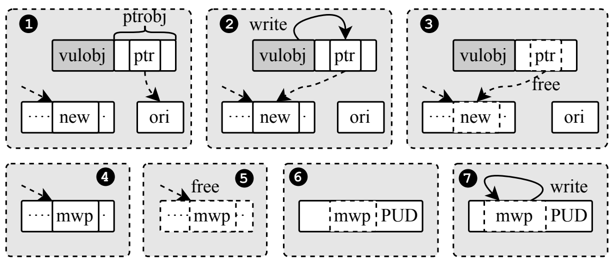
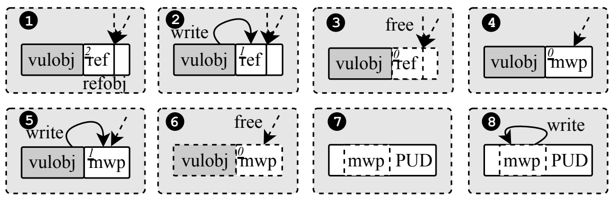


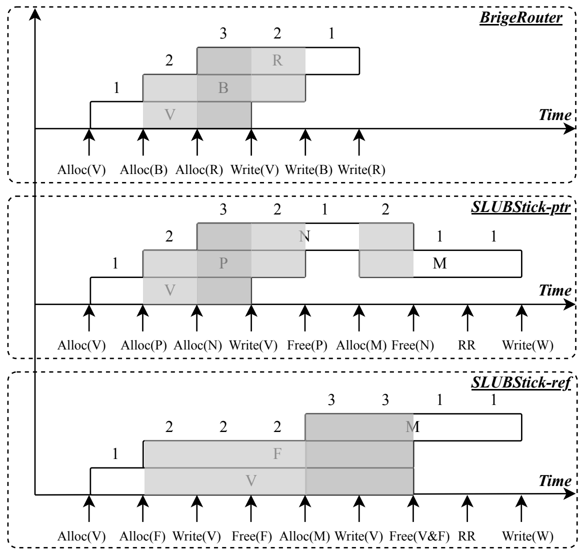
❶ ptrobj vulobj ptr new ori
❷ write vulobj ptr new ori
❸ vulobj ptr free new ori
❹ mwp
❺ free mwp
❻ mwp PUD
❼ write mwp PUD

(a) pivot through pointer (SLUBStick-ptr)



(b) pivot through reference counter (SLUBStick-ref)

Figure 1: Workflow of SLUBStick.



BrigeRouter
Time
Alloc(V) Alloc(B) Alloc(R) Write(V) Write(B) Write(R)
SLUBStick-ptr
Alloc(V) Alloc(P) Alloc(N) Write(V) Free(P) Alloc(M) Free(N) RR Write(W)
SLUBStick-ref
Alloc(V) Alloc(F) Write(V) Free(F) Alloc(M) Write(V) Free(V&F) RR Write(W)

Figure 2: Lifespan of critical objects during exploitation. V represents the $vulObj$, B represents the $bridge$, R represents the $router$, P represents $ptrObj$, N represents new, RF represents $refObj$, and M represents the mwp.

1. Attack Methodology of SLUBStick

Figure 1 illustrates the attack methodology of SLUBStick on OOB vulnerabilities. One of the key process is pivoting, which converts a given vulnerability to a dangling pointer that points to a memory-write-primitive object, enabling the adversary to write a controlled value at a chosen time. The pivoting can be achieved either through a pointer or a reference counter.

As illustrated in Figure 1a, pivoting through a pointer (SLUBStick-ptr) involves the following steps: ❶ allocate a pointer-containing object ($ptrObj$) and place it adjacent to the vulnerable object ($vulObj$); ❷ trigger the OOB vulnerability to overwrite $ptrObj \rightarrow ptr$, redirecting it from an original object (ori) to a new object (new), which is allocated in a memory page controlled by the attack (via heap spraying); ❸ release the memory area pointed to by $ptrObj \rightarrow ptr$, thereby freeing new; ❹ place a memory-write-primitive object (mwp) in the freed slot originally occupied by new; ❺ free all the objects in the page containing mwp, causing the page to be recycled and mwp to be illegally freed, thereby achieving a dangling pointer; ❻ reclaim the recycled page as a page upper directory (PUD) page through massive mapping of user-space pages; ❼ trigger the write operation on mwp through the dangling pointer to corrupt PUD entries, illegally mapping large physical memory regions to the user space, and finally achieving arbitrary memory writes through page table manipulation.

As illustrated in Figure 1b, pivoting through a reference counter (SLUBStick-ref) involves the following steps: ❶ allocate a refcount-containing object ($refObj$) and place it adjacent to the vulnerable object ($vulObj$); ❷ trigger the OOB vulnerability to overwrite $refObj \rightarrow ref$ to 1; ❸ unreference $refObj$ to reduce refcount to 0, thereby freeing $refObj$; ❹ place a memory-write-primitive object (mwp) in the freed slot originally occupied by $refObj$; ❺ trigger the OOB vulnerability again to overwrite $refObj \rightarrow ref$ to 1; ❻ unreference $refObj$ to reduce its refcount to 0, thereby freeing $refObj$ again, and then free all the objects in the page containing mwp, causing the page to be recycled and mwp to be illegally freed, thereby achieving a dangling pointer; ❼ reclaim the recycled page as a page upper directory (PUD) page through massive mapping of user-space pages; ❽ trigger the write operation on mwp through the dangling pointer to corrupt PUD entries, illegally mapping large physical memory regions to the user space, and finally achieving arbitrary memory writes through page table manipulation.

We use the exploitation of CVE-2022-0995 (Case 1 in §5.4) as an example to better illustrate how SLUBStick exploits an OOB vulnerability by pivoting through a reference counter. In this exploitation, the `watch_filter` object is used as $vulObj$, and the `anon_vma` object is used as $refObj$, while mwp is simulated. By allocating `anon_vma` adjacent to `watch_filter` in the SLUB allocator, SLUBStick triggers an OOB write vulnerability to tamper with $anon\_vma \rightarrow refcount$, setting it to 1. This allows premature deallocation of `anon_vma` when its reference counter reaches 0. mwp is then placed in the freed slot originally occupied by `anon_vma`. Re-triggering the OOB vulnerability resets $anon\_vma \rightarrow refcount$ to 1, enabling a secondary unreference to re-free the object. Concurrently, SLUBStick frees all objects in the page containing mwp, forcing page reclamation. The freed page is repurposed as a PUD page. Finally, the PUD entries are corrupted to enable arbitrary memory writes.

2. Analysis of Time Window

Figure 2 illustrates the correspondence between different types of objects mentioned in Figure 1 and their respective life cycles. For the sake of simplicity in our analysis, we assume that the time intervals between each object operation (including allocation, write, release, recycle & reclaim) are equal. The horizontal axis represents the progression of time, while the vertical axis indicates the number of concurrently active objects required at each timestamp. It is challenging to keep multiple objects alive simultaneously, and we highlight these intervals with darker shading. Therefore, the more dark-shaded blocks present during the exploitation process, the more difficult it is to successfully construct a time window that meets the required conditions. Based on the analysis, SLUBStick-ptr involves 9 dark-shaded regions, and SLUBStick-ref involves 12. In contrast, BRIDGEROUTER only has 7 dark-shaded regions. This indicates that BRIDGEROUTER is more likely to construct an appropriate time window during exploitation, making it easier to successfully exploit.