1. 天翼云报错现场

vpp 服务起来后，启动 ctcc_monitor 时触发断言报错，由于共享数据访问为 0 导致的。

```
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# /usr/bin/ctcc_monitor
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/conform-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/exceed-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/violate-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/conform-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/exceed-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/violate-action/dscp".
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldn't connect to server
curl: (7) Couldncurl: (7) Couldn't connec'tt  tcoo nsneercvte rt
o server
curl: (7) Couldn't connect to server
/root/bangyn/tgw.openEuler.0802/vpp/src/vlibmemory/memory_client.c:282 (vl_client_send_disconnect) assertion `shmem_hdr && shmem_hdr-
>vl_input_queue' fails
vl_msg_api_alloc_internal:74: vl_rings NULL
/root/bangyn/tgw.openEuler.0802/vpp/src/vlibmemory/memory_shared.c:75 (vl_msg_api_alloc_internal) assertion `0' fails
Aborted (core dumped)
```

```c
  if (shmem_hdr == 0)
    {
      clib_warning ("shared memory header NULL");
      return 0;
    }

  /* account for the msgbuf_t header */
  nbytes += sizeof (msgbuf_t);

  if (shmem_hdr->vl_rings == 0)
    {
      clib_warning ("vl_rings NULL");
      ASSERT (0);
      abort ();
    }

  if (shmem_hdr->client_rings == 0)
    {
      clib_warning ("client_rings NULL");
      ASSERT (0);
      abort ();
    }

  ap = pool ? shmem_hdr->vl_rings : shmem_hdr->client_rings;
  for (i = 0; i < vec_len (ap); i++)
    {
      /* Too big? */
      if (nbytes > ap[i].size)
```

天翼云排查发现，由于共享地址上的数据被非法修改，导致进程之间无法通过共享内存通信。通过 gdb watch 该共享地址，发现运行起来后 gdb 会报错，打印发现共享地址的值已经被修改了，但 gdb watch 没有停下来。

2. 问题排查过程

2.1. 解决 gdb 在多进程情况下 watch 报错

2.1.1.　gdb 报错现象

gdb 里 watch 共享地址后，报错退出：

```
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# gdb /usr/bin/ctcc_monitor
GNU gdb (GDB) 9.1 sw1.0.1
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-unknown-linux-gnu"...
Reading symbols from /usr/bin/ctcc_monitor...
(gdb) r
Starting program: /usr/bin/ctcc_monitor
^C
Program received signal SIGINT, Interrupt.
0x000040000079c818 in clock_nanosleep () from /usr/lib/libc.so.6
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb) c
Continuing.
[Detaching after vfork from child process 316204]
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/conform-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/exceed-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers/policer/violate-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/conform-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/exceed-action/dscp".
libyang[1]: Schema node "meter-action-mark-dscp" not found (../meter-action-type = meter-action-mark-dscp) with context node "/police
r:policers-state/policer/violate-action/dscp".
[New LWP 316205]
Aborted (core dumped)
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# ls
```

2.1.2. 排查过程

2.1.2.1. 定位 gdb 报错函数

借助 coredumpctl 命令操作系统下的 coredump 文件

coredumpctl list:　　　　　显示所有的 coredump 信息

coredumpctl debug [pid]:　　启动 debugger 调试对应的 coredump，默认用 gdb。

```
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# coredumpctl list
TIME                            PID UID GID SIG     COREFILE  EXE              SIZE
Mon 2024-08-19 10:37:24 CST 315822   0   0 SIGABRT present   /usr/libexec/gdb 12.0M
Mon 2024-08-19 10:40:09 CST 315609   0   0 SIGABRT truncated /usr/bin/vpp     32.8M
Mon 2024-08-19 10:41:53 CST 316152   0   0 SIGABRT present   /usr/libexec/gdb 12.0M
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# date
Mon Aug 19 10:43:45 AM CST 2024
[root@bqj-az1-sdwan-tgw-server02-10e2e7e103 ~]# coredumpctl debug 316152
           PID: 316152 (gdb)
           UID: 0 (root)
           GID: 0 (root)
        Signal: 6 (ABRT)
     Timestamp: Mon 2024-08-19 10:41:52 CST (2min 19s ago)
  Command Line: gdb /usr/bin/ctcc_monitor
    Executable: /usr/libexec/gdb
 Control Group: /user.slice/user-0.slice/session-c209.scope
          Unit: session-c209.scope
         Slice: user-0.slice
       Session: c209
     Owner UID: 0 (root)
       Boot ID: 45a244c889054121855fcb88f79b5402
    Machine ID: 1ab4351192ff403b96651f9c18562a49
      Hostname: bqj-az1-sdwan-tgw-server02-10e2e7e103
       Storage: /var/lib/systemd/coredump/core.gdb.0.45a244c889054121855fcb88f79b5402.316152.1724035312000000.lz4 (present)
     Disk Size: 12.0M
       Message: Process 316152 (gdb) of user 0 dumped core.

GNU gdb (GDB) 9.1 sw1.0.1
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-unknown-linux-gnu"...
Reading symbols from /usr/libexec/gdb...
(No debugging symbols found in /usr/libexec/gdb)
```

打印 gdb coredump 时的调用栈发现出错点在 remove_watchpoint：

```
warning: Unexpected size of section `.reg2/316152' in core file.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Core was generated by `gdb /usr/bin/ctcc_monitor'.
Program terminated with signal SIGABRT, Aborted.

warning: Unexpected size of section `.reg2/316152' in core file.
#0  0x000041543cff6768 in ?? () from /usr/lib/libc.so.6
(gdb) bt
#0  0x000041543cff6768 in ?? () from /usr/lib/libc.so.6
#1  0x000041543cf9ce6c in raise () from /usr/lib/libc.so.6
#2  0x000041543cf82760 in abort () from /usr/lib/libc.so.6
#3  0x000041e05df89694 in handle_sigsegv(int) ()
#4  <signal handler called>
#5  0x000041e05e3c28c4 in sw_64_linux_nat_target::remove_watchpoint(unsigned long, int, target_hw_bp_type, expression*) ()
#6  0x000041e05dcb6dc8 in remove_watchpoint(bp_location*, remove_bp_reason) ()
#7  0x000041e05dc9f180 in remove_breakpoint_1(bp_location*, remove_bp_reason) ()
#8  0x000041e05dc9ebf4 in detach_breakpoints(ptid_t) ()
#9  0x000041e05e085344 in handle_inferior_event(execution_control_state*) ()
#10 0x000041e05e080ea8 in fetch_inferior_event(void*) ()
#11 0x000041e05e052274 in inferior_event_handler(inferior_event_type, void*) ()
#12 0x000041e05e0f0dd0 in handle_target_event(int, void*) ()
#13 0x000041e05df84e04 in handle_file_event(file_handler*, int) ()
#14 0x000041e05df8542c in gdb_wait_for_event(int) ()
#15 0x000041e05df83b50 in gdb_do_one_event() ()
#16 0x000041e05df83c78 in start_event_loop() ()
#17 0x000041e05e133d84 in captured_command_loop() ()
#18 0x000041e05e136680 in captured_main(void*) ()
#19 0x000041e05e136794 in gdb_main(captured_main_args*) ()
#20 0x000041e05dbd8508 in main ()
(gdb)
```

2.1.2.2. 分析 gdb 源码，追查流程

gdb 在创建子进程的时候需要删除从父进程 copy 来的 watchpoint 信息，总体流程如下：

（1）父进程调用 add_initial_lwp 初始化 lp（保存着进程相关信息）

（2）父进程插入 watchpoint，调用 sw_64_linux_nat_target::insert_watchpoint

（3）Fork 子进程后需要 remove copy 来的父进程 watchpoint。例如 x86_debug_reg_state 或者 aarch64_debug_reg_state，该结构存储了进程断点的详细信息，fork 时会复制到子进程，因此子进程需要先 remove 该信息。申威架构将观察点信息存储到 arch_lwp_info 结构体，该结构体与进程 pid 强绑定关系。申威架构建立子进程时未复制父进程的该结构。Mips 架构未使用该结构，而是使用了一个断点链表 current_watches 来删除指定的地址。

