

Characterizing and Detecting Bugs at the Interfaces of OpenBSD Privilege-Separated Programs

Shibo Ai*
University of British Columbia
Vancouver, Canada

Shao Chen Zhang
University of British Columbia
Vancouver, Canada

Hugo Lefeuvre†
University of British Columbia
Vancouver, Canada

Margo Seltzer
University of British Columbia
Vancouver, Canada

Abstract

Compartmentalization is an approach that limits the amount of damage a bug or security breach can wreak by decomposing a program into isolated compartments that each run with the least degree of privilege necessary. When compartmentalizing a program, developers must carefully harden the communications between compartments (i.e., compartment interfaces) to prevent attacks from spreading across boundaries. Unfortunately, this hardening is error-prone and often ad-hoc, leading to Compartment Interface Vulnerabilities (CIVs), a class of bugs that can render compartmentalized programs no more secure than their monolithic counterparts. Despite the potential impact of these bugs, the community has little understanding of how effective today's interface defenses are.

This work systematically evaluates the strengths and limitations of CIV defenses deployed in production-grade compartmentalized programs. We audit a representative corpus of 25 popular compartmentalized programs in OpenBSD using a combination of manual analysis, fuzzing, and LLM agents. Our study identifies 81 CIVs, some over a decade old, with impacts as varied as memory corruption, information leakage, and logic bugs, and as severe as arbitrary code execution. We systematize our findings to extract high-level patterns that characterize these issues and derive more effective approaches to address them. Our results expose the strengths of OpenBSD's approach, such as how its avoidance of shared memory eliminates entire classes of issues. However, we also show that even programs compartmentalized by security experts remain vulnerable to recurring, high-impact CIV patterns. At the core, these findings highlight the need for clear (and ideally formal) specifications of compartment interfaces to achieve systematic counter-measures. This paper demonstrates flaws in existing compartmentalization schemes and provides a foundation for future work on the discovery and mitigation of CIVs.

CCS Concepts

• **Security and privacy → Systems security**.

*Also with EPFL.
†Also with Max Planck Institute for Security and Privacy.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846768

Keywords

Isolation, Compartmentalization, Privilege Separation, CIV

ACM Reference Format:
Shibo Ai, Shao Chen Zhang, Hugo Lefeuvre, and Margo Seltzer. 2026. Characterizing and Detecting Bugs at the Interfaces of OpenBSD Privilege-Separated Programs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846768

1 Introduction

Modern computer systems face constant attacks, and absolute security is unrealistic. This inevitability of a successful attack means that it is critical to design software in a way that minimizes damage in the presence of a security breach. One such approach is compartmentalization [46] (also known as *privilege separation* [59]), which splits a program into isolated compartments and grants each compartment only the minimum privileges needed to perform its tasks. This confines an attacker's capabilities to the compartment they compromise, preserving the integrity of the rest of the program.

Compartmentalizing an application creates new internal interfaces, called *compartment interfaces*, that mediate interactions between compartments. These interfaces define isolation boundaries while still allowing legitimate interactions compartments need to collaborate and function as a consistent program. For example, in a compartmentalized web browser, a higher-privileged renderer compartment might need to interact with a less-privileged JPEG parser to process a webpage. Unfortunately, these interactions inevitably lead to complex inter-compartment data and control flows. This results in attack surfaces that attackers can exploit from a compartment they compromise. Previously characterized as *Compartment Interface Vulnerabilities (CIVs)* [44], these attack surfaces can partially or fully undermine the effectiveness of compartmentalization.

As compartmentalization is becoming increasingly popular in both research and industry [46], CIVs become a growing problem [36, 44, 46]. Prior work investigated the prevalence of CIVs in unmodified code-bases [34, 36, 44], showing that correctly hardening compartment interfaces is both an *essential* and *particularly challenging* part of compartmentalizing programs. However, we still have little understanding of how effective our defenses are at addressing this issue. To date, no study has examined how real-world compartmentalized programs mitigate CIVs, or how effective these mitigations are. This gap leaves developers without clear guidance, encourages designs that appear safe but are fundamentally

fragile, and ultimately hinders the efficacy of compartmentalization techniques. Therefore, we ask the following questions: **How do existing mainstream compartmentalized programs protect against CIVs? What are the limitations of these techniques, and as a result, what CIVs can arise? How can we systematically address them?** As industry actors continue to adopt compartmentalization, answering these questions is urgent for guiding future compartmentalized designs.

We conduct a large-scale study of mainstream compartmentalized programs to answer these questions. Our goal is to better understand common interface hardening practices, identify real-world CIVs that remain in such designs, and build a taxonomy to extract recurring high-level patterns and insights about these flaws and their solutions. Specifically, we study OpenBSD [2], a security-focused operating system (OS) whose userland contains, to the best of our knowledge, the largest repository of open source compartmentalized programs used in production. The OpenBSD userland is well suited for this study, because it is (1) large-scale, widely used, and compartmentalized using consistent methodologies, (2) developed by experts who are aware of CIVs, and (3) representative of high-quality compartmentalized programs beyond the OpenBSD community. We adopt a hybrid methodology that combines the strengths of manual auditing, fuzzing, and Large Language Model (LLM) agents, to reveal a wide range of CIVs and insights. While fuzzing and LLM agents are not the core contributions of this paper, we contribute the design of a fuzzer based on off-the-shelf components that can effectively find CIVs, and show how widely-available LLMs can also be used to find these issues. Applying this methodology to 25 OpenBSD userspace programs, we identified 62 CIVs. We reported all findings to the OpenBSD security team, leading to fixes and to an independent audit that uncovered 19 more CIVs. Many programs in our corpus are widely deployed outside OpenBSD, and OpenBSD's privilege-separation approach established foundational design patterns adopted by other open-source programs, supporting the relevance of our findings beyond OpenBSD.

Overall, we show that even programs compartmentalized by experts remain prone to recurring CIVs, some as severe as sandbox escape followed by arbitrary code execution. These flaws undermine the effectiveness of compartmentalization, and many have persisted for over a decade. Specifically, we draw the following high-level insights: (1) CIVs are prevalent across OpenBSD programs and often follow generalizable patterns, though some programs are entirely CIV-free; (2) certain OpenBSD design choices eliminate some classes of CIVs while simultaneously enabling others; and (3) ideal mitigations should be more systematic and build on more formal specifications. Overall, we make the following contributions:

- A systematic review on program compartmentalization in the mainstream and how it approaches CIVs (§3);
- The design and implementation of a fuzzer based on off-the-shelf components to automatically find CIVs in mainstream compartmentalized programs (§5);
- The systematization of 81 new CIVs in popular compartmentalized programs (§6), along with avenues for mitigation (§7).

We open-sourced our artifacts (Appendix A).

2 Background

2.1 Program Compartmentalization

Compartmentalization takes many forms [46]. The historical and most popular approach to compartmentalization is called *privilege separation* [40, 59]. It decomposes applications into multiple processes, each running with minimal required privileges, that communicate through inter-process communication (IPC) mechanisms.

Consider OpenSSH, one of the first compartmentalized programs [59]. It consists of two sets of processes (compartments). Privileged compartments run as root, cannot do network operations, and perform security-sensitive tasks, e.g., managing private keys. Unprivileged compartments exchange network messages, potentially with attackers. Successful exploits from the network remain confined to unprivileged processes and require additional privilege-escalation primitives to escape. This proved effective, mitigating bugs that would otherwise have had significant impact [10–12, 14].

Though manual process-based isolation remains the most widely adopted compartmentalization method in production software [46], many subsequent works explored ways to compartmentalize at a finer grain, e.g., within a process [27, 33, 45, 57, 58, 64, 71, 75], and to (partially) automate compartmentalization [19–21, 30, 35, 37, 42, 47, 48, 62, 68, 73]. Compartmentalization also showed potential to strengthen memory-safe languages, e.g., by isolating unsafe parts [17, 41, 53, 61, 65, 70] or to thwart supply-chain attacks in memory-safe software [15]. These works increase the appeal for compartmentalization but leave the crux of the problem unchanged: creating and protecting new security interfaces in programs is hard.

2.2 Compartment-Interface Vulnerabilities

Most compartments cannot be fully confined and need to communicate to accomplish the task they are designed to do. In OpenSSH for example, the network compartment has to transmit credentials to the privileged compartment. This results in many new internal boundaries that enable legitimate interactions between compartments, while still enforcing isolation. These interfaces must be carefully protected as they are the internal attack surface of compartmentalized programs. *Compartment-Interface Vulnerabilities* [44] (CIVs) are bugs caused by insufficient validation of incoming or outgoing control- and data-flow. Attackers can exploit CIVs in a compromised compartment to propagate to adjacent compartments.

Consider one of the CIVs we found in a framework shared between five OpenBSD programs (`relayd`, `httpd`, OpenIKED, `snmpd`, `vmd`, see §6). This CIV affects one of the endpoints exposed by the privileged compartment to unprivileged compartments. Listing 1 shows the vulnerable part of the API: it uses two indices, `dst` and `n`, passed by unprivileged callers, to access an array and read the content into `iev` (①). However, the privileged code does not check that indices are within bounds, causing an array over-read and the subsequent storage of over-read content into `iev`. Worse, the code then interprets the over-read data as a function pointer and registers it as an event handler (② and ③), potentially giving attackers the ability to execute arbitrary code in the privileged context.

CIVs can cause three categories of impact for a compartment:[1] (1) **Pure Memory Safety Violations (MemSaf)**, where memory

[1]There are multiple ways to categorize security impacts. We use this categorization because (1) the categories correspond to different bug-finding techniques (§4.2); (2)

```
// Unprivileged compartments control dst and n.
void proc_accept(struct privsep *ps, ..., enum procid dst, uint n) {
  struct imsgev *iev = &ps->ps_ievs[dst][n]; ① // Simplified.
  event_set(&iev->ev, ..., iev->handler, ...); ②
  event_add(&iev->ev, NULL); ③
}
```

Listing 1: Simplified example of a CIV in the `proc` framework.

corruption (e.g., the buffer overflow shown above) or a memory-safety-related information leak (e.g., a leak of uninitialized memory) can be triggered across compartments; (2) **Non-Memory-Safety Information Leakage (InfoLeak)**, where legitimate, memory-safe cross-compartment interactions can be abused to cause the leakage of compartment-confidential data; and (3) **High-Level Logic Violations (Logic)**, where legitimate, memory-safe compartment interactions can be leveraged to trigger high-level logic bugs that violate the security objectives of compartmentalization (e.g., oracle attacks [51] in OpenSSH, where logic-level bugs give a malicious compartment access to an arbitrary signing/decryption oracle despite not having access to the keys). Prior work showed that CIVs can fully undermine the value of compartmentalization [44].

