A Heap of Trouble: Breaking the Linux Kernel SLOB Allocator

Dan Rosenberg

Virtual Security Research

drosenberg@vsecurity.com

January 22, 2012

1 Introduction

In this paper, we will systematically evaluate the implementation of the Linux kernel SLOB allocator to assess exploitability. We will present new techniques for attacking the SLOB allocator, whose exploitation has not been publicly described. These techniques will apply to exploitation scenarios that become progressively more constrained, starting with an arbitrary-length, arbitrary-contents heap overflow and concluding with an off-by-one `NULL` byte overflow.

2 SLOB Allocator Overview

2.1 Introduction to SLOB

Written by Matt Mackall, the SLOB allocator was committed to the upstream Linux kernel in Januray 2006 [1]. It is intended for usage in systems requiring a smaller memory footprint, especially embedded platforms. To accomplish this design goal, it uses a much simpler allocation scheme than the earlier SLAB allocator.

The SLOB allocator is in active usage in a number of widely distributed Linux kernels. Embedded Gentoo [2] uses the SLOB allocator by default, as do several kernel configurations provided by the OpenEmbedded Project [3]. The OpenWrt Project [4], designed to run on IP routers, uses the SLOB allocator, as do many commercial embedded devices. Notably, the mobile space has not seen much adoption of SLOB, perhaps due to the fact that modern smartphones no longer have significantly constrained memory footprint requirements.

2.2 Implementation

2.2.1 Page Metadata

The basic paradigm of slab allocation (not to be confused with SLAB, which is a specific implementation of a slab allocator) involves preallocating caches of contiguous pieces of memory designed to efficiently service allocation requests for objects of a certain type or size. SLOB does not exactly fit into this model, but like most slab allocators, SLOB organizes memory into pages. Allocations greater than a page size (including metadata) are passed to the page frame allocator directly, via a call to `alloc_pages()`. For smaller allocations, SLOB maintains three singly-linked lists of partially-allocated pages, each of which services requests for allocations of different sizes: less than 256 bytes, less than 1024 bytes, and all other objects less than a page size[1]:

```
#define SLOB_BREAK1 256
#define SLOB_BREAK2 1024
static LIST_HEAD(free_slob_small);
static LIST_HEAD(free_slob_medium);
static LIST_HEAD(free_slob_large);
```

Each page is represented by a `struct slob_page`, which is defined as a union with an actual `struct page`:

```
struct slob_page {
    union {
        struct {
            unsigned long flags;    /* mandatory */
            atomic_t _count;        /* mandatory */
            slobidx_t units;        /* free units left in page */
            unsigned long pad[2];
            slob_t *free;           /* first free slob_t in page */
            struct list_head list;  /* linked list of free pages */
        };
        struct page page;
    };
};
```

Each SLOB page is broken into individual chunks, which are referred to as *blocks*. Blocks consists of one or more `SLOB_UNIT`s, which are typically 2-byte measurements[2]. Initially, a SLOB page contains a single free block, which is fragmented as necessary to service smaller request sizes. Note that SLOB pages contain blocks of varying sizes, which differentiates SLOB from a classic slab allocator.

[1] All code samples are from the Linux kernel version 3.2.

[2] On kernels configured to support page sizes of 64KB or greater, `SLOB_UNIT` is a 4-byte measurement, but because the vast majority of embedded devices use smaller page sizes, this configuration is not in the scope of this paper.

