NUAF: No Use-ing After Free-ing

1. Background

Managing your own memory is hard. When writing sufficiently complex programs, memory errors are likely to happen even in the hands of the most experienced programmers. A great deal of existing code is written in unsafe languages, and this will likely continue to be the case for the foreseeable future. Aside from the unpredictable behavior of programs affected by these memory errors, these errors are highly problematic for the security community. Memory errors account for the vast majority of CVE's, and there needs to be necessary tooling to support software developers. This tooling typically comes as either a debugging tool or as a exploit prevention tool to be run in production applications. We will focus on the latter case, of exploit prevention.

In this project, we are focusing on use-after-free (UAF) errors. The aptly named 'use-after-free' error occurs when a program frees memory associated with a given pointer, and later uses this 'stale' pointer. This error becomes a problem when the memory underlying the stale pointer is reused, as the program will then enter a state in which two references are unknowingly sharing memory. This bad state can lead to unpredictable behavior in the best case, and a security vulnerability in the worst case. If a malicious party is able to trigger a UAF error and gain ownership over the recycled memory, they will be able to specifically craft objects that can, in many cases, hijack control flow and execute arbitrary code or change program behavior for malicious purposes.

Mitigations against UAF errors typically fall into the domain of either managing the references to objects, or by managing the objects themselves by restricting/preventing object reuse. Typical examples of these methods to restrict object reuse include quarantine lists, only allowing for object reuse within objects of a similar type, and preventing object reallocation altogether. Systems managing references include DANGNULL [8], DangSan [10], FreeSentry [11], and pSweeper [9]–These can typically provide full protection, but managing pointer propagation and handling pointer arithmetic can be complex, expensive, and sometimes incomplete. Also, these systems typically have overhead proportional to the number of references, and require source code for compiler instrumentation.

2. Overview

Use-after-free errors are a common cause of security vulnerabilities in deployed programs, and therefore the software ecosystem is in need of a satisfying solution to this issue. Existing solutions often provide incomplete protection, require source code, and/or have a cumbersome overhead.
We based NUAF on Oscar [4], a system designed to provides complete protection against UAF errors, double free errors, and partial protection against invalid frees. We store each object on a

unique virtual page (shadow page), thus we can use MMU to invalidate all the access to the freed object. Virtual pages are aliased over the same physical page frames, allowing for efficient use of space under this scheme. This system is simple, and provides complete protection against heap use-after-free errors. Because our implementation consists only of a runtime library, we provide a backwards-compatible, no-source-required solution. From our evaluation, we find that our system provides a low overhead relative other UAF systems, which would allow for its use in real-world applications once some of its issues are resolved. Additionally, since our overhead is proportional to the number of objects rather than the number of references, our solution provides practically no overhead for programs with low allocation density. Our program was tested on allocator benchmarks created/collected by Emery Berger in his Hoard project [1], a number of PARSEC benchmarks [3], and a collection of self-written correctness/performance tests.

3. Design

3.1 Using the MMU

The basic idea behind NUAF is dependent on using the MMU to our advantage to render stale references unusable. MMUs, built into most contemporary processors, are able to control access to memory on a per-page basis. If each object has its own virtual mapping, then rendering an object unreachable is simple through the use of page-protection mechanisms. Since the minimum allocation size in this scheme is one page, the old wisdom was to put one object per physical page. The highly inefficient use of the space made that these page-protection schemes have largely been considered unusable outside of limited debugging contexts.

3.2 Virtual Page Aliasing

In order to manage the space-complexity of a page-protection scheme, we adopted the idea of aliasing virtual pages to a single physical page frame introduced by Dhurjati and Adve [6]. Here we brief their method, but we implemented the same idea in a slightly different way. The libc allocator will put multiple objects on the same virtual page, which Dhurjati and Adve refer to as the canonical virtual page, and thus the objects share the same physical page. On top of that, for each object, a new mapping to the same physical address is created via mremap and thus the object has its own shadow virtual page. Only the shadow page address is exposed to the user, so the system can prevent use-after-free by controlling the permission for the freed object’s shadow page, and it won’t affect other objects on the same physical page. The diagram below, took from Oscar paper, demonstrates the genealogy.