（4）子进程调用 add_initial_lwp，初始化子进程 lp。

2.1.2.3. 原因和解决方法

常规的流程为 fork 子进程后，立马删掉子进程 watchpoint。对于申威架构来说此时的子进程还没调用 add_initial_lwp 初始化，导致为 NULL，非法访问，删除失败。

由于申威架构建立子进程时并未复制父进程的该结构，所以这里无需删除子进程 watchpoint，这种情况直接返回 0 表示成功即可。（后期 gdb 可能需要优化的地方：参考 x86 和 arm 架构，将观察点信息保存到一个单独的结构体链表中，避免与进程强绑定关系）

```
int sw_64_linux_nat_target::remove_watchpoint (CORE_ADDR addr, int len, enum target_hw_bp_type type, struct expression *cond)

{

        enum sw_64_hw_bp_type watch_type;

        struct lwp_info *lp = find_lwp_pid(inferior_ptid);//寻找子进程 lp，前提是调用 add_initial_lwp 初始化子进程 lp 并存入哈希表

//增加下列代码修复该问题：

        if(lp == NULL)//子进程刚创建时，还未调用 add_initial_lwp，因此 lp 为 NULL

                return 0;// 返回 0 表示成功


        pid_t lwp = inferior_ptid.lwp ();

        struct arch_lwp_info *priv = lp->arch_private;//非法内存访问（报错原因）


        watch_type = sw_64_hw_bp_type_from_target_hw_bp_type(type);

        if (sw_64_linux_del_one_watch(lwp, priv, watch_type,addr,len))

        {

                priv->watch_registers_changed =1;

                return 0;

        }

        return -1;

}
```

2.1.3. 父子进程 watch 问题验证

2.1.3.1. 编写测试 demo

```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

#include <semaphore.h>


int main() {

    const char *name = "/my_shm";

    const size_t size = sizeof(int);


    // Create shared memory object

    int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    if (shm_fd == -1) {

        perror("shm_open");

        exit(EXIT_FAILURE);

    }


    // Configure shared memory size

    if (ftruncate(shm_fd, size) == -1) {

        perror("ftruncate");

        exit(EXIT_FAILURE);

    }

```

```
// Map shared memory
int *ptr = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (ptr == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}
printf("Parent: %p\n", ptr);
pid_t pid = fork();

if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (pid == 0) {
    // Child process
    int number = 45;
    memcpy(ptr, &number, sizeof(int));
    printf("Child: Number %d written to shared memory.\n", number);

    // Clean up
    munmap(ptr, size);
    close(shm_fd);
    exit(0);
} else {
    // Parent process
    // Wait for child
    int number = 41;
    memcpy(ptr, &number, sizeof(int));
```

```
        printf("Parent: Number %d written to shared memory.\n", number);

        // Clean up

        wait(NULL);

        munmap(ptr, size);

        close(shm_fd);

        shm_unlink(name);

    }



    return 0;

}
```

2.1.3.2. 调试过程

```
$ gdb ~/fork

GNU gdb (GDB) 9.1 sw1.0.1-dirty

Reading symbols from /data/sbw/fork...

(gdb) start

Temporary breakpoint 1, main () at fork.c:12

12          const char *name = "/my_shm";

(gdb) watch *(int *)0x400000036000

Hardware watchpoint 2: *(int *)0x400000036000

(gdb) set detach-on-fork off

(gdb) c

Continuing.

Parent: 0x400000036000

[New inferior 2 (process 1192759)]



Thread 1.1 "fork" hit Hardware watchpoint 2: *(int *)0x400000036000



Old value = <unreadable>

New value = 41
```

```
0x000040000012109c in memcpy () from /usr/lib/libc.so.6
(gdb) info inferiors
  Num   Description       Executable
* 1     process 1192716   /data/sbw/fork
  2     process 1192759   /data/sbw/fork
(gdb) inferior 2
[Switching to inferior 2 [process 1192759] (/data/sbw/fork)]
[Switching to thread 2.1 (process 1192759)]
Reading symbols from /data/sbw/fork...
#0  0x0000400000151c50 in _Fork () from /usr/lib/libc.so.6
(gdb) watch *(int *)0x400000036000
Hardware watchpoint 3: *(int *)0x400000036000
(gdb) c
Continuing.

Thread 2.1 "fork" hit Hardware watchpoint 3: *(int *)0x400000036000

Old value = 41
New value = 45
0x000040000012109c in memcpy () from /usr/lib/libc.so.6
(gdb) c
Continuing.
Child: Number 45 written to shared memory.
[Inferior 2 (process 1192759) exited normally]
(gdb) inferior 1
[Switching to inferior 1 [process 1192716] (/data/sbw/fork)]
[Switching to thread 1.1 (process 1192716)]
#0  0x000040000012109c in memcpy () from /usr/lib/libc.so.6
(gdb) c
```

```
Continuing.

Thread 1.1 "fork" hit Hardware watchpoint 2: *(int *)0x400000036000

Old value = 41
New value = 45
0x000040000012109c in memcpy () from /usr/lib/libc.so.6
(gdb) c
Continuing.
Parent: Number 41 written to shared memory.
[Inferior 1 (process 1192716) exited normally]
(gdb) q
```

2.1.3.3. 验证后的结论

- gdb 调试父进程设置 watch 后，子进程还需要再次设置；
- gdb 若调试多个进程，则每个进程都需设置，每个进程里都要 watch；
- gdb 调试进程 A 并 watch 共享变量 x，此时进程 B 修改 x，进程 A 是监控不到修改行为的（若进程 A 会一直读 x 的话，在 x 被进程 B 修改后，进程 A 会读到 x 值的变化，会触发 read 观察点停下来）。

2.2. 排查 vpp 共享地址 watch 不到问题

2.2.1. vpp 场景 watch 失效现象

用户态程序 vpp 申请一块共享内存地址，用于和其他进程同步数据，该地址上存储的数据初始值为 2，但运行一段时间后，数据被修改为 0，导致后续一系列报错。

使用 gdb watch 功能监控该共享地址，通过 ptrace 成功将匹配地址、掩码、功能位写入 da_ match、da_mask、dc_ctl 后，dmesg 显示一直在匹配中，但始终没有匹配到地址修改。然而在匹配过程中，地址上的值已经从初始值 2 变成了 0。

vpp 场景下共有 4 个独立的进程能够操作该共享地址，根据之前的验证结论，gdb 需要同时跟踪上这 4 个进程，并在每个进程设置了内存观察点。操作后仍没有收到 SIGTRAP 信号，看 dmesg 发现观察点已经传递给内核并设置成功，但并始终没有匹配成功。

2.2.2. 复现 4 个进程 watch 过程

1、重启服务，查看能够操作该共享地址的 4 个独立进程的 pid

```
root      48697 8.7  0.0 291647600 443128 ?    Ssl  00:58   0:03 /usr/bin/vpp -c /etc/vpp/startup.conf
root      48876 0.0  0.0   6528  2352 pts/7    SN+  00:59   0:00 grep --color=auto vpp
root      48878 0.0  0.0   6528  2352 pts/7    SN+  00:59   0:00 grep --color=auto ctcc_monitor
root      48715 0.4  0.0 1215048 11816 ?       Ssl  00:58   0:00 /usr/bin/sysrepo-plugind -v2
root      48880 0.0  0.0   6528  2352 pts/7    SN+  00:59   0:00 grep --color=auto sysrepo-plugind
frr       48791 0.0  0.0 1288520 9024 ?        SNsl 00:58   0:00 /usr/lib/frr/zebra -d --log file:/var/log/frr/frr.log --log-level
debugging
root      48882 0.0  0.0   6528  2352 pts/7    SN+  00:59   0:00 grep --color=auto zebra
[root@localhost ~]# ps -aux | grep ctcc_monitor
root      48919 0.0  0.0   6528  2352 pts/7    SN+  00:59   0:00 grep --color=auto ctcc_monitor
[root@localhost ~]# journalctl -r -u vpp|grep shmem|less
[root@localhost ~]# journalctl -r -u vpp|grep shmem|less
[root@localhost ~]# journalctl -r -u vpp|grep shmem|less
[root@localhost ~]# ps -aux | grep ctcc_monitor
root      48941 0.0  0.0 477592  5024 pts/2    SN+  01:00   0:00 /usr/bin/ctcc_monitor
root      48958 0.0  0.0   6528  2352 pts/7    SN+  01:00   0:00 grep --color=auto ctcc_monitor
```

