Automatic Techniques to Systematically Discover New Heap Exploitation Primitives

Insu Yun, Dhaval Kapil, and Taesoo Kim

Georgia Institute of Technology
Heap vulnerabilities are the most common, yet serious security issues.

From “Killing Uninitialized Memory: Protecting the OS Without Destroying Performance”, Joe Bialek and Shayne Hiet-Block, CppCon 2019
Heap exploitation techniques (HETs) are preferable methods to exploit heap vulnerabilities

• Abuse underlying allocator to achieve more powerful primitives (e.g., arbitrary write) for control hijacking
  • Application-agnostic: rely on only underlying allocators
  • Powerful: e.g., off-by-one null byte overflow → arbitrary code execution

• Used to compromise (in 2019)
Example: unlink() in ptmalloc2

unlink():

\[ P->fd->bk = P->bk \]
\[ P->bk->fd = P->fd \]
Example: unlink() in ptmalloc2

unlink(): $P->fd->bk = P->bk$

$P->bk->fd = P->fd$
Example: Unsafe unlink() in the presence of memory corruptions (e.g., overflow)

unlink(): \(P\rightarrow fd\rightarrow bk = P\rightarrow bk\)

\[\Rightarrow fptr = evil\]
Security checks are introduced in the allocator to prevent such exploitations

unlink(): \texttt{assert} (P->fd->bk == P);
P->fd->bk = P->bk

This check is still \textit{bypassable}, but it makes HET more \textit{complicated}
Researchers have been studied reusable HETs to handle such complexities

Title: Once upon a free()
Author: anonymous author

Project Zero

Understanding it breaking it

News and updates from the Project Zero team at Google

All analyses are manual, ad-hoc, and allocator-specific!

Exploiting the win

From: "Phantasmal Phantom
Date: Mon, 23 Feb 2004 21:50:50 -0800

Posted by Chris Evans, Exploit Writer Underling to Tavis Ormandy
Problem 1: Existing analyses are highly biased to certain allocators

- ptmalloc2 (Linux allocator)
- tcmalloc
- DieHarder
- mimalloc
- mesh
- scudo
- Freeguard
Problem 2: A manual re-analysis is required in the changes of an allocator’s implementation.

ptmalloc2 (Linux allocator)

A new feature: thread-local cache (tcache)

Question: How to find HETs automatically?
Our key idea: ArcHeap autonomously explore spaces similar to fuzzing!
Technical challenges

Large search space

Lack of an efficient way to evaluate HETs
Technical challenges

- Large search space

Lack of an efficient way to evaluate HETs
Search space consisting of heap actions is enormous.

- **malloc(sz)**: Allocation
- **free(p)**: Deallocation
- **p[i]=v**: Heap write
- **buf[i]=v**: Buffer write
- **2^{64}**: Size of heap actions
- **size(p) \times 2^{64}**: Size of buffer write

Search space can be reduced using model-based search based on **common designs** of allocators!

- **overflow**: Overflow
- **freed[i]**: Write-after-free
- **freed**: Double free
- **non-heap freed**: Arbitrary free

Buggy actions
Common design 1: Binning

• Specially managing chunks in different size groups
  • Small chunks: Performance is more important
  • Large chunks: Memory footprint is more important

• e.g., ptmalloc
  • fast bin (< 128 bytes): no merging in free chunks
  • small bin ( < 1024 bytes): merging is enabled

• Sampling a size uniformly in the $2^{64}$ space $\Rightarrow$ $P(\text{fast bin}) = 2^{-57}$
ArcHeap selects an allocation size aware of binning

• Sampling in exponentially distant size groups

• ArcHeap partitions an allocation size into four groups:
  \((2^0, 2^5], (2^5, 2^{10}], (2^{10}, 2^{15}], \text{ and } (2^{15}, 2^{20}]\)

• Then, it selects a group and then selects a size in the group uniformly
  • e.g., \(P(\text{fast bin}) > P(\text{selecting a first group}) = \frac{1}{4}\)
Other common designs: Cardinal data and In-place metadata

• Cardinal data: Metadata in a chunk are either sizes or pointers, but not other random values