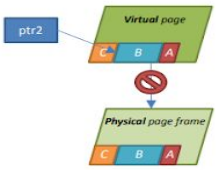


Figure 6: The virtual page has been made inaccessible: accesses to objects A, B or C would cause a fault.

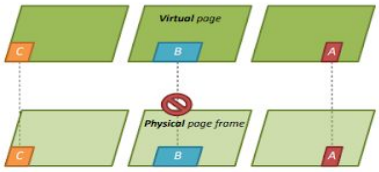


Figure 7: With one object per page, we can selectively disable object B.

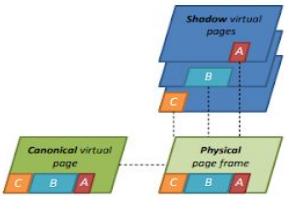


Figure 8: Each object has its own shadow virtual page, which all map to the same physical frame.

The virtual page aliasing is a effective method, providing full protection against the use-after-free errors and it uses space efficiently. However, mapping implies costly system calls, and as the amount of mappings is expected to increase proportional to the amount of allocated objects, the TLB pressure will increase too. For these two reasons, the time overhead will be a problem if without special design. In the next section, we will explain our implementation of virtual page aliasing and the engineering effort to reduce its time overhead.

4. Implementation

We wrote a custom allocator to detect all the use-after-free errors via virtual page aliasing. We implemented a high-watermark to provide a scheme wherein addresses can be unmapped and never reused.

4.1 Allocator

Our allocator was built using Heap Layers[2], an extensible memory allocation infrastructure. Through Heap Layers, we need only define malloc, free, and `malloc_usable_size`, `malloc_lock`, and `malloc_unlock`, and we have a fully functional allocator. To make use of the MMU, we force every new allocation to use `mmap`, and guarantee that each object enjoys a distinct virtual range, with the size at least one page. We treat small and large objects differently: for small objects, we implement a BiBoP-style allocator with a size-segregated heap and an embedded freelist and alias the virtual pages [7]. To gain the control of canonical page arrangement, the “heap” is actually a temporary backing file created at program construction time that is truncated to be arbitrarily large. This file can then be dynamically expanded as needed over the course of the program lifetime. On top of this, we can do `mmap` for each of the object and thus they have different shadow virtual addresses. In this way, when `mmap` is called, we can specify the offset which is effectively the object’s canonical address. For large objects, the ones occupies one page individually, we let the operating system to decide its physical pages. The virtual addresses are forced to be separated from the smaller objects for easy distinction and we change the permission of the virtual pages to inaccessible (`PROT_NONE`) when they are freed.

4.2 High Watermark

We borrowed the high watermark design from Oscar to guarantee that the same virtual page is never reused. This setting also buys us the opportunity to “`munmap`” when the object is freed.

Oscar reinvigorated the aliasing mechanism with the introduction of the “high watermark” as a means of freeing old mappings while ensuring that they will never be used again. The idea for the high watermark is founded in the insight that virtual address spaces are large enough that we can reasonably provide unique mappings for each new object. With a 256 TB address space, the likelihood a program will allocate enough objects to fill this within its lifetime is very low. The simplest way to assign each object a unique virtual page is to just institute a high-watermark, starting at `MMAP_MIN_ADDR` provided by the system. By adding `MAP_FIXED` flag to `mmap` and `MREMAP_FIXED` flag to mremap, we can force the virtual address to be at the current high-watermark and increment the high-watermark by one `PAGE_SIZE` every time a new object is created. With the high watermark in place, `vma_structs` and page table entries for freed objects are no longer necessary, and we have the potential for a far more workable system.

4.3 Small Objects

For small objects (i.e., any object < ½ `PAGE_SIZE`), aliasing is necessary to obtain the individual controls over individual objects.