2、获取 vpp 创建的共享内存地址，为 0x1300443cc

```
Aug 09 00:58:36 localhost.localdomain /usr/bin/vpp[48697]: vl_init_shmem:500: vl_init_shmem:500: api_main.shmem_hdr=0x1300443cc, vl_p
id=48697, is_private_region=0
```

该共享地址空间从/dev/shm/vpe-api 文件里 mmap 来的

```
[root@localhost ~]# cat /proc/48697/maps
100000000-100060000 rw-p 00000000 00:00 0
103060000-103092000 rw-p 00000000 00:00 0
106092000-1060c4000 rw-p 00000000 00:00 0
1090c4000-1090f6000 rw-p 00000000 00:00 0
10c0f6000-10c128000 rw-p 00000000 00:00 0
10f128000-10f15a000 rw-p 00000000 00:00 0
11215a000-11218c000 rw-p 00000000 00:00 0
11518c000-1151be000 rw-p 00000000 00:00 0
11ffde000-120000000 rw-p 00000000 00:00 0                          [stack]
120000000-120324000 r-xp 00000000 08:03 7349481                    /usr/bin/vpp
120332000-120334000 r--p 00322000 08:03 7349481                    /usr/bin/vpp
120334000-120338000 rw-p 00324000 08:03 7349481                    /usr/bin/vpp
120338000-120570000 rw-p 00000000 00:00 0                          [heap]
130000000-130022000 rw-s 00000000 00:14 994                        /dev/shm/global_vm
130022000-131044000 rw-s 00000000 00:14 995                        /dev/shm/vpe-api
131044000-134000000 rw-s 01044000 00:14 994                        /dev/shm/global_vm
4000000000-400000002e000 r-xp 00000000 08:03 2235539               /usr/lib/ld-linux.so.2
40000002e000-400000030000 r--p 00000000 00:00 0                    [vvar]
400000030000-400000032000 r-xp 00000000 00:00 0                    [vdso]
400000032000-40000003a000 rw-p 00000000 00:00 0
40000003e000-400000040000 r--p 0002e000 08:03 2235539              /usr/lib/ld-linux.so.2
400000040000-400000042000 rw-p 00030000 08:03 2235539              /usr/lib/ld-linux.so.2
```

3、启动 4 个 gdb 分别跟踪这 4 个进程，跟踪上后，首先打印共享内存地址的初始值，看到都为 2，并分别设置地址观察点：

```
[root@localhost ~]# gdb -p 48697
GNU gdb (GDB) 9.1 sw1.0.1-dirty
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-sunway-linux-gnu".
Attaching to process 48697
[New LWP 48701]
0x0000400001c04328 in epoll_pwait () from /usr/lib/libc.so.6
(gdb) c
Continuing.
^C
Thread 1 "vpp_main" received signal SIGINT, Interrupt.
0x0000400001c04328 in epoll_pwait () from /usr/lib/libc.so.6
(gdb) Quit
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb)
```

```
[root@localhost ~]# gdb -p 48715
GNU gdb (GDB) 9.1 sw1.0.0
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-unknown-linux-gnu".
/etc/gdbinit:9: Error in sourced command file:
Scripting in the "Python" language is not supported in this copy of GDB.
Attaching to process 48715
[New LWP 48779]
[New LWP 48780]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
0x00004000002cb2a8 in ?? () from target:/lib/libc.so.6.1
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb)
```

```
[root@localhost ~]# gdb -p 48791
GNU gdb (GDB) 9.1 sw1.0.1-dirty
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-sunway-linux-gnu".
Attaching to process 48791
[New LWP 48792]
[New LWP 48793]
[New LWP 48799]
0x000040000092bed0 in poll () from /usr/lib/libc.so.6
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb)
```

```
[root@localhost ~]# gdb -p 48986
GNU gdb (GDB) 9.1 sw1.0.0
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-unknown-linux-gnu".
/etc/gdbinit:9: Error in sourced command file:
Scripting in the "Python" language is not supported in this copy of GDB.
Attaching to process 48986
[New LWP 48988]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
0x000040000079e874 in clock_nanosleep () from /usr/lib/libc.so.6
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb)
```

4、continue 命令，继续执行这 4 个进程，查看 dmesg，发现下面的打印信息一直刷屏中：

```
[48561.588200] Restroe MATCH status, pid: 48791
[48561.588203] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
```

5、vpp 收到异常停下来了，再查看共享内存地址已经修改为 0 了，然而 dmesg 显示还是处于匹配中，并没有捕获到地址修改。

```
[root@localhost ~]# gdb -p 48697
GNU gdb (GDB) 9.1 sw1.0.1-dirty
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-sunway-linux-gnu".
Attaching to process 48697
[New LWP 48701]
0x0000400001c04328 in epoll_pwait () from /usr/lib/libc.so.6
(gdb) c
Continuing.
^C
Thread 1 "vpp_main" received signal SIGINT, Interrupt.
0x0000400001c04328 in epoll_pwait () from /usr/lib/libc.so.6
(gdb) Quit
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb) c
Continuing.

Thread 1 "vpp_main" received signal SIGSEGV, Segmentation fault.
0x0000400000065710 in ?? () from /usr/lib/libvlibmemory.so.19.01.1
1: *0x1300443cc = 0
(gdb)
```

```
[48584.889022] Restroe MATCH status, pid: 48697
[48584.903095] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48584.931223] Restroe MATCH status, pid: 48697
[48584.945628] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48584.970284] Restroe MATCH status, pid: 48697
[48584.970289] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48584.984720] Restroe MATCH status, pid: 48697
[48584.998790] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48585.026922] Restroe MATCH status, pid: 48697
[48585.041005] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48585.065874] Restroe MATCH status, pid: 48697
[48585.065879] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48585.083470] Restroe MATCH status, pid: 48697
[48585.097538] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48585.125659] Restroe MATCH status, pid: 48697
[48585.139729] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
```

```
[48722.275928] Restroe MATCH status, pid: 48791
[48722.275931] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48722.275956] Restroe MATCH status, pid: 48791
[48722.275960] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48722.275985] Restroe MATCH status, pid: 48791
[48722.276015] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48722.283190] Restroe MATCH status, pid: 48791
[48722.283195] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
[48726.031801] Restroe MATCH status, pid: 48791
[48726.031816] da_match:0x400001300443cc da_mask:0x1ffffffffffffc match_ctl:0x737
~
~
```

2.2.3. 验证 gdb 调试多进程 watch 共享地址问题

了解 vpp 共享内存地址的创建方式是通过 open 打开然后映射到各自的进程空间，编写测试用例构建出类似的场景，测试 gdb watch 功能正常。

2.3. 解决 gdb 对 watch 地址的额外处理

2.3.1. 修改现象

调试过程发现 gdb 会对 watch 的地址做一些处理，例如 watch 的地址为 0x1300443cc，但是 gdb 会处理成 0x400001300443cc，mask 掩码本来应该是 0x1 后面全 f，却变成了 fc 结尾。gdb 处理后将数据通过 ptrace 传给内核，导致内核里打印的数据和 gdb 命令行输入的不一致。

