CIVSCOPE: Analyzing Potential Memory Corruption Bugs in Compartment Interfaces

Yi Chien
Rice University
Houston, US

Vlad-Andrei Bădoiu
University Politehnica of Bucharest
Bucharest, Romania

Yudi Yang
Rice University
Houston, US

Yuqian Huo
Rice University
Houston, US

Kelly Kaoudis
Trail of Bits
New York, US

Hugo Lefeuvre
The University of Manchester
Manchester, UK

Pierre Olivier
The University of Manchester
Manchester, UK

Nathan Dautenhahn
Rice University
Houston, US

Abstract

Compartmentalization decomposes a program into separate parts with mediated interactions through compartment interfaces—hiding information that would otherwise be accessible from a compromised component. Unfortunately, most code was not developed assuming its interfaces as trust boundaries. Left unchecked, these interfaces expose confused deputy attacks where data flowing from malicious inputs can coerce a compartment into accessing previously hidden information on-behalf-of the untrusted caller.

We introduce a novel program analysis that models data flows through compartment interfaces to automatically and comprehensively find and measure the attack surface from compartment bypassing data flows. Using this analysis we examine the Linux kernel along diverse compartment boundaries and characterize the degree of vulnerability. We find that there are many compartment bypassing paths (*395/4394* driver interfaces have *22741* paths), making it impossible to correct by hand. We introduce CIVSCOPE as a comprehensive and sound approach to analyze and uncover the lower-bound and potential upper-bound risks associated with the memory operations in compartment boundary interfaces.

ACM Reference Format:
Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. 2023. CIVSCOPE: Analyzing Potential Memory Corruption Bugs in Compartment Interfaces. In *Workshop on Kernel Isolation, Safety and Verification (KISV '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3625275.3625399

KISV '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0411-6/23/10.
https://doi.org/10.1145/3625275.3625399

1 Introduction

Monolithic applications comprise many components that only require access to a small amount of runtime information. Without internal isolation, a buggy or outright malicious component can access any state within the environment, a violation of the principle of least privilege [22]. Compartmentalization [28] [29], addresses this problem by decomposing the system into separate parts, allowing cooperation through data sharing and RPC like interfaces. Unfortunately, a compartment may export interfaces that allow data flow paths from interface arguments to internal memory operations, letting untrusted callers influence pointers used in read and write operations [3–5, 10, 18, 26]. These Compartment Interfaces Vulnerabilities (CIVs) [10], operate as confused deputies allowing abuse of a compartment's privileges, completely bypassing compartment boundaries.

While compartmentalization has been a topic of exploration for decades, little is known about how to measure the potential threat of CIVs, the degree of CIVs for compartmentalized systems, or the severity of any given CIV. One solution is to use the type system to automatically prevent classes of CIVs [18] [14], but they neglect a wide range of software written in unsafe languages (e.g. operating systems, hypervisors, etc.). Recent work uses dynamic analysis on unsafe code [5, 10], but fall short for broader generalizations. They merely establish a *lower bound* for the number of CIVs and lack a comprehensive method to assess the attack surface or measure severity. A comprehensive characterization requires a sound and complete method.

In this work, we aim to systematically and comprehensively characterize the degree of compartment bypassing data flows for compartments (code, data, and interfaces) in complex software environments. Through a taint based static analysis we are able to identify paths leading from compartment interfaces to memory operations. The value of this objective is that we can use this methodology to estimate whether compartmentalization is even feasible and to what degree interface complexity requires significant refactoring. Further, our work opens the door for comparing between compartment configurations.

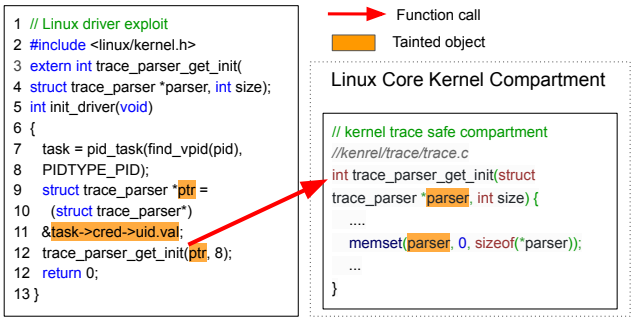

Figure 1. A malicious driver exploits the exported interface of isolated trace system and set uid to 0 for privilege escalation.

Our first contribution is a model and framework to automatically find all instances of the CMDP compartmentalization anti-pattern. An CMDP is a path from a compartmentalized interface that *may* influence the addresses, sizes, or indices of memory operations. This anti-pattern covers a broad range of compartmentalization weaknesses that can lead to exploitation.

