BRIDGEROUTER: Automated Capability Upgrading of Out-Of-Bounds Write Vulnerabilities to Arbitrary Memory Write Primitives in the Linux Kernel

Dongchen Xie[1], Dongnan He[1], Wei You[1,*], Jianjun Huang[1], Bin Liang[1], Shuitao Gan[2], Wenchang Shi[1]
[1]*School of Information, Renmin University of China, Beijing, China*
Email: {dongchenx, hedongnan, youwei, hjj, liangb, wenchang}@ruc.edu.cn
[2]*Laboratory for Advanced Computing and Intelligence Engineering*
Email: ganshuitao@gmail.com

***Abstract*—Memory corruption vulnerabilities pose a significant threat to the Linux kernel, with out-of-bounds (OOB) vulnerabilities receiving particular attention due to their prevalence. The existing kernel OOB exploitation techniques either require strong capabilities from the vulnerabilities, demand that the vulnerable and victim objects reside in the same memory allocator cache, or rely on extensive page table manipulation. These constraints restrict their applicability and lead to low success rates in completing a full exploitation chain. In this paper, we propose a practical approach that enables arbitrary memory writes from kernel OOB vulnerabilities with limited capabilities. Our method leverages two special kinds of kernel objects to upgrade the capability from an uncontrolled overwrite to a controlled overwrite, ultimately achieving arbitrary memory write. We develop a system to automatically identify and utilize these two kinds of kernel objects. Evaluations on a crafted vulnerability and 14 representative real-world vulnerabilities, along with a comparison against two state-of-the-art works, demonstrate the broad applicability of our approach.**

1. Introduction

The Linux kernel is a fundamental component of modern computing infrastructure, making its security critically important. Memory corruption vulnerabilities pose a significant threat to the Linux kernel. According to CVEdetails [6], memory corruption vulnerabilities accounted for 77% of all Linux kernel vulnerabilities from 2014 to 2024. These vulnerabilities can lead to severe consequences, such as privilege escalation that grants attackers full control over the targeted system.

For most Linux kernel vulnerabilities, a proof-of-concept (PoC) is often available to demonstrate that the vulnerability can be triggered. However, it is frequently unclear whether a given vulnerability is actually exploitable, such as whether it can be used for privilege escalation. Fixing vulnerabilities is typically a lengthy and tedious process [31]. To streamline this process, it is crucial to identify the most critical vulnerabilities among the vast number and prioritize those that are exploitable. Automatic exploit generation (AEG) [10], [12], [39], [47], [52] has emerged as a valuable solution. It can automatically generate exploits, allowing developers to focus on addressing high-risk vulnerabilities.

Out-of-bounds (OOB) vulnerabilities are particularly critical in the Linux kernel. Statistics indicate that OOB vulnerabilities have consistently ranked among the top three [5]. In response, various approaches [15], [16], [17], [40] have been developed to automate the exploitation of kernel OOB vulnerabilities. These methods focus on optimizing memory layouts and manipulating memory allocations to facilitate exploitation. However, they either require strong capabilities from the vulnerabilities, demand that the vulnerable and victim objects reside in the same memory allocator cache, or rely on extensive page table manipulation. As a result, these methods suffer from restricted applicability and generally exhibit low success rates in achieving an exploitation chain.

In this work, we propose a *practical* approach to kernel OOB write exploitation that *enables arbitrary memory writes from vulnerabilities with limited capabilities*. The basic idea of our method is to leverage two special kinds of kernel objects, which we refer to as *bridge* and *router* objects, to upgrade the capability. Specifically, a bridge object converts an *uncontrolled overwrite capability* that writes to an adjacent location in one memory allocator cache into a *controlled overwrite capability* that writes to an adjacent location in a different memory allocator cache. Meanwhile, a router object directs the destination and/or the source buffers of a memory copy operation, resulting in an *arbitrary memory write* primitive.

We develop a system, named BRIDGEROUTER, to automatically identify and utilize the bridge and router objects for exploiting kernel OOB vulnerabilities. Starting with the source code of the target kernel, we employ static analysis to identify bridge and router objects, and use fuzzing to generate the system call sequences needed to trigger the allocation and memory copy operations for the identified objects. Given a PoC of an OOB vulnerability, we match the corresponding vulnerable object with the appropriate bridge and router objects, and explore their potential capabilities for generating a prototype exploit. Finally, the complete exploit is assembled by integrating the prototype exploit with carefully designed memory management and system call scheduling strategies.

*Wei You is the corresponding author.

DOI 10.1109/SP61157.2025.00132

We evaluated BRIDGEROUTER on Linux kernel v6.6. Sampling experiments show that identifying bridge and router objects yields low false positive (FP) and false negative (FN) rates, with a 16% FP rate for bridge objects, a 38% FP rate for router objects, and an overall 0% FN rate. In practice, this performance is highly satisfactory. We believe that, for a potentially exploitable OOB vulnerability, BRIDGEROUTER can effectively identify the bridge and router objects needed to generate a complete exploit. We also demonstrated that BRIDGEROUTER is applicable to generic memory allocator caches in the Linux kernel and effective in exploiting 14 representative real-world vulnerabilities by chaining appropriate bridge and router objects to achieve arbitrary memory write capabilities. Comparison with two state-of-the-art (SOTA) tools (KOOBE [15] and SLUBStick [40]) further highlights the advantages of our approach.

Our contributions are summarized as follows.

- A practical exploitation technique is proposed that upgrades the limited capabilities of a kernel OOB vulnerability to enable arbitrary memory writes, facilitating further attacks such as privilege escalation.
- A prototype system is developed to automate the proposed exploitation approach. The experimental data and source code are available at a GitHub repository https://github.com/CheUhxg/BridgeRouter.
- A comprehensive evaluation is conducted to assess the accuracy of identifying bridge and router objects in the Linux kernel, the applicability across different memory allocator caches, and the effectiveness in exploiting real-world kernel vulnerabilities.

2. Background

Kernel Heap Memory Management. The Linux kernel employs a combination of the Buddy allocator [2] and the SLAB/SLUB allocator [8] for heap memory management. The Buddy allocator provides large, contiguous memory in chunks of page-order size (i.e., $2^n \times$`PAGE_SIZE`) to the SLAB/SLUB allocator, which organizes the memory into *caches*, each holding kernel objects of the same type (for dedicated caches) or similar sizes (for generic caches). Within each cache, memory is further divided into *slabs*, which are partitioned into multiple individual slots to store objects. When the objects are deallocated, they are returned to the corresponding slabs. If an entire slab is emptied, the SLAB/SLUB allocator releases the associated memory pages back to the Buddy allocator. This layered approach allows the Buddy allocator to manage overall memory allocation, while the SLAB/SLUB allocator efficiently handles small, fixed-size objects.

Exploitation of Heap OOB Write Vulnerability. The basic idea of exploiting a heap OOB write vulnerability is to overwrite a *victim object* containing critical data or pointer fields, which is positioned adjacent to a *vulnerable object* intended to be accessed. Figure 1 illustrates the typical exploitation workflow. Given a vulnerability, an adversary

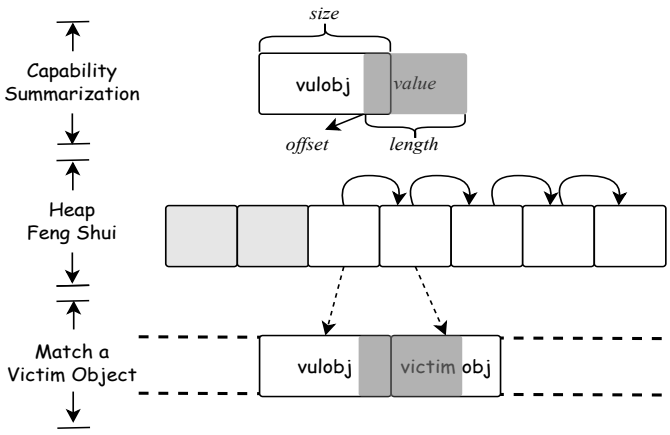


Figure 1: The typical exploitation workflow of heap OOB.

first summarizes its capability in terms of how far the write can reach (*offset*), how many bytes can be written (*length*), and what content can be written (*value*). The adversary then carefully selects a victim object that matches the vulnerable object, and leverages various heap feng shui techniques [17], [22], [32] to manipulate the memory layout, ensuring a victim object can be positioned adjacent to the vulnerable object.

Exploiting a heap OOB write in the Linux kernel is challenging due to its unique heap memory management mechanism. Objects must be of the same type or similar sizes to be placed in the same cache, allowing them to potentially be adjacent. In many cases, the adversary cannot find a suitable victim object that can be placed in the same cache as the vulnerable object. To overcome this challenge, *cross-cache* reuse techniques [33] have been proposed. Specifically, the adversary frees all slots of a slab page, causing the SLAB/SLUB allocator to return the page containing the vulnerable object to the Buddy allocator (i.e., recycling). This page can then be allocated for other slab caches (i.e., reclaiming), enabling the reuse of memory slot across slab caches with different types, allocation sizes, and properties.

Note that the capabilities of most heap OOB vulnerabilities are generally limited. A common approach in real-world exploits is to upgrade the OOB capability to arbitrary memory write primitives, enabling further attacks such as privilege escalation [34] and container escape [3]. For example, the exploit [4] for CVE-2022-0185 upgrades the OOB write vulnerability to arbitrary memory write, allowing it to overwrite the `euid` field of the `cred` structure associated with the current process to `GLOBAL_ROOT_UID` (i.e., a value of 0), thereby gaining root privilege.

Scope and Assumptions. This work is dedicated to developing an automatic exploitation technique that upgrades kernel heap OOB write vulnerabilities to arbitrary memory write primitives. We assume that the target kernel is protected by widely-deployed defenses, including kernel address space layout randomization (KASLR), kernel control flow integrity (KCFI), kernel page table isolation (KPTI), supervisor mode execution prevention (SMEP), and supervisor mode access prevention (SMAP). The automatic capability upgrade

```
1  int diWrite(tid_t tid, struct inode *ip) {
2    dtpage_t *p = JFS_IP(ip)->i_dtroot; // src dentry page
3
4    struct dinode *dp = read_metapage(jfs_ip->ipimap, ...);
5    dtpage_t *xp = &dp->di_dtroot; // dest dentry page
6
7    struct linelock *ilinelock = get_linelock(jfs_ip, ...);
8    struct lv *lv = ilinelock->lv; // log vector
9
10   for (int n = 0; n < ilinelock->index; n++, lv++) {
11     memcpy(&xp->slot[lv->offset], &p->slot[lv->offset],
12            lv->length << L2DTSLOTSIZE); // OOB write
13   }
14   ...
15 }
```

Figure 2: An OOB vulnerability discovered by syzbot.

technique proposed in this work does not violate these protections. Further attacks based on arbitrary memory write primitives may require bypassing these protections using other techniques [16], [27], [38], [49].

3. Motivation

We use a running example to demonstrate the limitations of existing techniques and motivate our approach.

3.1. Running Example

Figure 2 shows a simplified excerpt of the vulnerable code in Linux kernel v6.9.0-rc5 discovered by syzbot [42]. The vulnerable function `diWrite`, located in the journaled file system, is responsible for writing a portion of the in-memory inode (ip) to its corresponding on-disk inode (dp). In particular, certain directory entry slots are copied from the source directory entry page (p) to the destination directory entry page (xp), as specified by the log vector (lv) stored in the transaction lock ($ilinelock$). Each log item records the *offset* and *length* of a modification made to the file system. The offset is controllable through a specific system call from the user space, while the length is fixed and the values of the source directory entry page are uncontrollable. Due to insufficient bounds checking, a crafted offset may cause an OOB write, potentially allowing the memory adjacent to the destination directory entry page to be overwritten with uncontrolled values. At the time of writing, this vulnerability has not been officially fixed, and no public exploit is available.

3.2. Limitations of Existing Techniques

The most related SOTA research works that enable capability upgrades for exploiting kernel heap OOB vulnerabilities are KOOBE [15] and SLUBStick [40].

