From Release to Rebirth: Exploiting Thanos Objects in Linux Kernel

Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang

***Abstract*— Vulnerability fixing is time-consuming, hence, not all of the discovered vulnerabilities can be fixed timely. In reality, developers prioritize vulnerability fixing based on exploitability. Large numbers of vulnerabilities are delayed to patch or even ignored as they are regarded as "unexploitable" or underestimated owing to the difficulty in exploiting the weak primitives. However, exploits may have been in the wild. In this paper, to exploit the weak primitives that traditional approaches fail to exploit, we propose a versatile exploitation strategy that can transform weak exploit primitives into strong exploit primitives. Based on a special object in the kernel named *Thanos object*, our approach can exploit a UAF vulnerability that does not have function pointer dereference and an OOB write vulnerability that has limited write length and value. Our approach overcomes the shortage that traditional exploitation strategies heavily rely on the capability of the vulnerability. To facilitate using Thanos objects, we devise a tool named** `TAODE` **to automatically search for eligible Thanos objects from the kernel. Then, it evaluates the usability of the identified Thanos objects by the complexity of the constraints. Finally, it pairs vulnerabilities with eligible Thanos objects. We have evaluated our approach with real-world kernels.** `TAODE` **successfully identified numerous Thanos objects from Linux. Using the identified Thanos objects, we proved the feasibility of our approach with 20 real-world vulnerabilities, most of which traditional techniques failed to exploit. Through the experiments, we find that in addition to exploiting weak primitives, our approach can sometimes bypass the kernel SMAP mechanism (CVE-2016-10150, CVE-2016-0728), better utilize the leaked heap pointer address (CVE-2022-25636), and even theoretically break certain vulnerability patches (e.g., double-free).**

***Index Terms*— Vulnerability exploitation, transfer weak primitives, kernel security.**

I. Introduction

SOFTWARE vulnerabilities cause severe consequences in the real world [1], [33]. Among them, kernel vulnerabilities have the biggest impact, which can cause privilege escalation, information leakage, etc. For example, Linux kernel has more than twenty million lines of code, and its complicated mechanisms and internal functions make vulnerabilities emerge consecutively. During the past 5 years, 1,306 vulnerabilities were discovered in Linux kernel [9].

Manuscript received 12 August 2022; revised 25 October 2022; accepted 22 November 2022. This work was supported in part by the National University of Defense Technology Research Project under Grant ZK20-17 and Grant ZK20-09, in part by the National Natural Science Foundation China under Grant 62272472 and Grant 61902412, and in part by the Hunan Province Natural Science Foundation under Grant 2021JJ40692. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. George Loukas. *(Corresponding author: Pengfei Wang.)*

The authors are with the National University of Defense Technology, Changsha 410073, China (e-mail: pfwang@nudt.edu.cn).

Digital Object Identifier 10.1109/TIFS.2022.3226906

Since fixing vulnerabilities is time-consuming, not all of the discovered vulnerabilities can be fixed timely. For example, the continuous fuzz testing platform szbot [11] has exposed more than 4,000 vulnerabilities in recent years, but nearly 1,000 vulnerabilities have not been fixed yet (up to Jan. 2022). As has been investigated in [42], it takes an average of 51 days to fix a bug (over 3,396 fixed bugs), whereas it takes less than 0.4 day for syzbot to report a new bug. Hence, the Linux community prioritizes bug fixing based on exploitability. Vulnerabilities that are regarded as unexploitable in practice would be delayed to patch or even ignored. According to CVEDetails' [10] statistics, only 9.5% of vulnerabilities in the last 20 years have been proved to be exploitable. For the rest, there is a huge time span from vulnerabilities being found to being fixed. However, exploits may have already been in the wild.

Security researchers determine a vulnerability's exploitability based on the exploit primitives. Exploitable vulnerabilities have strong primitives that can read or write arbitrary bytes to the desired location, while unexploitable vulnerabilities only have weak primitives that can only read or write limited bytes of data to unimportant data structures. This greatly increases the difficulty of writing payload into the kernel and hijacking kernel control-flow. However, such "unexploitable" vulnerabilities can become exploitable in the real-world. Under certain circumstances, it is possible to transform weak exploit primitives into strong exploit primitives.

In 2021, Nguyen [24] successfully exploited such a weak heap out-of-bounds write vulnerability (CVE-2021-22555) that can only write two NULL bytes to the adjacent object. Using a special vulnerable object (i.e., `msg_msg`) in the kernel, they can transform a weak OOB write into a strong exploit primitive and achieve privilege escalation. However, their approach is not universal. First, it is pretty difficult for people to find such a usable vulnerable object to realize a workable exploit. Specifically, `msg_msg` is only usable in the Linux kernel from v5.9 to v5.14, while in other kernel versions, `msg_msg` is not usable as it is put into kmalloc-cg-* cache which is isolated from common vulnerable objects. Second, exploiting such vulnerable objects is complicated. For example, different vulnerabilities may overwrite at different offsets and different caches, which needs different vulnerable objects to match. Third, other vulnerability types, such as UAF should also be included. Thus, to find more such vulnerable objects and use them properly, an universal approach that can identify them automatically, evaluate their usability, and pair them with suitable vulnerabilities, is in demand.

In this paper, we name the above mentioned vulnerable object as the *Thanos object* and propose an versatile strategy to transform weak exploit primitives into strong exploit primitives based on Thanos objects. Using the heap pointer in the Thanos object, we can control the release of the memory that the heap pointer points to. We leverage the heap-related **use-after-free (UAF)** vulnerability and the **slab out-of-bounds (OOB)** write vulnerability as two typical scenarios to illustrate the exploitation of Thanos objects. For a common unexploitable vulnerability with weak primitives, it can only write limited bytes of data at a fixed offset without other harmful behaviors. However, using a Thanos object that has a heap pointer at exactly the same offset, we can trigger the UAF write or the OOB write to make the heap pointer point to another heap chunk (which already has a pointer pointing to it). In this way, we create a vulnerable overlapped situation where two pointers point to the same chunk. Then, we construct two release paths to free the overlapped chunk twice and use a victim object and a spray object to take up the chunk, respectively. Using the traditional heap spraying technique [5], the spray object can write full length and arbitrary values to craft the victim object, leading to control-flow hijacking and privilege escalation. This transformation can break the limitation of write length and write value. To sum up, by using Thanos object to release an overlapped memory twice, we can maximize our write capacity and make rebirth come true.

However, to implement the above-mentioned approach, we have to overcome three challenges. **First, is to automatically identify Thanos objects from the Linux kernel**. Different vulnerabilities may write at different offsets. For example, some OOBs write the first few bytes of the adjacent object, while some UAFs write the middle bytes of the freed object. Thus, we should search for as many Thanos objects as possible to satisfy the needs of different vulnerabilities. **Second, is to evaluate the usability of the identified Thanos objects**. Different Thanos objects have different heap pointers, and the allocation paths, as well as the release paths, are also different. The higher complexity in exploiting a Thanos object, the lower usability it has. **Third, is to pair vulnerabilities with suitable Thanos objects**. The heap pointer of the Thanos object should be able to be overwritten by the vulnerability capability and we should pair the vulnerability with a high usability Thanos object based on their structure and characteristics.

To overcome the above challenges, in this work, we propose a general approach to automatically identify Thanos objects and leverage them to transform weak exploit primitives into strong exploit primitives. We develop a tool named `TAODE`, standing for **Th**A**nos **O**bject **D**iscov**E**ry, based on LLVM static analysis. First, it applies backward inter-procedural control-flow analysis and data-flow analysis to identify all Thanos objects in the kernel. Then, it collects relevant constraints to evaluate the usability of the identified Thanos objects. Finally, it pairs appropriate Thanos objects to corresponding kernel vulnerabilities. Using this tool, we show that Thanos objects are pervasive in the kernel (Linux, FreeBSD, XNU) and useful in real-world vulnerability exploitation.

In summary, this paper makes the following contributions.

- We present a versatile exploitation strategy using Thanos objects to transform weak exploit primitives into strong exploit primitives. Our approach can exploit a UAF vulnerability that does not have function pointer dereference and an OOB write vulnerability that has limited write length and value. Besides, our approach can sometimes bypass the kernel SMAP scheme by controlling more kernel space to place ROP chain, better utilize the leaked information (e.g., ordinary heap pointer), and even theoretically break certain vulnerability patches (e.g., double-free).
- We implement a tool named `TAODE` based on LLVM static analysis. It can automatically search for available Thanos objects in the kernel and pair vulnerabilities with suitable Thanos objects according to the usability.
- We demonstrate the ability of `TAODE` in searching Thanos objects from real kernels (Linux, FreeBSD, XNU). We also validate our exploitation strategy using 20 real-world vulnerabilities with the identified Thanos objects.

II. Background

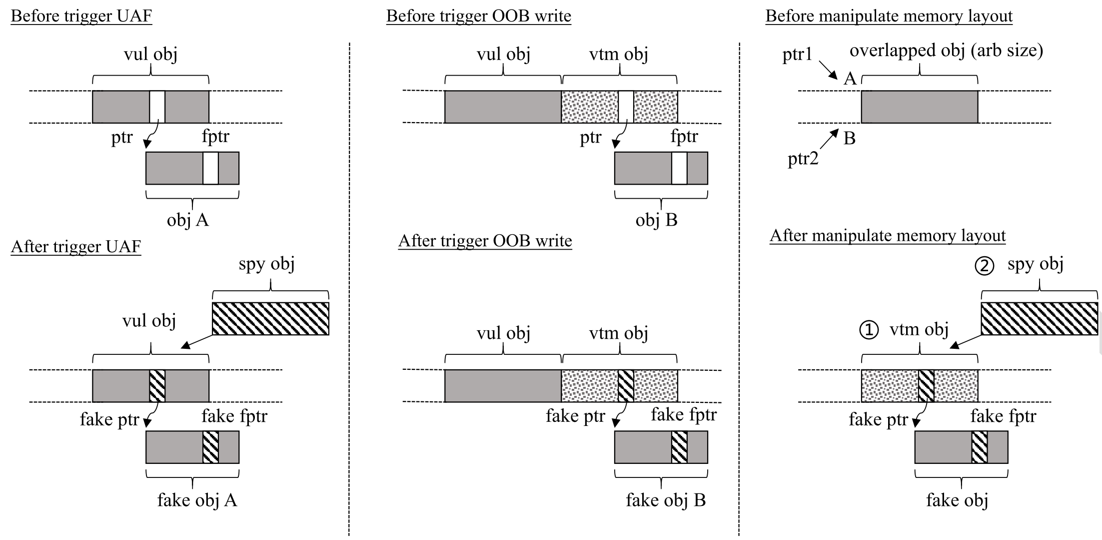
A. Kernel Memory Management

Linux kernel uses buddy system to manage physical memory pages. Buddy system allocates memory in units of page. However, most kernel structures need memory of less than one page. The slab allocator further divides a page into smaller objects, whose sizes are in units of bytes, like 8, 16, 32, etc. Basically, each slab cache is a linked list of slabs and each slab is an array of objects with similar sizes. Objects in the same slab cache are likely to be located in adjacent spaces. The heap spraying technique is exactly based on this principle. Objects in different slab caches are isolated in a sense, which means by leaking one slab's starting address, we cannot infer another different slab's starting address. When a vulnerable object locates in a slab cache that has less important data to corrupt, we can use the Thanos object to transform it into another cache that has abundant useful objects.