Our second contribution extends the CMDP analysis to find unprotected paths. Some data flows from interfaces to sensitive memory operations may have legitimate safeguards, such as bounds checks before array indexing, that prevent exploitation. Identifying unprotected paths exposes directly exploitable flows that the developer must protect.

Our third contribution is a taxonomy for measuring the severity of each CMDP, which enables systematic analysis and measurement, and indicates the most powerful patterns for exploitation, such as the getter/setter pattern introduced.

Our results indicate a large number of CMDPs in Linux drivers and the core Linux kernel. In particular, the core linux kernel exposes *648* interfaces, with *68* having a CMDP. These *68* interfaces contribute to a total of *24,900* unconditioned paths out of *50,300* paths, which is nearly *49.5%* of paths being directly exploitable. It's worth noting that the paths include all possible CMDPs, even when starting from the same source and ending at the same sink. Drivers expose *395* CMDP interfaces out of a total of *4,394*. The *395* interfaces account for a total of *19,555* out of *42,379*, nearly *46.1%* of which are unconditioned.

2 Background and Threat Model

A *compartmentalization* is an assignment of program objects and code to execution contexts that have full access to their own code and data and sharing policies for exposing interfaces and objects to external compartments [1, 2, 4, 19, 27, 29]. Prior approaches have explored this model, such as KSplit [6] for driver isolation in the Linux kernel, and other recent proposals [8, 9, 11–13, 15, 17, 20, 21, 23, 24] aimed at introducing isolation into the monolithic kernel. As indicated by prior work, compartment interfaces expose dependencies that can allow control over the compartment from others through compartment interface vulnerabilities (CIVs) [10, 18].

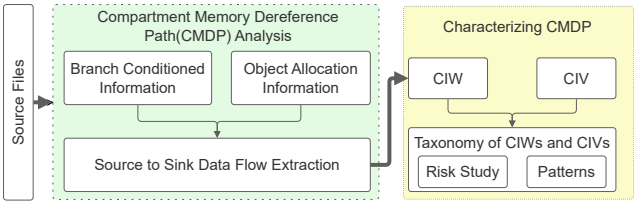

Figure 2. CIVScope Architecture: Source code is compiled into Compartment Memory Dereference Paths. The new Compartment Interface Weakness taxonomy maps CIWs to CIVs and is used to characterize and measure threats.

The goal of this paper is to provide a framework for understanding and measuring the threat exposure provided by a given compartment's externally accessible state through its interfaces. As such our *threat model* emphasizes a compartment centric view as opposed to a whole-system compartmentalization. This is necessary because many cases of compartmentalization might not have access to source code linking the objects or as systems evolves the callers and sharing policies might change. As such, we assume that an attacker may call any one of a compartment's exposed interfaces and provide any data object as an argument to the interface in an attempt to abuse the privileges of the called compartment to modify its internal objects. We do not consider privileges to data outside of a compartment or control that may be exerted over it through its externally accessible objects.

As an example, consider the Linux core kernel as a compartment, including all code and data in the top level kernel directory. Leveraging the exported trace interface (as depicted in Figure 1), *trace_parser_get_init*, a malicious device driver can forge a pointer to the *task_struct* object and ask *trace_parser_get_init* to set the uid to 0 on behalf of the interface. Since the *ftrace* system has the authority to access any object inside the core kernel, any driver can get root privilege and escalate through this confused deputy interface.

While this example clearly makes a poor choice by including the tracing system into the core kernel, such paths are known to exist in many interfaces [16] and we must have a systematic way of exposing them so at the very least these interfaces are not chosen as compartment boundaries. Beyond knowing what not to choose, it would be best if we provided a systematic characterization of such data flows to expose the aggregate surface area of these exploits as well as measure the power of each type of attack.

3 Model and Static Analysis

The CMDP analysis identifies all data flows from compartment interfaces to sensitive memory dereferencing operations. Memory dereferences are key attack vectors that allow an attacker to abuse a compartment as a confused deputy to leak or corrupt its isolated memory. Figure 2 shows the key components of our design: a taint analysis and a characterization of the tainted flows as described in section 4.

3.1 Compartment Memory Dereference Path

Concretely, we model a compartment as a set of interfaces that take inputs which could flow to critical memory operations, including the source, destination, and size parameters of the memory operation. If this happens, the attacker with control to the compartment interface could indirectly access all the privileged data within the compartment. Informally, an CMDP is a backward slice [30] from each sensitive memory operation to the calls of the compartment interfaces.