KOOBE. KOOBE is mainly designed to convert an OOB write vulnerability to a control flow hijack primitive. It requires a high-capability vulnerability that allows overwriting with a controllable value indicating a valid address. Furthermore, the search space for victim objects is restricted to the kernel objects that can be allocated in the same cache as the vulnerable object. As a result, KOOBE is less effective for OOB vulnerabilities with limited capability, such as the one in the running example, which only allows overwriting with uncontrolled values.

SLUBStick. SLUBStick converts a limited heap vulnerability to a page table manipulation, thereby granting the capability to read and write memory arbitrarily. It exploits an OOB vulnerability in three steps. First, it pivots the given vulnerability into a double-free vulnerability, resulting in a dangling pointer with a memory write primitive (MWP) that allows the adversary to write a controlled value at a chosen time. Second, it leverages the cross-cache reuse technique to recycle a slab page that contains the MWP and then reclaims the slab page as a page table. Finally, by triggering the MWP, it overwrites the page table entries to obtain an arbitrary memory read-and-write primitive.

SLUBStick provides a *generic* yet *heavy* approach for kernel OOB exploitation. While it proposes a side-channel leakage technique to improve the success rate of cross-cache reuse, the overall success rate of the full-chain exploitation remains low (less than 3% according to our evaluation in §5.3). The primary challenge lies in satisfying the complex and overlapping time windows between the free and use stages (i.e., the pivoting step) and between converting the dangling pointer to an MWP (i.e., before recycling) and when it is triggered (i.e., after reclaiming). The detailed analysis of the time window can be found in our repository [1]. Additionally, unintended page table manipulation will be detected and prevented by the page table check security defense [7], introduced since Linux kernel v5.17.

3.3. Our Approach

We propose a *practical* approach for kernel OOB exploitation, named BRIDGEROUTER, which enables arbitrary memory writes without the need for vulnerability pivoting, slab page recycling and reclaiming, or page table manipulation. It reduces the complexity of time windows and complies with the page table protection mechanism, thus ensuring the success of complete exploitation.

Our approach involves two core steps: *cross-cache overwrite migration* and *memory copy redirection*. In the first step, an *uncontrolled overwrite capability* that writes to an adjacent location within one slab cache is converted to a *controlled overwrite capability* that writes to an adjacent location in a different slab cache. In the second step, the controlled overwrite capability is then used to redirect the *destination* and/or the *source* of a memory copy operation, resulting in an arbitrary memory write primitive.

The assumptions regarding the capability of a heap OOB write vulnerability to be exploitable via our approach as follows: although the OOB write may be uncontrollable, the offset (how far the write can reach), value (what content can be written), and length (how many bytes can be written) of the OOB write must fall within an appropriate range. Specifically, the offset must be large enough to overwrite adjacent objects in the slab cache; the overwritten value

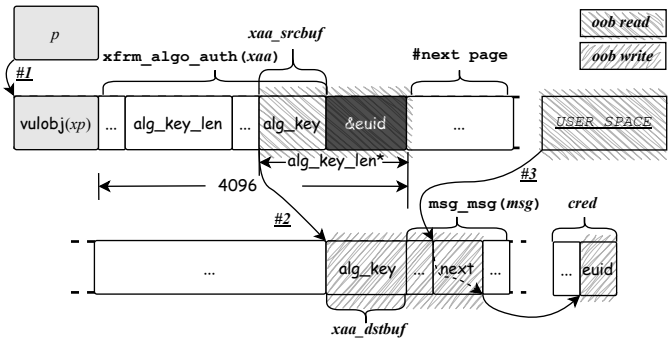


Figure 3: Exploiting the running example.

should also be sufficiently large to trigger an overflow when copying a bridge object. However, the length must not be excessively large to avoid the risk of corrupting critical fields in adjacent objects, which could result in a system crash.

Cross-Cache Overwrite Migration. It is achieved by placing a special kind of kernel object, which contains a length field, adjacent to the vulnerable object. This kind of object participates in a memory copy operation, where the source buffer is controllable, the destination buffer resides in the kernel heap, and the number of bytes to copy is determined by the length field of the object. We refer to this kind of object as a *bridge object*. When an OOB write is triggered on the vulnerable object, the length field of the adjacent bridge object is overwritten. This, in turn, causes the destination buffer associated with the bridge object to be overwritten with controlled values during the memory copy operation.

Memory Copy Redirection. It is achieved by placing a special kind of kernel object, which contains pointer fields, adjacent to the destination buffer associated with a bridge object. This kind of object participates in a memory copy operation (distinct from the one in cross-cache overwrite migration), where its pointer fields point to the destination buffer and/or the source buffer. We refer to this kind of object as a *router object*. When a controlled overwrite occurs on the destination buffer associated with the bridge object, the pointer fields of the adjacent router object, which point to the destination buffer and/or the source buffer of the memory copy operation, will be overwritten with controlled values. This leads to arbitrary memory writes when the memory copy operation occurs.

Exploiting Running Example. To exploit the vulnerability in the running example, we use an `xfrm_algo_auth` object as the bridge object and a `msg_msg` object as the router object, and arrange the memory layout as shown in Figure 3. In particular, the bridge object (xaa) is positioned adjacent to the vulnerable object (xp), and the router object (msg) is positioned adjacent to the destination buffer associated with the bridge object (xaa_dstbuf).

When the vulnerability is triggered (#1 in Figure 3), $xaa \rightarrow alg\_key\_len$ is overwritten with an uncontrolled value, which then affects the number of bytes copied in a `memcpy` operation. Although the overwritten value is uncontrolled, it originates from directory entry slots, making it likely to fall within a range suitable for an out-of-bounds length. The source buffer resides in the bridge object, and its overflow content can be controlled by pre-sprayed values. The destination buffer resides in a different slab cache from the one containing the vulnerable object and the bridge object. Consequently, the `memcpy` operation will perform an out-of-bounds copy (#2 in Figure 3), causing the $next$ field of msg, which is adjacent to the destination buffer, to be overwritten with a controlled value. The router object msg is involved in a `copy_from_user` operation (#3 in Figure 3). The source buffer resides in the user space, and is fully controllable by the adversary. The destination buffer is pointed to by $msg \rightarrow next$, which, as discussed earlier, can be overwritten with a controlled value. As a result, an arbitrary memory write is achieved, allowing a controlled value to be written to a controlled destination.

Technical Challenges. To perform the exploitation described above, an adversary should overcome several technical challenges. First, the adversary needs to identify the potential bridge objects and the router objects within the kernel. Given the complexity of kernel code and the wide range of kernel versions, manually auditing the code to pinpoint these kernel objects is an extremely labor-intensive task. Second, the adversary needs to trigger the allocation and memory copy operations associated with these identified objects through specific system call sequences. The vast search space makes blindly and exhaustively testing various system call combinations impractical and inefficient. Third, the adversary needs to chain the vulnerable object with the appropriate bridge and router objects, and explore their potential capabilities. Simply chaining these objects and triggering the corresponding allocation and memory copy operations is insufficient, as numerous noises may prevent full exploration of their potential capabilities. Finally, the adversary needs to synthesize attack components to generate a complete exploit. This requires simultaneously considering various factors, including memory layout manipulation, memory content spraying, and time window arrangement.

4. Design

Figure 4 outlines the workflow of BRIDGEROUTER, which is divided into four procedures. Starting with the source code of the target kernel, we first leverage static analysis to identify bridge and router objects (§4.1); then we perform a two-stage constraint-guided fuzzing process to generate the system call sequences required to trigger the allocation and memory copy operations for the identified objects (§4.2). Given a PoC of an OOB vulnerability, we match the corresponding vulnerable object with appropriate bridge and router objects, and explore their potential capabilities for generating a prototype exploit through a phased capability-guided fuzzing in a simulated environment. (§4.3). Finally, the complete exploit is assembled by integrating the prototype exploit with carefully designed memory management and system call scheduling strategies (§4.4).

Our approach combines static and dynamic analysis. This is because the proposed automated capability upgrading

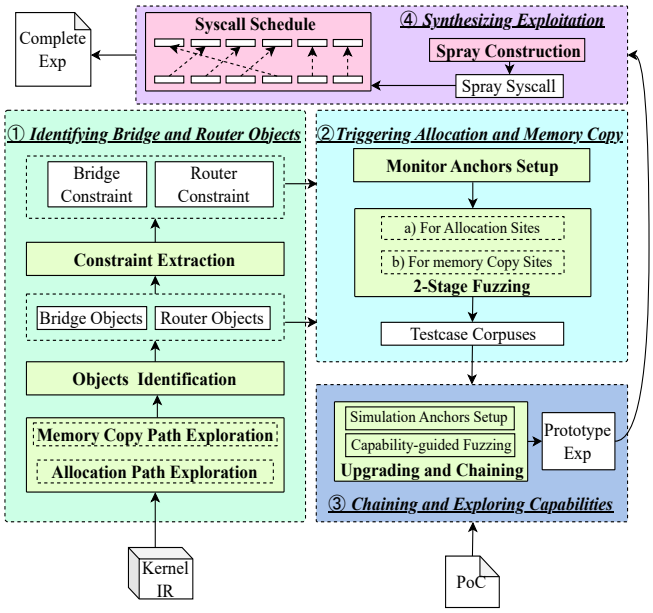


Figure 4: The overall workflow of BRIDGEROUTER.

technique requires not only identifying bridge and router objects but also determining the system call sequences that trigger the allocation and memory copy operations for these identified objects. While static analysis can help identify potential bridge and router candidates, it falls short in determining the triggering system call sequences due to the complexity and dynamic nature of system calls (e.g., inter-component indirect calls and stateful behaviors). These factors make it challenging for static analysis to accurately trace and link system calls to memory operations of the identified objects without runtime execution context.

4.1. Identifying Bridge and Router Objects

Identifying bridge and router objects requires exploring the allocation and memory copy paths of kernel objects. Additionally, we also need to extract the constraints along the memory copy paths. The list of kernel functions responsible for allocation and memory copy is shown in Appendix B.

Definitions. To facilitate discussion, we define some maps in Figure 5. `Alloc2SyscallSet` and `Copy2SyscallSet` map each allocation site and memory copy site to their respective sets of triggering system calls. `Type2AllocSet` maps each object type to its corresponding set of allocation sites. `Copy2SrcObjType`, `Copy2DstObjType`, `Copy2SizeObjType` map each copy allocation to the object type containing the fields that specify the source buffer, destination buffer and copy size, respectively. `Copy2SrcAllocSet` and `Copy2DstAllocSet` map each memory copy site to the set of allocation sites for the corresponding source and destination buffers. `Bridge2CopySet` and `Router2CopySet` map the bridge and router objects to their respective sets of memory copy sites. `Bridge2CstSet` and `Router2CstSet` map the bridge and router

```
Alloc2SyscallSet ::= AllocSite → {SystemCall}
 Copy2SyscallSet ::= CopySite → {SystemCall}
   Type2AllocSet ::= ObjType → {AllocSite}
 Copy2SrcObjType ::= CopySite → ObjType
 Copy2DstObjType ::= CopySite → ObjType
Copy2SizeObjType ::= CopySite → ObjType
Copy2SrcAllocSet ::= CopySite → {AllocSite}
Copy2DstAllocSet ::= CopySite → {AllocSite}
  Bridge2CopySet ::= ObjType → {CopySite}
  Router2CopySet ::= ObjType → {CopySite}
   Bridge2CstSet ::= ObjType → {Constraint}
   Router2CstSet ::= ObjType → {Constraint}
```

Figure 5: Definitions of some maps.

objects to their respective sets of constraints along their memory copy paths.

Allocation Path Exploration. Figure 6a illustrates the exploration of allocation paths. It begins with a backward inter-procedural control-flow analysis for each allocation site. If a path is identified that connects an allocation site $alloc$ with a system call $sysacall$ without requiring the root privilege, the relationship between $alloc$ and $syscall$ is recorded in `Alloc2SyscallSet`. Subsequently, a forward inter-procedural data-flow analysis is performed to determine the object type associated with each allocation site. Specifically, we track the use points of the return value from each allocation site, focusing on those instructions relevant to type casting, pointer dereferencing and argument passing. The operands of these instructions can be used to infer the object type. The relationship between each object type $type$ and its corresponding set of allocation sites $\{alloc\}$ is recorded in `Type2AllocSet`.