B. Weak Vs. Strong Exploit Primitive

Exploit primitives are machine states that violate security policies at various levels and indicate an attacker could get extra capabilities beyond the normal functionality provided by the original program [37], which is the foundation of generating an effective exploit. Exploit primitive includes read and write exploit primitive. Read primitive is used to leak key information, such as kernel function address and other useful pointers, and write primitive is used to hijack kernel control-flow or modify kernel credential.

1) Read Exploit Primitive: contains two characteristics. First, *is the number of bytes it can read*. If it can only read less than 4 bytes for one time or for several times in total, we regard it as a weak read primitive. As we know, at least 4 bytes of data are needed to bypass important mitigation in x86-64 kernel, like KASLR [23], for the higher 4 bytes of kernel address are fixed. Otherwise, if it can read arbitrary bytes of data as we control, we treat it as a strong read primitive. Second, *is the significance of the leaked data*. If the leaked data makes no sense (not secret information, like a cryptographic key) or does not contribute to mitigation bypassing or data crafting, it is treated as a weak read primitive. In contrast, if it can leak critical information, like function address and heap address, we treat it as a strong read primitive.



(a) Exploitation through UAF (b) Exploitation through OOB write (c) Versatile exploitation strategy

Fig. 1. Traditional exploitation techniques on UAF and OOB write, and a versatile exploitation strategy for both.

2) Write Exploit Primitive: contains three characteristics. First, *is the value it can write*. Sometimes, a vulnerability only allows writing NULL value or limited value, so we treat it as a weak write primitive. On the other hand, if it can write arbitrary value, we treat it as a strong write primitive. Second, *is the number of bytes it can write*. Writing more bytes is useful for placing malicious payload, like the ROP chain [36], an address sequence of code pieces to execute malicious code against the presence of executable space protection [35]. If it can only write less than 4 bytes of data for one time or for several times in total, we treat it as a weak write primitive. Third, *is the location it can write*. Writing important targets (such as function pointer, heap pointer, and kernel credential) can contribute to exploitation. Forging function pointer can help us to bypass mitigation mechanisms and hijack the control-flow, like `tty_operations->ioctl` and `tty_struct->ops`. Forging heap pointer can help us to place exploit payloads into memory or bypass some data checks in the execution path of the exp, like `msg_msg->next`. And forging kernel credentials can help us escalate privilege, like `cred->uid`. If it cannot overwrite these important data, we treat it as a weak write primitive. These important data are stored in kernel structures, which may locate in different caches.

In this paper, we focus on the write primitives as they are more harmful and can be easily turned into read primitives via an elastic object [4]. We use the above 3 write characteristics to judge whether a write primitive is weak or strong.

C. Traditional Exploitation Techniques

In this section, we use the UAF and OOB write vulnerabilities as examples to introduce traditional exploitation techniques and their limitations. In this paper, we do not focus on how to bypass kernel mitigation mechanisms, because there are many papers that have already proposed related solutions [6], [12], [13], [15], [39].

1) Exploitation Through UAF: As Fig. 1 (a) depicts, given a UAF vulnerability, we first find a function pointer `fptr` in the vulnerable object (i.e., `vul obj`, the object that is accessed after being released) or in an object `A` pointed to by a pointer `ptr` from the vulnerable object. Then we find an execution path that can dereference `fptr`. After the vulnerable object is released in the UAF, we use a spray object (`spy obj`) to overwrite the vulnerable object with crafted data, consequently, the function pointer `fptr` is tampered and points to malicious code. Finally, we hijack the control-flow by dereferencing the tampered function pointer.

If the vulnerable object in a UAF does not contain a function pointer or there is no execution path to dereference the function pointer, the UAF is regarded as having a weak exploit primitive. Since it cannot successfully tamper with the function pointer, the traditional exploitation technique is unworkable. Fig. 2 shows a UAF vulnerability (CVE-2021-26708) with a typical weak primitive that has limited write ability. After `virtio_transport_destruct()` has released the structure `virtio_vsock_sock` (line 4), function `virtio_transport_notify_buffer_size()` can still access this structure. Consequently, a UAF write occurs (line 12) when function `virtio_transport_notify_buffer_size()` writes to the freed object `vvs->buf_alloc`. However, the written value is checked to be no greater than 0xffffffff (line 9). Since `buf_alloc` is at offset 40 of structure `virtio_vsock_sock`, we can only write 4 bytes at offset 40 of the freed structure `virtio_vsock_sock`, which belongs to the kmalloc-64 slab. As structure `virtio_vsock_sock` does not have a function pointer, there is no function pointer dereference in any execution path. In summary, this UAF vulnerability can only write 4 bytes to the insignificant freed chunk and does not have function pointer dereference, so it is categorized as a weak exploit primitive. Moreover, in the kmalloc-64 slab that the vulnerable object belongs to, we cannot find both a suitable spray object and a victim object. Thus, traditional exploitation techniques fail to exploit this vulnerability.

2) Exploitation Through OOB Write: As depicted in Fig. 1 (b), given a vulnerability with OOB write, we first find a suitable victim object (`vtm obj`) that is located in the same

```c
void virtio_transport_destruct(struct vsock_sock *vsk){
 struct virtio_vsock_sock *vvs = vsk->trans;
 ...
 kfree(vvs);
}
void virtio_transport_notify_buffer_size(struct vsock_sock *vsk, u64 *val){
 struct virtio_vsock_sock *vvs = vsk->trans;
 ...
 if ( *val > VIRTIO_VSOCK_MAX_BUF_SIZE) // VIRTIO_VSOCK_MAX_BUF_SIZE == 0 xFFFFFFFFUL
   *val = VIRTIO_VSOCK_MAX_BUF_SIZE;
 ...
 vvs->buf_alloc = *val; // UAF write
 ...}
```

Fig. 2. CVE-2021-26708, a UAF vulnerability with a weak primitive that can only write 4 bytes.

```c
1 void xt_compat_target_from_user(struct xt_entry_target *t, void **dstptr, unsigned int *size){
2  const struct xt_target *target = t->u.kernel.target;
3  int pad;
4  ...
5  pad = XT_ALIGN(target->targetsize) - target->targetsize;
6  if (pad > 0)
7   memset(t->data + target->targetsize, 0, pad); // OOB write
8  ...}
```

Fig. 3. CVE-2021-22555, an OOB write vulnerability with a weak primitive that can only write two null bytes.

cache as the vulnerable object (`vul obj`), which is accessed out of bounds. The victim object must contain a function pointer or a data pointer `ptr` that points to an object `B` that contains a function pointer (`fptr`). Similarly, there should be an execution path that can dereference `fptr`. By elaborately manipulating the kernel memory layout, the victim object can be placed next to the vulnerable object. Then we trigger the OOB write to overwrite the victim object and tamper `fptr`. Finally, we hijack the control-flow by dereferencing the tampered function pointer `fptr`.

It is common to find that some OOBs can just write specific values or limited bytes. The former makes it unable to control the victim object's content, and the latter makes it hard to find a suitable victim object that contains a function pointer. These two weak exploit primitives make the traditional exploitation technique fail again. Fig. 3 shows an OOB vulnerability (CVE-2021-22555) with a weak primitive. In `xt_compat_target_from_user()`, user can control struct `target` in the kernel (line 2), then it calculates an alignment number `pad` at line 5. An OOB write occurs when filling `pad` NULL bytes at the end of the object (line 7). This vulnerability cannot write any significant data except two NULL bytes to the adjacent object. Thus, it is a weak exploit primitive that can't be exploited with traditional techniques.

3) A Versatile Exploitation Strategy: To sum up, if UAF has no function pointer dereference in the vulnerable object, or OOB has limited write value and write length, we treat them as weak exploit primitives. Traditional exploitation techniques cannot exploit these weak primitives to escalate privilege.

To overcome the limitations mentioned above, we propose to construct a versatile strong exploit primitive. As depicted in Fig. 1 (c), we first manipulate two pointers pointing to two objects (`A`, `B`) that are overlapped in the same memory space. Next, we release object `A` with one pointer and use a victim object (`vtm obj`) to take it up by memory re-allocation (see ①). The victim object should have a function pointer or a data pointer that points to another object that contains a function pointer (`fptr`). Then we release object `B` with the other pointer and use a spray object (`spy obj`) to overwrite the victim object with crafted data (see ②). The function pointer `fptr` in the victim object would be tampered to `fake fptr`. Finally, we dereference the tampered function pointer and hijack control-flow. In this strategy, we can decide the size of the two overlapped objects, thus we can choose a suitable victim object of any size we want. This makes up for UAF's lacking function dereference in the vulnerable object. Meanwhile, we can use heap spraying to craft a whole object with arbitrary value, which breaks the limitation of OOB's write value and write length.

It is very common that some exploit primitives can only write limited bytes of data to insignificant objects, namely weak primitives. The nature of kernel exploitation from vulnerability to privilege escalation is a process of transforming weak exploit primitives into strong exploit primitives. To realize the above-mentioned versatile exploitation strategy, a special object (we call it the *Thanos object*) plays a significant role. A Thanos object contains a heap pointer and a releasing path to release the memory pointed to by the heap pointer. By corrupting the heap pointer to point to another existing object, we can create a vulnerable overlapped state where two pointers point to the same object. In the following sections, we will introduce how we use Thanos objects to transform weak primitives into strong primitives with the examples of UAF and OOB write.

III. Transfer Weak Primitives to Strong Primitives via Thanos Objects

A. Thanos Object

To realize the versatile exploitation strategy, we need to use Thanos object in kernel to construct a vulnerable **overlapped state** that two pointers point to the same object, so that we can release two pointers, respectively, to tamper a function pointer by heap spraying, and finally hijack the control-flow. A Thanos object should meet the following requirements.

