Randomizing Malloc: Stomp the Code Monkeys

Joshua Ervin, Taran Lynn, Aakash Prabhu, Nikhil Yerramilli, and Matt Bishop

University of California at Davis
Davis, CA 95616-8562
USA
{jaervin,tflynn,aakprabhu,yvsami,mabishop}@ucdavis.edu

Abstract. Buffer overflows are still common vulnerabilities. Several techniques deal with stack-based overflows, but less has been done with heap-based overflows. This work examines the vulnerabilities introduced by failing to check for overflows when dealing with allocated memory. The C memory allocation function *malloc* is often a target of these attacks, because a buffer overflow enables an attacker to access the data header that *malloc* uses to track the next memory chunk. This enables the attacker to extract information about about the data layout on the host process, from which they could determine the stack structure, find the location of sensitive variables and functions, and other information useful for furthering the attack.
The assumption that the attacker makes is that *malloc* allocates memory contiguously, so the attacker can overflow a buffer allocated by *malloc* to tamper with the header of the next chunk (which may be allocated or may be on a free list). To invalidate this assumption, we add random amounts of space to the end of each block that *malloc* manipulates. We examine three different methods: the amount is random but constant for all calls to *malloc*, the amount is random but constant for each process' calls to *malloc*, and the amount is random for each call to *malloc*. We analyze each of these methods to determine their theoretical effectiveness, and then validate the efficacy by running attack tools against programs without and with the modification. Finally, we examine how attackers might react to these changes, and how this method may prevent a significant number of attacks.

1 Introduction

When a new vulnerability is discovered, attackers race to exploit it before the vulnerability can be patched without breaking existing functionality. Low level virtual memory allocation services are often victims of these exploits. Here, we study the memory allocation service provided by the standard C library memory allocator, `malloc`, to determine how to harden it against a class of attacks.

Most of the attacks against the *malloc* function take advantage of the data structure that *malloc* uses to manage memory allocation. The typical structure of every memory chunk allocated by *malloc* is shown in figure 1.

When *malloc* allocates space, it takes chunks from a contiguous area of memory, and as memory is allocated and deallocated, chunks of memory are added to and removed from free lists. These chunks of memory are contiguous, and allocated chunks are frequently adjacent to one another. The header of each chunk contains sensitive information such as the size of the current allocation and the address of the next allocated chunk. When an attacker overflows one chunk using standard buffer overflow techniques, the attack could alter subsequent chunks and thus allow the attacker

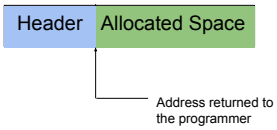


Fig. 1. Data structure used by *malloc*. It returns the address of the beginning of the allocated space.

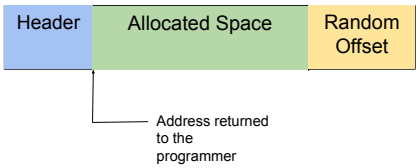


Fig. 2. Modified data structure used by *malloc*. Note the added space at the end. It is allocated by *malloc*, but not included in the size of the chunk.

to modify parts of memory that otherwise they could not modify. This often leads to various heap and stack exploits, including stack pivots.

McDaniel and Nance [10] showed that making a small change to the implementation of *malloc* broke a number of exploits that make assumptions about the memory layout and management performed by *malloc*. They simply added 16 bytes to the end of the allocated space. As the attack tools assumed no such space was added, they failed to compromise the system. The problem with this approach, of course, is that knowing the fixed size offset, the attackers could simply modify the tools to take the extra space into account.

This paper describes an embellishment of the McDaniel-Nance approach. Instead of adding a fixed size offset, we randomize the size of the offset in one of three ways (see Figure 2):

1. Choose a random offset size at system boot and use it for all subsequent calls to *malloc*;
2. Choose a random offset size for every process and use it for all subsequent calls to *malloc* that the process makes; and
3. Choose a random offset for every call to *malloc*.

We analyze the effectiveness of these three methods and show the probability of new attacks successfully compromising the system with our implementation of *malloc*. We also show the tradeoff between the number of attack tools failing and the total memory usage. Finally, we explore the possibility of effectively choosing random offsets using different distributions and analyze the choice of distributions versus the probability of exploits failing.

2 Related Works

Our method perturbs the location of allocated chunks of memory in random ways to thwart attackers. The general idea has been implemented in a number of ways.

