

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Death by a Thousand Drips: Uncovering Critical Resource Leaks in the Windows Ecosystem

Feng Dong and Jianting Gao, *Huazhong University of Science and Technology;* Yunpeng Tian, *The Hong Kong Polytechnic University;* Weifeng Yuan, *Huazhong University of Science and Technology;* Mu Zhang, *University of Utah;* Zesen Ye and Jietao Yang, *Beijing CyberKunlun Technology Co., Ltd;* Zhiniang Peng, *Huazhong University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity26/presentation/dong-feng

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Death by a Thousand Drips: Uncovering Critical Resource Leaks in the Windows Ecosystem

Feng Dong[1], *Jianting Gao*[1], *Yunpeng Tian*[2], *Weifeng Yuan*[1], *Mu Zhang*[3]
Zesen Ye[4], *Jietao Yang*[4], *Zhiniang Peng*[1]*
[1]*Huazhong University of Science and Technology*† [2]*The Hong Kong Polytechnic University*
[3]*University of Utah* [4]*Beijing CyberKunlun Technology Co., Ltd*
{dongfeng, laudedd}@hust.edu.cn, yun-peng.tian@connect.polyu.hk
{cougarsecu, whhhitc, jiushigujiu}@gmail.com, muzhang@cs.utah.edu, azure@cyberkl.com

Abstract

Windows remote services are a critical attack surface due to their privileged execution and widespread deployment. Under the shared service-host model (`svchost.exe`), a single resource-exhaustion bug can trigger fate-sharing failures across co-resident services. Yet most vulnerability discovery remains crash-centric and is thus blind to silent, long-horizon resource leaks (e.g., memory/handle exhaustion) that do not cause immediate exceptions, while the closed-source Windows ecosystem limits the applicability of source-level sanitizers. We present **ReDoSpector**, a fuzzing framework for discovering resource-oriented Denial-of-Service (DoS) vulnerabilities in opaque Windows binaries. **ReDoSpector** combines (1) lightweight in-process hook-based telemetry by intercepting native `ntdll` heap routines, (2) an amplification-based oracle that identifies persistent leaks via repeated execution and growth-trend analysis, and (3) a hybrid seed-generation pipeline that leverages static binary analysis and Large Language Models (LLMs) to reach deep protocol states. Evaluated on the latest builds of Windows 11 (24H2) and Windows Server Preview, **ReDoSpector** uncovers **19** vulnerabilities in core services (including MSMQ, Remote Desktop Gateway, and CDP), leading to **12** assigned CVEs. Microsoft confirmed seven pre-authentication DoS issues in CDP and performed architectural adjustments, demonstrating that **ReDoSpector** can expose resource-mismanagement flaws that evade conventional fuzzing.

1 Introduction

Windows remote services are widely exposed and therefore attractive targets for external attacks. Memory leak vulnerabilities, despite being a seemingly well-understood class of bugs, continue to plague these critical services. Recent vulnerabilities, including SMBleed [17], LSASS memory leaks [28], and Print Spooler memory leaks [25], demonstrate that memory leaks can still lead to severe security consequences, such as denial-of-service attacks, information disclosure, and security control failures resulting from resource exhaustion.

Many prior efforts have focused on detecting memory leak vulnerabilities; however, they all have notable limitations. Runtime tools such as Valgrind Memcheck [34], LeakSanitizer [8], and Dr. Memory [9] identify memory leaks by detecting memory objects that are unreachable from the application. While effective at capturing leak symptoms, this approach does not pinpoint the root-cause vulnerabilities in leaking programs.

Static leak detectors [10, 19, 46, 47] aim to identify unbalanced memory allocations and deallocations, but often suffer from inherent precision limitations. These limitations are further exacerbated when analyzing a class of stealthy memory-leak vulnerabilities that we term ***release bypass vulnerabilities***. Unlike conventional memory leaks, in which deallocation operations are entirely absent or unreachable, release bypass vulnerabilities arise when valid deallocation logic exists but can be bypassed by attackers through carefully crafted malicious inputs to a remote service. Prior work [14, 15, 22, 39, 46, 50] has identified a subset of *release bypass vulnerabilities* where missing deallocation in error-handling code leads to memory leaks. However, a more general solution that goes beyond narrowly focusing on error-handling code remains underexplored.

Fuzzing-based techniques can fundamentally address these limitations by precisely exercising vulnerable code paths and triggering leaks in a what-you-see-is-what-you-execute manner, thereby enabling accurate localization of the underlying vulnerabilities. However, the crash-based bug oracles commonly used by existing fuzzers, such as Peach Fuzzer [2], Sulley [1], and Boofuzz [30], are ineffective for detecting memory leaks that do not immediately lead to crashes.

Prior work [45] has explored the use of fuzzing to detect memory-leak vulnerabilities, but models them as cases of

*Corresponding author
†The full name of the authors' affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.

uncontrolled memory allocation, where allocation sizes can be adversarially inflated. While such behavior can indeed trigger resource exhaustion, it is comparatively easier to detect. In contrast, typical memory leaks manifest in an accumulative and gradual manner, making them significantly more subtle and difficult to identify.

To make such long-horizon failures detectable within a fuzzing campaign, we treat resource consumption as a first-class signal and explicitly *amplify* weak per-request effects. Concretely, we repeatedly execute candidate inputs and analyze growth trends using fine-grained heap telemetry and coarse-grained process metrics, then correlate persistent resource imbalances with stable call-site fingerprints to distinguish exploitable accumulation from benign caching behavior and allocator noise.

Challenges. Nevertheless, applying such detection oracles to fuzzing real-world Windows services is non-trivial. **(1)** The closed-source nature of many Windows services severely limits its visibility into internal operations, rendering source-based sanitizers inapplicable. **(2)** Even when fuzzing triggers abnormal resource consumption, determining whether it reflects a genuine vulnerability is difficult: resource usage naturally fluctuates under normal workloads, and distinguishing benign spikes from exploitable accumulation requires temporal analysis across multiple executions. **(3)** Windows remote services often implement complex, proprietary network protocols with deep state machines and intricate data structures. These protocols typically demand specialized input generation to exercise rare code paths, making it challenging for generic fuzzers to achieve sufficient coverage.

Consequently, there is a clear need for novel, custom techniques that can *(a)* operate on closed-source Windows services, *(b)* treat cumulative resource growth as a first-class signal, and *(c)* automatically distinguish exploitable trends from benign fluctuations.

Overview of REDOSPECTOR. To address these challenges, this paper presents **REDOSPECTOR**, a framework for discovering memory-leak vulnerabilities in closed-source Windows remote services. First, we implement a sanitizer as a dynamic-link library (DLL) that is injected into target service processes to hook low-level heap allocation functions. This design enables fine-grained, runtime monitoring of memory behavior without requiring source code access or binary rewriting. Second, we introduce a novel amplification-based evaluation strategy that repeatedly exercises the same input to magnify subtle resource-exhaustion effects, enabling the detection of temporal leaks that do not immediately trigger crashes or obvious hangs. Third, we analyze per-input resource-consumption patterns and apply automated false-positive filtering based on allocation–deallocation matching, call-stack classification, and proportionality checks, allowing **REDOSPECTOR** to distinguish genuine accumulation from benign caching behavior and allocator noise. Finally, to improve fuzzing coverage for opaque protocols, we integrate automated binary analysis with large language models (LLMs) to generate protocol-aware mutation seeds, enabling deeper exploration of complex service logic and proprietary handlers.

This paper makes the following main **contributions**:

- We develop **REDOSPECTOR**, which, to the best of our knowledge, is the **first** fuzzing-based framework for detecting memory leaks in closed-source Windows remote services.
- We introduce an amplification-based, resource-oriented fuzzing oracle that elevates cumulative resource growth to a first-class signal, enabling **REDOSPECTOR** to expose long-lived memory leaks and resource-retention vulnerabilities that crash-centric fuzzers systematically miss.
- We apply **REDOSPECTOR** to widely deployed Windows remote services and uncover **19 zero-day vulnerabilities** in core system components, including Microsoft Message Queuing (MSMQ), Remote Desktop Gateway, and the Connected Devices Platform (CDP), resulting in **12 assigned CVEs**. The remaining **7 zero-day vulnerabilities were confirmed by Microsoft**, which classified them as pre-authentication denial-of-service issues in CDP and implemented corresponding architectural mitigations.

2 Background

In this section, we provide the system context that motivates our threat model and oracle design. We first describe how Windows' shared-service hosting amplifies resource-exhaustion failures, and then review user-mode heap internals and why existing leak-detection tooling is difficult to deploy for closed-source services at fuzzing scale.

2.1 Memory Leak Vulnerabilities in the Windows Shared Service Model

In the Windows ecosystem, memory leak vulnerabilities present unique challenges within the context of the *shared service host* model. A key architectural feature is the consolidation of multiple logically independent services into a single `svchost.exe` instance to reduce memory footprint and process-management overhead. This design choice was historically practical because process creation and per-process working sets were comparatively expensive on resource-constrained systems.

However, shared hosting introduces strong *fate sharing*: services within the same host process typically share the same heap, virtual address space, and commit limits. Consequently, a memory leak in an apparently non-critical service can degrade or disable all co-hosted services by exhausting shared resources, fragmenting the heap, or consuming the process-wide virtual memory quota. In practice, such leaks are frequently harder to diagnose than discrete process failures, as

they manifest as a gradual increase in resource consumption rather than an immediate termination.

This aggregation effect complicates the isolation of memory management errors. A persistent leak in one component can induce allocation failures in neighboring services, effectively turning a localized implementation flaw into broad service instability. Unlike a standalone process where a leak is self-contained, a leak in a shared host affects the collective stability of the group. Furthermore, identifying the specific culprit is often difficult, as standard monitoring tools may attribute the memory growth to the host process generic `svchost.exe` rather than the specific sub-service responsible for the allocation.

