Persistent Memory Security Threats to Inter-Process Isolation

Naveed Ul Mustafa
Department of Computer Science, University of Central Florida (UCF), FL, USA

Yan Solihin
Department of Computer Science, University of Central Florida (UCF), FL, USA

***Abstract*—**
Persistent Memory Object (PMO) is a general system abstraction for holding persistent data in persistent main memory, managed by an operating system. PMO programming model breaks inter-process isolation as it results in sharing of persistent data between two processes as they alternatively access the same PMO. In this paper, we discuss security implications of PMO model. We demonstrate that the model enables one process to affect execution of another process even without sharing a PMO over time. This allows an adversary to launch inter-PMO security attacks if two processes are linked via other unshared PMOs. We present formalization of inter-PMO attacks, their examples, and potential strategies to defend against them.

THE RELEASE of DIMM-compatible Intel Optane PMem in 2018 enabled the incorporation of Persistent Memory (PM) into main memory. Though Intel recently discontinued Optane PMem, alternative products (such as Kioxia FL6 [1] or CXL-attached SSD) are appearing. Compared to DRAM, PM 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. Consequently, PM blurs the boundary of memory and storage.

When integrating it to computer system, one way to view PM is as persistent main memory used to host persistent data structures encapsulated in *objects* that are managed by the OS. These Persistent Memory Objects (PMO) were proposed initially in [3] and used in recent works e.g., [4]. There are only a few studies addressing security threats arising from PMO model, i.e. using PM as system objects hosting persistent data. For example, [3], [4] look into reducing exposure window of PMOs and PMO space layout randomization. However, they did not analyze what threats were possible and the situations under which the protection could be effective. A recent work [5] elaborates on the security threats stemming from the PMO model by showcasing *inter-process attacks* where one process (*payload*) successfully affects the execution of another process (*victim*) by overwriting pointers of a PMO shared between them. Because of the long-living nature of a PMO, the attack does not require simultaneous PMO sharing; successive sharing over time (even across system boots) is sufficient.

The inter-process attack described in [5] still 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

 Published by the IEEE Computer Society

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 [5], 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.

This paper makes the following contributions. (1) We present inter-PMO attacks by defining them and analyzing necessary requirements for them to occur, (2) present two examples of inter-PMO data-disclosure attacks, (3) present a step-by-step process to implement an example in a controlled environment, (4) evaluate the success rate of the implemented attack, and (5) discuss potential defense strategies.

BACKGROUND

Persistent Memory Object (PMO)

PMO is a general system abstraction for holding persistent data managed by operating system (OS) [3]. PMO data is not backed by files, and may permanently reside in physical memory. Data in a PMO is held in regular data structures, hence may contain complex data types and pointers, and is accessible directly with load and store instructions, unlike files[1]. The OS may provide file system-like namespace and permission settings to PMOs so that data in a PMO can be reusable across process lifetimes and basic access control can be provided.

Key primitives for a PMO are *attach()*, *detach()* and *psync()* system calls [6]. As PMO already resides in physical memory, and its data is already in data structure form, 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 laod/store to the address region where the PMO used to map result in protection faults.

Just like a file, a PMO may outlast process lifetime, it is conceivable that it will be attached and accessed by multiple processes at different times (or simultaneously read). Similarly, a single process may attach and access multiple PMOs simultaneously. When multiple processes alternatingly access a file, a process may make changes to the file that affect other processes. The same can occur to a PMO accessed by multiple processes. However, a PMO is mapped directly to the address space, hence a change to PMO data by one process directly affects the address space of another process.

Security Implications of PM vs Files

PMOs are expected to hold data structures, making them pointer rich. On the contrary, files are normally used to hold data (but can also be used to hold data structures, though less common). Since a file contains no pointers, cross-process attacks are much harder to carry out. Pointers are attractive targets for attacks, which makes PM protection more important.

Also, data placed in a PMO-resident data structure is more tightly coupled with execution flow of a process as it can be accessed with regular load/store instructions. Even non-pointer data in PMOs is more likely to be directly used to determine program control flow, making them attractive attack targets. In contrast, data from files is first de-serialized and placed in data structures before used in a process' execution flow.

Finally, unlike PMOs, file data is managed directly by the OS, any access (read/write) requires system calls, and the OS can perform security checks when serving the system calls. In contrast, PMO data can be manipulated directly by loads/stores transparent to the OS.

