

Linux Shared Memory Verification Using BLAST

Group members:

- 1) Anup Patel(07305042)
- 2) Sapan Shah(07305061)
- 3) Nilesh Padariya(07305064)

Goal:

To verify the behavior of “shmget API” function of SharedMemory module of Linux Kernel 2.6.9 using BLAST.

Description:

Shared memory is one of the method to achieve Inter Process Communication (IPC). The code that we verified is an API of shared memory implemented in XSI IPC module of linux 2.6.9.

To write or read a shared memory, we must have an identifier for the shared memory segment which is referred to by a non-negative integer. The identifier is an internal name for the shared memory object. A new shared memory segment is created using the *shmget* function.

```
int shmget(key_t key, size_t size, int flag);  
Returns: shared memory ID if OK, -1 on error
```

The *key* parameter is an external name of shared memory segment that is used by cooperating processes. If *key* is specified as `IPC_PRIVATE` then a new memory segment is always created. If user specifies a non-zero *key* then the function first checks whether a memory segment with this *key* exists. If it exists then the function returns it. If it doesn't exist and `IPC_CREAT` flag bit is set then new memory segment is created and returned.

The *size* parameter is the size of the shared memory segment in bytes.

The *flag* parameter is the permission bits to access the shared memory segment as shown below.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Kernel stores all the shared memory segments in the system datastructure *ipc_ids* as shown in the figure. It contains a list of ipc ids that are allocated. *in_use* field contains total number of segments currently in the memory. Whenever a new segment is created this field is incremented. We precisely check this property.

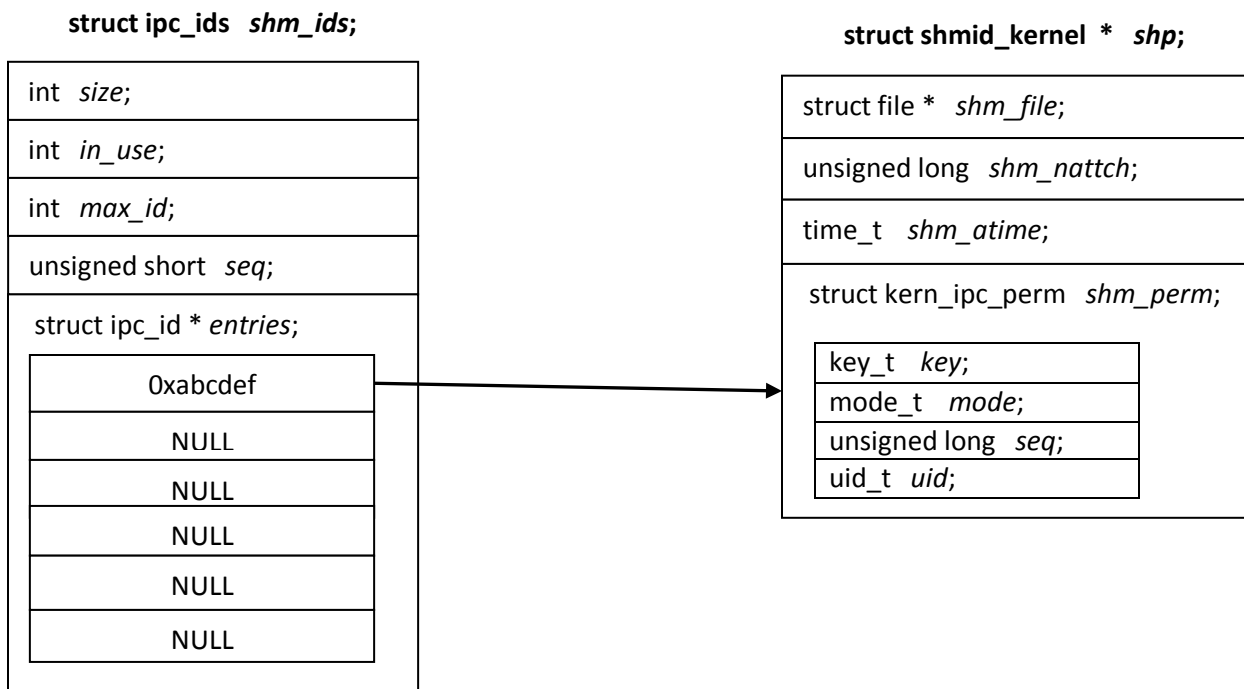


Figure: Kernel datastructures maintaining shared memory segments.

Strategy:

Following are the steps for the verification process that we applied.

- We first separated `sys_shmget()` and all dependent function into `sharedMemory.c` and all dependent structure definitions into `sharedMemory.h`
- We identified some structures that were not relevant for the property that we want to verify and replaced them with stub structures. These structures include `user_struct`, `task_struct`, `file_operations`, `file`, `dentry`, `inode`.
- All locking and memory functions are ignored and we allow blast to take most conservative approach for them. Apart from this, there were two functions which could have affected all the traces that BLAST considers which are, `security_shm_alloc` (line 124, `sharedMemory.c`) and `PTR_ERR` (line 140, `sharedMemory.c`). To abstract these functions we followed linux code convention that whenever function returns an error code, the error value in failure condition is less than zero. We abstracted these functions as shown below.

```
error = security_shm_alloc(shp);
if (error) {
    ipc_rcu_putref(shp);
    return error;
}
```

Is replaced by,

```
error = security_shm_alloc(shp);
if (error) {
    if(error<0){ //newly added code to assume linux error code convention
        ipc_rcu_putref(shp);
        return error;
    } //newly added code to assume linux error code convention
}
```

```
error = PTR_ERR(file);
if (IS_ERR(file))
    goto no_file;
```

Is replaced by,

```
error = PTR_ERR(file);
if (IS_ERR(file))
    if (error < 0) //newly added code to assume linux error code convention
        goto no_file;
```

- We have resolved all the dependencies. Now we insert the assertions required to verify the properties as described below.

The *in_use* count must be incremented by 1 whenever the function is called with IPC_PRIVATE flag set.

```
if((shm_ids.in_use != old_inuse + 1) && (key == IPC_PRIVATE) && (err > 0))//newly added code
    assert(0);
```

The *in_use* count must be incremented by 1 whenever any non existing key is provided with IPC_CREAT flag.

```
if((shm_ids.in_use != old_inuse + 1) && (id == -1) && (key != IPC_PRIVATE) && (err > 0) &&
    (shmflg & IPC_CREAT))//newly added code
    assert(0);
```

- Finally, we verified the code using following options of the BLAST.
“-traces” shows all the traces in an html file.
“-debug” prints warning and debug information and also prints location specific predicates.
“-talias” informs BLAST to consider predicates having lvalues of variables.

Results and Conclusion:

With the above mentioned settings we were successfully verified our properties. The entire verification process is not just passing a C code to BLAST, but the process essentially required assumptions and overapproximations to prove the properties. We also need to consider specific features of BLAST to deal with pointers. The challenging task of our study is how BLAST can be used to verify code that involved pointers.