Introduction

As the prevalence of memory-related exploitations increase, researchers have turned to writing custom allocators that emphasize security and means of preventing these exploits. Exploitable memory management errors include buffer overflows/underflows, uninitialized reads, invalid frees, double frees, and dangling pointers, or “use-after-free” attacks. Amongst these errors, use-after-free attacks are the second most prevalent and often the most catastrophic. Secure allocators often rely on randomness to provide a strong security guarantee without compromising performance, and allocators can reduce the likelihood of successful use-after-free attacks by randomizing object reuse or at least delaying reuse. However, even though many “secure” allocators employ randomized or delayed reuse, they often fail to measure the effectiveness, or they do so incorrectly.

That being said, we present two means of measuring the effectiveness of use-after-free mitigation techniques: entroprise and randomness tests. Entroprise measures the entropy of object reuse for allocators that rely on randomization, but it may give insignificant results for allocators that only delay reuse in a deterministic fashion. The randomness tests, specifically the Wald-Wolfowitz runs test [9] and Kolmogorov-Smirnov test [10, 11], are meant to measure the randomness of object reuse, which, although being similar to, is different from entropy, as will be discussed later.

Entroprise

To measure the likelihood of use-after-free attacks, we began with entroprise, which measures the entropy of object reuse. Entroprise works as follows: first, we allocate N objects, and then we free all of them to prime the allocator’s free list or any other data structure used to maintain free objects. Afterwards, we repeatedly call malloc and free back and forth for N iterations. After every cycle of malloc and free, we save the address in an unordered_map and increment the number of times we’ve seen that address thus far. After N iterations, we calculate the entropy using the formula,

$H(X) = -\sum_{k \geq 1} p_k log(p_k)$, where p_k is the probability that the kth object is reused, which is simply the number of times we observed it divided by N.

Initially, entroprise appears sound in practice. However, this implementation can yield incorrect results for allocators that utilize delayed reuse like FreeGuard. When determining objects to reuse, FreeGuard, in reality, only exhibits two bits of entropy, seeing as the only randomness comes from deciding which free list out of four to pull freed objects off of. Within each of these free lists, objects are actually just reused in a deterministic FIFO order [2], making it unlikely we'll encounter the same object again in our implementation of entroprise, which reports that FreeGuard has upwards of thirteen bits of entropy. Clearly, these are insignificant results, and another means of measuring the determinism of use-after-free mitigations is needed.

Wald-Wolfowitz Runs Test and Kolmogorov-Smirnov Test

In response to the drawbacks of the original entroprise, we decided that a better means of measuring determinism would be to utilize a battery of statistical tests normally used for measuring the quality of RNGs. Initially, we attempted to use the tests provided by the dieharder benchmark suite [12], but running them always resulted in the tests failing with a p-value of 0, regardless of the allocator. Instead, we implemented two of these tests themselves: the Wald-Wolfowitz runs test and the Kolmogorov-Smirnov test.

The runs test works as follows: given an array of N addresses, we first find the median. Then, we iterate across the array, counting the number of addresses above the median, addresses below the median, and the total number of runs. A run is defined as a contiguous sequence of either values above the median or values below the median. Additionally, addresses equal to the median are discarded. Finally, we calculate the Z-score, where $\mu = \frac{2n_1n_2}{n_1+n_2} + 1$ and $\sigma^2 = \frac{2n_1n_2(2n_1n_2-n_1-n_2)}{(n_1+n_2)^2(n_1+n_2-1)}$. n_1 is the number of addresses above the median, and n_2 is the number of addresses below the median [9]. Upon calculating the Z-score, we perform a table lookup and return the proper p-value.

