Integrating the Kernel Address Sanitizer into the Mimiker Operating System

(Integracja mechanizmu Kernel Address Sanitizer
z systemem operacyjnym Mimiker)

Julian Pszczołowski

Praca magisterska

Promotorzy: Krystian Bacławski
Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

7 lipca 2020

Abstract

Memory bugs are a source of an evergoing concern for the safety and correctness of programming languages like C and C++. The problem gets worse as the programs get bigger and more complex – operating system (OS) kernels, on which the thesis concentrates, are a perfect example of that as the Linux kernel has grown to nearly 30 million lines of code in 2020 [61]. Here I provide an overview of memory bugs and explain why they are so dangerous in terms of application security. Then I discuss generic ways of finding memory bugs as well as state-of-the-art tools available in various OS kernels. Lastly, the focus is put on the Kernel Address Sanitizer (KASan), a tool for finding buffer overflows and uses of freed memory in the kernel-space, which has been recently integrated into many popular OSes including Linux, macOS and NetBSD. I describe the KASan thoroughly, but most importantly, I present how I've added it to the Mimiker (an open-source OS developed at the University of Wrocław) and show what are the benefits of my work.

Błędy zarządzania pamięcią są nieustannym źródłem problemów dla programistów języków C i C++. Zagrażają one nie tylko poprawności, ale i bezpieczeństwu wielu aplikacji. Sytuacja jest tym poważniejsza, im więcej kodu ma dane oprogramowanie. Szczególnie odczuwalne jest to w jądrach systemów operacyjnych (SO) – dla przykładu, w 2020 roku jądro systemu Linux zawiera już prawie 30 milionów linii kodu [61]. Poniższa praca skupia się na błędach zarządzania pamięcią, szczególnie w kontekście jąder SO. Przedstawione są rodzaje takich błędów, powody dla których są one niebezpieczne, jak i ogólne techniki ich wykrywania. Następnie opisane są konkretne narzędzia służące do namierzania tego typu podatności, które obecnie można znaleźć w najpopularniejszych SO. Szczególny nacisk położony został na jedno z takich narzędzi o nazwie Kernel Address Sanitizer (KASan), wykrywające przepełnienia bufora i użycia zwolnionej pamięci w przestrzeni jądra, które w ostatnich latach pojawiło się w wielu systemach takich jak Linux, macOS czy NetBSD. W ramach poniższej pracy KASan został dodany do jądra systemu operacyjnego Mimiker (system rozwijany w Instytucie Informatyki Uniwersytetu Wrocławskiego), co również zostało dokładnie opisane. Na koniec przedstawiona jest lista podatności, które dzięki narzędziu KASan udało się znaleźć i wyeliminować z jądra Mimiker.

Contents

Chapter 1

Introduction

Neither an operating system (OS) nor its applications can be secure and reliable if they're based on a buggy OS kernel. The kernel, which is the main part of a system, has an unlimited control over all resources. It provides an abstraction layer that user programs need to launch, run, communicate, access memory, or perform input/output operations. It's also the only component in a system that executes in privileged processor mode. Therefore, the kernel's safety is crucial to the entire system's safety.

Many modern OS kernels are written in languages known for their vulnerability to memory management errors, with C programming language being the most notable example. Moreover, memory bugs are responsible for about 70% of software security issues, as reported by the Microsoft Security Response Center [42]. Numerous techniques have been developed in order to locate such bugs. One of them is the Kernel Address Sanitizer (KASan), a tool that can be integrated into OS kernel, which detects buffer overflows and uses of freed memory at run-time. The tool has been initially added to Linux in 2015 [57] and since then to some other well-known OSes, including NetBSD in 2020 [58]. There are systems, like FreeBSD, in which the integration of the tool is still a work in progress [60]. In this paper I show how I've added the KASan into the Mimiker kernel[1]. It is worth noting that the tool is architecture-dependent, and none of the aforementioned OSes implements the KASan for MIPS processors, as Mimiker does.

The rest of this paper is structured as follows: in chapter 2, I discuss memory bugs in general, show why they're a serious threat to any application's security, and how one can find them. In chapter 3, I show what are the state-of-the-art tools for detecting memory errors in modern OS kernels including the Linux kernel. Then, in chapter 4, I describe precisely how the KASan works. Finally, in chapter 5, I present my contribution which is adding the KASan into the Mimiker, including example

[1] Mimiker [67] is an open-source operating system developed at the University of Wrocław since 2015 for educational and research purposes. It is Unix-like and inspired mainly by the *BSD world. Currently it supports only MIPS32 architecture.

error report, implementation details, and a list of already found bugs.

Even though many terms and concepts are explained within the thesis, some basic knowledge of operating systems and computer architecture is required. If needed, please refer to [11] and [14].

Chapter 2

Memory bugs

In this chapter you will find an overview of memory bugs in lower-level languages like C and C++, but much information also applies to other programming languages.

2.1 Types of memory bugs

There are many different kinds of bugs related to accessing memory. They can be caused just by dereferencing a wrong address but also by inappropriate use of some memory allocation library (e.g. `malloc` library), or even wrong understanding of the language's semantics (e.g. whether or not an uninitialized local variable is zeroed).

Overflow and over-read Overflows (over-reads) happen when a program writes (reads) to (from) a location after the targeted buffer. They can be divided into subcategories depending on the buffer type, for instance: local, global or heap overflow (over-read).

Use-after-free This category contains bugs that occur when a program uses deallocated memory. There are other two closely related types which can be thought of as subcategories: use-after-return and use-after-scope. The main difference is how the memory was deallocated: by freeing a heap object, by returning from a function, or by exiting some local scope.

Invalid free If deallocation is done improperly, so not according to freeing function's documentation, it's considered invalid. Examples include double freeing an object or freeing wrong object (i.e. an address that does not come from an allocation request).

Use of uninitialized memory Uninitialized memory in C and C++ contains a value which is not predictable – for instance, a result of some previous computation. Therefore, computer programs shouldn't use it.

Memory leak A leak occurs when there is a buffer that a computer program no longer uses, but the buffer is not being deallocated. It reduces the amount of available memory which in some cases may cause application failure or trashing.

Invalid page fault Page faults can happen while accessing memory and that doesn't always indicate an error – for instance, Linux's copy-on-write technique uses page faults to detect first write to a shared memory page, which means that the kernel needs to create a local copy of that page ([10], section 2.4.1). However, a bug occurs when the kernel's page fault handler finds out that the memory access is truly illegal.

Race condition When two threads use some shared memory location but are not synchronized properly, a race condition happens and the result is unpredictable (it depends on interleaving of threads' actions).

2.2 Why memory bugs are dangerous?

Let's focus on what really can happen when a memory error occurs. Most such errors cause *undefined behavior* [16] so the result may be unpredictable. For instance, C standard states that the behavior of dereferencing a null pointer is undefined ([8], clause 6.5.3.2, paragraph 4), and Linux Programmer's Manual states that freeing a heap object more than once is also undefined [1]. Here some are some real-life examples of what such undefined behaviors can mean:

- The program crashes immediately (e.g. due to receiving a `SIGSEGV` signal in user-space or a hardware exception that leads to a kernel panic in kernel-space).
- The program modifies its memory (it can be a local variable but also memory allocator's metadata) in an unpredictable way which then leads to erroneous results.
- The program modifies its memory in an unexpected way (from the programmer's point of view) that can be controlled by an attacker – say, user providing program's input. This can lead to dangerous security vulnerabilities such as privilege escalation or arbitrary code execution.

Below you will find two simple examples of the third, most dangerous, option.

2.2.1 Buffer overflow example

Listing 2.1 contains code which reads two values from the standard input: username and whether or not the given user has admin privileges. Then it adds the user to some database. The buffer for username has 16 bytes (line 26) but is copied to

`user_t::name` buffer which has 8 bytes (line 7), using unsafe `strcpy` function[1]. If one specifies a name longer than 8 bytes, variable `user_t::is_admin` will be overwritten by non-zero value.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5
6  typedef struct {
7    char name[8];
8    bool is_admin;
9  } user_t;
10
11 void add_user(const char *name, bool is_admin) {
12   user_t *user = malloc(sizeof(user_t));
13
14   /* copy user data */
15   user->is_admin = is_admin;
16   strcpy(user->name, name); /* potential buffer-overflow! */
17
18   /* ... add user to the database ... */
19
20   if (user->is_admin)
21     puts("New admin created");
22 }
23
24 int main(int argc, char *argv[]) {
25   /* read name and privileges from stdin */
26   char name[16];
27   int is_admin;
28   scanf("%s %d", name, &is_admin);
29
30   add_user(name, is_admin);
31
32   return 0;
33 }
```

Listing 2.1: A code containing buffer overflow bug.

Let's see what will happen if we run the code and create a user with name „LongUserName" (more than 8 bytes) and no admin privileges.