- **A heap pointer**. A Thanos object always contains a heap pointer, which is used to be overwritten to point to another existing object to form a vulnerable overlapped state.
- **An allocation path**. It is an execution path through which we can control the allocation of this Thanos object. If the exploit primitive is UAF write, we can allocate a Thanos object to take up the vulnerable object. If the exploit primitive is OOB write, we can allocate a Thanos object right after the vulnerable object. Since in the userspace we usually use a syscall to do the exploit, an allocation path should start from a syscall and ends with the allocation site of a Thanos object.

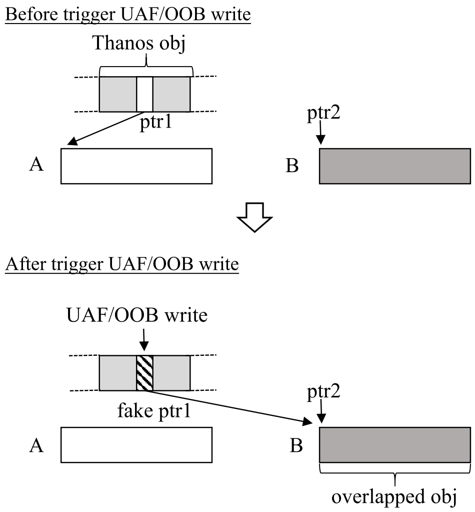


Fig. 4. Transform a weak exploit primitive into an overlapped state using a Thanos object.

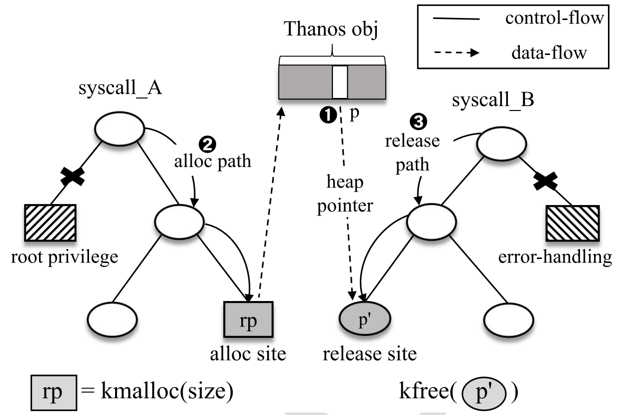


Fig. 5. The illustration of inter-procedural backward control-flow analysis and data-flow analysis. The kmalloc() and the free() are representatives of allocation and release functions (see Table I). The data-flow analysis starts from the return pointer (rp) of the allocation function and the release pointer (p'). We should avoid paths that require root privilege or pass an error-handling branch.

- **A release path**. It is a path that starts from a syscall to release the heap chunk pointed by the heap pointer in the Thanos object. Only by releasing the overlapped object twice with different pointers can we use a victim object and a spray object to take it up and hijack the control-flow.

B. Constructing Vulnerable Overlapped State

First, we assume the vulnerability can write at a specific offset of a freed object (in UAF) or an adjacent object (in OOB). As illustrated in Fig. 4, we find a Thanos object that is in the same cache as the vulnerable object. It owns a heap pointer `ptr1` at the offset that the vulnerability can overwrite. The heap pointer points to an object `A`. After that, we apply heap spraying techniques to let the Thanos object take up the vulnerable object in UAF or the adjacent object in OOB. Then we find another object `B` that is already pointed to by an existing pointer `ptr2` in the kernel. Finally, we trigger the UAF write or the OOB write to tamper `ptr1`, making it point to `B` as well. As a result, we succeed in constructing a vulnerable overlapped state where two pointers point to the same object. We can now perform the versatile exploitation strategy mentioned above to escalate privilege.

Both CVE-2021-26708 in Fig. 2 and CVE-2021-22555 in Fig. 3 can be exploited using a Thanos object. For CVE-2021-26708, we can transform a kmalloc-64 UAF into an overlapped state in kmalloc-4096, which would have both a useful victim object and a spray object to perform exploitation. For CVE-2021-22555, we can transform a limited OOB write into an overlapped state, which would have no limitation on write value and write length. This is because we can use a spray object to write arbitrary value and whole length to craft the victim object.

IV. Technical Approach

A. Identify Thanos Objects From the Kernel

Based on the requirements of the Thanos object, we first identify Thanos object candidates with heap pointers. Then we explore the allocation path starting from an allocation call site. Finally, we explore the release path starting from a release call site. The whole workflow is depicted in Fig. 5.

1) Identify Thanos Object Candidates: We mark kernel objects that contain heap pointers as Thanos object candidates. There are mainly two problems in identifying Thanos object candidates, recognizing heap pointers and nested structures. As our approach is based on LLVM intermediate representation (IR), specific pointer types are not clearly labeled. For example, there are several types of pointers, such as stack pointers, heap pointers, and function pointers. When we compile source code into LLVM IR, most definitions of pointers are indistinguishable, like `i8*`. Although some substructure pointers may have substructure name ahead, like `struct.msg_msgseg*`, indicating they are heap pointers, other pointers like `i8*` could be heap pointers too. We mark the objects as candidates as long as they contain pointers.

In the kernel, some objects may have nested structures. We concentrate on two types of nested structures. First, if a parent structure contains a substructure that has a heap pointer and they are in the same slab, we treat the parent structure as a Thanos object candidate; Second, if a parent structure contains a pointer that points to a substructure and the substructure contains a heap pointer, we treat the substructure as a Thanos object candidate. For the former, we can directly tamper with the heap pointer in the parent structure. However, for the latter, if we use the parent structure as a Thanos object, we have to first write the substructure pointer and then craft a fake substructure to tamper with the heap pointer. If an exploit primitive allows us to craft a structure, we may find another easier exploitation way. Thus, we suppose that a weak exploit primitive does not have such ability and we do not consider the parent structure of the latter case as a Thanos object.

2) Explore Allocation Path: To control the allocation of a Thanos object, we should explore its allocation path. As Fig. 5 illustrates, we first locate all allocation function call sites. There are two representatives of allocation functions on the heap, `kmalloc()` and `kmem_cache_alloc()` (Other allocation functions we used are listed in Table I).

TABLE I
THE ALLOCATION AND RELEASE FUNCTIONS WE USED IN LINUX KERNEL

Allocation	kmalloc(); kzalloc(); kcalloc(); kvzalloc(); kmalloc_node(); kzalloc_node(); kcalloc_node(); kmalloc_array(); kmalloc_array_node(); kmem_cache_alloc(); kmem_cache_zalloc(); kmem_cache_alloc_node()
Release	kvfree(); kfree(); kzfree(); kmem_cache_free()

The former allocates slab on the general cache while the latter allocates slab on the special cache. As has been stressed, the vulnerable object and the Thanos object should be on the same cache. Although most vulnerable objects are on the general cache, we still have to record all Thanos objects on the special cache because the kernel may call `find_mergeable()` to reduce memory fragmentation by merging objects. Notably, Thanos objects whose heap pointers point to special cache should be excluded.

Then, we perform backward inter-procedural control-flow analysis to explore the allocation path. We start from allocation function call sites and walk backward along the control-flow graph. If we can reach a syscall, it means we can use this syscall to control the object allocation. Meanwhile, we should ensure that this path does not require root privilege. Allocation function call sites that are not reachable from a syscall or require root privilege are excluded.

Finally, we perform forward inter-procedural data-flow analysis to obtain the object type we allocate. We start from the return pointer of the allocation function call sites and walk along the data-flow graph to collect the instructions that use the pointer and its alias as operands. We call these instructions **use points**. Some instructions like `getelementptr` and `bitcast` can reveal object types. The `getelementptr` instruction is used to get the address of a structure field member and perform address calculation but it does not access memory. The `bitcast` instruction is used to transform structure type. By recording the object type, the allocation function call site, the cache type, and the syscall, we can easily craft exploits. Objects that do not have a feasible allocation path will be excluded from Thanos object candidates.

3) Explore Release Path: To control the release of the memory pointed to by the heap pointer in a Thanos object, we should explore its release path. As Fig. 5 depictes, we first locate all release function call sites. There are two kinds of release functions, `kfree()` and `kmem_cache_free()`, which release slab on general cache and on special cache, respectively. Then we perform backward inter-procedural control-flow analysis to explore the release path. We start from a release function call site and check if we can reach a syscall. If we cannot reach a syscall or the release path requires root privilege, the release function call site is excluded.

Then, we perform backward inter-procedural data-flow analysis to figure out where the release pointer comes from (i.e., the source). If the release function is `kfree()`, the first parameter is the release pointer. We start from the release pointer and walk backward along the data-flow graph to record all potential sources. The following situations should be taken care of. (1) For a constant, a NULL pointer, a value from the `getelementptr` instruction, or a return value from an allocation function, we record it as a potential source. This is because these instructions might be the start points of a value. (2) For an instruction such as `phi`, `select`, `icmp`, binary operator, unary instruction, or call site to an ordinary function, we recursively traverse its operands to find the real sources. (3) For a formal argument or an instruction, like `bitcast` or `load`, we record it as a potential source and recursively traverse its pointer operand.

After collecting all potential sources of the release pointer, our next step is to determine that the release pointer is loaded from one Thanos object candidate. LLVM IR usually uses one `getelementptr` and one `load` instruction to acquire a field pointer from a structure. If we find a `getelementptr` instruction followed by a `load` instruction when traversing potential sources, we regard it as the real source of the release pointer and record the source structure and the offset of the field pointer. After filtering out Thanos object candidates that do not have release paths, we can finally record the detailed information of the remaining Thanos objects, including the release function call site, the syscall, the object type, the relative `getelementptr` instruction, and the offset of the field pointer.

Two issues should be resolved when identifying the release path. **Error-handling branches**. The kernel uses the error-handling branches to deal with errors, which may release the buffer, dump the error context, and return an error code. If the release path of a Thanos object passes an error-handling branch, then we cannot deterministically control the release anymore. This will make our exploitation unstable or even fail. We identify the error-handling branches by the branch label such as `error`, `exit`, and `fail` in a basic block, so that we can exclude them automatically when performing backward control-flow analysis.

Multiple release paths. When there is more than one path to release the same field pointer from the same Thanos object, we should track and record all release paths. As some release paths may implicitly pass error-handling branches, we would miss some true positives if we just track one release path.

B. Evaluate the Usability of Thanos Objects

We evaluate the usability of a Thanos Object by collecting the constraints of its field members. The higher complexity of the constraints, the lower usability of the Thanos object. When we tamper the heap pointer in a Thanos object, its adjacent field members can be overwritten as well, which can bring in side effect when releasing the heap pointer. We mainly focus on two field member types that tend to cause side effects.