Microsoft has evolved service grouping policies to mitigate these risks. For example, starting with Windows 10 version 1903 on sufficiently memory-rich client systems, many services are no longer grouped into shared host processes and instead run in separate processes, improving isolation and debuggability at the cost of additional memory overhead. This shift underscores that service-host consolidation is a performance–isolation trade-off, and that memory leak bugs in shared hosts remain a critical stability concern in configurations where consolidation is still used (e.g., legacy systems, constrained environments, or deliberately grouped services).

2.2 Windows User-Mode Heap Internals

In user mode, Windows memory management is primarily implemented in `ntdll.dll`. Low-level allocation routines such as `RtlAllocateHeap` and `RtlFreeHeap` form the core interface for dynamic heap management and are used by both the NT heap and the segment heap.

Heap architecture. Modern Windows employs a dual heap architecture: the legacy *NT heap* and the newer *segment heap*. As described in prior work [3–5, 24, 35, 37, 38, 42, 43, 48], the NT heap combines a *front-end allocator* (the Low Fragmentation Heap, LFH) for small, fixed-size allocations with a *back-end allocator* for larger and more varied allocation sizes. Figure 1 illustrates the internal organization of the Windows NT Heap.

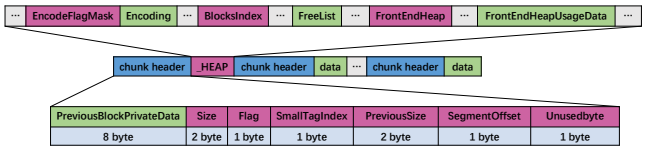


Figure 1: Internal structures of the Windows NT Heap.

Metadata integrity and hardening. The security of the NT heap heavily relies on the integrity of heap metadata, especially within the `_HEAP_ENTRY` structure. Critical metadata fields such as `Size` and `Flags` are XOR-encoded with a process-specific cookie stored in the `_HEAP` structure. In addition, the `SmallTagIndex` stores a 1-byte checksum to impede unauthorized header tampering.

Resource mismanagement. Despite these mitigations, resource mismanagement remains a serious threat. Memory leaks do not directly violate metadata integrity, but they pollute the heap's free lists and influence structures such as the `FrontEndHeapUsageData` bitmap, which controls LFH activation. Observing these effects at the level of `RtlAllocateHeap`/`RtlFreeHeap` provides a unified vantage point that applies to both NT and segment heap instances, abstracting away implementation differences among heap types.

2.3 Windows Heap Sanitizer Limitations

While leveraging established leak detectors as the oracle appears intuitive, existing solutions are fundamentally ill-suited for fuzzing closed-source Windows services at scale. First, dynamic binary instrumentation (DBI) frameworks face ongoing compatibility challenges within the modern Windows ecosystem. For instance, Valgrind lacks native Windows integration [27], and Dr. Memory, historically a prominent tool for Windows binaries [6], can conflict with the enhanced security mechanisms of Windows 11, particularly when analyzing system services. This interference often impedes the normal execution of testing procedures. Second, compiler-based instrumentation techniques, such as AddressSanitizer (ASan) and LeakSanitizer (LSan) [33], are structurally inapplicable to black-box targets. These solutions necessitate access to source code and integration into the build chain to inject tracking logic, prerequisites that are inherently unsatisfied when auditing proprietary binaries where source code is inaccessible and debug symbols are often stripped. Third, while system-wide telemetry mechanisms like Event Tracing for Windows (ETW) [26] provide granular allocation data, they are designed primarily for real-time audit logging and also support human-centric diagnosis. Correlating aggregated resource trends from high-volume event streams to specific causal inputs creates a significant attribution gap. Consequently, utilizing these tools necessitates prohibitive manual effort to filter system noise and verify individual leaks, making the approach unscalable for automated vulnerability discovery.

Beyond the limitations above, Windows hardening features such as Control Flow Guard (CFG), alongside the lack of comprehensive symbols in production binaries, make heavyweight debugging aids like PageHeap impractical for large-scale fuzzing. While dynamic binary instrumentation (DBI) frameworks like Pin [23] can observe execution without relying on symbols, they still introduce substantial runtime overhead that hinders fuzzing efficiency. These cumulative constraints motivate the adoption of more lightweight monitoring approaches, such as TinyInst [11], which can track heap activity within `ntdll.dll` without triggering security mitigations or incurring prohibitive performance penalties.

2.4 Threat Model and Scope

We consider remote attackers who can send network requests to Windows services (in pre-authentication or low-privilege contexts) with the intent of compromising service availability. Our primary focus is on detecting ***release bypass vulnerabilities***. Unlike classic memory leaks caused by completely absent release logic, or uncontrolled memory allocations that exhaust resources via a single oversized request, release bypass vulnerabilities are subtle and path-dependent. In our model, the target service contains valid cleanup logic for standard execution flows. However, the attacker crafts specific payloads to trigger *exceptional code paths*, typically within error-handling routines or complex conditional branches. On these paths, the service successfully allocates memory (e.g., for packet processing or state management) but encounters a parsing error, logic violation, or check failure induced by the malicious input. This triggers an early return or a deviation in control flow that bypasses the intended release instruction, leaving the allocated object orphaned on the heap.

Scope. We focus on *accumulative* resource exhaustion where the leakage per request is small, but repeated interaction leads to significant consumption. Our analysis targets user-mode heap manipulation (monitoring `RtlAllocateHeap` and `RtlFreeHeap`). Consequently, denial-of-service attacks based purely on CPU exhaustion, distributed bandwidth saturation, or kernel-pool leaks that do not reflect in user-mode heap telemetry are outside the scope of this work.

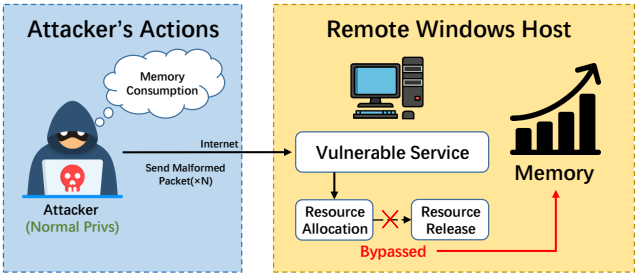


Figure 2: Attack Scenario

Representative attack scenario. A representative scenario involves a remote RPC or HTTP service hosted in `svchost.exe`. The attacker sends a stream of crafted requests, each containing a specific malformed field designed to trigger a sanitization check *after* an initial memory allocation has occurred. Upon processing each request, the service allocates a buffer, detects the malformed data, and enters an error-handling path to reject the request. Crucially, this error path fails to release the previously allocated buffer before returning control. As illustrated in Figure 2, repeated exploitation of this flawed error-handling logic causes unreleased heap allocations to accumulate over time. While the service remains responsive initially, the attacker automates the transmission of these requests, causing a monotonic growth in memory usage. This eventually exhausts the process's memory quota, leading to allocation failures that crash the host process or render all co-hosted services unresponsive.

3 System Design

3.1 Overview

REDOSPECTOR is specifically engineered to systematically discover ***release bypass vulnerabilities*** within closed-source Windows remote services. As highlighted in our threat model, these vulnerabilities manifest not as immediate crashes, but as *accumulative* resource exhaustion caused by specific error-handling paths or core logic that inadvertently skip valid cleanup logic.

REDOSPECTOR addresses the visibility gap in opaque binaries by shifting the detection paradigm from binary crash oracles to *amplified resource telemetry*. Rather than relying on coarse process termination signals, which are ineffective against gradual leaks, our system employs a *temporal amplification* strategy combined with fine-grained in-process monitoring. By repeatedly executing candidate inputs and correlating them with heap allocation events, **REDOSPECTOR** magnifies subtle, per-request resource retention into observable trends, enabling the precise identification of logic flaws where malicious inputs bypass intended deallocation routines.

At a high level, the framework combines three key ideas:

- *Semantic-aware seed generation* for proprietary protocols using static analysis and LLM assistance [51] (subsection 3.2);
- *In-process resource monitoring* via heap-level hooks that capture allocation/free events with call-site context (subsection 3.3);
- *Temporal amplification* and *proportionality-based analysis* to convert small, non-crashing leaks into clearly observable trends (subsection 3.4 and subsection 3.5).

Figure 3 presents an overview of the complete workflow of **REDOSPECTOR**, illustrating how seed generation, resource monitoring, amplified fuzzing, anomaly detection and telemetry analysis interact across the fuzzing pipeline.

The operational workflow of the framework is organized into four sequential phases:

Phase 1: Static Analysis and Seed Generation. To navigate proprietary protocols without source code, we perform structural extraction using reverse engineering frameworks integrated with corresponding MCP tools. The extracted artifacts are normalized and provided to a Large Language Model (LLM) to hypothesize protocol grammars and synthesize high-quality mutation seeds that target complex parsing logic and error-handling hotspots.

