A study on the overhead of memory-tagging in compression libraries

Edouard Michelin

Introduction

Memory-unsafe languages are still part of our lives

Linux, Chromium, Firefox, Nginx, and Git are all written in memory-unsafe languages (C, C++)

Importance of memory vulnerabilities

- ➔ ~70% of the vulnerabilities Microsoft assigns a CVE each year are memory safety issues

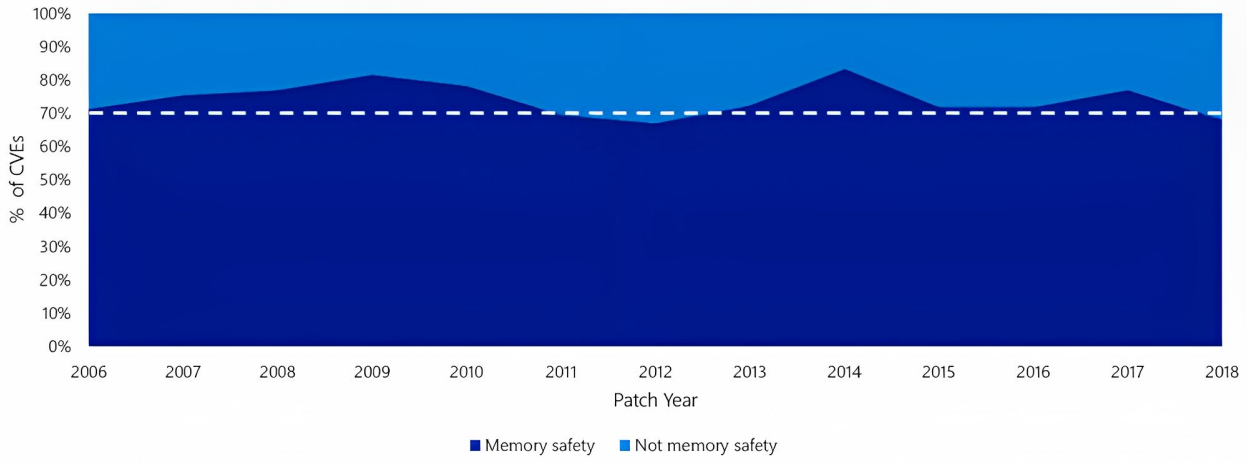
  Similar results were found by Google for Chromium and Android in 2018 and 2019
- ➔ Consequences can be devastating if not contained (e.g., data corruption)

Possible mitigation strategies

- ➔ Sandboxing, isolating components from each other at the process level
  - ◆ Problem: costly, sometimes too coarse-grained
- ➔ Intra-process memory isolation (`mprotect()`, Intel MPK)
- ➔ What can we protect? What can we not protect? Example of CVE-2018-25032 (memory corruption when deflating) and a synthetic one

Intel MPK (or PKU)

Relatively new (2017) → still needs to be studied
We will focus on its impact on performance



Microsoft Security Response Center (MSRC) — A proactive approach to more secure code

Background

The `mprotect()` way

Granularity = 1 page
System call for every access restriction modification
➔ inefficient when called multiple times

The MPK way

Memory-Tagging technique:
➔ Tag granularity = 1 page
➔ Tag size = 4 bits (→ 16 keys)
➔ Access restrictions set with PKRU register

`mprotect()` vs MPK

➔ MPK is thread-local
➔ Number of syscalls with `mprotect()` -way is linear in the number of permission change, MPK-way is constant

