Sheep's Clothing, Wolf's Data: Detecting Server-Induced Client Vulnerabilities in Windows Remote IPC

Fangming Gu[1,2,†], Qingli Guo[1,2,†,✉], Jie Lu[3,✉], Qinghe Xie[1,2], Beibei Zhao[1,2], Kangjie Lu[4], Hong Li[1,2,], Xiaorui Gong[1,2]
[1]*Institute of Information Engineering, CAS, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
[3]*Institute of Computing Technology, CAS, China*
[4]*University of Minnesota*
{gufangming, guoqingli, xieqinghe, zhaobeibei, lihong, gongxiaorui}@iie.ac.cn, lujie@ict.ac.cn, kjlu@umn.edu

***Abstract*—The Windows operating system employs various inter-process communication (IPC) mechanisms, typically involving a privileged server and a less privileged client. However, scenarios exist where the client has higher privileges, such as a performance monitor running as a domain controller obtaining data from a domain member via IPC. In these cases, the server can be compromised and send crafted data to the client.**

Despite the increase in Windows client applications, existing research has overlooked potential client-side vulnerabilities, which can be equally harmful. This paper introduces GLEIPNIR, the first vulnerability detection tool for Windows remote IPC clients. GLEIPNIR identifies client-side vulnerabilities by fuzzing IPC call return values and introduces a snapshot technology to enhance testing efficiency. Experiments on 76 client applications demonstrate that GLEIPNIR can identify 25 vulnerabilities within 7 days, resulting in 14 CVEs and a bounty of $36,000.

I. Introduction

Modern operating systems, including Windows and Linux, leverage Inter-Process Communication (IPC) for communication between processes, which may reside on the same or different machines. In this context, two primary roles are defined: the client and the server. The client initiates communication by sending requests, which the server receives, processes, and responds to. In remote IPC scenarios, where IPC data is transmitted over the network, client and server processes typically belong to different trust levels. Consequently, the operating system implements stringent security measures, such as access control and impersonation, to protect the trust boundary.

In recent years, numerous security research studies [1], [2], [3], [4], [5], [6], [7], [8], [9] have attempted to exploit the data exchanged between clients and servers to breach the trust boundary of servers. These studies consistently assume that the server is an entity of absolute trust, while the client is portrayed as an attacker, dispatching malicious payloads to uncover and exploit server vulnerabilities. This prevailing assumption is consistent with the typical use cases of IPC, where servers often handle critical operations and manage sensitive resources, thus operating with elevated privileges compared to clients. By June 2024, over 1,500 vulnerabilities associated with IPC servers have been reported to the Microsoft Security Response Center (MSRC) [10], highlighting that IPC servers are frequent targets for attackers.

However, existing research has overlooked a critical fact: in scenarios where IPC clients operate with higher privileges than IPC servers, which is common in remote management software [11], the data returned by the server can breach the trust boundary of the clients, leading to severe consequences.

Consider CVE-2024-38025 as an example, illustrated in Figure 1. The Windows domain controller employs a performance monitor to oversee the domain member. The performance monitor, acting as the IPC client, ❶ receives user interactions, such as clicks and text input. ❷ These interactions establish the necessary context for triggering an IPC call. ❸ The IPC protocol is then used to transmit this call, enabling data retrieval from the monitored machine,which plays the role

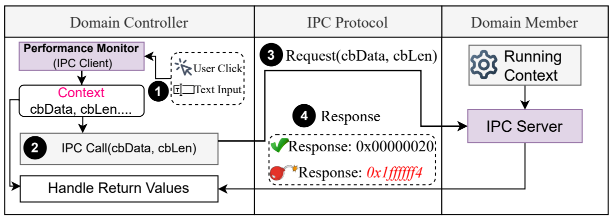


Fig. 1. A client vulnerability detected by GLEIPNIR (CVE-2024-38025). The vulnerable code segment is depicted in Figure 8.

† These two authors contributed equally to this work.
✉ Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241303
www.ndss-symposium.org

of the IPC server. If the domain member is compromised by an attacker, ❹the genuine data (`0x00000020`) is manipulated and altered to maliciously crafted data (`0x1ffffff4`). The performance monitor fails to verify this data, which can lead to memory corruption. Consequently, this vulnerability results in remote code execution and potentially compromising the entire domain controller. Due to blind trust in the server's return values and the limited research on such values, this vulnerability has been overlooked for 20 years.

With the increasing complexity and methodological diversity of modern network systems, operating system administrators are increasingly relying on various remote management software. Examples include Performance Monitor[12], Remote Desktop[13], Admin Center[14], Disk Management[15], TeamViewer[16], and VMware vCenter[17]. These tools employ IPC to manage lower-privileged computers within the network, which may be susceptible to attacker compromise. Consequently, IPC client-side management tools are exposed to significant threats that existing research has largely overlooked, leading to numerous potential vulnerabilities similar to CVE-2024-38025 in clients. Moreover, such vulnerabilities can be exploited in various ways, such as Man-in-the-Middle attacks and compromised network components, without necessarily requiring a malicious server. Therefore, there is an urgent need for a detection approach capable of identifying such client-side vulnerabilities.

A. Differences

Fuzzing is a well-established technique that has been proven effective in testing network applications [18], [19]. However, existing research has primarily focused on fuzzing servers, with limited attention given to client fuzzing. While some studies [20], [21], [18] claim to address client fuzzing, they often treat clients as servers, merely adapting server fuzzing techniques. This raises a critical question: is it effective to directly apply server fuzzing techniques to client fuzzing?

The answer is no, primarily due to the key difference in data processing between servers and clients. As shown in Figure 1, servers, once operational, typically receive and process mutated data immediately without building additional context. Although servers may need prior requests to create context for processing certain requests, when receiving an out-of-sequence request, they can still process or correctly reject it without disrupting normal operation.

