HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators

Insu Yun, Woosun Song, Seunggi Min (KAIST), Taesoo Kim (Georgia Institute of Technology)
Heap vulnerabilities are serious

Top vulnerability classes in systems software[1] at Microsoft (2016 through 2019)

- #1 – heap out-of-bounds
- #2 – use after free
- #3 – type confusion
- #4 – uninitialized use

From “Pursuing Durably Safe Systems Software”, Matt Miller, SSTIC 2020
Many secure allocators are proposed

**DieHarder: Securing the Heap**
Gene Novak
Dept. of Computer Science
University of Massachusetts Amherst
gnovak@cs.umass.edu
Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
emery@cs.umass.edu

**FreeGuard: A Faster Secure Heap Allocator**
Sam Silvestro
University of Texas at San Antonio
Sam.Silvestro@srsu.edu
Hongyu Liu
University of Texas at San Antonio
lifbysec@gmail.com
Corey Crosser
United States Military Academy
Corey.Crosser@usma.edu
Zhiqiang Lin
University of Texas at Dallas
zhiquang.lin@utdallas.edu
Tongping Liu
University of Texas at San Antonio
Tongping.Liu@srsu.edu

**Preventing Use-After-Free Attacks with Fast Forward Allocation**
Brian Wickman¹ Hong Hu² Insu Yun Dahee Jang
JungWon Lim Sandhya Kashyap³ Taesoo Kim
¹GTRI ²PennState ³GeorgiaTech ⁴EPFL

**LLVM Home | Documentation » Reference » Scudo Hardened Allocator**

**mimalloc**

**GrapheneOS**

**GrapheneOS / hardened_mimalloc**
Secure allocators support many security properties

- Prevent adjacent chunks
  - e.g., randomization
- Detect buffer overflow
  - e.g., heap canary
- Prohibit reusing memory
  - e.g., randomization
- Stop heap spray
  - e.g., randomization
- Prevent information leakage
  - e.g., separated heap metadata

The security properties are \textit{claimed} individually but attested with \textit{limited} test cases.
Problem 1: Hard to compare them with each other

Does it support all security properties?

Can we quantify this?
Problem 1: Hard to compare them with each other

Does it secure in **every** case? large allocation, negative allocation, even more ...
Example: Double free in DieHarder

```c
void* p0 = malloc(80KB);
free(p0);

void* tmp = malloc(100KB);
free(p0); // free 'p0' again

void* p2 = malloc(80KB);
free(tmp);

void* p3 = malloc(80KB);
assert(p2 == p3);
```

Double free a large chunk

⇒

Overlapping chunks

(Because DieHarder has no protection on large chunks)
Recall: ArcHeap (Usenix Security ’20)

Heap action generation

- `malloc(sz)`  
  Allocation
- `p[i_{overflow}] = v`  
  Overflow
- `free(p)`  
  Deallocation
- `free(p_{freed})`  
  Double free

Abnormality detection

- Chunk 1
- Chunk 2
  - Overlap with others
  - Outside of heap
  - Corruption in non-heap memory

Proof-of-concept generation

```c
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));
```
Problem 3: ArcHeap cannot evaluate secure properties

malloc(sz)
Allocation

free(p)
Deallocation

p[i_{overflow}]=v
Overflow

free(p_{freed})
Double free

Heap action generation

void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));

Proof-of-concept generation

Inflexible

Local
(i.e., a single instance)

Deterministic
Recall: secure allocators support many security properties

• Prevent adjacent chunks
  • e.g., randomization
• Detect buffer overflow
  • e.g., heap canary
• Prohibit reusing memory
  • e.g., randomization
• Stop heap spray
  • e.g., randomization
• Prevent information leakage
  • e.g., separated heap metadata
HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators

**Heap action generation**

- `malloc(sz)`
- `free(p)`
- `p[i_{overflow}] = v`
- `free(p_{freed})`

**Local abnormality detection**

- Chunk 1
- Chunk 2
- Freed chunk 1
- Chunk 2
- Reclaim

**Proof-of-concept generation**

```c
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));
```

**Sampling-based Testing**

- Extensible analysis
- Install hooks
- Statistical Significance Delta Debugging

**Success ratio**

- X %
Examples: adjacent chunks

- **Goal:** Check whether the secure allocator can avoid adjacent chunks

- **Analysis:**
  - Local: Check whether adjacent chunks happen by hooking allocations
  - Global: Calculate the probability of adjacent chunks

- **PoC:** Programs with a high chance to get adjacent chunks (e.g., > 25%)
Examples: heap spray

• **Goal:** Check whether the allocator is resilient from heap spray attacks

• **Analysis:**
  - Local: Record chunks’ start and size by hooking allocations
  - Global: Calculate the highest probability of the common address among multiple executions

• **PoC:** Programs with a high chance to get the common address
HardsHeap is extensible to cover various security properties