Phase 2: Hook-based Resource Monitoring. We inject a monitoring DLL into the target service using Microsoft Detours. This module intercepts native `ntdll` heap management routines (e.g., `RtlAllocateHeap`, `RtlFreeHeap`) and records

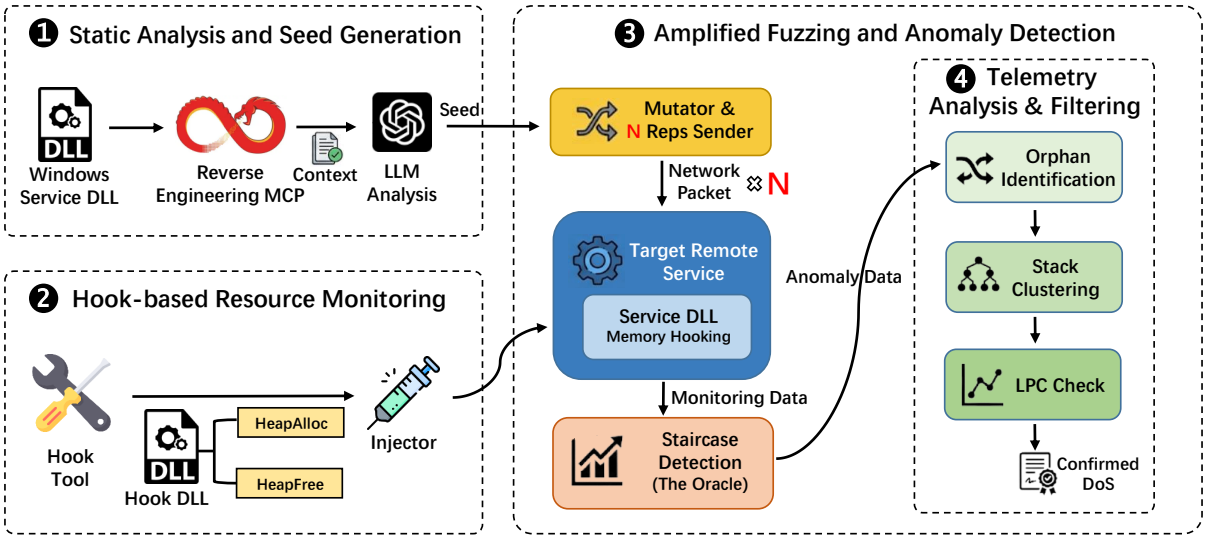


Figure 3: Overview of **REDOSPECTOR**. The controller drives fuzzing and amplification; the in-process monitor records heap-level telemetry inside Windows services; and the analyzer applies proportionality-based checks over amplified traces to identify resource-oriented DoS vulnerabilities.

detailed telemetry, including allocation sizes and call-site fingerprints, into a reentrancy-safe, preallocated buffer.

Phase 3: Amplified Fuzzing and Anomaly Detection. In the execution loop, we employ an amplification-based bug oracle. Each input is repeatedly transmitted to magnify minor resource leaks. The oracle monitors for "staircase" growth patterns in memory and handle counts, flagging suspicious inputs for further triage.

Phase 4: Telemetry Analysis and False-Positive Filtering. Finally, the framework processes the collected telemetry through an analysis pipeline. We perform address-level pairing to identify orphaned allocations and apply proportionality heuristics to isolate genuine leaks from background noise.

In our current implementation, the primary resource signals are user-mode heap allocations (as observed via `RtlAllocateHeap`/`RtlFreeHeap`), process private bytes, and per-process handle counts sampled via the Windows performance APIs. The fuzzing loop itself is mutation-based, operating in a black-box mode where seeds are scheduled based on past anomaly scores. The resource-aware oracle is therefore orthogonal to the underlying input generation strategy.

Across all phases, **REDOSPECTOR** operates in iterative fuzzing rounds. Each round selects a seed from the corpus, mutates it into one or more concrete inputs, executes them against the target while the in-process monitor records telemetry, optionally applies temporal amplification, and finally updates the seed corpus based on the calculated anomaly scores.

3.2 Automated Binary Analysis and Seed Generation

Effective fuzzing of complex, closed-source Windows services critically depends on high-quality, protocol-relevant seeds. We therefore combine traditional reverse engineering, automated static analysis, and LLMs to generate seeds that exercise diverse service behaviors.

We utilize reverse engineering frameworks integrated with corresponding MCP tools to extract structural artifacts from the target binaries, including exported functions, cross-references, string literals, and switch-based dispatch sites. We place special emphasis on functions that (i) read from externally reachable sockets or RPC channels, and (ii) contain complex branching logic or error-handling paths, as inferred from control-flow graph (CFG) features such as nested conditionals and multiple return sites.

To make these artifacts consumable by an LLM, we translate the outputs of the integrated reverse engineering and MCP tools into a compact, language-agnostic schema. For each suspected parser or dispatcher function, we record the function name and signature, the list of basic blocks that read from a candidate input buffer pointer, comparisons against constant values (e.g., magic numbers, command IDs, maximum lengths), and outbound calls that may indicate error handling, logging, or state updates. We also extract string constants (e.g., error messages, command names) that hint at protocol semantics. These records are serialized into a structured textual summary.

The LLM serves two purposes. (1) Infers valid input formats: The resulting seed templates capture field types and value constraints. (2) Function name guided call chain discovery: This recall-oriented approach reduces false negatives

compared to traditional dataflow analysis, and further infers call chains from external entry points to internal functions.

The normalized summaries are provided to an LLM (in our experiments, GPT-4o) via a constrained prompt that describes our goals (e.g., identify candidate length fields, message types, and flag combinations) and enforces a strict output schema. The LLM is asked to infer a small set of canonical message types and to produce a JSON-based packet template for each. Each template contains a list of fields, where each field has a name (possibly inferred from surrounding strings), a byte offset, a length, an optional enumeration of valid values, and mutation hints (e.g., "boundary", "rare flag", or "reserved"). This machine-readable schema allows the rest of the framework to treat LLM-generated templates as declarative specifications rather than free-form text.

The entire seed-generation pipeline is automated and does not require manual reverse engineering beyond specifying the binary under test. The LLM does not interact with the running service directly; its output is treated as *hints* that we subsequently validate and mutate. When the LLM produces inconsistent or partially incorrect templates (for instance, misidentifying a length field), the resulting seeds behave as regular fuzzing inputs: they may fail parsing and be rejected early by the service, but they do not break the framework.

During fuzzing, the seed corpus initially consists of LLM-generated protocol templates and a small number of manually constructed baseline requests (e.g., minimal well-formed packets). As the campaign progresses, the corpus grows to include inputs that previously triggered resource anomalies. The default scheduling policy is a simple round-robin over seed categories, but **ReDoSpector** can bias selection toward seeds that have historically produced higher anomaly scores, steering the search toward program states that already exhibit suspicious behavior.

For each selected seed, the controller generates one or more mutated variants. For structured seeds, we apply a mix of generic byte-level mutations (bit flips, arithmetic on integers, block insertions/deletions) and structure-aware operators that preserve inferred field boundaries. For example, if the LLM has labeled a 16-bit field as a *length* or *count*, mutators preferentially set it to boundary values (e.g., 0, 1, maximum representable value) or to values inconsistent with the following payload size.

3.3 Hook-based Resource Monitoring Module

The core of our monitoring design is an in-process hook DLL that instruments heap operations within target Windows services. We implement this DLL using Microsoft Detours to intercept native heap management routines exported by `ntdll.dll`. Instrumenting at this layer yields a unified vantage point across different user-mode allocator implementations.

The DLL installs hooks on `RtlAllocateHeap` and `RtlFreeHeap`. As shown in Figure 4, this provides a common observation point for both the legacy *NT Heap* and the modern *Segment Heap*. We choose to use `RtlAllocateHeap` rather than `VirtualAlloc` and `NtAllocateVirtualMemory` because it provides finer-grained, application-level object allocation semantics, as opposed to the page-level allocation provided by the latter. This helps avoid false positives caused by delayed frees in underlying caching systems. For each allocation or free operation, the hook records an observation tuple

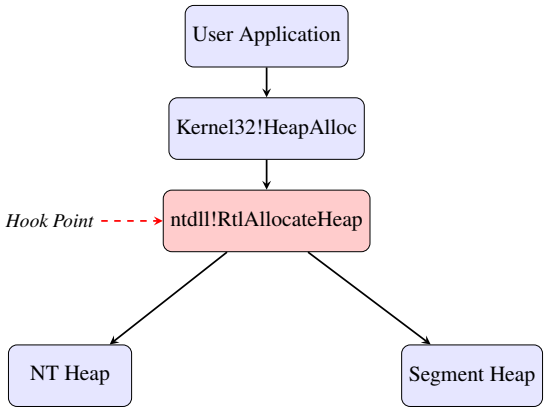


Figure 4: Hierarchical structure of Windows heap management and our instrumentation point.

$$\langle Addr, Size, TID, StackHash \rangle,$$

where `Addr` is the returned (or freed) address, `Size` is the requested allocation size, `TID` is the thread identifier, and `StackHash` is a compact fingerprint of the user-mode call stack.

The `StackHash` field is computed as a 64-bit hash over the sequence of return addresses in the user-mode call stack, collected up to a fixed depth (8 frames by default). To improve robustness under ASLR, we normalize each frame by subtracting the base address of the corresponding module, which we dynamically resolve at runtime. As a result, the same code path yields a stable hash across runs and processes, even when modules are relocated.

To ensure stability, the monitoring logic avoids invoking the instrumented heap APIs. We preallocate a large memory buffer during DLL initialization; hook handlers write telemetry exclusively into this region and never perform heap allocations or other operations that could trigger recursive re-entry. To support highly multithreaded environments (e.g., `svchost.exe`), we use atomic operations to reserve indices in a fixed-size ring buffer, thereby guaranteeing race-free logging without external locks.

For services hosted inside shared `svchost.exe` instances, we register a launcher service that restarts the target group under a wrapper process. The wrapper initializes Microsoft Detours, injects our monitoring DLL, and then transfers control to the original service entry point. For standalone services

(e.g., `mqsvc.exe`), we start the binary under the wrapper directly. In both cases, the injection cost is paid once per fuzzing session and does not affect per-request latency.