In contrast, clients are highly context-dependent. Without proper context, clients may fail to execute IPC calls or process return values. This makes client-side testing, particularly fuzzing, more complex and requires careful context preparation. This context, comprising memory data that must be prepared before fuzzing, plays a crucial role in client fuzzing. For example, in Figure 1, user interactions, such as clicks and text input, drive the client to generate memory values for variables like `cbData` and `cbLen`, collectively referred to as the context. The execution of the IPC call relies on this context. Furthermore, this context plays a significant role in subsequent IPC return value handling.

B. Challenges

The key difference highlights the critical importance of context building in client fuzzing. Since constructing context is time-consuming, it's natural to consider employing snapshot-based fuzzing techniques to preserve the context. This approach allows for repeatedly restoring these snapshots, avoiding repeatedly constructing the context while injecting mutated return values, thereby efficiently detecting client vulnerabilities. However, before applying snapshot-based fuzzing techniques to clients, three significant challenges must be addressed in context construction and fuzzing execution.

Challenge 1: How can we precisely and automatically identify IPC clients from large-scale binaries?

Before constructing the context and applying snapshot-based fuzzing, it is crucial to identify appropriate clients as fuzzing targets. Windows operating systems offer a rich landscape of potential targets, encompassing numerous applications and APIs that utilize IPC. Although some efforts [22], [23] have been made to identify IPC servers for server fuzzing, there is currently no work focused on identifying IPC clients. Furthermore, for subsequent context construction phases, it is essential to pair clients with their corresponding servers, yet this task remains unaddressed. Since Windows is a closed-source system with a vast number of binary files, automating the identification of clients and accurately pairing them with servers can be challenging.

Challenge 2: How can we effectively trigger as many IPC calls in clients as possible to construct the context?

Once a client is identified, our testing targets become its code that processes IPC return values. Since a client may contain multiple IPC calls, triggering as many calls as possible increases the likelihood of exposing potential vulnerabilities. However, compared to servers that merely respond to IPC requests, client-side applications encompass a diverse array of testing targets, including GUI-based and CLI-based applications and public APIs utilized by developers. To fuzz the return values of IPC calls in a client application, it is necessary to prepare a context capable of executing the target IPC call. This context is established through GUI click events, user text input, and sequences of IPC calls. While previous efforts [24], [25] have attempted to automate the execution of these programs on Windows, they have largely relied on random strategies for inputting user interactions. This approach has proven ineffective in triggering IPC calls consistently, as demonstrated by our experiments in Section IV. Moreover, the execution of a single IPC call often depends on the execution and specific return values of several other IPC calls. This interdependency adds complexity to the process of triggering IPC calls. Therefore, developing an effective method to trigger these client-side IPC calls remains a significant challenge.

Challenge 3: How can we efficiently perform snapshot-based fuzzing for IPC clients on Windows ?

When an IPC call is triggered, we capture a snapshot to preserve its context and employ a snapshot-based fuzzing methodology to test IPC clients. However, two significant factors impede the efficiency of this approach on Windows

IPC clients. **Network Communication:** as demonstrated by Nyx-net [18], network communications are time-consuming and can significantly impact fuzzing performance. Although Nyx-net's network API emulation proved effective for Linux, it falls short for Windows remote IPC due to Windows' more complex and proprietary API structure. Furthermore, due to the closed-source nature of Windows, achieving accurate emulation becomes extremely challenging. **Non-memory-exception tests:** when restoring a snapshot and injecting a test case, a memory exception in the client process clearly indicates the test's conclusion. However, most tests do not result in memory exceptions. IPC clients, such as GUI programs, often run continuously, making it difficult to determine when a test can be concluded. There is no simple method solution for concluding tests that do not result in memory exceptions. An alternative approach involves timeout settings, but this introduces its own set of challenges. Setting a timeout that is too long reduces testing efficiency, while one that is too short may compromise the test's effectiveness. Balancing these factors remains a significant challenge and obstacle in developing effective fuzzing techniques for Windows IPC clients.

C. Solutions

Automatically inferring test targets. To address the first challenge, we conducted a comprehensive analysis of Windows' IPC mechanisms. We find that Windows uses standardized APIs for remote IPC, enabling us to develop a bottom-up algorithm for client identification. Additionally, we discovered that clients use server IDs (SIDs) and method IDs (MIDs) to identify servers and remote methods, enabling us to map clients and servers .

Effectively triggering IPC call. To address the second challenge, we observed that GUI clients primarily involve simple clicks and text inputs, CLI programs provide help commands and IPC APIs are well-documented in MSDN. By combining these characteristics with large language models, we developed automated methods for triggering IPC calls. To compute the specific return values of other IPC calls before executing the target IPC call, we symbolize the variables storing these return values and perform constraint solving to determine the required values.

Efficient Snapshot-Based Fuzzing. To address challenge 3, we propose corresponding solutions for each efficiency challenge. **Direct IPC API emulation:** to tackle the issue of emulating network efficiently, we adopted an approach that directly emulates remote IPC APIs, bypassing the need for low-level network API emulation. Our approach implements an IPC API hooking mechanism, which allows us to modify the return values of IPC calls directly with mutated values, effectively circumventing the actual execution of IPC calls. This method significantly reduces the complexity and overhead associated with network stack emulation. Moreover, it enables client fuzzing without initializing the corresponding server. **Adaptive termination:** to address the challenge of determining when to stop testing efficiently, we designed an adaptive approach based on the observation that achieving a certain threshold of dirty pages indicates sufficient testing coverage. This dynamic termination strategy optimizes the fuzzing process by balancing thoroughness with efficiency, avoiding unnecessary prolongation of testing while ensuring comprehensive coverage.