Address space layout randomization (ASLR) changes the base address of segments in memory, so attacks that assume segments start at (virtual) address 0 will fail [15]. The new bases are generated randomly, so the strength of ASLR depends in large part on the amount of uncertainty (entropy)

in the selection of the location of the bases. If the entropy is 16 bits, a brute force attack can search the space quickly. But a larger uncertainty, such as 40 bits, will allow a brute force attack to be detected before it succeeds [14]. These require modification of the loader to add the randomness to the (physical) location of the segment base addresses. Variants of ASLR randomize the placement of functions and variables in memory [2]; these require that the compiler be changed also.

Other approaches place special random numbers, called *canaries*, around the allocated memory. The idea is that, after accesses, the canaries are compared to their original values, which are stored somewhere. If they differ, then an overflow or underflow has occurred. For example, StackGuard [3] places a canary on the stack before a function call, so if a buffer overflow overwrites the return address, it will change the canary. On return, the canary's current value is compared to the original value, and if they do not match, a buffer overflow attack may well be under way. PointGuard [4] work similarly, placing canaries next to function and longjmp pointers, and checking that they have not changed whenever they are dereferenced. Cowan *et al.* [5] provide a comprehensive survey of these techniques, and they have been extended to the heap. [11] These approaches require that the compiler be modified to add the canaries and the code to check them.

Another approach is to return chunks at random locations of memory when allocating space. As many attacks depend on chunks being contiguous, this method randomizes the selection of chunks from the appropriate free list, and thereby invalidates the assumption. Garnier [7], Jin [8], and Novark [12] apply different techniques to implement this approach.

McDaniel and Nance [10] based their approach on invalidating the assumption made by attackers, and embedded in their tools, that chunks of dynamically allocated memory are contiguous. They changed the positioning of the chunks by adding 16 extra bytes to each; this causes the existing attack tools to fail. But knowing this offset, attackers could easily modify the tools to succeed.

This approach is similar to other work. Jin [8] adds a small amount of random space at the end of each chunk as well as randomizing chunk selection. Kharbutli [9] uses a separate process to carry out allocation, and in addition to adding random padding between chunks, it also separates heap metadata from the allocated space. Iyer [16] adds padding both before and after the allocated chunk.

In contrast to these other approaches, we add padding to the end of the allocated chunk only. In addition to breaking the assumption of contiguousness, this also causes the location of allocated chunks to be perturbed due to the way the algorithm selects the space on the free list to be allocated. Further, we consider three variations, namely the amount of padding being set randomly at boot time, at process invocation, or at each call to *malloc*. We validate the effectiveness of our work by testing it with the exploits from *how2heap*, and show it successfully blocks those attacks. In addition, we show this method breaks the ordering of chunks in memory due to the additional random space, and has negligible impact on performance.

3 The Memory Allocator *malloc*

In order to understand how heap overflow attacks work, a review of the data structure used by *malloc* will provide necessary background [1].

3.1 How *malloc* Works

The Linux kernel divides memory into several segments. The segment of interest is called the "run time heap" (or simply "heap" for brevity). The space in this segment is used for dynamic memory

allocation, and it is extended by the system call *sbrk*. The C library interface to this system call is *malloc*. Using *malloc* rather than *sbrk* not only improves portability, as it is a standard C library function and hence implemented across a wide variety of systems, but also because it handles the cases where *sbrk* fails by using *mmap* to manage virtual memory.

The function *malloc* allocates space (called *chunks*). In order to keep track of the space, so the process can release chunks, the chunk includes a header. If the chunk is in use, the header consists of 1 word, which includes the size of the allocated space, which is always a multiple of 8. The 3 low order bits of the size, always being 0, are not included; instead, the three bits are used as flags. The flag P indicates whether the previous chunk is in use.[1] If the chunk is free (not in use), the first two words of the data area are pointers, and the last word is the size of the chunk, which is always the same as the size in the header; the low order 3 bits are 0, not flags. Figures 3 and 4 show these layouts.

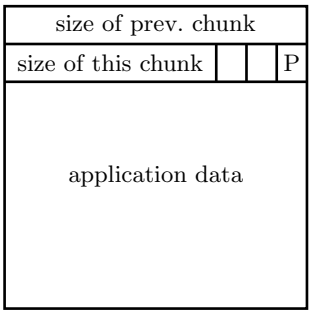


Fig. 3. Layout of an in-use chunk

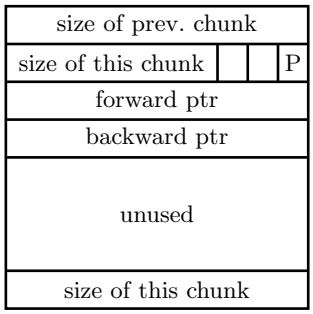


Fig. 4. Layout of a free chunk

The free chunks are organized into a number of free lists. The first, called *fastbin*, is a singly-linked list of small chunks. The second set of lists is a doubly-linked list. Chunks on this list have the first word of the data point to the next chunk in the list (a forward pointer) and the second word of the data point to the previous chunk in the list (a backwards pointer). As noted, the last word of each free chunk is the chunk size; this makes coalescing adjacent free chunks simpler.