A *Program Graph* is a graph <V, E> where vertices are program instructions and edges represent either a *Data Dependency* or *Control Dependency*. A *Data Dependency* exists when the value of an instruction depends on a value produced by another: if the *operand* of the instruction is the result from another instruction. A *Control Dependency* exists between a control instruction (*jmp*, *br*, or *call*) and the target instructions of its control transfer.

A CMDP is a path on the *Program Graph* that begins at a compartment interface (*i.e.,* setting the function argument variables) and terminates at a memory dereference operation or special memory operation function. Formally, the CMDP is a backward slice S_v of one vertex $v \in V$ by dataflow equation:

$$S_v = \{v\} \cup \{v' \in V | (v', v'') \in E \text{ and } v'' \in S_v\}$$

such that v is a critical memory operation, including:

1. pointer dereference (load instruction)
2. pointer reference (store instruction)
3. calling memory operation functions (memset, memcpy, memmove, strlen, strcpy, scanf, printf, snprintf, gets.) and source, destination, size operands.

Each path is labeled *conditioned* if it goes through through at least one control-flow operation, and *{source, destination, size}* based on which memory operand was influenced.

3.2 Implementation and the Implication

We implement CMDP using a Program Dependency Graph (PDG). The PDG captures both control and data dependencies for every operation within the program, ensuring a comprehensive examination of all possible paths and providing detailed debugging information such as conditioned or unconditioned for each path. Imprecision may arise due to indirect calls but will only effect the control dependencies and not the data dependency paths. Consequently, our results may include some false-positive paths, but certain paths may be hard or nearly impossible to reach. Nonetheless, this imprecision in indirect call handling still enables us to estimate the upper bound of CIVs that an interface can potentially encounter.

Furthermore, we use a pointer analysis to identify whether the possible targets of a memory dereference point at objects allocated by code in the compartment. This allows characterization of potential bad patterns later as described in Section 4.3. We track the following kernel allocators: *kmalloc*, *kzalloc*, *krealloc*, *kmalloc_array*, *krealloc_array*, *vmalloc*, *vzalloc*, *kvmalloc_node*, *kmem_cache_create*, *kmem_cache_create_usercopy*, and *kmem_cache_alloc*.

4 Compartment Interface Weaknesses

To automatically reason about and quantify CIVs, we present a taxonomy that classifies each CMDP based on its type and severity. The taxonomy (overview in Table 1) naturally produces a method for automatically quantifying and characterizing CIVs and can be used to measure the threat to a compartment as well as indicate the complexity required for refactoring. We present two CIV patterns that lead to complete bypass that can be programmatically found.

4.1 Taxonomy

Adopting a similar model to describing classes of program weaknesses (*i.e.,* CWEs) and their exploitable instances as vulnerabilties (*i.e.,* CVEs), we introduce *Compartment Interface Weaknesses (CIWs)*. The specific class of CIW introduced are *compartment memory bypass* CIWs, where an attacker can influence the source, destination, or size of memory operations. We present the following taxonomy that maps CIWs to CIVs, which summarizes the severity of each path type. We classify each based on how many operands it can influence. Intuitively, the more operands controlled the more powerful that path is. The most powerful is control over all operands, which is effectively providing a getter/setter to any objects the compartment has privilege to modify.

Source Pointer. Control of a source pointer allows an attacker to perform the following:

- **Unauthorized Data Access:** An attacker can manipulate the source pointer to access a different memory location within the protected compartment. For example, an invalid string without '\0' in a *strlen* operation can potentially leaking sensitive information if the length is later used in a read operation.
- **Injection Attacks:** The use of eBPF interfaces in the Linux kernel can create opportunities for injecting and executing data as code [7]. By manipulating the source pointer, an attacker may exploit this capability to execute malicious code.
- **Memory Disclosure:** The source pointer can be used to read sensitive information, such as encryption keys or addresses of functions or variables, leading to memory disclosure.
- **TOCTOU:** If the source pointer has a check at a certain location and there is a data modification before the read operation, it is possible for an attacker to exploit this temporal violation. By leveraging the modified source, the attacker may be able to bypass the original check.