D. Contributions

We implemented our approach in a new tool named GLEIPNIR, which consists of three phases corresponding to the above three proposed solutions. We evaluated GLEIPNIR on Windows 11 where it correctly discovered 76 IPC clients and successfully triggered 2169 IPC remote calls, reporting 25 new vulnerabilities that had never been found before within 7 days. To date, Microsoft developers have confirmed 19 of them. Of these, 14 have been assigned CVEs, with awards totaling $36,000. All vulnerabilities are memory corruption bugs and can be exploited to achieve remote code execution (RCE) or sensitive information leakage.

The contributions of this paper are summarized as follows:

- We comprehensively reveal the security threats of remote IPC clients, which have been overlooked in Windows, and highlight the differences and challenges between testing the server and the client.
- We have designed a novel technique to address the challenges we identified. Our technique can precisely identify clients and servers, effectively prepare contexts, and efficiently fuzz clients.
- We have implemented our technique in a tool called GLEIPNIR. The tool successfully identified 25 critical vulnerabilities in 76 IPC clients, with 19 confirmed by MSRC and 14 assigned CVE numbers.
- To promote the discovery of client-side vulnerabilities in Windows and the development of snapshot technology for Windows, GLEIPNIR will be open-sourced at https://github.com/Anonymous130301/GLEIPNIR .

II. BACKGROUND & THREAT MODEL

This section first reviews the IPC in Windows, and then introduces the threat model.

A. IPC in Windows

According to Microsoft's official definition [26], Windows provides nine different IPC mechanisms. Applications use these IPC mechanisms to perform cross-process data exchange. Through further manual analysis, we found that three of the nine IPC mechanisms provide remote communication capabilities, as shown in Table I. These three IPC mechanisms operate similarly on the client side, typically following a three-stage process: (1) establishing a remote connection with the server to obtain a proxy, (2) accessing data from the server through the proxy, and (3) releasing the resources occupied by the proxy upon completing the interaction. Windows provides APIs for each stage across all three communication mechanisms, which developers can utilize directly.

Gleipnir is a term from Norse mythology, referring to a magical chain used to bind the wolf Fenrir.

TABLE I
CLIENT-SIDE IPC APIS USED IN RPC, COM AND WINSOCK.

IPC Stage	RPC API	COM API	WinSock API
Initialization	`RpcStringBindingComposeW`, `RpcBindingFromStringBindingW`	`CoCreateInstanceEx`	`WSAStartup`, `connect`
Data Access	`NdrClientCall(V1∼V4)`, `NdrAsyncClientCall(V1∼V2)`,`Ndr64AsyncClientCall`	`ObjectStublessClient`(Proxy Methods)	`getaddrinfo`, `send`, `recv`
Finalization	`RpcBindingFree`	`NdrCStdStubBuffer_Release`	`closesocket`, `WSACleanup`

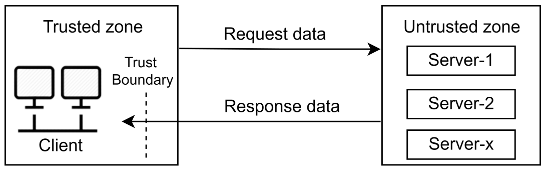


Fig. 2. The threat model discussed in this paper.

The remaining six IPC mechanisms are for local communication or have been deprecated. For example, *Pipes* are divided into *Named Pipes* and *Anonymous Pipes*, with the latter restricted solely to local communication. Although *Named Pipes* can support remote communication, they are primarily used as a foundational protocol for RPC. Other mechanisms, such as *Clipboard*, *File Mapping*, and *Data Copy*, are limited to local communication. Early mechanisms for remote IPC in Windows systems, such as *Dynamic Data Exchange* and *Mailslots*, have been deprecated and are no longer used in the latest versions of Windows.

B. Threat Model

Figure 2 illustrates the threat model discussed in this paper. In this model, contrary to common IPC scenarios, a client runs in a trusted zone and connects to one or more servers in an untrusted zone. If the client does not adequately validate the return values, these data could potentially breach the client's trust boundary, leading to severe vulnerabilities. These vulnerabilities, which result in trust boundary violations [27], are the primary focus of this paper.

This paper specifically focuses on detecting client-side vulnerabilities triggered through three remote IPC mechanisms: RPC, COM, and Winsocket. These mechanisms were chosen because they involve remote communication, where the client might run with higher privileges than the server. Local IPCs were excluded because, in local scenarios, clients typically have the same or lower privilege levels compared to servers, and thus are not within our scope. This decision was further validated through our discussions with Microsoft, who confirmed that client-side bugs in Local IPCs are not considered security issues.

It's also important to note that our fuzzing target is not the IPC protocol implementation but rather the client-side application code that processes IPC call return values. This distinction is crucial for understanding the scope of our study.

Furthermore, we focus solely on memory corruption vulnerabilities, which can lead to remote code execution and sensitive information disclosure impacts on the IPC client, thereby threatening the security of the administrator or domain controller in the network. In the most severe cases, attackers can exploit these vulnerabilities to take control of the entire network. Non-memory corruption vulnerabilities, such as access control vulnerabilities [28] or file hijacking [24], are not within the scope of our detection.

III. GLEIPNIR

Figure 3 presents a high-level overview of our proposed methodology. The tool operates through three distinct phases. In Phase 1, all binaries in Windows are read and static analysis techniques are used to identify clients using IPC and their corresponding servers. Phase 2 launches the clients and servers identified in Phase 1, using automated testing techniques such as UI automation and constraint solving to trigger as many IPC calls as possible. After each IPC call execution, snapshot technology is utilized to capture the context. In Phase 3, GLEIPNIR restores captured snapshots and employs hook techniques to directly modify return values of IPC calls with mutated values, thereby avoiding network communication. This phase also manages non-memory-exception tests by monitoring dirty pages and stopping the process when a specific threshold is reached. For memory-exception tests, GLEIPNIR captures and systematically categorizes exceptions. These categorized exceptions are then compiled into a comprehensive report, facilitating subsequent in-depth manual analysis.

