R09921A10 鄭翔予
CTF Account: stanleymusic

1. sandbox
   - Description
     - The challenge name literally explains what this program does. It'll take inputs from the client and interpret it as assembly code and execute it
   - Solution
     - There're some limitations on the inputs
       - Its length needs to be less than or equal to 0x200
       - It forbids several instructions
         - syscall
         - call <reg>
     - Our goal is to get the shell, so I decide to inject shellcode for "execve('/bin/sh')"
     - Although "syscall" is not allowed, an x64 architecture is still compatible with x86 instructions such as "int 0x80"
       - While using x86 instruction, the calling convention needs to follow x86 architecture instead of x64 architecture
     - Arguments for execve syscall
       - %eax = 0xb
       - %ebx = const char *filename
         - In this case, it's the address of the string "/bin/sh"
       - %ecx = const char *const *argv
         - In this case, Null(0) is enough
       - %edx = const char *const *envp
         - In this case, Null(0) is enough
     - In x64 architecture, use $rax instead of %eax if the value stored in the register is larger than 32 bit, e.g., address. Calling "int 0x80" will clear out the higher 32-bit value of registers to zero, but in this case, it doesn't matter
     - After sending the shellcode, it'll open a new shell
   - Flag
     - FLAG{It_is_a_bad_sandbox}
   - Reference
     - http://shell-storm.org/shellcode/files/shellcode-827.php
     - http://shell-storm.org/shellcode/files/shellcode-603.php
     - https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

2. Fullchain
   - Thanks to R10922056 for discussion about the possibilities of hijacking memset as mprotect and also the thoughts of modifying the value of cnt
   - Description
     - The program gives three chances to do three actions, set, read and write, on two kinds of variables, respectively local and global
   - Solution
     - There's a format string vulnerability in the mywrite function, which takes we can control the parameter of printf
       - The 7th parameter of printf is the address of input to printf
       - The 14th parameter contains the start of the content of the variable local
       - The 8th parameter is the address of __libc_csu_init
     - There's overflow in myread function, the size of local and global is 0x10 while the parameter of scanf is %24s, which means we can overflow 24-0x10 = 8 bytes
       - Combine with format string vulnerability, 14th, 15th, 16th parameters of printf contains the value of local[0:8], local[8:16], local[16:24] respectively
       - We can now read/write anything anywhere as long as it's readable/writable
     - Seccomp rule includes
       - exit_group, exit, close, open, read, write, mprotect, brk, mmap
       - No execve
     - The first step is to modify the value of cnt, which is a local variable to chal function same as local, because three chances aren't enough
       - local is at [rbp-0x28] while cnt is at [rbp-0x34], the offset is 0x34-0x28=0xc
       - Get the address of local by fmt with 7th parameter, and thus get the address of cnt by local_addr – 0xc
       - Write cnt_addr to the stack at 16th parameter of printf by myread
         - local[0:16] = ARBITRARY-VALUE
         - local[16:24] = p64(cnt_addr)
       - Write value to cnt_addr by fmt using %n
         - Currently, the only value we can write to cnt_addr is 5 due to payload = "write%16$n"
     - Now the cnt is set to 5, which is still far too small for the entire exploit. However, we can spend another 3 chances to modify the cnt value

again to an arbitrary value

- Write cnt_addr to the stack
- Setup the payload string in variable global
  - global = "%[VALUE-OF-CNT]c%16$hhn"
- By calling mywrite(global), we can write [VALUE-OF-CNT] to the 16th parameter which is now set to cnt_addr, and thus change the value of cnt

- Calculate libc base address
  - We can get the value of __libc_csu_init address by fmt with 8th parameter, and thus get the value of printf@got.plt address
    - Offset of __libc_csu_init address to binary = 0x1800
    - Offset of printf@got.plt address to binary = 0x4048
    - printf@got.plt_addr = __libc_csu_init_addr + 0x2848
  - Since we want to read the value at printf@got.plt_addr, we can use the same technique as modifying the value of cnt, while this time we use %s to read the value
    - Libc base address = (value at printf@got.plt address) – (printf offset to libc)
- Hijack memset to mprotect, and exit to write
  - Offset of memset@got.plt address to binary = 0x4058
  - Offset of exit@got.plt address to binary = 0x4070
  - We can calculate the address of mprotect and write by libc base address and use the same technique to write this value to the GOT
  - memset(addr, c, len) -> mprotect(addr, len, prot)
    - addr -> addr: the start of the region to be changed
    - c -> len: the size of the region to be changed
    - len -> prot: the desired protection
    - I'm going to write shellcode to address near variable global, so I want to change the protection of page, that contains global, to rwx(7)
  - exit(1) -> write(1)
    - After overwriting it to write, the program will never stop even though strncmp is not match
    - In this way, I can modify the value of ptr. Otherwise, the only two possible values of ptr are the address of global and local, which is not a valid address to mprotect since it needs the address to be the page boundary