The combination of longevity, direct byte-addressability and uncoordinated shared access distinguishes PMOs from both DRAM and traditional storage for memory protection, in terms of vulnerability, consequences of security breaches, as well as opportunities for novel solutions.

[1]While some files may contain serialized objects and hence pointers, accesses require system calls and serialization/deserialization.

PREVIOUS WORK

Most prior studies focused on the security vulnerabilities of PM fabric itself, rather than the PMO model. To address data remanence due to non-volatility, PM encryption was proposed, e.g. [7], [8]. The limited write endurance of PM may lead to early wear out if the attacker is allowed to write to them excessively. Hence, preventing redundant writes [10] and wear leveling are critical.

However, PM vulnerabilities go beyond just the fabric itself; hosting persistent data as in the PMO model, introduces new vulnerabilities while PM data is in active use. Memory Exposure Reduction and Randomization (MERR) [3] protects PMOs by reducing their exposure window (by attaching only when needed for access and detaching afterward) and hence the attack surface. They proposed splitting the page table to accelerate attach, and PMO Space Layout Randomization (PSLR), where the PMO is mapped to a different randomized location at each attach. Another work [4] focuses on mapping PMOs into separate domains, in order to leverage domain protection such as Intel Memory Protection Key (MPK). The intent was to restrict accesses to PMOs only to threads that access them. Both works [3], [4] seek to make *unauthorized accesses* to PMOs difficult *for the process accessing the PMO*.

A recent work by Mustafa et al. [5] showed that a vulnerable (i.e. *payload*) process with a PMO access can be used by an attacker to launch an attack on a different (i.e. *victim*) process that shares the same PMO successively over time. This paper exposes that such a requirement is indeed unnecessary, a security attack can be launched even when there is no shared PMO between the payload and victim.

THREAT MODEL

We consider a threat model where a process, referred as victim, has no known memory safety vulnerabilities that can be exploited by adversary. Another process, referred as *payload*, has memory safety vulnerabilities that the adversary can exploit. A third kind of process, referred as *transmitter*, shares a PMO with payload and a different one with the victim. Transmitters are not assumed to have memory safety vulnerabilities. An attack may start with the attacker exploiting some vulnerabilities in the payload, and transmit memory corruption over transmitters, to eventually affect the execution of the victim. This threat model is different from [5] where a shared PMO is required between the payload and victim.

The goal of an adversary is to use the payload process in order to compromise the victim process. We assume that adversary knows the addresses, data structures and layout of the PMOs that are part of the chain of transmitters but have no legit access to any of them. We assume data structures in PMOs may contain buffers and pointers and the payload process code may have regular known vulnerabilities (e.g., buffer-overflow, integer overflow, format string, etc.). We assume a trusted system software, such as the OS, which manages address space isolation between processes. PMOs are also managed by OS which applies permission checking while granting access to a PMO. This implies that access to a detached PMO is not permitted and results in segmentation fault. However, a process can read and write a legally attached PMO.

POINTER CLASSIFICATION

Based on the addressing mechanism, either absolute or relative pointers can be used to access PMOs and data-structures they hold. An absolute pointer contains virtual address, e.g. in Mnemsoyne [11] An absolute pointer is fast to dereference because it relies on the traditional address translation mechanism. However, it makes PMO Space Layout Randomization [3] costly; any time the PMO is mapped to a different virtual address region, pointers in the PMO must be rewritten accordingly. Finally, if multiple processes are allowed to simultaneously share a PMO, absolute pointers require the PMO to be mapped to the same virtual address range in all processes.

Alternatively, relative pointers can be used that combine PMO ID and offset in format of `object:offset`. A relative pointer can use a regular 64-bit format or use a fat pointer format where a pointer is represented by multiple fields. To dereference a pointer, a translation table is looked up to translate the system-wide unique PMO ID to its base virtual address [3], and then the offset is added to it. Unlike absolute pointers, relocating such PMOs is straightforward to perform. Note that example attacks we present

are valid irrespective of pointer type.

ATTACK EXPOSITION

To sketch an inter-PMO attack, the necessary condition is that the attacker can use the vulnerabilities of a payload process to affect the execution of the victim process through a series of PMOs and transmitters. In this section, we will define several terms and specify the necessary requirements for such attacks to succeed.