Destination Pointer. The destination pointer in a memory operation allows writing to memory on behalf of the protected compartment, and can be exploited to perform:

Table 1. Taxonomy for Mapping Attacker Controlled Source, Destination, and Length Capabilities to CIVs

Control	Capability	Compartment Interface Vulnerability
src	Point to anywhere src pointer with read permission	Unauthorized Data Access, Injection Attacks, Data Corruption, Denial of Service (DoS), Memory Disclosure, TOCTOU, Control Flow Hijack
dest	Point to anywhere dest pointer for write permission	Data Corruption, Buffer Overflow, Injection Attacks, Unauthorized Memory Write, DoS, TOCTOU
len	Arbitrary length for read or write	Buffer Overflow, DoS, Data Leakage, Data Manipulation, Control Flow Hijack
src+dest	Point to anywhere src and dest pointer for read and write	Memory Corruption, Arbitrary Memory Access, Data Leakage, Pointer Manipulation, Code Execution
src+len	Point to anywhere src pointer with the arbitrary length for read	Buffer Overflow, DoS, Data Leakage, Memory Corruption, Access Control Bypass, TOCTOU
dest+len	Point to anywhere dest pointer with the arbitrary length for write	Buffer Overflow, Data Corruption, DoS, Arbitrary Memory Write, Access Control Bypass, TOCTOU
src+dest+len	Point to anywhere for read and write with arbitrary length	Arbitrary Memory Access, Memory Corruption and Exploitation, Pointer Manipulation, Injection Attacks, Data Leakage and Manipulation, TOCTOU

- **Data Corruption:** Manipulating the destination pointer can lead to unintended writes to memory regions, resulting in data corruption. For example, if the attacker knows the *sk_buff* memory address and exploits a CIV in *memset* they can corrupt the *sk_buff*.
- **Buffer Overflow:** By writing to the length field of the protected compartment, an attacker can cause buffer overflow vulnerabilities.
- **Denial of Service (DoS):** Modifying the destination pointer to an invalid or inaccessible memory location can cause the system to deadlock, infinitely loop, or crash.
- **TOCTOU:** If the destination pointer is initially checked before accessing the protected compartment but later used without further validation, they can launch a TOCTOU.

Length. Manipulating the length operand can lead to the following CIVs:

- **Data Manipulation:** By truncating or extending data through the length pointer, an attacker can potentially alter the system's behavior or manipulate the data itself.
- **Control Flow Hijack:** Manipulating loop invariant or bypassing checks through the length pointer can enable an attacker to hijack the control flow of the program.

Combination of Operands The combination of these three operands can lead to even more powerful attack capabilities. If an attacker controls all three operands, they can potentially execute multiple attack gadgets and do things like privilege escalation or even more with the arbitrary read/write anywhere.

4.2 Unchecked Pattern

To identify problematic patterns that may result in different CIVs, we analyze the controlled source, destination, and length. One major issue we observe for paths flowing to the operands is the lack of sanity checks. When an interface allows an argument flow to the source, destination or size, it implies the path is guaranteed to be exploitable and can be mapped to the attack vectors in the taxonomy. Our assumption is that anything beyond the compartment boundary is susceptible to corruption, and the absence of a sanity check translates to a high likelihood of true positive CIVs. In such cases, attackers gain direct control over the source destination or size and potentially exploit vulnerabilities for unauthorized data access, injection attacks, data corruption, memory disclosures, and other security risks within these unchecked paths. The solution to secure these unchecked paths involves implementing customized access policies.

However, it is important to note that even if a path has an existing check, it does not guarantee to be safe from CIVs. The correctness of the check itself must be verified. Therefore, for paths with unverified checks, we provide an upper-bound threat analysis, detailed in section 5.1.3. This upper-bound threat analysis helps developers gain a comprehensive insight of the potential risks associated with the capabilities exposed and the mapping CIVs. It also works as a tool to evaluate the chosen compartment boundary safeness to CIVs and help determine the optimal boundary configuration that minimizes changes for compartmentalization while enhancing security against CIVs.

4.3 Getter/Setter Pattern

The getter/setter pattern provides a complete data flow paths as whom are allowed to assess the memory operations. Let's examine an example illustrated in Figure 3 to understand how CIVSCOPE can identify the CMDP and analyze the interface access pattern. In this scenario, we consider that the *trace* file is isolated through file-level compartmentalization. Since *trace_parser_get_init* has no check before the *memset*, it makes sense to implement a sanity check that enforces a customized policy to either whitelist or blacklist the malicious access if chosen as the compartment interface. However, this task is not straightforward and requires expert knowledge to

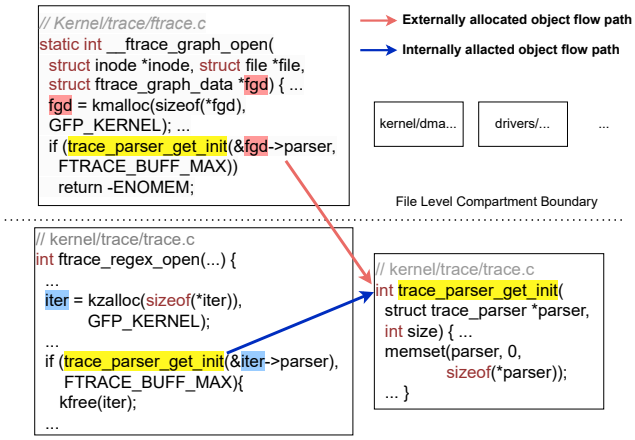

Figure 3. Call paths of different allocation sites.

comprehend how the system functions interact and derive the correct policy to safeguard the interface.