Prior works describe CIVs in compartmentalized applications [34, 44] and OS kernels [24, 36], some focusing on the MemSaf [24, 34] or Logic [51] impact classes. However, there is a clear *lack of work in contributing and evaluating defenses against CIVs*. In particular, no work has evaluated the effectiveness of existing manual defenses against CIVs and the kind of issues that remain. This ultimately hinders the reach of compartmentalization as a mainstream technique [46]. This work is the first study to systematically evaluate the presence and impact of CIVs in real-world compartmentalized programs *whose interfaces were meant to mitigate CIVs*.

3 Compartmentalization in OpenBSD

The OpenBSD userland includes approximately 45 compartmentalized programs, such as OpenSSH, OpenBGPD, and tmux, compartmentalized with consistent methodologies [46]. OpenBSD was a pioneer in compartmentalization efforts [59], and developed rigorous compartmentalization practices, including hardening interfaces to prevent CIVs. These OpenBSD compartmentalized utilities have greatly influenced subsequent compartmentalization efforts and have been ported to other operating systems. This makes OpenBSD a compelling, yet untapped, corpus for deriving generalizable insights into CIVs and their mitigations in real-world programs.

We now answer our first research question (*How do existing mainstream compartmentalized programs protect against CIVs?*) with a systematic review of compartmentalization in OpenBSD and of the mechanisms it implements to protect compartment interfaces.

3.1 Partitioning Strategy

OpenBSD implements classical privilege separation (§2.1) by splitting applications into *privileged* and *unprivileged* compartments. OpenBSD developers use two main strategies to define compartment boundaries: 1) *Privileged Proxy*-based compartmentalization and 2) *Subsystem*-based compartmentalization. These strategies lead to different compartment interfaces and thus influence which classes of CIVs arise and the validation used to mitigate them.

The *Privileged Proxy* approach splits programs into two compartments. Most of the program resides in a large unprivileged compartment. Code requiring system privileges is encapsulated in a second, smaller privileged *proxy* compartment. The unprivileged compartment communicates with the proxy to request privileged operations. This strategy is common in programs where compartmentalization was retrofitted into existing codebases, e.g., `tcpdump` [6], as it does not require invasive changes.

The *Subsystem* strategy decomposes a program into multiple compartments that map to functionality boundaries. Each compartment manages a distinct subsystem (e.g., DNS queries) and holds the privileges necessary for that purpose. Like the privileged proxy strategy, one privileged compartment is often dedicated to kernel interactions. For example, the BGP daemon OpenBGPD uses four compartments: a privileged compartment for kernel interaction, and three unprivileged compartments for network traffic handling, BGP route decision, and route validation. This strategy is more common in programs that are designed and compartmentalized from the ground up, as it requires up-front planning.

3.2 Abstractions and Mechanisms

OpenBSD implements compartments as processes using a set of new APIs designed to enable strong compartmentalization.

Process Isolation. Compartments run in separate processes, providing the following isolation properties. *(1) Memory isolation*: compartments have their own address space. OpenBSD has a strict approach to memory isolation, eliminating shared memory [9] to mitigate classes of CIVs (§6.5.1). This contrasts with modern compartmentalization schemes that heavily rely on shared memory to enable fast inter-compartment communication [46]. *(2) Access control*: compartments run under different users and groups to restrict rights on the file-system, system resources, and system calls. OpenBSD vertically integrates compartmentalization to unify access control across the kernel, standard library, and applications [26]. This allows OpenBSD to minimize the risk of OS-level confused deputy attacks that affect modern compartmentalization schemes [25, 63], as these often occur due to insufficient integration of protection mechanisms across the computing stack.

Compartment Address-Space Randomization. Address-Space Layout Randomization (ASLR) is a widely-deployed technique that randomizes the memory layout of processes to make specific code or data harder to locate when mounting an attack. ASLR also makes it more difficult to exploit CIVs in a compartmentalized program. However, compartmentalization schemes deployed in practice spawn compartments with the `fork()` system call, which creates processes with identical memory layouts, i.e., there is no ASLR across compartments. OpenBSD addresses this by spawning compartments with `exec()` after a `fork()`, which it enables by building compartments as separate binaries or as separate entry points in one binary. The `exec()` system call re-randomizes the address space to give each compartment a different layout.

System-Call Filtering. The system call API exposes a large privileged attack surface to compartments. OpenBSD minimizes it by

they provide a useful basis for comparing prior CIV work; and (3) they intuitively correspond to different compartment-interface checks (§3.3.3).

enabling developers to restrict the system calls available to each compartment using `pledge` [3] – an OpenBSD-specific system call that is similar to `seccomp` [5] in Linux. `pledge` restricts which fine-grained categories of system calls a process may use for the rest of its runtime. For example, the `rpath` category allows path traversal, retrieving file metadata, and opening files for reading, but not writing. Compartments can require different privileges at different execution stages; for example an initialization phase might require system calls that are not needed later. Compartments call `pledge` early and often to minimize the duration in which a compartment runs with high-privilege: as they finish tasks that require a specific system call that is not needed later, they relinquish privilege. Compartments can also choose to create resources that require high privilege (e.g., open file descriptors) ahead of time and relinquish that privilege early rather than retaining it for the entire execution.

File-System Access Control. The file-system represents another significant attack surface exposed to potentially malicious compartments. While user access-control can enforce coarse file-system restrictions, it cannot hide many public system files or constrain compartments that run as root. OpenBSD exposes `chroot` and `unveil`, which further restrict file-system access beyond user permissions. `chroot` changes the calling process's view of the file system by moving its root directory to a specified path, restricting the visible file-system tree to that directory and its descendants. `unveil` is an OpenBSD-specific system call intended for compartmentalization that enforces finer-grained allowlist policies: it exposes only a specified set of paths to the calling process and, unlike `chroot`, is not limited to files under a common parent directory.

3.3 Hardening Compartment Interfaces

Compartments need to interact with one another to function as a logically-consistent program. This results in many interfaces that must be carefully hardened. We now discuss how these interfaces are implemented in OpenBSD, and as a result of such design choices, what developers need to do to harden them.

3.3.1 Compartment Communication. OpenBSD compartments communicate via message passing. They invoke functionalities in other compartments by exchanging IPC messages, which invoke the corresponding handlers in the receiving compartments. This message passing is typically implemented on top of `pipes` or `sockets`. `sockets` also transmit file descriptors between compartments, making it easy to offload privileged operations that produce them into a privileged proxy compartment (§3.1).

Most programs use Remote Procedure Call (RPC) libraries instead of raw IPC APIs to avoid duplicated code and allow developers to focus on communication protocols. The `imsg` RPC library is the most common framework, used in 28 OpenBSD compartmentalized programs. It abstracts the details of low-level IPC, exposes a reliable channel that eliminates the possibility of partial message reception, and eases common operations such as getting the size of a payload or authenticating the sender of a message. Messages consist of 1) a fixed-length `imsg` header that ships metadata including a message type field used to build RPC protocols, followed by 2) a variable-length payload. The structure of the pre-defined, fixed-size message header provides a common format for all messages.

Table 1: Compartment-interface validations in OpenBSD.

Validation Class	Information Needed to Check	CIV Impacts Mitigated		
		MemSaf	InfoLeak	Logic
V1: *Length checks*	Syntax + Semantics	●	○	○
V2: *Indexing checks*	Syntax + Semantics	●	○	○
V3: *Type checks*	Syntax + Semantics	●	○	○
V4: *Authentication*	Semantics	○	●	●
V5: *Sequence checks*	Semantics	●	●	●
V6: *Policy checks*	Policy	●	●	●
V7: *Confused deputy checks*	Semantics	○	●	●
V8: *Sanitization*	Syntax + Semantics	●	●	○

3.3.2 Number of Message Handlers. While privileged compartments in the 25 programs we study later (§4.2) expose, on average, 10 handlers, the number of handlers varies significantly across applications and compartments. For example, OpenSSH exposes the most (28 privileged handlers), whereas `script` exposes none (communication is unidirectional: the privileged compartment takes no messages from other compartments). The distribution is also skewed: only 9/25 programs (36%) expose more than 10 handlers.

3.3.3 Validating and Sanitizing Messages. Frameworks such as `imsg` handle inter-compartment message delivery, but they do not know *how* compartments use these messages to communicate. It is thus developers' responsibility to ensure that compartments enforce the syntax, semantics, and policy of the RPC protocol to prevent CIVs. We systematize eight classes of compartment-interface validations in OpenBSD programs (V1-V8, Table 1).

V1: Length Checks. Upon receiving a message, compartments check that the number of bytes received matches the fixed or variable size of the message as specified by the RPC protocol. For example, the size of `sockaddr`-style structs, exchanged across compartments in OpenIKED, is specified in a member field. `imsg` facilitates these checks with APIs to extract the full length of a message. Improper length validation can allow a sender to trigger MemSaf CIVs. Writing such checks requires understanding the syntax and the static semantics of messages.

V2: Indexing Information Checks. Messages can transmit memory offsets or array indices, which must be checked to be within a valid range. These typically refer to entries in data structures within the memory of the receiver compartment, so the valid range for these entries is known to the receiver. Listing 1 shows a case of missing such a check: the message ships indices `dst` and `n` referring to an array that holds per-compartment data in the privileged compartment. These indices correspond to compartment type and instance numbers, so these should have been checked against the total number of compartment types and instances. Like V1, improper validation can lead to MemSaf CIVs. Correctly implementing V2 checks requires understanding the syntax and the dynamic semantics of messages.

V3: Type Checks. Messages commonly include objects that are validated with respect to their expected type, as defined by the RPC protocol. Strings and file paths are common, but more complex objects such as `sockaddr`-style structs for OpenIKED or `httpd`'s large `struct server_config` also appear. Developers verify that objects comply with both language-defined properties (e.g., string

termination, numeric representation) and developer-defined properties (e.g., a message type field must be between 0 and the total number of message types). Improper type validation can also result in MemSaf CIVs. This check requires understanding both syntax and static and dynamic semantics of messages.

V4: Sender Authentication. Compartments also verify that message senders are allowed to invoke the operations they request. Improper checks could allow a sender to impersonate another compartment and trigger an RPC for which they lack privileges, causing CIVs in the InfoLeak or Logic classes. For example, a lack of authentication in OpenSMTPD's queue compartment (§6.3.1) allows the `lka` compartment to delete e-mails from the queue, which it should not be able to do. Receivers authenticate senders with a custom API (§3.3.1) that lets them check the IPC endpoint from which a message was sent. Writing such checks requires understanding the high-level semantics of the compartmentalized program to assess which compartments should be allowed to invoke which RPCs.

V5: Message Sequence Checks. Certain messages have ordering requirements that the receiver compartment enforces. For example, the pre-authentication procedure in OpenSSH requires the unprivileged compartment to first transmit the user name via a call to `MONITOR_REQ_PWNAM`, followed by `MONITOR_REQ_AUTHPASSWORD` for the password. The developer must implement a state machine to check that the sequencing is correct, which requires understanding the full dynamic semantics of the RPC. Improper message sequence checks may result in CIVs from any of the three impact classes.