虽然 da_match 的高位 0x4 会被 mask 掩码位消除掉，但还是有些奇怪，另外 mask 最后 2bit 修改为 0，稍微放大了匹配范围。影响不大，但还是需要搞清楚原因。

```
[root@localhost ~]# gdb -p 48791
GNU gdb (GDB) 9.1 sw1.0.1-dirty
Copyright (C) 2020 Free Software Foundation, Inc.
This GDB was configured as "sw_64-sunway-linux-gnu".
Attaching to process 48791
[New LWP 48792]
[New LWP 48793]
[New LWP 48799]
0x000040000092bed0 in poll () from /usr/lib/libc.so.6
(gdb) display *0x1300443cc
1: *0x1300443cc = 2
(gdb) watch *0x1300443cc
Hardware watchpoint 1: *0x1300443cc
(gdb)
```

```
[48561.588200] Restroe MATCH status, pid: 48791
[48561.588203] da_match:0x400001300443cc da_mask:0x1ffffffffffc match_ctl:0x737
```

2.3.2. 修改原因

core3B 软件接口手册了规定 da_match 里 0-52 位用于比较的地址，53-54 位用于读写模式位设置，gdb 里添加高位 0x4 就是将模式位设置为“10”表示允许写地址比较。

gdb 里 sw_64_linux_try_one_watch 接口设置 watchpoint 时，将高位按照 core3B 规定设置成模式位：

```
if (wpt[0].valid)
{
        data = wpt[0].match & ((1L<<53)-1);

        wpt[0].match = sw_64_write;
        wpt->match <<= 53;
        wpt->match |= data;
        /* wpt->mask not changed */
}
```

掩码位最后 2 位为 0，表示匹配范围为 4 字节，即地址结尾为 0、4、8、c，gdb 里处理代码：

```
{
        debug("insert master wp %lx, da_match len = %d", (long)addr, len);
        wpt->match = wpt_type&0x3;
        wpt->match <<= 53;
        wpt->match |= addr & ((1L<<53)-1);
        data = len -1;
        wpt->mask = ~data & ((1L<<53)-1);
        wpt->valid = 1;
}
```

2.3.3. 原因和解决方法

该问题并无实质影响，所以之前使用 watch 时并没察觉。为避免困扰，修改方法便是在 gdb 源码里处理 da_match 时需要区分下 core3B 和 core4，core3B 将模式位设置到 da_match 高位，core4 将模式位设置到 dc_ctlp。

dv_mask 掩码最后几位为 0，是根据观察的地址长度 len 来自适应改变的。如果观察一个 int 类型，则为 4 个字节，len=4，data=len-1=3，data 二进制为 0011，取反后 1100，再与上全 f，最终 mask 低位就是 1100，即 c，加上前面的 f，便是 0xff...fc

若观察 8 个字节的 long 类型，则 mask 就会变成 0xff...f8，这里的 mask 掩码被修改是正确的。

2.4. 获取物理地址并 watch

gdb 里的小问题都修改完后，观察虚拟地址还是抓不到。只能想办法查看对应的物理地址，试下监控物理地址。

2.4.1. 获取物理地址的方法

内核源码里，获取物理地址的代码如下：

```
extern unsigned long show_va_to_pa(struct mm_struct *mm, unsigned long addr);


long arch_ptrace(struct task_struct *child, long request, unsigned long addr, unsigned long data)

{

    ……

    unsigned long addr_pa = show_va_to_pa(child->mm ,data);

    ……

}


struct task_struct {

    ……

    struct mm_struct        *mm;

    struct mm_struct        *active_mm;

    ……
```

```
}
```

2.4.2. 实现 gdb 里观察物理地址

实现思路：修改内核 ptrace 接口，gdb 第一次传入 va，内核打印 va 对应的 pa，再通过 gdb 传入 pa 即可。

1) gdb 先 watch va，然后 next 运行一步，此时会通过 ptrace 把 va 写到 163；
2) 内核在 arch_ptrace 判断是否为 163，若是则直接调用 show_va_to_pa()打印 va 对应的 pa；
3) 查看 dmesg 就会看到 printk 打印出来的 pa（这个 pa 还需要加上 va 后三位偏移量才是真正的 pa。如果怕 pa 不准确，可以根据偏移量查看下前后地址上的值）；
4) 在 gdb 里再次 watch，此时需要 watch pa（内核里添加 flag 判断地址高位，若高位为 0 则认为是 va，不为 0 认为 pa）；
5) gdb 里 continue 后，内核 do_match 接口会监控触发对 pa 的读写操作。

```c
extern unsigned long show_va_to_pa(struct mm_struct *mm, unsigned long addr);
```

```c
	case PTRACE_POKEUSR: /* write the specified register */
		if( addr == 163 ) {
			unsigned long flag;
			flag = (data >> 52);
			if( !flag ){
				addr_pa = show_va_to_pa(child->mm ,data);
				printk("+++++%s PTRACE_POKEUSR  va = data = %#lx   va to pa =%#lx  pid=%d\n", __func__, data, addr_pa, child->pid);
			}
			else {
				addr_pa = data & 0xfffffffffffffUL;
				on_each_cpu(write_da_match, NULL, 1);
			}
		}
		if(addr == 163 || addr == 164 || addr == 167){
			printk("+++++%s PTRACE_POKEUSR addr=%ld data=%#lx \n", __func__, addr, data);
			ret = 0;
		}
		else
			ret = put_reg(child, addr, data);
		break;
```

2.4.3. 验证物理地址监控效果

查看 dmesg 发现 vpp 相关的 4 个进程中的 va 对应同一个 pa，物理地址唯一，接下来监控物理地址即可。

```
Hardware watchpoint 1: *0x1300443cc
(gdb) c
Continuing.
insert watchpoint 0x1300443cc, da_match len = 4
[gdb] store_debug_register: pid = 6051, regno = 163 val = 0x1300443c
[gdb] store_debug_register: pid = 6051, regno = 164 val = 0x1ffffff
[gdb] store_debug_register: pid = 6051, regno = 167 val = 0x301
^C
Thread 1 "vpp_main" received signal SIGINT, Interrupt.
0x000040a722835a98 in epoll_pwait () from /usr/lib/libc.so.6
(gdb) p *0x1300443cc
$2 = 2
(gdb) i proc
process 6051
cmdline = '/usr/bin/vpp -c /etc/vpp/startup.conf'
cwd = '/'
exe = '/usr/bin/vpp'
(gdb)
```

```
[   332.436864] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x11ffff66c
[   332.436880] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[ 1395.678169] +++++arch_ptrace PTRACE_POKEUSR  va = date = 0x1300443cc   va to pa =0x0  pid=6246
[ 1395.678187] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300443cc
[ 1395.678203] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[ 1396.406365] +++++arch_ptrace PTRACE_POKEUSR  va = date = 0x1300443cc   va to pa =0xfff000013a5ea000  pid=6051
[ 1396.406377] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300443cc
[ 1396.406394] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[ 1397.271043] +++++arch_ptrace PTRACE_POKEUSR  va = date = 0x1300443cc   va to pa =0xfff000013a5ea000  pid=6131
[ 1397.271060] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300443cc
[ 1397.271077] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[ 1398.054780] +++++arch_ptrace PTRACE_POKEUSR  va = date = 0x1300443cc   va to pa =0xfff000013a5ea000  pid=6060
[ 1398.054793] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300443cc
[ 1398.054809] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[ 1401.666153] +++++arch_ptrace PTRACE_POKEUSR  va = date = 0x1300443cc   va to pa =0xfff000013a5ea000  pid=6246
[ 1401.666166] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300443cc
[ 1401.666181] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1ffffffffffff
[root@localhost ~]#
```

```
1 sw-107
[Detaching after fork from child process 6251]
[Detaching after fork from child process 6252]
[Detaching after fork from child process 6254]
[Detaching after fork from child process 6255]
[Detaching after fork from child process 6258]
[Detaching after fork from child process 6261]
[Detaching after fork from child process 6264]
[Detaching after fork from child process 6267]
^C
Thread 1 "ctcc_monitor" received signal SIGINT, Interrupt.
0x00004000007a91c0 in wait4 () from /usr/lib/libc.so.6
1: *0x1300443cc = 2
(gdb) i proc
process 6246
cmdline = '/usr/bin/ctcc_monitor'
cwd = '/root/xuhm'
exe = '/usr/bin/ctcc_monitor'
(gdb)
```

```
1 sw-107
Hardware watchpoint 1: *0x1300443cc
(gdb) c
Continuing.
insert watchpoint 0x1300443cc, da_match len = 4
[gdb] store_debug_register: pid = 6131, regno = 163 val
[gdb] store_debug_register: pid = 6131, regno = 164 val
[gdb] store_debug_register: pid = 6131, regno = 167 val
^C
Thread 1 "zebra" received signal SIGINT, Interrupt.
0x0000403ddeec5640 in poll () from /usr/lib/libc.so.6
(gdb) p *0x1300443cc
$3 = 2
(gdb) i proc
process 6131
cmdline = '/usr/lib/frr/zebra -d --log file:/var/log/frr
cwd = '/root/xuhm'
exe = '/usr/lib/frr/zebra'
(gdb)
```

```
1 sw-107
Hardware watchpoint 1: *0x1300443cc
(gdb) c
Continuing.
insert watchpoint 0x1300443cc, da_match len = 4
[gdb] store_debug_register: pid = 6060, regno = 163 val
[gdb] store_debug_register: pid = 6060, regno = 164 val
[gdb] store_debug_register: pid = 6060, regno = 167 val
^[[C^C
Thread 1 "sysrepo-plugind" received signal SIGINT, Inter
0x00004085e90772d8 in ?? () from target:/lib/libc.so.6.1
(gdb) p *0x1300443cc
$2 = 2
(gdb) i proc
process 6060
cmdline = '/usr/bin/sysrepo-plugind -v2'
cwd = '/'
exe = '/usr/bin/sysrepo-plugind'
(gdb)
```