First, we define $S\_PMO(p)$ as the set of PMOs accessed by process p during its life time and $S\_Proc(x)$ as set of processes *sharing* a PMO x, with sharing defined as successively accessing a PMO during the PMO's lifetime. Note that the lifetime of p and x are different, with x's lifetime expected to be long. If there exists a PMO x that is shared by two different processes P_i and P_j, we state that there is a *link* between them via x.

$$\begin{aligned} &L(P_i, P_j, x) \Rightarrow \\ &\exists x \in S\_PMO(P_i) : P_j \in S\_Proc(x) \end{aligned} \tag{1}$$

Figure 1 shows an example of five processes shown in circles, sharing five PMOs shown in rectangles (right), with their respective S_PMO and S_Proc shown (left).

PMO	S_Proc(PMO)	Proc	S_PMO(Proc)
a	{P_0,P_1,P_2}	P_0	{a}
b	{P_1, P_3, P_4}	P_1	{a, b}
c	{P_2}	P_2	{a, c}
d	{P_3, P_4}	P_3	{b, d}
e	{P_4}	P_4	{b, d, e}

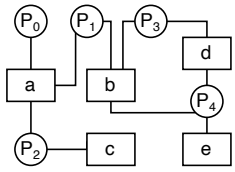

Figure 1: Example of S_PMO and S_Proc (left) and links between processes (right).

We define $Path$ between two different processes P_i and P_l as a set of links connecting them. Multiple paths might exist between two processes. For example, Figure 1 (right) shows a path between P_0 and P_2:

$$Path(P_0, P_2) = \{L(P_0, P_2, a)\}$$

and two paths between process P_0 and P_4:

$$\{L(P_0, P_1, a), (P_1, P_4, b)\}$$

and

$$\{L(P_0, P_1, a), L(P_1, P_3, b), L(P_3, P_4, d)\}$$

With $S\_PMO(p) \neq \emptyset$, there is always a path from a process p to itself.

To find a path between two processes P_i and P_l, we define a Set of Processes (SoP) initially with two members P_i and P_l. We state that a Path Exists (PE) between two processes P_i and P_l if they are the same process, linked by a PMO x, or there are distinct processes $P_j \notin SoP$ and $P_k \notin SoP$ such that P_i is linked to P_j via PMO y, P_k is linked to P_l via PMO z, and there exists a path between P_j and P_k.

$$\begin{aligned} &PE(P_i, P_l, SoP) \Rightarrow \\ &(P_i = P_l) \vee L(P_i, P_l, x) \vee \\ &(\exists P_j \notin SoP \wedge P_k \notin SoP : \\ &(L(P_i, P_j, y) \wedge L(P_k, P_l, z) \\ &\wedge PE(P_j, P_k, SoP \cup \{P_j, P_k\}))) \end{aligned} \tag{2}$$

Note that 2 is a recursive statement that resolves to *true* or *false*.

Conditions for Successful Attack

If pointers within a PMO attached by a process are not corrupted, reads/writes on that PMO are performed at addresses as *intended* by the attaching process. However, if the pointers are corrupted, reads/writes can be diverted to *unintended* addresses within the same PMO or a different but attached PMO, depending on corruption. We denote $P_i \xrightarrow[x]{r/w} y$ to state that P_i's read/write to PMO x are diverted to PMO y. Furthermore, we denote $x \mapsto x'$ to indicate that PMO x becomes x' if it has data or pointers that have been corrupted.

An adversary can potentially launch a successful attack on a victim process V by controlling payload process PL if ❶ a path exists between PL and V (i.e., $PE(PL, V, \{PL, V\})$ is true), ❷ sequence of attach-detach sessions on PMOs by processes along the path successively follows the direction of links in the path, and ❸ PL is able to mislead processes along the path to perform *unintended* reads and writes, with PL-determined data, on PMOs of their respective links.