V6: Configuration and Policy Checks. Some RPCs may be disabled by configuration or policy. For example, disabling GSS-API authentication in OpenSSH removes several RPCs responsible for that feature. Writing such checks requires understanding the high-level semantics of the compartmentalized program and its configuration space. Similarly to V5, improper checks can produce various errors.

V7: Confused Deputy Checks. Some RPCs expose restricted aspects of a privileged operation to unprivileged compartments – a pattern that commonly arises when sender-controlled fields flow into arguments to privileged system or library calls. Receivers validate inputs from untrusted callers to ensure that the resulting operation remains within the sender's intended authority. In `httpd` for example, the privileged compartment receives log file paths from unprivileged compartments and returns opened file descriptors. To avoid being abused as an arbitrary file opener, the privileged compartment accepts filenames relative to the log directory and sanitizes them against path traversal. Although this situation resembles Hardy's confused deputy [32], OpenBSD compartments do not use capabilities. Correctly implementing such checks requires understanding the syntax and full semantics of the message.

V8: Output Sanitization. Sending uninitialized data (or uninitialized object fields) to another compartment can leak information. Developers zero-out allocations sent to other compartments to minimize that risk. Some programs such as OpenSSH and rpki-client also rely on buffer-packing helpers that facilitate packing variable-length data into RPC packets to prevent leakage due to under-filled buffers and compiler-added paddings. These helpers also encourage developers to send only needed object fields as opposed to entire internal objects that can leak private data. Correctly sanitizing outputs requires RPC protocol syntax and static semantics.

3.4 Learning from OpenBSD's Experience

The techniques described in the previous section are the result of many refinements by OpenBSD developers over the past 25 years. We observe three drivers behind these changes.

(a) Some changes were done in response to reports of CIVs. In 2015 for example, an external audit of OpenSMTPD revealed several CIVs [60] (which it called "*Inter-Process Vulnerabilities*").

(b) Other changes were made to proactively reduce the attack surface. For example, in 2024, OpenSSH changed from re-executing the same binary for different compartments to shipping multiple executables [13, 54]. This reduced the shared code between privileged and unprivileged compartments, minimizing the attack surface for the network-facing listener compartment.

(c) Yet other changes were made to satisfy evolving functional requirements. In 2004, OpenBGPD changed the `imsg` framework to use sockets instead of pipes to support passing file descriptors between compartments [38]. In 2021, OpenBGPD added support for the RTR protocol, which it implemented in a new compartment following the subsystem partitioning strategy (§3.1) [39].

This approach, where experts carefully craft and refine compartment interfaces on a program-by-program and bug-by-bug basis, is arguably the best one can do to obtain strong security properties with compartmentalization. Prior works have demonstrated the importance and difficulty of hardening interfaces [34, 36, 44, 46], and to the best of our knowledge, all compartmentalization schemes rely on the assumption that developers can design interfaces and write checks that are correct. Even so, we still have little insight into *how effective developers truly are at designing strong compartment boundaries*. This makes OpenBSD programs a valuable source from which to understand the strengths and limitations of the expert-based approach to hardening compartment interfaces. This leads to our two next research questions: *What are the limitations of existing approaches to design compartment interfaces, and as a result, what CIVs can arise? How can we systematically address them?*

4 Study Overview

We perform a large-scale study of compartment interfaces in the OpenBSD userland to understand which CIVs persist when developers carefully design compartment boundaries, and what we can learn from these bugs to design stronger counter-measures. As motivated in §3, OpenBSD's compartmentalized programs are uniquely suited for such a study, because they are (1) numerous, widely used, and developed with consistent techniques and frameworks, (2) developed by domain experts with the intent of eliminating CIVs, and more generally (3) representative of high-quality compartmentalized programs beyond the OpenBSD community.

4.1 Threat Model

We assume that an attacker has compromised one or more compartments with (a) code execution and wants to (b) escalate privileges by (c) propagating to adjacent compartments. Assuming arbitrary code execution *(a)* in compromised compartments is the common

Table 2: Programs considered in this study.

Phase	Programs Considered	∑
Manual	OpenBGPD, OpenIKED, OpenNTPD, OpenSMTPD, OpenSSH, acme-client, dhcpd, dhcpleased, dvmrpd, eigrpd, file, httpd, isakmpd, ldapd, mountd, npppd, ospfd, radiusd, relayd, rpki-client, script, snmpd, syslogd, tcpdump, tmux	25
Fuzzing	*All from Manual except* script	24
LLM	*All from Manual*	25

threat model in prior works designing or testing compartmentalized programs [18, 21, 44, 46, 59]. With *escalating privileges (b)*, we mean compromising other compartments with different privileges, for example that run under a different user (`root` or not), with different `pledge` restrictions (enabling access to different system calls), different `chroot` or `unveil` restrictions (to access different parts of the file system), or different access control permissions. Causing other compartments to abnormally terminate is considered escalatory in OpenBSD, as compartments are generally availability boundaries. We are interested only in propagation *(c)* caused by bugs at compartment interfaces (CIVs). We define a CIV as a bug that can be triggered through inter-compartment communications with the above threat model, resulting in one of the impacts from §2.2 (MemSaf, InfoLeak, or Logic). Side channels are out of scope.

4.2 Methodology

We cover a broad range of CIVs across many programs and all three impact classes from §2.2, using a combination of techniques ranging from manual to automated.

Phase 1: Manual. We first perform a manual audit of 25 compartmentalized programs in OpenBSD (Table 2). We select a representative set of popular daemons, client programs, and system utilities. We start top-down with a study of the compartmentalization frameworks used for these programs (presented in §3) and extract the semantics of programs and of their compartments by reviewing publicly-available presentations, documentation, and code. Then, we proceed bottom-up by systematizing the checks done at these interfaces (detailed in §3.3.3), comparing them with the semantics gathered earlier, and investigating discrepancies as possible bugs. This results in a first set of 12 issues across all three impact types.

Phase 2: Mechanized. We then automate the audit to cover programs more comprehensively and find more complex bugs. We (1) design a fuzzer to mechanize the search for CIVs in the MemSaf and (part of the) InfoLeak classes. We describe our fuzzing approach in §5.1. We apply it to 24 programs (Table 2), i.e., all programs from the manual audit except `script`. `script` cannot be fuzzed, because its privileged compartment exposes no RPC handlers; we still analyze it manually and with LLM agents (see below). Fuzzing reveals 37 issues covering both targeted impact types. We (2) apply LLM agents (§5.2) to mechanize the search for CIVs in the InfoLeak and Logic impact classes, as well as for memory-safety related information leaks (part of MemSaf) to cover bugs that may not have been covered by our detectors.[2] We run the LLM search on the same 25 programs as in the manual audit (Table 2). This reveals 13 more issues across all targeted impact categories.

Triaging and reporting. We manually confirm and de-duplicate each CIV by analyzing its underlying root cause. For example, if several different RPC endpoints flow into a single shared vulnerable function, we count a single CIV. However, in the case of duplicated or similar vulnerable code, we count one CIV per copy of the vulnerable pattern. Finally, we construct PoCs in OpenBSD and, when possible, patch each bug before reporting it to the OpenBSD security team. We also track CIVs that OpenBSD developers fix beyond the ones we report and add them to our taxonomy. This leads us to list CIVs in a program (e.g., `vmd`) not mentioned in Table 2. Note that we do not request CVE identifiers as OpenBSD's policy is to only do so for end-to-end exploits, not sandbox escapes.

Systematization. We taxonomize bugs (§6) according to 1) their impact class, and 2) their root cause pattern. Our data set highlights eight different patterns across all three impact classes. We then derive insights (§6.5) and directions for more systematic fixes (§7).

5 Mechanized Auditing for CIVs

We begin with the design and implementation of our fuzzer and then proceed to explain how we leverage LLM agents to find CIVs.

5.1 Fuzzing for CIVs

Though fuzzing process-based compartments as found in OpenBSD shares similarities with prior works in fuzzing compartments [18, 44], system-level RPC interfaces [50], and network protocols [66], it also presents specific design and implementation challenges.

C1: Complex compartment interactions. Programs in our corpus feature many inter-connected compartments with complex trust relationships. For example, OpenSMTPD and `acme-client` have 7 and 8 compartments respectively. When fuzzing the interface between a malicious compartment C_{mal} and a privileged compartment C_{priv}, the (concurrent) interactions between C_{priv} and other compartments must also be considered. These lateral interactions make it challenging to build up the state necessary to fuzz C_{priv} effectively. Compartment interactions are also more ad-hoc and less specified than typical network or system-level RPC protocols, making it even harder to reach deep into compartment code.

C2: No standard compartmentalization framework. Although the compartmentalization techniques used in OpenBSD are consistent (§3), their use varies across programs. For example, while 28 programs use `imsg`, 9 of the programs in Table 2 (including popular applications such as OpenSSH and tcpdump) use a custom RPC framework. This differs from prior works in fuzzing compartments and RPC frameworks. For example, ConfFuzz [44] assumes compartment interfaces are functions with a C calling convention, and NASS [50] targets standard Android RPC frameworks. The lack of a standard compartmentalization framework makes it more difficult to interface the fuzzer with compartments.

C3: No suitable detectors. Instrumentation such as ASan [67] can detect MemSaf CIVs, but it is not available in OpenBSD, and native instrumentation (e.g., hardened memory allocator) is more limited. Further, there is no detector suitable for InfoLeak CIVs.

[2] While one could run the LLM search on all three complete impact classes, we decide to focus on areas that are most complementary to fuzzing.

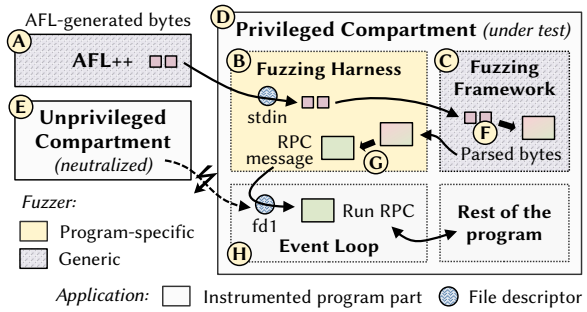


Figure 1: Overview of our fuzzing setup, applied to a program with two compartments (privileged/unprivileged).

Overview. These challenges motivate us to design and implement a fuzzer specialized for the interfaces of process-based compartments. We build on three components, shown in Figure 1. First, an off-the-shelf fuzzing engine (AFL++ [29], Ⓐ) allows us to reuse existing fuzzer parts that do not need to be specialized for compartments (e.g., the mutation engine). Second, a program-specific fuzzing harness (Ⓑ) interfaces the engine with program logic. Third, a generic fuzzing framework (Ⓒ) factors generic aspects out of the harness (e.g., most of the generation of RPC messages).