执行过程中，能够看到右边物理地址都被刷成了 0，vpp 程序也报错了：

```
Thread 1 "vpp main" received signal SIGSEGV, Segmentation fault.
0x000040a720c99710 in ?? () from /usr/lib/libvlibmemory.so.19.01.1
(gdb) p *0x1300443cc
$6 = 0
(gdb) i proc
process 6051
cmdline = '/usr/bin/vpp -c /etc/vpp/startup.conf'
cwd = '/'
exe = '/usr/bin/vpp'
(gdb) p *0x1300443cc
$7 = 0
(gdb)
```

```
Invalid object format !
user@user-8A:~$ rmem8a -o 96:0:0:0:0 -a 0x13a5ea3cc -l 0x100 -q -cache
0x013a5ea3cc:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea3ec:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea40c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea42c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea44c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea46c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea48c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x013a5ea4ac:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
time:61
user@user-8A:~$
```

通过 8A 维护命令，可以很直观的看到物理地址变化前后的值：

```
user@user-8A:~$ rmem8a -o 96:0:0:0:0 -a 0x1362a83cc -l 0x100 -q -cache
0x01362a83cc:   00000f7600000002 0000000130046480 0000000130077adc 00000001301c6b4c
0x01362a83ec:   0000000000000000 ffffffff00000000 0000000000000000 0000211300000000
0x01362a840c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a842c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a844c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a846c:   0000000000000000 0000000000000000 000000000000006c 0000000000000000
0x01362a848c:   0000008000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a84ac:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
time:64
user@user-8A:~$ rmem8a -o 96:0:0:0:0 -a 0x1362a83cc -l 0x100 -q -cache
0x01362a83cc:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a83ec:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a840c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a842c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a844c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a846c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a848c:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
0x01362a84ac:   0000000000000000 0000000000000000 0000000000000000 0000000000000000
time:37
```

若不确定物理地址是否准确，可以顺便打印前后的物理地址。因为这块地址保存的是结构体数据，前后地址保存的就是结构体里的前后字段信息，比如图中 f76 就是 pid 字段，2 就是 version 字段，vpp 报错根源就是这个 2 修改了。

2.5. watch 匹配物理地址

2.5.1. 方法 1：物理地址写入 do_match 文件

内核提供一套 match 文件接口，编译时在 kernel/arch/sw_64/kernel/Makefile 里加上 match.o，更换内核后就会在 /proc/sys/debug 目录下看到 da_match/dv_match 等文件（类似 unalign 文件记录对界异常），只需将地址、掩码、控制位正确写入 da_match 文件后，就会触发回调，开始匹配。

```
obj-y    := fpu.o traps.o process.o sys_sw64.o irq.o \
            signal.o setup.o ptrace.o time.o \
            systbls.o dup_print.o chip_setup.o \
            insn.o early_init.o topology.o cacheinfo.o \
            vdso.o vdso/ hmcall.o stacktrace.o idle.o reset.o match.o

obj-$(CONFIG_SUBARCH_C3B) += entry_c3.o tc.o
obj-$(CONFIG_SUBARCH_C4) += entry_c4.o

obj-$(CONFIG_ACPI) += acpi.o
obj-$(CONFIG_SMP)        += smp.o
obj-$(CONFIG_MODULES)    += module.o
obj-$(CONFIG_PM)         += pm.o
obj-$(CONFIG_SUSPEND)    += suspend_asm.o suspend.o
obj-$(CONFIG_PERF_EVENTS) += perf_callchain.o
obj-$(CONFIG_HIBERNATION) += hibernate_asm.o hibernate.o
obj-$(CONFIG_AUDIT)      += audit.o
obj-$(CONFIG_RELOCATABLE)    += relocate.o
obj-$(CONFIG_DEBUG_FS)   += segvdbg.o unaligned.o
obj-$(CONFIG_JUMP_LABEL) += jump_label.o
obj-$(CONFIG_DEBUG_MATCH)        += match.o

ifdef CONFIG_PERF_EVENTS
 NORMAL  Makefile
"~/vpp_bug_fixed/kernel/arch/sw_64/kernel/Makefile" 行 36 / 61 --59%-- 列 30-36
```

kernel/arch/sw_64/kernel/match.c 里面有具体实现过程，但这套机制目前还没在 5.10 版本适配完整，编译都会报错，更没人试用过。经我们试用后发现并没效果，不知道是数据写入格式的问题，还是其他问题，就放弃了这条路。感觉 4.19 版本这套方法相对完善一些。

2.5.2. 方法 2：将物理地址写入 CSR 进行匹配

2.5.2.1. 对所有 CPU 核进行匹配

从 match.c 文件里调用 write_da_match 接口设置 da_match/da_mask/dc_ctl，再通过 on_each_cpu 接口实现对 CPU 所有的核监控，保障每个进程 pcb 里都会设置上，相当于一个全局的 watch 效果，不然每个进程都要 watch 设置一遍 pcb 才行：

```
        case PTRACE_POKEUSR: /* write the specified register */
                if( addr == 163 ) {
                        unsigned long flag;
                        flag = (data >> 52);
                        if( !flag ){
                                addr_pa = show_va_to_pa(child->mm ,data);
                                printk("+++++%s PTRACE_POKEUSR  va = data = %#lx   va to pa =%#lx  pid=%d\n", __func__, data, addr_pa, child->pid);
                        }
                        else {
                                addr_pa = data & 0xfffffffffffffUL;
                                on_each_cpu(write_da_match, NULL, 1);
                        }
                }
                if(addr == 163 || addr == 164 || addr == 167){
                        printk("+++++%s PTRACE_POKEUSR addr=%ld data=%#lx \n", __func__, addr, data);
                        ret = 0;
                }
                else
                        ret = put_reg(child, addr, data);
                break;
```

```c
/*
 * Call a function on all processors.  May be used during early boot while
 * early_boot_irqs_disabled is set.  Use local_irq_save/restore() instead
 * of local_irq_disable/enable().
 */
void on_each_cpu(smp_call_func_t func, void *info, int wait)
{
        unsigned long flags;

        preempt_disable();
        smp_call_function(func, info, wait);
        local_irq_save(flags);
        func(info);
        local_irq_restore(flags);
        preempt_enable();
}
EXPORT_SYMBOL(on_each_cpu);
```

```c
static void
write_da_match(void *i)
{
                unsigned long dc_ctl, maskkk;
                maskkk = 0x1ffffffffffff8UL;

                sw64_write_csr(addr_pa, CSR_DA_MATCH);
                sw64_write_csr(maskkk, CSR_DA_MASK);
        dc_ctl = sw64_read_csr(CSR_DC_CTLP);
        dc_ctl &= ~((0x1UL << 3) | (0x3UL << DA_MATCH_EN_S)
                        | (0x1UL << DAV_MATCH_EN_S) | (0x1UL << DPM_MATCH_EN_S)
                        | (0x3UL << DPM_MATCH));
        dc_ctl |= 0x3f;
        sw64_write_csr(dc_ctl, CSR_DC_CTLP);
                pr_info(" PTRACE_POKEUSR SET CSR da_match:%#lx da_mask:%#lx match_ctl:%#lx\n", addr_pa , maskkk , dc_ctl);
}
```

