ARM Memory Tagging Extension

Fighting Memory Unsafety with Hardware

Akademy, 2021.

Alexander Saoutkin

a.saoutkin@gmail.com
Feverfew (IRC/Matrix/GitHub)

MEMORY UNSAFETY IN C/C++

Spatial Memory Safety Bugs

Temporal Memory Safety Bugs

Why is Memory Unsafety a Problem?

C/C++ is memory unsafe. That is, one can directly access and manipulate pointers to memory freely.

Useful if we want to avoid the overheads of automatic memory management (GC).

What are the downsides?

1. Cognitive overhead for programmer to manage memory manually.
2. If done incorrectly, invokes undefined behaviour!

Why is Undefined Behaviour a Problem?

What is undefined behaviour?

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

What are the implications?

1. Upon invocation of UB, anything can happen!
2. User could experience logical bugs, segmentation faults etc.

Why have UB?

1. Avoids the need for expensive run-time checks.
2. Compiler writer can simplify design of compiler.

Is it a Big Problem?

Yes! Causes countless bugs, many of which have security implications:

1. Morris Worm (1998) - took down large chunks of internet, $10,000,000 in damage.
2. Heartbleed - OpenSSL bug affecting 24%-55% of popular HTTPS sites.
3. Chromium - 70% of serious security bugs can be attributed to memory safety problems.

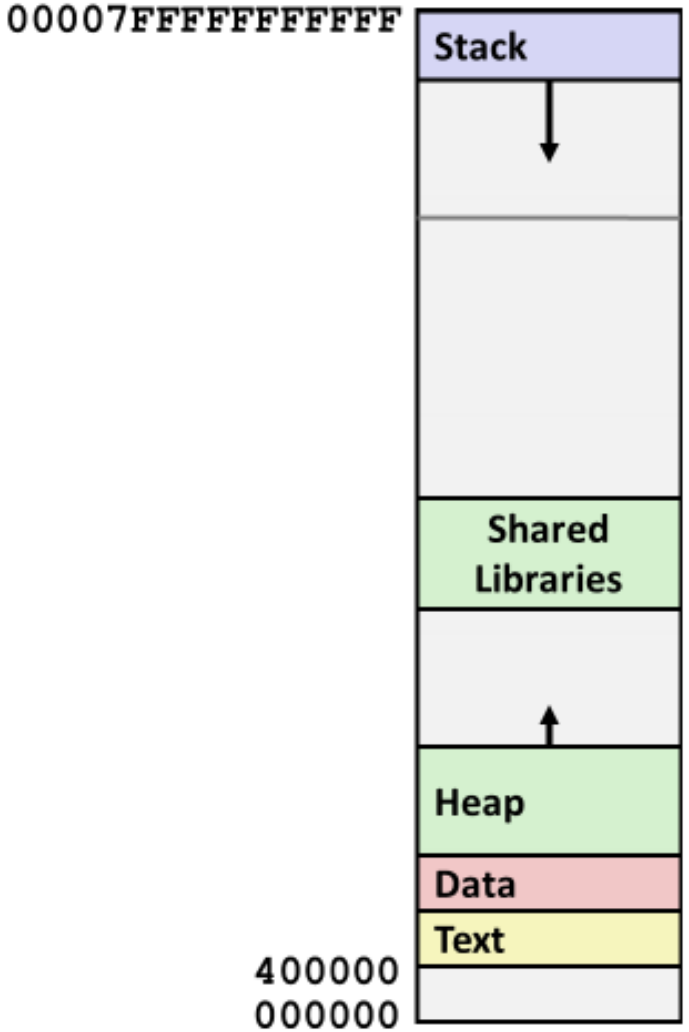
C/C++ programs very common!

1. Apache and Nginx - 66% of web server market.
2. Google Chrome - 64% of web browser market.
3. Windows - 75% of desktop market.
4. Android - 72% of mobile market.

Why Should KDE Care?

- Our software is based on an ecosystem of other C/C++ software (both kernel and user-space).
- Our software is written in C/C++ and we link against other C/C++ libraries, most notably, Qt.
- Developers constantly deal with crash reports, which are often triaged as VHI.
- Prevalance of memory safety bugs in our programs deters users away!

