On Inter-PMO Security Attacks [1]

Naveed Ul Mustafa
University of Central Florida

Yan Solihin
University of Central Florida

I. Motivation

Please click here for PDF of original paper. When integrating Persistent Memory (PM) to computer systems, one way to view PM is to host persistent data structures encapsulated in *objects*, referred as Persistent Memory Objects (PMOs) [4], [5], that are managed by the OS. A few existing studies address security threats arising from PMO model, i.e. using PM as system objects hosting persistent data. For example, Mustafa et al. [2] showcase *inter-process attacks* where one process (*payload*) successfully affects the execution of another process (*victim*) by overwriting pointers of a PMO shared between them. Such an attack requires the payload and the victim to share a common PMO (simultaneously or successively over time).

In this paper, we demonstrate that an adversary can launch successful attacks on a victim even when they *do not share* a PMO, *whether simultaneously or over time*. We refer to this new attack as an *inter-PMO* attack. The attack only requires a connectivity path of PMOs between processes including the payload and the victim, and exploits the path to propagate corruption. By relaxing the requirement of PMO sharing needed in the inter-process attack [2], the new attack significantly expands the capability of an adversary, and warrants the need to protect all PMOs irrespective of whether they are shared or not between the payload and the victim.

II. Background

PMO is a general system abstraction for holding persistent data managed by operating system (OS) without file-backing [5]. Data in a PMO is held in regular data structures. PMOs are managed by the OS which may provide filesystem-like namespace and permission settings to PMOs. Key primitives for a PMO are *attach()*, *detach()* and *psync()* system calls [4]. For a process to work on PMO data, it calls *attach()* system call to map the PMO into its address space. Once attached, the process can access it with regular loads/stores, without involving the OS. *psync()* persists PMO updates in a crash-consistent way. *detach()* unmaps the PMO from the address space, making it inaccessible. After detached, any load/store to the address region where the PMO used to map result in protection faults.

This work was supported in part by the U.S. Office of Naval Research through Grant N00014-23-1-2136 and Grant N00014-20-1-2750, and by the National Science Foundation through Grant 1900724 and Grant 2106629.

III. Threat Model

We considers a *victim* process lacking known memory safety vulnerabilities, a *payload* process having exploitable vulnerabilities, and *transmitter* process(es) that may or may not have vulnerabilities. In case of a single transmitter, it shares a PMO with the payload and a separate one with the victim. The goal of an adversary is to use the payload process in order to compromise the victim process. An attack can initiate by exploiting payload vulnerabilities, transmitting memory corruption over transmitter(s), and eventually impacting the victim's execution. This model differs from [2] in not mandating a shared PMO between the payload and victim. The adversary is assumed to possess knowledge of addresses, data structures, and layout of PMOs within the transmitter chain but lacks legitimate access to any of them. Data structures in PMOs may include buffers and pointers. A trusted system software, like the OS, enforces address space isolation between processes, and OS-managed permission checks restrict access to PMOs. Unauthorized access to a detached PMO leads to a segmentation fault. However, reading and writing to a legally attached PMO are permitted.

IV. Example Attack

A. Data Disclosure Attack By Hijacking Transmitters

Suppose that SQLite [3] is ported to PMOs with each table represented by a persistent B+ tree. Figure 1 shows a sample database consisting of PMO_0, PMO_1, and PMO_2 containing `faculty` and `prof` tables (B+ Trees rooted at *Src* and *Dst* PMO fields, respectively) of `CS`, `Phys`, and `Maths` departments, respectively. *Head* and *Tail* fields are pointers to linked-list of free nodes. P_0 is payload, P_1 is transmitter, and P_2 is victim process. The attack assumes transmitter P_1 has memory safety vulnerabilities.

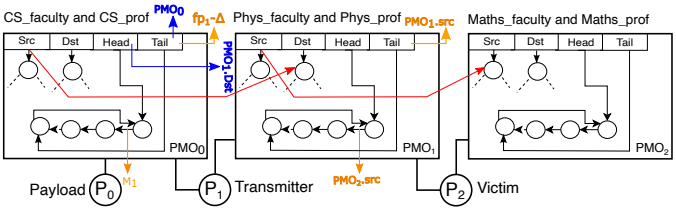

Fig. 1: Data disclosure attack. P_0 and P_2 share no PMO.

To launch the attack, adversary discovers a function pointer fp_1 in the volatile memory portion of P_1's address space and exploits P_1's memory vulnerabilities to inject a code block M_1, shown in Figure 2 (left), in its heap region. Note that Δ is

```
1 attach(PMO0);
2 attach(PMO1);
3 *(PMO0.Src)=*(PMO1.Dst);
4 node=*(PMO1.Head);
5 node->fd=&(PMO2.Src);
6 *(PMO1.Tail)=&(PMO1.Src)
      ;
```

```
1 void* allocNode(){
2 lastNode=*Tail;
3 firstNode=*Head;
4 lastNode->fd=
      firstNode->fd;
5 *Head=firstNode->fd;
6 return firstNode;}
```

Fig. 2: Injected code M_1 (left) and library code to allocate a node from free list (right).

address displacement between a free-list node and its *forward pointer* (fd) field. In the first attach-detach session, adversary uses payload process P_0 to attach PMO_0, overwrites the forward pointer *fd* of first node of its free list such that it points to M_1 and also overwrites the Tail field to point to location of fp_1 minus Δ (shown by orange arrows in Figure 1). Finally, adversary psyncs PMO_0, detaches it, and waits.