Design & Implementation

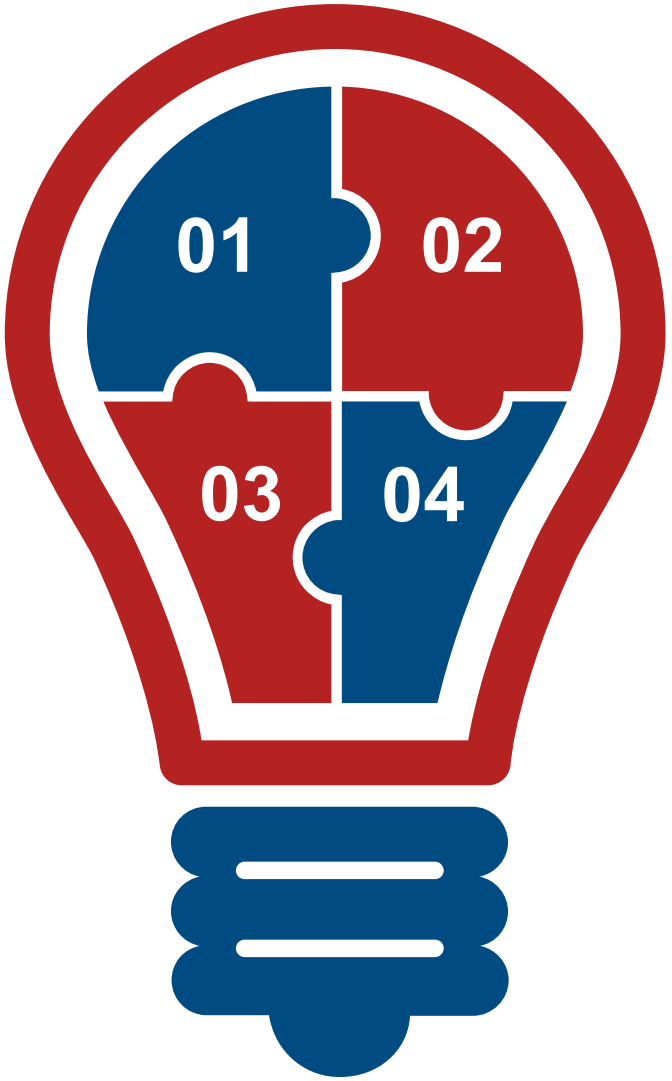
Design

Two memory regions

1. `SAFE` — Used by the main application
2. `SHARED` — Used mainly by Zlib

ZMP

- ➔ Wraps zlib APIs
- ➔ Locks `SAFE` before entering zlib, unlocks when returning from zlib
- ➔ Must offer allocation functions for the `SAFE` region
- ➔ One version with `mprotect()`, one with MPK



Access-Control Matrix

	`SAFE`	`SHARED`
Main app	RW	RW
Zlib	R or -	RW

Identifying the regions

- ➔ Zlib needs access to a structure containing the input and other meta-data for its operations (called `z_stream`)
  - read-only for some fields (e.g., input buffer)
  - read-write for some others (e.g., output buffer)
  - or everything in `SHARED` → the whole structure in read-write
- ➔ `z_stream` is allocated by the main application
- ➔ `SAFE` ideally contains everything else but can also be only the data chosen by the developer

Design & Implementation

Issues and limitations

- **Multi-threaded applications**
  - ➔ `mprotect()` is process-wide
    - ◆ If thread A enters zlib (`SAFE` is locked) then thread B can no longer access `SAFE`
- **Time**
  - ➔ ZMP provides intra-process heap-based memory isolation, **stack is not covered**
  - ➔ ZMP's custom allocator implementation is very primitive, deallocator has no effect
- **Limitations**
  - ➔ A vulnerability in zlib will not be detected by ZMP (e.g., CVE-2018-25032), the purpose of the design is to contain it

```
// main.c
int main()
{
    char *private_data = zmp_safe_alloc(100);
    z_streamp stream = malloc(sizeof(z_stream));
    // …
    int result = inflate(stream, Z_FINISH);
    // memory bug in inflate that can be used
    // to corrupt private_data
    // …
    return 0;
}
```

```
// zlib/inflate.c
int inflate()
{
    // …
    // put = output, copy = size to copy
    // put has been overwritten and now
    // points to private_data
    memcpy(put, next, copy);
    // crashes the application as private_data
    // is in locked SAFE region
}
```