• In-place metadata: Allocators place metadata near its chunk’s start or end for locality
Cardinal data and In-place metadata reduce search space in data writes

\[ p[i] = v \]

-8 \sim 8

1337

Heap write

Size

Random size

Other chunk’s size

Pointer

Other chunk

Buffer

0xdeadbeef

Container

An array that stores chunks
Technical challenges

- Large search space
- Lack of an efficient way to evaluate HETs
Automatically synthesizing full exploits is inappropriate in evaluating HETs

• Difficult: e.g., In the DAPRA CGC competition, only one heap bug was successfully exploited by the-state-of-the-art systems

• Inefficient: Takes a few seconds, minutes, or even hours for one try

• Application-dependent: A HET, which is not useful in a certain application, may be useful in general
Our idea: Evaluating impacts of exploitations (i.e., detecting broken invariants that have security implications)

1. Allocated memory should not be overlapped with pre-allocated memory
   - Overlapping chunks: Can corrupt other chunk’s data
   - Arbitrary chunks: Can corrupt global data

2. An allocator should not modify memory, which is not under its control (i.e., heap)
   - Arbitrary writes
   - Restricted writes

Easy to detect: Check this at every allocation

How about this? (NOTE: should be efficient)
Shadow memory can detect arbitrary writes and restricted writes

- Maintain external consistency
- Check divergence

```c
container[i] = malloc(sz)
container_shadow[i] = malloc(sz)
check: equal(container, container_shadow)
buf[i] = v
buf_shadow[i] = v
check: equal(buf, buf_shadow)
```

Divergence can only happen in the internal of allocators.
ArcHeap provides a minimized PoC code for further analysis

• Proof-of-Concept code: Converting actions into C code
  • Trivial, because they have one-to-one mapping

• Minimize the PoC code using delta-debugging
  • Idea: Eliminate an action, which is not necessary for triggering the impact of exploitations
  • Details can be found in our paper
Evaluation questions

1. How effective is ArcHeap in finding new HETs, compared to the existing tool, HeapHopper?

2. How general is ArcHeap’s approach?
ArcHeap discovered five new HETs in ptmalloc2, which cannot be found by HeapHopper

• Unsorted bin into stack: Write-after-free → Arbitrary chunk
  • Requires fewer steps (5 steps vs 9 steps)
• House of unsorted einherjar: Off-by-one write → Arbitrary chunk
  • No require heap address leak

All HETS cannot be discovered by HeapHopper because of its scalability issue (i.e., symbolic execution + model checking)

• Fast bin into other bin: Write-after-free → Arbitrary chunk
ArcHeap is generic enough to test various allocators

- Tested 10 different allocators
  - Cannot find HETs in LLVM Scudo, FreeGuard, and Guarder, which are “secure allocators”

Even found HETs in “secure” allocators

Works for ptmalloc2-unrelated allocators

<table>
<thead>
<tr>
<th>Allocators</th>
<th>P</th>
<th>I</th>
<th>Impacts of exploitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>✓</td>
<td>✓</td>
<td>AF, OV, WF</td>
</tr>
<tr>
<td>jemalloc</td>
<td>✓</td>
<td>✓</td>
<td>OV, DF</td>
</tr>
<tr>
<td>tcmalloc</td>
<td>✓</td>
<td>✓</td>
<td>OV, WF, DF</td>
</tr>
<tr>
<td>mimalloc-1.0.8</td>
<td>✓</td>
<td>✓</td>
<td>DF, NO</td>
</tr>
<tr>
<td>mimalloc-secure-1.0.8</td>
<td>✓</td>
<td>✓</td>
<td>AF, OV, WF</td>
</tr>
<tr>
<td>DieHarder-5a0f8a52</td>
<td>✓</td>
<td>✓</td>
<td>OV, WF</td>
</tr>
<tr>
<td>mesh-a49b6134</td>
<td>✓</td>
<td>✓</td>
<td>DF, NO</td>
</tr>
</tbody>
</table>

N: New techniques compared to the related work, HeapHopper [17]; only top three allocators matter. NO: No bug is required, i.e., incorrect implementations. I: In-place metadata, P: ptmalloc2-related allocators.
Case study 1: Double free $\rightarrow$ Overlapping chunks in DieHarder and mimalloc-secure

```c
// [PRE-CONDITION]
//   lsz : large size (> 64 KB)
//   xlsz: more large size (>= lsz + 4KB)
// [BUG] double free
// [POST-CONDITION]
//   p2 == malloc(lsz);
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));
```

Double free large chunk $\rightarrow$ Overlapping chunk

Same thing happens in both DieHarder and mimalloc-secure
Interestingly, these issues are irrelevant

Me: Is mimalloc related to DieHarder?