Supporting Complex Compartment Interactions. We begin to address C1 by *fuzzing for a slightly stronger threat model*: we consider all compartments but the compartment under test (Ⓓ, typically the privileged compartment) as malicious. This allows us to run only the compartment under test, replacing all other compartments (Ⓔ) with the fuzzer. Fuzzing the entire interface of the compartment under test helps us build the state necessary to cover complex paths. This also allows us to find CIVs that arise when *several malicious compartments collude* to compromise the compartment under test, a more complex class of bugs that is often difficult to find manually.

We further address C1 by *making the fuzzer protocol-aware*. The fuzzing harness (Ⓑ) and framework (Ⓒ) help the fuzzer quickly build up state by re-arranging AFL's semi-random inputs into inputs that (partially) respect the RPC protocol's syntax. For example, RPCs typically encode message types as 32-bit integers (including `imsg`), but programs usually use only a small subset of these values and reject the rest. For example, OpenSMTPD has only 149 message types, so the probability of a fuzzer randomly generating a valid type is 1 in 30 million. We generate messages with valid types in most cases to quickly reach interesting code paths, while still testing messages with invalid types.

Finally, we *seed the fuzzing engine with RPC messages captured from real-world runs* similarly to Nyx-Net [66]. This is necessary to generate messages containing data that the engine cannot generate on its own. For example, OpenSSH's RPCs only make progress if the unprivileged compartment provides valid credentials such as cryptographic keys, usernames or passwords.

Fuzzing Heterogeneous Compartments. We address C2 by *splitting the fuzzer into a generic part and a program-specific part.* RPC protocols used in OpenBSD share a common set of fields with similar syntax and semantics (e.g., message type, compartment id, file descriptor). The generic fuzzing framework (Ⓒ) takes advantage of this to re-arrange part of the fuzzer engine's (Ⓐ) semi-random inputs into a collection of well-formatted, commonly-used fields (Ⓕ). For example, the generic fuzzing framework (Ⓒ) takes a parameter that directs it to generate valid RPC types only. The program-specific harness (Ⓑ) then retrieves the generic message from the fuzzing framework (Ⓕ), performs additional re-arranging for non-generic fields (Ⓖ) (e.g., message ID), and sends it using program-specific IPC mechanisms (e.g., `imsg`). Figure 1 shows a program with an event loop (Ⓗ) that polls sockets that map to each compartment (a single compartment shown for space reasons), so the harness sends the RPC message to the socket that corresponds to the unprivileged compartment. Other programs can do this differently. This modular approach allows us to engineer a harness in an average of three hours, which is not a bottleneck in the audit.

Detecting CIVs. We address C3 by *fuzzing programs in FreeBSD* instead of natively in OpenBSD. This allows reusing existing detectors for MemSan CIVs, specifically ASan, MSan, and UBSan, all unavailable in OpenBSD. Eleven of the programs are portable and thus already run on FreeBSD. We port 13 others, which entailed patching the toolchain to build and link in FreeBSD, editing headers to point to correct library locations, and porting required library dependencies. LLM agents were used to facilitate the engineering. This allowed us to scale fuzzing to a total of 24 programs (Table 2).

We complement existing detectors with *custom instrumentation to detect leakage of pointer* values across compartment boundaries. Leaked pointers can be leveraged to break cross-compartment ASLR (§3.2), e.g., towards mounting a full exploit of MemSan CIVs. Though not directly applicable to OpenBSD, leaked pointers are also a key concern when using hardware capabilities such as CHERI [15, 75, 77] to compartmentalize programs, as leaked pointers grant privileges. We detect such leaks by scanning messages for (arbitrarily-aligned) 8-byte sequences that match known process mappings. We observe only 10 false positives throughout the fuzzing campaign.

Coverage. Since the fuzzer is not the core contribution of this paper, we omit a complete evaluation for space reasons. In general, we find line of code coverage to be a difficult metric to assess the fuzzer's effectiveness as it measures coverage across the entire program, whereas our fuzzer targets only RPC-related code in privileged compartments. We instead report the number of privileged handlers covered by the fuzzer, demonstrating the ability to discover interfaces and reach past RPC framework checks. Across the 247 privileged handlers of the 24 targeted programs, our fuzzer covers 240 (97%), missing three in `tcpdump` and four in OpenSSH. These occur either because handlers are disabled by configuration or the fuzzer did not generate valid sequences required by V5 checks.

5.2 Auditing with Large Language Model Agents

The lack of a specification for compartment and interface semantics makes it hard to use fuzzing to find bugs that do not manifest with memory-based detectors, i.e., InfoLeak CIVs that expose higher-level secrets, as well as Logic CIVs. We explore LLM agents to mechanize the manual audit and facilitate the discovery of such bugs. Recent works have shown the potential of LLM agents to find security bugs [28, 69], but to the best of our knowledge, we are the first to use them at compartment interfaces. We use OpenClaw [56]

with Claude 4.6 Sonnet, supplied with a copy of the OpenBSD code-base.

Prompt. Our prompt (§A) is structured in four parts. First, we load context on compartmentalization and CIVs and supply a preliminary draft of this paper. Second, we deploy two separate audit instances: (1) a leak-targeted audit focusing on InfoLeak CIVs and uninitialized-memory leaks, and (2) a logic-targeted audit focusing on Logic CIVs. This also lets us be explicit about the threat model, covering both the high-level threat model (§4.1) and its implications for each audit target, as we find agents to otherwise divert into finding non-privilege-escalation attacks. Third, we provide LLM agents with a methodology that mirrors our manual audit (§4.2): extract developer intentions and compartment and interface semantics, then find places in the code that contradict them. We instruct LLM agents to focus on a specific set of programs (Table 2). Finally, we request a report detailing the CIVs found.

Cost. Running one audit instance for information leakage (covering leaks classified as MemSaf or InfoLeak) or one for Logic CIVs costs about 60M tokens. Though this is a large number (for reference, this represents about 45M English words [16]), the audit completes in 20 minutes at a cost of ~US$15.

Accuracy. We observe variable accuracy across different types of CIVs. In line with our methodology (§4.2), we carefully review each report manually. Out of the six Logic CIVs reported by LLM agents, three are high-impact bugs that we missed in the manual audit, and three are false positives. Two false positives are due to an incomplete understanding of the trust relationships and privileges associated with each compartment. For example, agents reported a logic bug in OpenBGPD that allows compartments to block the update of kernel routing tables. However, they did not consider that these compartments can already block the update of kernel routing tables through other legitimate means, rendering the vector non-escalatory. The remaining false positive is due to LLMs failing to understand `unveil` (§3.2) and falsely claiming a capability that was, in fact, restricted. Interestingly, agents perform better as we make §3.3.3 more comprehensive: adding V7 unlocks the finding of one of the three true positives. On the other hand, all ten leakage bugs reported by the information-leakage audit are true positives: eight fall under InfoLeak, while two are MemSaf CIVs involving uninitialized memory. We suspect that this is because this class of issues is more localized in the code and depends less on higher-level logic, making it easier for LLMs to reason about.

6 CIV Findings & Security Impact Analysis

We now present the results of our study, answering the research question *"What are the limitations of these compartmentalization techniques, and how can they lead to CIVs?"*. We describe a total of 81 CIVs in 19 programs, 12 found by manual auditing, 37 by fuzzing, 13 by LLM agents, and 19 by the OpenBSD developers[3], leading to 49 fixes at the time of writing. Table 3 summarizes our results, and our artifact (§A) provides a full description of the bugs. We present the CIVs along two orthogonal dimensions: (1) the security impact categories introduced in §2.2 which reflect the severity of the attack and align with our impact-driven CIV discovery methodology (§4.2), and (2) the root causes, which pinpoint the specific failure in the interface design or implementation leading to the CIV.

[3]Note: these are not false negatives. Following our reports, OpenBSD developers independently audited a larger set of programs. For the programs we later covered, we credit them for the issues they already found even if our approach can detect them.

6.1 Pure Memory Safety Violations

MemSaf CIVs are the most common type of CIVs we observe (64/81 bugs), and they include some of the most severe bugs in our corpus. They arise from misuse of attacker-controlled memory-related information or exposure of uninitialized memory. While our CIV dataset is dominated by MemSaf CIVs, many of their root causes go beyond lacking memory-safety checks, stressing the need for a more systematic understanding of CIVs and their mitigations. We observe five recurring root-cause patterns.

6.1.1 Use of Corrupted Indexing Information: These CIVs arise when an attacker-controlled pointer, length, or size influences memory accesses due to insufficient V1-, V2- or V3-type validations. We observe 21 cases falling into two patterns.

- *Corrupted Array Index.* We observe six cases where arbitrary malicious values are used to address an array, five sharing a common root cause due to code duplication (B6, B10, B17, B21, and B22). The direct impact of this pattern is an out-of-bounds read or write. In the most severe of these cases (shown in Listing 1), the over-read value is later interpreted as a function pointer, potentially leading to arbitrary code execution. A notable feature of these bugs is that the vulnerable index does not represent an offset in the protocol semantics. In all six cases, it is instead an application-level identifier that is used by the compartment to index data in an array. This is error-prone as it obscures the invariant that must be enforced, may expose under-filled or uninitialized array entries (if the array is not fully-populated), and does not naturally encourage bounds checks.
- *Corrupted Size Information.* We observe 15 cases of incorrect validation of the size of objects in the payload of messages. For example, in B2 and B19 (×4), the receiver failed to validate the size field inside a `sockaddr`-style payload, while in B34 and B35, the receiver failed to validate the size of a string. These result in stack and heap over-reads, heap over-writes, as well as one case of stack buffer overwrite with attacker-controlled data when the attacker-controlled size was used as the length argument of `memcpy` (further discussed in §6.4.1).

6.1.2 Use of Corrupted Object. These CIVs arise from using an insufficiently-validated object (V3). We identify three patterns. Certain corrupted object size fields (e.g., in a `sockaddr` object) enter this category; we already cover them in §6.1.1. The two other patterns regard strings and numeric fields. In principle, CIVs in this category can be arbitrarily complex. However, privileged compartments in our corpus are designed to avoid interactions with complex objects, explaining why we find only relatively simple bugs in this category.

- *Corrupted String.* We observe 28 cases of missing `NUL` terminator checks (B1, B3 ×2, B8, B9 ×2, B11, B23 ×6, B24 ×7, B26, B37, B38 ×3, B40 ×3), highlighting the difficulty in systematically validating strings. These bugs cause heap and stack over-reads that, in some cases, leak memory contents into log files.

Table 3: CIVs found in OpenBSD privilege-separated programs. # shows the number of CIVs consolidated per line. *Sender* shows the name of the de-privileged compartment that can exploit the CIV in the *Victim* compartment. We classify CIVs according to their *Security Impact* (cf. §2.2) and summarize their observed effect under *Concrete Impact*. We also report the *Root Cause* of each issue, including the faulty or missing checks (*Check Missed*) that enable the bug. *Concrete Impact*: OOB = out-of-bounds. *Found By*: M: manual, F: fuzzer, L: LLM, O: OpenBSD developers. *Fixed*: ● = yes, ○ = no.