<table>
<thead>
<tr>
<th>Modules</th>
<th>LoC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacent</td>
<td>135</td>
<td>Check if chunks can be adjacent</td>
</tr>
<tr>
<td>Reclaim</td>
<td>119</td>
<td>Check if a dangling chunk is reclaimable</td>
</tr>
<tr>
<td>CheckOnFree</td>
<td>89</td>
<td>Check if an allocator can detect a corrupted chunk at free</td>
</tr>
<tr>
<td>Uninitialized</td>
<td>78</td>
<td>Check if we get metadata of allocators</td>
</tr>
<tr>
<td>Heap spray</td>
<td>64</td>
<td>Check if we can guess a fixed address for every execution</td>
</tr>
<tr>
<td>SizeCheck</td>
<td>61</td>
<td>Check if a chunk can be smaller than its request</td>
</tr>
<tr>
<td>ArcHeap</td>
<td>574</td>
<td>Other heap vulnerabilities</td>
</tr>
</tbody>
</table>

- **Usable**: ~100 lines of code
- **Extensible**: Various security properties
Due to randomized mechanisms, some test cases are non-deterministic

Is this action redundant?

```c
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));
...
```

Success (i.e., abnormal behavior)

Failure

Success
Recall: Delta Debugging

```c
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));
```

This action is redundant!

No, this action is not redundant!
HardsHeap addresses this issue by using Statistical Significance Delta Debugging (SSDD)

void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);
// [BUG] free 'p0' again
free(p0);
void* p2 = malloc(lsz);
free(p1);
assert(p2 == malloc(lsz));

This action is redundant if
1) Y is not significantly worse
or 2) Y is significantly better than X
Evaluation on real-world secure allocators

• Apply to **10** open-source *secure* allocators
  • 6 from academic works
    • DieHarder (CCS ’10), FreeGuard (CCS ’17),
    • Guarder (Security ’18), SlimGuard (Middleware ’19),
    • MarkUS (Oakland ’20), ffmalloc (Security ’21)
  • 4 from non-academic works
    • scudo (Android)
    • mimalloc (Microsoft)
    • hardened_malloc (GrapheneOS)
    • isoalloc (partially inspired by Chrome's PartitionAlloc)
Bugs found by HardsHeap

- **10 bugs** are discovered, **5** are fixed

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Module</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guarder</td>
<td>Adjacent</td>
<td>Insufficient randomness due to predictable seeds</td>
<td>R</td>
</tr>
<tr>
<td>FreeGuard</td>
<td>Adjacent</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>MarkUs</td>
<td>Reclaim</td>
<td>Unsafe reclamation in mmapped memory</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsafe reclamation due to failed allocation</td>
<td>P</td>
</tr>
<tr>
<td>mimalloc</td>
<td>Spray</td>
<td>Heap spray is possible due to memory overcommit</td>
<td>P</td>
</tr>
<tr>
<td>Guarder</td>
<td>SizeCheck</td>
<td>Integer overflow in memory allocation</td>
<td>A</td>
</tr>
<tr>
<td>FreeGuard</td>
<td>SizeCheck</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>isoalloc</td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>fmalloc</td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>SlimGuard</td>
<td>ArcHeap</td>
<td>Insufficient check for invalid free</td>
<td>R</td>
</tr>
</tbody>
</table>

R: Reported, A: Acknowledged, P: Patched
Example: adjacent objects in Guarder/FreeGuard

- **Claim:** malloc() return random chunks

```c
void* p0 = malloc(...);
void* p1 = malloc(...);
void* p2 = malloc(...);
void* p3 = malloc(...);
...
```

**use time() as random source:**
- seconds since 1/1/1970
- the same within 1 second

Two malloc 100% return adjacent objects in a short time period
Example: reclaim objects in MarkUs (1/2, Fixed)

- **Claim:** Do not reallocate an object if any reference exists

```c
void* p0 = malloc(-1);
void* p1 = malloc(0x80000);
free(p1);
void* p2 = malloc(0x40000);
assert(p1 <= p2 && p2 < p1 + 0x80000);
```

After the very large malloc fails (e.g., -1), MarkUs switches to unsafe reallocation.
Example: heap spray in mimalloc (Fixed)

• **Claim**: heap address is randomized within 64-bit address space

```c
void* p0 = malloc(4TB);
// p0 is always like 0x7FFFFFFFFxxx for any runs
```

mimalloc uses MAP_NORESERVE to overcommit memory, which is harmful for randomization

**Fix**: return NULL for large allocation > 1GB
HardsHeap also shows limitations of secure allocators (e.g., Large allocation)

- **Known**: DieHarder’s entropy is inversely proportional to size
  - HardsHeap found reliable adjacent chunks on very large allocation

- **Unknown**: Scudo’s entropy is similar to DieHarder’s
- **Unknown**: Guarder’s entropy becomes zero if we allocate very large chunks (> 512KB)

HardsHeap can discover these behaviors automatically!
SSDD is better than other minimization mechanisms

- Classic: Classical Delta Debugging
- Greedy: Only consider average probability without statistical significance

13.8% smaller than Greedy

48.7% higher reproducibility than Classic
Limitations & Discussion

• Limitations
  • Incompleteness
  • Lack of reasoning
  • Only Linux support

Q: HardsHeap results imply that secure allocators are useless?

A: No! They are not silver bullet but are very useful (See our paper). Please use them!
Conclusion

• HardsHeap: Automatic ways to evaluate secure allocators
  • Extensible framework
  • Sampling-based testing
  • Statistical Significance Delta Debugging (Please see our paper)

• 10 implementation bugs and many limitations of various secure allocators

• Open source: https://github.com/kaist-hacking/HardsHeap
Thank you