Challenge:-

Vuln Storage:-

Yo Hackers!!I created this new storage for storing data!!
I am pretty sure that there are some security bugs lying down in this software!!
Show your might by breaking this!!
"Rick Roll and Hack the planet"

Author:- Itz_me_strikerr

TL-DR:-

Leveraging a single null byte overflow primitive to create overlapping chunks which will allow us to tamper inline metadata and turn that to a fastbin dup primitive to yield a shell.

Writeup:-

We are given a docker file, shared libraries and the challenge binary.

Let’s do some basic enumeration on the challenge binary.

```
ajay@kali:~/tamil_ctf_2.0/heap_chall$ ./cute_little_vulnerable_storage
================================================================================
Please don't pwn me.. My creator did not made me up with best code practises
================================================================================
Storage space created 0/15

1.Create storage space

2.Destroy the storage space

3.Edit the storage space

4.View the storage space

5.Exit

>>
```

Looks like a normal heap note challenge.

With 5 options, Seems like first option is used to create storage space(malloc) and the second option is used to destroy storage space (free) and the third option is used to edit the storage space (read and put the contents to the allocated region) and the fourth option is used to view the storage space(read from the allocated space and write to output) will be useful to get memory leaks and the last option seems to simply calls exit.

Rather than some assumptions let's view this binary in ghidra to get a decompiled view for a better understanding..

```
void main(void)

{
  ssize_t sVar1;
  int local_144;
  int local_140;
  int local_13c;
  void *pvStack312;
  int aiStack304 [58];
  char local_48 [36];
  undefined4 local_24;
  void *local_20;
  ulong local_18;
  int local_10;
  uint local_c;

  puts("================================================================================");
  puts("Please don\'t pwn me.. My creator did not made me up with best code practises");
  puts("================================================================================");
  local_c = 0;
  do {
    printf("Storage space created %d/15\n\n",(ulong)local_c);
    puts(
        "1.Create storage space\n\n2.Destroy the storage space\n\n3.Edit the storage space\n\n4.View
         the storage space\n\n5.Exit\n"
        );
    printf(">>");
    __isoc99_scanf(&DAT_0010213c);
    switch(local_24) {
    default:
      puts("\nInvalid Index\n");
      break;
    case 1:
      if (((int)local_c < 0xf) && (-1 < (int)local_c)) {
        write(1,"\nEnter the size:",0x10);
        read(0,local_48,0x1f);
        local_18 = strtoul(local_48,(char **)0x0,0);
        if ((local_18 == 0x227ffb) || (((0x17 < local_18 && (local_18 < 0x3e9)) && (local_18 != 0)))
           ) {
          local_20 = malloc(local_18);
          if (local_20 == (void *)0x0) {
            puts("Chunk request failed");
          }
          else {
            (&pvStack312)[(long)(int)local_c * 2] = local_20;
            aiStack304[(long)(int)local_c * 4] = (int)local_18;
            memset(local_20,0,0x10);
            local_c = local_c + 1;
          }
        }
        else {
          puts("Request size should be between 24 and 1000");
        }
      }
      else {
        puts("Only 15 requests available");
```

```
      puts("Only 15 requests available");
    }
    break;
  case 2:
    write(1,"\nEnter the index:",0x11);
    __isoc99_scanf(&DAT_0010213c);
    if ((local_13c < 0) || (0xe < local_13c)) {
      puts("\nTrying out of bounds huh!!Don\'t make me laugh!! Try better!!\n");
    }
    else {
      if (((&pvStack312)[(long)local_13c * 2] == (void *)0x0) || ((int)local_c <= local_13c)) {
        puts("Trying UAF or Double free is a joke here\n");
      }
      else {
        free((&pvStack312)[(long)local_13c * 2]);
        (&pvStack312)[(long)local_13c * 2] = (void *)0x0;
        puts("\nFree\n");
      }
    }
    break;
  case 3:
    write(1,"\nEnter the index:",0x11);
    __isoc99_scanf(&DAT_0010213c);
    if ((local_140 < 0) || (0xe < local_140)) {
      puts("But we cannot give you any integer bugs to exploit");
    }
    else {
      if (((&pvStack312)[(long)local_140 * 2] == (void *)0x0) || ((int)local_c <= local_140)) {
        puts("What are you even trying uaf or double free!!Nothing works here");
      }
      else {
        write(1,"\nEnter the data:",0x10);
        sVar1 = read(0,(&pvStack312)[(long)local_140 * 2],(long)aiStack304[(long)local_140 * 4]);
        local_10 = (int)sVar1;
        memset((void *)((long)(int)sVar1 + (long)(&pvStack312)[(long)local_140 * 2]),0,1);
        puts("Data successfully written on chunk");
      }
    }
    break;
  case 4:
    write(1,"\nEnter the index:",0x11);
    __isoc99_scanf(&DAT_0010213c);
    if ((local_144 < 0) || (0xe < local_144)) {
      puts("Sorry no out of bound read using integer bugs");
    }
    else {
      if (((&pvStack312)[(long)local_144 * 2] == (void *)0x0) || ((int)local_c <= local_144)) {
        puts("Guys!! Don\'t be so stupid to try read after after here!!");
      }
      else {
        write(1,"\nHere is your chunk contents\n",0x1c);
        write(1,(&pvStack312)[(long)local_144 * 2],(long)aiStack304[(long)local_144 * 4]);
      }
    }
    break;
  case 5:
    puts("\nExit\n");
```

```
  case 5:
    puts("\nExit\n");
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
} while( true );
}
```

I know it’s a lot to absorb from here..

