A2 REFLECTION

INTRODUCTION:

In this assignment , the objective was to implement **SCM_PAGES** , and to begin we first needed to understand the implementation of **SCM_RIGHTS** to gain insights into the internalization and externalization functions. The process started by first navigating to the **/usr/src/sys/kern** directory and using the **grep utility** to locate the relevant code related to **SCM_RIGHTS**. I used the following command – **"grep -r SCM_RIGHTS"**.

```
~
s4733163@comp3301:/usr/src/sys/kern$ vim uipc_socket.c
s4733163@comp3301:/usr/src/sys/kern$ grep -r SCM_RIGHTS
kern_pledge.c:   * "unix", "dns".  SCM_RIGHTS requires "sendfd" or "recvfd".
uipc_socket.c:              mtod(control, struct cmsghdr *)->cmsg_type == SCM_RIGHTS)
uipc_socket.c:                      SCM_RIGHTS) {
uipc_socket.c:                          /* don't leak internalized SCM_RIGHTS msgs */
uipc_socket.c:                           * Dispose of any SCM_RIGHTS message that went
uipc_usrreq.c:   * This code only works because SCM_RIGHTS is the only supported
uipc_usrreq.c:  if (cm->cmsg_type != SCM_PAGES && cm->cmsg_type != SCM_RIGHTS)
uipc_usrreq.c:  if ((cm->cmsg_type != SCM_PAGES && cm->cmsg_type != SCM_RIGHTS) ||
uipc_usrreq.c:                              cm->cmsg_type != SCM_RIGHTS)
s4733163@comp3301:/usr/src/sys/kern$
```

This command **(-r)** recursively searched through the kernel files for instances where **SCM_RIGHTS** were referenced. Using **grep** allowed me to quickly identify the files and lines where **SCM_RIGHTS** was implemented. To further explore the code, I opened the file **uipc_socket.c** in **vim** editor and used the built-in search functionality to find all the instances of **SCM_RIGHTS** within the file .

```
void
unp_scan(struct mbuf *m0, void (*op)(struct fdpass *, int))
{
        struct mbuf *m;
        struct fdpass *rp;
        struct cmsghdr *cm;
        int qfds;

        while (m0) {
                for (m = m0; m; m = m->m_next) {
                        if (m->m_type == MT_CONTROL &&
                            m->m_len >= sizeof(*cm)) {
                                cm = mtod(m, struct cmsghdr *);
                                if (cm->cmsg_level != SOL_SOCKET ||
                                    cm->cmsg_type != SCM_RIGHTS)
                                        continue;
                                qfds = (cm->cmsg_len - CMSG_ALIGN(sizeof *cm))
                                    / sizeof(struct fdpass);
                                if (qfds > 0) {
                                        rp = (struct fdpass *)CMSG_DATA(cm);
                                        op(rp, qfds);
                                }
                                break;          /* XXX, but saves time */
                        }
                }
                m0 = m0->m_nextpkt;
        }
}

void
unp_discard(struct fdpass *rp, int nfds)
{
        struct unp_deferral *defer;

        /* copy the file pointers to a deferral structure */
        defer = malloc(sizeof(*defer) + sizeof(*rp) * nfds, M_TEMP, M_WAITOK);
        defer->ud_n = nfds;
        memcpy(&defer->ud_fp[0], rp, sizeof(*rp) * nfds);
        memset(rp, 0, sizeof(*rp) * nfds);
/SCM_RIGHTS
```

By typing **/SCM_RIGHTS**, I could jump to each occurrence of **SCM_RIGHTS** in the file , and pressing **n**, I navigated to the next instance of **SCM_RIGHTS**. From the results, I identified that **SCM_RIGHTS** were associated with the **sosend** function which is implemented in the **uipc_socket.c** . After reading through the code for the **sosend** function I found that sosend calls the **pru_send** function which in turn calls the **uipc_send** function which further calls the **unp_internalize** function which is in **uipc_usrreq.c**. The unp_internalize function handles the internalization of ancillary data which in the case of **SCM_RIGHTS** is file descriptors. In the **uipc_usrreq.c** file I repeated the process of using **vim's** search

functionality to find all the instances of **SCM_RIGHTS** to discover that there are three relevant functions that have instances of **SCM_RIGHTS** : "**unp_internalize**", "**unp_externalize**" and "**unp_scan**". After reviewing through the code for each function I realized that internalization and externalization of ancillary data is handled by **unp_internalize()** and **unp_externalize()** respectively.

FLOW CHART:

SECURITY RISKS

Risk Identification**:** While handling the case of MSB_SAME_PROT in the internalization process my code attempts to map memory pages one by one from the sender's virtual memory to the kernels virtual memory using uvm_share. To ensure that the pages within a single memshareblk are contiguous if at any point the mapping of a page fails, the entire block is remapped to ensure contiguous pages . However, during the process, the kernel memory may end up with some pages successfully mapped and some failed, but the already mapped in the case of a failure are not properly unmapped. This introduces a security risk because the already mapped pages in the kernel could potentially contain sensitive information, such as user data, confidential files, or other privileged information. Since the kernel's virtual memory can possibly be accessed by any process or thread within the system so if this memory is left mapped, this memory could possibly be accessed by unauthorized users or processes leading to data leakage or even unauthorized system control.

Exploitation of Risk: One of the ways the risk could be exploited is that improperly unmapped pages in kernel memory may contain some sensitive information that the sender could be sharing with the receiver . If any other process gains access to these pages it could access confidential data such as encryption keys, passwords, or sensitive files. A threat agent or adversary with local access to the system could trigger mapping failures and then could scan for residual data left in the unfreed memory . It violates the **confidentiality** aspect of the CIA triad which leads to compromising the security of the system. The threat agent can also repeatedly trigger memory allocation and failures leading to memory fragmentation leading to a DoS which violates the **availability** aspect of the CIA triad.

Mitigation Strategy: To mitigate this risk, I introduced a **free_memory**() function that ensures that any unmapped pages from the kernel's virtual memory in case of a failure. This function frees the memory region that was allocated but not successfully mapped preventing any potential leakage of data or misuse of kernel resources. This approach ensures that the kernel's virtual memory space remains clean with no leftover mappings that could be exploited. By implementing this I ensure that any sensitive data is removed and not left exposed reducing the risk of exploitation by an attacker.

CONCLUSION

In conclusion, the implementation of **SCM_PAGES** required a deep understanding of the kernel's internalization and externalization mechanisms, particularly through the study of **SCM_RIGHTS**. By thoroughly exploring the relevant functions, I was able to able to implement **SCM_PAGES** and also identify potential security vulnerabilities related to memory mapping in the kernel. Through this assignment, I gained valuable insights into modifying existing system calls, memory management, and security best practices, which are crucial for system-level programming.