This is where CIVSCOPE can be helpful. It helps detect all the whitelisted access paths and provides developers with a comprehensive view how the interface check should be implemented. As shown in Figure 3, CIVSCOPE analyzes that there are two data paths permitted to use the *memset* function: *fgd* originates from the object allocated in the *ftrace* compartment and is defined as external to the *trace* file, while the other is allocated inside the *trace* file itself, the *iter* object. By displaying all potential internal and external object data paths permitted to use the memory operation, we obtain a complete understanding of the data flow for the memory operation access policy. This information then assists developers in constructing the appropriate policy. We refer to this as the *getter/setter pattern* which assists the analysis of data flow for memory operations within interfaces.

5 Result and Discussion

5.1 Linux Compartmentalization Assessment

To assess the analysis result of CIVSCOPE, we have employed our methodology to analyze the Linux kernel, focusing on the core kernel's exported interfaces as compartment boundaries. We start our analysis with memory operations such as *memcpy*, *memset*, and etc. because these are the most frequently exploited operations. By gaining a comprehensive understanding of the safety status of each system with memory operations, we can then incrementally incorporate all memory dereferences, including load and store instructions, into our assessment process as load and store are more difficult mapping back to the source code level. This incremental approach ensures a thorough examination of potential vulnerabilities from the most easy to understand memory operations and delves into all memory dereferences later.

5.1.1 Unchecked Pattern A key insight from Figure 5 is the distribution of checked and unchecked paths within the Linux core kernel. The y-axis represents the distinct count

```
static struct page *xdp_linearize_page(...
    struct page *p, int offset, int page_off,
    unsigned int *len) { ...
- struct page *page = alloc_page(GFP_ATOMIC);
+ int tailroom = SKB_DATA_ALIGN(
+                 sizeof(struct skb_shared_info));
+ struct page *page;
+ if (page_off + *len + tailroom > PAGE_SIZE)
+     return NULL;
+ page = alloc_page(GFP_ATOMIC);
  if (!page)
    return NULL;
  memcpy(page_address(page) + page_off,
  page_address(p) + offset, *len);
  ... }
```

Figure 4. Linux commit for the missing check before memcpy

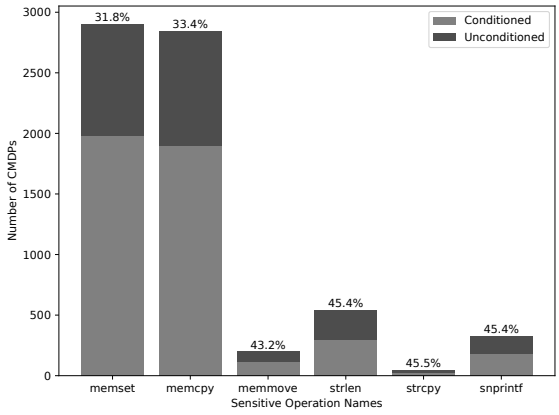

Figure 5. The number of CMDP and conditioned/unconditioned ratio for different memory operations in the core kernel.

of CMDP instances we have identified and the x-axis is the memory operations tracked. In the case of unchecked paths, this figure demonstrates how many of these paths can potentially lead to CIVs as outlined in Table 1, with approximately 40these paths lacking sanity checks. Notably, a substantial portion of these paths originates from the same interface argument to the operand. This means that the same combination of (argument, operand) can generate multiple paths due to various data flow scenarios. Our goal is to comprehensively enumerate and present all such paths because the absence of conditioning in any one of these paths makes it exploitable. Therefore, while the total count may appear alarming, many of them originate from identical (argument, operand) pairs, and exposing all of them is essential.

5.1.2 Getter and Setter Pattern We use the interface pointer's reference to private objects / (private objects + public objects) ratio to indicate the getter or setter interfaces. By analyzing accessible private objects, developers

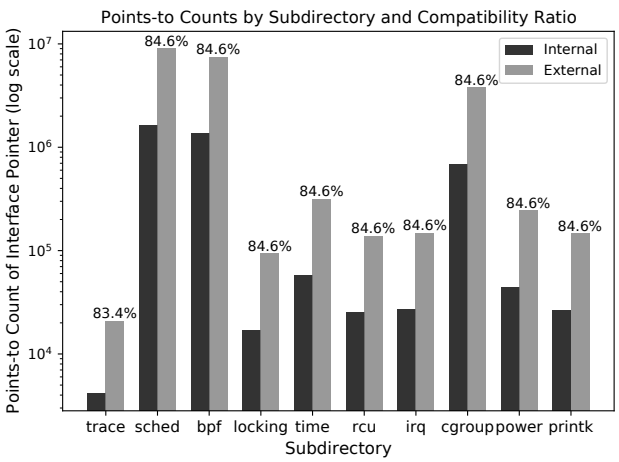

Figure 6. The compatibility ratio is defined as the (number of paths the CMDP pointer reference public objects)/(number of paths the CMDP pointer reference to any objects). Almost all subsystem interface shows the consistent ratio of 85%, indicating that the pointers have the potential to reference the same set of objects. Numbers are shown in log scale.

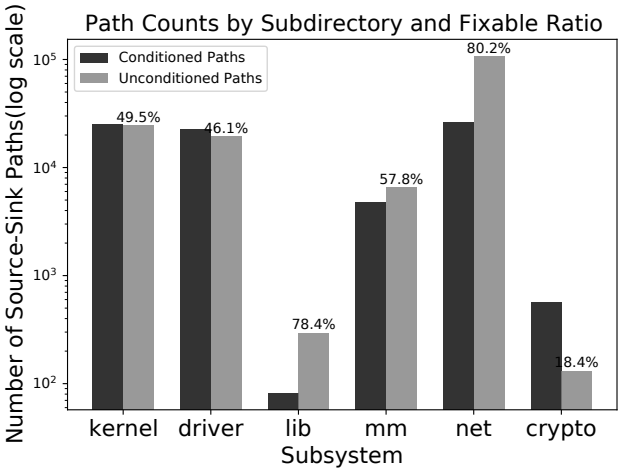

Figure 7. Conditioned/unconditioned ratio among subsystems in Linux. The unconditioned paths indicates exploitable paths. A higher ratio indicates a greater susceptibility to exploitation.

can easily identify which private objects are unintentionally exposed to external callers. Ideally, these interfaces should only expose a subset of the private objects. Our analysis show that only four interfaces in the core kernel have the ratio as 1 while the others have the ratio of (303 / (303+1858) = 0.14 in Figure 6. The four interfaces are *tracing_map_destroy*, *ftrace_create_filter_files*, *print_event_filter*, and *audit_make_reply*. In these cases, these interfaces explicitly permit external access to their private objects. If these interfaces are selected as compartment boundaries, a straightforward check policy can be established to confine memory access solely within the memory range of the respective compartment. This approach aligns with how these functions are designed, which involve the management of private objects. For the rest of the interfaces in the core kernel, more fine grained access control needs to be done to limit the interface to access only a subset of private objects.

5.1.3 Study of the Potential Risks To study the potential risks associated with the CMDP, we perform a comparative study of all subsystems in the Linux kernel and assess their vulnerability to source, destination, and length pointer corruption. The objective of this study is to demonstrate the safety of compartmentalizing different subsystems and determine the level of effort required to protect these interfaces from CIVs listed in Table 1.

First, we examine the CMDP count in Figure 7, along with the ratio of conditioned paths to unconditioned paths. From the unchecked pattern subsection, the unconditioned paths are found to be exploitable. And CIVScope assists in identifying these paths and alerts developers to address them before using them as compartment interfaces. On the other hand, conditioned paths cause a challenge in verifying whether the conditioned branches are robust checks to stop all possible CIVs. To tackle this issue, we highlighted all paths with conditioned and unconditioned checks and used Table 1 to measure the potential risk among subsystems. The network subsystem shows the highest ratio of unconditioned paths and the largest number of CMDP. This suggests that when compartmentalizing the network subsystem, operations such as memcpy of network packets or other memory operations require additional manual effort to secure the exposed interfaces.

Figure 8 provides insights into how each subsystems are exposed to CIVs by showcasing the percentage of src, dest, and len parameters. The more controlled elements, the larger capability the attacker gets. This highlights the degrees of susceptibility of each subsystem to different types of CIVs. For instance, the crypto subsystem shows the highest ratio of unchecked src+dest+len parameters, indicating that the exposed interfaces within this subsystem are particularly attractive targets for read/write anywhere gadgets.