```
1 $ gcc buffer-overflow.c -o buffer-overflow
2 $ echo "LongUserName 0" | ./buffer-overflow
3 New admin created
```

Listing 2.2: Exploiting buffer overflow bug in listing 2.1.

We clearly see that `user_t::is_admin` variable has been overwritten by buffer overflow and the user now is an admin (line 21)!

[1] `strcpy` does not take the buffer's length into account, thus is able to overflow the destination buffer [3].

Moreover, another issue is that creating an admin with 8-byte name is not possible since a null byte that follows the name would overwrite `user_t::is_admin`.

2.2.2 Use-after-free example

Other types of memory bugs can be exploited as well. Listing 2.3 contains vulnerable code with use-after-free bug. The code is inspired by task „Heap Two" from Exploit Education website [62]. Let's analyse what it does: there are two global pointers (`user_auth` and `user_name`, lines 10-11) to some user authentication data and username. There is also an infinite loop that reads commands from the standard input and depending on the command:

Command `auth` The program allocates new user authentication data (that contains `auth_t::logged_in` variable) and zeroes it (lines 22-24).

Command `reset` The program frees current user authentication data, but is not zeroing the `user_auth` pointer (lines 25-26).

Command `name [arg]` The program allocates a buffer for username and copies `arg` to the buffer (lines 27-28).

Command `auth` The program inspects `user_auth->logged_in` variable in order to check whether the current user is logged in (lines 29-34).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct {
6    /* ... some auth data ... */
7    int logged_in;
8  } auth_t;
9
10 auth_t *user_auth;
11 char *user_name;
12
13 int main() {
14   while (1) {
15     printf("[ auth = %p, name = %p ]\n", user_auth, user_name);
16
17     char command[128];
18     /* read command */
19     fgets(command, sizeof(command), stdin);
20
21     /* check which command has been given */
22     if (strncmp(command, "auth", 4) == 0) {
23       user_auth = malloc(sizeof(auth_t));
24       memset(user_auth, 0, sizeof(auth_t));
25     } else if (strncmp(command, "reset", 5) == 0) {
```

```
26        free(user_auth);
27      } else if (strncmp(command, "name", 4) == 0) {
28        user_name = strdup(command + 5);
29      } else if (strncmp(command, "login", 5) == 0) {
30        if (user_auth->logged_in) { /* potential use-after-free! */
31          printf("logged in successfully!\n");
32        } else {
33          printf("failed to log in!\n");
34        }
35      }
36    }
37  }
```

Listing 2.3: A code containing use-after-free bug.

The bug is that the command `reset` is not zeroing `user_auth` pointer after deallocating the data. There is a chance that giving `name` command just after `reset` will make `user_auth` and `user_name` point to the same buffer, as `strdup` function (line 28) calls `malloc` internally [2]. If so, `user_auth->logged_in` refers to bytes within username that one can control (and write non-zero data there).

```
$ gcc use-after-free.c -o use-after-free
$ ./use-after-free
[ auth = (nil), name = (nil) ]
auth
[ auth = 0x56241e340a80, name = (nil) ]
login
failed to log in!
[ auth = 0x56241e340a80, name = (nil) ]
reset
[ auth = 0x56241e340a80, name = (nil) ]
name John
[ auth = 0x56241e340a80, name = 0x56241e340a80 ]
login
logged in successfully!
```

Listing 2.4: Exploiting use-after-free bug in listing 2.3. Note that after `name` command both pointers have the same value.

Of course there are many other, sometimes very sophisticated, ways of exploiting security vulnerabilities in languages like C and C++. For more information I'd recommend reading various security-related books (e.g. [9]), blogs (e.g. [63], [64], [65]), or reading write-ups of tasks from CTF competitions[2].

[2]CTF (Capture The Flag) is a kind of information security competitions in which tasks are solved by exploiting security vulnerabilities.

2.3 Generic ways of detecting memory bugs

There are many techniques that one can use in order to detect memory errors in both user-space and kernel-space. The techniques, and tools that implement them, differ in:

- when the errors are detected (e.g. during compile time or run time),
- what types of errors are detected (e.g. some tools specialise in detecting concurrency issues only),
- what is the time and memory overhead.

Below you will find some general approaches to finding memory errors.

2.3.1 Formal verification

Formal verification is the process of checking whether a design satisfies some requirements (properties) [36]. It can be achieved i.a. using model checking or theorem proving [37]. Some interesting examples include:

- CompCert compiler [38]. CompCert is a compiler for the C programming language that has been formally verified using Coq, an interactive theorem prover.
- Formal verification of seL4 microkernel [39]. According to the authors, it is a first formal proof of functional correctness of a complete, general-purpose OS. In this case, Isabelle theorem prover has been used.

2.3.2 Other types of static analysis

In general, static analysis is a kind of analysis performed without running a program. There are many standalone tools able to detect errors that way, not to mention the ones built into IDEs and compilers. As far as C and C++ languages are concerned, it's worth noting that GCC 10 (next major release of GCC compiler) will have a brand new static analysis pass, able to detect e.g. some double-free bugs, that one can launch using `-fanalyzer` flag [41].

2.3.3 Code instrumentation

Instrumentation is a technique widely used in software profiling, virtualization, but also in error detection. It involves adding extra code to an application, either at compile time (static instrumentation) or at run time (dynamic instrumentation) [44]. Such additional code performs safety checks. Examples include:

- `libstdc++`'s debug mode, which requires additional `-D_GLIBCXX_DEBUG` compiler flag, and enables many run-time assertions detecting incorrect usage of the standard library (like `std::vector`'s out-of-bounds access) [46],
- Stack canaries – modifying functions' prologues and epilogues. In prologue, a randomly chosen integer (called canary) is placed on stack. In epilogue, the integer's value is checked to make sure it hasn't changed [48]. If it has, then stack buffer overflow is reported. This technique increases the difficulty of overwriting function's return address (stored within stack frame). Canaries can also be used by memory allocators – such allocator would then put a canary just after an allocated buffer.
- Address Sanitizer (ASan) – a tool which adds a run time check before each memory access in order to make sure that the access is valid. If it's not, an error is reported. Kernel variant of the tool (KASan) will be discussed in more detail in next chapters.

2.3.4 Dynamic recompilation

The term simply means recompiling a program (or its parts) during execution. Although dynamic recompilation is commonly used in emulators and virtual machines, it can also prove to be useful for finding memory bugs. For example, Valgrind – a well known memory debugging and profiling tool, translates the analysed program's machine code into an architecture-neutral intermediate representation (IR). Subsequently, the IR is modified (e.g. instrumented, depending on which mode was chosen) and then converted back into machine code ([49], section 3.2). Valgrind itself runs on real CPU and the analysed program runs on (conceptually) simulated CPU, so this approach shares some similarities with virtualization.

2.3.5 Fuzzing

Fuzzing is a way of testing the security of a program by programmatically providing it with many (valid or invalid) inputs, and seeing if the program crashes or exhibits other insecure behavior [51]. It's even more effective when used with other techniques like code instrumentation which will cause the tested program to crash immediately when an error is detected (instead of falling into undefined behavior). Many modern browsers, OS kernels etc. are constantly being fuzzed in order to find more and more overflows, race conditions, leaks, and the like.

Chapter 3

Tools for finding memory bugs in the kernel-space

In this chapter you will find an overview of various tools that modern OS kernels use to detect memory errors. The overview is not exhaustive, but covers all types of errors mentioned in chapter 2, and presents tools from a number of OSes like: Linux, Windows, macOS/iOS (XNU kernel), NetBSD, and FreeBSD.

3.1 kmemcheck

This tool was present in the Linux kernel for ten years, from 2007 to 2017. It was removed as right now the kernel sanitizers (described below in section 3.6) can detect even more errors with less overhead [17] [18]. I mention it in the thesis to show how the techniques have improved over time.

Generally speaking, kmemcheck is a dynamic checker for uninitialized memory usage. Here's how it works [23]:

- each allocation of a memory page in the kernel allocates an additional *shadow*[1] page, initially filled with zeros, which is a map showing which bytes in the original page have been already initialized,
- the kernel's page table is modified so that each access to the original page generates a page fault[2],

[1] Shadow memory is a way of keeping metadata about the main memory. Each byte in the shadow memory contains information about some chunk of original memory (in this case, the mapping is one byte to one byte). The technique is used in various dynamic analysis tools and more examples are discussed later in this thesis.

[2] Page table is a structure that keeps the mappings between virtual and physical addresses. Each mapping (also called PTE, *page table entry*) contains additional metadata (e.g. permissions) which can be modified so that a page fault is raised when accessing the virtual address ([11], section 9.3.2).

