Security in Era of Persistent Memory [1]

Naveed Ul Mustafa
University of Central Florida

Yuanchao Xu, Xipeng Shen
North Carolina State University

Yan Solihin
University of Central Florida

I. Motivation

Please click here for PDF of original paper. Persistent memory (PM), compared to DRAM, provides higher density, better scaling prospect, non-volatility, and lower static power consumption, while providing byte addressability and access latencies that are not much slower. As a large-capacity memory alternative to DRAM, PM can be viewed as a fast medium for hosting a filesystem with memory-mapped files, an approach used by several works (e.g., PMFS [2] and BPFS [3]). Alternatively, PM can be used to host persistent data structures encapsulated in *objects* that are managed by the OS. Several recent works [4]–[7] manage PM using this relatively new abstraction of persistent memory objects (PMOs).

Several existing works address security threats from the use of PM as main memory fabric [8]–[11]. However, only two studies (at the time this work was published) address security threats from the PMO abstraction and its programming model. They propose reducing the exposure window of PMOs and PMO layout randomization [4], [7] to make it difficult for attacks to succeed. However, they did not analyze what attacks were possible and under what situations the protection could be effective. This paper discusses threat models and vulnerabilities that are either new or increased in severity by the use of PMOs. For example, we discuss that the use of a PMO may break the inter-process isolation guarantee that is central to OS security protection through address spaces. This paper also discusses the security implications of using PMOs, by presenting how threats are affected by the underlying assumptions and the programming model. It highlights sample attacks made possible by the vulnerabilities and identifies potential windows of opportunity to defend against them.

II. Background

A PMO is a general system abstraction for holding pointer-rich data structures without file backing [7]. PMOs are managed by the OS which may provide filesystem-like namespace and permission settings to PMOs. A user process invokes *attach()*/*detach()* systems calls to map/unmap a PMO to/from its address space. Once attached, the PMO data can be accessed by load/store instructions without OS intervention. Once detached, a PMO becomes inaccessible for the user process. A PMO can outlast the lifetime of a process, and can be attached by multiple processes over time. Simultaneous attaches by multiple readers are allowed but attach by a writer must be mutually exclusive with other readers and writers.

This work is supported in part by ONR through award N00014-20-1-2750 and NSF through award CNS-1717425.

III. Security Implications of PMOs

Persistency of PMOs and the programming model has several security implications. ① Since a PMO keeps persistent data, any data corruption or bugs (dangling pointers, memory leak, etc.) are also persistent. ② Unlike the volatile ones, the effect of corruption on PMO-resident data structures cannot be erased by relaunching the process as it may cause a recurrence of the incorrect process's behavior. ③ As PMO data is long-lived, it can be reused across runs of the same or different applications. Therefore, data corruption caused by one run directly affects the security of other runs or even unrelated applications. ④ Unlike DRAM data, the attacker can slowly and incrementally determine the target locations of data to corrupt over a long period of time across different runs. ⑤ Since they host pointer-rich data structures, PMOs become an attractive target for an adversary to manipulate pointers for successful security attacks. ⑥ As the data of an attached PMO is accessed via load/store instructions, these accesses are not trapped by OS and hence not checked for security.

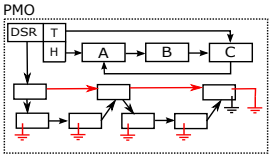
IV. Threat Model

We consider a threat model where the *payload* and *victim* processes share a PMO over time. While the *victim* has no known memory safety vulnerabilities, the *payload* does. The goal of an adversary is to use the payload process in order to compromise the victim process. We assume the adversary knows that a PMO is shared, knows the type of data structure, and knows the layout of the PMO. We assume a trusted system software such as the OS, which manages address space isolation between processes. PMOs are also managed by the OS which applies permission checking when granting access to a PMO. This implies that access to a detached PMO is not permitted and results in a segmentation fault. However, a process can read and write a legally attached PMO.

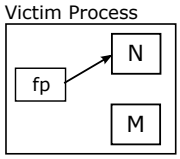
V. Attack Types

An adversary can potentially launch both control-data and non-control-data attacks on the victim by exploiting a PMO shared with the payload process. To save space, we only demonstrate an example control-data attack that alters a victim's control data (i.e., function pointer) in order to execute injected malicious code. Figure 1a shows a PMO hosting a skip list and a free list that manages deallocated nodes whereas the data structure root (DSR) field points to the start of the skip list. Nodes are allocated from the head of the free list. In Step 1 of the attack (Figure 1b), an adversary discovers a function pointer (fp) in the volatile memory portion of the

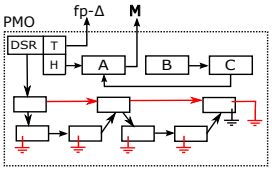
victim process's address space with the aim of redirecting it from code block N to M, where N is the legal block and M is an injected or out-of-context code block.