In terms of the design of the allocator, we implement a BiBoP-style allocator with a size-segregated heap and an embedded freelist and alias the virtual pages. This approach fits our needs very well, as our design infrastructure necessitates page-level granularity with respect to individual allocations, and we can avoid inlining any object metadata. Additionally, we believe this approach will simplify a future optimization which would involve multiple objects per virtual page.

We implemented a size-segregated heap and an embedded freelist.

On allocation, the size of the requested object is rounded up to the next power of two supported by our freelist. If the corresponding freelist is not empty, then this available address is popped from the freelist and returned to the user transparently. If the corresponding freelist is empty, then the a new object is mapped from the next physical region on the page for this-sized object in the backing data file to at the high watermark virtual address using `mmap` with `MAP_FIXED`, and these values are updated accordingly. It is worth noticing that, the address returned to the user is not necessarily the start of the newly mapped shadow virtual page, but with the in-page offset suggesting its relative address in this page.

On deallocation, objects are recycled by providing a new virtual address according to the high watermark using `mremap` with `MREMAP_MAYMOVE | MREMAP_FIXED`, and pushed onto the corresponding freelist. This single syscall will both make a newly usable object and unmap the old object, thereby destroying all stale references and preventing UAF.

Under this design, we only need one metadata per physical page to store the size info of the objects belonged to this page, and we set the metadata to be 16 byte to allow for aligned memory accesses.

4.4 Large Objects

Large objects (i.e., any object >= ½ `PAGE_SIZE`) receive special treatment, and are handled in a much simpler manner.

Upon allocation, the start of the memory allocated for the large object is denoted with a magic number followed by the number of pages this large object occupies. Large objects are provided whatever multiple of `PAGE_SIZE` is necessary to fit the given object and object metadata, and are `mmap`-ed using `MAP_ANONYMOUS`. Additionally, large objects are given a separate high watermark and live in the towards the top of the usable address space. By living in a separate region of the address space, distinguishing between large objects and small objects is simple even when provided with interior pointers, simplifying allocation/deallocation for both types respectively. For simplicity, we simply `munmap` large objects upon deallocation.

4.5 Support for parallel programs

NUAF works under multithreaded programs by coarse locking around regions updating the high watermarks, further optimization could be done relatively easily, however true parallelism of our program could be diminished even with no locking, due to the nature of mmap. Our system also supports multi-process programs. According to the `mmap` manpage, memory mapped by `mmap`() is preserved across `fork(2)`, with the same attributes. By using `MAP_PRIVATE` mappings, updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. We are not entirely sure that our system will support all configurations of `clone`, more testing and research is required.

4.6 Compared to Oscar

While the high-level ideas of our paper are essentially the same as Oscar, many implementation details are different. The primary divergence of our implementations is that Oscar uses `MAP_ANONYMOUS | MAP_SHARED` mappings with `mremap` as their means of creating aliases. On the other hand, we use a backing file and `MAP_PRIVATE` mappings. This difference simplifies physical memory management on their end, however introduces a lot of complexity and work when trying to support `fork()`-ed programs, as outlined in their paper. Additionally `MAP_PRIVATE` mappings are allegedly less expensive (according to Oscar).

Another major difference in our implementation is that they are working within the bounds of malloc/free internals, while we are simply creating our own allocator. They invoke libc's internal malloc implementation (changed to use `mmap` rather than `sbrk`) to get an object with some inline metadata within a physical page frame, use `mmap` to create a shadow for this object, and return the shadow to the user. Our scheme does not require inline metadata for objects, and the allocation and `mmap` are done in a single step.

5. Evaluation

Our evaluation consists of a number of questions.

- Does this system prevent all instances of use-after-free?
- Does this system have any false positives?
- Does this system allow programs to run properly?
- How much of an overhead does this system incur?