Design & Implementation

Implementation

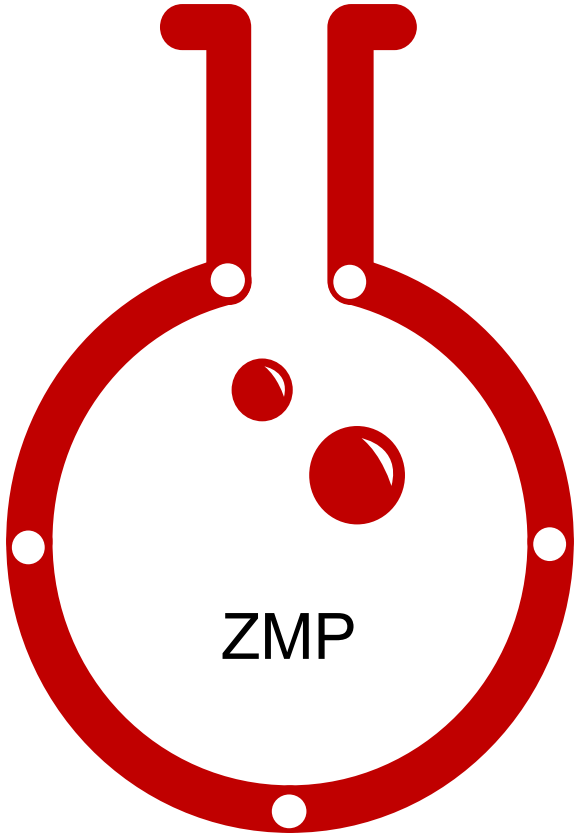
ZMP heap (interaction type 1)

```
struct zmp_heap {
    void *base_address;
    size_t size; // soft limit
    size_t used; // hard limit
    int protection_key; // MPK only
}
```

- ➔ `mmap()`-ed blocks → granularity of a block = 1 page
- ➔ blocks are contiguous, data are in addresses `[base_address, base_address + used]`

Allocations

- ➔ `void *zmp_safe_alloc(size_t size)`
  → allocates memory to the safe heap
  → if MPK: tags the allocated pages
- ➔ `void zmp_safe_free(void *pointer)`
  → frees memory allocated on the safe heap
  → currently has no actual effect
- ➔ `SHARED` allocations use standard library (i.e., `malloc()`, `free()`) as no restrictions will be applied



Wrappers (interaction type 2)

- ➔ calls to zlib are replaced by their wrappers
- ➔ have the same header as the original zlib functions (except for the name)
- ➔ manage `SAFE` region permission restrictions

```
// wrapper for int zlib_api(void *param)

int zmp_zlib_api(void *param)
{
    zmp_mem_lock(); // locks SAFE
    int result = zlib_api(param);
    zmp_mem_release(); // unlocks SAFE

    return result;
}
```

Locking mechanism differs between MPK and `mprotect()` version:

- ➔ **`mprotect()`**
  `mprotect([safe.base_addr, safe.base_addr + safe.used], PERM)`
- ➔ **MPK**
  `pkey_set(safe.protection_key, PERM)`

Design & Implementation

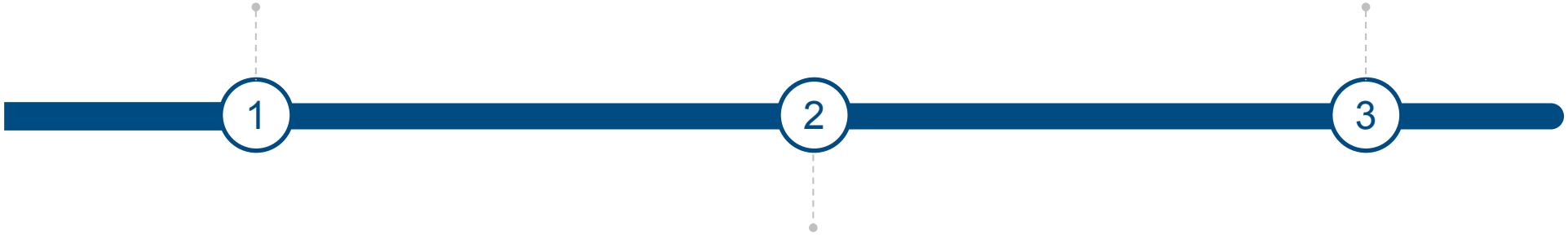
Program flow

ZMP init

1. Initialize `SAFE` heap
   If MPK →
   a. needs to allocate a protection key with the `pkey_alloc()` syscall
   b. store the key in the `protection_key` field
2. Loads zlib APIs (greedily)

Zlib usage

1. Enters ZMP's wrapper
2. `SAFE` memory locked
3. Zlib original function is called
4. `SAFE` memory unlocked
5. Returns result



Main()