The first condition requires that at least one path exists between the process PL and process V. If multiple path exists, the attacker has multiple options to attack V, with the shortest path potentially shortening the time to succeed. The sec-

ond condition imposes the order on attach-detach sessions performed by processes. For example, consider P_0 and P_4 of Figure 1 as payload and victim processes, respectively. For the following path from P_0 to P_4

$$\{L(P_0, P_1, a), L(P_1, P_3, b), L(P_3, P_4, d)\}$$

sequence of attach-detach sessions must be same as shown below for an attack to succeed.

$$1)\ P_0 \xrightarrow[a]{r/w} a \mapsto a' \quad 2)\ P_1 \xrightarrow[a']{r/w} b \mapsto b'$$

$$3)\ P_3 \xrightarrow[b']{r/w} d \mapsto d' \quad 4)\ P_4 \xrightarrow[d']{r/w} e \mapsto e'$$

where e is an unshared PMO of P_4. Note that we assume a process exclusively attaches PMOs in advance needed for both *intended* or *unintended* accesses. For example, for the second attach-detach session, P_1 attaches both a and b before reading/writing a.

The third condition requires that PL is able to mislead processes along the path to perform unintended writes e.g., by diverting their control flow when they access a corrupted PMO.

EXAMPLE ATTACKS

This section sketches two example proof-of-concept attacks, with the first requiring transmitter control flow hijacking while the second does not. The first example assumes that transmitter processes have memory vulnerabilities (e.g., buffer overflow) that can be exploited by an adversary. For both examples, consider SQLite [12], a database engine that allows multiple processes to read the database simultaneously but only one process can modify the database at any time. Suppose that SQLite is ported to PMOs for better performance, with each table represented by a persistent AVL tree. **Figure** 2 (left) shows a simple PMO that contains two B+ trees and a circular linked-list of free nodes. Two Data Structure Root (DSR) fields point to root node of the trees , while *Head* and *Tail* fields are pointers for the free list. Nodes are allocated/deallocated from/to the free list to perform insertion/deletion operation on trees. Figure 2 (right) shows library code to allocate a node from the free list. Suppose that DSRs, Head and Tail fields are contiguously laid out at fixed offsets from the PMO base address. These fixed offsets allow a program to locate the free list and the data structure.

We illustrate both data disclosure attacks by using PMO-ported SQLite in **Figure** 3.

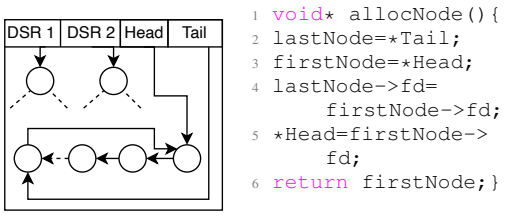

Figure 2: PMO Layout (left) and library code to allocate a node from free-list (right).

Data Disclosure Attack By Hijacking Transmitter Processes

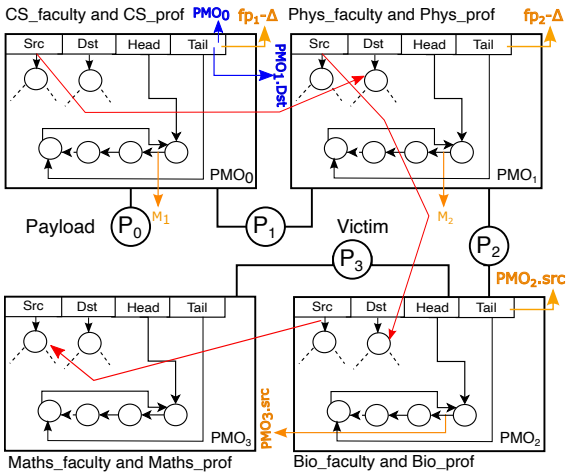

Figure 3: Setup for data disclosure attack. No PMO is shared between the payload and victim.

Let us assume that Source (Src) fields of PMO_0, PMO_1, PMO_2, and PMO_3 point to B+ trees representing `CS_faculty`, `Phys_faculty`, `Maths_faculty` and `Bio_faculty` tables, respectively. Destination (Dst) fields of PMO_0, PMO_1, PMO_2 and PMO_3 points to B+ trees representing `CS_profs`, `Phys_profs`, `Maths_profs` and `Bio_profs` tables respectively. PMO_3 is private to victim P_3 and we expect that payload process P_0 cannot read data from PMO_3. Note that following path exists between the payload and victim.

$$\{(P_0, P_1, PMO_0), (P_1, P_2, PMO_1),$$
$$(P_2, P3, PMO_2)\}$$

We demonstrate that payload can read from PMO_3 even though it is private to the victim and there is no shared PMO between the two processes.