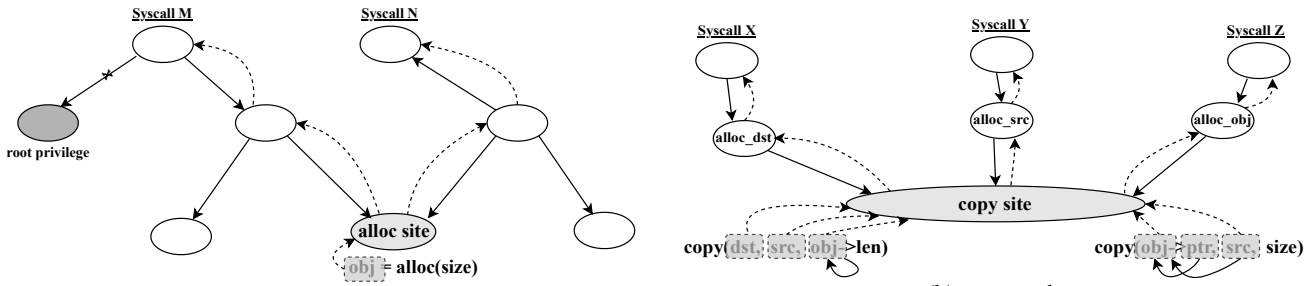
Memory Copy Path Exploration. Figure 6b illustrates the exploration of memory copy paths. Similar to allocation path exploration, we only consider those memory copy sites that are reachable from a system call without requiring root privilege. The relationship between each memory copy site $copy$ to its corresponding set of triggering system calls $\{syscall\}$ is recorded in `Copy2SyscallSet`.

For each memory copy site $copy$ under consideration, we perform a backward inter-procedural data-flow analysis on its dst, src, and $size$ arguments. If $size$ originates from a field of a heap object obj (e.g., $obj \rightarrow len$) and the type of obj is $type$, we record the relationship between $copy$ and $type$ in `Copy2SizeObjType`. Similarly, if src (or dst) originates from a field of a heap object obj (e.g., $obj \rightarrow ptr$), we record the relationship between the memory copy site and the type of the containing object in `Copy2SrcObjType` (or `Copy2DstObjType`).

We also need to locate the allocation sites of src and dst for each memory copy site $copy$. For a memory copy function that transfers data from the user space (e.g., `copy_from_user`), we mark the allocation set of src as `USER_SPACE`. If src (or dst) originates from a field of a heap object and the type of the field is $type$, the allocation sites of src (or dst) are given by `Type2AllocSet`$[type]$. Alternatively, if src (or dst) originates from a return value of an allocation function call, that function call site is

TABLE 1: Examples of the bridge and router objects.

Category	Struct	Caches	Offset (len/ptr)	Source Buffer	Destination Buffer	Constraints
Bridge	xfrm_algo_auth	$\geq$ kmalloc-96	[64, 68)	$\geq$ kmalloc-96	$\geq$ kmalloc-1k	NULL
Router	msg_msg	kmalloc-4k	[32, 40)	USER_SPACE	N/A	[24,32)$\leq$4048



(a) allocation path (b) copy path

Figure 6: Exploration of allocation paths and memory copy paths.

marked as the allocation site of src (or dst). The relationship between the memory copy site and the allocation sites of src (or dst) is recorded in Copy2SrcAllocSet (or Copy2DstAllocSet).

Bridge Objects Identification. A bridge object must satisfy the following requirements: ❶ it contains a length field that controls the number of bytes to copy in a memory copy operation; ❷ the destination buffer of the memory copy is allocated in the kernel heap; and ❸ the source buffer of the memory copy is allocated in the kernel heap or originates from the user space.

Given a memory copy site $copy$, let $alloc_{src}$ be Copy2SrcAllocSet[$copy$] and $alloc_{dst}$ be Copy2DstAllocSet[$copy$], let $type_{sizeobj}$ be Copy2SizeObjType[$copy$] and $alloc_{sizeobj}$ be Type2AllocSet[$type_{sizeobj}$], If ❶ $alloc_{sizeobj}\neq\emptyset$, ❷ $alloc_{dst}\neq\emptyset$, ❸ $alloc_{src}\neq\emptyset$ or $alloc_{src}$ =USER_SPACE, then a kernel object of type $type_{sizeobj}$ is considered as a bridge object. In this case, we record the relationship between $type_{sizeobj}$ and $copy$ in Bridge2CopySet.

Router Objects Identification. A router object must satisfy the following requirements: ❶ it contains a pointer field that points to the destination buffer of a memory copy operation; ❷a it contains a pointer field that points to the source buffer of the memory copy operation or ❷b the source buffer of the memory copy originates from the user space.

Given a memory copy site $copy$, let $alloc_{src}$ be Copy2SrcAllocSet[$copy$], $alloc_{dst}$ be Copy2DstAllocSet [$copy$]; also let $type_{srcobj}$ be Copy2SrcObjType[$copy$], $type_{dstobj}$ be Copy2DstObjType[$copy$], and $alloc_{dstobj}$ be Type2AllocSet[$type_{dstobj}$]. If ❶ $alloc_{dstobj}\neq\emptyset$ and $alloc_{dst}\neq\emptyset$, ❷a $type_{srcobj}=type_{dstobj}$ and $alloc_{src}\neq\emptyset$, or ❷b $alloc_{src}$=USER_SPACE, then a kernel object of type $type_{dstobj}$ is considered as a router object. In this case, we record the relationship between $type_{dstobj}$ and $copy$ in Router2CopySet.

Constraint Extraction. Given a bridge or router object, we trace all paths to its memory copy sites, with particular attention to the pointer dereferences and the branching conditions in which the manipulated fields of the object are enclosed or has data dependency with the variable involved. The adversary needs to ensure that the pointer references a legitimate memory area and the branch conditions are satisfied. A constraint is of the form "range|op|value". For example, the constraint "[24,32)$\leq$4048" indicates that the value at offset [24,32) in the target object must be less than or equal to 4048. The constraint sets are statically collected by analyzing LLVM IR to determine the usage of object fields in each path leading to memory copy sites of objects, with special consideration given to the CmpInst instructions involving object fields, as implemented in ELOISE [16]. We record the relationship between a bridge (or router) object and the constraints of the corresponding memory copy paths in Bridge2CstSet (or Router2CstSet).

Example. Table 1 presents two examples of the bridge and router objects. The bridge object xfrm_algo_auth is allocated in the generic slab caches with slot size larger than 96 bytes. The field at offset [64, 68) in xfrm_algo_auth affects the $size$ argument of a memory copy operation. The source and destination buffers of the memory copy are allocated in the generic slab caches with slot sizes larger than 96 bytes and larger than 1k bytes, respectively. There are no constraints related to the fields of xfrm_algo_auth along the memory copy paths. The router object msg_msg is allocated in the generic slab caches with a slot size of 4k bytes. The field at offset [32, 40) in msg_msg affects the dst argument of a memory copy operation. The source buffer of the memory copy originates from the user space. The constraints along the memory copy paths require the value at offset [24, 32) in msg_msg is no more than 4048.

4.2. Triggering Allocation and Memory Copy

The aforementioned procedure statically identifies potential bridge and router objects, along with their corresponding allocation and memory copy sites, as well as the triggering system calls and the associated path constraints. For a bridge

```
1  // bridge object: xfrm_algo_auth
2  buf1 = alloc_and_fill_in_values(<controlled_values>);
3  buf2 = alloc_and_fill_in_values(<controlled_values>);
4  xfrm_sock = socket(AF_NETLINK, SOCK_RAW, NETLINK_XFRM);
5  sendmsg(xfrm_sock, buf1, 0); // alloc syscall
6  sendmsg(xfrm_sock, buf2, 0); // copy syscall
7
8  // router object: msg_msg
9  buf = alloc_and_fill_in_values(<controlled_value>);
10 id = msgget(IPC_PRIVATE, IPC_CREAT | 0666));
11 msgsnd(id, buf, BODY_SIZE_4K, 0); // alloc & copy syscall
```

Figure 7: Code snippet that triggers the allocation and memory copy of the bridge object `xfrm_algo_auth` (*xaa*) and the router object `msg_msg` (*msg*). Specifically, Line 5 triggers the allocation of *xaa*, Line 6 triggers the memory copy of *xaa*, and Line 11 triggers both the allocation and memory copy of *msg*.

or router object to be useful for exploitation, the adversary must prepare a testcase corpus that can sequentially trigger the allocation and memory copy of the object. While fuzzing is a natural approach to achieve this, the extensive search space renders blind fuzzing highly ineffective. To address this challenge, we propose a two-stage constraint-guided fuzzing approach. The first stage is designed to trigger the allocation of a bridge or router object and its associated source and destination buffers. The second stage focuses on triggering the memory copy sites, based on the testcases generated in the first stage. The constraints along the memory copy paths are used to guide mutations in the second stage. We instrument the kernel source code with customized monitor functions (refer to as *monitor anchors*) to signal when the allocation or memory copy sites of interest are reached.

Algorithm 1 describes how the triggering process works. Given a bridge or router object of type $type$, we obtain its allocation sites (Line 4), memory copy sites (Line 5) and the constraints along the memory copy paths (Line 6). For each memory copy site, we further obtain the allocation sites of the source and destination buffers (Lines 10-11). We retrieve the system calls that can reach the allocation and memory copy sites for the various kinds of interest (Lines 7-8 and 12-16). After these initial steps, we proceed with the fuzzing process. Separate testcase queues are maintained for the two stages: $queue\_stage1$ for `Stage1` (represents the allocation-triggering stage) and $queue\_stage2$ for `Stage2` (represents the copy-triggering stage). The fuzzing process continues until the resource limit is reached, with `Stage1` or `Stage2` randomly selected at each iteration (Lines 17-21).

The process logic of `Stage1` is as follows. First, the monitor anchors set at the allocation sites of interest are activated (Line 24), while those at other sites are deactivated (Line 23). Next, there is an equal chance of creating new testcases by mutating existing ones randomly selected from $queue\_stage1$ (Lines 26-27), or generating them from scratch by combining the system calls that can reach the allocation sites of interest (Lines 29-32). After executing the new testcase, the state feedback from the activated monitor anchors is observed (Line 33). If all kinds of allocation sites of interest are reached, the new testcase is added to $queue\_stage2$ (Lines 34-35). Otherwise, if only some kinds are reached, the new testcase is added to $queue\_stage1$ (Lines 36-37).