1) Data Access: If the field member is a pointer, some instructions on the release path may read the content pointed to by it. If the member pointer is falsely overwritten to point to an invalid memory address, it can cause general page fault (GPF) or even kernel panic. Even if we tamper the field member to be a valid user space address, it can lead to a crash when accessing user space directly from kernel space because the kernel is acquiescently protected by the supervisor mode access prevention (SMAP) scheme [34]. In addition, the member pointer may also point to nested structures, which makes it harder to craft the data (discussed in Section VI-B.3).

2) Condition Check: If the field member is data, some instructions on the release path may check the field member to decide which branch to execute. If we falsely craft the field member, the kernel may choose the wrong branch and the expected release site will be missed. Then we cannot perform further exploitation. This type of field member could be a flag or a constant that indicates some kernel functionality.

Thus, it is necessary to collect all the data accesses and condition checks of the field members of a Thanos object on the release path to evaluate the complexity of the constraints. We perform forward data-flow analysis starting from Thanos objects to identify the constraints. For each field member, LLVM IR uses a `getelementptr` instruction and a `load` instruction to get it from a Thanos object. We can trace the data-flow to find all of its use points. There are four types of use points that we should further analyze.

- **Common instructions**, like `getelementptr`, binary operator, unary instruction, `select`, and `phi`, we recursively traverse their destination operands to find where they flow to.
- **Call instructions**, we follow up its callee function and analyze the corresponding formal argument to trace more use points.
- **Load instructions**, which loads a value from a pointer. We treat it as an **access point** if it is on the release path. If it is the first load instruction, it means it gets a field member directly from one Thanos object. Otherwise, it means there exist nested accesses (discussed in Section VI-B.3).
- **Compare instructions**, like `icmp`, we treat it as a **check point** if it is on the release path. If the first operand of `icmp` instruction originates from a Thanos object, we then perform backward data-flow analysis to find the source value of the second operand. Using the predicate and the source value, we can represent the constraint of the field member.

Finally, we use unified expressions to depict the constraints of a Thanos object, which is beneficial to pairing vulnerabilities with suitable Thanos objects. As Fig. 6 shows, there are mainly two expression types. First, if the field member is a kernel pointer and it does not appear in a compare instruction, we label it as an **access point** and then figure out if it points to nested structures. Only when there exists a nested access instruction exactly on the release path, can we label it pointing to nested structures. We use the expression $(off \mid kn)$ to represent such a constraint, where `off` denotes the offset of a field member in the Thanos object, and `kn` denotes that it is a kernel address and points to `n` layers of nested structures. Second, if the field member appears in a compare instruction, we use the expression $(off \mid range)$ to represent the constraint, where `range` denotes the range that the field member has to satisfy to reach the release site. For example, "[0, 8) == NULL" means that the first 8 bytes of a Thanos object should equal NULL. If the field member has a specific range, it would be easy for us to craft. However, if the field member is a kernel pointer, we should place a valid kernel address or even craft the memory area pointed to by the pointer, which is more difficult.

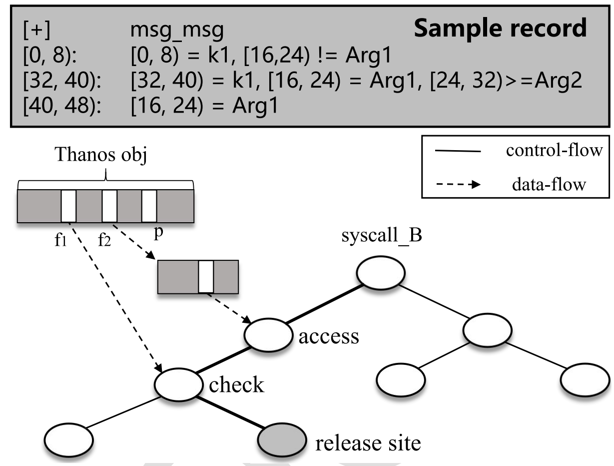


Fig. 6. Identifying filed member constraints. f1 denotes the field member flows into a compare instruction (condition check). f2 denotes the field member points to a substructure and it is accessed on the release path (data access).

C. Pairing Vulnerabilities With Thanos Objects

To pair the vulnerabilities with usable Thanos objects, we should extract the capability of the vulnerability. Recall that our target vulnerabilities are UAF which has no function pointer dereference and OOB write which has limited write length or write value, so we focus on the write capability of UAF and OOB.

First, we figure out which cache type the vulnerable object belongs to by pinpointing the allocation site of the vulnerable object. This is important because the Thanos object can be overlapped with or adjacent to the vulnerable object only if they are in the same cache. Then we debug the vulnerability to analyze its write capability when triggering the vulnerability. There are three factors that should be considered: (1) the offset where it can write in the vulnerable object (UAF) or the adjacent object (OOB), (2) the write length, and (3) the write value (i.e., arbitrary or limited value). We use a formal expression $(VCache, [(off_1, len_1, val_1), \ldots, (off_n, len_n, val_n)])$ to represent the write capability, where $VCache$ indicates the cache type, off_i, len_i and val_i represent the write offset, the write length, and the write value, respectively. For example, the write capability of CVE-2021-26708 can be represented as $(kmalloc-64, (40, 4, arb))$, which indicates it can write 4 arbitrary bytes at offset 40 of a slab from kmalloc-64 cache. The write capability of CVE-2021-22555 can be represented as $(kmalloc-4096, (0, 2, NULL))$, which indicates it can write 2 NULL bytes at the front of a slab from kmalloc-4096 cache. Notably, one vulnerability may have several write offsets.

With the expression of Thanos objects and vulnerabilities presented above, we can pair vulnerabilities with Thanos objects. Given a vulnerability, we first filter out Thanos objects that do not share the same cache with the vulnerable object. Then we check the write capability of the vulnerability to find whether it can overwrite the heap pointer of the remaining Thanos objects based on the expressions. This can further narrow down the Thanos objects useful for exploitation. Finally, we check if the vulnerability will bring side effects when overwriting the field members of the Thanos objects.

This can provide supplemental information for evaluating the complexity of exploitation.

There are two situations that we should pay attention to. First, if the vulnerability can only write limited value like NULL bytes, we should ensure that it can overwrite just one or two bytes of the 8-byte target heap pointer. Recall that the key of our exploitation approach is to tamper with the heap pointer to point to another overlapped object. By spraying many objects in the kernel memory and changing just one or two bytes of the heap pointer (the least significant bytes at best), we can make the heap pointer point to a certain object by chance and then perform further exploitation. In practice, by elaborately arranging memory layout, this chance is acceptable. However, if the vulnerability destroys more than two bytes of the heap pointer, the chance of hitting another object will be very low. Because the kernel heap address is not predictable and we cannot write an arbitrary value to craft the heap pointer. Second, if the vulnerability can write an arbitrary value, we can first leak the address of the overlapped object to make the exploitation deterministic. There are several approaches to leaking the address. For example, we can use other information leak vulnerabilities or elastic objects [4]. Sometimes a kernel warning can reveal kernel addresses, too. However, this is out of the research of this paper.

We design an automated algorithm to pair a kernel vulnerability with suitable Thanos objects. As Algorithm 1 shows, the algorithm inputs include the cache name of the vulnerable object ($VCache$), the capability of the vulnerability (Cap), and the set of all Thanos objects (S_{Tha}). The output is a set of the matched Thanos objects. First, we filter out the objects which are not in $VCache$ (Line 3). Then we traverse the vulnerability capability (Line 5) and the offset set of heap pointers in one Thanos object (Line 6). If the write value is a limited value (not a heap pointer) and the write length is more than two bytes (the least two bytes), then we skip this heap pointer of the Thanos object (Line 7-9). Otherwise, if it can write the least bytes of the heap pointer (including arbitrary value and limited value), we add the Thanos object into S (Line 10-11).

Algorithm 1 Pairing Vulnerability With Thanos Object

Input: $VCache$: The cache of the vulnerable object;
Cap: Capability set of 3-tuple <off, len, val>;
S_{Tha}: Set of all Thanos objects
Output: S: Set of the matched Thanos objects
1: Procedure MATCHVULTHA($VCache$, Cap, S_{Tha})
2: S = ∅
3: for all ThaO $r1$ using $VCache$ in S_{Tha} do
4: A_p = heap pointer offset in $r1$
5: for (off_v, len_v, val_v) in Cap do
6: for off_t in A_p do
7: if (val_v is limited) && (val_v is not hptr) then
8: if ($off_v \leq off_t$) && (off_t+2 $\leq off_v$+len_v) then
9: continue
10: if ($off_v \leq off_t$) && ($off_t \leq off_v$+len_v) then
11: $S = S \cup r1$
12: return S

V. Implementation

To realize the approach mentioned above, we implemented a static analysis tool named TAODE. As our static analysis is based on LLVM IR, we should first compile the kernel source code into LLVM bitcode files. Then, we perform inter-procedural control-flow analysis and data-flow analysis on the generated LLVM IR. During the initializing stage, we apply two-layer type analysis from [20] and [21] to construct a field-sensitive call graph and the build-in AliasAnalysis pass of LLVM to perform alias analysis. In the following, we present some implementation issues and solutions.

A. Privilege Check on the Path

Since our exploitation strategy requires normal privilege, we should ensure that the allocation path and the release path do not require root privilege. Linux kernel uses `capable()` to check the process credentials and decide whether the process has the privilege to execute this path. If its parameter is `CAP_SYS_ADMIN`, it means requiring root privilege. When we perform backward control-flow analysis, we also check if the path passes `capable(CAP_SYS_ADMIN)`. For other parameters, like `CAP_NET_ADMIN`, we don't exclude relevant paths as it is useful for exploitation if we can control a privileged container.

B. Special Cache Type

In the kernel, there are special slabs that are dedicated for specific objects (e.g.,`fuse_file`). If the heap pointer of the Thanos object points to such special slabs, it would be difficult to find suitable victim objects and spray objects to proceed with the exploitation. Thus, we should exclude such Thanos object candidates with special slabs. To have the overlapped object released into a general cache, we must make sure that the heap pointer points to a general slab. TAODE records the release sites of all potential Thanos objects to identify the one with special slabs (i.e., released by `kmem_cache_free()`) and exclude them.

VI. Evaluation

In this section, we conduct experiments to validate our versatile exploitation strategy proposed in this paper, aiming to answer the following research questions: **RQ1**: Can TAODE effectively identify Thanos objects from real-world OSes? **RQ2**: Are the identified Thanos objects usable in exploiting real-world vulnerabilities with the versatile strategy? **RQ3**: Does our exploitation strategy have any other side effects in kernel exploitation?