5.1 Protection

In all tests we have run, our system provides complete protection against both small and large object use-after-free. Because virtual addresses mappings are unique to a single logical object lifetime, and these mappings are removed upon free, it logically follows that all heap-based use-after-free errors are protected by this scheme. We include test programs that ensure our system implements this logic successfully. These tests include cases with large objects, small objects, and UAF's in various contexts. We found all instances triggered a segmentation fault as expected. Stack-based use-after-frees, while theoretically possible, are not much of an issue in the current ecosystem according to the oscar paper.

Our system also prevents double frees for the same reason that UAF are prevented. If a stale reference is freed a second time, there will be an error when trying to remap a nonexistent mapping, resulting in program termination. Invalid frees are generally protected for the same reason double frees would be protected. However, if it is an existing mapping in-use by the program generally, it will be unmapped and the memory corresponding to the old mapping will be placed on the freelist. An exploit around this functionality seems unlikely, and could be mitigated by checking to see if the object-to-be-freed has an address that makes sense within the bounds of our high watermarks.

Also, users own `mmap` calls or programs own mappings can potentially disturb our system. Right now we do not have a good solution to that, but since Linux 4.17, there is a flag MAP_FIXED_NOREPLACE, which can resolve part of the issue.

5.2 False Positives

This system does not have any false-positives in the normal sense of the word, as any segmentation fault caused by our system is a result of using memory after it has been freed. There are benign cases, however, where the UAFs happen before the reallocation of the new objects on the same physical page, in which case NUAF will catch and terminate the program.

This could potentially be avoided by only unmapping the last region upon the next allocation, however this seems unnecessary and it seems better to force correct memory management semantics.

5.3 Functionality in real-world programs

The starting high watermark for the object is currently hard-coded, irrespective of the initial memory layout of the loaded program, and NUAF clobbers any existing mappings that it runs into. What makes it worse is that programs compiled with different tools result in different memory layouts, and certain programs (e.g., cc1, a program forked & exec'd by gcc) will create their own memory mappings in unpredictable areas. Another example is that clang-built programs include memory mappings relatively low into the address space compared to gcc-built programs, and so we were forced to change our starting high watermark to accommodate this discrepancy. As of Linux 4.17, a flag has been added that prevents `mmap` from replacing existing mappings. With this flag, we would be able to implement a more workable solution. Another potential avenue would be to dynamically check memory mappings of a program upon construction, find a good starting high watermark and usable bounds, and catch all `mmap` calls

within the program. This seems unnecessary with kernel support for no-clobber, so we did not implement this.

In addition, there are several test programs killed by the OS after a short period of running with NUAF. We suspect that these programs involve many fast allocations, leading to many virtual mappings, which then leads to unexpectedly high kernel memory usage (from VMA structs and PTE's). There is likely some mechanism in our configuration of the linux kernel that kills processes that create too many mappings in a short period of time, although this is largely speculation. This issue was not encountered in any other implementations of this scheme that we have found, and is something to be researched further.