Program	#	Bug ID	Threat Model		Security Impact		Explanation			Found By	Fixed
			Sender	*Victim*	*Class*	*Concrete Impact*	*Root Cause*	*Note*	*Check Missed*		
OpenBGPD	1	B1	rde	OpenBGPD	MemSaf	Stack OOB read → over-read data logged	Corrupted Object	Un-term String	V3	**F**	○
	1	B2	rde	OpenBGPD	MemSaf	Stack OOB read/write	Corrupted Indexing, Corrupted Object	Size Information	V1, V3	**F**	○
	2	B3	session	OpenBGPD	MemSaf	Stack OOB read → over-read data logged	Corrupted Object	Un-term String	V3	**M**	●
	1	B4	session, rde, rtr	OpenBGPD	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**F**	●
	1	B5	rde	OpenBGPD	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**L**	○
relayd	1	B6	*any*	parent	MemSaf	Heap OOB R/W → arbitrary code exec.	Corrupted Indexing, Excessive RPC Exposure	Array Index	V2, V4	**F**	○
	1	B7	hce	parent	MemSaf	Heap OOB Read	Corrupted Indexing	Array Index	V2	**F**	●
	1	B8	hce	parent	MemSaf	Stack OOB read → over-read data logged	Corrupted Object	Un-term String	V3	**M**	●
	2	B9	pfe	parent	MemSaf	Stack OOB read	Corrupted Object	Un-term String	V3	**M**	●
httpd	1	B10	*any*	parent	MemSaf	Heap OOB R/W → arbitrary code exec.	Corrupted Indexing, Excessive RPC Exposure	Array Index	V2, V4	**F**	○
	1	B11	logger	parent	MemSaf	Heap OOB read → over-read data printed	Corrupted Object	Un-term String	V3	**F**	○
	1	B12	server	parent	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**M**	●
	1	B13	server, logger	parent	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**F**	●
	1	B14	server	parent	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**F**	●
	1	B15	server	parent	MemSaf	Uninit. stack leak	Sharing Under-Filled Buffers/Compiler Padding		V8	**F**	●
	1	B16	server, logger	parent	MemSaf	Uninit. stack leak	Sharing Under-Filled Buffers/Compiler Padding		V8	**F**	●
OpenIKED	1	B17	*any*	parent	MemSaf	Heap OOB R/W → arbitrary code exec.	Corrupted Indexing, Excessive RPC Exposure	Array Index	V2, V4	**F**	○
	1	B18	ikev2	parent	MemSaf	NULL deref.	Faulty Error Handling		N/A	**F**	○
	4	B19	ikev2	parent	MemSaf	Stack/heap OOB read	Corrupted Indexing, Corrupted Object	Size Information	V1, V3	**F**	○
	1	B20	ikev2	parent	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**L**	○
snmpd	1	B21	*any*	parent	MemSaf	Heap OOB R/W → arbitrary code exec.	Corrupted Indexing, Excessive RPC Exposure	Array Index	V2, V4	**F**	○
vmd	1	B22	*any*	vmd	MemSaf	Heap OOB R/W → arbitrary code exec.	Corrupted Indexing, Excessive RPC Exposure	Array Index	V2, V4	**F**	●
	6	B23	control	vmd	MemSaf	Stack/heap OOB read	Corrupted Object	Un-term String	V3	**O**	●
	7	B24	vmm	vmd	MemSaf	Stack OOB read	Corrupted Object	Un-term String	V3	**O**	●
OpenSMTPD	3	B25	*any*	priv	MemSaf	NULL deref.	Faulty Error Handling		N/A	**F**	●
	1	B26	*any*	priv	MemSaf	Heap OOB read	Corrupted Object	Un-term String	V3	**O**	●
	1	B27	lka	queue	Logic	Delete emails in queue	Excessive RPC Exposure		V4	**M**	○
	1	B28	*any*	priv	Logic	Arbitrary cmd exec. as non-root	Confused Deputy		V7	**L**	○
rpki-client	1	B29	parser	rpki-client	MemSaf	NULL deref.	Faulty Error Handling		N/A	**F**	●
tmux	1	B30	client	server	MemSaf	NULL deref.	API Ordering Issues		V5	**F**	●
	1	B31	client	server	MemSaf	Use-after-free	Faulty Error Handling		N/A	**F**	●
OpenNTPD	2	B32	ntp	priv	MemSaf	Undefined numeric conversion	Corrupted Object	Invalid Numeric	V3	**F**	●
	1	B33	ntp	priv	MemSaf	Undefined numeric conversion	Corrupted Object	Invalid Numeric	V3	**F**	○
radiusd	1	B34	main	priv	MemSaf	Heap OOB write	Corrupted Indexing	Size Information	V2	**M**	●
	1	B35	main	priv	MemSaf	Heap OOB write	Corrupted Indexing	Size Information	V2	**O**	●
	2	B36	main	priv	MemSaf	Heap OOB Write	Corrupted Indexing	Size Information	V1, V2	**F**	○
	1	B37	main	priv	MemSaf	Heap OOB read	Corrupted Object	Un-term String	V3	**F**	○
dhcpleased	3	B38	engine	dhcpleased	MemSaf	Stack OOB read	Corrupted Object	Un-term String	V3	**O**	●
	1	B39	engine, frontend	dhcpleased	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**L**	●
mountd	3	B40	parent	priv	MemSaf	Heap OOB read	Corrupted Object	Un-term String	V3	**M**	○
	1	B41	parent	priv	MemSaf	Uninit. stack leak	Sharing Under-Filled Buffers/Compiler Padding		V8	**M**	●
	2	B42	parent	priv	Logic	Export arbitrary path to network	Confused Deputy		V7	**L**	○
OpenSSH	1	B43	ssh unpriv. child	monitor	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**O**	○
tcpdump	1	B44	tcpdump	priv	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**L**	●
eigrpd	4	B45	eigrp engine, rde	eigrpd	InfoLeak	Pointer leak	Sharing Pointer-Containing Structures		V8	**L**	○
	1	B46	eigrp engine	eigrpd	MemSaf	Heap OOB read *or* NULL deref.	Corrupted Indexing	Size Information	V1	**F**	●
dvmrpd	2	B47	rde	dvmrpd	MemSaf	Uninit. stack leak	Sharing Under-Filled Buffers/Compiler Padding		V8	**L**	○
	1	B48	dvmrp engine	dvmrpd	MemSaf	Heap OOB read *or* NULL deref.	Corrupted Indexing	Size Information	V1	**F**	●
ospfd	1	B49	ospf engine	ospfd	MemSaf	Heap OOB read *or* NULL deref.	Corrupted Indexing	Size Information	V1	**F**	●
	1	B50	rde	ospfd	MemSaf	Heap OOB R/W *or* NULL deref.	Corrupted Indexing	Size Information	V1	**F**	●
	1	B51	rde	ospfd	MemSaf	Heap OOB read *or* NULL deref.	Corrupted Indexing	Size Information	V1	**F**	●
isakmpd	1	B52	child	monitor	MemSaf	Heap OOB read	Corrupted Indexing	Size Information	V1	**F**	●

Total: 19 programs; **81** bugs *(7 programs with no CIVs found)* **M** 12 **F** 37 **L** 13 **O** 19 ● 49 ○ 32

- *Invalid Numeric Representation.* We find three cases of missing numeric value checks (B32 ×2, B33). While these issues cause undefined behavior during numeric conversion, none are exploitable with the compiler used in OpenBSD.

6.1.3 API Ordering Issues. CIVs also arise from violations of protocol message ordering due to lacking V5 validations. We observe this in `tmux` (B30), whose server incrementally initializes per-client state via a series of `MSG_IDENTIFY_*` setup messages terminated by `MSG_IDENTIFY_DONE`. The handler for the final message accesses fields (e.g., `ttyname`) without checking that the prerequisite setup message (e.g., `MSG_IDENTIFY_TTYNAME`) was processed. A bad client can thus send a final message early to crash the server on uninitialized fields. Although prior work identified API ordering as a potential CIV class [44], we are the first to identify such CIVs in privilege-separated programs.

6.1.4 Faulty Error Handling. We observe six cases (B18, B25 ×3, B29, B31) where CIVs are due to bugs in the code that runs after a check fails as opposed to faulty checks. Consider a case where a victim compartment uses a helper function that returns an error code after processing attacker-controlled inputs. However, the caller fails to handle the error and resumes normal processing, resulting in the use of corrupted or partially initialized state, causing a use-after-free (e.g., B31) and `NULL`-pointer dereferences (e.g., B18, B29). These

issues show that error-path CIVs observed in prior works [44] on unhardened interfaces also exist at carefully-designed boundaries, and further motivate checking inputs as early as possible.

6.1.5 Under-Filled Buffers and Compiler Padding. We observe five cases of uninitialized memory leakage (B15, B16, B41, B47 ×2), all caused by a lack of buffer zeroing for outgoing messages, as required by V8. Although they may leak potentially sensitive content, we classify them as MemSaf CIVs, because they all involve uninitialized memory and thus violate memory safety. These come in two patterns. The first is *under-filled buffers*, where the message contains buffers that are not fully populated (e.g., strings stored in fixed-size char arrays) or buffers that are only filled under certain conditions (e.g., upon successful operation). The second is *compiler padding*, where the sender transmits an entire struct as-is, unintentionally including padding bytes between fields. For example, in `dvmrpd` (B47), the privileged compartment sends a stack-allocated `route_report` to unprivileged compartments with both uninitialized fields and padding bytes. `mountd` (B41) also leaks uninitialized buffer contents, because the message buffer is populated only upon operation success but also sent in the case of failure. These bugs demonstrate that leakage due to uninitialized memory is practical at compartment interfaces and highlight the difficulty of reasoning systematically about the state of outgoing memory. Similar to §6.1.3, under-filled buffers and compiler padding were identified as potential CIV patterns by prior work [44], but we are the first to identify concrete instances in privilege-separated programs.

6.2 Non-Memory-Safety Information Leakage

We observe 13 InfoLeak CIVs, all manifesting as pointer leakage. Although other kinds of non-memory-safety information leaks are possible, we did not find any.

6.2.1 Pointer-Containing Structures. The root cause is a lack of sanitization of pointer fields within outgoing messages, which violates V8 and leads to leaking pointer values to unprivileged compartments. This is another case in which prior work proposed a potential CIV pattern [44] but did not identify concrete instances of it. Since OpenBSD relies on process separation, leaking pointers does not break isolation per se, but it enables breaking cross-compartment ASLR (§3.2). We found 13 such CIVs (B4, B5, B12–B14, B20, B39, B43, B44, and B45 ×4). All of them fall in a pattern where the privileged compartment needs to send a complex object already represented as a C struct (e.g., configuration objects). Instead of constructing a sanitized representation or explicitly clearing pointer fields before sending, it sends the object as-is. For example, in B43, OpenSSH's privileged compartment sends a 1912-byte `Options` struct containing 48 pointer fields to unprivileged compartments. To the best of our knowledge, despite OpenBSD developers' awareness of the risk of leaking pointers from directly sharing objects, OpenBSD programs did not perform explicit pointer-field clearing prior to our work.