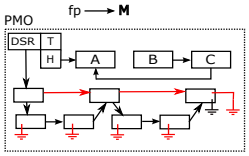
(a) PMO with skip list and free list.



(b) Step 1: Address discovery.



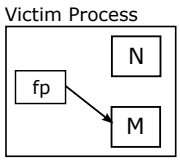
(c) Step 2: Exploit PMO.



(d) Step 3: Activation.

```
1 //C points to fp-Disp
2 last_node=*T;
3 first_node=*H;
4 //makes FP point to M
5 last_node->fd=
      first_node->fd;
6 *H=first_node->fd;
```

(e) Consolidation code.



(f) Step 4: Seize control.

Fig. 1: PMO-based cross-process/run pointer redirection attack.

In Step 2 (Figure 1c), the adversary uses a payload process to attach the PMO, by overwriting the forward pointer *fd* of the first node such that it points to M. Also, the tail pointer is overwritten to point to the location of fp minus a constant displacement Δ. The displacement is equal to the difference between the address of a node and its *fd* pointer. Then, the adversary persists the PMO, detaches it, and waits. When the victim attaches the same PMO and executes the free list consolidation code (Figure 1e) after allocating node A, it results in fp pointing to M (Step 3, Figure 1d). Finally, when the function pointer is used by the victim, the target code is executed (Step 4, Figure 1f), resulting in a successful attack altering the victim's execution flow. Note that despite not having an exploitable vulnerability, the victim process is successfully attacked because it shares the PMO with another process that does have an exploitable memory vulnerability. The time to carry out the attack can span multiple attach/detach sessions, and can even span across multiple process lifetimes, as long as fp and M remain constant over the span.

VI. Possible Defenses

Figure 2 shows the timeline of the steps of the above attack and opportunities for detecting and foiling the attack. First, the address of fp and M must remain the same between Step 1 and 3 for the attack to succeed. If the addresses change, the attack will corrupt the PMO but not result in

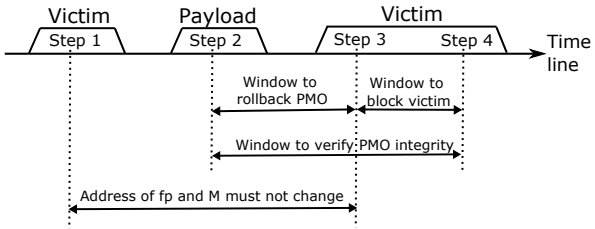


Fig. 2: Steps of cross-process/run PMO-based pointer redirection attack.

control flow hijacking. Second, the window of time between Step 2 and 4 is the window of opportunity to detect the attack by verifying the integrity of the data structures in the PMO. In other words, the integrity check must be performed before step 4 since the potential attacks get activated by that time. The integrity of the data structure(s) can be checked by performing topology verification. Third, in the window of time between Step 2 and 3, if a PMO integrity problem is detected, and a non-corrupted previous version is available, the PMO integrity can be restored and attack foiled. However, to foil the attack between Steps 3 and 4, the victim process must be blocked/terminated.

References

[1] 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).

[2] D. Subramanya R et al, "System software for persistent memory," In Proceedings of the Ninth European Conference on Computer Systems 2014 Apr 14 (pp. 1-15).

[3] C. Jeremy et al, "Better I/O through byte-addressable, persistent memory," In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles 2009 Oct 11 (pp. 133-146).

[4] X. Yuanchao, Y. ChenCheng, S. Yan, S. Xipeng, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) 2020 May 30 (pp. 680-692). IEEE.

[5] B. Daniel, A. Peter, M. Pankaj, L. Darrell DE, M. Ethan L, "Twizzler: a data-centric OS for non-volatile memory," ACM Transactions on Storage (TOS). 2021, Jun; 17(2):1-31.

[6] K. Awais, V. Sudharshan S, M. Jinsuk, O. Myeong-Hoon, K. Youngjae, "Persistent Memory Object Storage and Indexing for Scientific Computing," In IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC) 2020 (pp. 1-9). IEEE.

[7] 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).

[8] C. Siddhartha, S. Yan, "i-NVMM: A secure non-volatile main memory system with incremental encryption," In 2011 38th Annual international symposium on computer architecture (ISCA) 2011 Jun 4 (pp. 177-188). IEEE.

[9] Z. Pengfei, H. Yu, X. Yuan, "Supermem: Enabling application-transparent secure persistent memory with low overheads," In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture 2019 Oct 12 (pp. 479-492).

[10] Y. Mao, AZ. Kazi, M. Aziz, A. Amro, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," IEEE Transactions on Dependable and Secure Computing. 2019 Sep 13.

[11] M. Sparsh, IA. Ahmed, "A survey of techniques for improving security of non-volatile memories," Journal of Hardware and Systems Security. 2018 Jun;2(2):179-200.