A. Phase 1: Identify IPC Clients and Servers

In Phase 1, our tool leverages static analysis techniques to achieve three fundamental objectives: (1) identifying all client binaries, (2) identifying all server binaries, and (3) establishing precise client-server mapping relationships.

1) Identify IPC Clients: Windows supports nine remote communication modes, three of which are suitable for remote interactions as detailed in Table I. These three remote communication modes consistently utilize APIs to execute the steps listed in Table I.

Based on this finding, we designed a bottom-up analysis algorithm to identify clients. The algorithm employs a two-phase approach: initially, it identifies methods that invoke the IPC APIs documented in Table I, followed by a recursive traversal to locate all top-level methods (those not invoked by other methods). These top-level methods can be categorized into two distinct groups: publicly documented APIs in MSDN and entry points of Windows built-in applications. To accurately classify these methods, we first parsed MSDN documentation to create a public API list, then classified top-level methods by

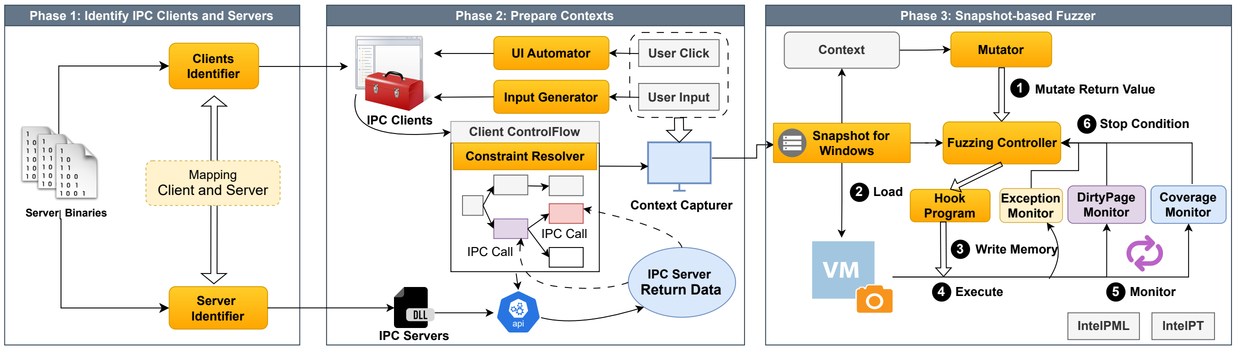


Fig. 3. Overview of GLEIPNIR. The orange boxes highlight the functional component of GLEIPNIR, which include the Client Identifier, Server Identifier, UI Automator, Input Generator, Constraint Resolver, Mutator, Hook, Fuzzing Controller, coverage Monitor, Dirtypage Monitor, and Exception Monitor.

checking their presence in this list. For built-in applications, we leveraged our observation that such applications must register as plugins within the Microsoft Management Console (MMC) to enable remote machine management. The registration process involves two methods: `RegisterSnapIn` and `RegisterSnapinGUID`. By analyzing binary files for calls to these registration methods, we can effectively identify built-in applications. Notably, built-in applications may also invoke public APIs, and in our classification scheme, these public APIs, when called by top-level methods, are considered clients themselves.

Furthermore, RPC and COM can be multiplexed, with their APIs supporting both local and remote method calls. When local methods are called, the client's execution privileges are lower than the server's, placing them outside our research scope. We need to distinguish between local and remote calls in the client. For COM, the API `CoCreateInstance` is used for local calls and is not considered as input for our algorithm. For RPC, we find that the `RpcStringBindingComposeW` API's second parameter, the `ProtSeq` variable, determines the protocol the program expects to use for remote RPC communication. Our observations show that the `ProtSeq` variable values 'ncacn_np' (Named Pipes protocol) or 'ncacn_ip_tcp' (TCP protocol) indicate that `RpcStringBindingComposeW` will be used to connect to a remote server. Based on this finding, our algorithm excludes local RPC calls from consideration.

2) Identify IPC Servers: To facilitate proper client-server interaction and enable IPC call triggering, it is crucial to identify the IPC server accurately. To achieve this, we first apply established methodologies [29], [1] to recover semantic information from binary code. Based on the recovered semantics, we identify IPC servers within extensive binary datasets by analyzing their specific code characteristics. The resulting output consists of server-containing binaries that facilitate server deployment.

RPC servers. To identify embedded RPC servers within binaries, we leverage the distinctive code characteristics of binaries containing RPC servers. Specifically, if a target binary includes an RPC server, its `.data` segment will contain a `MIDL_SERVER_INFO` structure. By scanning the binary's data segment for the presence of this structure, we can effectively identify binaries that implement RPC servers.

COM servers. The identification of COM servers requires a systematic examination of the Windows Registry, particularly within the *HKEY_CLASSES_ROOT**CLSID* key structure, where each COM server's registration information is maintained. This registration information includes the server's binary location. Servers restricted to local calls register their binary file locations in the *InprocServer32* key, while servers capable of remote invocation register their binary file location in the *LocalServer32* key. Given our threat model's focus on IPC with remote communication capabilities, we locate target COM servers by tracing the binary file paths specified in the *LocalServer32* key.

Winsock Servers. Winsock server programs establish port bindings through the server-side Winsock APIs, i.e., `bind` and `connect`. Therefore, GLEIPNIR identifies Winsock server binaries through static analysis specifically by detecting the concurrent invocation of `bind` and `connect` within the executable code.