The attack relies on following sequence of attach-detach sessions.

$$1)\ P_0 \xrightarrow[PMO_0]{r/w} PMO_0 \mapsto PMO_0'$$

$$2)\ P_1 \xrightarrow[PMO_0']{r/w} PMO_1 \mapsto PMO_1'$$

$$3)\ P_2 \xrightarrow[PMO_1']{r/w} PMO_2 \mapsto PMO_2'$$

$$4)\ P_3 \xrightarrow[PMO_2']{r/w} PMO_3 \mapsto PMO_3'$$

Note that first attach-detach session performs *intended* reads/writes on PMO_0 while others perform *unintended* reads/writes on PMO_1, PMO_2 and PMO_3.

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). To launch the attack, adversary first discovers function pointers fp_i in the volatile memory portion of P_i's address space and injects code blocks M_i, shown in **Algorithm** 1, in the heap region of P_i where $i = 1, 2$. Note that Δ is address displacement between a free-list node and its fd field.

Algorithm 1 Pseudocode of M_i for $i = 1, 2$

Require: addr(M$_{i+1}$), addr(fp$_{i+1}$), Δ

```
1: attach (PMO_{i-1}); attach(PMO_i);
2: *(PMO_{i-1}.DSR_Src)
       =*(PMO_i.DSR_Dst);
3: firstNode=*(PMO_i.Head);
4: firstNode->fd=&(M_{i+1});
5: *(PMO_i.Tail)=&(fp_{i+1})-Δ;
6: psync(PMO_{i-1}); psync(PMO_i);
7: detach(PMO_{i-1}); detach(PMO_i);
```

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 [2] (shown by orange arrows in Figure 3). 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 of Figure 2 removes first node from the list. Line 2 of the code gets `lastNode` by dereferencing `Tail`. Since `Tail` was overwritten by adversary, `lastNode` points to $fp_1 - \Delta$. 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 (see Algorithm 1) redirects $PMO_0.Src$ to root node of destination AVL tree of PMO_1 as shown by red arrow in Figure 3. 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 same way, third attach-detach session by P_2 redirects $PMO_1.Src$ to root node of destination B+ tree of PMO_2 and overwrites free-list pointers of PMO_2 while fourth attach-detach session by P_3 redirects $PMO_2.Src$ to root node of source B+ tree of PMO_3.

Now consider following sequence of query execution. P_3 attaches PMO_2 and PMO_3, executes query on `Bio_faculty` table. Since, $PMO_2.Src$ was redirected, the query extracts records from `Maths_faculty` table (i.e. PMO_3) of victim and inserts them to `Bio_prof` table. Afterwards, P_3 psyncs and detaches both PMOs. Next, P_2 attaches PMO_1 and PMO_2, executes query on `Phys_faculty` table that actually extracts records from `Bio_prof` table and inserts them to `Phys_prof` table (as $PMO1.Src$ was redirected) including those records that were copied over from `Maths_faculty` table. P_2 then psyncs and detaches both PMOs. Next, P_1 attaches PMO_0 and PMO_1, executes query on `CS_faculty` table that actually extracts

[2] Δ is 0 in our implementation of example attacks.

records from `Phys_prof` table 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 of the victim is disclosed to attacker by payload process even when they do not share a PMO.

Data Disclosure Attack Without Hijacking Transmitter Processes

The attack presented in previous section not only assumes memory vulnerabilities for transmitter processes, it also requires an adversary to find address of function pointers in volatile memory portion of processes' address space. The attack may not succeed either if a function pointer is not found for any one of P_1 or P_2, or if the address of function pointers or injected code blocks M_i change (e.g., due to relaunching of a process) anytime between address discovery and invocation of function pointers by processes. Furthermore, execution of code blocks injected in heap region of a process is possible only if Data Execution Prevention (DEP) is not supported on the platform. Otherwise, control flow of processes cannot be hijacked.

We observe that above attack can be launched even without hijacking P_1 and P_2. 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 3. 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, the `allocNode()` library function of Figure 2 dereferences `Tail` (line 2) to get `lastNode`. Since `Tail` was overwritten, `lastNode` actually points to $PMO_0.Src$. Line 3 of `allocNode()` dereferences `Head` to get `firstNode`. Since `Head` was overwritten, `firstNode` points to root node of destination B+ tree of PMO_1. With address displacement of zero between a node and its `fd` field, line 4 of `allocNode()` redirects $PMO_0.SRC$ to root node of destination B+ tree of PMO_1, as shown by red arrow in Figure 3, achieving same affect as in first example attack.