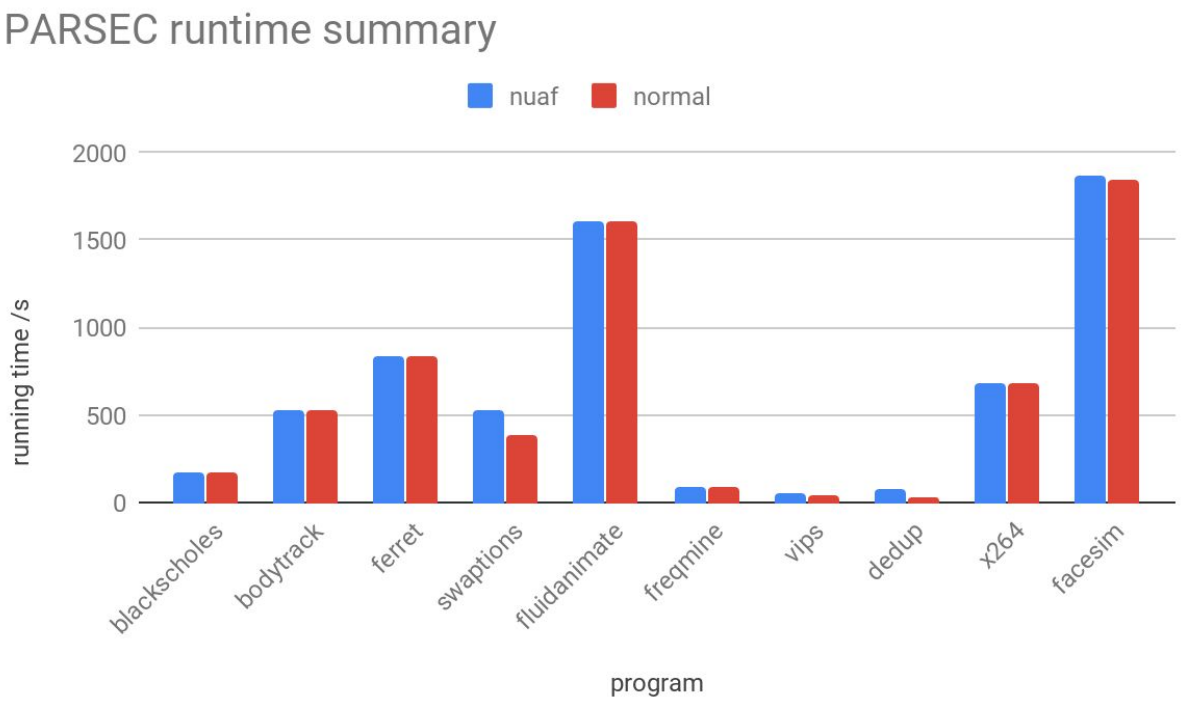
Our program also currently does not have a mechanism in place to handle the case in which we run out of virtual address space. Because our current implementation will cause issues upon the first bit of pre-mapped memory that we run into, we deferred this feature. This will cause issues with long-running allocation-intensive programs.

That being said, our program will work normally in most real-world applications that have normal memory layout, and intense use of memories.

5.4 Overhead

We tested our program on PARSEC 3.0 beta, configed by Charlie Curtsinger. We ran the programs on shannon and clarke, two computers located in the 3814. All the inputs were the

native dataset options, and the programs were configured in single thread mode.



The chart below provides the percent slowdown of nuaf version compared to the normal execution.

program	slow_down_%
blackscholes	-0.0613433623
bodytrack	0.6979557414
ferret	-0.1504535768
swaptions	36.34597347
fluidanimate	-0.08314817927
freqmine	-0.03008132702
vips	19.94431846
dedup	141.3575806
x264	0.3177725101
facesim	1.405861919

Surprisingly, NUAF does not make noticeable time overhead for most of the benchmark. Swaptions and dedup had a large overhead, we suspect that these two programs are memory intensive programs. Also, canneal, one of the Parsec program, got killed everytime we tried to run it. This implies NUAF still uses physical resources more than a normal mode, which may cause problems if there is no monitoring control.

We also ran test on the Concurrent Memory Allocator Benchmarks to look into the effect of multi-threading on NUAF. However, the results are not consistent, showing below:

program	thread	nuaf /s	Normal (glibc) /s
cache-scratch-1	1	18.454	18.55
cache-scratch-1	3	6.177	6.399
cache-scratch-1	5	4.968	5.585
cache-scratch-1	7	4.894	5.064
cache-scratch-1	9	5.131	5.797
cache-thrash	1	18.501	18.432
cache-thrash	3	6.211	6.399
cache-thrash	5	5.11	5.108
cache-thrash	7	4.903	4.86
cache-thrash	9	4.851	4.833
larson	1	15.015	15.003
larson	3	15.046	15.004
larson	5	15.09	15.009
larson	7	15.176	15.012
larson	9	15.606	15.012
threadtest	1	7.784	1.034
threadtest	3	19.607	0.367
threadtest	5	17.817	0.127
threadtest	7	20.007	0.123
threadtest	9	20.163	0.116
cache-scratch-2	1	7.344	7.353
cache-scratch-2	3	2.48	27.72
cache-scratch-2	5	2.01	22.15
cache-scratch-2	9	1.897	4.491

For cache-scratch, two different settings yield dramatically different run-times even for the normal execution. Multi-threads seems to have not big effect on run-time for larson with or without NUAF. And threadtest has a very negative effect on NUAF version. We are not sure why the results are they are now, but a more random setting selections may help to stabilize the result.
The settings can be found from the github repo, `run_CMA.py` script file.