A. Experiment Setup

1) Setup: All experiments are conducted in an Ubuntu-18.04 system running on a desktop with 128G RAM and Intel(R) Core i9-10900KF CPU @ 3.70GHz. Our TAODE is based on LLVM-10.0.0 and we use Clang-10.0.0 to compile Linux kernel-v5.3 into LLVM IR. Then TAODE can perform static analysis on the generated LLVM IR. To test real-world kernel vulnerabilities, we install QEMU-4.2.1 on Ubuntu.

2) Dataset: TAODE is evaluated using kernels including Linux 5.3, FreeBSD 12.1, XNU 10.15. We also evaluate our exploitation approach against 20 kernel vulnerabilities (9 UAF writes and 11 OOB writes) that have weak exploit primitives. Among them, 14 are associated with CVE IDs and the rest without CVE IDs are collected from syzbot [11]. As is depicted in Table IV, we summarized their limited write capabilities. The weakest primitive can only write one NULL byte at the front of the adjacent slab.

3) Mitigation Setting: To be close to real-world exploitation, we set up four common mitigation mechanisms for the kernel. We enabled KASLR [23], which loads the kernel to a random location in memory. We enabled SMEP [14] and SMAP [34] protection to prevent direct userspace access in kernel execution. We enabled KPTI [8] to prevent it from CPU side-channel attack. These four mitigation mechanisms are the fundamental configurations of recent major Linux release versions. If a generated exploit can hijack kernel control-flow and bypass these four mitigation mechanisms, we consider that it can perform successful exploitation.

4) Info-Leak Setting: As we mentioned in the Section IV-C, sometimes we have to know the address of the target overlapped object first, so as to forge the heap pointer of the Thanos object during exploitation. For there are existing approaches to perform info-leak and it is out of our research, we write a vulnerable driver to simulate an info-leak vulnerability or other info-leak techniques. The vulnerable driver can allocate, read and release a heap chunk. As the read size is not checked, we can perform an out-of-bounds read to leak the kernel address. This module is automatically loaded with the vulnerable kernel.

B. Thanos Object Identification

1) Overall Results: We first use TAODE to analyze the Linux kernel. We analyzed 17,554 bitcode files with 76,670 structures in Linux kernel and finally determine 63 potential Thanos objects. The analysis took 21 hours. Then, we analyze these objects manually and confirm 49 as true positives (listed in Table III). The false positives are nearly 22% (i.e., 14 false positives), which is acceptable for a static analysis approach.

To demonstrate the pervasiveness of Thanos objects, We also analyzed FreeBSD and XNU with TAODE. The overall results are depicted in Table V. It took 8 hours to analyze FreeBSD and finally 76 Thanos objects were found, with 61 confirmed. Since only a small portion of XNU's source code is available, it just took 2 hours to analyze XNU and 52 Thanos objects were found with 34 confirmed. The results indicate that Thanos objects are also pervasive in FreeBSD and XNU, and TAODE is effective in identifying Thanos objects in other OSes. TAODE needs minor modification (e.g., allocation and release APIs) to adapt to different OSes. Since it is difficult to find suitable vulnerabilities to validate the Thanos objects from other OSes, in the following analysis, we concentrate on the results of Linux. Detailed information on Thanos objects from other OSes is available with our released project.

2) Detailed Results: We list all the Thanos objects that we identify and confirm from Linux in Table III. The results in Table III (from the column on the left to the right) indicate (1) the caches to which a Thanos object belongs, (2) the structure type of a Thanos object, (3) the offset of the target heap pointer in a Thanos object, (4) the constraints that an adversary has to satisfy to successfully release the overlapped object pointed to by the heap pointer.

TABLE II

OVERALL RESULTS OF THANOS OBJECT IDENTIFICATION

Kernel	Files #	Total structures #	Time	Thanos objects #
Linux	17,544	76,670	21h	49
FreeBSD	5,896	52,867	8h	61
XNU	1,484	3,897	2h	34

Based on the observation of the results, we find that the identified Thanos objects cover most of the general caches and some special caches (e.g., rsb_cache). In the "cache" column, * denotes the size of the cache can be equal to or more than this number, which means these objects could belong to all the general caches equal to or greater than they are specified in the table. These size-alterable Thanos objects (12 out of 49) could significantly enrich our object choices during exploitation. In the "offset" column, we can see that some objects have multiple heap pointers, which can be used in the vulnerabilities that have different write capabilities. The two characteristics discussed above could potentially improve the exploitability of a vulnerability. In the last column of Table III, we specify the constraint set based on the data accesses and condition checks on the release path. To release the overlapped object successfully, we should ensure the field members satisfy the relevant constraints. Notably, some objects have no constraint (i.e., ∅), which means they are easy to craft during exploitation. The majority of constraints come from data accesses, so the relevant field members must point to proper memory to avoid access errors on the release paths.

3) False Reports: As a static approach, our approach inevitably introduces false positives and false negatives. The disposal of the following situations in TAODE can incur false reports.

a) Nested structures: When we perform backward data-flow analysis from the release pointer, one release pointer may originate from several object types. For example, `struct x509_certificate-> struct public_key_signature*sig->u8*digest`. The release pointer `*digest` belongs to structure `public_key_signature`, meanwhile, its structure pointer `*sig` also belongs to structure `x509_certificate`. Thus, we find two source object from the release pointer `*digest`. However, in this case, we would ignore the middle structure `x509_certificate` which is too complicated to craft this structure under the circumstance of a weak exploit primitive. Consequently, such simplification might cause false negatives.

b) Nested accesses: When there are nested accesses through the heap pointer on the release path of a Thanos object, we regard such Thanos objects as too complicated to exploit. This is because if we tamper with this heap pointer to point to the overlapped object, we must first elaborately craft valid data on the overlapped object. However, it is too difficult to craft complicated data (e.g., a valid pointer pointing to a substructure) except for constants on the overlapped object in advance, so the release path may trigger page fault and

TABLE III

THANOS OBJECTS IDENTIFIED AND CONFIRMED IN LINUX. IN THE "CACHE" COLUMN, * DENOTES THE SIZE OF THE CACHE CAN BE EQUAL OR MORE THAN THIS NUMBER. IN THE "CONSTRAINTS" COLUMN, ∅ DENOTES DATA DISCLOSURE IMPOSES NO CRITICAL CONSTRAINTS. *Arg* REPRESENTS A SYSTEM CALL ARGUMENT UNDER A USER'S CONTROL. *kn* STANDS FOR A KERNEL ADDRESS WHICH POINTS TO N LAYERS OF NESTED STRUCTURES

Cache	Struct	Offset	Constraints
kmalloc-16	cond_expr	8	[8, 16) == k1
	map_iter	0	∅
kmalloc-16*	cfg80211_nan_func_filter	0	∅
kmalloc-32	static_key_mod	0	[0, 8) == k1, [16, 24) == Arg
	perf_domain	8	[8, 16) == k1
	ip_sf_list	0	[0, 8) == k1, [8, 16) == 1, [16, 24) == 1
	role_trans	16	[16, 24) == k1
	nfs4_label	16	∅
	workqueue_attrs	8	∅
kmalloc-32*	pneigh_entry	0	[0, 8) == k1, [16, 24) == 0
	jffs2_full_dirent	8	[8, 16) == k1, [20, 24) $\geq$ 0xffff
	simple_xattr	16	[0, 8) == k1, [8, 16) == k1
kmalloc-64	ip_vs_sync_buff	24	[0, 8) == k1, [8, 16) == k1
	ip6_sf_list	0	[0, 8) == k1, [24, 32) == 1, [32, 40) == 1
	tipc_peer	8	[32, 40) == k1, [40, 48) == k1
	cond_node	8, 16, 24	[8, 16) == k1, [16, 24) == k1, [24, 32) == k1, [32, 40) == k2
	nfs4_client_reclaim	32	[0, 8) == k1, [8, 16) == k1
	orangefs_bufmap	24, 32, 40	[24, 32) == k1, [32, 40) == k1, [40, 48) == k1
	fuse_dev	16	[0, 8) == 0, [40, 48) == k1, [48, 56) == k1
	netlbl_lsm_catmap	40	[40, 8) == k1
	xhci_command	16	[0, 8) == k2
kmalloc-64*	msg_msg	0, 32, 40	0: [0, 8) == k1, [16,24) $\neq$ Arg1 32: [32, 40) == k1, [16, 24) == Arg1, [24, 32) $\geq$ Arg2 40: [16, 24) == Arg1
	sched_group	16	[0, 8) == 0, [16, 24) == k1
	ctl_table	0	[20, 22) $>$ 0, [24, 32) == 0, [32, 40) == 0
	ip_vs_sync_thread_data	24	[0, 8) == k2, [16, 24) == 0
	dfs_info3_param	16, 24	∅
kmalloc-96	request_key_auth	40	[16, 24) == k1, [24, 32) == k1, [32, 40) == 0
	smc_buf_desc	16	[0, 8) == k1, [8, 16) == k1
	usb_request	0	∅
kmalloc-96*	ctl_table_header	32	∅
	port_buffer	0	[40, 48) == k1, [48, 56) == k1, [56, 64) == k1, [64, 68) == 0
kmalloc-128	ip6_flowlabel	32	[12, 16) $>$ 1, [64, 65) == 1
	virtio_vsock_pkt	104	[80, 88) == k1, [88, 96) == k1, [96, 104) == k1
	cfg80211_nan_func	32, 48, 64	∅
kmalloc-128*	nft_object	32	[64, 72) == k3
kmalloc-192	x509_certificate	32, 40, 48, 56	[16, 24) == k2, [24, 32) == k2, [32, 40) == k1, [40, 48) == k1, [48, 56) == k1, [56, 64) == k1
	kernfs_open_file	120	[104, 112) == k1, [112, 120) == k1, [136, 137) == 0
	urb	8, 96, 136	[64, 72) == k2
	ring_buffer	16	[64, 72) == k2, [72, 80) == k1, [80, 88) == k1
kmalloc-256	ima_rule_entry	88, 96	[88, 96) == k1, [96, 104) == k1
	station_info	184	∅
	nft_flow_rule	192	∅
	afs_sysnames	8n ($0 \leq n \leq 15$)	[128, 132) $>$ 0
kmalloc-512	ax25_cb	32	[0, 8) == k1, [8, 16) == k1, [464, 468) $>$ 0
	smb_vol	0, 8, 16, 24, 32, 152	∅
kmalloc-1024	mpoa_client	120	[128, 132) $>$ 0
	policydb	256, 288, 328	[256, 264) == k1, [288, 296) == k1, [328, 336) == k1
rsb_cache	dlm_rsb	232	∅
xattr_datum_cache	jffs2_xattr_datum	64	[24, 32) == k1, [32, 40) == k1

fail to release it. A detailed example is given in Fig. 7. Structure `inet6_dev` is allocated at Line 3 and released at Line 13. However, Line 15 on the release path access `pmc->idev` (two layers of nested accesses) and `pmc` is the heap pointer `idev->mc_tomb` from structure `inet6_dev`. So we excluded the Thanos objects whose heap pointer is accessed on the release path in a complicated nested way, which could also cause false negatives.