In the same way, payload can perform the remaining two pointers redirections shown in Figure 3 by carefully overwriting PMO_0 provided that attach-detach sessions are performed in desired sequence by other processes along the path between P_0 and P_3. Details of these pointer redirections are not shown in the figure due to limited space. Once all the pointer redirections are materialized, payload can read private data of the victim in the same way as shown in the previous section.

Attack Prototyping

We implemented a proof of concept inter-PMO attack illustrated in Figure 3. We implemented it on Greenspan PMO system [6] that was built on Linux 5.14.18 to support PMO creation and management. We built a simple persistent database modeled after SQLite consisting of the eight tables from Figure 3, i.e., two tables per PMO representing one of four departments (i.e., CS, Physics, Bio, and Maths). Our implementation of processes P_1, P_2, and P_3 execute queries to find records of professors from the *faculty_table* and insert them in the *prof_table* of respective departments. P_0 repeatedly reads all records from the *prof_table* of the CS department. All processes other than P_3 (i.e., *victim*) have buffer overflow vulnerability. We implemented *step 1* of the attack, i.e., stitching the path (shown by red arrows in Figure 3), by exploiting buffer overflow vulnerability to inject shell-code for M_1 and M_2 on the stack of P_1 and P_2, respectively. Note that this step can be completed in other ways (without code injection), e.g., by return-to-libc or return-oriented programming. In *step 2*, P_0 (i.e., *payload*) repeatedly runs the query to read records from the *prof_table* of the CS department until it finds a record of a professor from the Maths department (i.e., private PMO_3 of the victim) indicating a successful attack.

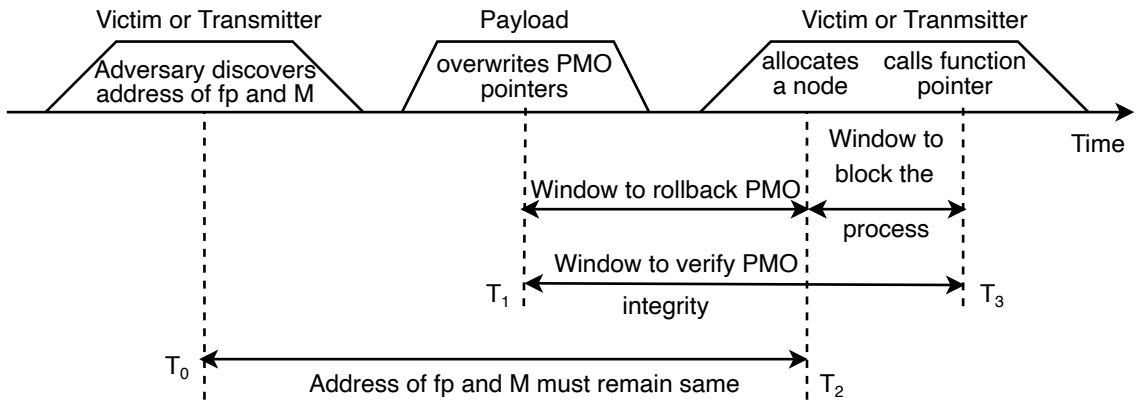

Figure 4: Steps to alter control flow of a process.

Evaluation

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 and P_3, and the propagation of results to PMO_0 takes at least 0.75 seconds. In a more realistic setting, with larger tables and processes also performing non-database accesses, the attack may take longer time to succeed.

POSSIBLE DEFENSE APPROACHES

Inter-PMO attacks based on hijacking of transmitter processes can be successful only if addresses of function pointers (fp_i) and injected code blocks (M_i) are not changed between two successive runs of the relevant process i.e. P_1, P_2 or P_3. If PMO Space Layout Randomization (PSLR) is enabled, the addresses of fp_i and M_i will be randomized on subsequent runs and hence the attack will fail to change the execution flow of transmitter processes and the victim. Furthermore, with M_i as code blocks injected by adversary in heap regions of processes, the attack can only be successful if Data Execution Prevention (DEP) is not supported or enabled.