1. Main program starts
2. Standard allocations are replaced by ZMP's safe allocator
3. Zlib state structure is allocated in the `SHARED` region alongside the input and output buffers

Evaluation

Evaluation setup

Minizip (minizip+miniunz)

- ➔ Small, zip and unzip tool
- ➔ Lives around zlib by construction → easier to perceive raw impact of ZMP
- ➔ Splits the input file in smaller chunks, compress/decompress the file chunk by chunk
- ➔ Program was modified:
  - ◆ `malloc()` → `zmp_safe_alloc()` except for the output buffer
  - ◆ Input buffer is in the `SAFE` region (here when locked, `SAFE` is read-only)
  - ◆ zlib state structure was allocated on the stack → allocated on the heap (`SHARED` region)

What can we test?

- ➔ Bigger input file → More chunks → More interactions with zlib → More domain switches
- ➔ Bigger safe region → More pages to lock/unlock
- ➔ Behavior of ZMP with CVE-2018-25032 and the synthetic example

Test suite

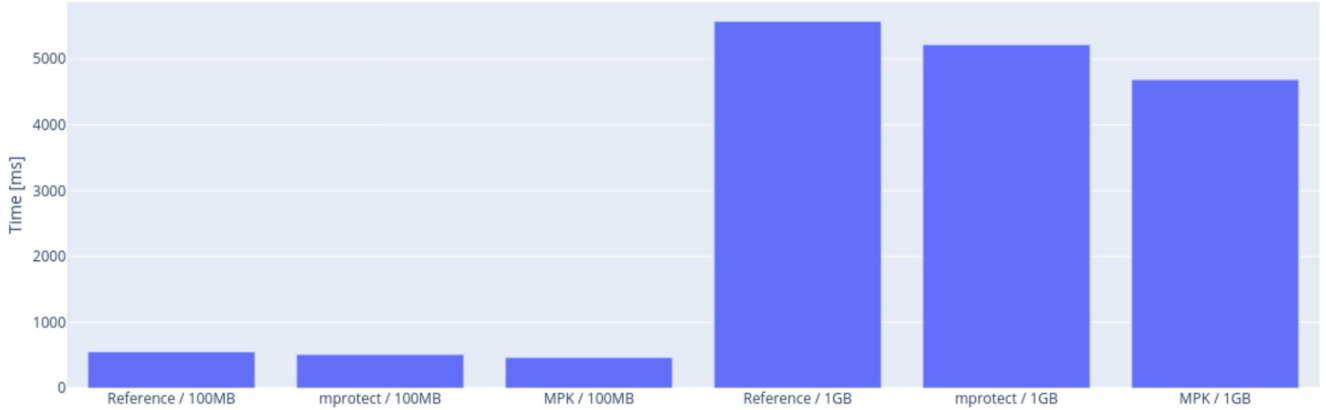
- ➔ `perf stat` as a testing tool — 100 rounds per test
- ➔ Run the following test suite:
  a. Remove the generated archive (`test.zip`)
  b. Zip the input (a file named `readme.txt`), this generates `test.zip`
  c. Rename the input (to `readme.old`)
  d. Unzip `test.zip`, this generates `readme.txt`

Evaluation

Results

Total elapsed time

- ➔ Reference version less performant (in user space)
  → implementation changes (stack vs heap, `malloc()` vs `mmap()`)
- ➔ Noticeable impact when total elapsed time is small (<3ms), on avg. ~4-5% (in user space only)
  → due to ZMP's init. having a higher weight
- ➔ overhead of `mprotect()` over MPK: 0.5% for 32kB, 10.6% for 100MB, and 11.3% for 1GB



Avg runtimes of the reference app., the mprotect and the MPK versions with input size of 100MB and 1GB