2.5.2.2. 成功抓到物理地址修改

抓到了物理地址命中，看到进程名为 ctcc_monitor，cause 为 0x1 表示写，还会捕获到其他，但都是 cause 为 0 表示读，我们只需要关注地址被写的情况：

```
[root@localhost ~]# dmesg
[16682.438257] +++++arch_ptrace PTRACE_POKEUSR  va = data = 0x1300483cc   va to pa =0xfff00001176a8000  pid=9181
[16682.438273] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x1300483cc
[16682.438288] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1fffffffffffff
[16682.438298] +++++arch_ptrace PTRACE_POKEUSR addr=167 data=0x301
[16707.119930]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119935]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119941]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119946]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119952]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119957]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119963]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.119998]  PTRACE_POKEUSR SET CSR da_match:0x1176a83cc da_mask:0x1ffffffffffff8 match_ctl:0x3f
[16707.120010] +++++arch_ptrace PTRACE_POKEUSR addr=163 data=0x100001176a83cc
[16707.120022] +++++arch_ptrace PTRACE_POKEUSR addr=164 data=0x1fffffffffffff
[16707.120032] +++++arch_ptrace PTRACE_POKEUSR addr=167 data=0x301
```

成功监控到地址修改后，发出 SIGTRAP 信号给 gdb

```
do_match: pid 4166, name = ctcc_monitor, cause = 0x1, mmcsr = 0x100, address = 0x13004a3c8, pc 0x41e3bde246b4
CPU: 4 PID: 4166 Comm: ctcc_monitor Not tainted 5.10.0-39.0.0.45.sw_64 #1
Hardware name: WIAT WIAT Platform Software, BIOS edk2-202105 May 21 2024
pc = [<000041e3bde246b4>]  ra = [<00000001200c7fac>]  ps = 0008    Not tainted
pc is at 0x41e3bde246b4
ra is at 0x1200c7fac
v0 = 000000012be1ff50  t0 = 0000000000000001  t1 = 000000000058fd8e
t2 = 000000000058fda6  t3 = 0000000000dd5224  t4 = 000000013004a3c0
t5 = 0000000132cc90f0  t6 = 000000013004a3c0  t7 = 0000000000000000
s0 = 000000011f871db8  s1 = 0000000000000001  s2 = 0000000000000000
s3 = 000000011f871dc8  s4 = 000041e3bd6e91f0  s5 = 0000000120118390
s6 = 000000011f871b70
a0 = 000000012be1ff50  a1 = 0000000000000000  a2 = 0000000000000000
a3 = 0000000000000000  a4 = 0000000000000000  a5 = 0000000000000011
t8 = 0000000000000000  t9 = 0000000000000001  t10 = ffffffffffffffb
t11= 0000000000000000  pv = 000041e3bde245e0  at = 0000000000000000
gp = 00000001201324f0  sp = 000000011f871b70
do_page_fault: want to send SIGTRAP, pid = 4166
ost ~]#
```

2.5.2.3. 查看修改地址的 pc

当前 pc 也能看到，但 gdb 里找不到该 pc 处的指令，现场已经被破坏掉了：

```
(gdb) display /10i 0x41e3bde246b4-20
3: x/10i 0x41e3bde246b4-20
   0x41e3bde246a0:      sys_call        0
   0x41e3bde246a4:      sys_call        0
   0x41e3bde246a8:      sys_call        0
   0x41e3bde246ac:      sys_call        0
   0x41e3bde246b0:      sys_call        0
   0x41e3bde246b4:      sys_call        0
   0x41e3bde246b8:      sys_call        0
   0x41e3bde246bc:      sys_call        0
   0x41e3bde246c0:      sys_call        0
   0x41e3bde246c4:      sys_call        0
(gdb)
```

```
[  614.570486] t11= 0000000000000000  pv = fff0000001168340  at = 0000000000000000
[  614.570491] gp = fff000000278dfb8  sp = fff0000107a97cd0
[  614.570496]  pid 4214 gdb not be ptraced, return
[  620.080951] do_match: pid 4166, name = ctcc_monitor, cause = 0x1, mmcsr = 0x100, address = 0x13004a3c8, pc 0x41e3bde246b4
[  620.080974] CPU: 4 PID: 4166 Comm: ctcc_monitor Not tainted 5.10.0-39.0.0.45.sw_64 #1
[  620.080980] Hardware name: WIAT WIAT Platform Software, BIOS edk2-202105 May 21 2024
[  620.080988] pc = [<000041e3bde246b4>]  ra = [<00000001200c7fac>]  ps = 0008    Not tainted
[  620.080997] pc is at 0x41e3bde246b4
[  620.081002] ra is at 0x1200c7fac
[  620.081006] v0 = 000000012be1ff50  t0 = 0000000000000001  t1 = 000000000058fd8e
[  620.081012] t2 = 000000000058fda6  t3 = 0000000000dd5224  t4 = 000000013004a3c0
[  620.081017] t5 = 0000000132cc90f0  t6 = 000000013004a3c0  t7 = 0000000000000000
```

内核里调用 show_code 接口，获取 pa 匹配命中时的 pc 处的指令机器码：

```
extern  void show_code(unsigned int *pc);
```

```
void show_code(unsigned int *pc)
{
        long i;
        unsigned int insn;

        printk("Code:");
        for (i = -6; i < 2; i++) {
                if (__get_user(insn, (unsigned int __user *)pc + i))
                        break;
                printk("%c%08x%c", i ? ' ' : '<', insn, i ? ' ' : '>');
        }
        printk("\n");
}
```

```
#elif defined(CONFIG_SUBARCH_C4)
int do_match(unsigned long address, unsigned long mmcsr, long cause, struct pt_regs *regs)
{
        kernel_siginfo_t info;
        unsigned long match_ctl, ia_match;
        sigval_t sw64_value;

        printk("%s: pid %d, name = %s, cause = %#lx, mmcsr = %#lx, address = %#lx, pc %#lx\n",
                        __func__, current->pid, current->comm, cause, mmcsr, address, regs->pc);


        switch (mmcsr) {
        case MMCSR__DA_MATCH:
        case MMCSR__DV_MATCH:
        case MMCSR__DAV_MATCH:
        case MMCSR__IA_MATCH:
        case MMCSR__IDA_MATCH:
        case MMCSR__IV_MATCH:
                show_regs(regs);

                show_code((unsigned int *)regs->pc);