- if the page fault happens and it's a write, the corresponding bytes in the shadow page are marked as `0xFF` (initialized) and the execution is resumed; on the other hand, if it's a read, and at least one of the corresponding shadow bytes is equal to `0x00` (uninitialized), an error is reported.

The memory consumption is doubled as compared to uninstrumented case, but the time overhead is even more severe due to the handling of page faults. Thus, the tool can only be used while debugging the kernel and is not suitable for regular workloads.

3.2 kmemleak

A tool for detecting memory leaks built into the Linux kernel [19]. It works similarly to garbage collectors, but instead of freeing unused objects, it reports them as leaked. More specifically, all memory allocations in the kernel are tracked: pointer to a block, its size and other data are stored in a red-black tree. Deallocations are also tracked, as they remove a corresponding entry from the tree.

Every ten minutes (configurable interval) a thread scans the entire kernel memory (including registers, stacks, data section) and reports information about data leaks, i.e. all allocated blocks from the red-black tree that are not referenced from the memory.

The tool allows some false positives and false negatives to happen, partially to keep the algorithm simple.

3.3 KLEAK

KLEAK [50] is a feature able to detect kernel memory disclosures, which was added to NetBSD in 2020 [58]. Kernel memory disclosure is a use of uninitialized memory error subtype, occurring when uninitialized data is copied to the user-space. Such data may contain secrets like cryptographic private key or information about kernel memory layout (letting an attacker bypass KASLR[3]).

KLEAK uses a simple form of taint tracking: memory returned by kernel heap allocators, which isn't requested to be zeroed, is filled with a repeated 1-byte marker. Also, compiler instrumentation is used to dynamically taint parts of the stack with the marker – it cannot be done once as the kernel continuously uses the stack, possibly overwriting the taint.

Memory is exchanged between the kernel-space and the user-space using a few specific functions like `copyout` or `copyoutstr` [4]. Those functions are instrumented

[3]Kernel address space layout randomization (KASLR), a technique that prevents exploitation of some bugs by randomizing the kernel virtual address space [26].

as well – if the marker is detected in any buffer crossing the kernel boundary, a disclosure is reported.

3.4 InitAll

InitAll [43] is a mechanism created by Microsoft in order to eliminate uninitialized stack memory vulnerabilities. It automatically initializes all stack variables during compilation (in front-end of the compiler). The rationale is that static analysis, fuzzing and code review were not effective enough to deal with this class of vulnerabilities.

More specifically, InitAll fills uninitialized local variables (scalars, arrays, structures) with a pattern of either `0x00` or `0xE2` bytes, depending on the Windows kernel build. According to Microsoft Security Response Center, zero initialization is the best in terms of performance, code size and security properties. Although, in some cases it may lead to unexpected results, for instance when an uninitialized pointer goes down a „NULL pointer" branch instead of causing a crash whilst being dereferenced.

Microsoft reports that since InitAll has been added to the Windows kernel in 2019, multiple vulnerability reports stopped being reproducible on the latest versions of the system.

3.5 Sparse

Sparse (Semantic Parser for C) [20] is a tool for static code analysis of the Linux kernel. It was originally written by Linus Torvalds. Sparse can find a variety of errors, also the kernel-specific ones.

The tool allows the programmer to annotate the code so that even more issues are detected. For instance:

- for a given lock `L`, functions can be marked with `__must_hold(L)`, `__acquires(L)` or `__releases(L)` (invalid usage of locking primitives is reported by Sparse as *context imbalance*, see example report below),
- pointer types can be annotated with `__user` or `__kernel` (Sparse verifies whether a kernel pointer isn't passed as a user pointer, or whether a user pointer isn't directly dereferenced[4]),

[4]Kernel code mustn't directly use addresses provided by a user (e.g. via a system call's argument) since they may be invalid. In the Linux kernel, functions like `copy_to_user` and `copy_from_user` must be used instead ([10], section 4.13).

- integer types can be annotated with `__bitwise` to create a restricted type that can only be used with bitwise operations and cannot be mixed with other integer types (e.g. to define little-endian and big-endian integer types).

Such annotations can be easily found whilst reading the kernel code. In the Linux kernel 5.0, macro `__user` is referenced in 2389 files [52], macro `__releases` in 243 files [53].

Here are some examples of warnings issued by Sparse while verifying the Linux kernel, taken from [22]:

```
1 mm/slub.c:2830:47: warning: context imbalance in '__slab_free' - ↩
    unexpected unlock
2 drivers/gpu/drm/i915/i915_debugfs.c:4351:29: warning: Variable ↩
    length array is used.
3 drivers/scsi/eata.c:1652:13: warning: cast to restricted __be32
4 kernel/bpf/sockmap.c:332:43: warning: Using plain integer as NULL ↩
    pointer
5 drivers/gpio/gpio-pci-idio-16.c:120:27: warning: dereference of ↩
    noderef expression
6 drivers/iommu/amd_iommu.c:3598:21: warning: symbol '↩
    stupid_workaround' was not declared. Should it be static?
```

Listing 3.1: Examples of warnings generated by the Sparse tool.

3.6 Sanitizers

To begin with, let's briefly discuss user-space sanitizers – they're a set of tools designed by Google that perform dynamic analysis of mainly C and C++ programs. Most popular examples are: Address Sanitizer (ASan, detects overflows and uses of freed memory), Memory Sanitizer (MSan, detects uninitialized memory), Thread Sanitizer (TSan, detects data races) and Undefined Behavior Sanitizer (UBSan, detects undefined behaviors). All sanitizers require compiler support (for code instrumentation), run-time support (a dedicated run-time library) and most of them use shadow memory (the concept of shadow memory was explained in section 3.1).

The ASan was among the first sanitizers that the GCC compiler began to support in 2013 [40]. It provides little run-time overhead compared to other similar tools [29] and since its release it has found numerous bugs in various applications, including Chrome, Firefox, Vim, LLVM and even GCC itself [30]. Its kernel-space counterpart, the KASan, was integrated into the Linux kernel in 2015 [57] and has also found many errors [31]. Both tools use the same method of finding bugs, although implementing the latter is more difficult.

Today we can find various sanitizers built into more and more widely-used OS kernels, like: Linux, *BSDs, macOS. The compile-time instrumentation (support

from GCC and LLVM) gives the sanitizers an advantage over other similar tools in terms of performance.

The following section gives an outline of how all the kernel sanitizers work. Please note that the KASan is discussed in greater detail in chapter 4.

3.6.1 Kernel Address Sanitizer

The KASan uses shadow memory to keep information about validity of each byte in the main memory (eight original bytes map to one shadow byte). Moreover, the compiler instruments the code by adding a shadow check before each memory access. For instance, the following code:

```
1 *ptr = 42; /* store of size 8 */
```

would be changed to:

```
1 __asan_store8(ptr);
2 *ptr = 42; /* store of size 8 */
```

Where `__asan_store8` function has to be implemented by KASan's run-time library. Basically, it should check the corresponding shadow byte and report an error if `ptr` points to an invalid address. Other `__asan_*` methods are discussed in the next chapter.

Furthermore, the compiler adds redzones (additional memory marked as invalid, or *poisoned*, in the shadow area) after global and local variables, so when an overflow happens, it's immediately reported by the KASan. Such redzones are also inserted by kernel memory allocators (or by `malloc`, as far as ASan is concerned) after each allocated block.

Use-after-free bugs are detected by poisoning deallocated memory blocks until they're returned by some subsequent allocation request. A quarantine that delays reuse of freed blocks can be introduced to reduce the number of false negatives.

3.6.2 Kernel Hardware-Assisted Address Sanitizer

KHWASan [24] is a mode of KASan that relies on Top Byte Ignore (TBI) feature of ARMv8 processors. This sanitizer is based on a memory tagging approach:

- each 16 bytes of main memory can be marked with a 1-byte tag, stored in the shadow memory,
- each pointer has a tag, stored within its top byte (due to TBI, the tag is ignored while dereferencing the pointer)[5],

[5] 64-bit architectures like x86-64 or AArch64 (ARMv8) currently support a 48-bit virtual address space, so 64-bit pointers don't use their top bytes anyway.

- when heap memory is allocated: a random tag T is generated, the returned memory is marked with T in the shadow area, and T is embedded into the returned pointer,
- memory accesses are instrumented, similarly as for KASan, and an error is reported when memory tag mismatches pointer tag.

Let's compare the two sanitizers: KWHASan uses 1:16 shadow ratio instead of KASan's 1:8 to reduce memory overhead but the drawback is that each tagged memory chunk has to be aligned to 16 bytes. Moreover, the hardware-assisted tool can only detect off-by-one overflow if it crosses a 16-byte boundary. Also, because of a limited number of possible tag values (256), there's a 0.4% probability of an error being missed. However, KHWASan has a big advantage: it can detect an overflow no matter how large it is (with no redzones required), and a use-after-free no matter how long ago the memory was deallocated (with no quarantine required).