The security relevance of pointer leakage depends on the availability of cross-compartment ASLR (§3.2). Many OpenBSD programs do not apply `fork+exec` systematically due to the additional engineering effort it requires: `exec` starts a compartment from a clean state, so the new compartment cannot inherit initialization state from the parent and requires explicit re-initialization logic. As a result, inter-compartment ASLR is only partially deployed in OpenBSD – we report issues only in programs that come with this protection. This highlights the relevance of the issues we find, as they break a mechanism that came at high engineering cost.

6.3 High-Level Logic Violations

We also observe Logic CIVs. We find four such issues via manual auditing and LLM agents. Prior work demonstrated similar attacks at other interfaces, such as TEE interfaces [49, 72]. However, to the best of our knowledge, aside from oracle attacks in OpenSSH [51], this class of issues has not been demonstrated in privilege-separated software.

6.3.1 Excessive RPC Exposure. A privileged compartment may expose excessive RPCs when V4 is implemented improperly. For example, OpenSMTPD's compartments do not authenticate the sender of incoming messages, allowing any unprivileged compartment to invoke handlers that it is not designed to access. In B27, an attacker who controls the `lka` unprivileged compartment can invoke queue-management handlers and delete queue entries, an operation that OpenSMTPD explicitly prohibits for that compartment [23]. Beyond directly causing CIVs, the lack of sender authentication also widens the set of compartments from which other CIVs can be exploited. For example, the vulnerable endpoint in B28 should be called only from one specific compartment, but a lack of authentication makes it callable by any unprivileged compartment (more in §6.4.2). A similar pattern appears in the CIV of Listing 1 (B6, B10, B17, B21, B22): the vulnerable endpoint is meant to be invoked only by privileged compartments to run inside unprivileged compartments. However, because it is registered in all compartments, unprivileged compartments can also invoke it in the privileged compartment, making the CIV exploitable.

6.3.2 Confused Deputy. A privileged compartment can become a confused deputy when V7 checks are improperly implemented. We observe three cases in OpenSMTPD (B28) and `mountd` (B42 ×2), all affecting interfaces where the privileged compartment exposes a mediated subset of privileged operations. These messages can be exploited by unprivileged compartments as gadgets to perform privileged operations that are outside their designed authority, partially or fully escaping isolation. For example, in B28 (further discussed in §6.4.2), unprivileged OpenSMTPD compartments can misuse the API to run arbitrary shell commands as arbitrary non-root users. In B42, `mountd` allows the de-privileged compartment to export an arbitrary path to the network with arbitrary permissions.

Addressing these issues requires non-trivial changes. For example, B28 affects an interface allowing transmission of shell commands. When the privileged compartment can determine that the shell command is safe (i.e., is it a mail delivery agent?), the interface is safe. However, in B28, the privileged compartment does not have the necessary information required to perform this safety check.

6.4 Case Studies

6.4.1 Stack Over-Write in OpenBGPD (B2). OpenBGPD is a popular BGP daemon with four compartments, one of which is privileged. B2 enables a malicious unprivileged compartment to over-write

```
struct bgpd_addr ② { // attacker-controlled (simplified)
  uint8_t labellen;  /* size of the labelstack */
  uint8_t labelstack[18];
};

const char *log_evpnaddr(const struct bgpd_addr *addr ①, ...) {
  // ... simplified
  uint32_t  vni; ③
  // ... simplified
  memcpy(&vni, addr->labelstack, addr->labellen);
  // ... simplified
}
```

Listing 2: Vulnerable endpoint for B2 (§6.4.1).

```
static void forkmda(..., struct deliver *deliver) {
  uid_t pw_uid;
  if (dsp->u.local.user) {/* use config-specified user */}
  else {
    pw_uid = deliver->userinfo.uid; ①
    // ... other user info
  }
  pid = fork(); ②
  if (pid > 0) {/* parent */}
  setgroups(1, &pw_gid); // also set other user ids ③
  mda_unpriv(..., deliver, ...); ④
}

void mda_unpriv(..., struct deliver *deliver, ...) {
  const char     *mda_command;
  if (deliver->mda_exec[0]) { mda_command = deliver->mda_exec; }
  execle("/bin/sh", "/bin/sh", "-c", mda_command, ...); ⑤
}
```

Listing 3: Vulnerable endpoint for B28 (§6.4.2).

stack data in the privileged compartment with arbitrary data. Listing 2 shows the vulnerable endpoint `log_evpnaddr`, which runs in the privileged compartment. It performs operations on `addr` that points to a sender-controlled `struct bgpd_addr` (②), whose `labellen` field specifies the size of `labelstack`. In `log_evpnaddr`, the privileged compartment uses the sender-controlled `labellen` as the size argument to `memcpy` when copying into `vni` ③ without checking that it is within the maximum size of `labelstack`. This leads to over-writing the stack beyond `vni` with attacker-controlled data from the `labelstack` field. Worse, because `bgpd_addr` sits in the middle of a larger sender-controlled message, the attacker controls enough bytes beyond `labelstack` to overwrite the return address on the stack with attacker-controlled values. Fortunately, OpenBGPD applies the `fork+exec` approach to implement inter-compartment ASLR, and OpenBSD enables stack canaries by default, requiring an attacker to obtain additional CIVs (such as the ones we describe in §6.2) to hijack the control flow.

6.4.2 Arbitrary Shell Command Execution in OpenSMTPD (B28). OpenSMTPD is a popular email server comprising 7 compartments. Users can program it to pipe incoming e-mails to shell commands, e.g., a mail delivery agent. The unprivileged mail delivery compartment cannot run commands as the recipient user, so the privileged compartment exposes an RPC to run a given command as a given user. Listing 3 shows the privileged handler for this RPC: it first retrieves user information from the `deliver` struct ①, forks a child process ② and switches to the sender-controlled user identity ③ (except for the root user). Finally, it enters the child process's logic ④, which eventually runs the specified shell command without validation ⑤. This enables the unprivileged compartment to run arbitrary commands under any non-root user, defeating isolation. This CIV has been present for 12 years.

6.5 Key Insights

6.5.1 Avoiding Shared Memory Eliminates Entire Classes of CIVs. OpenBSD's first compartmentalized program, OpenSSH, originally used shared memory for communication among compartments. Shared memory was later removed following a CIV [9], and all subsequent compartmentalized programs followed this design choice. This *eliminates classes of CIVs such as time-of-check-to-time-of-use and synchronization*, which are problematic in shared-memory schemes and for which *we do not have systematic solutions* [44].

6.5.2 Hardening Changes the Prevalence of Issues, Not Their Nature. Aside from CIVs eliminated by avoiding shared memory, the bugs we observe are largely similar to those observed by prior works at interfaces that were not designed for distrust [44]. The key difference is prevalence: on average, we find ~3 CIVs per program (including 7 programs with none), significantly better than that reported for non-hardened interfaces [44]. This shows that *manual hardening is effective, albeit insufficient to eliminate these issues.*

6.5.3 OpenBSD's CIV Mitigations Remain Unsystematic. OpenBSD has accumulated substantial experience with CIVs, as shown by our systematization (§3.3.3). However, these validations are still applied ad-hoc – a key reason why CIVs continue to arise in OpenBSD.

- In particular, *known checks are not consistently applied.* This is true even for classes of checks that OpenBSD developers have long recognized as necessary. For example, OpenSSH has been performing string termination checks since 2002 [7], yet they still account for a large part of our findings (§6.1.2). `vmd` omitted them altogether (B23 ×6, B24 ×7), and retrofitting them required changing nearly 1KLoC [74]. Some of these bugs remained undetected for years: e.g., 21 years for B1 (OpenBGPD).
- Similarly, *it is hard to turn problems into lasting practices.* Ten years before our study, OpenSMTPD underwent an external security audit that found multiple CIVs, including uninitialized memory leaks and memory errors caused by trusting sender-controlled size information and objects [60]. These issues were patched [8], but did not lead to deeper, systematic improvements. Our findings show that the same CIV patterns identified a decade ago still recur across OpenBSD in substantial numbers.
- Moreover, *getting the checks right is hard.* CIVs caused by the receiver failing to perform necessary checks on inputs (V1–V7) make up the majority of our corpus (57/81 cases). Interestingly, in 18% of the cases (10 bugs, B7, B19 ×4, B34, B35, B36 ×2, and B37), the receiver partially checked the object, albeit insufficiently. In `relayd` (B7) for example, developers implemented a range check on an array index but covered only the positive range. This hints that forcing developers to check objects is necessary but not sufficient to prevent CIVs.

6.5.4 Extracting Semantics is Hard. The availability of compartment semantics is a key bottleneck in finding CIVs at scale, particularly for Logic CIVs (§6.3). Developer documentation was invaluable for our manual audit, enabling the finding of B27. Although

Table 4: Coverage of validation classes by existing mitigations. ● = fully addressed, ◐ = partially, ○ = not addressed.

Existing Solutions	V1	V2	V3	V4	V5	V6	V7	V8
Syntax-Aware RPC Frameworks	◐	○	◐	○	○	○	○	◐
RLBox & Tainted Types	◐	◐	◐	●	○	○	◐	◐
Memory-Safe Languages	●	●	◐	○	◐	◐	○	◐
Hardware Capability Systems	●	●	◐	◐	◐	◐	◐	○

we find LLM agents more efficient than humans at extracting semantics, their error rate limits their utility to expert use (§5.2). This hints that more interface specifications, even in human language, would enable stronger defenses.

7 Discussion: Towards Systematic Defenses

This leads us to our last question: *How can we systematically address CIVs?* We discuss existing defenses based on our findings (§7.1), and conclude with avenues towards more complete solutions (§7.2).

7.1 Other Defenses

How systematic are other existing defenses, and which of these issues would they have prevented?

Syntax-Aware RPC Frameworks. Certain RPC frameworks allow developers to specify message formats and automatically generate the corresponding parsing and serialization code. A representative example is Protocol Buffers (protobuf) [4], used by gRPC [1]. Such frameworks can use their awareness of syntax to systematically enforce syntax-level validations (e.g., string termination), partially performing V1 and V3. However, their lack of awareness of semantics prevents them from fully replacing V1 and V3, as well as other checks that require semantic or policy information (V2, V4–V7). They also know the syntax of outgoing messages and could sanitize certain sensitive fields such as pointers (although existing frameworks do not) but cannot prevent the leakage of other sensitive data, thus only partially replacing V8. Overall, protobuf mitigates 28 out of 81 CIVs (the 28 missing-NUL-terminator CIVs).