`Stage2` follows a process logic similar to `Stage1` with the following differences. First, the monitor anchors activated are those placed at the memory copy sites of interest (Lines 38-39). Second, the mutation of existing testcases is guided by the path constraints (Line 42). Third, a new testcase is generated by combining existing testcases with system calls that can reach the memory copy sites of interest (Lines 44-

Algorithm 1: Triggering allocation and memory copy.

```
Input  : type:   ObjType
Output : corpus: ObjType ↦ {Testcase}
Main :
1  queue_stage1 ← ∅,  queue_stage2 ← ∅,  corpus ← {}
2  alloc_syscall_set_src ← ∅,  alloc_syscall_set_dst ← ∅
3  alloc_syscall_set_obj ← ∅,  copy_syscall_set_obj ← ∅
4  alloc_set_obj ← Type2AlocSet[type]
5  copy_set_obj ← Bridge2CopySet(type) ⋃ Router2CopySet(type)
6  cst_set_obj ← Bridge2CstSet(type) ⋃ Router2CstSet(type)
7  for alloc_obj in alloc_set_obj do
8      alloc_syscall_set_obj ⋃= Alloc2SyscallSet[alloc_obj]
9  for copy in copy_set_obj do
10     alloc_set_src ← Copy2SrcAllocSet[copy]
11     alloc_set_dst ← Copy2DstAllocSet[copy]
12     for alloc_src in alloc_set_src do
13         alloc_syscall_set_src ⋃= Alloc2SyscallSet[alloc_src]
14     for alloc_dst in alloc_set_dst do
15         alloc_syscall_set_dst ⋃= Alloc2SyscallSet[alloc_dst]
16     copy_syscall_set_obj ⋃= Copy2SyscallSet[copy]
17 while NotReachResourceLimit() do
18     if queue_stage2 ≠ ∅ and TossCoin() = HEAD then
19         Stage2()
20     else
21         Stage1()
22 return corpus
   Procedure : Stage1()
23 DeactivateAllMonitorAnchors()
24 ActivateMonitorAnchors(alloc_set_src, alloc_set_dst, alloc_set_obj)
25 if queue_stage1 ≠ ∅ and TossCoin() = HEAD then
26     testcase ← RandomSelect(queue_stage1)
27     testcase ← RandomMutate(testcase)
28 else
29     alloc_syscall_src ← RandomSelect(alloc_syscall_set_src)
30     alloc_syscall_dst ← RandomSelect(alloc_syscall_set_dst)
31     alloc_syscall_obj ← RandomSelect(alloc_syscall_set_obj)
32     testcase ← Combine(alloc_syscall_src, alloc_syscall_dst, alloc_syscall_obj)
33 state ← MonitoredExecute(testcase)
34 if state ⊨ ALLOC_SRC and state ⊨ ALLOC_DST and state ⊨ ALLOC_OBJ then
35     queue_stage2 ⋃= testcase
36 else if state ⊨ ALLOC_SRC or state ⊨ ALLOC_DST or state ⊨ ALLOC_OBJ then
37     queue_stage1 ⋃= testcase
   Procedure : Stage2()
38 DeactivateAllMonitorAnchors()
39 ActivateMonitorAnchors(copy_set)
40 testcase ← RandomSelect(queue_stage2)
41 if TossCoin() = HEAD then
42     testcase ← ConstraintGuidedMutate(testcase, cst_set_obj)
43 else
44     copy_syscall_obj ← RandomSelect(copy_syscall_set_obj)
45     testcase ← Combine(testcase, copy_syscall_obj)
46 state ← MonitoredExecute(testcase)
47 if state ⊨ COPY then
48     corpus[type] ⋃= testcase
```

45), rather than being created from scratch. Finally, if the memory copy sites of interest are reached, the new testcase is added to the final result *corpus* (Lines 46-48).

Monitor Anchors Setup. We instrument kernel source code and insert a monitor anchor right behind each allocation site and memory copy site of our interest. With these anchors, when kernel execution reaches these sites, a feedback will be provided to the fuzzing program through `copy_to_user`. Note that the kernel is instrumented only once, while the inserted monitor anchors can be activated and deactivated on-demand at runtime.

Due to the complexity of the Linux kernel, various other kernel routines may also reach the instrumented sites, such as exception signals from processes, interrupt signals from peripheral devices, activities from other kernel threads or user-land processes. To eliminate the interference, we enhance our monitor anchors with the ability to determine whether their invocation originates from the system calls of interest. Specifically, within each monitor anchor, we check whether the value of the kernel variable `system_state` equals to `SYSTEM_RUNNING`, and if the `pid` of the current process matches that of the fuzzing program.

Fuzzing Strategy. In the first stage, we perform random mutation to add or remove system calls, and change the values of system call parameters. In the second stage, the focus shifts to modifying parameter values and repeating existing system calls, without adding new system calls or removing existing ones. During this stage, we probe which system call parameters affects the fields associated with the constraints, and prioritize the mutation of these parameters.

Example. In Figure 7, we present the code snippets that trigger the allocation and memory copy of the bridge object `xfrm_algo_auth` and the router object `msg_msg`. The first and second invocations of the `sendmsg` system call, with different buffer contents, result in the allocation and memory copy of an `xfrm_algo_auth` object, respectively. The allocation and memory copy of a `msg_msg` object occur sequentially via the invocation of the `msgsnd` system call.

4.3. Chaining and Exploring Capabilities

After identifying the bridge and router objects and generating a corpus of testcases that trigger the relevant allocation and memory copy sites, the next step is to chain the vulnerable object with the appropriate bridge and router objects for a given PoC and explore the potential capabilities. Simply combining the PoC with testcases that trigger the corresponding allocation and memory copy operations is insufficient, as various interferences may impede complete exploration of potential capabilities. The major interferences include: ❶ the uncertain order of allocation and memory copy operations on bridge and router objects caused by unscheduled system calls, and ❷ the uncertain memory layout introduced by heap fengshui techniques. The simulated environment eliminates uncertainty by directly enforcing the write capability on the bridge and router objects in sequence, which would otherwise require carefully crafted memory layout management and system call scheduling strategies in the real-world environment. By this design, we simplify the complex exploit generation into two distinct processes: capability exploration in a simulated environment, and exploitation synthesis in a real-world environment.

Algorithm 2: Chaining and exploring capabilities.

```
Input  : poc:                Testcase
         vulobj_type:        ObjType
         bridge_set:         {ObjType}
         router_set:         {ObjType}
         corpus:             ObjType ↦ {Testcase}
Output : exp_set:            {Testcase}
Main   :
1  exp_set ← ∅
2  ActivateSimulationAnchors(vulobj_type)
3  capability ← SimulatedExecute(NULL, poc)
4  DeactivateAllSimulationAnchors()
5  testcase2capability ← {poc ↦ capbility}
6  while NotReachResourceLimit() do
7      testcase_1 ← RandomSelect(KeySetOf(testcase2capability))
8      capability_1 ← testcase2capability[testcase_1]
9      if GetCapabilityType(capability_1) = CAP_UNCONTROL_OVERWRITE then
10         type ← RandomSelect(bridge_set)
11     else if GetCapabilityType(capability_1) = CAP_CONTROL_OVERWRITE then
12         type ← RandomSelect(router_set)
13     testcase_2 ← RandomSelect(corpus[type])
14     if Match(capability_1, type) then
15         Chain(testcase_1, testcase_2, type)
16 return exp_set
   Procedure : Chain(testcase_1, testcase_2, type)
17 capability_1 ← testcase2capability[testcase_1]
18 capability_ori ← capability_1
19 while True do
20     testcase_2 ← CapabilityGuidedMutate(testcase_2, capability_1)
21     ActivateSimulationAnchors(type)
22     capability_new ← SimulatedExecute(capability_1, testcase_2)
23     DeactivateAllSimulationAnchors()
24     if capability_new ⋡ capability_ori then
25         testcase_new ← Combine(testcase_1, testcase_2)
26         capability_ori ← capability_new
27     else
28         capability_new ← capability_ori
29         break
30 if GetCapabilityType(capability_new) = CAP_ARBITRARY_WRITE then
31     exp_set ⋃= testcase_new
32 else if capability_new ∉ ValueSetOf(testcase2capability) then
33     testcase2capability[testcase_new] = capability_new
```

We perform capability-guided fuzzing to generate prototype exploits. Algorithm 2 outlines the process for chaining and exploring capabilities. We maintain a global map $testcase2capability$, which records testcases along with their capabilities. Given a PoC, we first execute it in the simulated kernel environment to determine its capability (Lines 2-4) and record it in $testcase2capability$ (Line 5). Then, we perform the fuzzing process until the resource limit is reached (Lines 6-15). In each iteration, we randomly select a testcase from $testcase2capability$ (Line 7) and examine its current capability (Line 8). If the selected testcase has an uncontrolled overwrite capability (Line 9), we attempt to chain it with a randomly selected bridge object (Line 10). Otherwise, if the selected testcase already has a controlled overwrite capability (Line 11), we attempt to chain it with a randomly selected router object (Line 12). Before performing

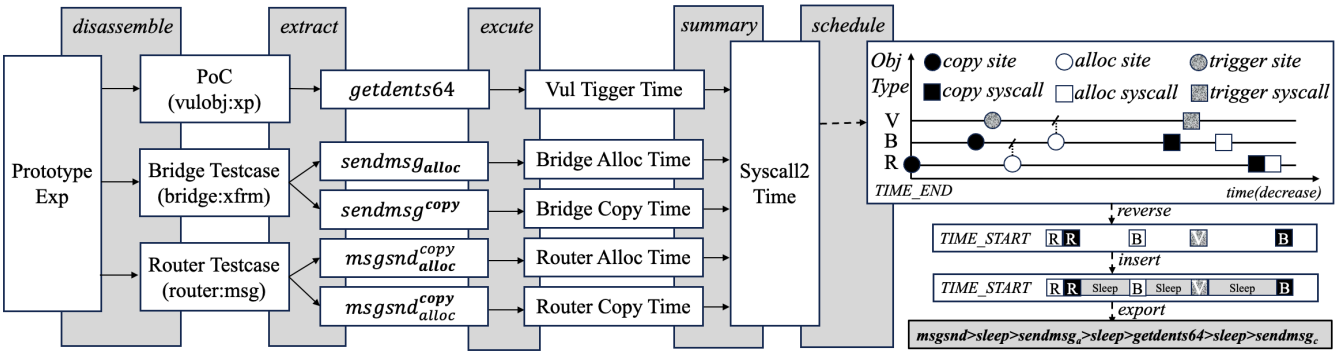


Figure 8: Example of the system call scheduling.

the chaining, we judges whether the capability matches the selected object (Lines 14-15).

The processing logic of chaining is as follows. First, we retrieve the capability of $testcase_1$ (Line 17) and record it as the original capability (Line 18). Then, we perform capability-guided mutation to explore the capability bound (Lines 19-29). Specifically, $testcase_2$ is mutated based on the analysis of historical mutations; and the capability of $testcase_1$ is enforced on the execution of mutated $testcase_2$ (line 20), simulating the capability achieved by executing the combination of $testcase_1$ and $testcase_2$. The exploration of the capability bound terminates when no stronger capability can be achieved through further mutations (Line 24). If the resulted capability allows for arbitrary memory write, then a prototype exploit is produced (Lines 30-31). Otherwise, if a previously unseen capability is observed, we record the combination of testcases along with the new capability (Lines 32-33). The simulation anchors are activated and deactivated before and after the simulated execution (Lines 21 and 23).

If multiple combinations of bridge and router objects are available for a given OOB vulnerability, we prioritize selecting those with critical fields positioned near the beginning (to minimize the impact on other fields during tampering), those with fewer constraints on the fields being tampered with (to simplify heap spraying and payload construction), and those whose allocation and copying occur in separate system calls (to expand the time window and increase the likelihood of success).

Simulation Anchors Setup. Simulation anchors are instrumented before each memory copy site of our interest. They can be activated on-demand at runtime to enforce a specified write capability on a given object and track the resulting write capability. To enforce a specified write capability, the memory copy operation on the given object is intercepted and modified to directly write the specific value at the designated offset of the given object, as indicated by the capability.

The resulting write capability is tracked by redirecting the memory copy operation of the given object to two distinct pre-allocated memory areas. One area is filled with byte values of 0x0, while the other is filled with byte values of 0xFF. By comparing the contents of these two memory areas after the redirection, we can determine the exact offset and value that the write capability has affected.

4.4. Synthesizing Exploitation

The prototype exploits produced in the previous stage are functional only within the simulated environment. The final stage is dedicated to synthesize a complete exploit that performs successfully in a real-world setting. This requires a careful design of memory management and system call scheduling strategies. The memory management strategy involves memory layout manipulation and memory content spraying, which are standard in kernel heap vulnerability exploitation [17], [22]. This subsection specifically focuses on the system call scheduling strategy.

Given a prototype exploit, we first decompose it into three distinct components, each corresponding to the vulnerable, bridge, and router objects. Next, we extract the system calls responsible for the allocation and memory copy operations for each component. We then execute the prototype exploit for multiple times, measuring the average execution time of each system call under consideration, from its invocation to the point where the allocation or memory copy sites of interest are reached. With this information, we can accurately schedule the system calls. In particular, we need to ensure: ❶ the allocation of the bridge object occurs before the triggering of the vulnerability; ❷ the allocation of the router object occurs before the memory copy on the bridge object; ❸ the triggering of the vulnerability, the memory copy on the bridge object, and the memory copy on the router object occur in sequential order. The allocation of the bridge object and router object, memory copy on bridge object and router object, and the triggering of the vulnerability do not need to be executed individually by unique system calls, as long as their order satisfies the three requirements. The corresponding system calls are then scheduled in accordance with these requirements.