So let me tell you what every option does here:-

1)Create storage space

It does size checks and see whether the user tries to pass some negative request size and the request size lies between 0x17 to 0x3e9.

And from the decompiled view there is something wrong..Slightly ghidra messed up this one.

if((size == system+0x121f63) || (size > 0x17) && (size < 0x3e9) && (size > 0))

This is given to reduce the burden of creating an unsorted bin attack near __free_hook to create a fake size field which will become more complex .So as to ease this it will also allocate if your request size matches with the */bin/sh memory location*

And checks how may times the chunk are created.. The maximum is 15 chunks.

And after malloc checks the return type to see whether malloc function worked well..

2)Delete storage space

Asks the index for the chunk and does some checks to prevent uaf or double free scenarios

If legitimate chunk, it frees it.

3)Edit storage space

Asks the index and does some important checks to determine whether it is a legitimate chunk or freed chunk to prevent write after free scenarios

4)View the storage space

Asks the index and after some checks to determine whether it is a legitimate chunk it shows the memory contents ,if already freed it will not work to prevent read after free scenarios.

5)Exit

It simply calls the exit.

Bug:-

On edit the storage function, our binary seems to place a null byte as a string terminator. The bug occurs when we give multiples of 8 but not multiples of 16 which allows us to write the data until the end of the chunk and finally it will place a null byte after the chunk which will be at the next chunk size field.

Let’s have a look in visual manner

```
0x55555555a820  0x0000000000000000      0x0000000000000021      ........!.......
0x55555555a830  0x0000000000000000      0x0000000000000000      ................
0x55555555a840  0x0000000000000000      0x0000000000000021      ........!.......
0x55555555a850  0x0000000000000000      0x0000000000000000      ................
0x55555555a860  0x0000000000000000      0x00000000000207a1      ................        <-- Top chunk
pwndbg>
```

Let’s allocate a couple of 32 bytes chunk.

```
pwndbg> x/10xg 0x55555555a820
0x55555555a820: 0x0000000000000000      0x0000000000000021
0x55555555a830: 0x4141414141414141      0x4141414141414141
0x55555555a840: 0x0a41414141414141      0x0000000000000000
0x55555555a850: 0x0000000000000000      0x0000000000000000
0x55555555a860: 0x0000000000000000      0x00000000000207a1
pwndbg>
```

After allocating some values in 1st chunk we can see that the A’s with the newline plus a final null byte has null’ed the 2nd chunk size field.

Now as we got the bug . We can think about the way we can use this bug..

The main primitive we can obtain from this bug is use after free (both read and write) to create overlapping chunks which we will use it to get leaks and finally a fastbin dup primitive to write __malloc_hook with the address of system function and call it with the argument of /*bin/*sh memory address.

And a main point to cover is the challenge runs in glibc 2.25 version.. So we have no mitigations present that will be obstacle to create our overlapping chunks.

We can exploit this bug in two ways:-

1)House of Einherjar

2)Poison Null Byte

Exploit plan for House of Einherjar:-

1)Allocate 7 chunks of size in order 0x98,0x68,0x88,0x68,0x98,0xf8 and 0x28.

2)Edit the 5th chunk and create a fake prev size field.

3)Free 1st chunk to trigger backward consolidation

4)Free chunk 3 6 2 and finally 4 to get both libc leak and heap leak.

5)Allocate a huge 0x3a8 chunk to get the overlapping chunk that will overlap across all the free chunks.

6)Read the required leaks now to get libc base address and heap address

7)After getting the required leaks ,Use the write after free primitive from the overlapping chunks to edit inline metadata of already freed 2nd chunk to perform a fastbin dup primitive and three more malloc's needed to allocate near the __malloc_hook where their is a viable size field of 0x7f and edit the lastly allocated chunk to overwrite the __malloc_hook with system function address and then call malloc with the location of /*bin/*sh to yield a shell.

Exploit plan for poison null byte:-

1)Allocate 4 chunks of sizes in order 0x68,0x208,0x98 and 0x28.

2)Free the second chunk and edit the first chunk fully which will place a null byte in the 2nd chunk size field

3)Then allocate 2 more chunks 0xf8 and 0x68.

4)Then free the 5th,1st and 3rd chunk to trigger the backward consolidation..

5)Free the 6th chunk and allocate a chunk of size 0x1a8 to overlap the fastbin and the smallbin.

6)And finally using this overlapping chunks to read the leaks to get the base address of libc and heap.

7)Edit the 7th chunk aka the overlapping chunk to setup a fastbin dup primitive and some couple of malloc's to allocate the chunk near the __malloc_hook using the 0x7f size field and finally edit the lastly allocated chunk to overwrite __malloc_hook with system function address with address of /*bin/*sh as size argument which will finally yield us a shell.

You can find the exploit for this binary in:-
[Will be uploaded in github and link will posted here during publish time]