        - Modify the value of ptr at [rbp-0x30] to page boundary that includes global
            - Get address of ptr by cnt_addr + 0x4
            - Address of page boundary = global_addr – 0xb0
            - Use the same technique to modify the last two bytes of ptr
            - After modifying the value of ptr, to avoid polluting it, the next loop when the program asking for global or local, we need to respond with a value other than these two
            - Then we can change the protection near global_addr to rwx
        - Write open-read-write shellcode to address near stack using the same technique
        - GOT hijack value of exit to the address our shellcode, so calling exit(1) turns out to be calling our shellcode
    - Flag
        - FLAG{Emperor_time}
3. fullchain-nerf
    - Description
        - Similar to fullchain while this time no suitable function to hijacked as mprotect
        - Bytes to ve overflowed are larger
    - Solution
        - The first step is the same, we need to modify the value of cnt
        - The second step is also the same, we need to calculate the libc base address
        - The next step is slightly different. In fullchain, we construct the entire shellcode and write it to address near the global, while in fullchain-nerf, I'm going to use an open-read-write ROP chain constructed from both libc gadgets and binary gadgets and write it to the return address of chal
            - Offset to return address from local is 56
            - Use the same technique to write ROP to the position which contains the return address
            - write string /home/fullchain-nerf/flag to global and use the address of global as parameter to open and read
        - After the chal function returns, it'll start executing our ROP chain
            - To accelerate the process, we can set cnt to 1 for the last step writing path to global. And then the loop will be finished
    - Flag

        - FLAG{fullchain_so_e4sy}
4. Final
    - Description
        - A classic menu challenge with function buy/release/change and play
    - Solution
        - There're dangling pointers
        - The size of the name of each animal is controllable
        - The play option will show the type and the name of the object
        - Use unsorted bin attack to leak libc base address
            - Note that we need an extra in-used chunk to avoid top chunk consolidation
            - The value of unsorted bin's bk/fd is the address of main_arena which is in the libc
            - The buy and change option uses the read function on name, which won't add \x00 at the end and thus we can control the value and length of the name to leak the value of fd/bk by UAF
        - Use the similar technique to leak tcache's fd to calculate heap base address
        - By UAF and heap overflow, we can control the function pointer and the parameter of an object
            - bark -> system
            - type -> b'/bin/sh\x00' + b'A'*0x8(padding)
            - Therefore the play option now becomes system('/bin/sh\x00AAAAAAAA', name) and thus get shell
    - Flag
        - FLAG{do_u_like_heap?}
5. easyheap
    - Description
        - A classic menu challenge with function add/delete/edit/list and find
    - Solution
        - There're dangling pointers, which can be leveraged to UAF and double free
        - The maximum amount of books is 0x10
        - book size is 0x20, and the chunk size is 0x30, so after freeing, the chunk will be either in tcache bin or fast bin

```
pwndbg> x/10gx 0x5650fefcb290
|   |   |   |   | meta data      |      | chunk size     |
0x5650fefcb290: 0x0000000000000000      0x0000000000000031
|   |   |   |   | name address   |      | index = namelen|
0x5650fefcb2a0: 0x00005650fefcb2d0      0x0000000000000020
|   |   |   |   | price          |      | namelen (never assigned)|
0x5650fefcb2b0: 0x000000000000000a      0x0000000000000000
```

- name size is controllable
- Leak heap base address
  - list option will show all the books' index, name and price
  - The position of the index will store the value of tcache bin's key if the chunk is freed
  - By utilizing UAF we can leak the value of tcache bin's key and calculate the heap base address
    - Heap base address = key – 0x10
- Leak libc base address
  - Utilize fast bin double free and UAF to read/write anything anywhere
  - The architecture of freed chunks

```
fastbin[0x30] -> book4 -> book5 -> book4(double free)
tcache[0x30] -> book3 -> book2 -> book2_name -> book1 -> book1_name -> book0 -> book0_name
tcache[0x40] -> book3_name
tcache[0x50] -> book4_name (Although we freed book4 twice, the second time book4_name is invalid address)
unsortedbin  -> book5_name
```

    - The first three books have the same size of name's chunk = 0x30
    - The rest three books have different sizes of name's chunk to make the double free easier to understand
    - The chunk of the book5's name is in the unsorted bin
  - After double free, the fd of chunk 4 is also the name pointer of chunk 4 and it points to chunk5. Therefore, by UAF, we can control 0x20 bytes of chunk 5 which includes the name pointer of chunk 5
  - Therefore if we overwrite the name pointer of chunk 5 to points to the fd/bk of chunk 5, we can get the address of main_arena which is in libc, and thus calculate the libc base address
  - Moreover, since the address of chunk 5's name pointer is controllable by editing chunk 4's name value, we can also write any value to any address
- write system address to __free_hook using the same technique for leaking libc base address
  - The address of the system is calculated by libc base address +

offset of the system

- write some chunk's name as "/bin/sh" and delete it to invoke __free_hook(which is now system)
  - To avoid corruption, I choose a size that is never used before