When a chunk is removed from a doubly linked list, the forward pointer is copied into the forward pointer of the previous chunk in the list, and the backward pointer is copied into the backward pointer of the next chunk in the list.

3.2 An Attack on *malloc*

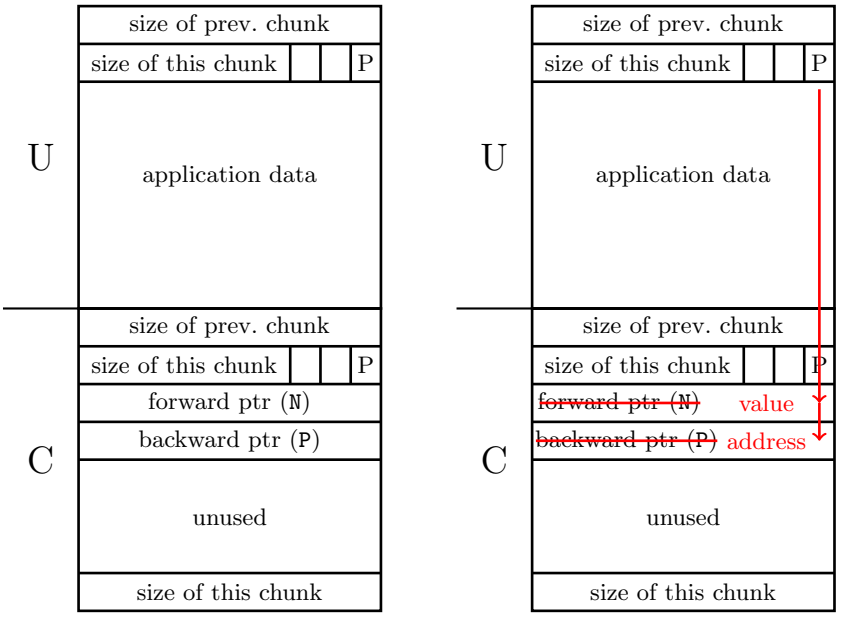
One common set of attacks relies on chunks being adjacent. The attacker writes more into the in-use chunk's space than the size of that space. This overwrites the header of the adjacent chunk. If that chunk is free, the attacker also overwrites the forward and backward pointers in the data area. This enables the attacker to overwrite arbitrary locations in memory.

Suppose we have a free chunk list, and consider the specific chunk `C`. The forward pointer in `C` points to the chunk `P` (for "previous") and the backward pointer in `C` points to the chunk `N` (for

[1] The other two are not relevant to this discussion.

"next"). The chunk preceding chunk `C` *in memory* is in use. The attacker writes more into the in-use chunk than has been allocated to the chunk, and in doing so overwrites the forward and backward pointers of the adjacent chunk, `C`.

The next call to *malloc* that allocates `C` will copy the value in `C`'s forward pointer area to the address, plus an offset, in `C`'s backward pointer area, and vice versa. By appropriately choosing the address and value, the attacker can overwrite writeable addresses in memory.



(a) Memory layout before attack (b) Memory layout after attack

Fig. 5. Layout of chunks before and after the attack

Figure 5 shows this attack. Figure 5(a) shows the initial layout of the chunks. Memory chunks `U` and `C` are adjacent; `U` is allocated and `C` is not. When `C` is allocated, the forward pointer is copied to the chunk at the address given by the backward pointer, and the backward pointer is copied to the chunk at the address given by the forward pointer. In both cases, the actual address changed is the address given by the pointer plus an offset so that the correct fields are changed. When launching the attack, the attacker writes into the application data area of `U`, overflowing it, and overwrites the header of `C`, as shown by the red arrows in Figure 5(b). The overwriting alters the forward pointer with a value (the red value replacing the forward pointer), and the backwards pointer with the address that value is to be written to (the red address replacing the backwards pointer). Then, when `C` is allocated, *malloc* copies the value to the given address (plus the offset). It also copies the address to the location with the address given by the value (plus the offset), so this attack is typically used to alter pointers — that is, the value is an address.

The key here is that the attacker must know the position of the two pointers in the freed chunk with respect to the in-use space. If this position can change, the attacker must compensate for these changes. This suggests adding an offset to the size of the in-use chunk's data area. If it is a random offset, this will complicate the attacker's efforts. The rest of this paper explores this idea.