Mimalloc developer: No!

DieHarder \texttt{unmap}(p_{\text{large}})

mimalloc \texttt{check}(p_{\text{large}})

\texttt{free}(p_{\text{large}})

No check!

Wrong check!
Our PoC has been added in a malloc’s regression test

```c
55 + static void double_free2() {
56 +   void* p[256];
57 +   uintptr_t buf[256];
58 +   // [INFO] Command buffer: 0x327b2000
59 +   // [INFO] Input size: 182
60 +   p[0] = malloc(712352);
61 +   p[1] = malloc(786432);
62 +   free(p[0]);
63 +   // [VULN] Double free
64 +   free(p[0]);
65 +   p[2] = malloc(786440);
66 +   p[3] = malloc(917504);
67 +   p[4] = malloc(786440);
68 +   // [BUG] Found overlap
69 +   // p[4]=0x433f1402000 (size=917504), p[1]=0x433f14c2000 (size=786432)
70 +   printf(stderr, "p1:%p, p2: %p
", p[4], (uintptr_t)786432);
71 + }
```
Case study 2: Overflow → Arbitrary chunk in dlmalloc-2.8.6

• dlmalloc: ancestor of ptmalloc2 but has been diverged after its fork

```c
void* p0 = malloc(sz);
void* p1 = malloc(xlksz);
void* p2 = malloc(lsz);
void* p3 = malloc(sz);

// [BUG] overflowing p3 to overwrite top chunk
struct malloc_chunk *tc = raw_to_chunk(p3 + chunk_size(sz));
tc->size = 0;

void* p4 = malloc(fsz);
void* p5 = malloc(dst - p4 - chunk_size(fsz) \n    - offsetof(struct malloc_chunk, fd));
assert(dst == malloc(sz));
```

Looks complicated…
Its root cause is more complicated!

// Make top chunk available
void* p0 = malloc(sz);
// Set mr.mflags |= USE_NONCONTIGUOUS_BIT
void* p1 = malloc(xlsz);
// Current top size < lsz (4096) and no available bins, so dlmalloc calls sys_alloc
// Instead of using sbrk(), it inserts current top chunk into treebins
// and set mmapped area as a new top chunk because of the non-continous bit
void* p2 = malloc(lsz);
void* p3 = malloc(sz);
// [BUG] overflowing p3 to overwrite treebins
struct malloc_chunk *tc = raw_to_chunk(p3 + chunk_size);
  tc->size = 0;
// dlmalloc believes that treebins (i.e., top chunk) has enough size
// However, underflow happens because its size is actually zero
void* p4 = malloc(fsz);
// Similar to house-of-force, we can allocate an arbitrary chunk
void* p5 = malloc(dst - p4 - chunk_size(fsz) \n  - offsetof(struct malloc_chunk, fd));
assert(dst == malloc(sz));
Discussion & Limitations

• Incompleteness: Unlike HeapHopper that is complete under its model
  • But HeapHopper’s model cannot be complete because of its scalability issue

• Overfitting: Our strategy might not work for certain allocators
  • In practice, our model is quite generic: found HETs in seven allocators out of ten except for secure allocators

• Scope: ArcHeap only finds HETs and does not generate end-to-end exploits for an application
Conclusion

• Automatic ways to discover HETs
  • Model-based search based on common designs of allocators
  • Shadow-memory-based detection

• Five new HETs in ptmalloc2 and several ones in other allocators
  • Including secure allocators, DieHarder and mimalloc secure

• Open source: https://github.com/sslab-gatech/ArcHeap
Thank you!