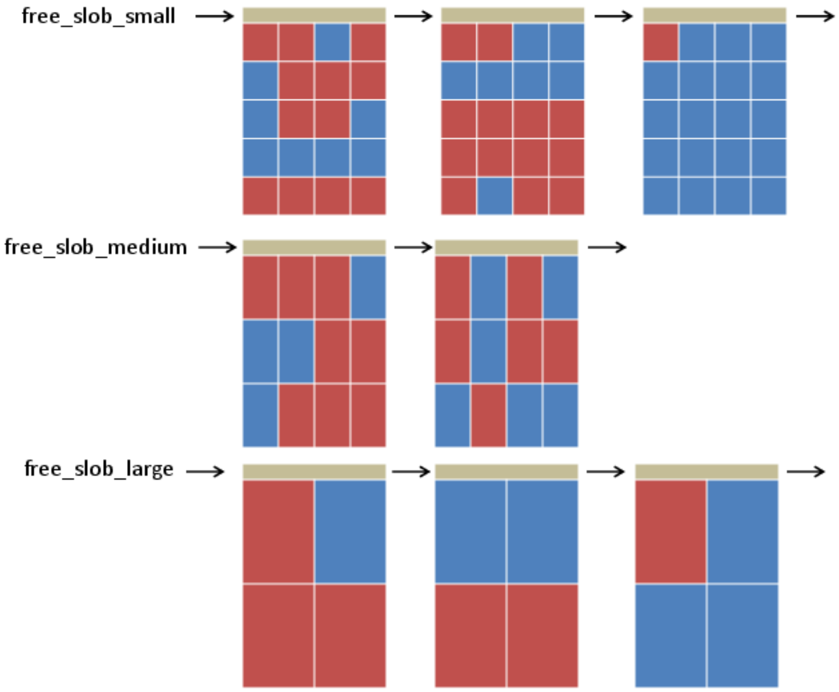

Figure 1: SLOB maintains three linked lists of partially-filled pages.

2.2.2 Block Metadata

Both free and allocated blocks include inline metadata. Free blocks containing more than one `SLOB_UNIT` of data include a 4-byte header, where the first two bytes contain the size of the block, and the second two bytes contain the offset (in `SLOB_UNIT`s) of the next free block from the base of the current page. Free blocks containing a single unit of space (referred to in this paper as "small" blocks) have a 2-byte header containing the offset of the next free block from the base of the current page, multiplied by negative one. Finally, allocated blocks include a 4-byte header that contains the size of the allocated block.

2.2.3 Allocation

To perform an allocation, SLOB first iterates through the linked list of pages responsible for servicing allocations of the requested size. The `units` field for each `slob_page` is checked against the requested allocation size. If the total amount of space available in the page is sufficient, the allocation is attempted. Note that the reported space in a SLOB page is not necessarily contiguous, so the allocation is not guaranteed to succeed.

When attempting to allocate a block within a page reporting sufficient

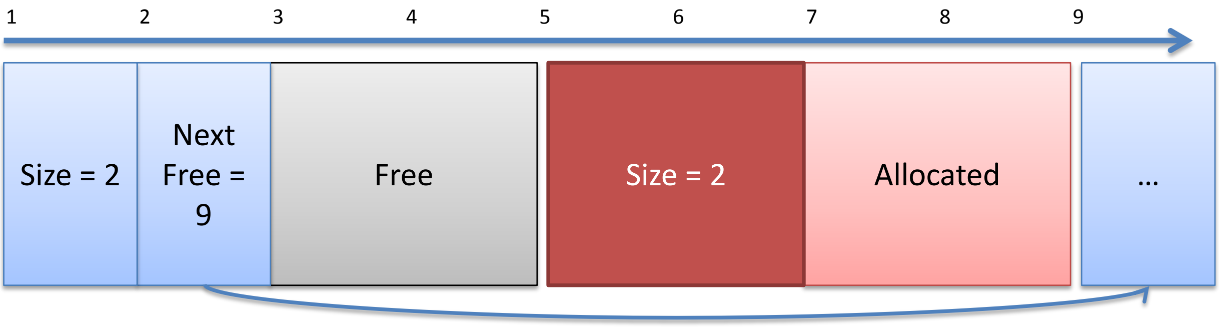

Figure 2: Example of SLOB blocks. Units are `SLOB_UNIT`s (2 bytes).

total space, a simple first-fit algorithm is used. SLOB iterates over the freelist for the chosen page, checking the reported size of each free block based on its header as follows:

```
static slobidx_t slob_units(slob_t *s)
{
    if (s->units > 0)
        return s->units;
    return 1;
}
```

If the size of a particular block is insufficient, the next block in the freelist is retrieved by adding the next-free index of the current block to the base address of the current page, as follows:

```
static slob_t *slob_next(slob_t *s)
{
    slob_t *base = (slob_t *)((unsigned long)s & PAGE_MASK);
    slobidx_t next;

    if (s[0].units < 0)
        next = -s[0].units;
    else
        next = s[1].units;
    return base+next;
}
```

If a suitable block is found, it is fragmented as necessary (if it is sufficiently large) and unlinked from the freelist, prepended with a 4-byte size header, and returned. Unlinking a block from the freelist involves adjusting the next-free index of the previous free block to point to the free block after the block that is being allocated. Note that due to the simplicity of the singly-linked freelist, classic unlink-style attacks leading to a "write4" condition are not possible.