Please note that KASan does not use memory tagging, as it only keeps information about validity of each byte from the main memory.

3.6.3 Kernel Undefined Behavior Sanitizer

Many undefined behaviors (UB) in C/C++ cannot be detected using static code analysis. For instance, evaluating expression `42 / x` causes UB only if the integer `x` is equal to zero. Of course, `x`'s value may not be available during compilation. That's why a run-time tool is needed.

The KUBSan [7] is a run-time checker for undefined behaviors. When it's enabled, the compiler instruments the kernel code by adding checks that detect:

- division by zero,
- invalid bitwise shift operation (e.g. shifting by a negative number),
- signed integer overflow as a result of addition, subtraction, multiplication or negation,
- NULL pointer dereference,
- storing a value other than `true` or `false` into a boolean,
- etc.

When such error is noticed, appropriate `__ubsan_handle_*` function (that has to be implemented by the kernel's run-time) is called.

The KUBSan does not require shadow memory nor any initialization, as opposed to the other sanitizers.

3.6.4 Kernel Memory Sanitizer

The KMSan [27] [32] is a detector of uninitialized memory usage. It uses shadow memory (1:1 mapping, so for each bit of kernel memory the tool keeps one shadow bit) to keep track of uninitialized data. Shadow bit `0x0` stands for initialized (defined) and `0x1` for uninitialized (undefined).

The tool uses compiler instrumentation and a run-time library to:

- detect when an uninitialized variable is used,
- perform *shadow propagation* – which means that the state of being undefined propagates through variables, e.g. adding two variables that are marked as undefined yields an undefined result,
- track origin of uninitialized data, since a value may undergo multiple copies and transformations between allocation and usage (this greatly eases understanding an error report).

The KMSan has significantly lower overhead than the `kmemcheck` described in section 3.1. Note that due to the lack of compile-time instrumentation, `kmemcheck` intercepts memory accesses by causing them to trigger page faults.

3.6.5 Kernel Thread Sanitizer

This sanitizer [33] is a dynamic data-race detector. It tracks memory accesses, scheduler operations and usage of locking primitives to calculate a happens-before relation which is then used to detect race conditions. The algorithm is described thoroughly in [28].

The KTSan uses shadow memory to store a history of memory accesses. For each 8 bytes of kernel memory there are 32 bytes of shadow memory which contain the information about a few last accesses to these 8 bytes.

There is also an alternative data-race detector based on the same compiler instrumentation but with different run-time library: the Kernel Concurrency Sanitizer (KCSan) [25]. It has smaller performance overhead and memory overhead (as it doesn't use shadow memory) but is less precise and cannot detect some „subtle bugs, such as a missing memory barrier".

3.7 syzkaller

syzkaller is an unsupervised kernel fuzzer [34]. It generates C programs that use system calls, looking for the codes that will cause a kernel crash. Many OSes are

supported, i.a. Linux, *BSDs and Windows. The fuzzer is coverage-guided which means that it measures how much of the kernel code was executed while running the C program (using kcov [21], a tool for measuring kernel code coverage). The C program is constantly mutated to increase the coverage and detect more errors. The whole process is even more efficient when the fuzzed kernel is built with one of the sanitizers, as they make latent bugs cause an immediate crash.

There's also syzbot, a bot that constantly fuzzes a number of kernels using syzkaller and reports the results online [35]. For instance, we can see that NetBSD is being fuzzed using a regular kernel build, but also using builds with KMSan and KUBSan enabled. Furthermore, the Linux kernel is fuzzed using KASan, KMSan and KCSan builds. Over the last three years, syzbot has found approx. 3000 bugs in the Linux kernel, with approx. 700 of them still waiting to be fixed. That gives nearly 3 bugs per day.

3.8 Compile-time debug options

Last but not least, many OS kernels can be compiled with debug options that enable additional run-time checks (usually assertions that cause a kernel panic when evaluated to `false`). In a regular kernel build such checks are not that frequent due to the time and memory constraints.

Each system has its own unique debug options which can be kernel-wide, subsystem-wide or even file-wide. As an example, here are some options from the NetBSD kernel. Please note that the list is not exhaustive.

- Option `DIAGNOSTIC` [5] – adds internal consistency checks which will cause panic if corruption of internal data structures is detected. Decreases performance up to 15%.
- Option `LOCKDEBUG` [5] – adds code that detects incorrect use of locking primitives (e.g. deadlocks or when a memory being freed contains an initialised lock). Decreases performance up to a factor of three.
- Option `KMEM_SIZE` [6] – adds code to `kmem` memory allocator that compares whether the block size given in `kmem_free()` matches the block size allocated using `kmem_alloc()`. Any mismatch triggers a panic. Enabled by default in `DIAGNOSTIC` build.

The options have to be specified during compilation since they insert additional code via preprocessor directives like `#ifdef`.

Please note that all the tools mentioned in this chapter are able to detect various memory management bugs but are not always a mitigation against exploitation

of undetected errors in production builds. Techniques designed to make kernel exploitation more difficult (e.g. Kernel address space layout randomization or ARM Pointer Authentication) are not within scope of this thesis, though.

Chapter 4

Kernel Address Sanitizer in detail

Here you can find a comprehensive description of how the KASan works. The description is independent of any particular system or processor architecture. A MIPS-specific KASan implementation for the Mimiker OS is presented in chapter 5.

As the documentation for kernel sanitizers is limited, please note that the information below is mostly based on [29] and my own experiences of implementing the KASan in Mimiker, which involved: reading the code of sanitizers within the Linux [54] and the NetBSD [55] kernels and reading parts of the GCC source code related to sanitizers [56].

4.1 Shadow memory

The tool needs a shadow memory that keeps information about validity of each byte in the main memory. As the mapping is eight original bytes to one shadow byte, in order to cover the whole memory, one-eight of the virtual address space is needed. The shadow address is calculated using a formula: `(address / 8) + offset`. The offset (also called *the shadow offset*) is chosen for each architecture depending on where the shadow addresses reside within the virtual address space.

Figure 4.1 shows an example of a virtual address space layout and how its segments are mapped to one another using the aforementioned formula. The shadow area itself is mapped to a „bad" region which is made inaccessible by setting appropriate bits in the page table.

Each shadow byte contains a value which means:

- if it is equal to 0, the corresponding eight bytes are valid,
- if it is equal to k, where $1 \leq k \leq 7$, only the first k bytes are valid,

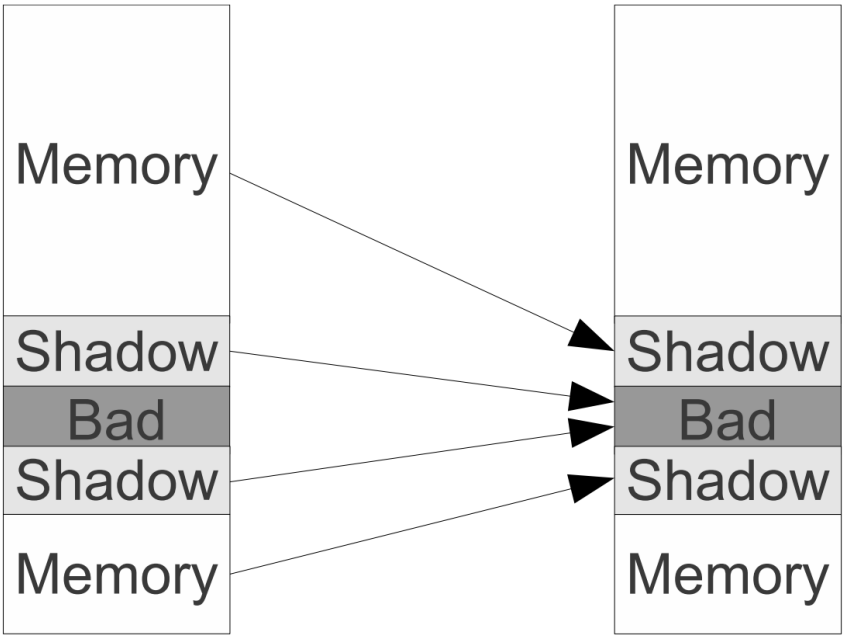

Figure 4.1: The virtual address space mapping. Diagram taken from [29].

- otherwise, the corresponding eight bytes are invalid.

In the third case, different shadow values indicate different types of poisoned memory. For instance, redzones after local variables have a unique shadow value which lets the tool distinguish between stack buffer overflow and other types of bugs.

Below you will find a pseudocode that checks validity of a 8-byte access to address `addr`:

```
shadow_addr = (addr / 8) + offset;
shadow_value = *shadow_addr;
if (shadow_value != 0)
  report_error();
```

and here's is a check for a smaller access of size `len`:

```
shadow_addr = (addr / 8) + offset;
shadow_value = *shadow_addr;
last_byte = (addr + len) mod 8;
if (shadow_value != 0 && shadow_value < last_byte)
  report_error();
```

Such check is performed before every memory access, as explained below.