Example. We illustrate system call scheduling using the running example, as depicted in Figure 8. The `getdents64` system call triggers the OOB vulnerability. The `msgsnd` system call performs both the allocation and memory copy of the router object simultaneously in a single invocation. The `sendmsg` system call performs the allocation and memory copy of the bridge object in different invocations, depending on the different parameters. Given the execution time of each system call (measured from its invocation to

the point where the allocation or memory copy sites of interest are reached or the target vulnerability is triggered) and the three requirements on the order of critical kernel operations, we place the corresponding events in the timeline in reverse chronological order to infer the invocation time of each system call. Finally, we determine the scheduling order for these system calls: $\texttt{msgsnd}^{copy}_{alloc} > \texttt{sendmsg}_{alloc} > \texttt{getdents64} > \texttt{sendmsg}_{copy}$. Here, the expression "$syscall_A > syscall_B$" means $syscall_A$ should be invoked before $syscall_B$. We use time stamp counter (referred to as sleep) to introduce a delay between consecutive system calls, ensuring sufficient time before executing the next one. The final exploit that works for the running example is shown in Appendix D.

```
typedef struct {
        unsigned off;
        unsigned len;
        void __user *val;
} param_t;

void *vulobj;

SYSCALL_DEFINE2(oob, unsigned, cmd, void *, param) {
  param_t p;
  copy_from_user(&p, param, sizeof(param_t));
  switch (cmd) {
    case ALLOC:
      vulobj = kmalloc(p.len, GFP_KERNEL); break;
    case WRITE:
      copy_from_user(vulobj+p.off, p.val, p.len); break;
  }
  return 0;
}
```

Figure 9: The crafted OOB write vulnerability.

5. Evaluation

We evaluate BRIDGEROUTER from three aspects.

- **Accuracy:** How accurate does BRIDGEROUTER identify the bridge and router objects? (§5.1)
- **Applicability:** Is BRIDGEROUTER applicable to generic caches of different sizes? (§5.2)
- **Effectiveness:** How effective does BRIDGEROUTER upgrade OOB vulnerability to the capability of arbitrary memory write? (§5.3)

We measure the accuracy by examining the false positives and false negatives. The applicability is evaluated on a crafted OOB write vulnerability that works for arbitrary size of generic caches, while a 14 real-world vulnerabilities are further evaluated and BRIDGEROUTER is compared with KOOBE [15] and SLUBStick [40]. Two case studies (§5.4) detail the effectiveness of BRIDGEROUTER.

Experimental Settings. All experiments are conducted on a machine with 13th Gen Intel® Core™ i9-13900 CPU and 32 GB memory. We choose Linux kernel v6.6 (the latest with long-term support) as the target kernel. The target kernel is run in QEMU, with multiple kernel defenses enabled, including KASLR, KCFI, KPTI and SMEP/SMAP. All vulnerabilities in the experiments are ported to the kernel.

5.1. Accuracy of Objects Identification

As discussed earlier, BRIDGEROUTER relies on static analysis to identify the bridge and router objects. Due to inherent limitations of static analysis, BRIDGEROUTER inevitably introduces false positives (i.e., mistakenly identifying certain objects as bridge or router objects) and false negatives (i.e., failing to identify actual bridge or router objects). We design an experiment to evaluate the false positives (FP) and false negatives (FN).

To evaluate FP, we apply an automated approach along with our manual effort. For an object identified as a bridge or router object by BRIDGEROUTER, we first use the fuzzing technique proposed in §4.2 to trigger its allocation and memory copy sites. If the fuzzing technique fails, we manually investigate whether the failure stems from inaccurate the static analysis or the limitation of the fuzzing technique. An identified object is classified as a FP only if both automated fuzzing and our manual review cannot find a concrete input that triggers its allocation and memory copy sites.

To evaluate FN, the ideal approach would be to review all the allocation and memory copy sites for each kernel object and manually determine whether it aligns with the definitions of the bridge and router objects as presented in §4.1. However, this is impractical due to the complexity and vast codebase of the Linux kernel. Therefore, we evaluate FN using a random sampling approach as [16]. Specifically, we randomly sampled 500 out of 2,373 heap-allocated kernel objects and manually identify all of its allocation and memory copy paths. This process takes two months of effort from an experienced Linux kernel code reviewer. We identify FN by comparing our manually analyzed results with those of BRIDGEROUTER.

Results. BRIDGEROUTER reports 44 bridge objects and 13 router objects in the target Linux kernel, of which we confirmed 37 true positives for bridge objects and 8 for bridge and router objects. Thus, the FP rates for bridge and router objects are 16% and 38%, respectively. By comparing these results with the randomly sampled and manually confirmed objects, we found that our manually confirmed objects are a subset of those identified by BRIDGEROUTER. While this finding does not directly conclude 0% FN rate, as the scale of the kernel limits our ability to manually audit all kernel objects, it suggests that the false negatives of BRIDGEROUTER are minimal. We argue that such performance is entirely satisfactory in practice. We believe that, for a potentially exploitable OOB vulnerability, BRIDGEROUTER can effectively identify the bridge and router objects needed to generate a complete exploit (see §5.3). The full list of accurately identified bridge and router objects can be found in Appendix C.

5.2. Applicability to Generic Caches

To evaluate the applicability of BRIDGEROUTER to generic caches of different sizes, we introduce a crafted OOB

TABLE 2: Applicable bridge and router objects for generic caches of different sizes.

VulObj Cache	Bridge	Router
kmalloc-8	-	-
kmalloc-16	xfrm_sec_ctx	msg_msg
kmalloc-32	kioctx_table	urb
kmalloc-64	kioctx_table	urb
kmalloc-96	xfrm_sec_ctx	msg_msg
kmalloc-128	ip6t_replace	hfsplus_sb_info
kmalloc-192	cfg80211_beacon_data	urb
kmalloc-256	snd_kcontrol	hfsplus_sb_info
kmalloc-512	cfg80211_beacon_data	ubifs_info
kmalloc-1k	xfrm_algo_auth	ubifs_info
kmalloc-2k	xfrm_algo_auth	ubifs_info
kmalloc-4k	xfrm_algo_auth	ubifs_info

write vulnerability into the Linux kernel. The vulnerable code allocates memory on-demand to fit any supported cache size, and allows for out-of-bounds writes. BRIDGEROUTER will upgrade the OOB write capability to an arbitrary memory write primitive. The code snippet of the crafted vulnerability is shown in Figure 9. Specifically, an additional system call `sys_oob` is created to manipulate the vulnerable object $vulobj$ (Line 7). The system call consists of memory allocation and write functionalities. In the allocation functionality (Lines 13-14), the length for allocation is obtained from the user-provided parameter $param$, and a memory chunk is allocated from an appropriate generic cache. In the write functionality (Lines 15-16), all the three arguments of `copy_from_user` are affected by the user space. With the two functionalities, the crafted vulnerability enables memory allocation from any generic cache (ranging from kmalloc-8 to kmalloc-4k) and allows for overwriting arbitrary values at any offset. Without loss of generality, the crafted vulnerability is configured to overwrite a fixed byte value (0xF0).

Results. For vulnerable objects residing in generic caches ranging from kmalloc-16 to kmalloc-4k, BRIDGEROUTER is able to find appropriate bridge and router objects to chain together, achieving arbitrary memory write. BRIDGEROUTER did not find appropriate bridge and router objects for vulnerable objects reside in kmalloc-8. We argue that the likelihood of an OOB vulnerable object being allocated from kmalloc-8 is very low, as the allocated chunk is quite small (no more than 8 bytes), while OOB vulnerabilities typically occur in larger objects. We inspected all public OOB vulenrabilities in recent years and found that fewer than 1% involve vulnerable objects allocated in kmalloc-8. Therefore, it is reasonable to claim that BRIDGEROUTER is applicable to generic caches of different sizes. Table 2 presents examples of applicable bridge and router objects for generic caches of different sizes. The "`VulObj Cache`" column indicates the cache size of the vulnerable object.

5.3. Effectiveness on Real-World Vulnerabilities

To further demonstrate the effectiveness of BRIDGEROUTER in exploiting real-world vulnerabilities, we perform a statistical analysis of 97 (74 from CVE and 23 from Syzbot) heap OOB write vulnerabilities discovered in the Linux kernel over the last 4 years (2021˜2024). Among these 97 vulnerabilities, 23 have publicly available reproducible PoCs, and 14 have publicly available exploitable Exps. We compare the effectiveness of BridgeRouter, KOOBE, and SLUBStick in exploiting the 23 vulnerabilities with public PoCs.

TABLE 3: Exploitation on real-world vulnerabilities. KB for KOOBE, SS for SLUBStick, and BR for BRIDGEROUTER. The percentages shown in the table represent the success rate of the exploitation, which is measured by running the exploit 100 times and repeating the process 10 times to compute the mean value.

ID	Capability	Exp	KB	SS	BR
CVE-2022-34918	kmalloc-64[0:48]=*	✔	✔	✔(<1%)	✔(36%)
CVE-2022-27666	kmalloc-4k[0:*]=*	✔	✔	✔(2%)	✔(23%)
CVE-2022-2639	kmalloc-4k[0:*]=*	✔	✔	✔(1%)	✔(27%)
CVE-2022-0995	kmalloc-96[0:32]=1bit	✔	✘	✔(3%)	✔(39%)
CVE-2022-0185	kmalloc-4k[0:*]=*	✔	✔	✔(6%)	✔(46%)
CVE-2021-42327	kmalloc-1k[0:*]=*	✔	✔	✘	✔(39%)
CVE-2021-42008	kmalloc-4k[0:*]=*	✔	✔	✔(2%)	✔(31%)
CVE-2023-6931	kmalloc-64[0:*]=rnd_val	✔	✘	✘	✔(21%)
CVE-2023-2598	kmalloc-*[0:*]=*	✔	✔	✔(13%)	✔(58%)
aa6df9d3... [42]	kmalloc-4k[0:*]=rnd_val	✘	✘	✘	✔(18%)
4f7a1fc5... [44]	kmalloc-96[0:*]=fix_val	✘	✘	✘	✔(23%)
dc3b1cf9... [43]	kmalloc-96[0:*]=*	✘	✘	✘	✔(33%)
797c55d2... [46]	kmalloc-96[0:32]=1bit	✔	✘	✔(3%)	✔(36%)
57028366... [45]	kmalloc-1k[0:*]=rnd_val	✘	✘	✘	✔(16%)

Two SOTA kernel OOB exploitation approaches (KOOBE and SLUBStick) are compared with BRIDGEROUTER. There are three noteworthy concerns. First, KOOBE is designed to hijack control flows rather than performing arbitrary memory writes. If KOOBE can modify the pointer in an adjacent object to a controlled value, we consider it capable of exploiting the vulnerability. Second, SLUBStick is not fully automated. Following the authors' instructions, we make every effort to generate exploits by implementing the techniques such as vulnerability pivoting, page recycling and reclaiming, as mentioned in their paper. Third, SLUBStick cannot exploit any vulnerability when the page table check defense is enabled. For the purposes of our evaluation, we disable this defense for SLUBStick.

Results. Table 3 shows the vulnerabilities and the results. The CVE vulnerabilities are labelled with their CVE IDs, and syzbot vulnerabilities use the commit hashes as their IDs. The "`Capability`" column describes one of the capabilities of the vulnerability. The last three columns denote the exploitability and success rate of the vulnerabilities using KOOBE, SLUBStick and BRIDGEROUTER. KOOBE succeeds in pointer modification in 7 vulnerabilities and SLUBStick can perform arbitrary memory writes in 8 vulnerabilities. In contrast, BRIDGEROUTER acquires the capability of arbitrary memory writes in 14 vulnerabilities.