3) Map Client and Server : Upon identifying the client and server, GLEIPNIR facilitates the mapping of IPC methods from the client side to their respective counterparts on the server side. To invoke a remote method via IPC, the client must specify a unique Server ID (SID) that distinctly identifies the method's service provider. This SID enables the IPC protocol to locate the corresponding server accurately. Additionally, a Method ID (MID) is required to pinpoint the exact method within the server. Therefore, achieving a static mapping between the client and server necessitates the accurate identification of both the SID and MID. The identification method is designed based on the IPC mechanism.

a) Identifying SIDs and MIDs on Client: **(1)RPC.** The `NdrClientCall` methods listed in Table I use the first parameter, `MIDL_STUBLESS_PROXY_INFO`, to identify parameters for the remote RPC request, including the SID and MID. We need to parse this parameter to obtain the SID and MID. The parsing process, illustrated in Figure 4, involves first parsing the first parameter of `MIDL_STUBLESS_PROXY_INFO` to obtain the `MIDL_STUB_DESC` structure, then parsing the first field of

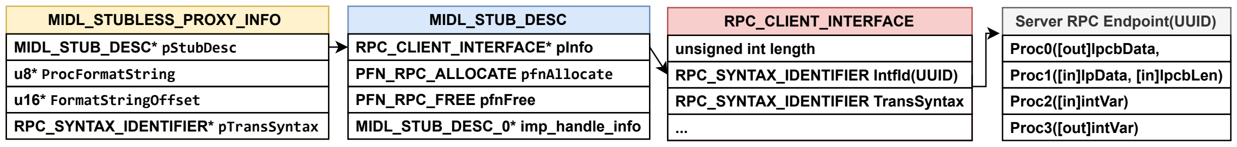

MIDL_STUBLESS_PROXY_INFO
MIDL_STUB_DESC* pStubDesc
u8* ProcFormatString
u16* FormatStringOffset
RPC_SYNTAX_IDENTIFIER* pTransSyntax
MIDL_STUB_DESC
RPC_CLIENT_INTERFACE* pInfo
PFN_RPC_ALLOCATE pfnAllocate
PFN_RPC_FREE pfnFree
MIDL_STUB_DESC_0* imp_handle_info
RPC_CLIENT_INTERFACE
unsigned int length
RPC_SYNTAX_IDENTIFIER IntfId(UUID)
RPC_SYNTAX_IDENTIFIER TransSyntax
...
Server RPC Endpoint(UUID)
Proc0([out]lpcbData,
Proc1([in]lpData, [in]lpcbLen)
Proc2([in]intVar)
Proc3([out]intVar)

Fig. 4. Parsing Procedure of MIDL_STUBLESS_PROXY_INFO.

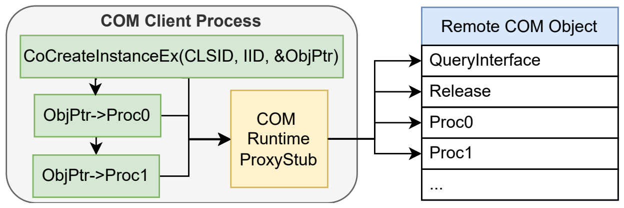


Fig. 5. CLSID, interface id and the method

`MIDL_STUB_DESC` to get the `RPC_CLIENT_INTERFACE` structure. The second field of `RPC_CLIENT_INTERFACE` contains the `RPC_SYNTAX_IDENTIFIER` structure, which identifies the SID of the target interface for the remote call initiated by `NdrClientCall`, i.e., the SID. Subsequently, we can obtain the method number of the remote IPC interface from the second parameter of `NdrClientCall`, nProcNum, i.e., the MID. By combining the SID and MID, we can uniquely determine the specific method in the RPC server that `NdrClientCall` intends to access. **(2)COM.** We found that the `CoCreateInstanceEx` method listed in Table I has remote calling capabilities. As shown in Figure 5, the first and second parameters of `CoCreateInstanceEx` represent the CLSID and IID of the remote COM object request, respectively. `CoCreateInstanceEx` returns a proxy(i.e., `ObjPtr`), which calls the remote method as if calling a regular method. Therefore, we can determine the remote method by the method signature. In summary, the CLSID and IID constitute the SID, and the method signature is the MID. **(3)Winsock.** The client connects to the server using the APIs `connect` and the `WSAStartup` , specifying the port. The client then uses the `send` and `recv` methods to send requests and receive responses. Thus, the port serves as the SID, with `send` corresponding to the server's `recv`, and `recv` corresponding to the server's `send`.

b) Identifying SIDs and MIDs on Server: (1)**RPC.** In the process of identifying RPC servers, the `MIDL_SERVER_INFO` structure is discovered. Utilizing the established tool findrpc [29], we parsed this structure to develop the Server RPC Endpoint data structure, as illustrated in Figure 4. This structure comprises a UUID field, serving as the SID, which uniquely identifies each RPC server. Additionally, it includes MIDs, labeled as Proc0, Proc1, etc. **(2)COM.** Utilizing the automated interface decompilation and enumeration methods provided by [30] and the virtual table recovery methods from [1], we recovered the CLSIDs, IIDs, and virtual table method lists of COM servers. Each COM object is uniquely identified by a CLSID and IIDs, forming the SID. Each COM object contains a method list, where each method is a member of the COM object and is callable remotely. These method signatures serve as MIDs. **(3)Winsock.** The server utilizes the `bind` and `connect` methods for network initialization. In this context, the SID corresponds to the port number specified in these methods. The server also uses the `recv` method for request handling and the `send` method for response. The signatures of these two methods are MIDs.

B. Prepare Contexts

To effectively perform fuzz testing on clients, it is essential to trigger as many IPC calls within the client as possible. This requires preparing the context to trigger IPC calls. Figure 3 illustrates how Phase 2 prepares the context. The *UI Automator* component is used to automate the triggering of UI events. The *Input Generator* serves public APIs and CLI programs, generating the necessary programs and commands to drive API execution. The *Constraint Resolver* is used to solve the expected return values of IPC calls. Next, we will analyze the characteristics of client application behaviors in sequence and demonstrates how these characteristics guided our component design decisions.

a) UI applications: For UI applications, IPC calls are primarily triggered by two types of events: user inputs and click events. The typical objective of the client UI program is to monitor the server, often requiring user input in the form of the monitored machine's address. For example, in the performance monitor scenario shown in Figure 1, to acquire data from a monitored machine, the user must input the target machine's IP address. Consequently, whenever user input is required, we supply the server's IP address. Furthermore, we observed that client programs are typically monitoring applications, and their UI events primarily involve simple click actions to retrieve data without the need for complex operations. Based on this observation, we implemented our own *UI Automator* to retrieve information about the UI and send inputs to controls. Using *UI Automator*, we enumerate all clickable and input elements in a GUI program's interface. Furthermore, we observe that the monitored applications demonstrate a structured hierarchy in the sequential ordering of their UI element interactions. Based on this observation, we adopt a depth-first traversal strategy starting from the top-level UI element, traversing different combinations of UI elements, and eventually covering all operational combinations of the UI. This methodical traversal triggers the UI application to generate numerous IPC requests.

TABLE II
EXAMPLES OF TESTING NFSADMIN.EXE

Prompt Type	Template	Instantiation
	Interaction 1: CmdLine Arguments Generation	
CmdLine Context	**Start Prompt:** [As a professional security researcher, we want to test the CommandLine based <AppName> App. We need to pass proper arguments to make it functional. Here's the default output of the App: <Hint>]	As a professional security researcher, we want to test the CommandLine based "nfsadmin.exe" App. We need to pass proper arguments to make it functional. Here's the default output of the App: "Invalid option argument. Usage: nfsadmin [server | client | mapping] [\\ host]. For detailed help type nfsadmin [server|client|mapping] /?"
Command Memory	[**List of Tested Commands:** "listgroups, addmembers, ..., <Output>"	[nfsadmin.exe, {Help Text}]
Command Question	What is the command currently being tested? What is the execution result of the command? (<CommandLine>+<result>)	What is the Command currently being tested? What is the execution result of the Command? ("nfsadmin.exe", "Invalid option argument")
Input question	What is the next commandline to execute?	What is the next commandline to execute?
LLM Answer	Current command: "<Command>". Status: <result>. Operation: "<NextInput>"	Current Command: "List Help". Status: Yes. Operation: "nfsadmin mapping /?"
	Interaction 2: CmdLine Arguments Generation	
CLI Context	The tested result is : <ReturnValue>	The tested result is: nfsadmin mapping [computer name] [common_options]...
Command Memory	[**List of Tested commands:** "listgroups, addmembers, ..., <Output>"	[nfsadmin.exe, "nfsadmin.exe mapping /?", {Help Text}]
Command Question	What is the command currently being tested? What is the execution result of the command? (<CommandLine>+<result>)	What is the command currently being tested? What is the execution result of the command? ("nfsadmin.exe mapping /?", "Yes")
Input Question	What is the next commandline to execute?	What is the next commandline to execute?
LLM Answer	Current command: "<command>". Status: result. Operation: "<NextInput>"	Current command: "List mapping's help". Status: Yes. Operation: "nfsadmin -u -p mapLookup"

TABLE III
EXAMPLES OF TESTING MPRAPI

Prompt Type	Template	Instantiation
	Interaction: Public API Generation	
API Context	**Start Prompt:** [We want to test APIs of <HeaderFile>. API-1 <InitAPI> is responsible for initializing connections, API-2 <ProcessAPIs> is responsible for processing user data inputs. Here is the reference information about the APIs <Reference>]	We want to test APIs of <MprAPI.h>. API-1 <MprAdminServerConnect> is responsible for initializing connections, API-2 <MprAdminPortEnum> is responsible for processing user data inputs. Here is the reference information about the APIs {MSDN-Docs}
Input question	Please compose C++ client for correctly using the two APIs.	Please compose C++ client for correctly using the two APIs.
LLM Answer	<Composed Client Code>	Simplified version is in Figure 6, complete version of generated client code is in the Appendix A.

```
#include <windows.h>
#include <mprapi.h>
...
// API-1: MprAdminServerConnect
dwError = MprAdminServerConnect(NULL,
  ↪  &hMprServer);
DWORD dwTotalEntries = 0;
DWORD dwEntriesRead = 0;
PMPR_PORT_0 pPorts = NULL;
DWORD dwLevel = 0;
// API-2: MprAdminPortEnum
dwError = MprAdminPortEnum(hMprServer, dwLevel,
  ↪  NULL, 0, &pPorts, &dwEntriesRead,
  ↪  &dwTotalEntries);
MprAdminServerDisconnect(hMprServer); // Added by
  ↪  GPT-4.
```

Fig. 6. Part of generated code by the query in Table III

b) CLI applications: For CLI applications, we found that they all include a `HELP` parameter that outputs usage information for the application. Therefore, for applications that interact with users via CLI, we extract the detailed `HELP` information provided by the command itself to automate

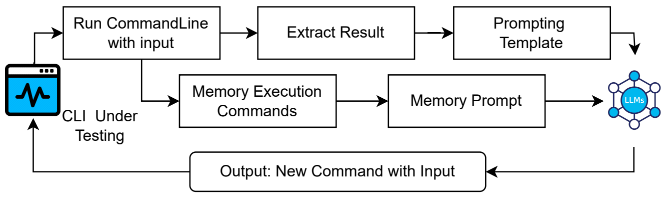

Run CommandLine with input
Extract Result
Prompting Template
CLI Under Testing
Memory Execution Commands
Memory Prompt
LLMs
Output: New Command with Input

Fig. 7. Workflow of testing CLI clients

the acquisition of essential elements needed to invoke target commands, using a large language model(LLM) to guide our input for reasonable CLI calls.

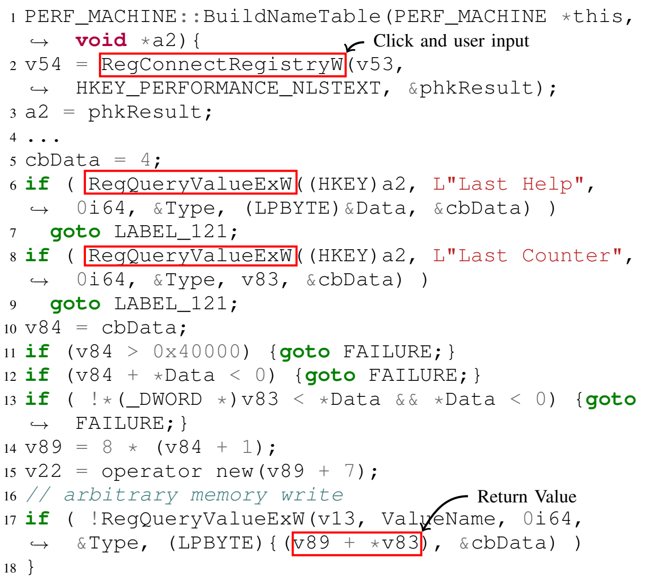
However, a CLI application has a nested command structure, a hierarchical organization where commands and options are arranged on multiple levels, resembling a tree. The root level contains the main command, subsequent branches represent subcommands that further specify actions, and the leaves are options or arguments providing detailed configuration. To obtain all possible commands of the CLI application, all paths in this tree must be traversed.

Based on this concept, we designed the workflow of the *Input Generator* in Figure 3 for testing the CLI application, as shown in Figure 7. The *Input Generator* iteratively executes the CLI application and uses the output results and execution history to guide the LLM in exploring all possible commands. The *Input Generator* first runs the target CLI program and extracts the program's output, incorporating it into the prompt. The output of the CLI program can be of two types: if the command is invalid, the output is help information, which guides the LLM to continue exploring valid commands in the next iteration; if the input is valid, the output is the result of the command execution, which guides the LLM on whether to terminate exploration on that path. Additionally, the *Input Generator* remembers the commands already executed and their outputs, including them in the prompt. This avoids repetitive exploration and guides the LLM in the next step's exploration direction. The combined prompts are then sent to the LLM to obtain the next command to be executed.

Table II shows the prompts for testing the nfsadmin.exe with LLM. **Interaction 1**: Initially, we execute `nfsadmin.exe`, and the `nfsadmin` program prints the help information. We put the help information into the *Prompt context* and use *Command Memory* to remember the current command. Then, we ask questions using *Command question* and *Input question*. The *Prompt context*, *Command memory*, *Command question*, and *Input question* form a prompt sent to the LLM, and we get an answer, including the next command to execute. **Interaction 2**: Next, we execute the next command given by the LLM and feedback the result into the *CLI Context*, adding the command to *Command Memory*. We then form a prompt with the *Command question* and input question and send it again to the LLM. We get the next command to execute: 'nfsadmin -u -p mapLookup', which is a valid command and triggers IPC execution. We will repeat the above process until no new commands are added to *Command Memory*.

c) Public APIs: For public APIs, the *Input Generator* extracts the usage context of each API from the MSDN documentation and uses an LLM to generate valid API calls. We also need to address dependencies between APIs. In Section III-A1, we identify which IPC APIs are included in the public API during client identification. As shown in Table I, there are dependency relationships between IPC calls. We derive public API dependencies based on these relationships. For instance, if public API-1 includes only initialization methods (e.g., `RpcBindingFromStringBindingW`) and public API-2 includes only data access methods (e.g., `NdrClientCall`), we infer that API-2 must be executed after API-1.

We create an API context by combining the MSDN documentation and the derived dependency relationships. Together with the input question, we use this context to prompt the LLM to generate C++ programs that drive the API calls. Figure 6 presents the simplified code generated by the LLM. The correct requests for API-1 and API-2 were



```
1  PERF_MACHINE::BuildNameTable(PERF_MACHINE *this,
   ↪  void *a2){
2  v54 = RegConnectRegistryW(v53,
   ↪  HKEY_PERFORMANCE_NLSTEXT, &phkResult);
3  a2 = phkResult;
4  ...
5  cbData = 4;
6  if ( RegQueryValueExW((HKEY)a2, L"Last Help",
   ↪  0i64, &Type, (LPBYTE)&Data, &cbData) )
7    goto LABEL_121;
8  if ( RegQueryValueExW((HKEY)a2, L"Last Counter",
   ↪  0i64, &Type, v83, &cbData) )
9    goto LABEL_121;
10 v84 = cbData;
11 if (v84 > 0x40000) {goto FAILURE;}
12 if (v84 + *Data < 0) {goto FAILURE;}
13 if ( !*(_DWORD *)v83 < *Data && *Data < 0) {goto
   ↪  FAILURE;}
14 v89 = 8 * (v84 + 1);
15 v22 = operator new(v89 + 7);
16 // arbitrary memory write
17 if ( !RegQueryValueExW(v13, ValueName, 0i64,
   ↪  &Type, (LPBYTE){(v89 + *v83), &cbData) )
18 }
```

Fig. 8. CVE-2024-38025: A client vulnerability in *Windows Performance Monitor*. The text at the starting point of the arrow indicates the conditions required to trigger the instruction pointed to by the arrow. `RegConnectRegistryW` and `RegQueryValueExW` serve as wrapper methods for the IPC APIs `RpcStringBindingComposeW` and `NdrClientCall` listed in Table I, respectively.