c) Error handling branches: When the release path of a Thanos object passes an error-handling branch, then we cannot reliably control the release anymore, which would fail the exploitation. `TAODE` identify the error-handling branches by

```
static struct inet6_dev *ipv6_add_dev(struct
    ↪ net_device *dev){
 struct inet6_dev *ndev;
 ndev = kzalloc(sizeof(struct inet6_dev),
     ↪ GFP_KERNEL); // alloc site
 ...
}
static void mld_clear_delrec(struct
    ↪ inet6_dev *idev){
 struct ifmcaddr6 *pmc, *nextpmc;
 pmc = idev->mc_tomb;
 for (; pmc; pmc = nextpmc) {
  nextpmc = pmc->next;
  ip6_mc_clear_src(pmc);
  in6_dev_put(pmc->idev); // nested access
  kfree(pmc); // release site
 }
 ...
}
```

Fig. 7. An example that nested access through the heap pointer occurs on the release path.

```
1  static struct map_info *build_map_info(
       ↪ struct address_space *mapping, ...){
2   struct map_info *curr = NULL;
3   struct map_info *prev = NULL;
4   struct map_info *info;
5   int more = 0;
6   ...
7   do {
8    info = kmalloc(sizeof(struct map_info),
         ↪ GFP_KERNEL); // alloc site
9    if (!info) { // error-handling branch
10    curr = ERR_PTR(-ENOMEM);
11    goto out;
12   }
13   info->next = prev;
14   prev = info;
15  } while (--more);
16 out:
17  while (prev)
18   prev = free_map_info(prev);
19  return curr;
20 }
21 static inline struct map_info *free_map_info
       ↪ (struct map_info *info){
22  struct map_info *next = info->next;
23  kfree(info); // release site
24  return next;
25 }
```

Fig. 8. An example that error-handling branch calls release function.

the branch label such as `error`, `exit`, and `fail` in a basic block and exclude such candidates. However, some labels (e.g., `out`, `clean`, and `free`) are used by both the error-handling branches and normal branches. TAODE ignores such equivocal labels when identifying the error-handling branches, which might result in false positives. A detailed example is given in Fig. 8. Structure `map_info` is allocated at Line 8. If the allocation fails, it would jump to the error-handling branch (Line 9) and calls `free_map_info()` to do release work. Fortunately, the false positives can be removed easily by manual analysis.

> **Answer to RQ1**: Based on the above analysis, we can conclude that TAODE can effectively identify Thanos Objects from real-world OSes with acceptable false rates.

C. Exploitation on Real-World Vulnerabilities

To prove the usefulness of the identified Thanos objects, we use them to exploit 20 real-world vulnerabilities. We list all the kernel vulnerabilities used for our evaluation in Table IV. From the column on the left to right, the results shown in the table indicate (1) the CVE-ID or Syzkaller-ID of the vulnerability, (2) the vulnerability type, (3) the cache type of the vulnerable object, (4) the capability of the vulnerability summarized manually, (5) which weak type the vulnerability belongs to, (6) whether traditional techniques can exploit the vulnerability, (7) the number of suitable Thanos objects useful for the exploitation of the vulnerability, (8) whether the vulnerability can be exploited by using Thanos objects.

1) Summary of Real-world Vulnerability Exploitation: Of the 20 vulnerabilities, 15 are successfully exploited using Thanos objects. Among the 15 exploited vulnerabilities, 8 OOB vulnerabilities have very limited write capabilities and 4 UAF vulnerabilities have no function pointer dereference, making traditional exploitation techniques fail. One OOB (CVE-2017-7184) that has unlimited write capability can be exploited both by traditional techniques and Thanos objects. This indicates that Thanos objects are effective in exploiting both weak primitives and strong primitives. Moreover, all exploitable vulnerabilities except CVE-2016-4557 have more than one Thanos object available for exploitation. Some vulnerabilities have a great many useful Thanos objects, this is because they have better write capabilities in relative or they can corrupt heap data in various caches. For example, CVE-2017-7184 can corrupt 7 cache types with arbitrary length, so there are 43 objects at most that can be used to exploit. This implies that TAODE could provide a security researcher with various approaches to craft a working exploit.

2) Case Study of CVEs: We first take CVE-2017-7533 as an example to show how the Thanos object is used in the exploitation. As Fig. 9 shows, CVE-2017-7533 is an OOB write vulnerability that can overwrite 11 arbitrary bytes to the adjacent heap chunk in kmalloc-96. The function `inotify_handle_event()` first calculates a length `alloc_len` (Line 8 to Line 10) and allocates a buffer (Line 13) to store the vulnerable object `inotify_event_info`. Then it copies a string `file_name` to the buffer (Line 16). However, another thread may change the `file_name` to a longer string between Line 9 and Line 16, which results in a buffer overflow. Though it can write 11 arbitrary bytes, it can't overwrite any function pointers, thus, we regard it as having a weak primitive.

To exploit CVE-2017-7533 using our strategy, first, we should find a Thanos object also in cache kmalloc-96 with a heap pointer in the front, so that the heap pointer can be overwritten by the vulnerability. We found 6 eligible Thanos objects: `cfg80211_nan_func_filter`, `pneigh_entry`, `msg_msg`, `ctl_table`, `usb_request`, and `port_buffer`. Then we use heap spraying techniques to put the selected Thanos object (i.e., `port_buffer`) right

TABLE IV

THE SUMMARY OF EXPLOITABILITY OF THE VULNERABILITIES WE USED. IN THE "CAPABILITY" COLUMN, *arb* DENOTES THAT THE VULNERABILITY CAN WRITE ARBITRARY VALUE, * DENOTES THAT THE WRITE OFFSET AND THE WRITE LENGTH CAN BE ARBITRARY, AND *hptr* DENOTES THAT THE WRITE VALUE IS A HEAP POINTER. IN THE "TRADITIONAL EXPLOITATION" AND "USING THANOS OBJECTS" COLUMNS, WE USE ✓ AND ✗ TO SHOW IF THE EXPLOITATION SUCCEED BY USING TRADITIONAL TECHNIQUES OR THANOS OBJECTS. # IN THE SEVENTH COLUMN INDICATES THE NUMBER OF THANOS OBJECTS USEFUL FOR THE EXPLOITATION OF THE CORRESPONDING VULNERABILITY

CVE-ID or Syzkaller-ID	**Type**	**Cache**	**Capability**	**Weak Type**	**Traditional Exploitation**	**Suitable Objects #**	**Using Thanos Objects**
a84d... [28]	OOB	kmalloc-32	(0, 4, arb)	write unimportant data	✗	4	✓
aaa3... [32]	OOB	kmalloc-256	(0, 4, NULL)	write limited value; write unimportant data	✗	0	✗
b0f0... [27]	UAF	kmalloc-32	(0, 32, arb)	no fptr-dereference	✗	10	✓
bf96... [30]	UAF	ip_dst_cache	(64, 4, arb)	no fptr-dereference	✗	0	✗
e4be... [29]	OOB	kmalloc-64	(0, 16, arb); (16, 8, 192); (24, 40, NULL)	write unimportant data	✗	8	✓
f2ae... [31]	OOB	kmalloc-256; kmalloc-512; kmalloc-1024; kmalloc-2048; kmalloc-4096	(*, 4, arb)	write unimportant data	✗	7	✓
CVE-2022-25636	OOB	kmalloc-32; kmalloc-128; kmalloc-192; kmalloc-256; kmalloc-512; kmalloc-1024; kmalloc-2048; kmalloc-4096;	$((56+80n)\%s, 8, hptr)$ (n = 1,2,3,...; s = 32, 128, 192, 256, 512, 1024, 2048, 4096)	write limited value; write unimportant data	✗	8	✓
CVE-2021-42008	OOB	kmalloc-4096	(14, *, arb)	write unimportant data	✗	7	✓
CVE-2021-26708	UAF	kmalloc-64	(40, 4, arb)	no fptr-dereference	✗	3	✓
CVE-2021-22555	OOB	kmalloc-4096	(0, 2, NULL)	write short length; write limited value; write unimportant data	✗	5	✓
CVE-2018-18559	UAF	kmalloc-2048	(1328, 8, arb)	-	✓	0	✗
CVE-2017-7533	OOB	kmalloc-96	(0, 11, arb); (11, 1, NULL)	write unimportant data	✗	6	✓
CVE-2017-7184	OOB	kmalloc-32; kmalloc-64; kmalloc-96; kmalloc-128; kmalloc-192; kmalloc-256; kmalloc-512	(0, *, arb)	-	✓	43	✓
CVE-2017-15649	UAF	kmalloc-4096	(2160, 8, arb)	-	✓	0	✗
CVE-2017-15265	UAF	kmalloc-512	(16, 64, arb); (304, 28, arb);	no fptr-dereference	✗	9	✓
CVE-2016-6516	OOB	kmalloc-64; kmalloc-96; kmalloc-128; kmalloc-196; kmalloc-256; kmalloc-512; kmalloc-1024; kmalloc-2048; kmalloc-4096	(*, 4, NULL)	write limited value; write unimportant data	✗	0	✗
CVE-2016-6187	OOB	kmalloc-8; kmalloc-16; kmalloc-32; kmalloc-64; kmalloc-128	(0, 1, NULL)	write short length; write limited value; write unimportant data	✗	9	✓
CVE-2016-4557	UAF	kmalloc-256	(56, 16, arb)	no fptr-dereference	✗	1	✓
CVE-2016-10150	UAF	kmalloc-64	(24, 16, arb)	-	✗	7	✓
CVE-2016-0728	UAF	kmalloc-256	(0, 8, arb)	-	✗	6	✓

after the vulnerable object (i.e, `inotify_event_info`). Next, we trigger the overflow and overwrite the heap pointer of the Thanos object to point to another existing heap chunk in kmalloc-1024, which has appropriate victim object and spray object for exploitation. After that, we release the overlapped chunk twice using the existing pointer and the fake heap pointer of the Thanos object, respectively. Finally, we use a victim object (such as `tty_struct` or `pipe_buffer`) and a spray object (such as the linear buffer of `sk_buff`) to take up the released chunk respectively. The spray object can craft

```c
int inotify_handle_event(..., const unsigned char *file_name, ...){// vulnerable function
   struct inotify_event_info *event;
   int len = 0;
   int alloc_len = sizeof(struct inotify_event_info);
   ...
   if (file_name) {
      len = strlen(file_name);
      alloc_len += len + 1;
   }
   ...
   event = kmalloc(alloc_len, GFP_KERNEL);
   ...
   if (len)
      strcpy(event->name, file_name); // overflow point
   ...
}
struct inotify_event_info { // vulnerable object
  struct fsnotify_event fse;
  int wd;
  u32 sync_cookie;
  int name_len;
  char name[];
};
struct port_buffer { // Thanos object
  char *buf; // target heap pointer
  size_t size;
  size_t len;
  size_t offset;
  dma_addr_t dma;
  struct device *dev;
  struct list_head list;
  unsigned int sgpages;
  struct scatterlist sg[0];
};
```

Fig. 9. Source code snippet of CVE-2017-7533.

a fake function pointer (`pipe_buffer->ops->release`) in the victim object to hijack the control-flow.

