Enhancing Memory Error Detection for Large-Scale Applications and Fuzz testing

Wookhyun Han, Byunggil Joe, Byoungyoung Lee*, Chengyu Song†, Insik Shin

KAIST, *Purdue, †UCR
Memory error

- Information leakage – Heartbleed
- Privilege escalation – Shellshock
- Remote code execution – Shellshock, glibc, Conficker
Memory error detection

• **Pointer-based** [SoftBound+CETS, Intel MPX]
  • Hardware support (cannot detect temporal memory errors)
  • Challenges to support complex applications

• **Redzone-based** [AddressSanitizer (ASan)]
  • **Compatible** to complex applications
  • **Most popular in practice**
    - Google Chrome, Mozilla Firefox, Linux Kernel
    - American Fuzzy Lop (AFL), ClusterFuzz, OSS-Fuzz
Redzone-based memory error detection

- Buffer overflow (spatial memory errors)

Object $objX$ with pointer $ptrX$

Shadow memory: a bitmap to validate all addresses

Check before access

Accessible
Redzone-based memory error detection

• Buffer overflow (spatial memory errors)

Shadow memory: a bitmap to validate all addresses

Redzone: inaccessible region between objects

Accessible
Inaccessible (redzone)
Redzone-based memory error detection

• Buffer overflow (spatial memory errors)

Shadow memory: a bitmap to validate all addresses

Redzone: inaccessible region between objects

accessible
inaccessible (redzone)
Redzone-based memory error detection

- **Use-after-free** (temporal memory errors)
Redzone-based memory error detection

• Use-after-free (temporal memory errors)

Region is invalidated and quarantined, but not actually deallocated

- Accessible
- Inaccessible
- Shadow memory
Redzone-based memory error detection

• Use-after-free (temporal memory errors)

Hold the region until quarantine zone is full (FIFO)
Redzone-based memory error detection

• Use-after-free (temporal memory errors)

The region is actually deallocated, and can be allocated to a new object
Limitations of redzone-based approach

1. What if a pointer accesses *beyond* redzone?
Limitations of redzone-based approach

1. What if a pointer accesses beyond redzone?

Spatial memory error
Limitations of redzone-based approach

1. What if a pointer accesses \textit{beyond} redzone?

2. What if a dangling pointer accesses \textit{after} another object is allocated in the region?

Spatial memory error
Limitations of redzone-based approach

1. What if a pointer accesses *beyond* redzone?

2. What if a dangling pointer accesses *after* another object is allocated in the region?

Spatial memory error

Temporal memory error
Limitations of redzone-based approach

1. What if a pointer accesses *beyond* redzone?

2. What if a dangling pointer accesses *after* another object is allocated in the region?

Cannot detect!

Spatial memory error

Temporal memory error
Motivation

• To enhance detectability of redzone-based memory error detection
  • P1. Large gap to detect spatial memory errors
  • P2. Large quarantine zone to detect temporal memory errors
Motivation

• To enhance detectability of redzone-based memory error detection
  • P1. Large gap to detect spatial memory errors
  • P2. Large quarantine zone to detect temporal memory errors
Motivation

- To enhance detectability of redzone-based memory error detection
  - P1. Large gap to detect spatial memory errors
  - P2. Large quarantine zone to detect temporal memory errors

Huge physical memory required
MEDS overview

• Enhances detectability of redzone-based memory error detection

• **Idea:** Fully utilize 64-bit virtual address space to support
  • P1. Large gap to detect spatial error
  • P2. Large quarantine zone to detect temporal error

• **Approach:** minimize physical memory use
  • Page aliasing allocator and page protection
  • Hierarchical memory error detection
Page aliasing (P1)

- Maps multiple virtual pages to single physical page
Page aliasing (P1)

- Maps multiple virtual pages to single physical page
Page aliasing (P1)

- Maps multiple virtual pages to single physical page

Redzone itself does not occupy physical memory
Page protection (P1)

- Redzone only pages are unmapped
Page protection (P1)

- Redzone only pages are unmapped

Do not occupy shadow memory and physical memory

Virtual

Physical

A memory page
Unmapped page
Allocated
Redzone
Page aliasing
Page aliasing & Page protection (P2)
Page aliasing & Page protection (P2)

A memory page
Unmapped page
Allocated
Redzone
Page aliasing
Page aliasing & Page protection (P2)

Reuse physical memory immediately, while not reusing virtual addresses

- A memory page
- Unmapped page
- Allocated
- Redzone
- Page aliasing
Hierarchical memory error detection

• Many different ways to represent redzones
  ➔ Further optimizing physical memory uses
Hierarchical memory error detection

• Many different ways to represent redzones
  ➔ Further optimizing physical memory uses

#1. Shadow memory is invalid
Hierarchical memory error detection

- Many different ways to represent redzones
  ➔ Further optimizing physical memory uses