```

此时可以看到 pc 处的机器码<ae250008>：

```
[ 4344.131410]  pid 61930 gdb not be ptraced, return
[ 4387.929951] do_match: pid 61883, name = ctcc_monitor, cause = 0x1, mmcsr = 0x100, address = 0x1300463c8, pc 0x418dd098e6b4
[ 4387.929974] CPU: 5 PID: 61883 Comm: ctcc_monitor Not tainted 5.10.0-39.0.0.45.sw_64 #1
[ 4387.929981] Hardware name: WIAT WIAT Platform Software, BIOS edk2-202105 May 21 2024
[ 4387.929988] pc = [<0000418dd098e6b4>]  ra = [<00000001200c7fac>]  ps = 0008    Not tainted
[ 4387.929997] pc is at 0x418dd098e6b4
[ 4387.930001] ra is at 0x1200c7fac
[ 4387.930006] v0 = 000000012be1ff50  t0 = 0000000000000001  t1 = 000000000059058e
[ 4387.930012] t2 = 00000000005905a6  t3 = 0000000000dd5224  t4 = 00000001300463c0
[ 4387.930017] t5 = 0000000132cc90f0  t6 = 00000001300463c0  t7 = 0000000000000000
[ 4387.930023] s0 = 000000011ff0fdb8  s1 = 0000000000000001  s2 = 0000000000000000
[ 4387.930028] s3 = 000000011ff0fdc8  s4 = 0000418dd02531f0  s5 = 0000000120118390
[ 4387.930032] s6 = 000000011ff0fb70
[ 4387.930036] a0 = 000000012be1ff50  a1 = 0000000000000000  a2 = 0000000000000000
[ 4387.930041] a3 = 0000000000000000  a4 = 0000000000000000  a5 = 0000000000000011
[ 4387.930047] t8 = 0000000000000000  t9 = 0000000000000001  t10 = fffffffffffffffb
[ 4387.930052] t11= 0000000000000000  pv = 0000418dd098e5e0  at = 0000000000000000
[ 4387.930056] gp = 00000001201324f0  sp = 000000011ff0fb70
[ 4387.930060] Code:
[ 4387.930063]  fc3f0008
[ 4387.930066]  42410541
[ 4387.930069]  c0200019
[ 4387.930073]  48630122
[ 4387.930076]  ae250000
[ 4387.930079]  43ff075f
[ 4387.930082] <ae250008>
[ 4387.930086]  ae250010

[ 4387.930094] do_page_fault: want to send SIGTRAP, pid = 61883
[root@localhost ~]#
```

2.5.2.4. 反汇编 pc 处的指令

先将对应机器码按下述格式保存成.S 文件，再调用汇编器编译成目标文件，最后通过 objdump 反汇编，发现是条 64 位写指令 stl $r17,8($r5)

```
lhx@107:~/vpp_bug_fixed/dump_code$ cat code.S
.long 0xfc3f0008
.long 0x42410541
.long 0xc0200019
.long 0x48630122
.long 0xae250000
.long 0x43ff075f
.long 0xae250008
.long 0xae250010
lhx@107:~/vpp_bug_fixed/dump_code$ as code.S -o code
lhx@107:~/vpp_bug_fixed/dump_code$ objdump -d code

code:     文件格式 elf64-sw_64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   08 00 3f fc     ldih    $r1,8
   4:   41 05 41 42     cmple   $r18,$r1,$r1
   8:   19 00 20 c0     beq     $r1,0x70
   c:   22 01 63 48     subl    $r3,0x18,$r2
  10:   00 00 25 ae     stl     $r17,0($r5)
  14:   5f 07 ff 43     nop
  18:   08 00 25 ae     stl     $r17,8($r5)
  1c:   10 00 25 ae     stl     $r17,16($r5)
lhx@107:~/vpp_bug_fixed/dump_code$
```

2.5.2.5. 利用 systemtap 打印堆栈

追查 stl $r17,8($r5)这条指令来源于哪个程序，从哪里过来的？出错时 gdb 现场已经被破坏，无法打印堆栈。内核的 dump_stack()只能查内核里面的调用栈。

此时可以利用 systemtap 打印 do_match 调用时的用户态、内核态的调用栈。

编写 systemtap 脚本，当内核函数 do_match 被调用时，打印堆栈：

```
stap -v -e 'probe kernel.function("do_match"){printf("%s\n%s",sprint_ubacktrace(),sprint_backtrace())}'
```

systemtap 打印出的结果：

```
-------

[5493](ctcc_monitor)address=0x1300443c8 mmcsr=0x100 cause=0x1 regs=0xfff000010c65beb0

memset+0xd4 [libc.so.6]

compute_tgw_region_bandwidth+0x540 [ctcc_monitor]

main+0xfe8 [ctcc_monitor]

__libc_init_first+0x98 [libc.so.6]

__libc_start_main+0xc4 [libc.so.6]

__start+0x60 [ctcc_monitor]



do_match+0x0 [kernel]

do_match+0x0 [kernel]

do_page_fault+0x4fc [kernel]

entMM+0x120 [kernel]

0x4000007786b4



---------
```

通过 systemtap 成功找到了报错时的调用栈：ctcc_monitor-->compute_tgw_region_bandwidth-->memset

2.5.2.6. 调试 ctcc_monitor，定位原因

要求天翼云提供 debug 版本 ctcc_monitor 和源码后，在 compute_tgw_region_bandwidth 函数打断点，很快就找到了源头：ctcc_monitor 执行 713 行的 memset 后，共享地址从 2 变成 0

```
713         memset(cross_region_value_sum, 0, sizeof(cross_region_value_sum));
1: *0x1300443cc = 2
(gdb) n
```

```
714         cross_region_value_num = 0;
1: *0x1300443cc = 0
(gdb) i reg pc
pc             0x1200c7fb4         0x1200c7fb4 <compute_tgw_region_bandwidth+1352>
```

查看 713 行周围代码，发现 cross_region_value_sum 这个结构体占用空间较大，调用 memset 时操作的内存地址范围为 0x12be1ff50~0x132cc90f0，会把这一段内存刷掉，其中就覆盖了共享变量的地址 0x1300443cc。

```
(gdb) p/x sizeof(cross_region_value_sum)
$2 = 0x6ea91a0
(gdb) bt
#0  compute_tgw_region_bandwidth (cur_timestamp=1723818927)
    at /root/bangyn/tgw.openEuler.0802/ctcc_monitor/src/vapi/vapi_get_info.c:714
#1  0x0000000120066908 in main (argc=1, argv=0x11ffff7b8)
    at /root/bangyn/tgw.openEuler.0802/ctcc_monitor/src/monitor/monitor.c:611
(gdb) p/x sizeof(cross_region_value_sum)
$3 = 0x6ea91a0
(gdb) ptype cross_region_value_sum
type = struct {
    char tenant_id[64];
    sw_if_tgw_tunnel_stat_t tgw_tunnel_statis[1000];
    int tenant_tunnel_sum;
} [500]
(gdb) ptype &cross_region_value_sum
type = struct {
    char tenant_id[64];
    sw_if_tgw_tunnel_stat_t tgw_tunnel_statis[1000];
    int tenant_tunnel_sum;
} (*)[500]
(gdb) p &cross_region_value_sum
$4 = (sw_if_tgw_stat_sum_t (*)[500]) 0x12be1ff50 <cross_region_value_sum>
(gdb) p/x 0x12be1ff50+0x6ea91a0
$5 = 0x132cc90f0
(gdb) 0x12be1ff50-----0x132cc90f0
Undefined command: "0x12be1ff50-----0x132cc90f0".  Try "help".
(gdb) i proc
process 9649
cmdline = '/root/ctcc_monitor/ctcc_monitor'
cwd = '/root/ctcc_monitor'
exe = '/root/ctcc_monitor/ctcc_monitor'
(gdb)
```

```
[root@localhost tgw.openEuler.0802]# cat /proc/9649/maps
11ffde000-120000000 rw-p 00000000 00:00 0                  [stack]
120000000-120108000 r-xp 00000000 08:03 25617067           /root/ctcc_monitor/ctcc_monitor
120118000-12011a000 r--p 00108000 08:03 25617067           /root/ctcc_monitor/ctcc_monitor
12011a000-120130000 rw-p 0010a000 08:03 25617067           /root/ctcc_monitor/ctcc_monitor
120130000-130000000 rw-p 00000000 00:00 0
130000000-130022000 rw-s 00000000 00:13 1474               /dev/shm/global_vm      0x12be1ff50-----0x132cc90f0
130022000-131044000 rw-s 00000000 00:13 1475               /dev/shm/vpe-api
131044000-134000000 rw-s 01044000 00:13 1474               /dev/shm/global_vm
134000000-13a97a000 rw-p 00000000 00:00 0                  [heap]
```

2.5.2.7. 追溯 vpp 代码根源，提出解决方法

根据 ctcc_monitor 里 memset 地址覆盖的报错位置，进一步定位到 vpp 源码里 svm_get_global_region_base_va 函数，