For CVE-2021-26708, which can write 4 arbitrary bytes at offset 40 in kmalloc-64, `TAODE` found 3 Thanos objects available: `orangefs_bufmap`, `netlbl_lsm_catmap` and `msg_msg`. While for CVE-2021-22555, which can overwrite 2 NULL bytes in the adjacent kmalloc-4096, `TAODE` found 5 Thanos objects available: `cfg80211_nan_func_filter`, `pneigh_entry`, `ctl_table`, `msg_msg` and `port_buffer`. Both CVE-2021-26708 and CVE-2021-22555 can overwrite the heap pointer of corresponding Thanos objects to point to an existing heap chunk and release it. Then we can use the versatile exploitation strategy to hijack control-flow and escalate privilege.

3) Analysis of the Failed Cases: Among the 20 tested vulnerabilities, 5 of them are failed to find suitable objects for their exploitation. We classify these failures into two categories. First, some vulnerabilities can only write at a special cache or write at an unusual offset. For example, the vulnerable object of `bf96\ldots` [30] is in a special cache named `ip_dst_cache`, and there is no Thanos object in the same cache found. As for CVE-2018-18559 and CVE-2017-15649, they can only write at a large offset (1328 and 2160) but we cannot find a Thanos object with a heap pointer at the same offset. Second, some vulnerabilities will write 4 NULL bytes to the adjacent object, such as `aaa3\ldots` [32] and CVE-2016-6516. We use an example to illustrate the difference of exploitation between 2 NULL bytes write and 4 NULL bytes write. Assume the vulnerable object is in the kmalloc-256 cache and there are two pointers pointing to two adjacent heap chunks in the kmalloc-256 cache. The first chunk is at 0xffffc9d0nnnnn000 pointed to by `ptr1`, while the second chunk is at 0xffffc9d0nnnnn100 pointed to by `ptr2`. The variable `n` can be an arbitrary hexadecimal number ($0 \leq n \leq$ 0xf). We use the two write capabilities to change the least two or four bytes of `ptr2` respectively and calculate the possibility that `ptr2` will point to the first chunk. The 2 NULL bytes write can change `ptr2` to 0xffffc9d0nnnn0000, while the 4 NULL bytes write can change `ptr2` to 0xffffc9d000000000. As the kernel heap address is randomized, the chances that the first chunk is at 0xffffc9d0nnnn0000 and 0xffffc9d000000000 are 1/16 and 1/1048576 (1/0 $\times$ 100000). Therefore, the success rate of creating the overlapped state using 4 NULL bytes write is quite low. An appropriate heap spraying strategy can improve the success rate a little, but it is still unacceptable in practice. This is why we cannot find suitable Thanos objects for `aaa3\ldots` [32] and CVE-2016-6516. As we have tested in practice, only 1 or 2 NULL bytes write could have an acceptable success rate.

> **Answer to RQ2**: Based on the exploitation of real-world vulnerabilities, we can conclude that the identified Thanos objects are usable as long as they are matched with suitable vulnerabilities.

D. Extra Benefits of Using Thanos Objects

1) Bypassing SMAP: In our experiments, two UAFs (CVE-2016-10150 and CVE-2016-0728) failed to be exploited by the traditional exploitation method owing to the protection of the SMAP scheme in Linux, however, they can still be exploited using our strategy. This is because, when exploiting these two vulnerabilities, the vulnerable objects in them are too small to place the exploit payloads. Traditional exploitation methods seek to place the payload in the user space, but reading user content directly from the kernel is prohibited by the SMAP scheme. Therefore, a precondition of the traditional exploitation method is disabling SMAP, otherwise, the exploitation would be failed. However, our strategy can use Thanos objects to transform the vulnerable objects to bigger kernel slabs that have more space to craft exploit payloads, which bypasses the SMAP. This indicates an advantage of using the Thanos object is bypassing certain kernel protection scheme.

2) Utilizing Leaked Heap Pointer: Using Thanos objects has another merit that it can better utilize the leaked information. For traditional exploitation techniques, the leaked information is useful only when it is a function pointer or an address of a global variable, which can be helpful to bypass KASLR. Whereas information such as the address of an ordinary heap pointer is mostly useless. However, for our approach, the address of a heap pointer is also useful, which can be used to construct the vulnerable overlapped state. For example, CVE-2022-25636 can leak the address of a heap pointer that points to object `net_device`. Based on this heap pointer, we can use a Thanos object to release object `net_device` and use a spray object to tamper the function table pointer in object `net_device`, finally, hijacking the control-flow. Hence, *our exploitation method can take advantage of the seemingly useless leaked heap pointer in the exploitation*. Given such a reason, Thanos objects can be used to break certain vulnerability patches. For example, given an exploitable double-free vulnerability, even when it has been patched (usually by eliminating one redundant free operation), the address of the vulnerable object is still known. The traditional exploitation approach is unworkable as only one free operation is left. However, our approach is still feasible as the address of the vulnerable object is known and the object is pointed to by a pointer. We can find a suitable Thanos object to release the vulnerable object and use a spray object to tamper with the function pointer in it, finally, the control-flow can be hijacked. Since we haven't found a real example, it is only a theoretical assumption.

> **Answer to RQ3**: Using Thanos objects has extra benefits, such as bypassing the SMAP scheme and better utilizing the leaked heap pointer, both can facilitate the exploitation.

VII. Discussion

A. The Accuracy of Object Identification

The accuracy of Thanos object identification is determined by the static analysis used in `TAODE`. First, `TAODE` employs the two-layer type analysis to construct control-flow graph and the LLVM built-in alias analysis pass to do alias analysis. Then, `TAODE` performs inter-procedural control-flow and data-flow analysis to explore the allocation path and the release path, which is the main part of `TAODE`. Therefore, the false positives and the false negatives mainly originate from these two procedures. Due to the resource constraints and the nature of static analysis, we cannot get an accurate control-flow and data-flow graph, making it hard to find the allocation path and the release path in deeper paths. And this is also hard to confirm by manual analysis in such a huge system. Second, as we have mentioned in the evaluation, nested structures, nested accesses and error handling branches during the control-flow and data-flow analysis can also bring in false reports.

B. Application of the Thanos Object

In reality, large numbers of vulnerabilities are regarded as "unexploitable" or underestimated owing to the difficulty in exploiting the weak primitives. Using the Thanos objects, we can transform weak primitives into strong primitives. Under this circumstance, such "unexploitable" vulnerabilities are reborn and would have a serious impact on the system security. We have proved with real-world vulnerabilities the feasibility of transforming weak primitives to strong primitives using Thanos objects. More importantly, we have identified numerous eligible Thanos objects from Linux, XNU, and FreeBSD. These Thanos objects can be paired with suitable vulnerabilities to make the exploitation feasible, and some of them can even bypass the existing mitigation mechanisms. For example, CVE-2016-10150 and CVE-2016-0728 are two UAF vulnerabilities that failed to exploit with traditional techniques owing to the SMAP mechanism in the Linux kernel. However, using Thanos objects, both of them can bypass SMAP and become exploitable again. Moreover, the exploitation approach with Thanos objects can better utilize the leaked information, such as the address of an ordinary heap pointer (CVE-2022-25636). Based on this, Thanos objects can be used to break certain vulnerability (e.g., double-free) patches.

C. The Distinction From Double-Free

Our versatile exploiting strategy needs to release an overlapped object twice so as to use a victim object and a spray object to take up the vulnerable object respectively. Though similar to the commonly seen exploitation of the double-free vulnerability, our strategy is different from it. First, in the double-free exploitation, the vulnerable overlapped state is caused by the same object (i.e., the vulnerable object), while in our versatile exploitation strategy, the vulnerable overlapped state can be caused by the same object type or different object types (as long as they are in the same cache). Second, in the double-free exploitation, the cache of the overlapped state is fixed, while in our versatile strategy we can decide the size of the overlapped object by controlling the heap pointer in the Thanos object. Third, in some caches, it is difficult to find both a perfect victim object and a perfect spray object at the same time. It fails to exploit the double-free if the vulnerable object falls in one of these caches and has no function pointer dereference itself. However, in our versatile strategy, it is much easier to exploit by constructing the vulnerable overlapped state in a different cache that has abundant victim objects and spray objects available. In summary, using the Thanos object, our versatile exploitation strategy is more flexible and practical than the traditional double-free exploitation. Besides, our strategy is also useful to exploit a double-free vulnerability.

D. Potential Mitigation Mechanisms

To defend against the versatile exploitation strategy based on the Thanos object, we can use the following alleviation approaches. First, the structure layout randomization [7] can randomize the offsets of field members in a structure, preventing an adversary from predicting the location of sensitive structure fields in kernel memory. However, Chen et al. [4] put forward a solution to bypass it. Second, we can isolate Thanos objects that `TAODE` identifies into individual shadow caches, which prevents an adversary from putting the Thanos object at or next to the vulnerable object. However, this approach should consider the performance overhead and it requires searching out all available Thanos objects.

VIII. Related Work

A. Kernel Exploitation

SemFuzz [40] uses Natural Language Processing to extract vulnerability-related text (e.g., CVE reports and Linux git logs) and guide the semantics-based fuzzing process to generate PoC exploits automatically. Lu et al. [22] proposed a deterministic

TABLE V
COMPARISON WITH OTHER TOOLS

Tool	Vulnerability	Target	Technique
FUZE	UAF	-	fuzz/SE
KEPLER	UAF/OOB/DF	special gadget	static analyse
SLAKE	UAF/OOB/DF	spray/victim object	static/dynamic analyse
KOOBE	OOB	-	fuzz
ELOISE	UAF/OOB/DF	elastic object	static analyse
TAODE	weak UAF/OOB	Thanos object	static analyse

stack spraying technique and an exhaustive memory spraying technique to facilitate the exploitation of uninitialized uses. FUZE [38] utilizes kernel fuzzing along with symbolic execution to identify, analyze, and evaluate the system calls valuable and useful for kernel UAF exploitation. KEPLER [37] can automatically generate a "single-shot" exploitation chain to facilitate the evaluation of control-flow hijacking primitives in the Linux kernel. SLAKE [5] uses static and dynamic analysis techniques to explore the kernel objects that are useful for kernel heap spraying, and the author proposed a technical approach to facilitate the slab layout adjustment.