Hooking low-level APIs in `ntdll.dll` raises concerns about compatibility with Control Flow Guard (CFG) and other hardening features. Our implementation preserves the original function prologues and epilogues and redirects control through Detours' officially supported trampolines. We avoid modifying indirect call targets or kernel-mode structures, thereby staying within the bounds of user-mode supported behavior. Across multiple Windows 10/11 and Server builds, we did not observe any CFG violations or PatchGuard-related crashes attributable to our hooks.

Recording a full stack trace for every allocation can be expensive on allocation-heavy workloads. To mitigate this overhead, we limit stack depth, cache module base addresses to normalize return pointers, and optionally enable sampling: users can configure the monitor to only log allocations above a configurable size or to sample one out of every k events per call site.

During each fuzzing round, the protocol-specific harness sends mutated inputs to the target via the appropriate transport (e.g., raw TCP, HTTPS, or RPC over TCP), performs any required handshakes or authentication steps, and relies on the in-process monitor to record heap events and process metrics for subsequent analysis.

3.4 Amplification-based Bug Oracle

Detecting "silent" leaks requires overcoming the noise inherent in complex system services. Background activity, one-time initialization, and benign caching can all produce transient increases in resource usage. To distinguish genuine per-request leaks from such noise, we augment traditional oracles with a *temporal amplification* strategy, as outlined in algorithm 1.

Algorithm 1: Amplification-based Bug Oracle

Input : Fuzz Input I, Protocol Type $\mathcal{P}$
Config : n repetitions, Threshold k, Baseline μ, σ

$Trace \leftarrow \emptyset$;
if ***isStateful***$(\mathcal{P})$ **then**
  $S \leftarrow$ `CreateSession ()` ; `// Reuse session mode`
for $i \leftarrow 1$ **to** n **do**
  if ***isStateless***$(\mathcal{P})$ **then**
    $C \leftarrow$ `OpenConn ()`;
    `SendReq` (C, I);
    `CloseConn` (C) ; `// Isolate connection state`
  else
    `SendReq` (S, I) ; `// Stress state machine`
  $m_i \leftarrow$ `SampleMetrics ()` ; `// Heap, Handles, PvtBytes`
  $Trace$.append(m_i);
  Timeout **or** Crash **return**
  ReportBug("Availability/DoS");
if *S is active* **then**
  `CloseConn` (S) ; `// Trigger cleanup checks`
`// Oracle: Trend detection & validation`
$\beta \leftarrow$ `CalcSlope`$(Trace)$;
$is_staircase \leftarrow$ `IsStaircase`$(Trace)$;
$allocs \leftarrow$ `AnalyzeOrphans`$(Trace)$;
if $\beta > (\mu + k \cdot \sigma)$ **then**
  if *is_staircase* ***and*** *allocs.isConsistent()* **then**
    return ReportBug("Resource Leak", *allocs*);
return `PASS`;

Depending on the protocol semantics, amplification can be performed over independent connections (stateless mode) or within a single long-lived session (stateful mode). For HTTP-based services and simple RPC interfaces, we open a fresh connection for each repetition to avoid conflating per-connection state with per-request leaks. For protocols that naturally maintain state (e.g., message queues), we support both modes: by default, we reuse a single session to stress long-lived state machines, but we also perform a smaller number of repetitions across fresh sessions to distinguish connection-scoped from process-scoped leaks.

For each candidate fuzz input, the system sends a controlled sequence of n repeated requests (typically $n \in [50, 200]$ for screening and up to 1,000 for confirmation). During this window, we record (i) per-request heap allocation and free events and (ii) coarse-grained process metrics (private bytes and handle counts) sampled at fixed intervals. To account for normal variability, we first establish a baseline by performing m repetitions of a known-benign input and estimate the mean μ and standard deviation σ of the relevant metrics.

An input is considered anomalous if the slope of its resource usage over the n repetitions exceeds $\mu + k\sigma$ for a configurable constant k (we use $k = 3$ by default), and if this trend is corroborated by orphaned allocations at a consistent call site (subsection 3.5). Importantly, this cumulative repetition amplifies the resource signal to make subtle leaks detectable; otherwise, the per-run leakage is negligible. This trend-based detection facilitates the attribution of persistent allocations to specific code paths identified via stack fingerprints.

The oracle specifically identifies "staircase" growth patterns, where resource usage increases approximately linearly with each packet but fails to decline after the connection is terminated or the session is reset. Staircase-shaped curves in private bytes or handle counts indicate that each request leaves behind a small residue of unreclaimed state. By combining global process metrics with per-call-site orphaned allocation counts, we can distinguish such behavior from benign bursty allocation patterns that eventually return to baseline.

Although our primary goal is to detect non-crashing resource leaks, amplification sometimes triggers crashes or hangs as a side effect (e.g., when a leak eventually causes an out-of-memory condition). We treat such events as first-class outcomes: the controller detects loss of responsiveness

Algorithm 2: False-Positive Filtering via LPC Analysis

Input : Raw telemetry trace $\mathcal{T}$ (Sequence of Alloc/Free events)
Config : Amplification factor n, Tolerance ε (e.g., 0.1)
Output : Set of suspicious call stacks $\mathcal{S}_{suspicious}$

```
/* Phase 1: Set-Difference Analysis (Isolate Orphans)      */
M_active ← ∅ ;                  /* Map: Address → CallStack */
for event ∈ T do
    if event.type = ALLOC then
        M_active[event.addr] ← event.stack;
    else if event.type = FREE then
        if event.addr ∈ M_active then
            Remove (event.addr from M_active);
/* Phase 2: Orphan Aggregation by Fingerprint              */
Counts ← ∅ ;                    /* Map: CallStack → Integer */
for stack ∈ Values(M_active) do
    Counts[stack] ← Counts[stack] + 1;
/* Phase 3: Linear Proportionality Check (LPC)             */
S_suspicious ← ∅;
for (σ, k) ∈ Counts do
    /* k is |OrphanedAllocations_σ|                        */
    C ← Round (k/n);
    /* Filter 1: Ignore noise (C < 1 implies sub-linear
       growth)                                             */
    if C ≥ 1 then
        /* Filter 2: Check tolerance band |k − C·n| ≤ ε·n   */
        δ ← Abs (k − C·n);
        if δ ≤ ε·n then
            S_suspicious ← S_suspicious ∪ {σ};
return S_suspicious;
```

via timeouts and service health checks, records the last few inputs and telemetry snapshots, and restarts the service under the same instrumented environment. These cases are reported as availability-impacting bugs regardless of whether they manifest as slow DoS or hard crashes.

3.5 Input Analysis and False-Positive Filtering

The final phase isolates high-confidence DoS-inducing inputs by processing the raw telemetry dataset through an automated triage pipeline. The analyzer classifies unbalanced memory allocation calls based on their call stacks and subsequently counts the occurrences for each category. Categories with higher counts are deemed more likely to be the root causes of memory leaks.

The analyzer first performs a set-difference analysis on the recorded heap addresses. For each allocation tuple, we check whether a corresponding free record exists for the same `Addr`. Any allocation record with a matching free record is marked as reclaimed; the remaining records are considered *orphaned* allocations. This isolates outstanding allocations that remain after the amplification sequence and helps separate long-lived objects from short-lived, fully reclaimed ones.

We then classify orphaned allocations by their call-stack fingerprints and apply a *Linear Proportionality Check (LPC)*. Given an amplification factor n, the leak index L_σ for a specific call stack σ is defined as:

$$L_\sigma = \frac{|\text{OrphanedAllocations}_\sigma|}{n}, \tag{1}$$

where $|\text{OrphanedAllocations}_\sigma|$ denotes the cardinality of the set of allocations originating from stack σ that were not freed during the amplification window. Intuitively, when each request leaks a fixed number of objects, L_σ should be close to a small integer.

In practice, we implement the proportionality condition with a tolerance band. Let C be the hypothesized number of leaked objects per request (typically a small integer). We mark a call stack σ as suspicious if

$$\big|\,|\text{OrphanedAllocations}_\sigma| - C \cdot n\,\big| \le \varepsilon \cdot n, \tag{2}$$

where ε is a small tolerance parameter (we use $\varepsilon = 0.1$). This requires the number of outstanding allocations to grow approximately linearly with n and to be close to an integer multiple of the amplification count. Stacks that fail this check (e.g., one-off initialization leaks or sporadic background allocations) are deprioritized. The complete procedure for identifying orphaned allocations and applying the LPC filter is summarized in algorithm 2.

Raw anomaly detection often produces multiple inputs that trigger the same underlying bug. To reduce triage effort, we cluster suspicious inputs based on their dominant leaking call stacks and resource-impact signatures. Two inputs are placed in the same cluster if they share the same top-k stack hashes (for a small k, e.g., 3) and their per-request leak sizes fall within a tolerance band. For each cluster, we select a representative input that minimizes length and complexity, akin to testcase minimization in traditional fuzzing.

Not all leaks have equal impact. We prioritize candidates by a composite score that combines (i) the total leaked bytes per amplification window, (ii) the number of leaked objects, and (iii) the fraction of the process's total allocations that remain unreclaimed. Large, rapidly growing leaks are surfaced first, while tiny leaks that would require millions of requests to become observable are deprioritized. When multiple stack fingerprints in the same run satisfy the LPC, we retain all of them as candidates but rank them by their contribution to the overall allocation–free imbalance and the absolute size of leaked objects.

Telemetry logs can grow quickly for long amplification windows. To keep analysis tractable, we process logs in a streaming fashion: instead of loading all records into memory, we maintain per-stack counters and address sets as we scan the buffer once. For large-scale campaigns across multiple

services, the analyzer can run on a separate machine and consume telemetry asynchronously over a lightweight RPC channel, allowing fuzzing and analysis to proceed in parallel.