4.2 Compiler instrumentation

The sanitization is enabled by passing `-fsanitize=kernel-address` flag during compilation. Then, the compiler instruments the code in many places by inserting

`__asan_*` function calls. However, the compiler doesn't deliver their implementation – it's the sanitizer's run-time library that has to implement them. Since such library contains parts that are system-specific or even architecture-specific, it must be created independently for each OS kernel.

Now let's see some examples of `__asan_*` methods and where they're inserted:

- `__asan_readN` (`__asan_writeN`) call is added before each read (write) of size `N` to check its validity,
- `__asan_register_globals` call is added to an *initialization routine* of each file that contains global variables – addresses of such routines can be found in `.ctors` object file segments and are used by the run-time library to poison redzones after global variables during kernel boot process (for more information about initialization routines in GCC, see [45]),
- `__asan_alloca_poison` call is added when the compiler allocates memory of non-compile-time size on the stack in order to create redzones for the allocated memory,
- `__asan_handle_no_return` call is added at the end of functions marked by `noreturn` specifier – in Mimiker it's used to perform a cleanup of current stack's shadow memory, but some OS kernels implement this method as no-op.

Each function that uses local variables is also instrumented. The injected code dynamically creates redzones after stack variables, but in order to poison (in function prologue) and unpoison (in function epilogue) them, the shadow offset must be known during compilation. Therefore, one has to specify it using `-fasan-shadow-offset` flag.

Moreover, memory for redzones after global variables is added statically by the compiler. For example:

```
1 int global; /* 4 bytes */
```

may be changed, depending on redzone size, to:

```
1 struct {
2   int original; /* 4 bytes */
3   char redzone[60];
4 } global;
```

Please note that the instrumentation (e.g. whether or not stack accesses are instrumented) is fully customizable via different KASan compiler flags. For more information, please refer to section 5.2.1, where changes in Mimiker's build systems are discussed.

4.3 Run-time library

The sanitizer needs a kernel library that implements the following:

- various `__asan_*` functions that the compiler inserts into the code,
- error reporting,
- interface for the rest of the kernel that allows modifying the shadow memory, mainly for allocators as they need to poison and unpoison memory blocks,
- initialization of the shadow memory, which is greatly machine-dependent (it requires, among other things, altering the page table) and should be called during early kernel boot process,
- poisoning of redzones after global variables,
- wrappers for functions implemented in assembly language – as the assembly code is not instrumented by the compiler, and some functions like `memcpy` or `copyin` [4] tend to be written in asm, the run-time library can create wrappers for them that additionally perform shadow memory checks,
- etc.

4.4 Instrumentation of allocators

Everything mentioned earlier enables the kernel to detect invalid accesses to local and global variables. With appropriate changes in kernel's memory allocators, heap overflows and use-after-free bugs will be detected as well. The allocators can be modified in many ways, but the most common one is to do the following:

- each allocation request returns memory block with an (already poisoned) redzone after the requested space – bigger redzones increase memory overhead but let the sanitizer detect larger overflows,
- deallocation of a memory block poisons the whole block, so that a use-after-free bug can be detected,
- deallocated blocks are kept in a quarantine, which means that they won't be returned upon an allocation request anytime soon (so even more use-after-free errors are detected),
- additional information like the allocation call stack or thread ID are stored inside redzones – when the sanitizer reports an invalid heap access, it can use the information to provide more details (e.g. where and by whom the memory was allocated).

Please note that even though the Kernel Address Sanitizer obviously can miss some bugs (a long-delayed use-after-free, an overflow accessing addresses beyond the redzone), it cannot report false positives.

Chapter 5

Kernel Address Sanitizer within the Mimiker OS

As mentioned in chapter 1, Mimiker is an open-source OS developed at the University of Wrocław since 2015. It currently supports MIPS32 architecture only. The Mimiker project, in which I'm personally involved since mid-2018, is led by Krystian Bacławski [66]. As the codebase has been steadily growing for the last five years, some time ago I decided that it's a good time to integrate the Kernel Address Sanitizer into the Mimiker. Not only should it find bugs already present in the kernel but also (hopefully) prevent many more from being added in the future.

In this chapter you will find how to use Mimiker's KASan, how it is implemented, and what errors it has already found. Please note that the implementation is partially based on the sanitizers within NetBSD and Linux kernel.

5.1 Example usage and bug report

All information about getting the Mimiker OS and all required packages, and then building and running it is available in a README file [68]. However, if everything is already set up, building the kernel with KASan enabled is as simple as running `make KASAN=1` (instead of `make`) command.

The system has a unit test suite that can be run using `./launch test=all` command. In order to see the sanitizer in action, let's add a buggy kernel unit test (please refer to listing 5.1) that contains a use-after-free in line 21. The memory error should be reported by the KASan when the test suite (containing the buggy test) is launched.

```
1 /* file sys/tests/kasan.c */
2 #include <sys/mimiker.h>
3 #include <sys/ktest.h>
4 #include <sys/malloc.h>
```

```
5
6  #define BUFSIZE 64
7
8  static int buggy_test_kmalloc(void) {
9    /* allocate 64 bytes using kmalloc */
10   int8_t *ptr = kmalloc(M_TEMP, BUFSIZE, 0);
11
12   /* use the memory */
13   for (int i = 0; i < BUFSIZE; i++)
14     ptr[i] = 42;
15
16   /* free the memory */
17   kfree(M_TEMP, ptr);
18
19   /* use-after-free! */
20   for (int i = 0; i < BUFSIZE; i++)
21     assert(ptr[i] == 42);
22
23   return 0;
24 }
25
26 KTEST_ADD(buggy_test_kmalloc, buggy_test_kmalloc, 0);
```

Listing 5.1: Kernel unit test containing a use-after-free bug.

Let's run all kernel tests:

```
1  $ make KASAN=1 && ./launch -d test=all
2  [...]
3  Running test "buggy_test_kmalloc".
4  ===========KernelAddressSanitizer===========
5  ERROR:
6  * invalid access to address 0xc02d5c90
7  * read of size 1
8  * redzone code 0xff (kmalloc use-after-free)
9  =============================================
10 [TEST FAILED]
11 Failure while running multiple tests.
12   buggy_test_kmalloc  <---- FAILED
```

Listing 5.2: Example KASan report.

We can see a KASan report which states the error details. Specifying `-d` flag to our `launch` script enables debugging via `gdb`, which now can be used to see the exact location of the error.

```
1  (gdb) backtrace
2  #0  panic_fail () at sys/kern/assert.c:6
3  #1  0xc0133ed0 in ktest_failure () at sys/kern/ktest.c:65
4  #2  0xc010ddc4 in shadow_check (size=0x1, addr=0xc0010c78) at sys/↩
       kern/kasan.c:172
5  #3  __asan_load1_noabort (addr=0xc0010c78) at sys/kern/kasan.c:239
6  #4  0xc014e134 in buggy_test_kmalloc () at sys/tests/kasan.c:21
```

```
#5  0xc0134038 in run_test () at sys/kern/ktest.c:117
#6  0xc01343fc in run_all_tests () at sys/kern/ktest.c:193
#7  0xc013466c in ktest_main () at sys/kern/ktest.c:251
#8  0xc0133aac in kmain () at sys/kern/main.c:20
#9  0xc0121170 in thread_self () at sys/kern/thread.c:122
```

Listing 5.3: Example `gdb` backtrace (simplified) containing the error's location.

Listing 5.3 shows a result of running `backtrace` command within the `gdb` debugger. The backtrace (also called stack trace) is a report that describes all stack frames that were active when the KASan reported an invalid memory access. We can see that the running thread was executing kernel tests (frames 5-7) and crashed in `buggy_test_kmalloc` function (frame 4, file `sys/tests/kasan.c`, line 21). That's correct!

For a list of actual bugs found by the sanitizer, please refer to section 5.4.

5.2 Implementation details

The implementation required modifying or creating over 20 files, with approximately 700 new lines of code added to the kernel. As it would be impossible to fully describe it here, in this section I present some key parts. Please note that:

- all referenced files can be found online at Mimiker's GitHub repository [68] or OpenGrok [69],
- all GitHub pull requests that added KASan to Mimiker are marked with „KASAN" milestone and can be found using this link: [70].

5.2.1 Changes in the build system

Mimiker OS is compiled using GNU Make. Mimiker's build system is rather complicated since the OS consists of many files and directories. Moreover, different rules and flags apply for kernel code and for user code (currently the OS provides several user-space programs like the `ksh` shell). The build system also supports choosing a target architecture – it can be either MIPS32 or ARM64 – but the implementation of Mimiker for ARM64 is still a work in progress.