4 Implementation

Our implementation adds a random number of bytes to the size of each chunk of memory allocated by *malloc*. This increase occurs at the start of the allocation procedure and is invisible to users.

Specifically, in the internal `__libc_malloc` function, we added the line of code as shown in Listing 1.1, where the underlined portion is added (`randm` is short for **randomized malloc**).

```
1  void *
2  __libc_malloc (size_t bytes)
3  {
4    mstate ar_ptr;
5    void *victim;
6
7    bytes += randm_get_offset();
8
9    void *(*hook) (size_t, const void *)
10   ...
11 }
```

Listing 1.1. Adding a random offset to malloc.

`randm_get_offset` returns a random offset amount that varies across the implementations. Each implementation uses a 64 byte seed to initialize the pseudorandom number generator. The seed is initialized and updated in one of three ways.

1. *Per malloc*: On the first call, the seed is initialized by reading from `/dev/urandom`. For purposes of demonstration, each subsequent call it is updated by the following:
   `seed = seed * 6364136223846793005ULL + 1442695040888963407ULL;`
   We use the upper 32 bytes of the result. Note that any (pseudo)random number generator could be used here.
2. *Per process*: When a process first calls *malloc*, the seed is initialized by reading from `/dev/urandom`. It is not updated across subsequent calls, so all *malloc*s for that process use the same offset.
3. *Per boot*: hen a process first calls *malloc*, the seed is initialized by reading from `/tmp/malloc_randm_version`, where *version* is a unique identifier for the current implementation. If the file does not exist, then we obtain a value from `/dev/urandom`, write it to the file, and use that as the seed. If the file exists, we get the seed from it. When the system is booted, all files in `/tmp` are deleted, and a new seed will be generated as described.

For the first, there is an increase in time for each *malloc*(), but the increase is due to 1 to 3 fetches (depending on implementation), a store (of the seed), and 3 arithmetic operations including the extraction of the upper 32 bits. This is marginal and, in the absense of a very larg number of calls to *malloc*, has little to no impact on the run time.

Initially, as space is allocated, the addition of the offset has the effect of adding space between the allocated chunks used by the application. This is because the space making up the offset is invisible to the application. As noted earlier, this decreases the success rate of the attack tools.

It also has another effect, that of compromising assumptions the attack tools make about the ordering of the chunks. Consider a sequence of four allocations by a process for which the offset is 128: (see Listing 1.2).

```
1  a = malloc(100);
2  b = malloc(100);
3  c = malloc(100);
4  d = malloc(100);
5  (void) free(b);
6  (void) free(c);
7  e = malloc(300);
```

Listing 1.2. Order of chunks after intermingled *malloc*s and *free*s

With this offset, the space released by the two *free*s will be $2(h + 100 + 128) = 2h + 456$ bytes, where h is the size of the header. The last *malloc* will allocate $h + 300 + 128 = h + 428$ bytes, and so the chunk for `e` will be put between the chunk for `a` and `d`. Thus, the order of these chunks is `a e d`. However, without the offset, after the two *free*s there is not enough room between the chunks for `a` and`d`, so that chunk will be placed after the chunk for `d`. Thus, the order of these chunks is `a d e`, which is different. Thus attacks depending on where blocks are positions as they overwrite the following chunks will fail, as will those that assume chunks that are contiguous in the version without random offsets are still contiguous with the offset.

The GNU *malloc* handles multi-threading. Multiple heaps are created, each in an *arena*. When a thread calls *malloc*, it selects an arena, locks it, allocates space for the thread within that arena, and then unlocks the arena. It remembers the last arena used. When it needs to allocate more space, it tries to do so in the arena it used last. If that arena is in use, the thread blocks.

A cache called the *tcache* ameliorates this risk. If the *tcache* is enabled, each thread will have a *tcache* containing a set of chunks that can be used without going to an arena. This reduces the probability of blocking until an arena is unlocked.

5 Theoretical Analysis

Assume an adversary knows how our defense works. How likely is the attacker to succeed with this knowledge?

This is essentially a guessing game. The attacker tries to guess the random offset that the system generates. The game is for the system to select a random number, and the attacker tries to guess it. Here, the method chosen for the system to generate the size of the offset is not relevant, as each *malloc*() will have a fixed size for the offset. Thus, we can assume the system's number is constant, and consider how the attacker generates her number. Throughout, we assume there are n possible offsets.

First, the attacker randomly tries different offsets. Then the probability of success is the probability that the attacker will select the same number as the system. Let A and S be the random

variables corresponding to the attacker's and system's selection, respectively. These are independent, so the probability of the two being the same is:

$$\sum_{i=1}^{n} P(A=n, S=n) = \sum_{i=1}^{n} P(A=n)P(S=n) = \sum_{i=1}^{n} \frac{1}{n}\frac{1}{n} = n\frac{1}{n^2} = \frac{1}{n}$$

Let H be the number of attempts that the attacker needs to generate a correct guess. We want to know the probability that the attacker can guess the offset in k attempts or fewer. This is a geometric distribution:

$$P(H \leq k) = 1 - \left(1 - \frac{1}{n}\right)^k = 1 - \left(\frac{n-1}{n}\right)^k = 1 - \frac{(n-1)^k}{n^k} = \frac{n^k - (n-1)^k}{n^k}$$

Next, suppose the attacker selects one value in the range $[1, n]$. Then, given that value, what is the probability that the system's value is the same, is $P(S = n|A = n)$. As A and S are independent, this is the same as $P(S = n) = \frac{1}{n}$.

Thus, in either case, the probability of the attacker choosing the same offset as the space allocated by any particular call to *malloc*() is $\frac{1}{n}$.

Next, consider the space complexity; how much will this increase the space used by a process? We allocate space for the offset in fixed-size increments; let that increment size be b. We also will assume there are m possible offsets; thus, the size of the offsets will be $b, 2b, \ldots, mb$. Finally, assume on the average, *malloc*() allocates s bytes of space for each call.

If the distribution of offsets is uniform and random, the average allocation size s' is:

$$s' = s + \frac{b + 2b + \ldots + mb}{2} = s + \frac{m(m+1)b}{4}$$

Thus, the average increase in space allocated for each *malloc*() is

$$\frac{s'}{s} = 1 + \frac{m(m+1)b}{4s}$$

.

Analyzing this formula, the more possible offsets, the greater the increase in the average space allocated. The amount is proportional the m^2, and linearly proportional to b. This means that, for a program with large average allocations, a large number of possible offsets will not produce a large increase in the space allocated. But if the calls to *malloc* request small amounts of space, a large number of offsets will dramatically increase the amount of memory used. Similarly, if the base size of the offset b is small, the increase will be less than if it were big. This corresponds to intuition.

As examples, if $m = 100$, then the average increase is $1 + \frac{2525b}{s}$. If $m = \frac{s}{b}$, then the average increase in space will be $1 + \frac{s+b}{4b}$. Similarly, if $b = 16$, then the average increase is $1 + \frac{4m(m+1)}{s}$.

6 Results

The goal of the testing is to establish the effectiveness of this defense. We assume the attacker has modified the attack tools to try different offsets until the tool succeeds. The efficacy of the defense is measured by looking at the time it takes for the attack to succeed, and the percentage of successful attacks.

We first created a customized program to try to guess the offsets. We then ran existing attack tools against programs compiled with and without our new version of *malloc*.

6.1 Randomized Guessing

We examined two ways that an attacker might try to guess the offsets. In the first, the attacker guesses from the range of possible offsets, which the attacker could find by reading the source code. In the second, the attacker begins the same way as the first, but once a guess is successful, from then on the attacker uses that number.

For each approach, we ran 1,000,000 tests with allocations of random length in the range of 0–512 bytes, each allocation being a multiple of 16. Additionally, we ran timing tests that allocated 1 GB in 1 KB chunks.

When using randomization, we saw different trends in the number of possible offsets m, execution time t, and the percentage p of successful attacks. These trends are summarized for per *malloc* randomization is shown in Figure 6.

In general, we see that $t \propto m$ and $p \propto \frac{1}{m}$. The first result indicates that allocation time with *malloc* is proportional to the amount of data allocated. The second result is what we would expect for randomized guessing among m objects. Interestingly, keeping the first guess tends to work better than taking a random guess each time. This is because taking the first guess eliminates zero from the range of offsets, giving a higher probability of a hit. When performing the same experiment with per process randomization, we get generally the same curve for completely random guessing, but for keeping the first hit the success rate for attacks rises to over 98%.

Per boot randomization can at least match the success rate of per process. The offset will still be constant for a given process and thus the same techniques may be applied. If it can be determined that a system is using per boot randomization, an even higher success rate may be achieved due to the offset being maintained across multiple processes. The offset would only need to be guessed once, in a single process, and then applied everywhere.

Number of offsets	Time (s)	% Correct (no keep)	(keep)
1	0.31	50.05	98.23
2	0.39	33.31	49.07
4	0.40	19.99	24.98
8	0.44	11.13	12.29
16	0.40	5.91	6.21
32	0.46	3.02	3.16
64	0.53	1.55	1.55

Table 1. Per *malloc* randomization runtime and vulnerability results.

Number of offsets	Time (s)	% Correct (no keep)	(keep)
1	0.32	50.03	98.23
2	0.32	33.40	98.25
4	0.32	20.12	98.24
8	0.36	11.12	98.23
16	0.36	5.92	98.21
32	0.34	3.00	98.24
64	0.54	1.54	98.28

Table 2. Per process randomization runtime and vulnerability results.

Tables 1 and 2 provide a full listing of the data. In those tables, *Time* is the execution time for a program that performs 1,000,000 calls to `malloc` and then 1,000,000 calls to `free`. *% Correct* is the percentage of correct guesses; the *keep* column refers to using only the first offset that was successfully attacked, and the *nokeep* column refers to generating a new offset on each invocation of *malloc*.

6.2 Standard Attacks

Our tests wwith standard attacks used attacks from the "how2heap" repository [13]. We ran these attacks multiple times to see if our approach reliably thwarted them. Figures 7 and 8 show how

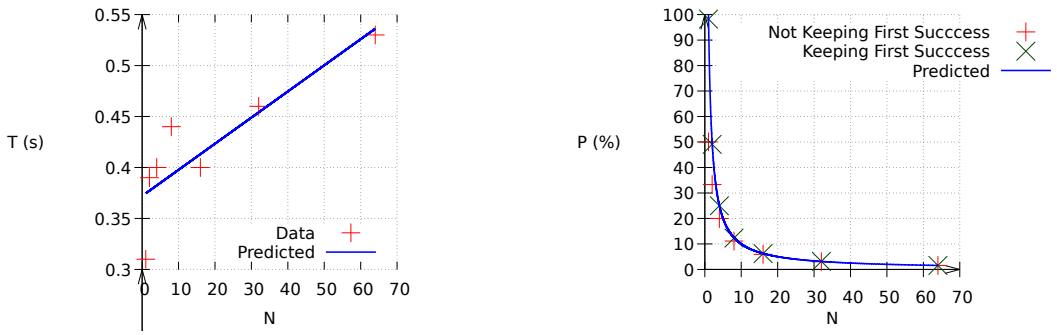


Fig. 6. Results of per *malloc* randomization

attack success relates to different implementations across several standard attacks. A breakdown of the various attacks used can be found on the how2heap repository [13]. The graphs clearly show that adding randomization decreases or in some cases completely mitigates these attacks.

Interestingly, the use of *tcache* did not affect the results. This is because the processes were single-threaded, so the arena associated with the allocation was never locked.

Most of the attacks used were not modified to account for the randomization. The "overlapping chunks" test was split into four types of tests. The first type tested whether there is any overlap at all. The "overlapping chunks unknown size" version behaves the same, but disallows the use of the `malloc_usable_size` function to test the effectiveness if the function had been modified to not include our random buffer in its count – in other words, how effective the attack is without our modification to *malloc*(). An interesting case is the "overlapping chunks 2 full overlap." This test simulated an environment where an attacker could try a guess-and-check method of allocation and, unlike the previous tests, would not show success if there were only a partial overlap of the chunks. The guess-and-check method for this test relied on freeing a chunk and reallocating it if `malloc_usable_size` did not return the desired size. Thus, this implementation depended on being run against the per-*malloc* randomization and would not work if the offset remained constant as each subsequent *malloc* call would be the same. We can clearly see that per-process randomization outperforms per-*malloc* in this test. An alternate guess-and-check method of requesting different size chunks each time may have performed better against the per-process or per-boot randomization. Such a technique would not rely on getting different size chunks when repeating the same request.

Another test of interest is the "fastbin dup into stack" vulnerability, which showed a very small success rate with per-*malloc* randomization but entirely failed against the per-process randomization. One possible explanation is that the attack's success depends on the sum of the various offsets. With per-*malloc*, this is the sum of independent random numbers, while with per-process it is a multiple of one random number. Thus if our sum is n, then the sum of two random numbers is n times more likely to be the correct sum than choosing a single random number whose multiple is the exact sum. This could give a potential advantage to using per-process randomization. In either case the results show a decrease in attacks when *malloc* is modified as suggested.

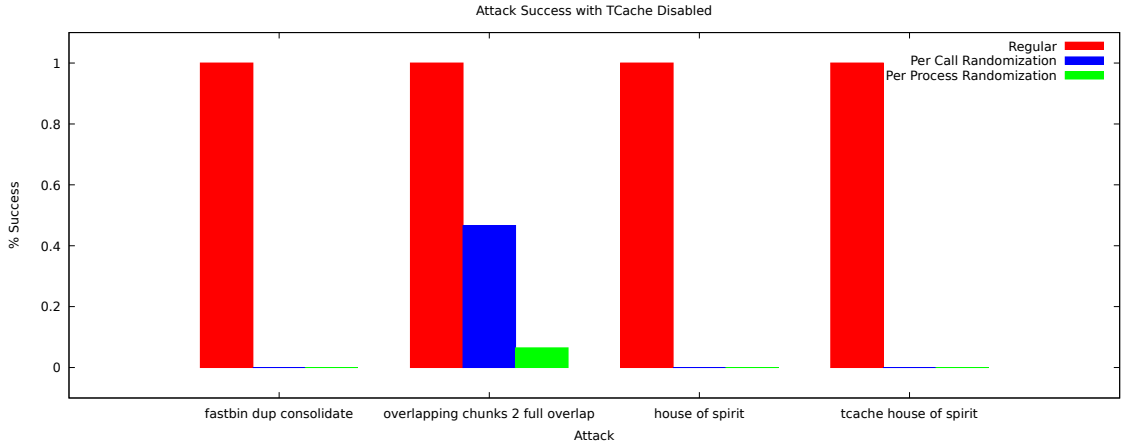


Fig. 7. Attacks against *malloc* with *tcache* disabled

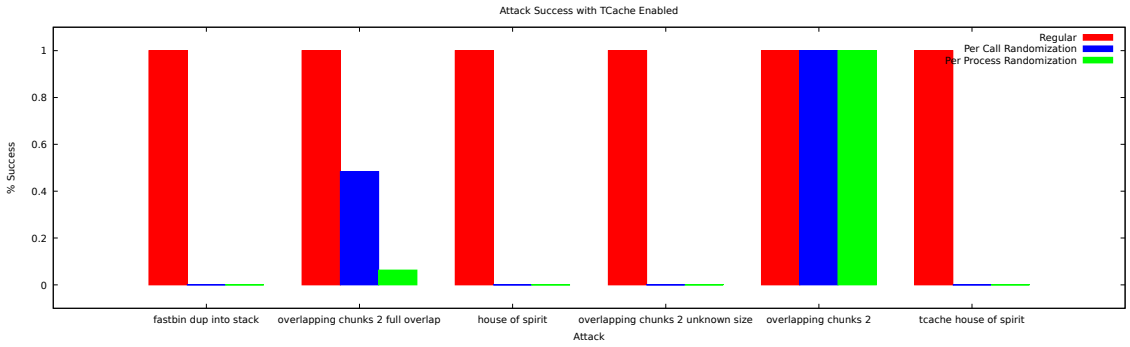


Fig. 8. Attacks against *malloc* with *tcache* enabled.

7 Limitations

This defense focuses on decreasing the probability that two chunks will be contiguous when they would be contiguous were this defense not used. It also focuses on ensuring that the attacker cannot overflow one chunk to overwrite the header of another chunk.

If the attacker can write data into space allocated on the heap, or control the amount of data loaded into space allocated on the heap, two avenues of attacking this defense are possible.

If the attacker does not need to know where the header of the next chunk is positioned in memory, the attacker can use heap spraying to overwrite locations in memory with the values desired to replace the header data in the next chunk. The attack will succeed assuming the alignment of the attacker's data matches that of the data in the chunk's header that is to be replaced. This defense is not intended to prevent such attacks, and can be combined with defenses similar to BuBBle [6] but that look for large parts of memory containing the same data (here, the addresses to overwrite).

In some cases, the attacker can determine the amount of padding added to each allocated chunk. Roughly, the attacker deliberately overflows the allocated memory. If the excess does not exceed the

added padding, the program will continue to function correctly. If the excess does extend beyond the padding into the next chunk's header, the process is likely to malfunction or crash at a later time. By increasing the length of the data until the process malfunctions or crashes, the attacker can determine where the next chunk begins. This will work when the amount of padding is determined at boot time. If the process does not crash — for example, when the process is a server that forks to service clients — this technique will also work against per-process random padding. But it will not work when per *malloc* padding is used.

8 Conclusion

Many attacks on allocation systems rely on being able to find and modify the header information for certain allocations. It has been shown that adding a fixed offset before the header can mitigate many of these attacks [10]. However, this can be overcome by taking into account the fixed offset when designing attacks. Randomizing the size of the offsets counters this problem.

Several interesting patterns became apparent when testing this approach. First, if an attacker guesses randomly, the probability of a successful attack is $\frac{1}{m}$ for all implementations, where m is the number of possible offsets. Second, if the attacker keeps using the first successful offset, then per call success rate remains at $\frac{1}{m}$, while other implementations can rise to a 100% success rate. Third, per call randomization can actually perform worse than per process or per boot randomization if certain relationships among multiple *malloc*s offsets are met. For example, if two calls to *malloc* produce offsets that have a certain sum, then per process randomization might outperform per *malloc* randomization.

