless:

```
// sources/src/platform/dune/vmcall_fixup.c

enum syscall_param_fixup_flags {
    SYSCALL_PARAM_VALID     = 1 << 0,

    SYSCALL_PARAM_USER_PTR  = 1 << 1,
    SYSCALL_PARAM_IOVEC     = 1 << 2,
    SYSCALL_PARAM_PTR_PTR   = 1 << 3,
    SYSCALL_PARAM_MSGHDR    = 1 << 4,

    // indicates that the parameter is the last interesting one, no need to check the rest
    SYSCALL_PARAM_LAST      = 1 << 7
};

static const u8 g_syscall_param_fixup_flags[][SYSCALL_MAX_ARGS] = {
```

```
16      #include ↩
            "dangless/build/common/syscall_param_fixup_flags.c"
17  };
```

This generates the array `g_syscall_param_fixup_flags` which contains an entry for each known system call number. The using code, `vmcall_fixup_args()`, looks up the entry for the current system call number and checks which flags are set. **TODO: user ptr / iovec / ptrptr / msghdr are all mutually exclusive, should not be part of the flags, but rather a 2-bit counter**

3.3.4 Rewriting the pointers

Now that we know which arguments to rewrite or "fix-up", we can use the same logic as `dangless_free()` to get the canonical pointer from a potentially-remapped one (see Section 3.2.4). We then replace the remapped pointer value with the canonical one in the system call arguments.

In case of nested pointers this involves modifying the referenced in-memory data, meaning we cannot simply replace the pointer. This is because the original pointer was a remapped pointer, allocated via Dangless, and will be deallocated via Dangless. However, due to the nested pointer fix-up, the user code can now potentially access the canonical (non-remapped) pointer, opening the door to dangling pointer errors. Furthermore, should a canonical pointer be passed to `dangless_free()`, it cannot invalidate the remapped memory region as it doesn't know where it might be.

To demonstrate, consider the following code:

```
1  char *first = malloc(32);
2  strcpy(first, "Hello ");
3
4  char *second = malloc(32);
5  strcpy(second, "world!");
6
7  struct iovec iov[2];
8  iov[0].iov_base = first;
9  iov[0].iov_len = strlen(first);
10 iov[1].iov_base = second;
11 iov[1].iov_len = strlen(second);
12
13 writev(STDOUT_FILENO, iov, 2);
```

Due to pointer rewriting the system call succeeds, but afterwards we end up with `iov[0].iov_base != first` and `iov[1].iov_base != second`, as they have been replaced by their canonical counterparts to make the system call succeed on the host kernel.

Later, we deallocate the buffers:

```
1 free(second);
2 free(first);
```

This is fine, since the `first` and `second` variables were not affected by the pointer rewriting, so Dangless correctly invalidates the remapped regions in `dangless_free()`. But then, later:

```
1 fprintf(stderr, "Attempted writev() with '%s' and
    '%s'!\n", iov[0].iov_base, iov[1].iov_base);
```

Here we have an attempted memory access through two dangling pointers. Recall that, due to pointer rewriting, `iov[0].iov_base` and `iov[1].iov_base` are the canonical pointers (as returned by `sysmalloc()`) and so do not point into the remapped region that was invalidated by the earlier `dangless_free()` calls. Therefore, this error will not be caught by Dangless!

To resolve this situation, for every nested pointer fix-up, Dangless records the pointer location (a pointer to the user pointer) as well as the original value stored there (the Dangless-remapped pointer value) in a buffer. After the vmcall returns but before it jumps back into user code, we then go through the records and restore any such rewritten pointer values to their original ones, preventing the user from being exposed to non-remapped pointer values.

TODO: Note that the implementation of the above is currently on the vmcall_rewrite_improved_v2 branch and needs to be tested before merging in

3.3.5 Limitations

This approach is limited in that Dangless can only handle system calls, arguments, and argument types that `linux-syscallmd` recognizes when building Dangless.

For instance, `linux-syscallmd` currently does not understand or process preprocessor macros, such as `#if` and `#ifdef` sections. This means that system call signatures not relevant for the current system will also be parsed, leading to conflicting signatures for some system calls, such as `clone()`.

`linux-syscallmd` currently does not handle this situation, and will just pick the last occurrence of the same system call in the source file, which may be different than the signature actually used by the kernel. Because of this, Dangless has special handling of the `clone()` system call, but naturally that can't extend to, for example system calls introduced in the future.

As of writing, I do not know of a better way to approach this. Fixing the present limitation would involve knowing what values were used for each preprocessor macro while building the kernel, and I'm not aware of any way in which the kernel exposes this information.

Another issue is understanding which arguments can hold pointer or nested pointer values. For the vast majority of system calls this is straightforward, as `linux/syscall.h` consistently marks the pointer arguments as `__user *`, and the pointed type is obvious, whether it's `char` or `struct iovec`. But some system calls will interpret the same argument differently depending on the context of the call, such as `ptrace()`. The signature of `ptrade()` is as follows:

```
1 long ptrace(enum __ptrace_request request, pid_t pid,
     void *addr, void *data);
```

Notice that `addr` and `data` are both untyped (`void`) pointers. How they are interpreted differs depending on the value of the `request` argument. Some examples:

- `PTRACE_TRACEME`: both `addr` and `data` are ignored.
- `PTRACE_PEEKTEXT`: `addr` does *not* correspond to pointer in the address space of the caller, but rather, refers to a location in the address space of the target application. The same pointer value may reference a memory region that's unmapped, or worse, mapped for something completely different in the address space of the calling process. As such, Dangless should not touch it. `data` is ignored.
- `PTRACE_POKEDATA`: `addr` refers to a memory address in the target process. `data` might not be a pointer value at all, but rather the word to be copied into the target process' memory.
- `PTRACE_GETREGS`: `data` is an actual pointer in the calling process, while `addr` is ignored.

- `PTRACE_GETREGSET`: `addr` is not a pointer value at all, but rather an enumeration. `data` is an actual pointer to the calling process' memory, but it references a `struct iovec` value, meaning that it will contain nested pointers that also have to be fixed up by Dangless.

`ptrace()` is a special case due to its very specialized nature, and it's unlikely to be used at all in the vast majority of applications that Dangless would be relevant for. Because of this Dangless ignores `ptrace()`, even though it would be possible to cover all of these scenarios.

There may be other system calls that have similar behaviour, although likely not as pathological as `ptrace()`. These are currently not handled in any way by textttlinux-syscallmd nor Dangless. Supporting all of these scenarios would inevitably involve extending Dangless to handle each on a case-by-case basis.

TODO: Probably have a chapter about using Dangless, such as how to build Dangless, what system requirements does it have, and how to make it work on an existing application.

Bibliography

[1] 0ctf 2017 - easiestprintf (pwn 150). https://blog.dragonsector.pl/2017/03/0ctf-2017-easiestprintf-pwn-150.html. Accessed 2019-07-09.

[2] Github source for linux-syscallmd. https://github.com/shdnx/linux-syscallmd. Accessed 2019-07-09.

[3] GNU C library - printf() can use malloc(). https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html#Hooks-for-Malloc. Accessed 2019-07-08.

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

[5] 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.

[6] Hello world analysis. http://osteras.info/personal/2013/10/11/hello-world-analysis.html. Accessed 2019-07-08.

[7] NetBSD queue.h header file - reference. https://netbsd.gw.com/cgi-bin/man-cgi?queue. Accessed 2019-06-30.

[8] 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.