RLBox and Tainted Types. RLBox [55] mitigates CIVs by treating data received from untrusted compartments as tainted, automating simple syntax checks, and forcing manual verification for V1–V3 and V7. Still, RLBox cannot guarantee the correctness of these checks and, as shown in §6.5.3, manual checks can fail. RLBox can also sanitize outgoing data (V8); for example, it prevents a compartment from sending out raw pointers and requires developers to sanitize the output, but the effectiveness still depends on the manual sanitization logic. Finally, while RLBox can enforce a form of caller authentication (V4), it mandates checks on interface-crossing data, not on control flow, and thus cannot prevent faulty V5 or V6 validations or faulty error paths. Counting only automated checks, RLBox mitigates 41 CIVs (28 missing-NUL-terminator CIVs and 13 pointer leakages). Granted all manual checks are correct, RLBox mitigates 74 out of 81 CIVs caused by missing V1–V4, V7, or V8.

Memory-Safe Languages. Rust completely replaces validations directly related to memory safety (V1, V2). However, it cannot replace V3, V5, and V6, since missing these can also manifest as logical errors. Similarly, missing V4 and V7 leads to logic errors that memory-safety cannot prevent. Although Rust forbids the use of uninitialized memory, partially covering V8, it cannot prevent the leakage of pointers and other sensitive data. Moreover, memory-safe languages usually fail hard on errors by default, whereas programs such as OpenBGPD need to recover rather than terminate on errors, still requiring non-trivial error-handling logic. Safe Rust mitigates 64 out of 81 CIVs (all MemSaf CIVs).

Hardware Capabilities. CHERI [15, 75, 77] provides memory-safety guarantees, fully enforcing V1 and V2. Using CHERI facilitates, but does not fundamentally enforce, V4. For V3, V5, and V6, CHERI mainly helps when a missing check manifests as a memory violation; pure logic issues remain out of reach. CHERI also does not perform V8; in fact, pointer leaks are more severe since pointers are themselves capabilities granting additional privileges. Although capabilities can address confused-deputy problem [32] (V7) CIVs (§7.2), it requires representing inter-compartment interactions as capability delegations rather than message passing and is not automatic. CHERI mitigates 59 out of 81 CIVs (all MemSaf CIVs except the five involving uninitialized memory).

7.2 Avenues for Systematic Mitigation

In principle, combining syntax-aware RPC frameworks, tainted types, and safe languages would mitigate most of the issues we observe. However, they still largely rely on developers writing correct and complete manual checks—which, as §6.5.3 shows, can fail. Below we outline three avenues towards closing this gap and achieving (more) systematic mitigations.

Avenue 1: Interface and Compartment Specifications. Complete defenses require complete specification of interface semantics. These fundamentally cannot be extracted from source code and have to be provided by developers. Ideally, semantics would be provided in a formal way to enable the generation of complete interface hardening and the automated testing of interfaces. However, as we show in §6.5.4, semantics in human language already bring added value in facilitating manual and LLM-supported audits.

Avenue 2: Lightweight Specifications for Best-Effort Mitigation. A full specification is not necessary to go most of the way. 30% (24/81) of CIVs stem from missed validations that cannot be addressed without understanding interface or compartment semantics. However, 21 of these 24 cases require only a small set of lightweight hints: (1) embedded sizes, (2) array indices, (3) API ordering, and (4) API authentication. Extending syntax-aware RPC frameworks with such hints would substantially increase their mitigation power (mitigating 78/81 bugs) at modest engineering cost. Most checks can be generated directly from the declared semantics; runtime-dependent array bounds can use developer-provided callbacks or framework-generated runtime checks.

Avenue 3: Capabilities. Confused deputies are the issues remaining after a lightweight specification (3/81 bugs, B28, B42 ×2), and capabilities are the natural solution to address them [32, 52]. CHERI systems have explored this [15, 75, 77], but a similar effect can also be achieved with simpler capability schemes. Capsicum-like [76] software capabilities would be sufficient in the case of the bugs we observe. We observe that much of the information needed to

provision such capabilities is already available at startup: in the confused-deputy cases we observed, legitimate targets are largely enumerated in configuration files that the privileged compartment already parses (e.g., `/etc/exports` for `mountd`'s exportable directories). This makes the privileged compartment a natural place to provision and delegate capabilities up front and then drop the underlying privileges (in the spirit of `pledge`, §3.2).

8 Threats to Validity

We now discuss aspects of our methodology that might affect our conclusions and how we mitigate them.

Completeness. We do not cover all OpenBSD privilege-separated programs, nor do we search exhaustively for all possible CIVs (there may be false negatives). Our results therefore do not provide an exhaustive inventory of CIVs or a complete taxonomy of all possible CIV patterns. While being complete would be desirable, this does not affect the value of our contributions. As described in §4.2, we select a representative corpus that includes widely deployed, popular programs and programs that established foundational approaches to modern privilege-separation practices. Covering these programs already allows us to establish our core contributions: demonstrating that CIVs remain a practical concern in real-world privilege-separated software, developing a taxonomy of their recurring patterns, and identifying avenues for stronger mitigations.

Generalizability. Our study focuses on OpenBSD, so our findings may not apply to all possible privilege-separated programs. Still, there are strong indicators that our results generalize beyond OpenBSD. First, 11 of the 25 programs in our corpus, including OpenSSH and OpenSMTPD, have been ported to, and are widely deployed on, other systems, such as Linux. Second, OpenBSD's privilege-separation approach, pioneered by OpenSSH, established foundational design patterns that were later adopted by many other privilege-separated programs [46, 59]. The CIV patterns we identify arise from common design choices established by this approach, such as message-based interfaces. Our taxonomy therefore applies to many other programs inspired by OpenBSD's design. Third, our discussion in §7 derives general principles for systematically mitigating CIVs in future privilege-separated programs.

Fuzzing on FreeBSD. Because we fuzz OpenBSD programs ported to FreeBSD, findings are not guaranteed to hold on the original OpenBSD program (e.g., due to different security mechanisms). We validate findings by constructing a PoC for every CIV in the original OpenBSD program and reporting each issue to the OpenBSD security team (cf. §4.2). The OpenBSD security team acknowledged all issues. B19 is the only bug whose impact differs between FreeBSD and OpenBSD (over-read and over-write on FreeBSD but only over-read on OpenBSD). Note that, in this case, the FreeBSD version is an official port used in production in other OSes.

9 Related Works

Compartment Interface Vulnerabilities (CIVs). Several works have studied bugs at the interfaces of compartments [24, 34, 36, 44, 51]. Except Marchenko et al. [51], all focus on MemSaf CIVs arising when retrofitting compartmentalization at interfaces *that were not designed as a trust boundary*. Marchenko et al. [51] focus on specific kinds of Logic CIVs in compartmentalized cryptographic software. In contrast, we study the full spectrum (i.e., MemSaf, InfoLeak, and Logic) of real-world CIVs that arise at interfaces carefully designed by security experts to prevent CIVs. This allows us to cover CIV classes that were not categorized in prior taxonomies [44], such as faulty error handling, excessive RPC exposure, and confused-deputies. Together, these findings allow us to understand the effectiveness of existing CIV defenses and how we can improve them, a step towards making compartmentalization more widespread [46].

Bugs at Other Interfaces. Prior works have demonstrated related attacks at other (non-compartment) interfaces, e.g., TEEs [49, 72] or the system call API [22]. While these attacks are similar in principle, interfaces differ in nature, motivating dedicated work on CIVs.

CIV Defenses. The traditional way to prevent CIVs relies on security experts carefully designing the interfaces of compartments [31, 40, 59]. This is the approach taken by the OpenBSD software we evaluate in this study. Several works have contributed techniques to partially automate this manual effort. RPC frameworks such as gRPC [1] automate syntax-level checks. RLBox's tainted types [55] provide a compiler-based mechanism to force developers to check interface-crossing data. CHERIoT [15] builds on hardware capabilities [75, 77] to provide new hardware-software primitives that facilitate the design of strong interfaces. As discussed in §7.1, all are good steps forward but address only part of the issues we observe. Our study enables viewing these works in a new light, highlighting which part of the problem space they can and cannot address.

Bug Finding and Compartments. Prior works used fuzzing [18, 44, 66], static analysis [24, 36], and symbolic execution [34] to find bugs at compartment interfaces. Other closely-related works explored fuzzing system-level RPC interfaces [50]. While prior fuzzing works influenced our approach, fuzzing process-based compartments as found in OpenBSD presents unique challenges that require a new design (cf. §5.1). Static analysis and symbolic execution present different trade-offs compared to the techniques we use. While both are more systematic, scaling them to large and complex system software is challenging. Last, while LLMs increasingly gain traction for code analysis [28, 69], to the best of our knowledge, they have not been applied in the context of compartments prior to this study. Overall, there remains a lack of bug finding techniques for compartmentalized software [46]. We conclude on a hopeful note by showing that one can build effective bug finding tools for compartment interfaces from off-the-shelf components.

10 Conclusions

We presented the first systematic study of bugs affecting compartment boundaries in real-world applications. We audited OpenBSD's rich corpus of compartmentalized programs to find 81 bugs that weaken or bypass isolation. Our results show that even mature compartmentalized software designed by security experts remains vulnerable to recurring interface bugs as severe as code execution. We highlight both the wisdom and weaknesses of OpenBSD's approach: avoiding shared memory eliminates entire classes of bugs, but ad-hoc validations remain vulnerable. We conclude with a call for more systematic validations, well-specified interfaces, and wider use of capabilities in compartment interface design.

Acknowledgments

We thank the anonymous reviewers for their comments and insights, and our colleague Caroline Lemieux for her insights on fuzzing. We are immensely grateful to the OpenBSD community for their contributions to privilege separation and receptiveness to this work. We acknowledge that Shibo Ai is supported by SNSF grant 200021-236559. We also acknowledge support from the Natural Sciences and Engineering Research Council of Canada (NSERC).

A Open Science

Our artifact is available open-source at https://github.com/ubc-systopia/openbsd_civ_artifact. It includes (1) sources of our fuzzing harness and framework (§5.1), along with documentation to exercise them for the applications listed in Table 2; (2) the prompts we designed for auditing with LLM agents (§5.2); and (3) extended descriptions of each of the bugs we reported (listed in Table 3 with **M**, **F**, or **L** in the *Found By* field), exclusive of proof-of-concepts for ethical reasons (see below).

B Ethical Considerations

We do not collect user data; our study strictly concerns bugs already present in open-source software. Still, there is an inherent risk in developing tools that can find bugs in popular software. We mitigate this with responsible bug reporting practices: we give developers detailed information (technical report, PoCs) and time (>90 days) to address issues, and suggest patches when possible. As a result, most bugs have already been addressed (Table 3). Furthermore, we do not publish PoCs. Lastly, increasing awareness of these issues and how to fix them will also strengthen the software ecosystem. We believe that this is enough to balance the risks.

The use of LLM agents to find bugs itself has ethical implications: LLMs are highly resource-intensive and their development and deployment carry social and societal implications [43]. We believe that the social and societal benefits of making software significantly more trustworthy through this work outweigh that cost.