In order to add the KASan to the kernel, the build system had to be modified. The most important change was adding new rules to `build/arch.mips.mk` file (see listing 5.4). The rules can be described in the following way:

- All kernel files are now compiled with additional `-DKASAN=1` or `-DKASAN=0` flag, depending on whether the sanitizer is enabled. This enables us to use `#if KASAN` directive inside source code. It's needed, for instance, during kernel

boot process: if KASan is turned on, we need to add additional entries to the page table that describe the shadow memory. Otherwise, the memory wouldn't be accessible.

- Files that are sanitized (i.e. all files except the run-time library and a file that e.g. initializes the page table) receive additional compiler flags defined within `CFLAGS_KASAN` variable. These flags enable sanitization in `kernel-address` mode, enable instrumentation of specific memory access types (global, stack etc.) and set shadow memory offset.

```
# Set KASAN flags
ifeq ($(KERNEL), 1)
KASAN ?= 0
ifeq ($(KASAN), 1)
  # Added to files that are sanitized
  CFLAGS_KASAN = -fsanitize=kernel-address \
                 -fasan-shadow-offset=0xD8000000 \
                 --param asan-globals=1 \
                 --param asan-stack=1 \
                 --param asan-instrument-allocas=1
endif
# Added to all files
CFLAGS += -DKASAN=$(KASAN)
endif
```

Listing 5.4: New build rules added to `build/arch.mips.mk` file.

5.2.2 Shadow memory initialization

In order for the Kernel Address Sanitizer to work properly, the shadow memory has to be initialized. More specifically, the kernel has to:

1. allocate physical memory for the shadow area,
2. map virtual shadow addresses to the allocated physical memory by adding appropriate entries in the page table,
3. zero (unpoison) the whole shadow area.

Only after such process can the KASan be turned on, so it should happen as soon as possible.

Let's focus on items 1. and 2. – the most difficult ones. In Mimiker, they are implemented in the first C function ever called from the assembly, that is in `mips_init`. The function is responsible for, among other things, setting up the page table and the TLB[1]. Listing 5.5 shows a fragment of `mips_init` that initializes the

[1]Translation lookaside buffer – a cache of page table entries ([11], section 9.6.2).

shadow area. Please note that variable `pte` is an array of page table entries and `pde` is an array of page directory entries – the page table is hierarchical and has two levels. For more information about multi-level page tables, see ([11], section 9.6.3).

```
#if KASAN /* Prepare KASAN shadow mappings */
  /* Start of the virtual shadow area */
  vaddr_t va = KASAN_MD_SHADOW_START;
  /* Allocate physical memory for shadow area */
  paddr_t pa = (paddr_t)bootmem_alloc(KASAN_MD_SHADOW_SIZE);
  /* How many PDEs should we use? */
  int num_pde = KASAN_MD_SHADOW_SIZE / SUPERPAGESIZE;
  for (int i = 0; i < num_pde; i++) {
    /* Allocate a new PT */
    pte = bootmem_alloc(PAGESIZE);
    /* Create PDE-PT mapping */
    pde[PDE_INDEX(va)] = PTE_PFN((intptr_t)pte) | PTE_KERNEL;
    /* Fill all entries inside PT */
    for (int j = 0; j < PT_ENTRIES; j++) {
      pte[PTE_INDEX(va)] = PTE_PFN(pa) | PTE_KERNEL;
      va += PAGESIZE;
      pa += PAGESIZE;
    }
  }
#endif /* !KASAN */
```

Listing 5.5: A fragment of `mips_init` function, from file `sys/mips/boot.c`, that initializes the shadow memory (with additional comments).

Please notice symbols prefixed with `KASAN_MD`. They belong to a machine-dependent part of the implementation, defined in file `include/mips/kasan.h`.

5.2.3 Interface

There's the sanitizer's interface – a set of public functions that the kernel can call to interact with KASan. It basically contains:

- a function to initialize the KASan's subsystem, that has to be called after the shadow memory is initialized,
- a set of *mark* functions to modify the shadow memory,
- a set of *quarantine* functions (the quarantine will be described in section 5.2.7).

```
/*  Initialize KASAN subsystem */
void kasan_init(void);

/* Mark bytes as valid (in the shadow memory) */
void kasan_mark_valid(const void *addr, size_t size);

/* Mark bytes as invalid (in the shadow memory) */
```

```
8  void kasan_mark_invalid(const void *addr, size_t size,
9                          uint8_t code);
10
11 /* Mark first 'size' bytes as valid (in the shadow memory) and the
12  * remaining (size_with_redzone - size) bytes as invalid with
13  * a given code. */
14 void kasan_mark(const void *addr, size_t size,
15                 size_t size_with_redzone, uint8_t code);
16
17 /* Initialize given quarantine structure */
18 void kasan_quar_init(quar_t *q, void *pool, quar_free_t free);
19
20 /* Add an item to a quarantine */
21 void kasan_quar_additem(quar_t *q, void *ptr);
22
23 /* Release all items from a quarantine */
24 void kasan_quar_releaseall(quar_t *q);
```

Listing 5.6: The KASan's interface, a fragment of `include/sys/kasan.h` file.

The functions are mostly used by heap allocators (to create redzones or quarantine memory blocks) and by the run-time itself. If the kernel is built without sanitization, all methods from listing 5.6 are defined as no-op. Please note that `kasan_mark`'s signature was inspired by its NetBSD's counterpart.

5.2.4 Linker script

As described in chapter 4, the compiler instrumentation uses initialization routines (also called *constructors*) to make poisoning of global redzones feasible. Each object file gets a initialization routine that calls run-time `__asan_register_globals` method, passing information about the file's globals and redzones that need to be poisoned. The address of such constructor is then stored in `.ctors` section of each object file.

In order to have access to the constructors and call them during KASan initialization, the kernel linker script [47] had to be modified. While linking object files into a single kernel executable, all `.ctors` sections should be gathered together and made accessible to the C code. This was achieved by adding new linker rules (listing 5.7, lines 7-10) to Mimiker's linker script. Now the constructors are placed in kernel's `.text` section, visible between symbols `__CTOR_LIST__` and `__CTOR_END__`.

```
1 .text 0xc0102000 : AT(0x102000) ALIGN(4096)
2 {
3   __kernel_start = ABSOLUTE(.);
4   __text = ABSOLUTE(.);
5   *(.text .text.*)
6   __etext = ABSOLUTE(.);
7   /* Constructors are used by KASAN to initialize global redzones */
8   __CTOR_LIST__ = ABSOLUTE(.);
```

```
9    *(.ctors)
10   __CTOR_END__ = ABSOLUTE(.);
11 } : text
```

Listing 5.7: A fragment of the kernel linker script from file `sys/mips/malta.ld`. Lines 7-10 were added.

And now calling all constructors is as simple as:

```
1 for (uintptr_t *ptr = &__CTOR_LIST__; ptr != &__CTOR_END__; ptr++) {
2   void (*func)(void) = (void (*)(void))(*ptr);
3   (*func)();
4 }
```

Listing 5.8: A fragment of function `call_ctors` from file `sys/kern/kasan.c`.

5.2.5 Run-time library

The run-time library, located at `sys/kern/kasan.c` path, contains what was mentioned in section 4.3. It has over 300 lines of code, with an implementation of: all `__asan_*` methods, the interface for the rest of the kernel, error reporting, wrappers for several assembly functions etc.

For instance, here you can see the sanitizer initialization that is called from method `platform_init` while booting the kernel, right after the shadow memory is set up:

```
1 void kasan_init(void) {
2   /* Set entire shadow memory to zero */
3   kasan_mark_valid((const void *)KASAN_MD_SANITIZED_START,
4                    KASAN_MD_SANITIZED_SIZE);
5   /* KASAN is ready to check for errors! */
6   kasan_ready = 1;
7   /* Setup redzones after global variables */
8   call_ctors();
9 }
```

Listing 5.9: Function `kasan_init` from file `sys/kern/kasan.c`.

5.2.6 Instrumentation of allocators

Currently the Mimiker's kernel provides the following allocators: `pool`, `kmalloc`, `kmem`, `bootmem`, `physmem`, `vmem`. As it may seem a lot compared to user-space (where usually one `malloc` suffices), more mature OS kernels tend to have even more. As an example, please refer to FreeBSD's allocators ([12], sections 6.2 and 6.3).

To detect memory bugs related to accessing heap memory, the allocators were modified as described in section 4.4. Not all of them were instrumented, though –

`bootmem` and `physmem` were omitted as they allocate physical memory, `vmem` was omitted as it's a general-purpose allocator, not only for memory. Also, `kmem` detects use-after-free bugs only – overflows aren't tracked as the redzones would have to occupy 4 KiB per each allocation (`kmem` has page granularity). However, both `pool` and `kmalloc` were fully instrumented.