For kernel OOB vulnerabilities, KOOBE [3] applies a novel capability-guided fuzzing solution to uncover hidden capabilities, and a way to compose capabilities together to further enhance the likelihood of successful exploitation. For kernel non-inclusive multi-variable races, EXPRACE [18] can turn hard-to-exploit races into easy-to-exploit races by manipulating an interrupt mechanism during the exploitation. Zeng et al. [41] proposed a new stabilization technique, called Context Conservation, to improve exploitation reliability for double-free and UAF vulnerabilities. SyzScope [42] and GREBE [19] both apply a new kernel fuzzing technique to explore all the possible error behaviors that a kernel bug might bring about. However, no research can tackle the problem when a vulnerability has a weak exploit primitive. Specifically, a UAF may have no function pointer dereference and an OOB write may have limited write length and write value. Using Thanos objects, we can transform a weak exploit primitive into a strong exploit primitive to promote the exploitation.

B. Bypassing Kernel Mitigation Mechanisms

Kem et al. [15] proposed a new kernel exploitation technique, called return-to-direct-mapped memory (ret2dir), which bypasses all existing ret2usr defenses, namely SMEP [14], SMAP [34], PXN [2], KERNEXEC [26], UDEREF [25], and kGuard [16]. When kernel physmap was set to be non-executable, Xu et al. [39] proposed two practical memory collision attacks to exploit UAF: An object-based attack that leverages the memory recycling mechanism of the kernel allocator to achieve freed vulnerable object covering, and a physmap-based attack that takes advantage of the overlap between the physmap and the SLAB caches to achieve a more flexible memory manipulation. In the wild, the adversary usually constructs ROP chain [36] to bypass SMEP and flips corresponding bits in the cr4 register [17] to bypass SMAP. There are several approaches to defeating KASLR. Gruss et al. [12] and Jiang et al. [13] utilize hardware attributes and side-channel attacks to leak kernel information. Cho et al. [6] present a generic approach that converts stack-based information leaks in Linux kernel into kernel-pointer leaks. ELOISE [4] utilizes static/dynamic analysis methods to pinpoint elastic kernel objects that can be used to leak kernel information and then employs constraint solving to pair them to corresponding kernel vulnerabilities. Though our work does not focus on bypassing kernel mitigation mechanisms, the existing techniques can be auxiliary. Especially when we begin to corrupt the target heap pointer of the Thanos object, we can use the techniques above to leak some kernel addresses first.

IX. CONCLUSION

In this paper, we proposed a versatile strategy that can transform weak exploit primitives into strong exploit primitives. Using a special object in the kernel called the Thanos object, our strategy can exploit a UAF that does not have function pointer dereference or an OOB write that just has limited write length and write value. We facilitate the strategy, we devised a tool `TAODE` to search for eligible Thanos objects from the kernel and pair them with appropriate vulnerabilities. We have successfully identified numerous Thanos objects from Linux, XNU, and FreeBSD. Using the identified Thanos objects, we have proved the feasibility of our approach with 20 real-world kernel vulnerabilities, most of which traditional techniques fail to exploit.

ACKNOWLEDGMENT

The authors would like to sincerely thank all the reviewers for your time and expertise on this article. Your insightful comments help us improve this work.

REFERENCES

[1] K. Alspach. (2017). *Major Attacks Using Log4J Vulnerability 'Lower Than Expected*. [Online]. Available: https://venturebeat.com/2022/01/24/major-attacks-using-log4j-vulnerability-lower-than-expected/

[2] *Architecture Reference Manual*, document ARMv7-A and ARMv7-R Edition, A. ARM, 2012.

[3] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 1093–1110.

[4] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1165–1184.

[5] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the Linux kernel," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1707–1722.

[6] H. Cho et al., "Exploiting uses of uninitialized stack variables in Linux kernels to leak kernel pointers," in *Proc. 14th USENIX Workshop Offensive Technol. (WOOT)*, 2020, p. 1.

[7] K. Cook. (2017). *Security Things in Linux v4.13*. [Online]. Available: https://outflux.net/blog/archives/2017/09/05/security-things-in-linux-v4-13/

[8] J. Corbet. (2018). *A Page-Table Isolation Update*. [Online]. Available: https://lwn.net/Articles/752621/

[9] T. M. Corporation. (2021). *Common Vulnerability and Exposures*. [Online]. Available: https://cve.mitre.org/cve/

[10] C. Details. (2022). *Linux: Vulnerability Statistics*. [Online]. Available: https://www.cvedetails.com/vendor/33/Linux.html

[11] Google. (2018). *Syzbot: Google Continuously Fuzzing the Linux Kernel*. [Online]. Available: https://syzkaller.appspot.com/upstream

[12] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 368–379.

[13] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 380–392.

[14] M. Jurczyk and G. Coldwind. (2011). *Smep: What is it, and How to Beat it on Windows*. [Online]. Available: https://j00ru.vexillium.org/2011/06/smep-what-is-it-and-how-to-beat-it-on-windows/

[15] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 957–972.

[16] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *Proc. 21st USENIX Secur. Symp. (USENIX Secur.)*, 2012, pp. 459–474.

[17] A. Konovalov. (2017). *Exploiting the Linux Kernel Via Packet Sockets*. [Online]. Available: https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html

[18] Y. Lee, C. Min, and B. Lee, "Exprace: Exploiting kernel races through raising interrupts," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 2363–2380.

[19] Z. Lin et al., "GREBE: Unveiling exploitation potential for Linux kernel bugs," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 2078–2095.

[20] K. Lu and H. Hu, "Where does it go?: Refining indirect-call targets with multi-layer type analysis," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 1867–1881.

[21] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1769–1786.

[22] K. Lu, M.-T. Walter, D. Pfaff, S. Nuernberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[23] LWN. (2013). *Kernel Address Space Layout Randomization*. [Online]. Available: https://lwn.net/Articles/569635/

[24] A. Nguyen. (2021). *CVE-2021–22555: Turning @00@00 into 10000$*. [Online]. Available: https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html

[25] PAX. (2013). *Homepage of the PAX Team*. [Online]. Available: http://pax.grsecurity.net/

[26] B. Spengler. (2022). *The Guaranteed end of Arbitrary Code Execution*. [Online]. Available: https://grsecurity.net/PaX-presentation.pdf

[27] Syzbot. (2017). *Kasan: Use-After-Free Read in MPI_Free*. [Online]. Available: https://syzkaller.appspot.com/bug?id=b0f0a3d34f0e9d551e1c0ab1fd911aaaa18bdcb9

[28] Syzbot. (2018). *Kasan: Slab-Out-of-Bounds Write in Crypto_Dh_Encode_Key*. [Online]. Available: https://syzkaller.appspot.com/bug?id=a84d6ad70b281bfc5632f272f745104fb43d219d

[29] Syzbot. (2018). *Kasan: Slab-Out-of-Bounds Write in Sha512_Final*. [Online]. Available: https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c

[30] Syzbot. (2018). *Kasan: Use-After-Free Write in Dst_Release*. [Online]. Available: https://syzkaller.appspot.com/bug?id=bf967d2c5ba62946c61152534c8b84823d848f05

[31] Syzbot. (2020). *Kasan: Slab-Out-of-Bounds Write in Hiddev_Ioctl_Usage*. [Online]. Available: https://syzkaller.appspot.com/bug?id=f2aebe90b8c56806b050a20b36f51ed6acabe802

[32] Syzbot. (2021). *Kasan: Slab-Out-of-Bounds Write in Xfrm_Attr_Cpy32*. [Online]. Available: https://syzkaller.appspot.com/bug?id=aaa35b314220404bbc2b1c66067b0f9a623baa89

[33] Wikipedia. (2017). *Wannacry Ransomware Attack*. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[34] Wikipedia. (2021). *Supervisor Mode Access Prevention*. [Online]. Available: https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention

[35] Wikipedia. (2022). *Executable Space Protection*. [Online]. Available: https://en.wikipedia.org/wiki/Executable_space_protection

[36] Wikipedia. (2022). *Return-Oriented Programming*. [Online]. Available: https://en.wikipedia.org/wiki/Return-oriented_programming

[37] W. Wu, Y. Chen, X. Xing, and W. Zou, "Kepler: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, vol. 2019, pp. 1187–1204.

[38] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 781–797.

[39] W. Xu et al., "From collision to exploitation: Unleashing use-after-free vulnerabilities in Linux kernel," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 414–425.

[40] W. You et al., "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2139–2154.

[41] K. Zeng et al., "Playing for K(H)eaps: Understanding and improving Linux kernel exploit reliability," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 71–88.

[42] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing high-risk security impacts of fuzzer-exposed bugs in Linux kernel," 2021, *arXiv:2111.06002*.

Danjun Liu received the B.S. and M.S. degrees in computer science and technology from the National University of Defense Technology, Changsha, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree in cyberspace security. His research interests include operating systems and software security.

Pengfei Wang received the B.S., M.S., and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 2011, 2013, and 2018, respectively. He is currently an Assistant Professor with the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Xu Zhou received the B.S., M.S., and Ph.D. degrees from the School of Computer Science, National University of Defense Technology, China, in 2007, 2009, and 2013, respectively. He is currently an Associate Professor with the School of Computer Science, National University of Defense Technology. His research interests include operating systems and security.

Wei Xie received the Ph.D. degree in communication network security from the School of Electronic Science, National University of Defense Technology, China, in 2014. He is currently an Associate Professor with the College of Computer, National University of Defense Technology. His recent research interests include firmware vulnerability detection, web pentest, and AI-based security.

Gen Zhang received the B.S., M.S., and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 2016, 2018, and 2022, respectively. His research interests include fuzzing and software testing.

Zhenhao Luo received the B.S. and M.S. degrees in cyberspace security from the National University of Defense Technology, Changsha, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree in cyberspace security. His current research interests include binary code similarity detection and vulnerability detection.

Tai Yue received the B.S. degree from the Department of Mathematics, Nanjing University, Nanjing, in 2017, and the M.S. degree from the College of Computer, National University of Defense Technology, Changsha, in 2019, where he is currently pursuing the Ph.D. degree. His research interests include operating systems and software security.

Baosheng Wang received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, China. He is currently a Professor with the School of Computer Science, National University of Defense Technology. His current research interests include internet architecture, high-performance computer networks, and network security.