In terms of performance we can expect the space complexity to have an overhead of $1 + \frac{m(m+1)b}{4s}$. This can lead to large increases in memory usage for processes that make many small allocations. For processes that make large allocations, the overhead will be small. For example, in our experiments, $m = b = 16$, where b is the increment for the size of the m offsets, so there is less than a 0.1% increase when allocations have a size of $s = 128$kB. Time complexity also increases linearly with offset size, as it is linearly related to total memory usage.

Overall, adding offsets can mitigate or reduce attack success by an order of magnitude. However, the space/time costs can overtake these benefits. It is therefore preferable to use this technique with programs that make large allocations, such as back-end code that manipulates large data sets.

This approach can be improved in several ways. First, because attack success is equal to $\frac{1}{m}$ but space/time complexity is proportional to mb, decreasing b could lead to greater gains. Unfortunately, the version of *malloc* used requires all allocated chunks to be multiple of 16 bytes, so b must be a multiple of 16. Reducing the offset would require a rewrite of *malloc*, which is outside the scope of this work. It would be a good way to increase the entropy of the size of the offset, and is ripe for future work.

Second, per-process randomization has been shown to be more effective than per-call randomization under specific circumstances. More research into determining why some attacks are more successful with one method than another could produce some interesting characterizations and insights into defenses. It would also allow a simple type of moving target defense, in which the version of *malloc* called would be controlled by the characteristics of the process calling it. How to determine whether a process meets the characteristics for a particular randomized *malloc*() defense depends on what the characteristic are and how best to determine them.

Finally, this work dealt only with a single-threaded process. Repeating this experiment, using multi-threaded processes, would enable the effects of the *tcache*, and how that caching improves — or inhibits — the existing attacks to be understood.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grants No. DGE-1934279 and OAC-1739025 to the University of California at Davis. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the University of California.

References

1. Malloc internals, May 2019.
2. Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 15th USENIX Security Symposium*, pages 255–270, Berkeley, CA, USA, July 2005. USENIX Association.
3. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Seventh USENIX UNIX Security Symposium*, pages 63–77, Berkeley, CA, USA, January 1998. USENIX Association.
4. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard$^{\text{TM}}$: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Berkeley, CA, USA, August 2003. USENIX Association.
5. Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the Foundations of Intrusion Tolerant Systems*, OASIS '03, pages 227–237, Los Alamitos, CA, USA USA, December 2003. IEEE Computer Society.
6. Francesco Gadaleta, Yve Younan, and Wouter Joosen. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In Fabio Masacci, Dan Wallach, and Nicola Zannone, editors, *Proceedings of the Secnd International Symposium on Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, Germany, February 2010. Springer Berlin Heidelberg.
7. Thomas Garnier. Randomizing the linux kernel heap freelists, September 2016.
8. Zhiyong Jin, Yongfu Chen, Tian Liu, Kai Li, Zhenting Wang, and Jiongzhi Zheng. A novel and fine-grained heap randomization allocation strategy for effectively alleviating heap buffer overflow vulnerabilities. In *Proceedings of the 2019 Fourth International Conference on Mathematics and Artificial Intelligence*, pages 115–122, New York, NY, USA, April 2019. ACM.
9. Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. *ACM SIGARCH Computer Architecture News*, 34(5):207–218, October 2006.
10. Lucas McDaniel and Kara L. Nance. Mitigating 0-days through heap techniques - an empirical study. In Tung X. Bui and Ralph H. Sprague Jr., editors, *49th Hawaii International Conference on System Sciences, HICSS 2016, Koloa, HI, USA, January 5-8, 2016*, pages 5569–5577. IEEE Computer Society, 2016.
11. Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors, *Proceedings of the Tenth Internationa Conference on the Detection of Intrusion and Malware, and Vulnerability Assessment*, volume 7967 of *Lecture Notes in Computer Science*, pages 177–196, Berlin, Germany, July 2013. Springer Berlin Heidelberg.

12. Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 573–584, New York, NY, USA, October 2010. ACM.
13. Shellphish (pseudonym). how2heap.
14. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomizaton. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307, New York, NY, USA, October 2004. ACM.
15. PaX Team. Address space layout randomization, July 2001.
16. Iyer Vivek, Amit Kanitkar, Partha Dasgupta, and Raghunathan Srinivasan. Preventing overflow attacks by memory randomization. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, Piscataway, NJ, USA, November 2010. IEEE.