Virtual Address Space Layout



Stack Overflows

A function vulnerable to Stack Overflow:

```
void fillBuf(int a, int b) {
    char buf[100];
    scanf(”%s”, buf);
}
```

The program allocates a buffer of 100 bytes.

But scanf does no bounds checking of the input - assumes that input will fit in buffer.

If this is not true then we have a stack overflow! What are the implications of this?

How to Exploit a Stack Overflow?

Goal of attacker: insert input (shellcode) that allows me to take control of program.

input b
input a
return address
saved frame pointer
local variable buf

input b
input a
overwritten return address
shellcode
shellcode

1. Overwrite return address to point to shellcode.
2. Need to correctly guess where return address is? No, just write it many times.
3. Need to correctly guess where shellcode starts? No, use nop instruction many times.

Also require that buffer is large enough for shellcode!

How to Mitigate a Stack Overflow?

Write-or-execute pages:

- Intuition: Why is stack executable?
- Workaround: return-to-libc. e.g. system("\bin\sh").

Address Space Layout Randomisation (ASLR):

- Intuition: Randomise address space, hard to guess return address!
- Workaround: 32-bit not enough entropy. 64-bit - Hacking Blind.

This is just a small snippet of mitigations, many more exist. Nonetheless, software still commonly exploitable!

Heap Overflows

```
struct data {
    char name[64];
};
struct fp {
    int (*fp)() ;
};
int main(int argc, char **argv) {
    struct data *d;
    struct fp *f;
    d = malloc(sizeof(struct data));
    f = malloc(sizeof(struct fp));
    f->fp = exit;
    strcpy(d->name, argv[1]);
    f->fp();
}
```

This relies on heap being executable!

Use-After-Free Vulnerabilities

Example of use-after-free:

```
int main() {
    int *a;
    a = malloc(sizeof(int));
    *a = 42;
    free(a);
    *a = 24;
    return 0;
}
```

This example is for illustration, not exploitable by itself.

How to Exploit Use-After-Free?

At this point, the memory has been freed but we still have a dangling pointer to it.

Consider the following scenario:

- Program assumes data contains function pointer.
- Attacker is able to input return address in new allocation.
- Program then uses dangling pointer, which now points to attacker controlled data.

Seems a bit contrived… is this a big issue?

Yes, half of high severity memory safety-related security bugs in Chromium are use-after-free vulnerabilities!

Use-After-Return Vulnerabilities

Similar to use-after-free but on the stack.

```
volatile int *p = 0;
// Use-after-return
int * func() {
    int x = 0;
    return &x;
}
int main() {
    p = func();
    *p = 5;
    return 0;
}
```

Use-After-Scope Vulnerabilities

```
volatile int *p = 0;
// Use-after-scope
int main () {
{
    int x = 0;
    p = &x;
}
*p = 5;
return 0;
}
```

If attacker can place it's own data where the stack variable is located, then one can use a similar technique to use-after-free exploits to exploit the above two vulnerabilities.

MEMORY ERROR DETECTORS

ASAN

HWASAN

ARM MTE

Heap Tagging

Stack Tagging

What About Prevention?

We've now looked at several vulnerabilities and discussed how to *mitigate* them.

Can we potentially *prevent* them appearing in production software in the first place?

In this section, we'll discuss such tools and evaluate their effectiveness, which will motivate the existence of ARM MTE.

What is ASAN?

Address Sanitizer (ASAN) consists of compiler instrumentation and a run-time library that detects out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free bugs.

Easy to use and supported by Clang, GCC and MSVC:

```
cmake -DECM_ENABLE_SANITIZERS='address'
```

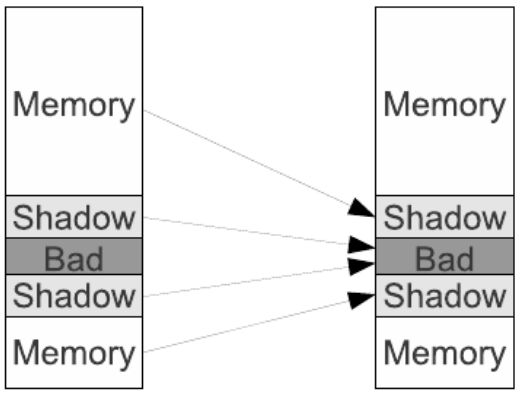
Aimed at preventing memory safety bugs entering production.

