XNU: Reference Count Leak in POSIX Shared Memory Object

Zhuo Liang

April 2, 2019

Background

XNU supports Shared Memory[1] for inter process communication. The kernel provides two kinds of memory-sharing mechanisms: POSIX shared memory and System V shared memory.

```c
int main( int argc, char** argv ) {
    int fd;
    unsigned* addr;
    /* Create a new memory object */
    fd = shm_open( "/bolts", O_RDWR | O_CREAT, 0777 );
    /* Set the memory object's size */
    ftruncate( fd, sizeof( *addr ) );
    /* Map the memory object */
    addr = mmap( 0, sizeof( *addr ), PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0 );

    /* Write to shared memory */
    *addr = 1;

    /* The memory object remains in the system after the close */
    close( fd );
    /*
     * To remove a memory object you must unlink it like a file.
     * This may be done by another process.
     */
    shm_unlink( "/bolts" );
    return EXIT_SUCCESS;
}
```

Listing 1: POSIX shared memory usage

[1] https://en.wikipedia.org/wiki/Shared_memory

Listing 1 is a typical usage[2] of POSIX shared memory. The steps can be divided into following items:

1. **shm_open** Create a new shared memory object and put it into cache. Several times of **shm_open** is supported and they will share the common shared memory object from kernel cache.

2. **ftruncate** Allocate backend sharing memory for shared memory object and this operation will mark the object as **PSHM_ALLOCATED**.

3. **mmap** Map the allocated memory into the process's space and the returned value is the start of the shared memory.

4. **Direct Read/Write** Since the memory is already mapped in the process's task space, the process can read and write the shared memory now.

5. **close** Release the file descriptor and decrease the reference of the shared memory object.

6. **shm_unlink** Unlink the path, this operation would decrease the reference count of shared memory object and mark the object as **PSHM_REMOVED**.

Leak Issue

This issue is about the management of shared memory object. The **close** operation of POSIX shared memory object is **pshm_closefile** which will call **pshm_close**.

```
1  // bsd/kern/posix_shm.c
2  static int
3  pshm_close(struct pshminfo * pinfo, int dropref)
4  {
5      int error = 0;
6      struct pshmobj *pshmobj, *pshmobj_next;
7
8      /*
9       * If we are dropping the reference we took on
10      * the cache object, don't enforce the
11      * allocation requirement.
```

[2] www.qnx.com shared memory

```
     */
    if (!dropref &&
        ((pinfo->pshm_flags & PSHM_ALLOCATED)
        != PSHM_ALLOCATED)) { // [a]
        return (EINVAL);
    }               /* DIAGNOSTIC */
    pinfo->pshm_usecount--; /* release this fd's reference */ // [b]
    ...
}
```

Listing 2: **pshm_close** function

At **[a]**, **PSHM_ALLOCATED** is checked and this flag is only set in **ftruncate**. **pshm_closefile** passes **0** as the second parameter to this function and this means if we open the shared memory and close at once, the **pshm_usecount** will not be decreased. Let's see what would happen if we perform following steps:

```
    const char *shm_name = "/test.shm";
    int shm_fd = shm_open(shm_name, O_RDWR | O_CREAT, 0666); // [c]
#define MAX_OPEN_TIMES 0xff
    for (size_t i = 0; i < MAX_OPEN_TIMES; i++) {
        int reopen_shm_fd = shm_open(shm_name, O_RDWR); // [d]
        close(reopen_shm_fd); // [e]
    }
```

Listing 3: Corrupting steps

1. **[c]** Create the memory object and the put it into cache, the **pshm_usecount** is 2 now. One for file descriptor and the other for cache.

2. **[d]** Open the same path, this will search the object from kernel cache and increase the **pshm_usecount**.

```
// bsd/kern/posix_shm.c
int
shm_open(proc_t p, struct shm_open_args *uap, int32_t *retval) {
    /*
     * If we find the entry in the cache, this
     * will take a reference, allowing us to
     * unlock it for the permissions check.
     */
    error = pshm_cache_search(&pinfo, &nd, &pcache, 1);
}
```

Listing 4: Search object from cache

3. **[e]** Close the reopened file descriptor, recall the aforementioned close operation that this will not decrease the **pshm_usecount** because **ftruncate** has not been called yet.

After above steps, the **pshm_usecount** will be **0x101**. One for **shm_fd** which we still hold, one for cache and 0xff for reopened file descriptors which we already closed. And the **pshm_usecount** is a 32 bit integer which means if we set **MAX_OPEN_TIMES** to **0xffffffff**, the result of **pshm_usecount** will be **1**, but we still hold one file descriptor. If we **unlink** the path, which will decrease the usecount and of course release the object memory, and then do anything on that file descriptor, an use-after-free issue occurs!

Fixing

Apple adds a function named **pshm_deref**, which will be called when closing a handle of POSIX shared memory object or unlinking the path, to fix this issue.

```
int64 pshm_deref(__int64 a1, __int64 a2)
{
  __int64 v2; // rsi
  int v3; // eax
  __int64 result; // rax
  _QWORD *i; // rbx
  _DWORD *v6; // r13
  __int64 v7; // rax
  __int64 v8; // [rsp-8h] [rbp-30h]

  v8 = a1;
  v2 = 1LL;
  lck_mtx_assert(&psx_shm_subsys_mutex, 1LL);
  v3 = *(_DWORD *)(a2 + 4);
  result = (unsigned int)(v3 - 1);
  *(_DWORD *)(a2 + 4) = result;
  if ( !(_DWORD)result )
  {
    ...
  }
  return result;
}
```

Listing 5: **pshm_deref** function

Timeline

1. **2018/12/04** Discovery of this issue.
2. **2018/12/11** Reported to product-security@apple.com.
3. **2019/03/13** Checked that the issue was fixed in Beta4.