Confirmed candidates that pass LPC and prioritization thresholds are subsequently re-executed in an isolated environment to ensure that the observed increases are monotonic and non-transient. These runs also serve to generate minimal PoC testcases and to measure the number of repetitions required to produce user-visible denial-of-service symptoms. Inputs whose anomaly scores exceed a configurable threshold are promoted to the "candidate" set and stored with their associated call-stack fingerprints and resource impact summaries, and can be fed back into the seed corpus to guide subsequent mutations toward vulnerable code paths.

3.6 Implementation Details

We implemented **REDOSPECTOR** entirely in C/C++, consisting of approximately 12,500 lines of code. The system comprises an in-process monitoring DLL, a Detours-based launcher, and a central controller that manages the fuzzing loop, amplification logic, and analysis pipeline. All components are compiled as native 64-bit binaries.

The system is logically divided into three components: (i) the *monitor*, which resides inside the target service and logs heap events; (ii) the *controller*, which manages fuzzing, amplification, and service lifecycle; and (iii) the *analyzer*, which processes telemetry logs and produces ranked bug reports. Components communicate via a lightweight RPC channel over named pipes, with telemetry data transferred as binary blobs.

Users configure targets through a simple YAML specification that lists the service name, binary path, startup arguments, network endpoints, and any required authentication or session-setup steps. For common Windows services (e.g., CDP, MSMQ), we provide pre-defined harness templates that encapsulate protocol handshakes and stateful operations. This separation allows security analysts to adapt the framework to new services without modifying core code.

REDOSPECTOR supports two deployment modes. In *single-target* mode, a researcher focuses on one service at a time, with a dedicated virtual machine snapshot per target. In *batch* mode, the controller cycles through a list of services, applying a fixed fuzzing budget to each before moving on. In both modes, we rely on VM snapshots to restore the system to a clean state between campaigns, ensuring deterministic starting conditions for repeated experiments.

4 Evaluation

We evaluate the effectiveness and efficiency of **REDOSPECTOR** in discovering release bypass vulnerabilities in the Windows ecosystem. Our evaluation is guided by three questions:

RQ1. Can **REDOSPECTOR** uncover previously unknown resource leaks and DoS vulnerabilities in closed-source Windows services?
RQ2. How effective is the amplification-based oracle compared to a standard crash-centric oracle?
RQ3. How do individual components, specifically LLM-assisted seeding and false-positive filtering, contribute to the system's coverage and precision?

4.1 Experimental Setup

Target Environment. We conducted our experiments on two platforms: Windows 11 24H2 (Build 26100.1) and Windows Server Preview (Build 26360). All targets ran on VMware Workstation 17 Pro virtual machines configured with 4 vCPUs and 8 GB of RAM.
Fuzzing Configuration. We used the automated seed generation module described in subsection 3.2, leveraging the integrated reverse engineering and MCP tools for structural analysis and GPT-4o for semantic interpretation of protocol fields. To balance crash detection with false alarms, we empirically set the threshold described in subsection 3.4 to 3. The amplification factor n in our bug oracle was set to 1,000, as further increases provide diminishing returns. For each target, we ran both **REDOSPECTOR** and a baseline configuration (subsection 4.3) for 48 wall-clock hours using identical seed corpora and mutation strategies.

4.2 RQ1: Overall Vulnerability Discovery

To ensure the practical relevance and representativeness of our evaluation, we strategically selected a dataset of five Windows services based on three core considerations. First, we focused on services that are enabled by default or commonly active (e.g., CDP), as they constitute highly practical and widespread attack surfaces. Second, we prioritized services that remain underexplored in the literature, deliberately excluding heavily reverse-engineered targets such as SMB. Third, our selection encompasses diverse Windows operational scenarios, including OS deployment (WDS), data center storage (StorageManagement), cloud-based information exchange (MessageQueue), and human-machine/cloud interfaces (RDG).

Evaluating against this diverse dataset, **REDOSPECTOR** uncovered a total of 19 previously unknown release-bypass vulnerabilities. Table 1 summarizes these findings, detailing the disclosure status, affected services, vulnerable modules, and specific vulnerable functions. Across the zero-day vulnerabilities we identified, 11 are not related to error-handling.
Security Impact. Our findings have contributed to the security posture of Windows, resulting in 12 CVE IDs assigned to vulnerabilities discovered by **REDOSPECTOR**. These cases, along with seven Denial-of-Service (DoS) vulnerabilities in the Connected Devices Platform (CDP) service, were submitted to the Microsoft Security Response Center (MSRC) be-

Table 1: Summary of vulnerabilities discovered by **REDOSPECTOR**.

No.	Status / CVE	Service / Component	Vulnerable Module	Vulnerable Function
1	CVE-2024-38015	Remote Desktop Gateway	aaedge.dll	CAXXXon
2	CVE-2025-29957	Windows Deployment Services	wdstftp.dll	CCXXXst
3	CVE-2025-21174	Standards-Based Storage Management Service	concrete.dll	CIXXXCE
4	CVE-2025-27485	Standards-Based Storage Management Service	concrete.dll	coXXXer
5	CVE-2025-27486	Standards-Based Storage Management Service	concrete.dll	coXXXgs
6	CVE-2025-33068	Standards-Based Storage Management Service	concrete.dll	HTXXXte
7	CVE-2024-49096	Microsoft Message Queuing	mqise.dll	HtXXXoc
8	CVE-2025-21230	Microsoft Message Queuing	mqqm.dll	CSXXXed
9	CVE-2025-21251	Microsoft Message Queuing	mqqm.dll	CSXXXted
10	CVE-2025-21270	Microsoft Message Queuing	mqqm.dll	GeXXXty
11	CVE-2025-21289	Microsoft Message Queuing	mqqm.dll	SaXXXue
12	CVE-2025-21290	Microsoft Message Queuing	mqqm.dll	SaXXXId
13	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdp.dll	MeXXXta
14	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdp.dll	MeXXXata
15	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdp.dll	cdXXXta
16	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdp.dll	cdXXXata
17	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdpsvc.dll	maXXXed
18	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdpsvc.dll	maXXXred
19	Confirmed (Arch. Change)†	Connected Devices Platform Service	cdpsvc.dll	stXXXed

†**Confirmed (Architectural Change):** These vulnerabilities were confirmed by Microsoft and addressed via fundamental architectural hardening in the Connected Devices Platform. Per MSRC policy, individual CVEs are not assigned for structural changes that mitigate multiple flaw instances simultaneously.

tween December 2024 and January 2025. Microsoft validated the reports and released fixes in staged updates from January to June 2025. Notably, rather than assigning individual CVE identifiers to the seven DoS vulnerabilities, Microsoft chose to address them by redesigning the underlying communication stack, addressing the architectural nature of the underlying issues.

Exploiting Vulnerabilities. To assess practical exploitability beyond qualitative impact, we quantify how quickly an attacker can drive vulnerable services into loss of availability using only repeated remote requests. Table 2 reports, for representative DoS cases, the approximate number of requests required to cause either a process-level termination or a persistent service outage, together with the corresponding wall-clock time under a fixed sending rate in our testbed. We emphasize that these thresholds are environment-dependent (e.g., hardware, service configuration, and whether the target is co-hosted in a shared `svchost.exe`); nevertheless, the results consistently show that small per-request resource retention can accumulate rapidly, making the attacks feasible even without high traffic volume. In Table 2, "service outage" denotes sustained loss of correct service behavior (e.g., persistent unresponsiveness, repeated failures, or refusal of new connections) without necessarily crashing the hosting process; recovery may require restarting the service or, in some cases, the host process/system.

4.3 RQ2: Detection Effectiveness

To quantify the benefit of our resource-oriented oracle, we compared **REDOSPECTOR** with Boofuzz, a widely used open-source network protocol fuzzer, which represents traditional approaches that are not originally designed to capture incremental memory leaks. To ensure a fair comparison, the Boofuzz baseline was configured to use a standard crash-centric oracle (detecting process termination and connection failures) but shared the same seed corpus, mutation strategy, and target services as **REDOSPECTOR**. Specifically, to maintain identical mutation behaviors, we applied byte flipping, insertion, and deletion for binary inputs, and value range-guided permutation for text inputs across both fuzzers. Additionally, we aligned the structural handling of protocol packets between both tools. For target packets containing length fields, we configured the Boofuzz harness to utilize its native Size element to dynamically calculate the length, matching the structural handling of length fields in **REDOSPECTOR**. While our evaluation framework supports both fuzzable and structural (non-fuzzable) configurations for mutating these fields, we employed the structural configuration for both fuzzers in our comparative experiments to ensure consistency. The primary remaining difference was the lack of our in-process heap monitor in Boofuzz.

Table 3 summarizes the results extended across all five selected services. Under the same 48-hour time budget, **REDOSPECTOR** successfully triggered thousands of cumulative crashes and identified a total of 19 unique DoS vulnerabili-

Table 2: Requests/time to loss of availability for representative DoS vulnerabilities uncovered by **REDOSPECTOR**. Results are measured in our testbed under a fixed request rate. "Service outage" denotes persistent loss of correct service behavior without necessarily terminating the hosting process.

Service / component	Hosting process	Requests to failure	Time to failure	Failure mode
CDP Service	`svchost.exe`	4.1M	6 min	service outage / process crash
Remote Desktop Gateway	`svchost.exe`	204M	9 min	service outage / process crash
Windows Deployment Services	`svchost.exe`	8.3M	7 min	service outage / process crash
Standards-Based Storage Management Service	`svchost.exe`	5.7M	9 min	service outage / process crash
Microsoft Message Queuing	`mqsvc.exe`	14M	8 min	service outage / process crash

ties (7 in CDP, 1 in RDG, 1 in WDS, 4 in MSStrgSvc, and 6 in MessageQueue). In stark contrast, Boofuzz failed to detect a single memory leak bug. In these cases, the target services remained responsive to simple heartbeat checks and did not crash within Boofuzz's time window, even though they continuously leaked substantial amounts of memory or handles. This highlights the fundamental limitation of traditional crash-centric fuzzers when dealing with cumulative resource exhaustion attacks.