References

[1] [n. d.]. gRPC: A high performance, open source universal RPC framework. https://grpc.io/. Viewed: 2026-04-26.
[2] [n. d.]. OpenBSD. https://www.openbsd.org/. Viewed: 2025-12-30.
[3] [n. d.]. pledge(2) — OpenBSD System Calls Manual. https://man.openbsd.org/pledge.2. Viewed: 2026-01-10.
[4] [n. d.]. Protocol Buffers. https://protobuf.dev/. Viewed: 2026-04-29.
[5] [n. d.]. seccomp(2) — Linux manual page. https://man7.org/linux/man-pages/man2/seccomp.2.html. Viewed: 2026-01-10.
[6] [n. d.]. tcpdump(1) — dump traffic on a network. https://www.tcpdump.org/manpages/tcpdump.1.html. Viewed: 2026-09-03.
[7] 2002. bufaux.c: helpers for operating on buffers. https://github.com/openssh/openssh-portable/blob/V_3_2_2_P1/bufaux.c. Viewed: 2026-04-20.
[8] 2015. Announce: OpenSMTPD 5.7.2 released. https://www.opensmtpd.org/announces/release-5.7.2.txt. Viewed: 2026-04-22.
[9] 2016. OpenSSH 7.4 Release Notes. https://www.openssh.com/txt/release-7.4. Viewed: 2026-04-26.
[10] 2022. OpenSSH 9.0: near miss in sshd(8). https://www.openbsd.org/71.html. Viewed: 2023-09-24.
[11] 2023. double-free vulnerability in OpenSSH (CVE-2023-25136). https://seclists.org/oss-sec/2023/q1/92. Viewed: 2024-03-23.
[12] 2023. OpenSSH 9.2: unprivileged pre-authentication double-free in sshd(8). https://www.openssh.org/txt/release-9.2. Viewed: 2026-01-09.
[13] 2024. OpenSSH 9.8: splitting the sshd(8) server. https://www.openssh.org/txt/release-9.8. Viewed: 2026-04-10.
[14] 2024. Signal handler race condition vulnerability in OpenSSH (CVE-2024-6409). https://nvd.nist.gov/vuln/detail/CVE-2024-6409. Viewed: 2026-02-05.
[15] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2025. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *Proc. of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP'25)*. ACM, 67–84. doi:10.1145/3731569.3764844
[16] Anthropic. [n. d.]. Claude API Pricing. https://platform.claude.com/docs/en/about-claude/pricing. Viewed: 2026-04-26.
[17] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. 2023. TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *Proc. of the 32nd USENIX Security Symposium (USENIX Security'23)*. USENIX Association.
[18] Nils Bars, Lukas Bernhard, Moritz Schloegel, and Thorsten Holz. 2025. Empirical Security Analysis of Software-based Fault Isolation through Controlled Fault Injection. In *Proc. of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS'25)*. ACM. doi:10.1145/3719027.3765027
[19] Markus Bauer and Christian Rossow. 2021. Cali: Compiler Assisted Library Isolation. In *Proc. of the 16th ACM Asia Conference on Computer and Communications Security (AsiaCCS'21)*. ACM. doi:10.1145/3433210.3453111
[20] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, USA, 309–322.
[21] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. of the 13th USENIX Security Symposium* (San Diego, CA) *(USENIX Security'04)*. USENIX Association, USA.
[22] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM. doi:10.1145/2451116.2451145
[23] Gilles Chehade. 2017. OpenSMTPD: Current State of Affairs. https://www.opensmtpd.org/presentations/eurobsdcon2017-smtpd/eurobsdcon2017-opensmtpd.pdf. Viewed: 2026-04-02.
[24] 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 *Proc. of the 1st Workshop on Kernel Isolation, Safety and Verification (KISV'23)*. ACM, 33–40. doi:10.1145/3625275.3625399
[25] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association.
[26] Theo de Raadt. 2016. Privilege Separation and Pledge. In *dotSecurity 2016*. OpenBSD. https://www.openbsd.org/papers/dot2016.pdf Viewed: 2026-01-10.
[27] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *Proc. of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. ACM. doi:10.1145/3576915.3623079
[28] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In *Proc. of the 33rd USENIX Security Symposium (USENIX Security'24)*. USENIX Association, Philadelphia, PA, 829–846.
[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.
[30] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS'15)*. ACM, 1016–1031. doi:10.1145/2810103.2813611
[31] Munawar Hafiz, Ralph Johnson, and Raja Afandi. 2004. The security architecture of qmail. In *Proc. of the 11th Conference on Patterns Language of Programming (PLoP'04)*.
[32] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (1988).
[33] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proc. of the 2019 USENIX Annual Technical Conference (ATC'19)*. USENIX Association, 489–504.
[34] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. 2015. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *Proc. of the 20th European Symposium on Research in Computer Security (ESORICS'15)*, Günther Pernul, Peter Y A Ryan, and Edgar Weippl (Eds.). Springer International Publishing, Cham, 312–331. doi:10.1007/978-3-319-24177-7_16
[35] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The ART of App Compartmentalization: Compiler-Based Library Privilege Separation on Stock Android. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM. doi:10.1145/3133956.3134064

[36] Yongzhe Huang, Kaiming Huang, Matthew Ennis, Vikram Narayanan, Anton Burtsev, Trent Jaeger, and Gang Tan. 2025. Beyond Driver Isolation - Triaging Threats against Driver Isolation. In *Proc. of the 41st Annual Computer Security Applications Conference (ACSAC'25)*.

[37] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *Proc. of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 613–631.

[38] Claudio Jeker. 2009. OpenBGPD: Bringing Full Views to OpenBSD Since 2004. https://www.openbsd.org/papers/asiabsdcon2009-bgpd.pdf. Viewed: 2026-04-02.

[39] Claudio Jeker. 2021. Add RTR support to OpenBGPD. https://cvsweb.openbsd.org/log/src/usr.sbin/bgpd/rtr.c%2Cv?sort=File. Viewed: 2026-04-10.

[40] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *Proc. of the 2003 USENIX Annual Technical Conference (ATC'03)*. USENIX Association, 273–284.

[41] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proc. of the 17th European Conference on Computer Systems (EuroSys'22)*. ACM, 17 pages. doi:10.1145/3492321.3519582

[42] Maxwell Krohn. 2004. Building Secure High-Performance Web Services with OKWS. In *Proc. of the 2004 USENIX Annual Technical Conference (ATC'04)*. USENIX Association.

[43] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the Carbon Emissions of Machine Learning. *arXiv preprint arXiv:1910.09700* (2019).

[44] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2023. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proc. of the 30th Annual Network & Distributed System Security Symposium (NDSS'23)*. doi:10.14722/ndss.2023.24117

[45] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proc. of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS'22)*. ACM, 467–482. doi:10.1145/3503222.3507759

[46] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2025. SoK: Software Compartmentalization. In *Proc. of the 2025 IEEE Symposium on Security and Privacy (S&P'25)*. doi:10.1109/SP61157.2025.00075

[47] Maxwell Levatich and Stephen A. Edwards. 2026. Fast, Flow-Sensitive C Program Partitioning via Iterative Value-Flow Refinement.. In *Proc. of the International Conference on Software Engineering (ICSE'26)*.

[48] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS'17)*. ACM, 2359–2371. doi:10.1145/3133956.3134066

[49] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proc. of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*. Internet Society. doi:10.14722/ndss.2017.23227

[50] Philipp Mao, Marcel Busch, and Mathias Payer. 2025. NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage. In *Proc. of the 34th USENIX Conference on Security Symposium (USENIX Security'25)*. USENIX Association. https://www.usenix.org/system/files/usenixsecurity25-mao.pdf

[51] Petr Marchenko and Brad Karp. 2010. Structuring Protocol Implementations to Protect Sensitive Data. In *Proc. of the 19th USENIX Security Symposium (USENIX Security'10)*. USENIX Association, USA.

[52] Noah Mauthe, Eric Ackermann, and Sven Bugiel. 2026. SoK: Capability Operating Systems: Is the Future Finally Here?. In *Proc. of the 35th USENIX Security Symposium (USENIX Security'26)*. USENIX Association.

[53] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *Proc. of the 29th Annual Network & Distributed System Security Symposium (NDSS'22)*. doi:10.14722/ndss.2022.24078

[54] Damien Miller. 2024. Start the process of splitting sshd into separate binaries. https://marc.info/?l=openbsd-cvs&m=171590564107097. Viewed: 2026-04-10.

[55] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 699–716.

[56] OpenClaw Community. [n. d.]. OpenClaw: Personal AI Assistant. https://openclaw.ai/. Viewed: 2026-04-26.

[57] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proc. of the 2019 USENIX Annual Technical Conference (ATC'19)*. USENIX Association, 241–254.

[58] Dinglan Peng, Congyu Liu, Tapti Palit, Tapti Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. μSWITCH: Fast Kernel Context Isolation with Implicit Context Switches. In *Proc. of the 2023 IEEE Symposium on Security and Privacy (S&P'23)*. doi:10.1109/SP46215.2023.10179284

[59] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proc. of the 12th USENIX Security Symposium* (Washington, DC) *(USENIX Security'03)*. USENIX Association, USA.

[60] Qualys. 2015. Qualys Security Advisory: OpenSMTPD Audit Report. https://www.qualys.com/2015/10/02/opensmtpd-audit-report.txt. Viewed: 2026-04-10.

[61] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Proc. of the 37th Annual Computer Security Applications Conference (ACSAC'21)*. ACM. doi:10.1145/3485832.3485903

[62] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Jonathan M. Smith, Andre DeHon, and Nathan Dautenhahn. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proc. of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID'21)*. ACM, 296–311. doi:10.1145/3471621.3471839

[63] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proc. of the 31st USENIX Security Symposium (USENIX Security'22)*. USENIX Association.

[64] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association.

[65] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. 2025. Building bridges: Safe interactions with foreign languages through omniglot. In *Proc. of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI'25)*. 595–613.

[66] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proc. of the 17th European Conference on Computer Systems* (Rennes, France) *(EuroSys'22)*. ACM. doi:10.1145/3492321.3519591

[67] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (ATC'12)*. USENIX Association.

[68] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)*. USENIX Association, 553–567.

[69] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *ACM Comput. Surv.* 58, 5 (Nov. 2025). doi:10.1145/3769082

[70] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proc. of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE'20)*. ACM, 143–156. doi:10.1145/3381052.3381326

[71] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proc. of the 28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 1221–1238.

[72] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proc. of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM. doi:10.1145/3319535.3363206

[73] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Proc. of the 25th Annual Network & Distributed System Security Symposium* (San Diego, California) *(NDSS'18)*. doi:10.14722/ndss.2018.23131

[74] Dave Voutila. 2025. vmd(8): make imsg objects opaque and sanitize char[]s. https://github.com/openbsd/src/commit/acb6652bd014. Viewed: 2026-04-20.

[75] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proc. of the 2015 IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, 20–37. doi:10.1109/SP.2015.9

[76] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proc. of the 19th USENIX Security Symposium (USENIX Security'10)*. USENIX Association, USA, 29–45.

[77] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf