

# HEAP OVERFLOW



# Khalil Ezhani

**Senator.of.Pirates@gmail.com**

<http://www.facebook.com/SenatorofPirates>

Not all buffers are allocated on the stack. Often an application doesn't know how big to make certain buffers until it is running. The heap is used by applications to dynamically allocate buffers of varying sizes. These buffers are susceptible to overflows if user-supplied data isn't checked, leading to a compromise through an attacker overwriting other values on the heap.

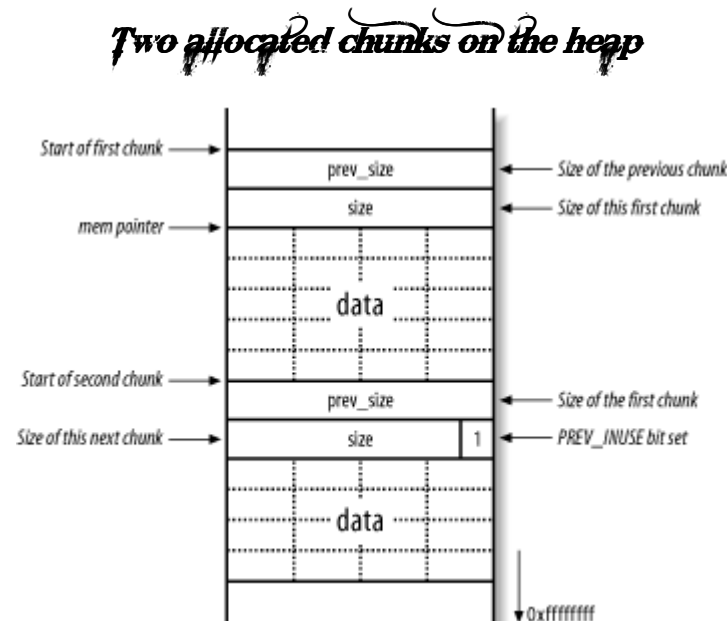
Where the details of stack overflow exploitation rely on the specifics of hardware architecture, heap overflows are reliant on the way certain operating systems and libraries manage heap memory. Here I restrict the discussion of heap overflows to a specific environment: a Linux system running on an Intel x86 platform, using the default GNU libc heap implementation (based on Doug Lea's *dldmalloc*). While this situation is specific, the techniques I discuss apply to other systems, including Solaris and Windows.

Heap overflows can result in compromises of both sensitive data (overwriting filenames and other variables on the heap) and logical program flow (through heap control structure and function pointer modification). I discuss the threat of compromising logical program flow here, along with a conceptual explanation and diagrams.

## 1 - OVERFLOWING THE HEAP TO COMPROMISE PROGRAM FLOW

The heap implementation divides the heap into manageable chunks and tracks which heaps are free and in use. Each chunk contains a header structure and free space (the buffer in which data is placed).

The header structure contains information about the size of the chunk and the size of the preceding chunk (if the preceding chunk is allocated). Figure 13-12 shows the layout of two adjacent allocated chunks.



In Figure 13-12, `mem` is the pointer returned by the `malloc( )` call to allocate the first chunk. The `size` and `prev_size` 4-byte values are used by the heap implementation to keep track of the heap and its layout. Please note that here I have drawn these heap diagrams upside down (when compared with the previous stack diagrams), therefore `0xffffffff` is downward in these figures.

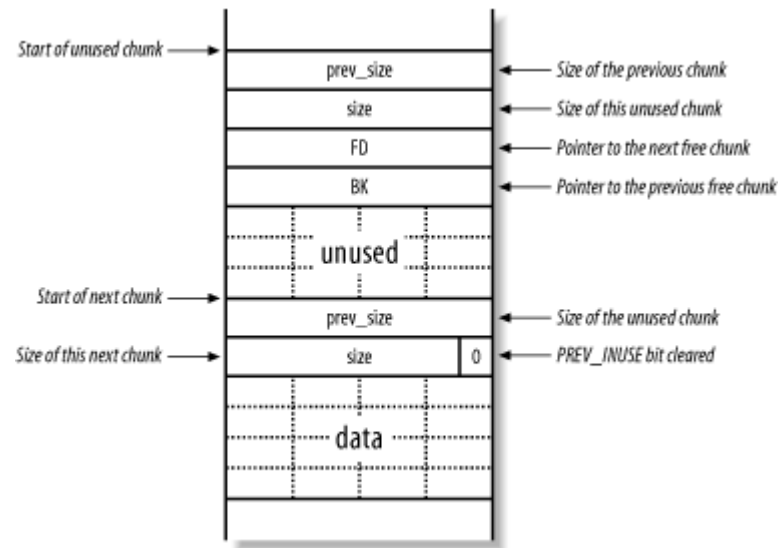
The *size* element does more than just hold the size of the current chunk, it also specifies whether the previous chunk is free or not. If a chunk is allocated, the size element of the next chunk has its least significant bit set, otherwise this bit is cleared. This bit is known as the PREV\_INUSE flag; it specifies whether the previous chunk is in use.

When a program no longer needs a buffer allocated via `malloc( )`, it passes the address of the buffer to the `free( )` function. The chunk is deallocated, making it available for subsequent calls to `malloc( )`. Once a chunk is freed, the following takes place:

- The PREV\_INUSE bit is cleared from the size element of the following chunk, indicating that the current chunk is free for allocation
- The addresses of the previous and next free chunks are placed in the chunk's data section, using `bk` (backward) and `fd` (forward) pointers

Figure 13-13 shows a chunk on the heap that has been freed, including the two new values that point to the next and previous free chunks in a doubly linked list (`bk` and `fd`), which are used by the heap implementation to track the heap and its layout.

## TWO CHUNKS, OF WHICH THE FIRST IS FREE FOR ALLOCATION



When a chunk is deallocated, a number of checks take place. One check looks at the state of adjacent chunks. If adjacent chunks are free, they are all merged into a new, larger chunk. This ensures that the amount of usable memory is as large as possible. If no merging can be done, the next chunk's `PREV_INUSE` bit is cleared, and accounting information is written into the current unused chunk.

Details of free chunks are stored in a doubly linked list. In the list, there is a forward pointer to the next free chunk (`fd`) and a backward pointer to the previous free chunk (`bk`). These pointers are placed in the unused chunk itself. The minimum size of a chunk is always 16 bytes, so there is enough space for the two pointers and two size integers.

The way this heap implementation consolidates two chunks is by adding the sizes of the two chunks together and then removing the second chunk from the doubly linked list of free chunks using the `unlink( )` macro, which is defined like this:

```

#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}

```

This means that in certain circumstances, the memory pointed to by `fd+12` is overwritten with `bk`, and the memory pointed to by `bk+8` is overwritten with the value of `fd` (where `fd` and `bk` are pointers in the chunk). These circumstances include:

- A chunk is freed
- The next chunk appears to be free (the `PREV_INUSE` flag is unset on the next chunk after)

If you can overflow a buffer on the heap, you may be able to overwrite the chunk header of the next chunk on the heap, which allows you to force these conditions to be true, which, in turn, allows you to write four arbitrary bytes anywhere in memory (because you control the `fd` and `bk` pointers). Example 13-7 shows a simple vulnerable program.

### EXAMPLE 13-7. A VULNERABLE HEAP-UTILIZING PROGRAM

```
int main(void)
{
    char *buff1, *buff2;

    buff1 = malloc(40);

    buff2 = malloc(40);

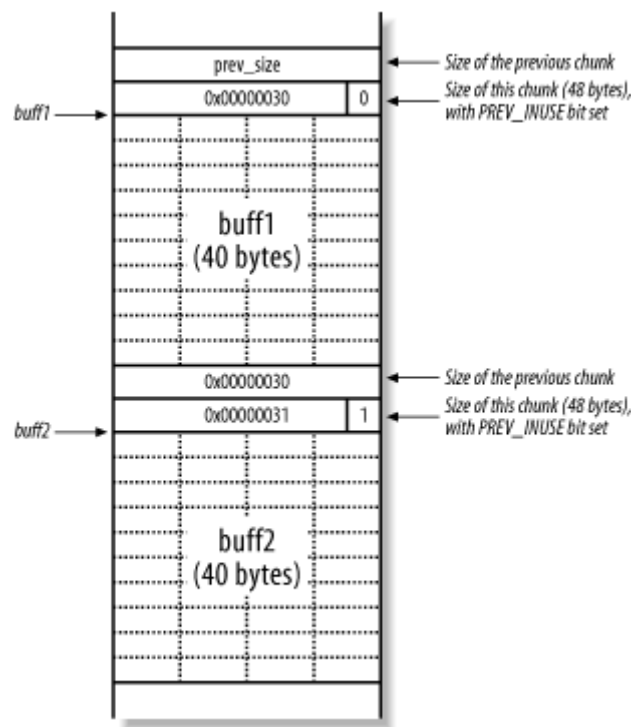
    gets(buff1);

    free(buff1);

    exit(0);
}
```

In this example, two 40-byte buffers (buff1 and buff2) are assigned on the heap. buff1 is used to store user-supplied input from gets( ) and buff1 is deallocated with free( ) before the program exits. There is no checking imposed on the data fed into buff1 by gets( ), so a heap overflow can occur. Figure 13-14 shows the heap when buff1 and buff2 are allocated.

## THE HEAP WHEN BUFF1 AND BUFF2 ARE ALLOCATED



The `PREV_INUSE` bit exists as the least significant byte of the size element. Because size is always a multiple of 8, the 3 least-significant bytes are always 000 and can be used for other purposes. The number 48 converted to hexadecimal is `0x00000030`, but with the `PREV_INUSE` bit set, it becomes `0x00000031` (effectively making the size value 49 bytes).

To pass the `buff2` chunk to `unlink( )` with fake `fd` and `bk` values, you need to overwrite the size element in the `buff2` chunk header so the least significant bit (`PREV_INUSE`) is unset. In all of this, you have a few constraints to adhere to:

- `prev_size` and `size` are added to pointers inside `free( )`, so they must have small absolute values (i.e., be small positive or small negative values)
- `fd` (next free chunk value) + `size` + 4 must point to a value that has its least significant bit cleared (to fool the heap implementation into thinking that the chunk after next is also free)

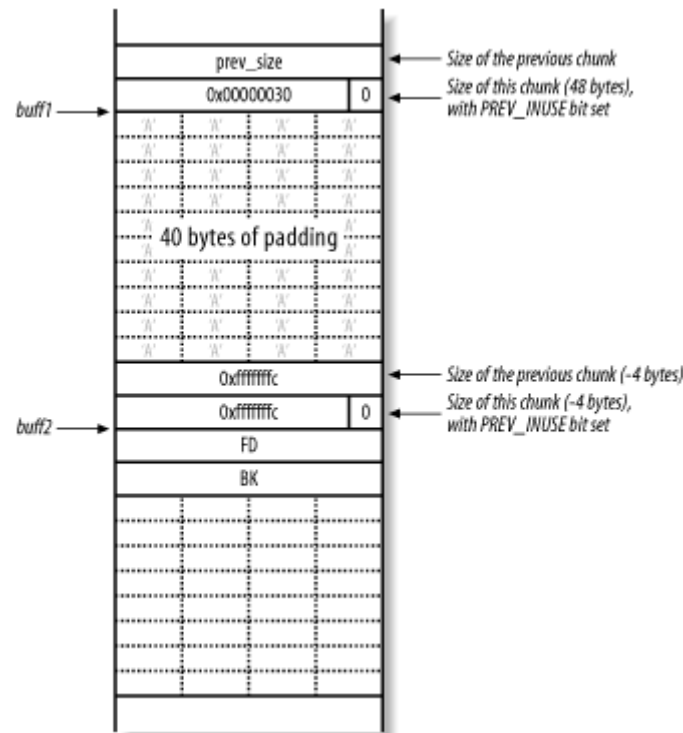


- There must be no NULL (\0) bytes in the overflow string, or gets( ) will stop copying data

Since you aren't allowed any NULL bytes, use small negative values for *prev\_size* and *size*. A sound choice is -4, as this is represented in hexadecimal as 0xffffffffc. Using -4 for the size has the added advantage that  $fd + size + 4 = fd - 4 + 4 = fd$ . This means that free( ) thinks the buff2 chunk is followed by another free chunk, which guarantees that the buff2 chunk will be unlinked.

Figure 13-15 shows the heap layout when you overflow the buff1 buffer and write the two -4 values to overwrite both *prev\_size* and *size* in the header of the buff2 chunk.

### OVERWRITING HEAP CONTROL ELEMENTS IN THE NEXT CHUNK



Because `free( )` deallocates `buff1`, it checks to see if the next forward chunk is free by checking the `PREV_INUSE` flag in the third chunk (not displayed in these diagrams). Because the size element of the second chunk (`buff2`) is `-4`, the heap implementation reads the `PREV_INUSE` flag from the second chunk, believing it is the third. Next, the `unlink( )` macro tries to consolidate the chunks into a new larger chunk, processing the fake `fd` and `bk` pointers.

As `free( )` invokes the `unlink( )` macro to modify the doubly linked list of free chunks, the following occurs:

- `fd+12` is overwritten with `bk`.
- `bk+8` is overwritten with `fd`.

This means that you can overwrite a 4-byte word of your choice, anywhere in memory. You know from smashing the stack that overwriting a saved instruction pointer on the stack can lead to arbitrary code execution, but the stack moves around a lot, and this is difficult to do from the heap. Ideally, you want to overwrite an address that's at a constant location in memory. Luckily, the Linux Executable File Format (ELF) provides several such regions of memory, two of which are:

- The Global Offset Table (GOT); contains the addresses of various functions
- The `.dtors` (destructors) section; contains addresses of functions that perform cleanup when a program exits

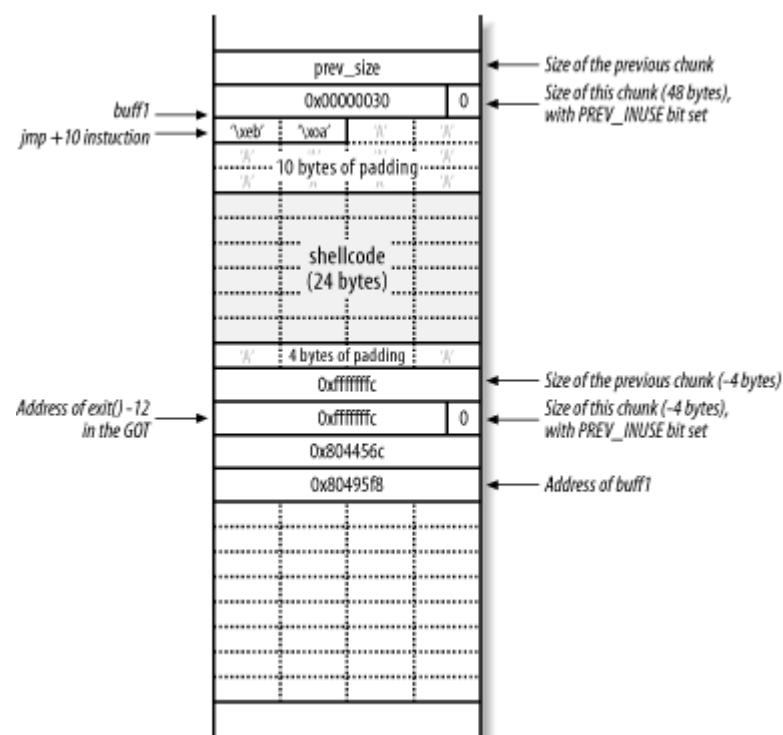
For the purposes of this example, I'll overwrite the address of the `exit( )` function in the GOT. When the program calls `exit( )` at the end of `main( )`, execution jumps to whatever address I overwrite the address of `exit( )` with. If you overwrite the GOT entry for `exit( )` with the address of shellcode you supply, you must remember that the address of `exit( )`'s GOT entry is written 8 bytes into your shellcode, meaning that you need to jump over this word with a `jmp .+10` processor instruction.

You need to set the next chunk variables and pointers to the following:

- `fd` = GOT address of `exit( )` - 12
- `bk` = the shellcode address (`buff1` in this case)

Figure 13-16 shows the desired layout of the heap after the program has called `gets( )` with the crafted `0xffffffff` values for `prev_size`, `size`, `fd`, and `bk` placed into the `buff2` chunk.

## OVERWRITING FD AND BK TO EXECUTE THE SHELLCODE



You effectively overwrite the GOT entry for `exit()` (located at 0x8044578) with the address of `buff1` (0x80495f8), so that the shellcode is executed when `exit()` is called by the program.

## OTHER HEAP CORRUPTION ATTACKS

The heap can be corrupted and logical program flow compromised using a small number of special techniques. Heap off-by-one, off-by-five, and double-free attacks can be used to great effect under certain circumstances. All these attacks are specific to heap implementations in the way they use control structures and doubly linked lists to keep track of free chunks.

## **HEAP OFF-BY-ONE AND OFF-BY-FIVE BUGS**

As with little endian architectures and stack off-by-one bugs, the heap is susceptible to an off-by-one or off-by-five attack, overwriting the PREV\_INUSE least significant bit of prev\_size (with an off-by-one) or size (with an off-by-five). By fooling free( ) into consolidating chunks that it shouldn't, a fake chunk can be constructed, which results in the same attack occurring (by setting arbitrary fd and bk values).

## **DOUBLE-FREE BUGS**

The fd and bk values can also be overwritten using a *double-free* attack. This attack doesn't involve an overflow; rather the heap implementation is confused into placing a freed chunk onto its doubly linked list, while still allowing it to be written to by an attacker.

**SO THANKS GUYS FOR READING MY PAPER AND I WOULD LIKE TO THANK TO FRIENDS :  
WLAD BLOC – TIFLET – MOROCCO**