The modified allocators can be found in the following files:

- `sys/kern/malloc.c`,
- `sys/kern/pool.c`,
- `sys/kern/kmem.c`.

5.2.7 Heap quarantine

Quarantine was introduced to make finding use-after-free bugs easier – memory blocks after deallocation won't be immediately given to another request of allocation. Strictly speaking, quarantine is a structure added to each pool of heap items, with the following content:

```
typedef struct {
  struct {
    void *items[KASAN_QUAR_BUFSIZE];
    int head;          /* first unoccupied slot */
    int tail;          /* last occupied slot */
    int count;         /* number of occupied slots */
  } q_buf;             /* cyclic buffer of items */
  quar_free_t q_free; /* function to free items after quarantine */
  void *q_pool;        /* pool from which the items come */
} quar_t;
```

Listing 5.10: Quarantine structure defined in `include/sys/kasan.h` file.

The structure contains `q_buf` – a cyclic buffer of pointers to quarantined objects, `q_free` – a pointer to function to free unquarantined object (e.g. `_kfree`), and `q_pool` – a pointer to the heap pool that the quarantine is related to.

Let's look at an example usage: implementation of `kfree` (`kmalloc`'s procedure to free given memory block) function.

```
void kfree(kmalloc_pool_t *mp, void *addr) {
  if (addr == 0)
    return;
  /* lock kmalloc's pool mutex */
  SCOPED_MTX_LOCK(&mp->mp_lock);
  /* mark the memory block as invalid in the shadow memory */
  kasan_mark_invalid(addr, abs(addr_to_mem_block(addr)->mb_size),
                     KASAN_CODE_KMALLOC_FREED);
  /* add the memory block to the quarantine */
```

```
10    kasan_quar_additem(&mp->mp_quarantine, addr);
11  #if !KASAN
12    /* without KASAN, call regular free method */
13    _kfree(mp, addr);
14  #endif /* !KASAN */
15  }
```

Listing 5.11: Implementation of `kfree` function, from file `sys/kern/malloc.c` (with additional comments).

Function `kfree` is just a quarantine wrapper, whereas `_kfree` is the regular deallocation function. When `kmalloc`'s quarantine runs out of space, it uses `_kfree` (which is stored in `quar_t::q_free`) to free least recently added object.

Please note that file `sys/kern/kasan_quar.c` contains the implementation of all `kasan_quar_*` methods.

5.2.8 Moving kernel stack to virtual addresses

Last but not least, this section describes what else I was required to do in order to enable sanitization of stack accesses.

The MIPS32 virtual address space is divided into five fragments: one user's `useg`, from `0x0000.0000` to `0x7FFF.FFFF`, and four kernel's `kseg0-kseg3`, cumulatively from `0x8000.0000` to `0xFFFF.FFFF` ([15], section 4.3). Kernel segments differ in whether or not they're mapped (translated through the TLB) or cached. All in all, the kernel should mainly use mapped and cached segments. However, unmapped ones can prove useful during early stages of OS development (note that they provide a window directly into the physical memory, so they're not really virtual).

When I started integrating the KASan into the Mimiker OS, the Mimiker's kernel used `kseg2` segment (mapped & cached) most of the time, but the kernel's stack still resided in `kseg0` (unmapped & cached). Due to memory constraints, only `kseg2` has its corresponding shadow memory, so only memory accesses to that segment are sanitized by the KASan. Therefore, in order for the stack accesses to be sanitized, which is required to detect stack overflows, I had to move the kernel stack to the mapped `kseg2` segment.

The main challenge was that accesses to stack can now trigger TLB Refill Exception – a hardware exception that happens when requested virtual address is not present in the TLB, so the kernel needs to find the corresponding virtual-physical mapping in the page table and refill the TLB ([15], section 6.2.10). An occurrence of such exception was problematic in some assembly procedures that save (restore) CPU and FPU context onto (from) the stack[2], so I had to partially rewrite these

[2]TLB Refill Exception overwrites several MIPS registers (e.g. `Status`, `Cause`, `EPC`, `BadVAddr`) which is troublesome if it occurs in a low-level procedure that tries to save the original values of these registers.

procedures. Full code change that moved the kernel stack from `kseg0` to `kseg2` can be found here: [72].

5.3 Overhead

Unfortunately, Mimiker does not currently support any dedicated benchmarking tools. However, there are over 100 kernel tests, some of them entering the user-space, and launching them sequentially can be used to approximate overall system performance and memory usage. Please note that the tests are always run in a random order so the measurements can vary to some extent between runs.

The average time of running all tests is 52 seconds without KASan and 100 seconds with KASan enabled, which gives a 92% slowdown. This overhead is comparable to up to 100% slowdown reported for KASan in NetBSD [59] and 73% for user-space ASan [29].

As far as the memory overhead is concerned, there are several factors that should be taken into account:

- the shadow memory, which occupies 16 MiB – it allows to sanitize 128 MiB (8×16 MiB) of virtual addresses, a lot more than the kernel uses and will use in a near future,
- global variables, which due to the redzones grew from 56 KiB to 127 KiB ($2.3\times$),
- peak heap usage, which due to the redzones and quarantine grew on average from 32 KiB to 86 KiB ($2.7\times$) for `pool` allocator and from 59 KiB to 198 KiB ($3.4\times$) for `kmalloc` allocator,
- stack usage, as local variables also have their redzones, but currently the kernel has no straightforward way to measure it, so I haven't done it.

Such overhead is definitely noticeable but not to an extent that would make the kernel unusable.

5.4 Bugs found so far

Here is a list of bugs that the Kernel Address Sanitizer has reported in the existing codebase. Each has a reference to a corresponding GitHub issue or a pull request.

- Buffer overflow in `strntoul` function (kernel's standard library) [71],
- Stack buffer overflow in `videomode_write` function (video driver) [74],

- Stack use-after-return in `callout_thread` (callout subsystem) [76],
- `pool` use-after-free in `thread_reap` function (thread subsystem) [73],
- `pool` use-after-free in `device_add_resource` (device drivers infrastructure) [77],
- `kmalloc` use-after-free in `tmpfs_alloc_dirent` function (tmpfs filesystem) [75],
- `kmem` use-after-free in `ctx_set_retval` function (context management, but the bug was caused by a race condition in `kmem_free` function) [78].

Each Mimiker's code change is both reviewed and automatically verified by a continuous integration tool which runs over 100 kernel tests. This procedure finds many errors at an early stage but some bugs (like the ones listed above) may still be difficult to detect without a tool like the KASan. For instance, the use-after-free within `device_add_resource` function resulted from improper resource deallocation[3] in GT-64120 PCI controller's driver. The driver lacked removing the resource from a corresponding device's tail queue before memory deallocation, so the tail queue contained a freed entry. The long-delayed use-after-free happened while adding another entry to the tail queue, as it requires modifying *next* and *prev* pointers of nearby (already freed) entries.

All the errors have been already fixed: [75] by Jakub Urbańczyk, [76] by Krystian Bacławski, [78] by Jakub Piecuch, and the rest by me. Please refer to GitHub for more details.

[3] „Resource" here means hardware resource, such as interrupt-request line, I/O port, or I/O memory. For more information, please see [13].

Chapter 6

Conclusions

In this thesis I have discussed the memory management bugs, which are an inevitable part of every application or system. Moreover, I've described different ways of tackling them that are present in various modern OS kernels. One of such way is a tool named the Kernel Address Sanitizer, which in the past few years has been added to i.a. Linux, macOS and NetBSD.

I have also presented an implementation of the Kernel Address Sanitizer for the Mimiker OS, specifically for MIPS32 architecture. The sanitizer is able to detect non-trivial buffer overflows and uses of freed memory without introducing much overhead. It has already proved to be valuable by finding a number of bugs and thus improving the Mimiker kernel's correctness and safety, not to mention bringing the University of Wrocław's system closer to the state of the art. More vulnerabilities are expected to be found in the future as the tool has been successfully integrated into Mimiker's workflow – the kernel grows rapidly and each new code change is now required to pass all tests with KASan enabled.

Further work here includes enhancing KASan's error reports so that they contain extra information about an invalidly accessed heap block: its size, stack trace of allocation and deallocation, and the like. It can be achieved by storing appropriate metadata inside redzones. Additionally, in order to find even more memory bugs, other sanitizers and a fuzzer can be added to the Mimiker kernel.

Bibliography

[1] `MALLOC(3)`, Linux Programmer's Manual

[2] `STRDUP(3)`, Linux Programmer's Manual

[3] `STRCPY(3)`, Linux Programmer's Manual

[4] `COPY(9)`, BSD Kernel Developer's Manual

[5] `OPTIONS(4)`, NetBSD Kernel Interfaces Manual

[6] `KMEM(9)`, NetBSD Kernel Developer's Manual

[7] `KUBSAN(4)`, OpenBSD Device Drivers Manual

[8] ISO/IEC 9899:2017 (C language) standard

[9] Robert C. Seacord, *Secure Coding in C and C++*, Pearson Education, 2013

[10] Wolfgang Mauerer, *Professional Linux Kernel Architecture*, Wiley Publishing, 2008

[11] R. Bryant, D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd edition, Pearson Education, 2016

[12] M. McKusick, G. Neville-Neil, R. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd edition, Pearson Education, 2015

[13] Joseph Kong, *FreeBSD Device Drivers: A Guide for the Intrepid*, No Starch Press, 2012

[14] William Stallings, *Operating Systems: Internals and Design Principles*, 8th edition, Pearson Education, 2015

[15] MIPS Architecture For Programmers, Vol. III: MIPS32 / microMIPS32 Privileged Resource Architecture, 2015

[16] C++ reference, Undefined behavior, `https://en.cppreference.com/w/c/language/behavior`

[17] Linux Kernel Mailing List archive, *Re: [PATCH] mm: kill kmemcheck again*, `https://lkml.org/lkml/2017/9/30/142`

[18] Linux Kernel Mailing List archive, *[PATCH 4.14 079/167] kmemcheck: remove annotations*, https://lkml.org/lkml/2018/2/21/703

[19] The Linux Kernel documentation, Kernel Memory Leak Detector, https://www.kernel.org/doc/html/v4.14/dev-tools/kmemleak.html

[20] The Linux Kernel documentation, Sparse, https://www.kernel.org/doc/html/latest/dev-tools/sparse.html

[21] The Linux Kernel documentation, kcov, https://www.kernel.org/doc/html/latest/dev-tools/kcov.html

[22] Reports generated by Sparse (a static code analyser for the Linux kernel), https://kernel.ubuntu.com/~kernel-ppa/static-analysis/daily/sparse

[23] LWN (Linux Weekly News), kmemcheck, https://lwn.net/Articles/260068

[24] LWN (Linux Weekly News), khwasan: kernel hardware assisted address sanitizer, https://lwn.net/Articles/763684

[25] LWN (Linux Weekly News), Concurrency bugs should fear the big bad data-race detector, https://lwn.net/Articles/816850

[26] LWN (Linux Weekly News), Kernel address space layout randomization, https://lwn.net/Articles/569635

[27] E. Stepanov, K. Serebryany, MemorySanitizer: fast detector of C uninitialized memory use in C++, https://research.google/pubs/pub43308

[28] K. Serebryany, T. Iskhodzhanov, ThreadSanitizer – data race detection in practice, https://research.google/pubs/pub35604

[29] K. Serebryany et al., AddressSanitizer: A Fast Address Sanity Checker, https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

[30] google/sanitizers repository, Address Sanitizer Found Bugs, https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs

[31] google/kasan repository, Found Bugs, https://github.com/google/kasan/blob/master/FOUND_BUGS.md

[32] google/kmsan repository, https://github.com/google/kmsan

[33] google/ktsan repository, https://github.com/google/ktsan

[34] google/syzkaller repository, https://github.com/google/syzkaller

[35] syzbot, https://syzkaller.appspot.com

[36] Yuji Kukimoto, Introduction to Formal Verification, https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html

[37] John Franco, What is Formal Verification, Why is its Importance Increasing, and How is it Developing? http://gauss.ececs.uc.edu/Courses/c626/lectures/Intro/std.pdf

[38] The CompCert project, http://compcert.inria.fr

[39] G. Klein et al., seL4: Formal Verification of an OS Kernel, https://web1.cs.columbia.edu/~junfeng/09fa-e6998/papers/sel4.pdf

[40] GCC 4.8 release notes, https://gcc.gnu.org/gcc-4.8/changes.html

[41] GCC 10 release notes, https://gcc.gnu.org/gcc-10/changes.html

[42] Microsoft Security Response Center, Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape, BlueHat IL 2019, https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

[43] Microsoft Security Response Center, Solving Uninitialized Stack Memory on Windows, https://msrc-blog.microsoft.com/2020/05/13/solving-uninitialized-stack-memory-on-windows

[44] Software Instrumentation, Wiley Encyclopedia of Computer Science and Engineering, https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386

[45] GNU Compiler Collection (GCC) Internals, How Initialization Functions Are Handled, https://gcc.gnu.org/onlinedocs/gccint/Initialization.html#Initialization

[46] The GNU C++ Library, Debug Mode, https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode.html

[47] GNU linker `ld`, Linker Scripts, https://sourceware.org/binutils/docs/ld/Scripts.html

[48] N. Heninger and D. Stefan, CSE 127: Computer Security, Low-level mitigations, https://cseweb.ucsd.edu/classes/fa19/cse127-ab/slides/3-lowlevelmitigations.pdf

[49] N. Nethercote and J. Seward, Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, https://dl.acm.org/doi/10.1145/1250734.1250746

[50] T. Barabosch, M. Villard, KLEAK: Practical Kernel Memory Disclosure Detection, https://www.netbsd.org/gallery/presentations/maxv/kleak.pdf

[51] B. Chen et al., Fuzzing OpenSSL, https://courses.csail.mit.edu/6.857/2019/project/11-Chen-Kim-Lam.pdf

[52] The Linux kernel 5.0 source code, search __user, https://elixir.bootlin.com/linux/v5.0/A/ident/__user

[53] The Linux kernel 5.0 source code, search __releases, https://elixir.bootlin.com/linux/v5.0/A/ident/__releases

[54] The Linux kernel 5.0 source code, file mm/kasan/generic.c, https://elixir.bootlin.com/linux/v5.0/source/mm/kasan/generic.c

[55] The NetBSD kernel source code, file sys/kern/subr_asan.c, https://nxr.netbsd.org/xref/src/sys/kern/subr_asan.c

[56] The GCC source code, directory libsanitizer/asan, https://github.com/gcc-mirror/gcc/tree/master/libsanitizer/asan

[57] The Linux 4.0 release notes, https://kernelnewbies.org/Linux_4.0

[58] NetBSD 9.0 release notes, https://www.netbsd.org/releases/formal-9/NetBSD-9.0.html

[59] Kamil Rytarowski, Taking NetBSD kernel bug roast to the next level: Kernel Sanitizers, EuroBSDcon 2018, https://www.netbsd.org/~kamil/eurobsdcon2018_ksanitizers.html#slide7

[60] FreeBSD Wiki, SummerOfCode2019Projects, KernelSanitizers, https://wiki.freebsd.org/SummerOfCode2019Projects/KernelSanitizers

[61] Linux in 2020: 27.8 million lines of code in the kernel, https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd

[62] Exploit Education website, *Heap Two* task, https://exploit.education/protostar/heap-two

[63] Gynvael Coldwind's blog, https://gynvael.coldwind.pl

[64] Google Project Zero blog, https://googleprojectzero.blogspot.com

[65] A brief introduction to Binary Exploitation, http://mzr.re/posts/intro-to-binary-exploitation

[66] Krystian Bacławski's website, http://cahirwpz.cs.uni.wroc.pl

[67] The Mimiker Project, https://mimiker.ii.uni.wroc.pl

[68] The Mimiker's GitHub repository, https://github.com/cahirwpz/mimiker

[69] The Mimiker's OpenGrok, https://mimiker.ii.uni.wroc.pl/source/xref/mimiker

[70] The Mimiker's „KASAN" milestone, `https://github.com/cahirwpz/mimiker/milestone/4?closed=1`

[71] cahirwpz/mimiker repository issue #635: *strntoul's implementation shouldn't read more than n bytes*, `https://github.com/cahirwpz/mimiker/issues/635`

[72] cahirwpz/mimiker repository pull request #636: *Access kernel stack using only KSEG2 addresses*, `https://github.com/cahirwpz/mimiker/pull/636`

[73] cahirwpz/mimiker repository pull request #645: *Fix use-after-free inside thread_reap*, `https://github.com/cahirwpz/mimiker/pull/645`

[74] cahirwpz/mimiker repository pull request #647: *Fix stack-buffer-overflow in videomode_write*, `https://github.com/cahirwpz/mimiker/pull/647`

[75] cahirwpz/mimiker repository issue #653: *Use-after-free in tmpfs_alloc_dirent*, `https://github.com/cahirwpz/mimiker/issues/653`

[76] cahirwpz/mimiker repository issue #678: *callout_thread uses freed stack memory*, `https://github.com/cahirwpz/mimiker/issues/678`

[77] cahirwpz/mimiker repository issue #685: *Use-after-free in device_add_resource*, `https://github.com/cahirwpz/mimiker/issues/685`

[78] cahirwpz/mimiker repository pull request #705: *Fix race between kmem_free() and kmem_alloc()*, `https://github.com/cahirwpz/mimiker/pull/705`