- Flag
  - FLAG{to_easy...}

6. Beeftalk
   - Description
     - A program that provides several functions including chatroom, user signup, user delete, user profile update
   - Solution
     - There're dangling pointers that can be leveraged to UAF
       - As long as we memorize the token, we can edit with freed chunks
     - Sizes of chunks created by init_user
       - User: 0x50
       - name: 0x30
         - It can later be reallocated if the length of the name is larger than 0x20
         - The largest length = 0x100
       - desc: 0x50
       - job: 0x20
       - fifo0/fifo1: 0x30
     - Leak heap base address
       - The address that the name pointer points to will be the position of fd of a tcache
       - While signup, the safe_read function is used for reading our input as the value of name which won't append '\x00' at the end, so we can leak the value of fd if this chunk is previously freed
       - The least significant byte will be overwritten by our new name, but it's still possible to calculate the heap base address because the least significant byte will be eliminated by offset
     - Leak libc base address
       - To leak libc base address, we need the chunks to be in a small bin or an unsorted bin
       - I sign up 8 users with the length of name = 0x100 and free them all
         - Since tcache only contains 7 chunks in each entry and the result of consolidation, the last freed user will lead to

multiple chunks in small bins and unsorted bins

- ✧ It turns out that one of the small bin chunks will contain bk in the position that'll be in the area of name if the chunk is allocated again. This chunk is the fourth user
  - ➢ Due to safe_read function, we can leak the value of bk and thus calculate the libc base address

- ■ So far, there are four user's chunks in freed chunk and four user's chunks in-used
  - ✧ Manipulating with the first free chunk and the second free chunk can lead to arbitrary write, which can help us write the address of system to __free_hook

```
free chunk 1
0x5604660ddb90: 0x00005604660dddb0      0x00007fb411ccbc70
0x5604660ddba0: 0x00005604660ddc30

free chunk 2
0x5604660dddc0: 0x00005604660ddfe0      0x00005604660ddb80
0x5604660dddd0: 0x00005604660dde60
```

  - ✧ From the figure, we note that the pointer of name of free chunk 1 points to the address that is 0x10 bytes away from the value of pointer of name of the free chunk 2
  - ✧ By heap overflow, we can control the value of the pointer of name of the chunk2 and combine it with the update option provided by the program. We can edit the name at the address we just overwritten
    - ➢ Overwrite the value of the pointer of name of the free chunk2 to address of __free_hook
    - ➢ Using update option to edit the name of free chunk 2 to the address of system
  - ✧ To avoid corruption, the rest should remain with the same value
- ■ Create one more chunk with name = "/bin/sh" and free it
  - ◆ This will lead to free("/bin/sh") and since we've overwritten __free_hook with system, it'll become system("/bin/sh")

- ● Flag
  - ■ FLAG{beeeeeeOwO}

7. FILE note
   - ● Description
     - ■ A program that provides several functions including create_note,

write_note, and save_note

- Solution
  - create_note will help us construct a valid file structure in memory, but it can only be called once
  - write_note uses gets function which leads to overflow. We can overflow the previously mentioned file structure and construct our own fake _IO_FILE structure
  - save_noet uses the fwrite function which can be leveraged to arbitrary read/write
  - Leak libc base address
    - Since there's no output, we need to first utilize fwrite to write things to stdout
    - Modify the _fileno to 1 represents stdout
    - fwrite will write everything between _IO_write_base and _IO_write_ptr
      - Due to overwrite we can control the value of these two variables but we don't know any valid address and don't know where is our target yet
      - Instead of specifying the value of _IO_write_base, we can overwrite the least-significant byte of original _IO_write_base to '\x00', so we can still leak some values out
        - Fortunately, there is one address that lies in libc
      - Another issue is about the value of _flags. To avoid new_do_write executes _IO_SYSSEEK, we need to set _IO_IS_APPENDING to 1 and also we need to set _IO_CURRENTLY_PUTTING to 1 too to bypass other uncontrollable conditions
  - Hijack vtable function pointer address
    - All the addresses lie in libc
    - fwrite can also lead to arbitrary write
      - The address between the value of _IO_write_ptr and _IO_write_end is where the destination of fwrite
      - Again, using overflow to control these two variables to our target address
    - Overwrite the vtable function pointer address with one_gadget
      - There are many functions to choose as the target, but we need to select those that can be triggered after overwriting

and also meet the conditions of one_gadget

- ✧ After several attempts, I figure out that the _IO_defaul_uflow works

- Flag
  - FLAG{f1l3n073_15_b3773r_7h4n_h34pn073}
- Reference
  - https://xz.aliyun.com/t/5853