- #1. Shadow memory is invalid
- #2. Virtual page is unmapped
Hierarchical memory error detection

- Many different ways to represent redzones
  ➔ Further optimizing physical memory uses

- #1. Shadow memory is invalid
- #2. Virtual page is unmapped
- #3. Shadow memory is unmapped
Evaluation

• Configuration

<table>
<thead>
<tr>
<th></th>
<th>ASan</th>
<th>MEDS</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redzone</td>
<td>8-1024 bytes</td>
<td>4MB</td>
<td>16,384x</td>
</tr>
<tr>
<td>Quarantine</td>
<td>128MB</td>
<td>80TB</td>
<td>65,536x</td>
</tr>
</tbody>
</table>

• ASan cannot use configuration for MEDS (lack of memory)

• Compatibility

• Performance: 2 times slowdown

• Detection (fuzz testing): 68% more detection
Compatibility

• Unit tests from real-world applications
  • Test cases in Chrome, Firefox, Nginx
    • All Passed

• Memory error unit tests
  • ASan unit tests
    • All Passed
  • NIST Juliet test suites
    • All Passed except random access tests
      ➔ ASan: 35% vs. MEDS: 98%
Micro-scale performance overhead

• TLB misses
  • 5 times more than ASan (more virtual pages with page aliasing)

• Number of system calls
  • mmap(), munmap(), and mremap()
  • 32 times more than ASan (page aliasing and page protection)

• Memory footprint
  • 218% more than baseline
  • 68% more than ASan (much larger redzone and quarantine)
End-to-end performance overhead

- 108% compared to baseline, 86% to ASan
End-to-end performance overhead

• 108% compared to baseline, 86% to ASan
End-to-end performance overhead

- 108% compared to baseline, 86% to ASan

![Graph showing performance overhead](image)

- Large number of small objects on stack
  - 41% to baseline
  - 22% to ASan
  - 243% to baseline
  - 211% to ASan
Detection (fuzz testing)

• Run AFL (8 cores, 6 hours)
• Despite the performance overhead, explore **68.3% more unique crashes** than ASan
Detection (fuzz testing)

- Run AFL (8 cores, 6 hours)
- Despite the performance overhead, explore **68.3% more unique crashes** than ASan

MEDS finds more unique crashes in initial phase, but saturated in the end
Detection (fuzz testing)

- Number of unique crashes with time spent (metacam)
How MEDS explores more crashes?

• More input sets can be detected
  • Higher probability to detect
  • Bugs can be found earlier than ASan
  • Fuzzer can focus on the other paths

• MEDS can detect the cases that ASan cannot detect
  • Always bypass redzone
  • e.g., Miscalculation of structure array size
    • Size of the structure is larger than redzone size
    • Access to certain element cannot be detected.
How MEDS explores more crashes?

- More input sets can be detected
  - Higher probability to detect
  - Bugs can be found earlier than ASan
  - Fuzzer can focus on the other paths

- MEDS can detect the cases that ASan cannot detect
  - Always bypass redzone
  - e.g., Miscalculation of structure array size
    - Size of the structure is larger than redzone size
    - Access to certain element cannot be detected.

```c
int a[10];
a[x] = x;
```

```c
struct A {
    int num[10];
};
struct A *a = malloc(sizeof(struct A));
...
(a+i)->num[8] = i;
```
How MEDS explores more crashes?

• More input sets can be detected
  • Higher probability to detect
  • Bugs can be found earlier than ASan
  • Fuzzer can focus on the other paths

• MEDS can detect the cases that ASan cannot detect
  • Always bypass redzone
  • e.g., Miscalculation of structure array size
    • Size of the structure is larger than redzone size
    • Access to certain element cannot be detected.

```c
int a[10];
a[x] = x;
```

```c
struct A {
    int num[10];
};
struct A *a = malloc(sizeof(struct A));
...
(a+i)->num[8] = i;
```
How MEDS explores more crashes?

• More input sets can be detected
  • Higher probability to detect
  • Bugs can be found earlier than ASan
  • Fuzzer can focus on the other paths

• MEDS can detect the cases that ASan cannot detect
  • Always bypass redzone
  • e.g., Miscalculation of structure array size
    • Size of the structure is larger than redzone size
    • Access to certain element cannot be detected.

```c
int a[10];
a[x] = x;
```

```c
struct A {
    int num[10];
};
struct A *a = malloc(sizeof(struct A));
...
(a+i)->num[8] = i;
```
Conclusion

• **Idea**
  • Support *large gap* and *large quarantine zone*

• **Approach**
  • *Page aliasing* and *page protection*
  • *Hierarchical* memory error detection

• Despite overhead (108%), MEDS finds more crashes during fuzz testing (68.3%)

• **Open source – will be available soon**
  • [https://github.com/purdue-secomp-lab/MEDS](https://github.com/purdue-secomp-lab/MEDS)
  • Please use to detect bugs
Thank you for listening!