We then perform the runs test 100 times in total. Given the natural properties of p-values, their distribution should approximate a uniform distribution. We test for this approximation by performing the Kolmogorov-Smirnov test. This test is simpler than the runs test: first, we sort the 100 p-values. Then, we calculate the highest value in the ECDF above the CDF of the uniform distribution as well as the smallest value in the ECDF below the CDF of the uniform distribution. From there, we simply return the value that is further away from the CDF of the uniform distribution. At the 5% significance level, if the maximum distance between the ECDF and the CDF of the uniform

distribution is greater than 0.565, then we reject the null hypothesis. Otherwise, we fail to reject the null hypothesis [10, 11]. Our null hypothesis states that “this allocator returns an infinitely long, unique sequence of perfectly random addresses.” [12]

Unlike the original entroprise, the runs and KS tests aren’t only checking for object reuse. Rather, they are also checking for how predictable addresses are. For example, if an allocator performs lengthy sequences of returning addresses greater than the median, then an attacker has a better chance of guessing which objects will be reused. Additionally, an allocator can show high entropy with entroprise, but fail miserably during the randomness tests, as will be seen in a moment. Although entropy and randomness are intertwined, they are not necessarily the same.

Security Analysis

Now, we delve into a security breakdown for several allocators, some of which emphasize security whereas others focus on performance and memory overhead. In each section, a summary of its results from running entroprise and KS is provided as well as a summary of its other security countermeasures. A table with a quick overview of the results and security mechanisms of every allocator is provided below, where the reuse entropy and randomness tests were run with 10000 allocations, 16 B objects, and one thread. Note that some of the results are misleading, which will be described in greater detail later in this section.

Security Feature	malloc	DieHarder	FreeGuard	Guarder	Scudo
Segregates heap metadata		✓	✓	✓	
Detects invalid frees	✓	✓	✓	✓	✓
Detects double frees	✓	✓	✓	✓	✓
Guard pages		✓	✓	✓	
Overprovisioning		✓		✓	
Randomized placement		✓	✓	✓	✓
Randomized reuse		✓	✓	✓	
Delayed reuse			✓		✓

Destroy-on-free		✓	✓	✓	
Canaries			✓	✓	✓
Encrypted free lists					
Reuse entropy (bits)	0	12.741	13.286	13.287	13.287
Randomness test result	Failed	Passed	Failed (with changes)	Passed	Failed

Security Feature	jemalloc	tcmalloc	Mesh	smimalloc
Segregates heap metadata	✓	✓		✓
Detects invalid frees		✓		
Detects double frees				✓
Guard pages				✓
Overprovisioning				
Randomized placement			✓	✓
Randomized reuse				
Delayed reuse				✓
Destroy-on-free				
Canaries				
Encrypted free lists				✓
Reuse entropy (bits)	0	0	0	11.857
Randomness test result	Failed	Failed	Failed	Failed

malloc

Seeing as objects are reused in a LIFO order, malloc displays complete determinism regarding object reuse. It consistently demonstrates 0 bits of entropy across all object

sizes, number of objects, and number of threads. Additionally, it returns a p-value of 0 for every iteration of the runs test, causing it to fail KS as well.

Not surprisingly, malloc has the worst regard for security amongst these allocators, which, combined with its ubiquity, makes for a large motivation to improve security within the area of allocators. Malloc's only discernible security countermeasures are the OS's underlying ASLR mechanism providing random addresses from mmap and being able to detect double and invalid frees [13].

DieHarder

Given the results of entroprise and the randomness tests, DieHarder likely has the strongest security guarantee out of the allocators mentioned. One of DieHarder's critiques in "FreeGuard: A Faster Secure Heap Allocator" and "Guarder: A Tunable Secure Allocator" was that as size classes increase, entropy decreases. As shown by the results of entroprise, DieHarder exhibits stable entropy across all sizes, objects, and threads. Additionally, DieHarder outputted an approximately uniform distribution of p-values in the runs and KS tests.

In addition to randomized object reuse, DieHarder also destroys the contents of freed objects to further reduce the effectiveness of use-after-free attacks. To counteract buffer overflows, DieHarder allocates objects randomly, intentionally allocates more objects than will ever be used so that overflows are more likely to affect unused memory, and adopts a sparse page layout, where guard pages are implicitly placed on either side of mapped pages, causing an overflow or overread to crash the application when it tries to write or read unmapped memory. Additionally, DieHarder adopts its replicated mode to detect uninitialized reads, where several copies are made of the same program which "vote" on the correct output. Finally, like all secure allocators, DieHarder segregates heap metadata from objects, detects invalid frees, and detects double frees [1].

FreeGuard

Regarding all the allocators mentioned, FreeGuard has most likely been the most difficult to work with. In reality, FreeGuard only utilizes two bits of entropy, seeing as it only randomly picks one out of four free lists when reusing an object. However, objects are reused in a FIFO order in each freelist, causing entroprise to output high entropy and pass the randomness tests with flying colors. However, we can overcome FreeGuard's FIFO object reuse by priming its free lists with only N objects, but cycling back and forth between malloc and free $N \times 100$ times so that the same FIFO pattern

gets repeated over and over again. With this modification, we observe that although FreeGuard passes the first few runs tests, it ultimately fails the rest, causing it to fail the KS test as well.

As mentioned before, the only source of randomness in FreeGuard's implementation is its random selection of one of four free lists or bump pointers, offering only two bits of entropy. Although FreeGuard reuses objects in a FIFO order, which can make use-after-free attacks more difficult, there still remains complete determinism over object reuse in each free list. FreeGuard also has a destroy-on-free feature like DieHarder, but it is disabled by default, which, under many circumstances, is similar to not having it at all, as many developers have no desire to concern themselves with the internals of secure allocators. To mitigate the effectiveness of buffer overflows, FreeGuard also inserts an 8-bit canary at the end of every object, and upon freeing an object, it checks the object's canary as well as the canaries of the surrounding four objects. However, canaries are only as useful as often as they are checked, and they can double memory consumption in the worst case. Additionally, FreeGuard randomly inserts guard pages into every bag, where every page has a 10% chance of being selected as a guard page. Finally, FreeGuard segregates heap metadata from objects and can detect double as well as invalid frees [2].

Guarder

Amongst the allocators mentioned, Guarder is probably the second most secure, next to DieHarder. Guarder also performed the best with entroprise, often achieving the maximum entropy across any number of objects, number of threads, and object sizes. However, with the randomness tests, Guarder consistently fails the first five iterations of the runs test, outputting a p-value of 0. Upon further examination, Guarder has a tendency of allocating a fairly good mix of addresses above and below the median at first, but as the number of allocations increases, Guarder suddenly stops allocating objects less than the median and only allocates objects above the median. Interestingly, this phenomenon only occurs for about the first 70,000 allocations. After that, Guarder allocates a good mix of addresses above and below the median, ultimately approximating a uniform distribution with its p-values.

Given that Guarder is essentially a continuation of FreeGuard, Guarder and FreeGuard share many security mechanisms including segregation of heap metadata and objects, detecting invalid and double frees, randomized guard pages, canaries, and destroy-on-free, which is still disabled by default. However, Guarder adopts DieHarder's over-provisioning mechanism, and it attempts to introduce more randomization with

object placement and reuse. Guarder also randomizes the order of bags, which is fairly redundant, seeing as objects are already allocated randomly, and customization of the security mechanisms is also introduced. However, customization is only as useful as a user is familiar with the system, and many developers are likely not interested in concerning themselves with entropy and the frequency of guard pages. It is also worth noting that the creators of Guarder may have calculated entropy incorrectly, as described in section 5.4, “Randomization Entropy.” [3]

Scudo

With Scudo, as object sizes and the number of objects increases, entropy appears to decrease. Interestingly, once object sizes reach 4096 B, entropy drops to 0 suddenly, the reason for which is discussed in the next paragraph. It is worth noting that Scudo’s object reuse policy is very similar to FreeGuard’s. Therefore, the results from entroprise should be taken with a grain of salt. For the randomness tests, Scudo consistently fails every iteration of the runs test, which is interesting given that it delays object reuse like FreeGuard, but modifying the randomness tests was not necessary to cause the tests to fail. Upon closer inspection, during the runs test, Scudo appears to go long periods without allocating any object either below the median or above the median. Specifically, it can go as long as 500 allocations without returning an address below the median.