Our automated capability upgrading approach may fail in any phase of exploring potential capabilities (Phase I), generating prototype exploits (Phase II), or synthesizing a complete exploit (Phase III). We perform step-by-step experimental analysis on the 23 heap OOB write vulnerabilities

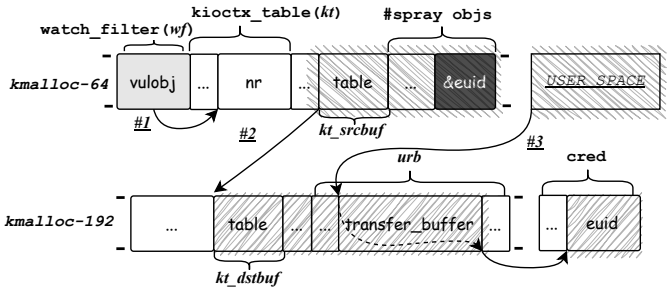


Figure 10: Memory layout for Case 1.

with reproducible PoCs, and found that 8 (35%), 1 (4%), 0 (0%) fail in Phase I, Phase II, Phase III, respectively.

We have also investigated why KOOBE and SLUBStick fail in some cases. The major goal of KOOBE is to hijack the control flow with diversified vulnerability capabilities. It requires that the victim object contains pointer fields and the vulnerabilities can take effect on the fields, modifying the original pointer to a self-defined one. Therefore, the data written via OOB must be user-controllable, making KOOBE behave poorly for cases with weak vulnerability capabilities.

SLUBStick, on the other hand, pivots an OOB vulnerability to a dangling pointer for exploitation. To this end, SLUBStick enforces the OOB capability on the objects with reference counters or pointers. For reference counter, the counter needs to be decreased by the OOB write, making the exploitation quite difficult when the modified value is relatively large. For pointer, certain bits of the pointer should be zeroed via an OOB write, i.e., the zeroing capability of the OOB vulnerability is required. Therefore, the requirement of pivoting an OOB vulnerability by SLUBStick can be induced as that sufficiently small values can be written via the OOB vulnerability. However, some vulnerabilities in Table 3 do not satisfy this requirement. Additionally, for some vulnerabilities, we are unable to find suitable objects for pivoting. These factors limit the applicability of SLUBStick.

For the exploitable cases of SLUBStick and BRIDGEROUTER, we further measure the success rate. Note that we measure the success rate of the complete exploitation chain, rather than individual components, such as cross-cache reuse in SLUBStick and cross-cache overwrite migration in BRIDGEROUTER. An exploit execution is considered successful if it achieves arbitrary memory writes. We ran each exploit 100 times to determine the success rate, which we repeated 10 times to compute the mean value. Overall, SLUBStick achieved a success rate of less than 4%. In comparison, BRIDGEROUTER achieved an average success rate of 32%, significantly outperforming SLUBStick.

5.4. Case Studies

We present two case studies to further demonstrate the effectiveness of BRIDGEROUTER. In these case studies, after upgrading the OOB vulnerability to enable arbitrary memory writes, we leverage the upgraded capability to modify the `euid` field of the `cred` structure to `GLOBAL_ROOT_UID`, thus gaining root privilege.

Case 1: CVE-2022-0995. The PoC triggers the vulnerability by writing an integer at a fixed position of a vulnerable object allocated in kmalloc-96 and set one bit in a limited scope to `1`. Our capability exploring technique (§4.3) reveals that the vulnerability also works in kmalloc-64.

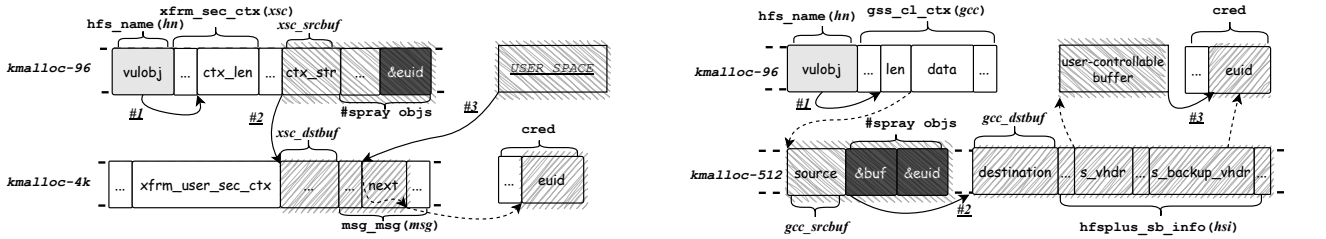
Because the data to be written is uncontrollable, the victim object's field cannot be manipulated to point to a specific address. As a result, KOOBE is unable to exploit the vulnerability to modify the pointer field in an adjacent object, preventing it from achieving control-flow hijacking. Through the vulnerability, SLUBStick can corrupt the reference counter of the `anon_vma_name` object allocated in kmalloc-96. By triggering the vulnerability twice, SLUBStick manipulates the reference counter to induce a double-free condition, which can be further exploited to obtain the capability of arbitrary memory writes. However, SLUBStick requires an additional pivot process, which makes the exploitation unstable. Moreover, the need to trigger the vulnerability twice further reduces the success rate, resulting in an overall success rate of only 3%.

As a contrast, BRIDGEROUTER can successfully exploits CVE-2022-0995 with a success rate of 39%. We select a `kioctx_table` object as the bridge object and a `urb` object as the router object. To carry out the exploitation, we construct the memory layout as in Figure 10 via heap feng shui techniques [17], [22]. The vulnerable object (wf) is adjacent to the bridge object (kt), and the destination buffer (kt_dstbuf) is adjacent to the router object (urb). Besides, multiple user-controllable buffer objects next to the source buffer (kt_srcbuf) can be heap sprayed.

When the vulnerability is triggered (#1 in Figure 10), the fifth bit of $kt{\rightarrow}nr$ is set to `1`, resulting in the number of bytes copied by `memcpy` exceeding the buffer size. The source buffer (kt_srcbuf) is allocated adjacent to the bridge object (kt) within the same slab cache, and its overflow content can be controlled by pre-sprayed values. Thus the `memcpy` operation will override $urb{\rightarrow}transfer\_buffer$ adjacent to the destination buffer (kt_dstbuf) with a controlled value (#2 in Figure 10). Finally, $urb{\rightarrow}transfer\_buffer$ serves as the destination for a `copy_from_user` operation (#3 in Figure 10). Since both the value of $urb{\rightarrow}transfer\_buffer$ and the user-space data for `copy_from_user` are user-controllable, arbitrary memory writes are achieved. Specifically, if an adversary overwrites $urb{\rightarrow}transfer\_buffer$ to point to $cred \rightarrow euid$ and set the user-space data to `GLOBAL_ROOT_UID`, root privilege will be obtained.

Case 2: dc3b1cf9111ab5fe98e7. For this syzbot vulnerability, the PoC leverages the vulnerable object allocated in kmalloc-96 to to perform an oversized string copy operation, which enables the overwriting of non-zero byte values. By using BRIDGEROUTER, we developed two exploits.

One exploit (Figure 11a) uses a `xfrm_sec_ctx` object as the bridge object (xsc), which is adjacent to the vulnerable object (hn) allocated in kmalloc-96. A `msg_msg` object is chosen as the router object (msg), adjacent to the destination



(a) `xfrm_sec_ctx` as bridge and `msg_msg` as router (b) `gss_cl_ctx` as bridge and `hfsplus_sb_info` as router

Figure 11: Memory layout for Case 2.

buffer (xsc_dstbuf). The exploit follows the same steps as Case 1 to gain the capability of arbitrary memory writes and to modify $cred \rightarrow euid$.

The other exploit (Figure 11b) takes a `gss_cl_ctx` object as the bridge object and a `hfsplus_sb_info` object as the router object. The vulnerable object (hn) is adjacent to the bridge object (gcc) in kmalloc-96, and a user-controllable object is sprayed after the source buffer (gcc_srcbuf). The destination buffer (gcc_dstbuf) is allocated just before the router object (hsi). After triggering the vulnerability (#1 in Figure 11b), $gcc \rightarrow len$ is overwritten with an oversized value. Then, a `memcpy` operation (#2 in Figure 11b) uses $gcc \rightarrow len$ as the size to copy, leading to the overwriting of $hsi \rightarrow s\_vhdr$ and $hsi \rightarrow s\_backup\_vhdr$ with pre-spayed values. An adversary can carefully craft pre-spayed values to make $hsi \rightarrow s\_vhdr$ point to a user-controllable kernel buffer (filled with the value of `GLOBAL_ROOT_UID`) and $hsi \rightarrow s\_backup\_vhdr$ point to $cred \rightarrow euid$. Finally, when another `memcpy` occurs (#3 in Figure 11b), the buffer pointed to by $hsi \rightarrow s\_vhdr$ will be copied to the buffer pointed to by $hsi \rightarrow s\_backup\_vhdr$, leading to the attainment of root privilege.

SLUBStick fails to exploit this vulnerability due to its limited write capability. SLUBStick provides two techniques for pivoting an out-of-bounds vulnerability to a double-free vulnerability. The first technique uses OOB vulnerability to modify an object's reference count and repeatedly triggering the vulnerability to bypass reference count protections. However, since this vulnerability cannot be triggered consecutively, the object can only be freed once, which prevents a double-free condition. The second technique requires corrupting a pointer within an object and zeroing out the two least significant bytes of the pointer. However, this vulnerability only allows for writing non-zero byte values, the corrupted pointer cannot reliably point to the start address of a page. This complicates the precise release of the victim object and hinders the ability to induce a double-free condition.

6. Discussion

Scientific Contribution of Our Work. Prior research works primarily leverage pivoting to elevate the capability of a limited vulnerability. In contrast, our work introduces a new approach for capability elevation in a gradual upgrading manner. Specifically, in this paper, we target kernel OOB write vulnerabilities and propose a solution that gradually upgrades an uncontrolled overwrite capability to a controlled overwrite capability, ultimately achieving an arbitrary memory write primitive. In theory, the concept of gradual capability upgrading can be applied to other types of vulnerabilities as well. Take the use-after-free (UAF) vulnerability as an example. After freeing the vulnerability object, we use a bridge object to occupy the freed slot. Then, by leveraging the UAF capability, we corrupt a field of the bridge object, causing it to use a larger length parameter during the copy operation. The subsequent process of chaining the router object is similar to the OOB exploitation method described in this paper. We leave the in-depth exploration of this possibility as future work.

Uniqueness of Our Work. We summarize our work to possess the following uniqueness. First, different with the current page-level attacks, we propose a novel slot-level cross-cache attack method. Second, our approach does not involve vulnerability pivoting. Instead, it enhances the vulnerability capability step by step and upgrades the restricted OOB capability to arbitrary memory write. Third, the critical objects for exploitation are automatically identified, rather than based on any prior knowledge. Fourth, we automate the full exploitation chain, not just a few core exponents.

Our work can be distinguished from SOTA works from difficult aspects. In terms of cross-cache attack, SLUBStick proposes page recycling and reclaiming to realize page-level cross-cache attacks, while BRIDGEROUTER realizes the slot-level cross-cache attack based on copy operations on bridge objects. Shortening the exploitation chain can decrease the influence of time windows. Regarding the usage of auxiliary objects, ELOISE [16] utilizes elastic objects to leak kernel data, SLUBStick [40] uses auxiliary objects to pivot vulnerabilities for cross-cache write, and the work [37] uses thanos objects to create the double-free state. Our work leverages bridge and router objects to gradually upgrade the vulnerability capability.

Comparison between BRIDGEROUTER and SLUBStick. Both BRIDGEROUTER and SLUBStick aim to elevate a limited heap vulnerability to an arbitrary memory write primitive. However, BRIDGEROUTER outperforms SLUBStick in the following aspects, making it more stable and reliable for vulnerability exploitation. In terms of capability upgrading, SLUBStick pivots different types of vulnera-

bilities into dangling pointers, and further leverages these dangling pointers to obtain a memory write primitive. This process requires the coordinated use of multiple auxiliary objects, which must remain simultaneously active within specific time windows to enable their interactions. In contrast, BRIDGEROUTER strategically establishes an exploit chain based on vulnerability capabilities, utilizing fewer auxiliary objects and imposing fewer temporal constraints. In terms of cross-cache attacks, SLUBStick relies on a heavy-weight page recycling and reclaiming mechanism to achieve page-level cross-cache attacks. In contrast, BRIDGEROUTER reuses existing kernel copy operations to achieve slot-level cross-cache attack, eliminating the uncertainties associated with the page recycling and reclaiming mechanism. In terms of arbitrary memory write, SLUBStick achieves it by tampering with page table entries to illegally map physical memory, which could be detected and prevented by the page table check security defense [7]. In contrast, BRIDGEROUTER achieves it by manipulating the dst parameter of normal kernel copy operations, bypassing page table checks entirely.

Characteristics of Bridge and Router Objects. Bridge and router objects assume different roles during the exploitation process. A bridge object transforms an uncontrolled overwrite capability (which writes to an adjacent location within one memory allocator cache) into a controlled overwrite capability (which writes to an adjacent location in a different memory allocator cache). Meanwhile, a router object directs the destination and/or the source buffers of a memory copy operation, ultimately enabling an arbitrary memory write primitive. We clarify that the bridge and router objects are representative objects that widely exist in the kernel codebase and can be leveraged for gradual capability upgrading. While we have made our best effort to generalize the definitions of bridge and router objects in terms of capability upgrading, it is possible that other types of objects operating similarly to bridge or router objects exist.

Mitigation. The main factors that impact the success of our exploitation approach include memory layout management (supported by the heap fengshui technique) and specific memory allocation and copy patterns (supported by the bridge and router objects). Mitigations can be designed by addressing these two factors. Recent Linux kernels provide defenses against the heap fengshui technique. Some of these defenses, such as shuffle page allocator and page table check, target page-level cross-cache attacks (as used by SLUBStick), do not impact slot-level cross-cache attacks (as used by our approach). Other defenses, such as slab freelist randomization and random kmalloc caches, may reduce the success rate of slot-level cross-cache attacks. However, these defenses incur significant performance overhead, hindering their adoption as default configuration. Moreover, some advanced adversary techniques have been proposed to bypass these defenses.

One possible mitigation is isolation for memory allocation, which could help prevent slot-level cross-cache attacks by segregating memory regions used by different objects. Different isolation solutions have been proposed, which differ in the grains of isolation. For example, XNU's kalloc_type [9] provides a type isolation, ensuring that once a particular address is used for a given type of object, only objects of that type can occupy that address for the lifetime of the program. While this approach works well for mitigating temporal memory safety issues (e.g., UAF), it fails to mitigate spatial memory safety issues (e.g., OOB). From version v5.14, Linux kernel moves specific objects from the generic caches to the accounted caches of the same size (e.g., msg_msg is moved from kmalloc-4k to kmalloc-cg-4k). Despite these efforts, we have still identified appropriate combinations of bridge and router objects in generic caches. In theory, segmenting each bridge and router object to a dedicated slab cache can defend against our attack approach. However, as many of these bridge and router objects are most widely used kernel structures, isolation all of them could result in significant performance overhead.

Another possible mitigation is to randomize the structure layout of bridge and router objects, or check the integrity of their critical fields. Structure layout randomization can hinder our attack by making it difficult to locate critical field offsets, thereby increasing the complexity of exploitation. Integrity checks for structure fields can be implemented using a canary-like mechanism, which places guard values before sensitive fields and verifies whether these values are altered whenever an associated memory copy occurs. This approach can help detect and prevent unauthorized modifications to critical fields, thereby mitigating OOB-based attacks. Again, since bridge and router objects are widely used within the kernel, these mitigations (which involve changing the implementation of these objects) may introduce additional memory fragmentation.

7. Related Work

OOB Exploitation. The most relevant work is the exploitation of heap OOB vulnerability. Pivoting the vulnerability is one common approach to OOB exploitation. ELOISE [16] identifies elastic kernel objects on popular OSes, which can pivot OOB write to arbitrary read in the kernel. SLUBStick [40] converts the given OOB vulnerability into a double-free vulnerability and uses dangling pointer to manipulate the page table, granting the capability of arbitrary memory read and write. Similarly, DirtyCred [35] pivots the vulnerability to get a dangling pointer, perform heap spray and occupy the freed spot with a high-privileged credential object, to escalate privilege. Some others like KOOBE [15] focus on capability extraction and OOB vulnerability evaluation automation.

Kernel Exploitation. Heap spraying [17], [18] is critical in kernel vulnerability exploitation. Page-level cross-cache techniques [33], [40], [51] are proposed to handle the circumstance when victim and vulnerable objects are not in the same cache. Recently, side-channel attacks against the SLUB allocator [29], [40] can make heap allocation more predictable. Return-oriented programming (ROP) [27], [28] modifies the control flow to user space or directly mapped memory. Data-oriented programming [24], [25] identifies data oriented gadgets and chains disjoint gadgets in an expected

order to realize attacks. The techniques that can bypass kernel mitigation are also paid attention to. Researchers employ side-channel attacks to leak the kernel information [13], [21], [26], to bypass KASLR. Meltdown [36] leverages out-of-order execution and side-channels on modern processors to bypass SMEP/SMAP from a user space program.

Exploit Automation. Automated exploitation has long been studied [11], [20], [23], [53]. Brumley et al. [12] first introduce the automatic patch-based exploit generation problem, using the patch to identify the vulnerability point and then generating corresponding inputs to trigger it. Several studies [10], [14] automate the exploit writing pipeline for stack-based buffer overflow. For automated heap vulnerability exploit generation, Repel et al. [41] leverage concolic execution to search for exploitation primitives; Revery [48] and HeapHopper [19] employ symbolic execution and fuzzing to discover IP hijacking and heap allocation primitives. Extensive expertise is usually required for kernel exploitation. ExpRaise [30] raises interrupts to expand time windows, increasing the success rate of race-condition exploitation. FUZE [50] and TAODE [37] analyze the root cause of UAF vulnerabilities and then automatically generate exploits.

8. Conclusion

In this paper, we propose a practical approach for kernel OOB exploitation to achieve arbitrary memory writes. Two special kinds of kernel objects, i.e., bridge and router, are automatically identified from the kernel, to chain the vulnerable object and a victim object. Associated allocation and memory copy operations can be then leveraged to achieve arbitrary memory writes. A prototype system, BRIDGEROUTER, is developed to automate the whole procedure of exploit generation. Experiments have demonstrated the accuracy of BRIDGEROUTER in identifying bridge and router objects and the effectiveness of BRIDGEROUTER in upgrading OOB vulnerabilities to arbitrary memory writes.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments. The work is supported in part by National Natural Science Foundation of China (NSFC) under grants 62272465 and 62272464, the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29, and Public Computing Cloud, Renmin University of China.

References

[1] Attack methodology of slubstick. https://github.com/CheUhxg/BridgeRouter/blob/main/comparison.pdf.

[2] Buddy memory allocation. https://en.wikipedia.org/wiki/Buddy_memory_allocation.

[3] Escaping containers using the dirty pipe vulnerability. https://securitylabs.datadoghq.com/articles/dirty-pipe-container-escape-poc.

[4] Exploit by blasting cred. https://github.com/CheUhxg/BridgeRouter/blob/main/exploits/blast_cred/exploit.c.

[5] Exploiting kernel races through taming thread interleaving. https://i.blackhat.com/USA-20/Thursday/us-20-Lee-Exploiting-Kernel-Races-Through-Taming-Thread-Interleaving.pdf.

[6] Linux kernel: Product details, threats and statistics. https://www.cvedetails.com/product/47/Linux-LinuxKernel.html?vendor_id=33.

[7] Page table check. https://www.kernel.org/doc/html/v5.17/vm/page_table_check.html.

[8] Slab allocation. https://en.wikipedia.org/wiki/Slab_allocation.

[9] Towards the next generation of xnu memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/.

[10] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[11] Teresa Nicole Brooks. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. *CoRR*, abs/1702.06162, 2017.

[12] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (SP 2008), 18-21 May 2008, Oakland, California, USA*, pages 143–157. IEEE Computer Society, 2008.

[13] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: break it, fix it, repeat. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 481–493. ACM, 2020.

[14] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394. IEEE Computer Society, 2012.

[15] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. KOOBE: towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1093–1110. USENIX Association, 2020.

[16] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1165–1184. ACM, 2020.

[17] Yueqi Chen and Xinyu Xing. SLAKE: facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1707–1722. ACM, 2019.

[18] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 327–336. ACM, 2010.

[19] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 99–116. USENIX Association, 2018.

[20] Behrad Garmany, Martin Stoffel, Robert Gawlik, Philipp Koppe, Tim Blazytko, and Thorsten Holz. Towards automated generation of exploitation primitives for web browsers. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 300–312. ACM, 2018.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 368–379. ACM, 2016.

[22] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 763–779. USENIX Association, 2018.

[23] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1689–1706. ACM, 2019.

[24] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 177–192. USENIX Association, 2015.

[25] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 969–986. IEEE Computer Society, 2016.

[26] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 191–205. IEEE Computer Society, 2013.

[27] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 957–972. USENIX Association, 2014.

[28] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 459–474. USENIX Association, 2012.

[29] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing side-channel based linux kernel heap exploitation technique. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 6825–6842. USENIX Association, 2023.

[30] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. Exprace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2363–2380. USENIX Association, 2021.

[31] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2201–2215. ACM, 2017.

[32] Runhao Li, Bin Zhang, Jiongyi Chen, Wenfeng Lin, Chao Feng, and Chaojing Tang. Towards automatic and precise heap layout manipulation for general-purpose programs. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[33] Zhenpeng Lin. How autoslab changes the memory unsafety game. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game.

[34] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1963–1976. ACM, 2022.

[35] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1963–1976. ACM, 2022.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[37] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. From release to rebirth: Exploiting thanos objects in linux kernel. *IEEE Trans. Inf. Forensics Secur.*, 18:533–548, 2023.

[38] William Liu, Joseph Ravichandran, and Mengjia Yan. Entrybleed: A universal KASLR bypass against KPTI on linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2023, Toronto, Canada, 29 October 2023*, pages 10–18. ACM, 2023.

[39] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[40] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. Slubstick: Arbitrary memory writes through practical software cross-cache attacks within the linux kernel. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[41] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*, pages 25–35. ACM, 2017.

[42] Syzbot. Kasan: slab-out-of-bounds write in diwrite. https://syzkaller.appspot.com/bug?extid=aa6df9d3b383bf5f047f.

[43] Syzbot. Kasan: slab-out-of-bounds write in hfs_asc2mac. https://syzkaller.appspot.com/bug?extid=dc3b1cf9111ab5fe98e7.

[44] Syzbot. Kasan: slab-out-of-bounds write in hfs_bnode_read_key. https://syzkaller.appspot.com/bug?extid=4f7a1fc5ec86b956afb4.

[45] Syzbot. Kasan: slab-out-of-bounds write in hfsplus_bnode_read_key. https://syzkaller.appspot.com/bug?extid=57028366b9825d8e8ad0.

[46] Syzbot. Kasan: slab-out-of-bounds write in watch_queue_set_filter. https://syzkaller.appspot.com/bug?id=797c55d2697d19367c3dabc1e8661f5810014731.

[47] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. Alphaexp: An expert system for identifying security-sensitive kernel objects. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4229–4246. USENIX Association, 2023.

[48] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1914–1927. ACM, 2018.

[49] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. KEPLER: facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1187–1204. USENIX Association, 2019.

[50] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 781–797. USENIX Association, 2018.

[51] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 414–425. ACM, 2015.

[52] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2139–2154. ACM, 2017.

[53] Bin Zhang, Jiongyi Chen, Runhao Li, Chao Feng, Ruilin Li, and Chaojing Tang. Automated exploitable heap layout generation for heap overflows through manipulation distance-guided fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4499–4515. USENIX Association, 2023.

Appendix

A. Ethics Consideration

The Linux kernel exploitation approach proposed in this paper is fundamentally aimed at strengthening the security of the Linux kernel, rather than providing tools for malicious purposes. We used the exploit data solely for experimental purposes via an anonymous repository and ensured no public disclosure prior to our testing. Additionally, all experiments were conducted in a virtual environment, eliminating any potential risks to public or live systems. We do not disclose any details that could endanger the Linux kernel or its users.

B. Allocation and Memory Copy Functions

Table 4 lists the kernel functions responsible for memory allocation and memory copy operations. The functions `kmem_cache_zalloc()` and `kmem_cache_alloc()` allocate kernel objects on dedicated slab caches, while the other allocation functions allocate kernel objects on generic slab caches. The function `memcpy()` copies data within the kernel space, while the other memory copy functions transfer data from the user space to the kernel space.

TABLE 4: Kernel functions for allocation and memory copy.

Category	Functions
Allocation	kmalloc(); kzalloc(); kcalloc(); kmalloc_node(); kmalloc_array_node(); kcalloc_node(); kzalloc_node();
Memory Copy	memcpy(); copy_from_user(); copyin(); skb_put_data(); skb_copy_to_linear_data_of();

C. Bridge and Router Objects

Table 5 shows the full list of accurately identified bridge and router objects. The columns from left to right are the object category, object type, slab caches where the object is allocated, the offset of the critical field, slab caches for the source and destination buffers, and the path constraints. We do not claim that Table 5 provides an exhaustive list of all bridge and router objects. It is possible that some bridge and router object are missed due to the limitations of our static analysis and fuzzing approaches. However, the bridge and router objects listed in Table 5 are sufficient to cover all generic caches of different sizes except for kmalloc-8.

We have also conducted evaluations on two additional versions: v5.15 (the latest long-term support v5.x version) and v4.20 (the latest stable v4.x version). We have successfully identified potential bridge and router candidates across all the three versions, including 85, 88, 44 bridge candidates in v4.20, v5.15, v6.6, respectively; and 23, 20, 13 router candidates in v4.20, v5.15, v6.6, respectively. We have the following observations. First, the combinations of bridge and router objects for generic caches of different sizes listed in Table 2 are available across different versions. Second, different versions may have some unique bridge or router objects, e.g., blk_mq_tag_set serves as a bridge for v5.15 and tty_struct serves as a router for v4.20. Third, same object may play different roles in different versions, e.g., ubifs_inode serves as a bridge in v5.15, but can serve as both a bridge and a router in v4.20.

D. Simplified Exploit for Running Example

Figure 12 presents the final Exp for the running example in §3.1. For clarity, the code is simplified, with tedious details omitted.

```
1  void * race_page;
2  struct msghdr msg;
3  int xfrm_socket, fd, fuse;
4  int pipe[2];
5  fuse = open(...)
6  race_page = mmap(0x1338000, 0x1000, fuse, ...);
7  sleep(...);
8  /* Alloc Router Object*/
9  msgsnd(, (void*) (race_page - 8), 4056);
10 xfrm_socket = init_xfrm_socket();
11 craft_of_msg1(msg);
12 sleep(...);
13 /* Alloc Bridge Object */
14 sendmsg(xfrm_socket, &msg, 0);
15 craft_of_fd(fd);
16 sleep(...);
17 /* Alloc and Copy Vulnerable Object */
18 syscall(__NR_getdents64, fd, &ent, 0x5dul);
19 craft_of_msg2(msg);
20 sleep(...);
21 /* Copy Bridge Object */
22 sendmsg(xfrm_socket, &msg, 0);
23 sleep(...);
24 /* Copy Router Object */
25 write(pipe[1], ...);
```

Figure 12: Simplified exploit of running example

TABLE 5: The full list of accurately identified bridge and router objects.

Category	Struct	Caches	Offset (len/ptr)	Source Buffer	Destination Buffer	Constraints
Bridge	xfrm_algo_auth	≥ kmalloc-96	[64, 68)	≥ kmalloc-96	≥ kmalloc-1k	∅
Bridge	xfrm_sec_ctx	≥ kmalloc-16	[2, 4)	≥ kmalloc-16	kmalloc-4k	∅
Bridge	ip_options_rcu	≥ kmalloc-32	[24, 25)	≥ kmalloc-32	≥ kmalloc-32	[0, 8) != null, [8, 16) == kaddr
Bridge	cfg80211_ibss_params	kmalloc-512	[56, 57)	kmalloc-512	kmalloc-8k	[0,8) != null, [8, 16) != null, [16, 24) != null, [24, 28) ¡ 14
Bridge	cfg80211_connect_params	kmalloc-512	[40, 48)	kmalloc-512	kmalloc-8k	[0,8) != null, [8, 16) != null, [16, 24) != null , [24, 32) != null,
Bridge	kioctx_table	≥ kmalloc-32	[16, 20)	≥ kmalloc-32	≥ kmalloc-64	∅
Bridge	cfg80211_scan_request	≥ kmalloc-1k	[32, 40)	kmalloc-512	≥ kmalloc-1k	[0, 8) != null, [24, 32) != null
Bridge	ext4_xattr_info	≥ kmalloc-32	[16, 24)	≥ kmalloc-32	kmalloc-4k	[0, 8) != null, [8, 16) != null
Bridge	cfg80211_beacon_data	≥ kmalloc-192	[96, 104)	≥ kmalloc-192	≥ kmalloc-192	[0,8) != null, [8, 16) != null, [16, 24) != null, . . .
Bridge	gss_cl_ctx	≥ kmalloc-192	[48, 52)	≥ kmalloc-192	≥ kmalloc-192	[24, 32) != null, [40, 48) != null
Bridge	xhci_segment	kmalloc-64	[44, 48)	≥ kmalloc-8	≥ kmalloc-8	[0,8) != null, [8, 16) != null, [16, 24) == kaddr, [24, 32) == kaddr,
Bridge	ip6t_replace	≥ kmalloc-128	[44, 48)	≥ kmalloc-128	≥ kmalloc-64	∅
Bridge	ieee80211_if_ibss	kmalloc-8k	[120, 121)	kmalloc-8k	kmalloc-512	[24, 32) == kaddr, [56, 64) == kaddr
Bridge	public_key	kmalloc-64	[4, 8)	≥ kmalloc-8	≥ kmalloc-16	∅
Bridge	snd_kcontrol	≥ kmalloc-192	[0, 4)	≥ kmalloc-256	≥ kmalloc-192	∅
Bridge	xfrm_policy	kmalloc-1k	[372, 373)	kmalloc-1k	kmalloc-1k	[0,8) != null, [8, 16) != null, [16, 24) != null,. . .
Bridge	nfs_fh	kmalloc-192	[0, 2)	kmalloc-192	kmalloc-192	∅
Bridge	hid_device	kmalloc-8k	[40, 44)	≥ kmalloc-32	≥ kmalloc-32	[0, 8) != null, [16, 24) != null, [32, 40) != null
Bridge	hid_report	kmalloc-4k	[2124, 2128)	≥ kmalloc-64	≥ kmalloc-64	[0,8) != null, [8, 16) != null, [16, 24) != null, . . .
Bridge	nfs_client	kmalloc-1k	[160, 168)	kmalloc-1k	kmalloc-1k	∅
Bridge	property_entry	≥ kmalloc-32	[1, 2)	≥ kmalloc-16	≥ kmalloc-16	∅
Bridge	svc_rqst	buddy_page	[168, 176)	buddy_page	≥ kmalloc-512	[16, 24)!=null, [24, 32) != null, [32, 40) != null
Bridge	ip_options	kmalloc-64	[3, 4)	kmalloc-64	kmalloc-2k	∅
Bridge	ip_options_rcu	kmalloc-64	[3, 4)	kmalloc-64	kmalloc-64	∅
Bridge	fdtable	kmalloc-64	[0, 4)	≥ kmalloc-8	≥ kmalloc-8	∅
Bridge	ipt_replace	≥ kmalloc-128	[40, 44)	≥ kmalloc-128	≥ kmalloc-64	∅
Bridge	ipv6_rpl_sr_hdr	≥ kmalloc-16	[1, 2)	≥ kmalloc-16	≥ kmalloc-2k	∅
Bridge	netprio_map	≥ kmalloc-128	[16, 20)	≥ kmalloc-128	≥ kmalloc-128	[0, 8) != null, [8, 16) == kaddr
Bridge	sock_reuseport	≥ kmalloc-2k	[20, 22)	≥ kmalloc-2k	≥ kmalloc-2k	[0, 8) != null, [8, 16) == kaddr
Bridge	cfg80211_conn	kmalloc-8k	[40, 48)	kmalloc-8k	≥ kmalloc-256	[0,8) != null, [8, 16) != null, [16, 24) != null , [24, 32) != null,
Bridge	fuse_io_args	kmalloc-256	[184, 188)	≥ kmalloc-16	≥ kmalloc-16	[160, 168) == kaddr, [168, 176) != null, [176, 184) != null
Bridge	hbucket	≥ kmalloc-64	[24, 25)	≥ kmalloc-64	≥ kmalloc-64	[0, 8) != null, [8, 16) == kaddr
Bridge	tcp_md5sig_key	kmalloc-192	[0, 4)	kmalloc-192	≥ kmalloc-1k	∅
Bridge	ceph_snap_realm	kmalloc-256	[88, 92)	≥ kmalloc-64	≥ kmalloc-64	[8, 16) != null, [32, 40) != null, [40, 48) != null, [80, 88) != null]
Bridge	p9_conn	kmalloc-512	[104, 108)	kmalloc-512	kmalloc-8k	[0,8) != null, [8, 16) != null, [16, 24) != null ,. . .
Bridge	jffs2_sum_dirent_mem	≥ kmalloc-64	[30, 31)	≥ kmalloc-64	≥ kmalloc-32	[0, 8) != null
Router	msg_msg	kmalloc-4k	[32, 40)	USER_SPACE	N/A	[24,32)≤ 4048
Router	msg_msgseg	kmalloc-4k	[0, 8)	USER_SPACE	N/A	∅
Router	seq_file	seq_file_cache	[0, 8)	USER_SPACE	N/A	∅
Router	urb	kmalloc-192	[96, 104)	USER_SPACE	N/A	[8, 16) != null, [24, 32)!=null [32, 40) != null, . . .
Router	xfrm_state	xfrm_state_cache	[432, 440)	N/A	N/A	[0,8) != null, [8, 16) != null, [16, 24) != null , . . .
Router	ubifs_info	kmalloc-4k	[2720, 2728)	N/A	N/A	[0,8) != null, [8, 16) != null, [152, 160) != null , . . .
Router	btrfs_fs_info	kmalloc-4k	[776, 768)	N/A	N/A	[24,32) != null, [32, 40) != null, [40, 48) != null ,. . .
Router	hfsplus_sb_info	kmalloc-512	[24, 32)	N/A	N/A	[0,8) != null, [8, 16) != null, [16, 24) != null,. . .

E. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary. The paper presents an exploitation technique for the Linux Kernel, designed to transform ”uncontrolled” Out-of-Bounds (OOB) vulnerabilities (in which the attacker does not control the values being written) into Arbitrary Memory Writes. The proposed methodology involves identifying two specific types of kernel objects: Bridges and Routers. Specifically, it utilizes a combination of static analysis and fuzzing to find suitable Bridge and Router objects and determine how to use them as part of an exploit. The paper shows how the proposed methodology can be applied to exploit synthetic and real-world bugs.

E.2. Scientific Contributions.

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

E.3. Reasons for Acceptance.

1) The development of an automated system for generating exploits for Linux kernel vulnerabilities advances the current state of the art in Automatic Exploit Generation (AEG). Future work in this area could build upon the proposed techniques.
2) The paper introduces a novel technique for exploiting vulnerabilities in the Linux kernel, effectively demonstrating the exploitability of multiple real-world bugs.
3) By introducing a novel exploitation technique that incorporates a combination of static and dynamic analysis, the paper marks a substantial progression in the AEG field.