Table 3: Comparison of **REDOSPECTOR** and Boofuzz across five selected services.

Target	REDOSPECTOR		Boofuzz	
	Crashes	Unique Bugs	Crashes	Unique Bugs
CDP Svc (cdp.dll, cdpsvc.dll)	1423	7 DoS	0	0
RDG Svc (aaedge.dll)	672	1 DoS	0	0
WDS Svc (wdstftp.dll)	621	1 DoS	0	0
MSStrgSvc (concrete.dll)	855	4 DoS	0	0
MessageQueue (mqqm.dll, mqise.dll)	1389	6 DoS	0	0

For example, in CVE-2025-33068, each malformed request leaked exactly `0x820` bytes. From the perspective of Boofuzz's crash-only oracle, each execution appeared normal. In contrast, **REDOSPECTOR** identified this leak within 50 amplified iterations by detecting a stable, linear increase in the process's private bytes and a growing imbalance between allocations and frees at a specific call site.

4.4 RQ3: Ablation Study

To understand the contribution of **REDOSPECTOR**'s key components, we performed ablation studies focusing on the seed generation pipeline and the false-positive filtering mechanism.

Impact of LLM-assisted Seed Generation. We evaluate the efficacy of our semantic-aware seed generation by analyzing its ability to satisfy the strict protocol validation logic within `CDPSvc`, a structural barrier that effectively blocks format-agnostic baselines from interacting with the service's core logic.

The entry point for the CDP protocol, handled by `cdp::MessageAccumulator::AccumulateMessageData` in `cdp.dll`, enforces rigid structural constraints on incoming packets. To proceed to the stateful processing logic, a request must satisfy three conditions derived from the MS-CDP specification:

(i) **Signature Match:** The first two bytes must equal `0x3030`.
(ii) **Minimum Length:** The declared message length must be at least `0x2A` bytes.
(iii) **Length Consistency:** The declared length must match the actual number of bytes received.

Random or format-agnostic mutations consistently fail these checks, resulting in early termination. In contrast, by analyzing the normalized structural artifacts from `cdp.dll`, the LLM correctly inferred these invariants and synthesized valid seed templates. As illustrated in Figure 5, the LLM-generated seed precisely aligns with the authentication logic:

```
// inferred protocol magic (MS-CDP)
0x30, 0x30,             // Signature == 0x3030

// inferred length field
0x00, 0x2F,             // MessageLength (47 bytes)
                        // Satisfies length >= 0x2A

// ... complete message header & payload ...
```

Figure 5: An LLM-synthesized seed template for the MS-CDP protocol. The generated input correctly satisfies the critical signature (`0x3030`) and length constraints, enabling the fuzzer to bypass the initial structural barrier in `CDPSvc`.

Because the generated seeds satisfied the `Signature` and `MessageLength` constraints, **REDOSPECTOR** successfully penetrated the validation layer, thereby gaining access to the deep, post-authentication state machine. In contrast, the baseline fuzzer, lacking this semantic insight, generated inputs that were rejected at the gate, rendering the complex core logic inaccessible to the analysis.

Efficacy of Proportionality-based Filtering

A core challenge in resource leak detection is distinguishing genuine leaks from benign noise (e.g., lazy initialization or caching). We evaluated the effectiveness of our Linear Proportionality Check (LPC) and temporal amplification by comparing three oracle configurations:

- **Naive Threshold:** Flags any input where memory usage grows by more than a fixed threshold (e.g., 10 KB) after transmission.
- **Amplification Only:** Repeats the input n times and checks for growth, but without call-stack correlation or linearity

checks.

- **REDOSPECTOR (Amplification + LPC):** The full oracle described in subsection 3.5.

Table 4: Ablation study of the anomaly detection oracle on CDPSvc. True Positives (TP) are confirmed leaks; False Positives (FP) are benign noise.

Oracle Configuration	Total Alerts	FP	Precision
Naive Threshold	1,402	1,401	<0.1%
Amplification Only	145	121	16.5%
REDOSPECTOR (Amp + LPC)	**21**	**1**	**95.2%**

Table 4 presents the number of candidates reported and the False Positive Rate (FPR) during a one-hour run specifically on the CDPSvc target. Ground truth was established through rigorous manual verification.

```
SslClientCertInfoSize = 0x820
v12 = (struct _HTTP_SSL_CLIENT_CERT_INFO *)realloc(i,
↪  SslClientCertInfoSize);
v11 = HttpReceiveClientCertificate(v7, v6, 0, v13,
↪  SslClientCertInfoSize, &BytesReceived, 0i64);
if ( v11 != 0xEA )
{
  ...
  return 0;
}
```

Figure 6: Specific control flow branch in MSStrgSvc that results in a memory leak.

The *Naive Threshold* generated an unmanageable number of alerts due to normal background noise. While *Amplification Only* reduced the overall volume, it still resulted in a high false positive rate (121 out of 145 alerts), as it failed to distinguish genuine leaks from benign caching behaviors where resources grew monotonically but non-linearly.

#	Function Call Site
00	concrete!concrete::HTTPListener::ValidateClientCertificate+0x8f
01	concrete!concrete::HTTPListener::Run+0x103
02	concrete!concrete::ThreadProc+0x109

Figure 7: Call stack of the memory leak in MSStrgSvc (CVE-2025-33068).

The full **REDOSPECTOR** configuration successfully filtered out virtually all background noise. Utilizing LPC, it enforced a strict linear relationship ($y \approx kx$) at specific call sites to eliminate non-linear caching behaviors. Furthermore, through our manual analysis of residual false positives, we identified that the most common benign anomalies occur during packet-processing initialization. To address this, **REDOSPECTOR** establishes a baseline residual footprint by sending benign packets prior to testing. During fuzzing, allocations originating from the same initialization call stacks are explicitly treated as false positives if their residual memory does not exceed this established baseline. Consequently, the full configuration reported only 21 candidates, 20 of which were confirmed as genuine leaks triggering the discovered DoS vulnerabilities in `CDPSvc`, leaving only a single false positive. This demonstrates that combining strict proportionality analysis with initialization baseline filtering is critical for achieving high precision in practical deployment.

4.5 Case Studies

Case Study I: MSStrgSvc Memory Leak (CVE-2025-33068). One significant vulnerability uncovered by **REDOSPECTOR** is CVE-2025-33068, a persistent DoS flaw in the Windows Standards-Based Storage Management Service (`MSStrgSvc.exe`), implemented within `concrete.dll`. The vulnerability resides in `concrete::HTTPListener::ValidateClientCertificate`. As shown in Figure 6, when the service is configured to require client certificates, it allocates `0x820` bytes via `realloc` for certificate-related data structures. When an attacker sends a crafted HTTP request lacking the expected client certificate, the internal call to `HttpReceiveClientCertificate` returns a specific status code (e.g., `0x490` "Element not found"). This return value steers the execution into a specific control flow branch. Crucially, within this branch, the function executes an early return statement without releasing the previously allocated buffer. As shown in Figure 7, this leak occurs in a high-frequency listener thread. By repeatedly sending such crafted HTTP requests, an attacker can leak memory at a constant rate until the process exhausts its address space and eventually crashes.

Case Study II: CDP Service Pre-auth DoS (CVE-2025-21207 Bypass). Another representative vulnerability is a pre-authentication resource consumption flaw in the Connected Devices Platform (CDP) service (`CDPSvc`), implemented in `cdp.dll`. This ubiquitous Windows 11 component manages cross-device connectivity and interactions. The flaw arises from asymmetric resource management during protocol exception handling. Upon receiving a connection packet, the CDP service invokes `cdp::MessageAccumulator::AccumulateMessageData` to validate the message length (`msgLen`). If `msgLen` is smaller than the mandatory base header size (`0x2A`), the service detects the anomaly and throws an exception, as shown in Figure 8. The core issue is the missing cleanup before `cdp::CdpThrow` is invoked. While the malformed packet is correctly identified, the service throws the exception without calling `cdp::TcpTransport::Disconnect` or any equivalent cleanup routine. As a result, the memory and handles associated with the `TcpTransport` session are never reclaimed. This bug effectively bypasses the previous fix for CVE-2025-21207 by shifting the leak into a different error path. Figure 9 shows a representative call stack for the resource leak.

```
// cdp!cdp::MessageAccumulator::AccumulateMessageData
cdp::BigEndianStreamReader::ReadNumber<unsigned short>(v28,
↪  msgLen);

if (msgLen < 0x2Au) {
    cdp::CDPSourceLocationInfo::CDPSourceLocationInfo(
        v30, "messageaccumulator.cpp", 101);
    v22 = cdp::MakeException<
              cdp::HResultException<-2147221246>>(
              v34, v30,
              "Message must have a length longer than the base
              ↪  header size");
    // Root cause: throwing exception without cleaning up
    ↪  TcpTransport
    cdp::CdpThrow<cdp::HResultException<-2147221246>>(v30, v22);
}
```

Figure 8: Error-handling path in CDP that leaks `TcpTransport` resources.

#	Function Call Site
00	`cdp::MessageAccumulator::AccumulateMessageData`
01	`cdp::TransportManager::GetMessagesFromData+0x107`
02	`cdp::TransportManager::OnTransportReceivedDataInternal`
03	`cdp::OneCoreWorkItemDispatcher::ProcessWorkitems+0x1f1`
04	`cdp::OneCoreWorkItemDispatcher::ThreadPoolProc+0x48`

Figure 9: Call stack demonstrating the crash path. Note that the excessive recursion occurs within `AccumulateMessageData`.

4.6 Summary

Our evaluation shows that **REDOSPECTOR** is effective at uncovering critical resource-exhaustion vulnerabilities that elude traditional crash-centric fuzzers. The discovery of 19 high-impact vulnerabilities, including issues that prompted architectural changes in core Windows services, highlights the importance of temporal resource telemetry and resource-aware oracles in modern vulnerability research. At the same time, since our analysis is performed offline, the additional runtime overhead introduced by heap monitoring and temporal amplification remains moderate (approximately 20% on average), making **REDOSPECTOR** practical for standard testing campaigns.

5 Discussion

In this section, we discuss the scope and limitations of **REDOSPECTOR**, its relationship to existing fuzzing paradigms, and several design choices that may affect its applicability in practice.

Generality, Impact, and Applicability. While **REDOSPECTOR** currently focuses on detecting non-crashing, release-bypass vulnerabilities in closed-source Windows remote services via `ntdll`-level heap telemetry and process metrics (e.g., private bytes and handles), the underlying risks broadly apply to long-running server programs across various operating systems. Within Windows alone, these services form a massive attack surface deployed across billions of devices, where a single service failure can trigger widespread disruption (e.g., LSASS leaks, CrowdStrike outage). Crucially, memory exhaustion in a single service causes system-wide disruption by continuously consuming physical memory and starving co-located processes. Because these attacks are cumulative and normal memory usage is dynamic under varying loads, naive mitigations such as increasing physical memory or enforcing per-process limits are ineffective. Although **REDOSPECTOR** excels at identifying user-space per-request leaks and state retention, it may currently miss purely kernel-side leaks or CPU-only DoS; however, extending its monitoring to additional APIs and resource counters remains a straightforward future direction.

Reliance on Heap Telemetry. Heap-centric observation is uniform across allocator implementations, but it can be ambiguous: benign caching/pooling may look like leaks over short windows, allocator reuse can weaken address-level pairing, and a single allocation site may correspond to multiple logical object types. **REDOSPECTOR** mitigates these issues with temporal amplification and call-stack–correlated proportionality checks, but some false positives/negatives can remain, especially on highly asynchronous services where attribution to specific inputs is harder.

Protocol Awareness and LLM Limitations. We use LLMs conservatively to generate protocol templates from static artifacts to improve reachability of deep protocol states. However, incorrect field inference (e.g., length fields) can waste fuzzing budget, and output quality depends on the completeness of the extracted binary features. Closing the loop by using runtime feedback to iteratively refine templates is a promising direction.

Trade-offs in Temporal Amplification. Amplification increases sensitivity to low-amplitude leaks but reduces throughput by repeating each input many times, and it can trigger secondary crashes/timeouts that complicate triage. Our two-stage schedule (screening then confirmation) balances cost and confidence, but optimal settings remain target-dependent.

Interaction with Coverage-guided Fuzzing. REDOSPECTOR is orthogonal to the input-generation strategy, but amplification changes the cost model of testcases and can distort naive coverage-guided scheduling. A practical approach is to decouple exploration and confirmation: run cheap coverage-driven fuzzing most of the time, and periodically subject promising inputs to **REDOSPECTOR**'s amplified resource oracle.

Operational Considerations. We recommend isolated VMs with snapshots, since instrumentation and aggressive fuzzing can stress both services and the OS. In some enterprise environments, security products or policies may interfere with DLL injection or heap instrumentation; while our design aims to remain compatible with CFG and avoids kernel modifications, such operational constraints may still limit applicability on certain targets.

6 Related Work

Our work bridges the gap between closed-source vulnerability discovery and resource exhaustion detection.

Static Resource Analysis. Static analysis detects bugs without execution but relies heavily on source code visibility. Early approaches like Saturn [47] and FastCheck [7] utilized boolean constraints and value-flow analysis to track memory states in C programs. In the context of OS kernels, resource leaks frequently reside in error-handling paths. Tools like Hector [32] and EPEx [49] specifically inspect error paths for missing resource releases. More recently, MLEE [44] detects leaks on "early-exit" paths by cross-checking deallocations against normal paths, while IPPO [20] and IMMI [19] leverage path similarity and Large Language Models (LLMs) to infer inconsistent memory management intentions. While effective for open-source ecosystems, these methods are inapplicable to proprietary Windows services where source code and precise control-flow graphs are unavailable. Additionally, static tools often struggle to distinguish "logical leaks" (unbounded cache growth) from legitimate retention.

Resource-Oriented Fuzzing. Unlike traditional coverage-guided fuzzers (e.g., AFL, WinAFL) that prioritize code reachability, resource-oriented fuzzers optimize for resource consumption. SlowFuzz [31] and PerfFuzz [18] guide mutation toward maximizing execution path length and instruction counts to expose algorithmic complexity vulnerabilities (e.g., CPU exhaustion). Focusing on memory, MemLock [45] tracks heap usage to uncover uncontrolled memory allocation and recursion bugs. However, these tools typically require compile-time instrumentation (e.g., LLVM passes) to obtain fine-grained resource feedback, making them unsuitable for binary-only targets. Furthermore, existing approaches often maximize absolute resource usage, which can be noisy in complex stateful services. **REDOSPECTOR** differentiates itself by using hook-based binary telemetry and an amplification oracle to detect proportional growth trends rather than absolute spikes.

Dynamic Detection and Protocol Fuzzing. Dynamic Binary Instrumentation (DBI) tools like Valgrind [27] and Dr. Memory [6] are the industry standard for binary leak detection. Despite their precision, they incur high runtime overhead and often conflict with Windows security mechanisms (e.g., Protected Processes), rendering them impractical for fuzzing system services. In the domain of network services, protocol fuzzers like Peach and Boofuzz [30] generate structured inputs but rely on coarse-grained crash oracles (e.g., timeouts), missing silent resource leaks. While recent works leverage LLMs for protocol understanding [19], they typically lack the specialized resource telemetry required to identify Denial-of-Service conditions in opaque binaries.

Windows Platform Security. Several works focus on the Windows platform. Winnie [16] synthesizes lightweight harnesses to bypass GUI code and directly invoke target functions, while WinFuzz [36] achieves fast in-memory looping via target-embedded snapshotting that resets program state on each iteration. Digtool [29] detects kernel vulnerabilities on binary code through a virtualization monitor. Other tools target specific Windows attack surfaces: PrintXPSurge [41] uses error messages to guide coverage-based fuzzing of XPS parsing in printing components, and OLExplore [40] systematically analyzes embedded-object (OLE) vulnerabilities; GLEIPNIR [13] detects server-induced client vulnerabilities in Windows Remote IPC, and XALPC [21] hunts vulnerabilities across the Windows ALPC attack surface at scale. For the Component Object Model (COM), COMRace [12] detects data race vulnerabilities via static binary analysis and PoC validation, and COMFUSION [52] is the first to identify union type confusion in Windows COM.

7 Conclusion

Resource-exhaustion bugs in Windows remote services can silently accumulate across requests and, under the `svchost.exe` fate-sharing model, degrade the availability of multiple co-hosted services. This work introduced **REDOSPECTOR**, a resource-oriented fuzzing framework for closed-source Windows binaries that combines lightweight heap telemetry, amplification-based detection of linear growth trends, and proportionality-based filtering to separate real leaks from noise, with protocol-aware seeding to reach deep states. Evaluated on recent Windows 11 and Windows Server builds, **REDOSPECTOR** uncovered 19 vulnerabilities (12 assigned CVEs) across core services and helped motivate vendor-confirmed hardening, demonstrating that resource-aware oracles can expose release bypass vulnerabilities that crash-centric fuzzing typically misses.

Acknowledgment

This work was partly supported by the National NSF of China (grants No.62302181) . Mu Zhang was supported in part by NSF OAC-2530911 and Battelle Energy Alliance, LLC Contract No. 313367. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agency.

A Ethical Considerations

Our work aligns with the ethical principles outlined in the Menlo Report, namely respect for persons, beneficence, justice, and respect for law and public interest. Although this study does not involve human subjects or personal data, the security weaknesses analyzed in critical Windows components carry inherent dual-use risks. Specifically, the defensive value of identifying resource-exhaustion vulnerabilities must be balanced against the possibility that malicious actors could adapt these techniques to degrade the availability of deployed services.

Decision to Conduct and Publish: Our decision to conduct and publish this research is grounded in the principle of beneficence, specifically the need to address a systemic asymmetry in security testing. Resource-exhaustion vulnerabilities in proprietary operating system components are frequently overlooked by traditional crash-centric fuzzing, leaving critical infrastructure quietly exposed. By systematically analyzing these weaknesses and publishing our methodology, we aim to raise awareness within both the vendor ecosystem and the broader security community. Proving the feasibility of these utility-degrading attacks is a necessary precursor to developing robust, systematic defenses, as it shifts the industry focus from reactive patching to proactive resource-limit validation.

We identified several stakeholders affected by this research. Direct stakeholders include Microsoft and the Microsoft Security Response Center (MSRC), who are responsible for triage and remediation; system administrators and developers, who bear the operational burden of patching and maintaining service availability; and end users whose productivity depends on these systems. Indirect stakeholders include the broader security community and cloud providers, while malicious actors represent adverse stakeholders. The primary harms we considered include service degradation, operational disruption, and financial loss if these techniques are applied to production systems, as well as the risks of premature public disclosure lowering the barrier to abuse.

To mitigate these potential harms and maximize defensive value, we implemented the following safeguards:

- **Controlled Testing:** All experiments were conducted in controlled, isolated environments (such as test machines and virtual machines with restricted network exposure) to prevent any impact on production systems or third parties.
- **Coordinated Disclosure:** We followed a Coordinated Vulnerability Disclosure (CVD) process, reporting all confirmed issues to MSRC and sharing sufficient technical details to facilitate triage and patch development.
- **Ethical Artifact Release Strategy:** In releasing our replication package, we carefully balanced academic reproducibility with harm reduction. We opted for a tiered release strategy rather than withholding artifacts entirely, as complete withholding often imposes a disproportionate burden on defenders who must secure multiple complex systems:
  - *What we release to support reproducibility:* We make the complete source code of our instrumentation infrastructure (the allocation-tracking Hook DLL and injector tool), five service-specific fuzzing harnesses (for CDP, MSMQ, RDG, Storage Management, and WDS), the Python scripts for conducting BooFuzz comparison experiments, and LLM prompts publicly available. This equips defensive researchers with the tooling necessary to verify our findings and test their own systems.
  - *What we withhold to prevent abuse:* We withhold all pre-compiled binaries of our testing tools and any end-to-end, ready-to-use exploit payloads. This policy applies universally; even though patches may be available for some services, real-world patch deployment is often delayed, meaning that public exploits would still pose an active risk to unpatched systems.

  By decoupling the testing and instrumentation framework from ready-to-use exploit delivery, we attempt to provide a practical template for future security research to maintain high reproducibility standards while minimizing immediate abuse potential by low-skilled actors.

Through these measures, we aim to contribute to ecosystem security by enabling the detection and remediation of a class of resource-exhaustion vulnerabilities that is often overlooked in crash-centric testing.

B Open Science

To support reproducible research and facilitate follow-on work, we release the complete source code of our artifacts at: https://doi.org/10.5281/zenodo.20623257. This repository contains the core components of our evaluation: (1) the instrumentation infrastructure (including the allocation-tracking Hook DLL and injector tool), (2) complete service-specific fuzzing harnesses and configurations for all five evaluated Windows services (CDP, MSMQ, RDG, Storage Management, and WDS), (3) baseline Boofuzz testing scripts, (4) protocol seed templates, and (5) the LLM prompts used for protocol constraint extraction.

In formulating our release strategy, we balance the goal of open security research with the risk of potential misuse. We recognize that withholding fuzzing harnesses and testing frameworks often imposes a disproportionate burden on defenders—who must secure multiple services—relative to

targeted attackers. Consequently, we provide the full instrumentation suite and all five service-specific harnesses to support defensive evaluation. To prevent immediate misuse, we exclude pre-compiled binaries and specific end-to-end exploit payloads that trigger denial-of-service on unpatched systems. This approach provides independent researchers and defenders with the necessary tools for verification and testing, while avoiding the dissemination of ready-to-use exploit utilities.

References

[1] "sulley," https://github.com/OpenRCE/sulley, 2014.

[2] "Peach Fuzzer," https://peachtech.gitlab.io/peach-fuzzer-community/, 2024.

[3] Angelboy, "Windows kernel heap," 2020.

[4] C. Bayet, "Windows 10 pool party," 2017, accessed: 2025-03-01. [Online]. Available: https://github.com/cbayet/Exploit-CVE-2017-6008/blob/master/Windows10PoolParty.pdf

[5] F. Bayet, "Pool overflow exploitation since windows 10 19h1," https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf, 2020.

[6] D. Bruening and Q. Zhao, "Dr. memory: a memory debugging tool," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2011, pp. 213–223.

[7] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 480–491.

[8] Clang, "Leaksanitizer," https://clang.llvm.org/docs/LeakSanitizer.html, accessed: 2026-02-03.

[9] Dr. Memory, "Dr. memory," https://drmemory.org/, accessed: 2026-02-03.

[10] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in *The 2021 Annual Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[11] Google Project Zero, "Tinyinst," https://github.com/googleprojectzero/TinyInst.

[12] F. Gu, Q. Guo, L. Li, Z. Peng, W. Lin, X. Yang, and X. Gong, "COMRace: Detecting data race vulnerabilities in COM objects," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3019–3036. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/gu-fangming

[13] F. Gu, Q. Guo, J. Lu, Q. Xie, B. Zhao, K. Lu, H. Li, and X. Gong, "Sheep's clothing, wolf's data: Detecting server-induced client vulnerabilities in windows remote IPC," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[14] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[15] Z. Jia, S. Li, T. Yu, X. Liao, J. Wang, X. Liu, and Y. Liu, "Detecting error-handling bugs without error specification input," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19, 2020.

[16] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[17] R. Lakshmanan, "Smbleed: A new critical vulnerability affects Windows SMB protocol," https://thehackernews.com/2020/06/SMBleed-smb-vulnerability.html, accessed: 2020-06-09.

[18] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 254–265.

[19] D. Liu, Z. Lu, S. Ji, K. Lu, J. Chen, Z. Liu, D. Liu, R. Cai, and Q. He, "Detecting kernel memory bugs through inconsistent memory management intention inferences," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4069–4086. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-detecting

[20] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen, and Q. He, "Detecting missed security operations through differen-

tial checking of object-based similar paths," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 1627–1644.

[21] H. Liu, F. Dong, Y. Tian, M. Zhang, X. Li, F. Gu, Z. Peng, and H. Wang, "Needle in a haystack: Automated and scalable vulnerability hunting in the windows alpc sea," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 2054–2068. [Online]. Available: https://doi.org/10.1145/3719027.3765180

[22] H.-Q. Liu, J.-J. Bai, Y.-P. Wang, and S.-M. Hu, "Bpminer: Mining paired functions from the binary code of drivers for error handling," in *2014 21st Asia-Pacific Software Engineering Conference*, 2014.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[24] T. Mandt, "Kernel pool exploitation on windows 7," https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf, 2011, accessed: 2025-01-17.

[25] Microsoft, "Performance issues are caused by a memory leak on a windows server 2008-based print server that has many windows vista-based printer clients," https://support.microsoft.com/en-us/topic/performance-issues-are-caused-by-a-memory-leak-on-a-windows-server-2008-based-print-server-that-has-many-windows-vista-based-printer-clients-83d45d42-14a9-6e3f-4c78-4018a42c7d8e, accessed: 2026-02-03.

[26] ——, "Event tracing for windows (etw)," https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-portal, 2024, accessed: 2024-05-20.

[27] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 89–100.

[28] R. Noureen, "Microsoft rolls out fix for lsass memory leak bug affecting windows server," https://petri.com/microsoft-fix-lsass-memory-leak-bug-windows-server/#:~:text=To%20prevent%20LSASS%20Memory%20Leak,process%20monitoring%2C%20and%20behavioral%20analytics., accessed: 2025-06-04.

[29] J. Pan, G. Yan, and X. Fan, "Digtool: a virtualization-based framework for detecting kernel vulnerabilities," in *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017, pp. 149–165.

[30] J. Pereira, "Boofuzz: A fork and successor of the sulley fuzzing framework," *GitHub repository*, 2019, https://github.com/jtpereyda/boofuzz.

[31] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2155–2168.

[32] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[34] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005.

[35] Y. Shafir, "Windows heap backed pool: The good, the bad, and the encoded," https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Windows-Heap-Backed-Pool-The-Good-The-Bad-And-The-Encoded.pdf, 2021, presented at Black Hat USA.

[36] L. Stone, R. Ranjan, S. Nagy, and M. Hicks, "No linux, no problem: fast and correct windows binary fuzzing via target-embedded snapshotting," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4913–4929.

[37] A. Tarakanov, "Data only: Pwning Microsoft Windows kernel - exploitation of kernel pool overflows on Microsoft Windows 8.1," Presented at Black Hat USA 2014, 2014, accessed: 2025-03-01. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Tarakanov-Data-Only-Pwning-Microsoft-Windows-Kernel-Exploitation-Of-Kernel-Pool-Overflows-On-Microsoft-Windows-8.1.pdf

[38] N. Tarakanov, "Automating windows kernel pool overflow/corruption exploits development," Presented at Power of Community 2018, 2018, accessed: 2025-03-01.

[Online]. Available: https://powerofcommunity.net/poc2018/nikita.pdf

[39] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017.

[40] Y. Tian, F. Dong, H. Liu, M. Xu, Z. Peng, Z. Ye, S. Li, X. Luo, and H. Wang, "Be careful of what you embed: Demystifying ole vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[41] Y. Tian, F. Dong, J. Wang, M. Zhang, Z. Peng, Z. Ye, X. Luo, and H. Wang, "Error messages to fuzzing: Detecting xps parsing vulnerabilities in windows printing components," in *Proceedings of the 32nd ACM Conference on Computer and Communications Security (CCS'25)*, Taipei, Taiwan, October 2025.

[42] C. Valasek, "Understanding the low fragmentation heap," http://illmatics.com/Understanding_the_LFH.pdf, 2010.

[43] ——, "Windows 8 heap internals," https://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf, 2012, presented at Black Hat USA 2012.

[44] W. Wang, "MLEE: Effective detection of memory leaks on early-exit paths in OS kernels," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 31–45.

[45] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020.

[46] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2041–2058. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wu-qiushi

[47] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005.

[48] M. V. Yason, "Windows 10 segment heap internals," https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals.pdf, 2016, presented at Black Hat USA 2016.

[49] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API usages through semantic cross-checking," in *25th USENIX Security Symposium*, 2016, pp. 363–378.

[50] D. Zhan, X. Yu, H. Zhang, and L. Ye, "Errhunter: Detecting error-handling bugs in the linux kernel through systematic static analysis," *IEEE Transactions on Software Engineering*, 2023.

[51] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "How effective are they? exploring large language model based fuzz driver generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1223–1235. [Online]. Available: https://doi.org/10.1145/3650212.3680355

[52] Y. Zhang, X. Zhu, D. He, M. Xue, S. Ji, M. S. Haghighi, S. Wen, and Z. Peng, "Detecting union type confusion in component object model," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.