5.2 Uncovering Existing Bugs

To assess the impact of CIVSCOPE we looked among the recently patched vulnerabilities from the Linux kernel that could be used to exploit a CIV in a theoretical compartmentalized scenario. Commit *853618d* is one such example. The code snippet in Figure 4 highlights the patch, which focuses on the *xdp_linearize_page* function. This function passes pointer type arguments to the *memcpy* function without any checks. Given *xdp_linearize_page* as an isolation boundary, CIVSCOPE identifies an CMDP equivalent to the vulnerability introduced by the missing checks. Upon applying commit *853618d* the path is no longer accessible to CIVs.

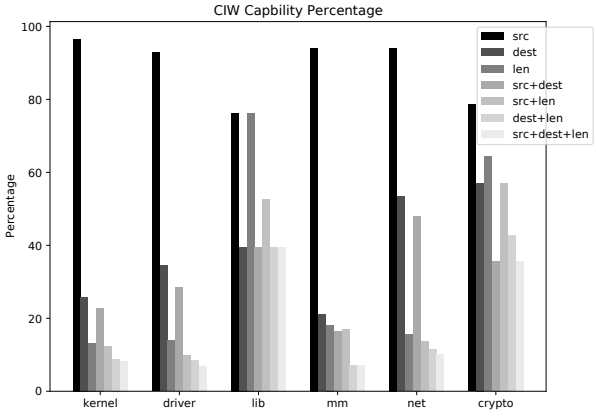

Figure 8. Degree of Attacker-Controlled Operands in Linux Subsystems: This measure quantifies the percentage of exploitable operands within each Linux subsystem. The greater the degree of control operands, the higher the capability the subsystem gets when exploited.

This is evidence that the absence of proper checks remains a significant problem in Linux. Further, when we move toward compartmentalizing Linux, the issue is much more pronounced because we need to address the confused deputy problem and all other CIVs.

6 Related Work

Compartment interface safety studies, ConfFuzz [10] is an in-depth study on how CIVs can be used to exploit compartmentalized software. The authors use an in memory fuzzer to find existing CIVs into commonly compartmentalized software such as Apache modules and libssl [25].

However, fuzzing has limitations, such as issues with code coverage, and may only uncover a relatively small portion of true positives. Other limitations include lack of integration with non-user space software such as the kernel and high manual effort from the developer to analyze and fix identified CIVs.

DUI [5] has similarities with our work, as both approaches utilize static analysis approach, DUI analyzes at the binary level and CIVSCOPE looks at LLVM IR level. DUI also leverages the symbolic execution, and dynamic taint analysis to identify pointer dereference vulnerabilities within compartment interfaces. However, our work complements DUI in several ways. Firstly, we extend the scope by examining true positive instances of CIVs in unconditioned paths. Secondly, we introduce a methodology to assess the safety of selected compartment boundaries based on the count of identified CMDP instances. Thirdly, we provide a taxonomy to map the CIVs into attack vector categories as a way to examine the severity of CMDP. Lastly, we provide a tool that refines the focus on potentially exploitable memory dereference locations and suggests additional checkpoints to incrementally secure interfaces.

7 Conclusion

In conclusion, CIVScope aims to systematically identify and address the CMDP. We introduce a tainted flow analysis that identifies all CMDP resulting from dereferencing the problematic pointers in memory operations. By applying the analysis to the Linux core kernel, we discovered that *68* out of *648* exposed interfaces have data flows bypassing the compartment boundaries. This finding highlights the impracticality of manually securing the CMDP. However, our work offers a promising outcome, a significant number of paths can be automatically identified, enabling the assessment of potential interface risks.

Acknowlegements This research was supported in part by National Science Foundation Awards #2146537 and #2008867, a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), and the EU grant agreement 758815 (CORNET).

References

[1] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (2014), pp. 90–102.

[2] Costan, V., and Devadas, S. Intel sgx explained. Cryptology ePrint Archive (2016).

[3] Cui, R., Zhao, L., and Lie, D. Emilia: Catching iago in legacy code. In Proceedings of 29th Network and Distributed System Security (NDSS) (2022), NDSS'22.

[4] Gudka, K., Watson, R. N., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Marinos, I., Neumann, P. G., and Richardson, A. Clean application compartmentalization with soaap. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (2015), pp. 1016–1031.

[5] Hu, H., Chua, Z. L., Liang, Z., and Saxena, P. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In Proceedings of the 20th European Symposium on Research in Computer Security (Cham, 2015), G. Pernul, P. Y A Ryan, and E. Weippl, Eds., ESORICS'15, Springer International Publishing, pp. 312–331.

[6] Huang, Y., Narayanan, V., Detweiler, D., Huang, K., Tan, G., Jaeger, T., and Burtsev, A. Ksplit: Automating device driver isolation. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (2022), pp. 613–631.

