Preventing Use-After-Free Attacks with Fast Forward Allocation

Introduction to UAF

Memory-unsafe languages, like C and C++, are widely used to implement key programs such as web browsers and operating systems. Therefore, we have seen innumerable memory safety issues detected and abused in these systems . Among all memory safety issues, the use-after-free bug is one of the most commonly reported and exploited security problems .
Researchers have proposed different methods to detect or mitigate use-after-free bugs. The first method is to label each memory chunk to indicate whether it is allocated or freed. Before each memory access, a label is checked to detect after-free use. However, this approach introduces high overhead to program execution and thus has not been widely adopted in released applications. A second way is to actively invalidate dangling pointers once the object is freed, like setting them to a NULL value. These tools must maintain the inverse points-to relationship between an object and its references, which significantly slows down the protected execution. As a special case, Oscar makes the freed objects inaccessible to achieve the same goal. A more recent approach ignores the normal free request, and utilizes spare CPU cores to independently identify and release garbage objects (i.e., objects without any reference). This method requires extra computation resources, and may have limited scalability.

Previous solutions for use-after-free bugs have limitations, mainly because attackers can easily reclaim and modify freed memory due to memory managers reusing released memory. If freed memory is never reused, attackers lose control over its content, making exploitation impossible, though the program may run normally or crash.
Inspired by this, we propose one-time allocation (OTA), where each memory address is assigned only once and never reused. OTA does not eliminate use-after-free bugs but makes them unexploitable.
Implementing OTA faces three challenges: high memory overhead due to page-level management, limits on the number of virtual memory area (VMA) structures in the kernel, and potential slowdown from frequent system calls.
Our solution, Fast Forward Memory Allocation (FFmalloc), addresses these challenges with a two-fold approach. Small allocations are grouped using a size-based binning allocator, while large allocations are handled sequentially in a continuous region, reducing wasted memory and VMA fragmentation. FFmalloc also batches memory mapping and unmapping to minimize system calls by caching extra memory internally and releasing it only when multiple contiguous pages are freed.

FFmalloc is implemented in 2,117 lines of C code. It requests memory from the kernel in 4MB chunks, groups requests under 2KB via binning, and allocates larger requests sequentially. Testing FFmalloc on nine programs with eleven exploitable use-after-free bugs showed that all exploits failed, confirming no memory overlap.
Performance evaluation on SPEC CPU2006, PARS

EC 3, ChakraCore, and NGINX showed that FFmalloc introduces moderate overhead: 2.3% CPU and 61% memory on SPEC CPU2006, 33.1% CPU and 50.5% memory on PARSEC 3, while maintaining negligible overhead on ChakraCore and comparable performance to other secure allocators. Compared to state-of-the-art tools, FFmalloc offers lower CPU overhead for strong protection.
Contributions:

- Revives OTA to prevent use-after-free attacks, offering strong security.
- Designs and implements FFmalloc, the first practical OTA prototype for Linux and Windows.
- Demonstrates through extensive evaluation that OTA is practical for real-world protection.

Approach Overview

A successful use-after-free exploit requires three steps: free, reallocate, and use, with reallocate being the key step attackers exploit. By preventing the reuse of freed memory, attackers cannot occupy or modify it, making use-after-free bugs unexploitable. This memory management strategy is called one-time allocation (OTA). While the program may crash or behave abnormally, exploitation is blocked.
Implementing a practical OTA allocator is challenging. Previous approaches either lacked sufficient security or caused high performance overhead: DieHarder randomized allocation addresses but only provided probabilistic protection; Archipelago and Oscar used separate pages for allocations but incurred over 40% overhead due to frequent system calls; Cling prevented reuse between mismatched types but left some use-after-free opportunities. OTA was previously considered impractical because of memory usage concerns.
Despite these challenges, OTA offers straightforward design and strong security guarantees without complex intelligence or external dependencies. By carefully addressing overhead while preserving security, we developed solutions that make a practical OTA implementation possible. Evaluations show that FFmalloc, our OTA allocator, incurs minimal overhead in most cases.

FFmalloc Design

FFmalloc is a one-time allocation (OTA) memory allocator that prevents reuse of freed memory, making use-after-free bugs unexploitable. It combines binning for small allocations and continuous allocation for large allocations to balance memory efficiency, performance, and security.

- Small allocations (≤2 KB): Handled via a binning allocator. Objects of similar size are grouped into bins, which reduces memory overhead caused by long-lived small allocations. Once a bin is fully freed, the pages are returned to the OS.
- Large allocations (>2 KB): Managed via a continuous allocator to minimize waste due to alignment requirements. Large allocations are served from continuous pools of memory pages.

Memory pools: FFmalloc requests memory from the OS in pools of 4 MB, which are the basic allocation unit. For very large allocations, a jumbo pool is created just for that request. Each CPU core is associated with a distinct pool to reduce lock contention.

Security features:

- The starting address of FFmalloc is offset from the existing heap, allowing coexistence with the glibc allocator.
- Pools are allocated at increasingly higher addresses using MAP_FIXED_NOREPLACE to preserve ASLR benefits and avoid overlaps.
- Memory is never reused, so attackers cannot reclaim or modify freed objects.

Efficiency measures:

- FFmalloc minimizes system calls by allocating and freeing memory in large contiguous blocks, reducing performance overhead.

- Batching allocations and unmapping pages ensures fewer OS interactions and avoids excessive VMA usage.

This hybrid design allows FFmalloc to provide strong security against use-after-free exploits while keeping memory and performance overhead practical for real-world programs.

Conclusion

Last but not least , our edit on glibc will prevent the user or the attacker to access the freed addresses , in other words we ensure that each time we want to allocate a new address which is never used before even if there are freed addresses we do not care about them .

and this is briefly how the ffmaloc prevent UAF therefore it prevent Tcach poisoning