If the end of the freelist is reached (determined by reaching a free block that is page-aligned, typically a `NULL` address) and no suitable block has been found, the next page reporting sufficient space is consulted. If no pages can service the request, a fresh page is allocated. Appropriate page flags are adjusted, for example if a partially full page is now full as a result of the allocation. Finally, as an optimization, the appropriate linked list of pages is cycled such that the next search will begin at the page used to successfully service the most recent allocation.

2.2.4 Freeing

Freeing of a SLOB block uses a similarly simple algorithm. If the page that the block belongs to is full (completely allocated), then the freelist pointer for that page is updated to point to the block and the page is reinserted into the appropriate linked list of partially full pages. The `units` field of the `slob_page` is also updated to contain the size of the freed block.

If the page that the block belongs to is already partially allocated, SLOB must find a suitable point to reinsert the block into the existing freelist. This is done as an address-ordered first fit. In other words, the freelist is walked until a block is found with an address higher than the just-freed block. At this point, SLOB checks whether or not the just-freed block is contiguous with the previous or next free blocks in the list. If so, the contiguous free blocks are merged into a single larger block. Finally, all relevant size and next-free index fields are modified appropriately.

The simplicity of SLOB merging does not provide significant new vectors to an attacker. When two consecutive free chunks are merged, the only adjustment that takes place is to set the size field of the first chunk to the new total size, and to alter the next-free index of the first chunk to contain the value previously contained in the second chunk's next-free index.

3 Pre-Exploitation

For most heap exploits, massaging the heap into an exploitable state is essential for successful exploitation. SLOB has several advantages over SLUB in terms of an attacker's ability to control the allocation and freeing of adjacent heap structures. One major benefit from the perspective of an attacker is that the SLOB pages segregate blocks based on a wider range of allocation sizes. Additionally, objects created in a special-purpose cache using `kmem_cache_alloc()` are tracked as belonging to that special-purpose cache but actually reside in ordinary SLOB pages, co-resident with general-purpose blocks allocated using `kmalloc()`. As a result, all allocations under 256 bytes will be serviced from the same SLOB page, giving us a much greater degree of flexibility in choosing heap primitives during exploitation.

Finally, unlike SLUB, the SLOB allocator has global rather than per-CPU caches, so we do not need to concern ourselves with binding our exploit process to a particular CPU.

3.1 Massaging the Kernel Heap

If lack of available space in existing SLOB pages forces the allocation of a fresh page, subsequent allocations will return adjacent blocks within this page. Accordingly, the first step in being able to control the heap layout for exploitation is to fill existing SLOB pages. Just as with SLUB, this can be accomplished by triggering the repeated allocation of a persistent heap object with an appropriate size. Unfortunately, the `/proc/slabinfo` interface, which provides debugging information for the SLAB and SLUB allocators and is useful for exploitation, is not available for SLOB. This interface may soon be unavailable for SLUB/SLAB exploits as well [5]. As a result, a blind approach must be taken, where the attacker makes a reasonable guess as to how many allocations are necessary. It was experimentally determined that a few hundred allocations, even of a small block size, is sufficient to exhaust partially-filled pages in all but the most contrived scenarios.

As mentioned in Section 2.2.3, each linked list of pages is cycled such that the next search begins at the page most recently used to service an allocation for that size range. As a result, it is possible to trigger allocations of large objects within a particular list, and after the allocation of a fresh page, smaller allocations will still occur consecutively within that page despite the fact that other partially-full pages may be able to service these smaller allocations. This may reduce the number of allocations required during pre-exploitation.

4 Exploitation

The SLOB allocator presents a number of vectors that are ideal for exploitation. We will enumerate and describe several techniques here, focusing on those that do not have analogous techniques that apply to the SLAB and SLUB allocators.

4.1 Attacks on Object Data

4.1.1 Block Data Overwrite

In the event of a heap overflow with sufficient attacker control of the overflow length and contents, one option is to attempt to overwrite the data of an adjacent allocated block. This technique has been thoroughly described by twiz and sgrakkyu [6].