Upon detection of bug aborts program with useful debugging information.

How Does ASAN Work?

To detect spatial memory bugs, employs use of *redzone* around allocations.

To detect temporal memory bugs, employs use of *quarantine* around allocations.



Shadow memory stores metadata (i.e., addressable? redzone?) in one byte for each 8 bytes of memory.

Evaluation of ASAN

At time of release in 2011 dramatically reduced overheads compared to other tools such as Valgrind.

Valgrind has 10x run-time overhead vs ASAN's 73%. ASAN had 3.4x increased memory usage. Overheads tolerable for debugging, not so much for in production use as a mitigation.

Commonly used by both Chrome and Mozilla to find bugs. Found 300 bugs in Chromium code base in first 10 months.

However, it is a dynamic analysis tool - analysis only as effective as test suite. Not to worry! Combine with fuzzing and reap the rewards!

What is HWASAN?

Hardware Address Sanitizer (HWASAN) Consists of compiler instrumentation and a run-time library that detects a similar class of bugs to ASAN. Relies on ARM TBI hardware feature to store metadata known as *tags*, which is used to perform *memory tagging*.

Easy to use but only supported by Clang and only supported on ARM (for now, maybe Intel soon. See Intel LAM):

```
make CFLAGS+='-fsanitize=hwaddress -fno-omit-frame-pointer' \
LDFLAGS+='-fsanitize=hwaddress'
```

What is ARM TBI?

Registers are 64-bit on AARCH64 but virtual addresses only 48 bits. Currently other 16 bits are set to 0xFFFF or 0x0.

ARM TBI relaxes requirement and allows top byte to store any value. In this case, HWASAN will store a tag associated with a pointer.

Is this a good idea? Intel has already introduced 5-level paging (i.e., 57 bit address space)!

How Does Memory Tagging Work?

First define two key words:

- *Tagging Granularity* - amount of memory associated with any given tag (e.g., 16 bytes).
- *Tagging Size* - possible size of a given tag (e.g. 1 byte).

Each granule of memory is associated with a tag and all pointers to that same location in memory should have the same tag.

Upon each memory access:

1. Compare tag of pointer with tag of memory it points to.
2. If they don't match, abort!

How does this catch memory safety bugs?

How Does HWASAN Work?

Just like ASAN, still requires compiler instrumentation to check each memory access. Requires storage of tags.

Modify memory allocator for heap tagging:

- On allocation, align to TG and randomly assign a tag to memory and pointer.
- On de-allocation, assign a new random tag to memory.

Modify compiler for stack tagging:

- Align all local variables to TG and assign a tag to memory and pointer.
- On function exit, assign a new random tag to memory.

For the stack, a single base tag is used and all other tags are derived from it.

Have 1/256 probability of missing memory safety bugs.

Evaluation of HWASAN

In what way is HWASAN better than ASAN?

1. Smaller RAM overhead (10-35%). Just need storage for tags and alignment.
2. Better at detecting non-adjacent out-of-bounds access.
3. Better at detecting use-after-free over time.

In what ways is ASAN better?

1. Bug detection is deterministic for ASAN but only probabilistic for HWASAN.
2. Can’t detect overflows within the granule.

What is ARM MTE?

ARM MTE is a new hardware feature, that is currently not in hardware. Otherwise, available on QEMU.

An implementation of the memory tagging concept with tagging granularity of 16 bytes and tagging size of 4 bits located in top-byte via ARM TBI.

Mapping of granules to tag values is stored in memory. New instructions to generate, manipulate and view tags.

Two modes of usage (assumed memory mapped correctly):

1. Synchronous - Tag checked on load and store. Segmentation fault generated with faulting address.
2. Asynchronous - Tag checking delayed until next context switch. Segmentation fault generated without faulting address.

Heap Tagging with glibc

Implementation very similar to HWASAN! Of note, is that a tag value (0) is reserved for internal data structures. The TG alignment is the same as the current alignment required by glibc.