Time in kernel space

32kB

- ➔ MPK presents no overhead
- ➔ mprotect() has a 4.1% overhead

100MB and 1GB

- ➔ MPK presents no overhead
- ➔ `mprotect()` introduces a 59.3% overhead

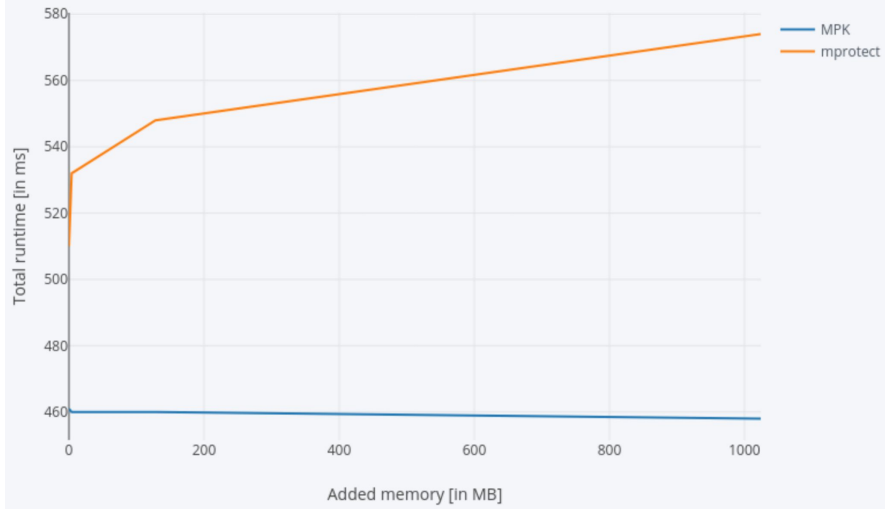
Avg time in kernel space for the ref. app., the mprotect and the MPK versions with input size of 100MB and 1GB

Memory metrics (100MB input)

	Minizip	Miniunz
`SAFE`	5.9%	28.25%
`SHARED`	94.1%	71.25%
TOTAL	354,648 B	67,224 B

Impact of a bigger `SAFE` region (for a 100MB input)

- ➔ No effect on MPK
- ➔ 13% overhead for `mprotect()` in the assessed range



Average runtimes for mprotect() and MPK with increasing allocation size in the `SAFE` region, and a 100MB input.

Evaluation

CVE-2018-25032 and a theoretical example

CVE-2018-25032

- ➔ Allows for memory corruption during deflation
- ➔ Corrupts the output archive
- ➔ Caught by AddressSanitizer
- ➔ Not detected by ZMP as in zlib
- ➔ If used to craft a more advanced attack which spreads to the `MAIN` region, could be detected

Theoretical example

```
// main.c
int main()
{
    char *private_data = zmp_safe_alloc(100);
    z_streamp stream = malloc(sizeof(z_stream));
    // …
    int result = inflate(stream, Z_FINISH);
    // memory bug in inflate that can be used
    // to corrupt private_data
    // …
    return 0;
}
```

```
// zlib/inflate.c
int inflate()
{
    // …
    // put = output, copy = size to copy
    // put has been overwritten and now
    // points to private_data
    memcpy(put, next, copy);
    // crashes the application as private_data
    // is in locked SAFE region
}
```

Discussion & Conclusion

Memory isolation with `mprotect()`

- ➔ Proven technique
- ➔ Non-negligible performance overhead
- ➔ Not very scalable

With MPK

- ➔ Highly scalable
  - ◆ In terms of number of pages to manage (size of `SAFE`)
  - ◆ In terms of the interactivity between zlib and the main application
- ➔ Negligible performance overhead in most scenarios

Portability of ZMP

- ➔ Easily portable to other compression libraries (e.g., ZStd)

Future Work

- ➔ Stack isolation
- ➔ More advanced memory allocator
- ➔ Large scale application as a test case
- ➔ A more general library (standalone or improve already existing libraries)

Summary

- ➔ Process's memory split into 2 regions, `SAFE` and `SHARED`
- ➔ When zlib is being used, the `SAFE` region is locked (either Read-Only or No-Access) in one of two fashions (`mprotect()` or MPK)
- ➔ ZMP does not protect zlib, it detects illegal accesses to the `SAFE` region that could result from vulnerabilities in zlib
- ➔ `mprotect()` cannot be used in a scalable implementation of intra-process memory isolation
- ➔ MPK is a very efficient (performance-wise) way of implementing page-based permission restrictions
  - ◆ Negligible performance overhead in most scenarios
- ➔ Memory-tagging (particularly MPK in our case) is a promising opportunity to detect memory-related vulnerabilities
- ➔ Albeit security was not a main concern in this project, it is an important factor to take into consideration
- ➔ The result of this project can serve as a foundation for further exploration and optimization