这个函数里采用了默认的适用于 x86 的硬编码地址，其中 aarch64 根据自己架构特征计算出一个基地址，sw64 也要类似 aarch64 计算出来一个合理的地址才行。

错误根源：

由于 sw64 与 x86/aarch64 架构的虚拟地址空间分布的不同之处，而申威 vpp 源码 svm_get_global_region_base_va 函数里并未未添加 sw64 分支代码，导致沿用了默认的 x86 架构的硬编码地址 0x130000000，该地址与申威虚拟地址空间冲突，需结合申威虚拟地址空间特性，动态计算出一个合适的基地址（后经分析，发现可以沿用 aarch64 的分支代码，加上“#if __aarch64__ || __sw_64__”，和 aarch64 同样方法计算出一个基地址，经验证该方法有效，成功解决该问题）。

```
u64
svm_get_global_region_base_va ()
{
#if __aarch64__
  /* On AArch64 VA space can have different size, from 36 to 48 bits.
     Here we are trying to detect VA bits by parsing /proc/self/maps
     address ranges */
  int fd;
  unformat_input_t input;
  u64 start, end = 0;
  u8 bits = 0;

  if ((fd = open ("/proc/self/maps", 0)) < 0)
    clib_unix_error ("open '/proc/self/maps'");

  unformat_init_clib_file (&input, fd);
  while (unformat_check_input (&input) != UNFORMAT_END_OF_INPUT)
    {
      if (unformat (&input, "%llx-%llx", &start, &end))
        end--;
      unformat_skip_line (&input);
    }
  unformat_free (&input);
  close (fd);

  bits = count_leading_zeros (end);
  bits = 64 - bits;
  if (bits >= 36 && bits <= 48)
    return ((1ul << bits) / 4) - (2 * SVM_GLOBAL_REGION_SIZE);
  else
    clib_unix_error ("unexpected va bits '%u'", bits);
#endif

  /* default value */
  return 0x130000000ULL;
}
```

2.6. 追查虚拟地址 watch 不到的原因

2.6.1. 测试用例

以下测试用例对共享地址进行初始化操作，循环使用 vstd $V1,0($r5)指令(动态库使用向量指令)对地址范围 0x400000036000-0x400000036190 共 400 个字节进行写操作，$r5 操作数依次为 0x400000036000、0x400000036020、0x400000036040 ……，每次增加 32 个字节，gdb 输入命令 watch *（int *）0x400000036004，监测以地址 0x400000036004 为始的四个字节，因为 0x400000036004 未出现在操作数集合内，所以不能引发硬件地址监测中断，gdb 不能打印内存变化。修改 watch 命令为 watch *（intv32 *）0x400000036004，掩掉地址后 5 位，监测范围扩大至 0x400000036000-0x40000003601f，其中 0x400000036000 为操作数，成功引发中断，gdb 接收到 SIGTRAP 信号打印内存变化。

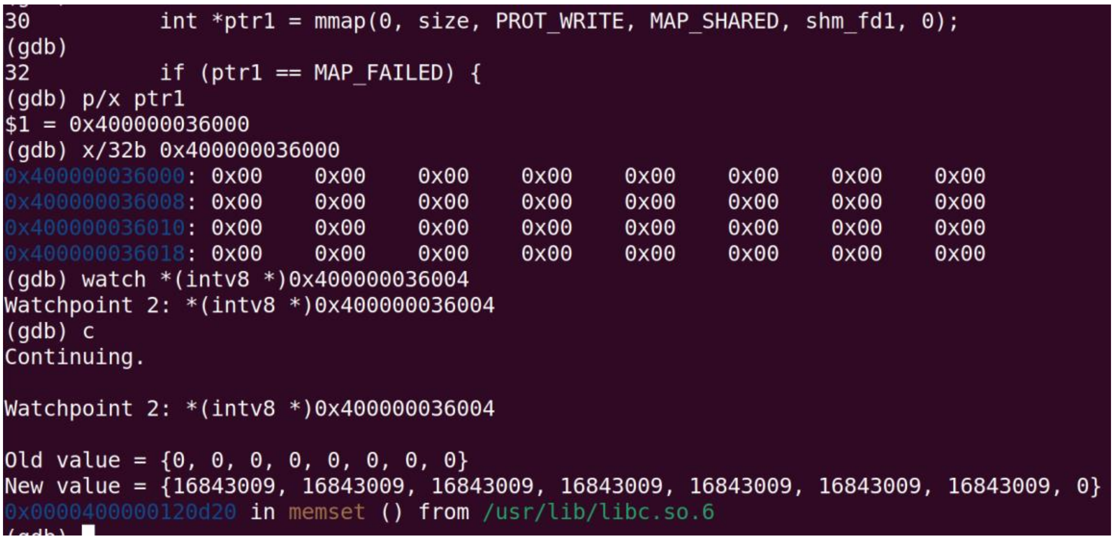
```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <unistd.h>

#include <string.h>


int main() {

    typedef int intv32 __attribute__ ((vector_size (32)));

    intv8 aa={1,2,3,4,5,6,7,8};

    const char *name = "shm";

    const size_t size = 100*sizeof(int);


    int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    if (shm_fd == -1) {

        perror("shm_open");

        exit(EXIT_FAILURE);

    }


    if (ftruncate(shm_fd, size) == -1) {
```

```
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }

    int *ptr = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    int number = 0x2;
    memset(ptr, 3, size);
    printf("Number %d written to shared memory.\n", number);
    munmap(ptr, size);
    close(shm_fd);
    shm_unlink(name);
    return 0;
}
```

图 1 watch 功能失效

```
30          int *ptr1 = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd1, 0);
(gdb)
32          if (ptr1 == MAP_FAILED) {
(gdb) p/x ptr1
$1 = 0x400000036000
(gdb) x/32b 0x400000036000
0x400000036000: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036008: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036010: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036018: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) watch *(int *)0x400000036004
Hardware watchpoint 2: *(int *)0x400000036004
(gdb) c
Continuing.
[gdb] insert watchpoint 0x400000036004, da_match len = 4
[gdb] store_debug_register: pid = 219478, regno = 163 val = 0x400000036004
[gdb] store_debug_register: pid = 219478, regno = 164 val = 0x1ffffffffffffc
[gdb] store_debug_register: pid = 219478, regno = 167 val = 0x301
Number 1 written to shared memory.
[Inferior 1 (process 219478) exited normally]
```

图 2 watch 功能正常



```
30          int *ptr1 = mmap(0, size, PROT_WRITE, MAP_SHARED, shm_fd1, 0);
(gdb)
32          if (ptr1 == MAP_FAILED) {
(gdb) p/x ptr1
$1 = 0x400000036000
(gdb) x/32b 0x400000036000
0x400000036000: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036008: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036010: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x400000036018: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) watch *(intv8 *)0x400000036004
Watchpoint 2: *(intv8 *)0x400000036004
(gdb) c
Continuing.

Watchpoint 2: *(intv8 *)0x400000036004

Old value = {0, 0, 0, 0, 0, 0, 0, 0}
New value = {16843009, 16843009, 16843009, 16843009, 16843009, 16843009, 16843009, 0}
0x0000400000120d20 in memset () from /usr/lib/libc.so.6
```

2.6.2. 计算对齐地址

常规版 memset，使用标量访存指令 stl，地址对齐要求：

watch *（long *）addr，stl 指令写 8 个字节，watch 应设置为 8 字节对齐，屏蔽地址后 3 位，监测 8 字节内存地址范围。

向量优化版 memset，使用向量访存指令 vstd，地址对齐要求：

watch *（intv8 *）addr，vstd 指令写 32 个字节，watch 应设置为 32 字节对齐，屏蔽地址后 5 位，监测 32 字节内存地址范围。

2.6.3. 结论

若使用常规版 memset，程序使用 stl 指令访存时则需要保证观察地址 8 字节对齐。

若使用向量优化版 memset，此时针对 vstd 访存指令则需要观察 32 字节对齐的地址。