- On allocation and deallocation, align to TG and randomly assign a tag to memory and pointer.
- On de-allocation, assign a new random tag to memory.
- calloc is no more expensive than malloc due to stzgm instruction.
- realloc always retags!

Stack Tagging with Clang

To tag stack variables requires changes in the code generation in the backend of the compiler.

Tagging stack variables is implemented as a simple function pass.

Similarly to heap-tagging, each allocation must be 16-byte aligned!

Stack Safety Analysis can be employed to determine which stack variables are safe at compile-time, which means tagging is unnecessary.

Stack Tagging a Single Variable

C Source	Original Assembly	Assembly with Stack Tagging

`void f() {`
`  int x = 42;`
`  use(&x);`
`}`
	`str x30, [sp, #-16]!`
	
	`mov w8, #42`
	`add x0, sp, #12`
	`str w8, [sp, #12]`
	`bl use`
	
	`ldr x30, [sp], #16`
	`ret`
		`sub sp, sp #32`
		`str x30, [sp, #16]`
		
		`irg x0, sp`
		`mov w8, #42`
		`stgp x8, xzr, [x0]`
		
		`bl use`
		`stg sp, [sp], #16`
		`ldr x30, [sp], #16`
		`ret`

Stack Tagging Multiple Variables

C Source	Original Assembly	Assembly with Stack Tagging

`void f() {`
`  int a, b, c;`
`  use(&a);`
`  use(&b);`
`  use(&c);`
`}`
	`add x0, sp, #12`
	`bl use`
	`add x0, sp, #8`
	`bl use`
	`add x0, sp, #4`
	`bl use`
		`irg x19, sp`
		`addg x0, x19, #32, #1`
		`bl use`
		`addg x0, x19, #32, #1`
		`bl use`
		`mov x0, x19`

Evaluation of ARM MTE

In what way is ARM MTE better than HWASAN?

1. Smaller CPU overhead. Tag checking done in hardware! Hopefully low enough (<5%) to use as mitigation in production!
2. Smaller RAM overhead (3-5%). Just need storage for tags and alignment for stack variables.
3. Smaller Code Size overhead (<5%). No need to instrument each access.
4. For heap tagging, recompilation is not required.

In what ways is HWASAN better?

1. Tagging size is smaller. Roughly 7% chance of missing out on bug!

UNCONVENTIONAL USE CASES

Infinite Hardware Watchpoints

Data Race Detection

Watchpoints with ARM MTE

With help of ptrace, one can manipulate another process including its tags. The following algorithm implements a watchpoint with ARM MTE:

- Ensure the process is running with synchronous tagging enabled and that memory is being mapped correctly.
- For a given region in memory, find it’s tag(s) and change it to another random value.
- If granule is accessed tag mismatch will occur as pointer has wrong value.
- Use combination of faulting address and instruction pointer (and disassembly) to determine if access in region of interest.

What is a Data Race?

Two operations that access main memory are called conflicting if the physical memory they access is not disjoint, at least one of them is a write and they are not both synchronised accesses. A program has a data race if it can be executed on a multiprocessor in such a way that two conflicting memory accesses are performed simultaneously (by processors or any other device).

Data races in multithreaded programs are notoriously hard to debug and reproduce so developing techniques to find them is important!

DataCollider

DataCollider is a data race detector by Microsoft Research which uses hardware watchpoints to find data races.

Algorithm is as follows.

1. Set code breakpoints uniformly over all instructions that access memory.
2. Upon code breakpoint employ two strategies to spot data races.
3. *repeated read* - read current value and after short delay, read again and see if value has changed.
4. *watchpoint* - set watchpoint on region and after short delay, see if it has been triggered.

Very low overhead, which is unusual for a data race detector!

Infinite Watchpoints

With low number of watchpoints, what if we could have more? If we have infinite watchpoints with ARM MTE we can set as many code breakpoints as we want, that is, increasing the sampling rate.

We now have a trade-off between the overhead induced on the program and the sampling rate.

Without the actual hardware it's hard to see how the trade-off will turn out. But this will be an interesting question to explore in the future!