However, DEP does not protect against all kind of attacks. For example, our example of inter-PMO attack *without hijacking* transmitter processes can succeed in presence of DEP as it does not rely on code injection. On the other hand, it can also fail if PSLR is enabled as address range at which PMOs are mapped in address space of a process is randomized on next attach call. However, some attack frameworks can breach PSLR or DEP defense schemes [9]. Therefore, additional protection is needed to detect and foil inter-PMO attacks.

Figure 4 shows the timeline of the steps performed by payload to alter control flow of a process as in data disclosure attack of Figure 3. The figure shows opportunities for detecting and foiling the attack.

First, address of fp and M must remain the same between T_0 and T_2, for altering control flow of the target process. If the addresses change, the attack will corrupt PMO but not result in control flow hijacking. Second, the window of time between T_1 and T_3 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 T_3 as the control flow gets altered by that time. The integrity of data structure(s) can be checked by performing topology verification and data structure-specific invariant checks. Third, in the window of time between T_1 and T_2, if PMO integrity problem is detected, and a non-corrupted previous version exists, the PMO can be restored and attack foiled. But, between T_2 and T_3, to foil the attack, the target process must be blocked/terminated.

ACKNOWLEDGMENT

This work was supported in part by ONR through grants N00014-23-1-2136 and N00014-20-1-2750, and by NSF through grants 1900724 and 2106629.

CONCLUSION

In this paper, we have shown that multiple processes may access persistent data in an uncoordinated way under PMO programming model. This allows a process to affect the execution of other processes even when they do not share a PMO. This makes PMOs a new tool for launching security attacks. We presented formalization of such attacks, demonstrated and evaluated example attacks, and provided discussion on possible defense approach. The paper makes the case for increased memory safety protection when persistent memory is used.

Naveed Ul Mustafa is a postdoctoral researcher at ARPERS lab, UCF, FL. Contact him at unknown.naveedulmustafa@ucf.edu.

Yan Solihin is Director for Cyber Security and Privacy Cluster and Professor of Computer Science at University of Central Florida. Contact him at yan.solihin@ucf.edu.

REFERENCES

1. Kioxia, SLC NAND Flash Memory Kioxia - United States. [Online]. Available: https://americas.kioxia.com/en-us/business/ssd/enterprise-ssd/fl6.html
2. SNIA, Persistent memory hardware threat model. Technical Whitepaper. [Online]. Available: https://www.snia.org/educational-library/persistent-memory-hardware-threat-model-technical-white-paper-2018
3. X. Yuanchao, Y. Solihin, X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," *Proc. Twenty-Fifth ASPLOS*, pp 987–1000, 2020.
4. X. Yuanchao, C. Ye, Y. Solihin, X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," *Proc. Forty-Seventh Ann. IEEE ISCA*, pp 680–692, 2020.
5. N.U. Mustafa, X. Yuanchao, X. Shen, Y. Solihin, "Seeds of SEED: New Security Challenges for Persistent Memory," *Proc. First IEEE SEED*, pp 83–88, 2021.
6. G. Derrick, N.U. Mustafa, Z. Kolega, M. Heinrich, Y. Solihin, "Improving the Security and Programmability of Persistent Memory Objects," *Proc. Second IEEE SEED*, pp 157–168, 2022.
7. Z. Pengfei, Y. Hua, Y. Xie, "SecPM: a Secure and Persistent Memory System for Non-volatile Memory.," *Proc. Tenth USENIX Workshop on Hot Topics in Storage and File Systems*, 2018.
8. Z. Pengfei, Y. Hua, Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," *Proc. Fifty-Second Ann. IEEE MICRO*, pp 479–492, 2019.
9. S. Kevin Z., F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, A.R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," *Proc. IEEE Symposium on Security and Privacy*, pp 574–588, 2013.
10. M. Sparsh, A.I. Alsalibi, "A survey of techniques for improving security of non-volatile memories," *Journal of Hardware and Systems Security.*, vol. 2, pp. 179–200, 2018.
11. V. Haris, A.J. Tack, M.M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News.*, vol 39, no. 1, pp. 91–104, 2011.
12. B.S. T., T. Patil, P. Patil, "Sqlite: Light database system," *Int. J. Comput. Sci. Mob. Comput.*, vol 44, no. 4, pp. 882–885, 2015.