[7] Jia, J., Sahu, R., Oswald, A., Williams, D., Le, M. V., and Xu, T. Kernel extension verification is untenable. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems (2023), pp. 150–157.

[8] Khan, A., Xu, D., and Tian, D. J. Ec: Embedded systems compartmentalization via intra-kernel isolation. In 2023 IEEE Symposium on Security and Privacy (SP) (2023), IEEE, pp. 2990–3007.

[9] Khan, A., Xu, D., and Tian, D. J. Low-cost privilege separation with compile time compartmentalization for embedded systems. In 2023 IEEE Symposium on Security and Privacy (SP) (2023), IEEE, pp. 3008–3025.

[10] Lefeuvre, H., Bădoiu, V.-A., Chien, Y., Huici, F., Dautenhahn, N., and Olivier, P. Assessing the impact of interface vulnerabilities in compartmentalized software. In Proceedings of the 30th Annual Network and Distributed System Security Symposium (2023), NDSS'23.

[11] Lefeuvre, H., Bădoiu, V.-A., Jung, A., Teodorescu, S. L., Rauch, S., Huici, F., Raiciu, C., and Olivier, P. Flexos: Towards flexible os isolation. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2022), pp. 467–482.

[12] Lefeuvre, H., Bădoiu, V.-A., Teodorescu, Ş., Olivier, P., Mosnoi, T., Deaconescu, R., Huici, F., and Raiciu, C. Flexos: Making os isolation flexible. In Proceedings of the Workshop on Hot Topics in Operating Systems (2021), pp. 79–87.

[13] Li, J., Miller, S., Zhuo, D., Chen, A., Howell, J., and Anderson, T. An incremental path towards a safer os kernel. In Proceedings of the Workshop on Hot Topics in Operating Systems (2021), pp. 183–190.

[14] Li, L., Bhattar, A., Chang, L., Zhu, M., and Machiry, A. Checked-cbox: Type directed program partitioning with checked c for incremental spatial memory safety. arXiv preprint arXiv:2302.01811 (2023).

[15] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F. Software fault isolation with api integrity and multi-principal modules. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (2011), pp. 115–128.

[16] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., and Kaashoek, M. F. Software fault isolation with api integrity and multi-principal modules. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 115–128.

[17] McKee, D., Giannaris, Y., Perez, C. O., Shrobe, H., Payer, M., Okhravi, H., and Burow, N. Preventing kernel hacks with hakc. In Proceedings 2022 Network and Distributed System Security Symposium. NDSS (2022), vol. 22, pp. 1–17.

[18] Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., and Stefan, D. Retrofitting fine grain isolation in the firefox renderer. In Proceedings of the 29th USENIX Security Symposium (Aug. 2020), USENIX Security'20, USENIX Association, pp. 699–716.

[19] Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., and Stefan, D. Retrofitting fine grain isolation in the firefox renderer (extended version). arXiv preprint arXiv:2003.00572 (2020).

[20] Olivier, P., Barbalace, A., and Ravindran, B. The case for intra-unikernel isolation. Workshop on Systems for Post-Moore Architectures 3, 7 (2020), 8–12.

[21] Roessler, N., Atayde, L., Palmer, I., McKee, D., Pandey, J., Kemerlis, V. P., Payer, M., Bates, A., Smith, J. M., DeHon, A., et al. μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (2021), pp. 296–311.

[22] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. Proceedings of the IEEE 63, 9 (1975), 1278–1308.

[23] Sartakov, V. A., Vilanova, L., and Pietzuch, P. Cubicleos: A library os with software componentisation for practical isolation. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (2021), pp. 546–558.

[24] Sung, M., Olivier, P., Lankes, S., and Ravindran, B. Intra-unikernel isolation with intel memory protection keys. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (New York, NY, USA, 2020), VEE '20, Association for Computing Machinery, p. 143–156.

[25] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. Erim: Secure, efficient in-process isolation with protection keys mpk. In 28th USENIX Security Symposium (USENIX Security 19) (2019), pp. 1221–1238.

[26] Van Bulck, J., Oswald, D., Marin, E., Aldoseri, A., Garcia, F. D., and Piessens, F. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1741–1758.

[27] Vasilakis, N., Karel, B., Roessler, N., Dautenhahn, N., DeHon, A., and Smith, J. M. Breakapp: Automated, flexible application compartmentalization. In NDSS (2018).

[28] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In Proceedings of the fourteenth ACM symposium on Operating systems principles (1993), pp. 203–216.

[29] Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In 2015 IEEE Symposium on Security and Privacy (2015), IEEE, pp. 20–37.

[30] Weiser, M. Program slicing. IEEE Transactions on Software Engineering SE-10, 4 (July 1984), 352–357.