In the examples in this paper, the target block is a structure with triggerable function pointers, but other objects may also be used, such as structures with pointers through which a kernel write may be triggered, allowing the creation of an arbitrary kernel memory write exploit primitive. In all cases, the end result is achieving control of kernel execution flow.

The classic attack has the following steps:

1. Repeatedly trigger the allocation of appropriately sized blocks to force the allocation of a fresh page.

2. Trigger the allocation of two consecutive blocks. The first of these should be a block for which we can cause a heap overflow (the "vulnerable" block). The second block should be any object for which control of its contents could lead to code execution, such as an object with triggerable function pointers (the "target" block).

3. Trigger the overflow, overwriting the target block with attacker-controlled data.

4. Trigger the invocation of a function pointer in the target block.

SLOB introduces one extra hurdle when using this approach, as compared to SLUB. Because allocated blocks in SLOB contain a 4-byte size header, an attacker leveraging this technique must take care to adjust this value to keep the heap in a consistent state. Note that overwriting this size field with a value smaller than or equal to its original value does no harm; because the size field does not directly affect the freelist, the only condition we need to avoid is growing a block such that it corrupts adjacent blocks.

If the attacker has sufficient control of the overflow, it may be possible to overwrite this size field with a four-byte value equal to or less than its original value, after which no further adjustments need to be made. If this is not possible, an attacker may complete the exploit attempt and repair this value post-exploitation, prior to causing the overflowed object to be freed. Failure to restore this field may result in heap corruption, and may ultimately cause a kernel panic if SLOB attempts to service subsequent allocations near the corrupted block.

4.2 Attacks on Object Metadata

4.2.1 Free Pointer Overwrite

Unlike the SLUB allocator, where free objects have an actual pointer to the next free object in the current slab, SLOB free objects contain indexes relative to the base of the SLOB page. Additionally, these indexes are 16-bit values. As a result of these constraints, it is not possible to overwrite the free pointer of an adjacent free object and subsequently cause an allocation to return a fake object residing in userspace, as has been demonstrated with other slab allocators.

However, it is possible to make some adjustments to the "Overwrite-into-Free-Object-Metadata" technique previously described by sgrakkyu and twiz [7] in order to apply it to the SLOB allocator. The basic idea is that if an attacker can cause SLOB to allocate an object he controls on top of some other target object, it would be possible to overwrite the contents of the target object (and gain control of execution). The following attack may be used in the event of a controllable three or four byte heap overflow (see Figure 3):

1. Repeatedly trigger the allocation of appropriately sized blocks to force the allocation of a fresh page.

2. Cause the allocation of multiple target objects, followed by a vulnerable object. At this point, the vulnerable object will be allocated immediately before a free block.

3. Trigger the overflow, overwriting both the size and next-free index of the adjacent free block with a value chosen to cause the next-free index to point to an earlier allocated block (containing a target object). If possible, when overwriting the size value, use a small enough value so that the free block with overwritten metadata is skipped when traversing the freelist for the next allocation.

4. Trigger an allocation, which will result in an attacker-controlled object being placed directly on top of a target block. If it was not possible to overwrite the adjacent free chunk's size header with a sufficiently small value, two allocations may be required: the first will allocate the free chunk whose header we overflowed, and the second will allocate our attacker-controlled object.

5. Provide suitable data to be copied into the newly allocated block, which resides in the same memory region as the target block.

6. Trigger the invocation of a function pointer in the target block, which now contains attacker-controlled data.

This attack may be adjusted to work on both little and big endian architectures.

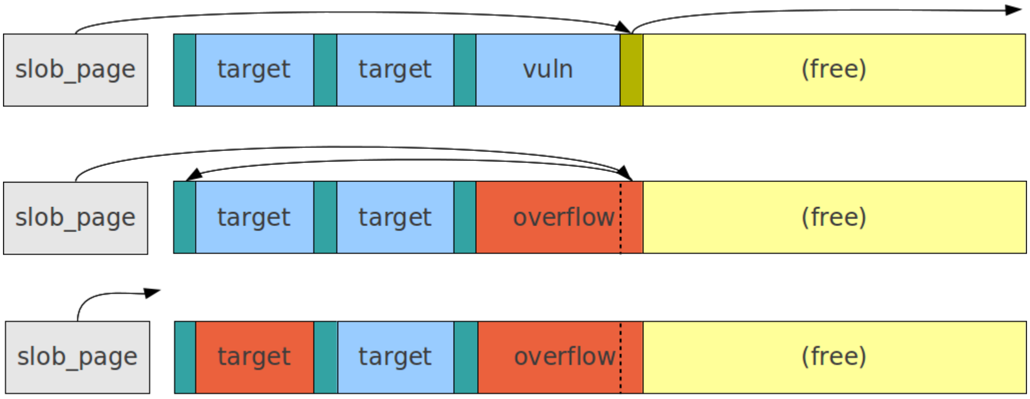

Figure 3: Free pointer overwrite.

4.2.2 Fake Small Block Attack

In the event where we only have a controllable one byte (on big endian) or two byte (on little endian) overflow, a variation on the above attack is possible. Recall that if the first two bytes of a free block's header is a negative short value, then the block is understood to be a "small" block one `SLOB_UNIT` in size and the header value is used as the negative index to the next free block. In other words, by causing the first two bytes of a free block's header to be negative, we can corrupt the freelist and trigger the same condition as above.

The steps of the free pointer overwrite can be replicated exactly, except instead of overwriting three or four bytes of a free block's header, which includes both the size field and next-free index, we only overwrite the first one or two bytes. On big endian architectures, we can overwrite the first byte with an `0xff` character, at which point the first two bytes will be interpreted as a negative short value belonging to a fake small block and subsequent allocations will be placed on top of preexisting data. On little endian architectures, overwriting both of the first two bytes is necessary to produce a negative index and create our fake small block.

4.2.3 Block Growth Attack

The presence of the 4-byte size header in allocated blocks or the 2-byte size header in free blocks provides a new target that may be exploited in the case of an off-by-one or other small overflow. Since this field is used by SLOB to determine the amount of space available for an allocation, the following attack may be performed (see Figure 4):

1. Repeatedly trigger the allocation of appropriately sized blocks to force the allocation of a fresh page.

2. Trigger the allocation of three consecutive blocks. The first of these should be the vulnerable block, for which we can cause an off-by-small heap overflow. The second block is a dummy block that may be any object that resides in the same SLOB list. The third block should be the target object for which control of its contents could lead to code execution.

3. Trigger the heap overflow in the first block. Overwrite the first byte(s) of the adjacent dummy block's size field to contain a value larger than the initial contents.

4. Trigger the freeing of the dummy block. This will result in SLOB misinterpreting the region containing both the dummy block and the target block as a single large free block.

5. Trigger the allocation of a fourth object whose contents you can control and is larger than the original dummy block. This allocation will return a heap block that overlaps with the data stored in the target block.

6. Provide suitable data to be copied into the new block, which resides in the same memory region as the target block.

7. Trigger the invocation of a function pointer in the target block, which now contains attacker-controlled data.

This attack is applicable to both little and big endian systems.

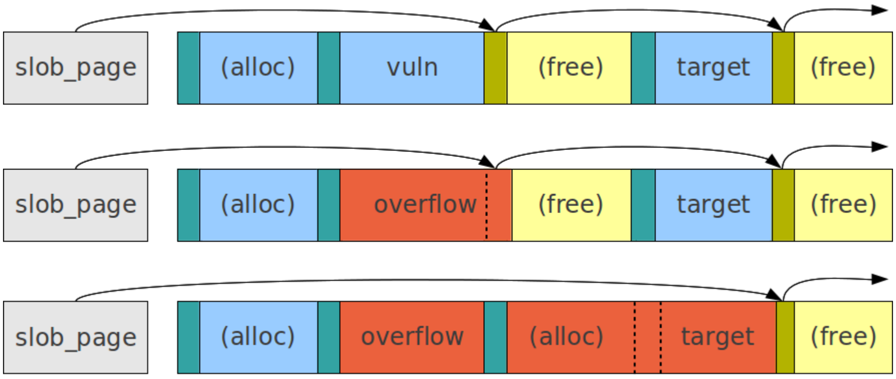

Figure 4: Block growth attack.

4.2.4 Little Endian Block Fragmentation Attack

Perhaps the most constrained heap overflow scenario is an off-by-one `NULL` byte overflow. In this case, it is no longer useful to overwrite the least significant byte of an allocated or free block's size header, since the end result would be harmless block shrinkage with no freelist corruption.

As a result, the only option in this case is to overwrite the first byte in a "small" free block's header. Keeping in mind that small free blocks have a two byte header that represents the negative index of the next free block, overwriting the first byte with `NULL` adjusts the index to point to a higher address within the page (on a little endian machine). At this point, exploitation resembles previous examples involving freelist corruption, where the end goal is to cause an allocation of attacker-controlled data on top of a useful target.

In order to perform this attack, we need to be able to cause a small free block to reside immediately after a vulnerable block. Because of the minimum granularity of SLOB, which in practice is almost always eight bytes, it's not possible to cause an allocation that, when freed, would be stored as a small free block. Instead, we have to rely on SLOB fragmentation to produce the small free block we wish to overflow. The attack can be performed as follows (see Figure 5):

1. Repeatedly trigger the allocation of appropriately sized blocks to force the allocation of a fresh page.

2. Trigger the allocation of three dummy objects. The first should be chosen to be four bytes greater in size than our vulnerable object. This should be followed by two additional dummy objects of any relatively small size, followed by several target objects.

3. Trigger the freeing of the last of the three dummy objects.

4. Trigger the freeing of the first dummy object.

5. Trigger the allocation of our vulnerable object, which will cause the fragmentation of the space formerly occupied by the first dummy object. As a result, our allocated vulnerable object will be immediately followed by a small free block.

6. Trigger the overflow into the small free block, overwriting the least significant byte and corrupting the freelist.

7. Trigger the allocation of a new attacker-controlled block, which will be placed in the same memory region as one of the target blocks.

8. Provide suitable data to be copied into the newly allocated block.

9. Trigger the invocation of a function pointer in the target block, which now contains attacker-controlled data.

The reason for two additional dummy blocks beyond the one that is fragmented is subtle. Because we can only overflow the small free block's header with a NULL byte, our overflow will adjust the next-free index of the small free block to point at a higher address in the page than its original value. Therefore, it's necessary that the small free block's next-free index initially points to a free block that resides before our target blocks in memory, such that when our overflow occurs the next-free index is changed to point into the target blocks. To accomplish this, we need to place the two additional dummy blocks prior to the target blocks, as described above, and free the second of these blocks. Note that a single block is insufficient, because if the block immediately after our small free block is another free block, the two blocks will be merged into a single larger free block and exploitation would become impossible.

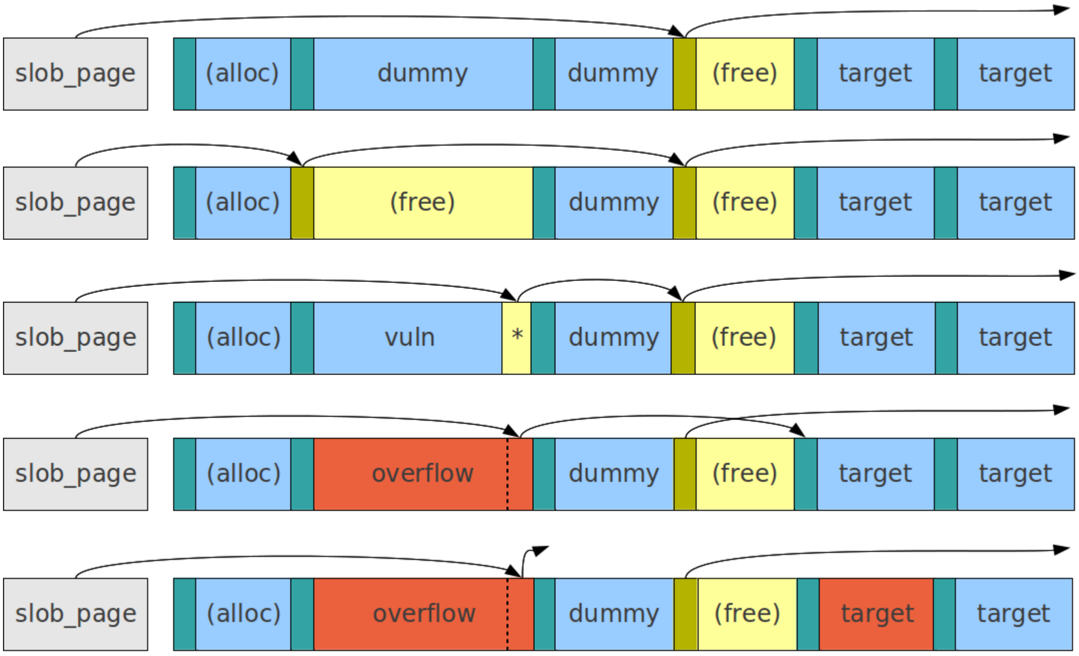

Figure 5: Little endian fragmentation attack.

4.2.5 Big Endian Block Fragmentation Attack

The previous attack will not work as described on big endian architectures, because on these systems the first byte of a small free block's header is the most significant byte, and overwriting this with a NULL byte causes the first two bytes to be interpreted as a positive value in an ordinary (not small) free block.

However, causing a small free block to be interpreted as a larger free block creates a situation identical to the block growth attack. The full attack can be carried out as follows (see Figure 6):

1. Repeatedly trigger the allocation of appropriately sized blocks to force the allocation of a fresh page.

2. Trigger the allocation of a dummy object, chosen to be four bytes greater in size than our vulnerable object, followed by a target object.

3. Trigger the freeing of the dummy object.

4. Trigger the allocation of our vulnerable object, which will cause the fragmentation of the space formerly occupied by the dummy object. As a result, our allocated vulnerable object will be immediately followed by a small free block.

5. Trigger the overflow into the small free block, overwriting the most significant byte. This results in the previously negative short value contained in the first two bytes to become a positive value corresponding to the block's new reported size.

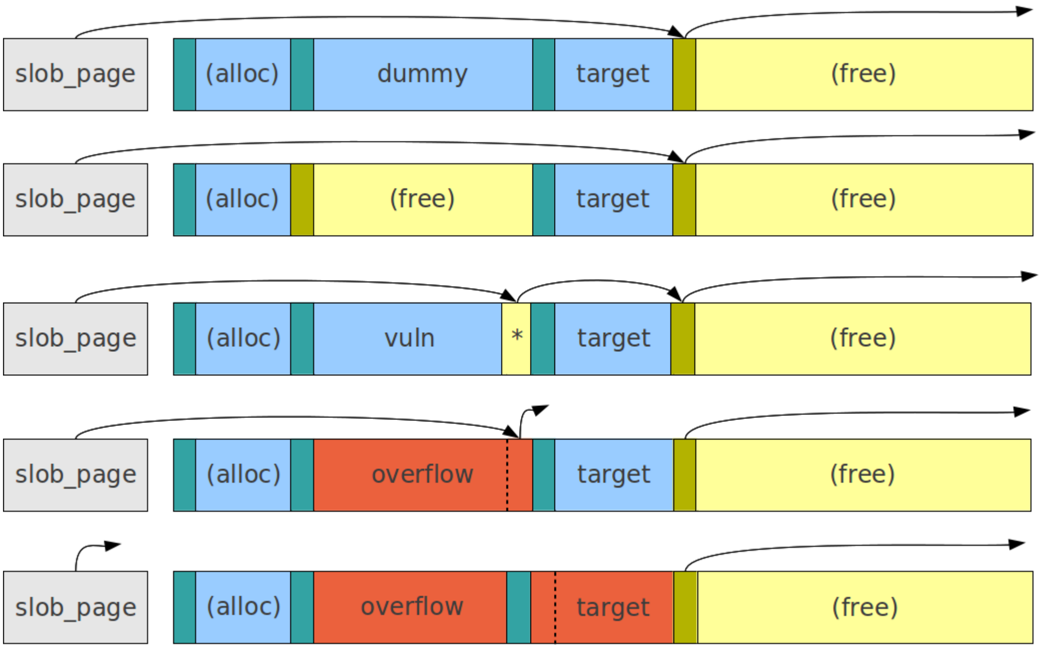

Figure 6: Big endian fragmentation attack.

6. Trigger the allocation of a new object whose contents you can control. This allocation will return a heap block that overlaps with the data stored in the target block.

7. Provide suitable data to be copied into the new block, which resides in the same memory region as the target block.

8. Trigger the invocation of a function pointer in the target block, which now contains attacker-controlled data.

5 Post Exploitation

Several of the previously described techniques result in corrupting the freelist of a SLOB page. Without repairing this corruption or performing some sort of cleanup, the kernel would quickly panic due to unrecoverable errors in the allocator. This risk is amplified when using SLOB as compared with previous allocators, due to the global nature of its caches. Some amount of heap corruption may be tolerable on allocators such as SLUB, where caches contain only objects of narrow ranges of sizes or types, and on SMP systems may be bound to a particular CPU. In contrast, any corruption of a SLOB page will almost certainly affect the stability of the system if uncorrected.

Fortunately, it is relatively straightforward to repair a corrupted SLOB page. If we can force the traversal of the freelist to terminate early, then any subsequent corrupted free blocks will be ignored by the allocator. The code responsible for checking if a free block is the end of the freelist is straightforward:

```
static int slob_last(slob_t *s)
{
        return !((unsigned long)slob_next(s) & ~PAGE_MASK);
}
```

In other words, if the next-free index returns any address that is page-aligned, SLOB assumes the current free block is the last block in the freelist. On systems with 4KB pages, this means that if we overwrite any free block's next-free pointer with any multiple of `0x800` (keeping in mind the index is in `SLOB_UNIT`s, so it will be multipled by two) or `NULL`, the freelist will be terminated early.

In practice, it's sometimes possible to perform this freelist termination by re-triggering the overflow to overwrite a freelist pointer with a termination value. Other times, cleanup must be performed after code execution has been achieved. In the event that a function pointer overwrite was used, the base of the overflowed structure will likely reside in some register, allowing us to easily find the location of the overflowed chunk and manually adjust the freelist. In other cases, it may be necessary to access the global linked list of SLOB pages of the appropriate size, find the first page (due to the page rotation optimization, we know our overflow took place in the first page in the list), and tweak its freelist manually.

When exploiting systems with very limited amounts of memory, wasting an entire page by terminating its freelist abruptly may contribute to OOM (out-of-memory) conditions, negatively affecting the stability of the system. In these cases, it may be necessary to repair damage done by the exploit by manually reconstructing the freelist for the corrupted SLOB page.

6 Future Work

During the discussion of heap exploitation techniques in this paper, we've assumed that we have knowledge of a wide variety of useful allocation primitives that can be used to massage the heap state and provide useful targets. sgrakkyu and twiz have described several useful kernel heap structures that may accomplish these goals, including the `shmid_kernel` and `sctp_ssnmap` structures, but these may not be usable in all situations.

Being able to automatically identify useful kernel heap data structures would make writing heap exploits significantly easier. Some preliminary progress has been made in this direction [8], but more advanced analysis that assesses the usefulness of a structure as a target and the ability of an unprivileged user to trigger a persistent allocation of that structure would be a major improvement.

Finally, this paper's focus on attacks should provide insight for those interested in defending the kernel against heap exploitation. Projects such as Subreption's KERNHEAP [9] provide a strong framework for implementing heap hardening, and future work should continue to refine these defensive techniques.

7 Acknowledgements

Thanks to John Redford, Kees Cook, Lorenzo Hernandez, argp, and Michael Coppola for their helpful comments and suggestions.

References

[1] [PATCH] slob: introduce the SLOB allocator. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=commit;h=10cef6029502915bdb3cf0821d425cf9dc30c817`.

[2] Embedded Gentoo. http://www.gentoo.org/proj/en/base/embedded.

[3] OpenEmbedded. http://www.openembedded.org.

[4] OpenWrt. https://openwrt.org.

[5] [PATCH] make /proc/slabinfo 0400. https://lkml.org/lkml/2011/3/3/306.

[6] twiz and sgrakkyu. Attacking the Core: Kernel Exploitation Notes. http://www.phrack.org/issues.html?issue=64&id=6.

[7] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Syngress, 2010.

[8] kstructhunter. https://github.com/jonoberheide/kstructhunter.

[9] Larry H. Linux Kernel Heap Tampering Detection. http://phrack.org/issues.html?issue=66&id=15.