In the second attach-detach session, P_1 attaches PMO_0 and PMO_1 and allocates a node from free-list of PMO_0. The `allocNode()` library function (Figure 2, right) removes first node from the list. Since `Tail` was overwritten by adversary, `lastNode` points to $fp_1 - \Delta$ (line 2). The left side of the assignment statement in line 4, `lastNode ->fd`, points to a location pointed by `lastNode` plus address displacement between `lastNode` and its `fd` field i.e. $(fp - \Delta) + \Delta = fp$. Since `firstNode->fd` was set by adversary to point to M_1, line 4 makes fp point to M_1. Finally, when the function pointer is used by the P_1, M_1 is executed. Execution of M_1 (Figure 2, left) redirects $PMO_0.Src$ to root node of destination B+ tree of PMO_1 as shown by red arrow in Figure 1. M_1 also overwrites PMO_1's Tail field and *fd* pointer of first node in the free list, shown by orange arrows. Finally, M_1 psyncs PMO_0 and PMO_1, and detaches them. In the third attach-detach session, P_2 attaches PMO_1 and PMO_2, and allocates a free node from free-list of PMO_1 resulting in redirection of $PMO_1.Src$ to $PMO_2.Src$ shown by red arrow in Figure 1.

Consider that each process independently executes a query on an attached PMO to extract records from its faculty table (i.e., source B+ tree) with the designation of professor and insert them into professor table (i.e., destination B+ tree). Now assume following sequence of query execution. P_2 attaches PMO_1 and PMO_2, executes query on `Phys_faculty` table. Since, $PMO_1.Src$ was redirected, the query extracts records from `Maths_faculty` table (i.e. PMO_2) of victim and inserts them to `Phys_prof` table. Afterwards, P_2 psyncs and detaches both PMOs. Next, P_1 attaches PMO_0 and PMO_1, executes query on `CS_faculty` table that actually extracts records from `Phys_prof` table (including those records that were copied over from `Maths_faculty`) and insert them to `CS_prof` table, (as $PMO0.DSR\_Src$ was redirected). Finally, when P_1 psyncs and detaches PMO_0, process P_0 can attach PMO_0 to read records of `Maths_prof` inserted in `CS_prof`. The attack demonstrates that private data (i.e., records from PMO_2) of the victim P_2 is disclosed to attacker by payload P_0 process *even when they do not share a PMO*.

B. Data Disclosure Attack Without Hijacking Transmitters

We observe that above attack can be launched even without hijacking P_1. In such case, neither address discovery for function pointers nor code injection is needed. Though attack steps become more convoluted but not impossible. As an example, payload P_0 can attach PMO_0 and overwrite its Tail field with the address of PMO_0 and Head field with the address of $PMO_1.DST$, shown by blue arrows in Figure 1. Assuming that adversary knows address of PMO_1 and its layout, address of $PMO_1.DST$ is calculated as $address(PMO_1) + Size(SRC)$. Finally P_0 psyncs PMO_0, and detaches it. When P_1 attaches both PMO_0 and PMO_1, and allocates a node from free-list of PMO_0, it results in redirecting $PMO_0.SRC$ to root node of destination B+ tree of PMO_1, as shown by red arrow in Figure 1, achieving same affect as in first example attack. In the same way, payload can perform the second redirection shown in red in Figure 1 by carefully overwriting PMO_0 provided that attach-detach sessions are performed in desired sequence by the transmitter process P_1. Details of these steps are not shown in the figure due to limited space.

V. Attack Prototyping and Evaluation

We implemented a proof of concept inter-PMO attack illustrated in Figure 1, with two transmitters, on Greenspan PMO system [4] that was built on Linux 5.14.18 to support PMO creation and management. We consider an attack successful when P_0 can obtain a record of Math's professor. We define *time budget* as the duration within which an attack is attempted and *success rate* as the number of successful attacks divided by total number of attack attempts for a given time budget. We observe that the success rate is 1 for time budgets greater than or equal to 0.75 seconds and 0 otherwise. This shows that 0.75 second is the minimum time for the example attack to succeed. The attack fails for lower time budgets as the execution of queries by P_1, P_2 (transmitters) and P_3 (victim), and the propagation of results to PMO_0 (payload) takes at least 0.75 seconds.

References

[1] U.M. Naveed, S. Yan., "Persistent Memory Security Threats to Inter-Process Isolation," IEEE Micro. 2023; 43(5): 16-23.

[2] U.M. Naveed, X. Yuanchao, S. Xipeng, S. Yan., "Seeds of SEED: New Security Challenges for Persistent Memory," In IEEE International Symposium on Secure and Private Execution Environment Design (SEED) 2021 Sep 20 (pp. 83-88).

[3] Bhosale, S., Patil, T. & Patil, P. Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput*. 44, 882-885 (2015)

[4] G. Derrick, U.M. Naveed, K. Zoran, H. Mark, S. Yan., "Improving the Security and Programmability of Persistent Memory Objects," In IEEE International Symposium on Secure and Private Execution Environment Design (SEED), 2022 Sep 26 (pp. 157-168).

[5] X. Yuanchao, S. Yan, S. Xipeng, "MERR: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems 2020 Mar 9 (pp. 987-1000).