6. Conclusion & Future work

We believe that with further testing and support, this system could be used in real-world systems to prevent use-after-free errors.

6.1 Further Support

With the introduction of 5-level paging, virtual address spaces will grow from 256 TB to 128 PB. Under this change, even long-living, allocation-heavy programs will be able to run for weeks on end, and could simply be restarted when necessary. Additionally, larger TLB's will alleviate some of the TLB pressure associated with this scheme. A feature that would have the most impact on the performance of this scheme would be finer-grained page-permissions (i.e., sub-page permissions). This would allow us to keep one object per sub-page, and this would greatly reduce TLB pressure.

6.2 Further features

The primary feature that we would look to implement would be keeping multiple objects per virtual page. This would necessitate a quarantine list and add some complexity, however the creators of Oscar/Oscar++ [5] found the performance overhead was nearly cut in half in many instances. This would reduces the number of syscalls dramatically, and also lessen TLB pressure. Other features we could include would be adding nondeterminism to our allocator to make it less predictable for attackers, as spatial memory errors could be exploited far easier with our current system, which does not have a ASLR'd heap (although ASLR is a relatively weak protection generally) and is entirely deterministic.

A backtracking feature that would allow further information regarding the UAF error when an unmapped area is touched would also be beneficial. Currently, we just provide a segmentation fault with an address. More debug information would prove beneficial for using this tool in a debugging context.

Integration with a system that provides spatial memory protection would be another thing to consider.

6.3 Takeaways

With no-clobber support, we believe our system could have a real use-case. It provides a low-overhead, perfect protection solution against UAF and double free errors with no source

required. Additionally, since the overhead is merely proportional to the number of objects rather than references, our system only imposes an overhead to allocation-intensive programs. These are very desirable properties. This system could also be used as a verification tool to ensure there are no use-after-free errors along a program's execution path.

Page-protection schemes have historically been looked down upon as workable solutions for real-world problems. We believe that this is a mistake, and they certainly have a place in the security ecosystem. TLB pressure and high syscall usage is unavoidable, but the effects do not appear to be quite as dramatic as one would initially think in the average use-case.

References

1. Berger, E. D., MCKINLEY, K. S., BLUMOFE R. D., WILSON, P. R. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In ASPLOS (2000).
2. BERGER, E. D., MCKINLEY, K. S., ZORN B. Composing High-Performance Memory Allocators. In PLDI (2001).
3. BIENIA, C. Benchmarking Modern multiprocessors. Ph. D. Thesis. Princeton University, January 2011.
4. Dang, T. H. Y., MANIATIS, P., WAGNER, D. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In 26th USENIX Security Symposium (2017).
5. Dang, T. H. Y. Towards Improved Mitigations for Two Attacks on Memory Safety. Ph. D. Dissertation. UC Berkeley, 2017.
6. DHURJATI, D., AND ADVE, V. Efficiently detecting all dangling pointer uses in production servers. In Dependable Systems and Networks (2006), IEEE, pp. 269–280.
7. Gene N., Berger, E. D. DieHarder: Securing the Heap. In CCS (2010).
8. LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing Use-after-free with Dangling Pointers Nullification. In NDSS (2015).
9. LIU, D., Zhang, M., Wang, H. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In CCS (2018).
10. VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: Scalable Use-after-free Detection. In EuroSys (2017), pp. 405–419.
11. YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In NDSS (2015).

* The GitHub repo is here: https://github.com/gaffordb/nuaf