Regarding security mechanisms, Scudo embeds 16-bit canaries within every object’s header. Again, canaries are only as effective as frequently as they are checked, which is when the object is freed. Scudo also utilizes a delayed free list that, by default, only quarantines object sizes up to 2048 B for a maximum of 256 kB on 64-bit machines. Such is the reason that Scudo suddenly drops to no entropy when allocating 4096 B and also why as the number of objects increases, entropy decreases as well, seeing as the delayed free list becomes full and returns all objects to the regular free list. For similar reasons as FreeGuard, a delayed freelist alone is not adequate for preventing use-after-free attacks. Additionally, Scudo does allow you to customize the security options, but many people will not want to concern themselves with testing for the ideal values of these options to suit their needs. Scudo also fails to segregate heap metadata from objects, possibly the first thing to do when writing a secure allocator, though it can detect double and invalid frees [4].

jemalloc

Similar to malloc, jemalloc performs poorly with both entroprise and the randomness tests. Across all sizes, objects, and threads, it exhibits zero entropy, and it outputs a p-value of 0 for all iterations of the runs test.

In addition to ASLR giving random mappings, jemalloc's only other security mechanism is its segregation of objects and heap metadata [5].

tcmalloc

The results of tcmalloc are identical to that of malloc and jemalloc. Both its entropy and p-values are always 0, and objects are reused in a fully deterministic manner.

Like jemalloc, tcmalloc benefits from random mapping from ASLR and segregates heap metadata. Additionally, tcmalloc can detect invalid frees [6].

Mesh

Mesh outputs an entropy of zero for nearly all size classes, numbers of objects, and numbers of threads, except occasionally for 32 kB objects. Unfortunately, it also crashed for a fair chunk of the tests. With the randomness tests, Mesh fails all of them with consistent p-values of 0.

Like other allocators that deal with performance and memory overhead, Mesh is not particularly secure. It does, however, provide randomness with object placement as part of its implementation, but entroprise and the randomness tests only test object reuse randomization. Like all other allocators mentioned, Mesh also benefits from random mappings with ASLR [7].

smimalloc

Again, since smimalloc utilizes a delayed free list like FreeGuard, the results of entroprise are not entirely accurate. As object sizes and the number of objects increase, the entropy of smimalloc dramatically drops, likely a result of the delayed free list becoming full. Additionally, smimalloc completely fails the randomness tests, returning a p-value of 0 for every iteration of the runs test. Upon closer examination, smimalloc suffers from a similar issue as Scudo during the randomness tests. It will go long

periods without allocating any object either above or below the median, going as many as 3000 allocations while only returning addresses above the median.

For security mechanisms, smimalloc places guard pages between every “mimalloc page," which is usually 64 kB, to prevent buffer overflows and overreads. Additionally, object placement is random, and free lists are encrypted using an XOR cipher. Obviously, the effectiveness of this mechanism is dependent on the location of the key, which depends on the randomness provided by ASLR in smimalloc’s case. It also depends on the strength of the encryption, which XOR ciphers alone tend to not be cryptographically strong. Additionally, smimalloc allocates internal objects in a heap separate from application allocated objects, which can prevent an attacker from overwriting virtual method tables to execute arbitrary code. Finally, smimalloc segregates heap metadata from objects, can detect double frees, and employs a delayed freelist as mentioned above [8].

Lessons Learned

In summary, the majority of these allocators have little to no regard for security, even those that claim to be secure. Although malloc, jemalloc, tcmalloc, and Mesh clearly have little concern for security, they are intended to be performance and memory-oriented. On the other hand, Scudo, smimalloc, and FreeGuard claim to offer security, but they fail to implement essential mechanisms like segregating heap metadata and randomizing object reuse. Something that seems to be poorly misunderstood is not only the importance of nondeterminism, but how to create strong nondeterminism. Although Scudo and FreeGuard utilize some sort of delayed object reuse, the reuse policy is still completely deterministic, and in FreeGuard’s case, providing only two bits of entropy is hardly useful. Additionally, although Guarder shows high entropy from entroprise, it consistently fails the first seven iterations of the runs test. Entropy in general appears to be a source of confusion, given that its value wasn’t even calculated correctly in the Guarder paper.

Even when implementing entroprise and the randomness tests, the direction to take wasn’t always obvious. Initially, we started with just measuring the entropy of malloc, DieHarder, FreeGuard, and Guarder, which, on paper, seemed quite straightforward. However, FreeGuard’s object reuse policy gave misleading results, where entroprise would output as high as thirteen bits of entropy when FreeGuard only utilized two bits. Therefore, the next step was to measure the randomness of object reuse. Even until fairly recently, entroprise had been broken when calculating entropy for 32 B objects because the C++ STL had been affecting the results. Additionally, although it has

received little mention in this paper thus far, we must have spent about a month trying to get dieharder to work with addresses instead of just numbers from RNGs until we finally gave up with it and implemented the tests ourselves. Even on our ends, not everything was so clear-cut.

Regarding my personal experience over the semester, essentially everything I know about allocators and research methodology came from this independent study. Looking back at my notes from when I first started, I see questions like “What exactly is a free list?” and “What is a page?”, and I think about the distance I’ve come since starting. However, I believe that I could have done a better job analyzing these allocators if I had seen some of these stated exploitations in practice instead of just reading about how they work in general. Additionally, even though I find it fairly easy to follow the implementation of these allocators when it’s described to me at a high level, the actual implementation is still a mystery to me whenever I try reading source code.

Above all else, I think something that surprised me the most is the seeming lack of care put into writing secure allocators. I believe research in this area suffers from people asking “Why not?” before asking “Why?” Many of these allocators tack on several mechanisms to allegedly make their allocators more “secure”, but much of the time, these mechanisms don’t come together to form a complete solution. Particularly with Scudo and smimalloc, the creators simply took existing allocators that didn’t emphasize security, sanitizer_common and regular mimalloc, tacked on a few features, and advertised them as being “secure” alternatives. Even with those that emphasize security from the beginning like FreeGuard and Guarder, they can fall behind by failing to implement essential mechanisms like randomized object reuse and not answering simple questions like “What is entropy?” I believe that an allocator claiming to be secure, but not being secure, is worse than having no security at all, and such can be the case when researchers overlook important mechanisms like randomized reuse. Entroprise and the randomization tests provide a means of testing the determinism of object reuse, and hopefully researchers can utilize these tools or something similar when measuring the determinism of their allocators before jumping to the conclusion that everything they’ve made is sound in terms of security.

References

[1] Novark, G., & Berger, E. D. (2010). DieHarder. *Proceedings of the 17th ACM Conference on Computer and Communications Security - CCS 10*. doi: 10.1145/1866307.1866371

[2] Silvestro, S., Liu, H., Crosser, C., Lin, Z., & Liu, T. (2017). FreeGuard. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS 17*. doi: 10.1145/3133956.3133957

[3] Silvestro, S., Liu, H., Liu, T., Lin, Z., Liu, T. (2018). Guarder: A Tunable Secure Allocator. *Proceedings of the 27th USENIX Security Symposium - Security 18*.

[4] Scudo Hardened Allocator. (n.d.). Retrieved from http://releases.llvm.org/9.0.0/docs/ScudoHardenedAllocator.html.

[5] Evans, J. (2006, April 16). A Scalable Concurrent malloc(3) Implementation for FreeBSD.

[6] Ghemawat, S. (2007, February 24). TCMalloc: Thread-Caching Malloc. Retrieved from https://gperftools.github.io/gperftools/tcmalloc.html.

[7] Powers, B., Tench, D., Berger, E. D., & McGregor, A. (2019). Mesh: compacting memory management for C/C applications. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*. doi: 10.1145/3314221.3314582

[8] Leijen, D., Zorn, B., & Moura, L. D. (2019). Mimalloc: Free List Sharding in Action. *Programming Languages and Systems Lecture Notes in Computer Science*, 244–265. doi: 10.1007/978-3-030-34175-6_13

[9] 1.3.5.13. Runs Test for Detecting Non-randomness. (n.d.). Retrieved from https://www.itl.nist.gov/div898/handbook/eda/section3/eda35d.htm.

[10] 1.3.5.16. Kolmogorov-Smirnov Goodness-of-Fit Test. (n.d.). Retrieved from https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm.

[11] Santoshh, S. (2016, June). kolmogorov smrinov test in c . Retrieved from https://stackoverflow.com/questions/37625280/kolmogorov-smrinov-test-in-c.

[12] Brown, R. G. (2019). Robert G. Brown's General Tools Page. Retrieved from http://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[13] Lea, D. (2000, April 4). A Memory Allocator. Retrieved from http://gee.cs.oswego.edu/dl/html/malloc.html.