generated, ensuring the dependencies were maintained. The LLM also initialized the correct values for the API parameters (lines 6-9). Interestingly, we did not specify the use of `MprAdminServerDisconnect` in the prompt. The LLM automatically included this API call. We did not include `MprAdminServerDisconnect` because this API is only used for finalization and does not return a value.

d) IPC calls: When we need to trigger a target IPC call, the return values of a preceding series of IPC calls must meet specific conditions, as these return values appear in the control conditions for executing the target IPC call. Therefore, *Constraint Resolver* derives the path constraints from the last executed IPC call to the target IPC call. We use backward data flow analysis to collect these path constraints. If a loop is encountered, it should be unrolled 10 times. To accelerate constraint solving and minimize memory corruption, avoiding program errors, we symbolically mark the return value of the IPC call with the fewest constraints and use Z3 for constraint solving, setting a timeout of 5 minutes for each attempt. If a solution is found, we forcibly modify the return value to the derived solution. If not, we proceed to symbolically mark the return value of the IPC call with the second fewest constraints, and so on.

Taking Figure 8 as an example, the last executed IPC call occurs at line 8, and our target IPC call is at line 17. Three `if` statements between these lines (lines 11–13) generate four constraints: `v84 > 0x40000`, `v84 + *Data < 0`, `*v83 < *Data`, and `*Data < 0`. The variable `*v83` is involved in one constraint, `v84` in two constraints, and `Data` in three constraints. These variables hold return values from the IPC calls at lines 6 and 8. Consequently, we prioritize the symbolization of variables based on the number of constraints each variable influences: first `*v83`, then `v84` if unresolved, and finally `*Data` if necessary.

C. Snapshot-based Fuzzer

After Phase 2, multiple IPC calls are triggered, and their corresponding execution snapshots are captured. In Phase 3, GLEIPNIR iteratively tests these IPC calls until the execution time exceeds a user-defined threshold. For each IPC call, GLEIPNIR initiates a fuzzing controller to oversee and manage the testing process, as shown in Figure 3. The fuzzing process aligns with traditional snapshot-based fuzzing techniques by incorporating a *Mutator* for seed mutation, a coverage monitor for tracking code coverage, and an exception monitor for reporting potential vulnerabilities. To address the efficiency challenges associated with network communication and non-memory exception tests, GLEIPNIR employs two key components: a hook program and a dirty page monitor. These components enable two optimizations: direct IPC API emulation and adaptive termination, as described in Section I.

1) Fuzzing controller: As illustrated in Figure 3, the fuzzing controller manages the entire fuzzing process through six distinct steps. ❶ The fuzzing controller retrieves mutated return values from the *Mutator*, which generates these values by applying mutation operations to the existing entries in the seed queue. ❷ Upon acquiring this mutated value, it sends a command to the virtual machine to load a snapshot. ❸ Subsequently, it invokes the hook program to inject the mutated value into the memory address that stores the return value. ❹ The controller then instructs the virtual machine to resume executing the client process. ❺ At the start of the testing phase, the fuzzing controller initiates three distinct monitors to enable continuous data collection. ❻ Based on the collected monitoring data, these monitors make critical decisions regarding test progression: either terminating the current mutation test or halting the entire fuzzing process. Upon receiving a signal to terminate the mutation, the fuzzing controller begins a new iteration of steps ❶–❺ with the next mutation. Alternatively, if instructed to halt the entire fuzzing process, the controller saves the seed queue values and clears the queue. These saved values are subsequently restored when the same IPC call undergoes further testing iterations. The controller then advances to the next IPC call in the test sequence and reinitializes the testing cycle through steps ❶–❺. Additionally, the *Exception Monitor* outputs details of exceptions, facilitating manual analysis for potential vulnerabilities.

Notably, the fuzzing controller can obtain multiple mutated values from the *Mutator* at once and execute a snapshot for each value on separate CPU cores, enabling parallel testing of these mutated values.

2) Mutator: This component first checks for the existence of a previously persisted seed queue for the current IPC call. If such a queue is found, it restores saved values. Otherwise, it extracts the return value from the snapshot of the targeted IPC call and initializes the seed queue with this value as

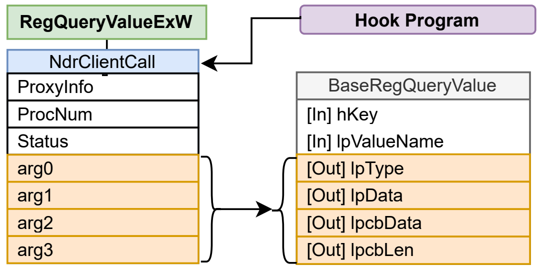


Fig. 9. The hook example for IPC call in Figure 8. `BaseRegQueryValue` is the server method. `NdrClientCall` is the IPC API in Tab I.

the primary seed for mutation. Subsequently, it employs AFL mutation strategies to generate a series of mutated values, updating the seed queue according to coverage feedback.

3) Hook program: Figure 9 illustrates our hook mechanism using the IPC call example presented in Figure 8. IPC calls utilize parameters (arg0–arg3) from the IPC APIs in Figure I to store memory addresses that point to buffers designated for storing return values. To transform mutated values into return values, we implement an interception methodology using inline hooks. This technique modifies the entry point of IPC APIs to redirect execution to a custom hook program, enabling inspection and manipulation of data being written to the target buffers, whose addresses are obtained from the API parameters.

The hook program operates through several distinct stages. Initially, it retrieves the mutated value corresponding to the IPC call. It then identifies the parameters that point to return value buffers. For RPC and COM APIs with variable-length parameter lists, parameters serve dual purposes: storing either request buffer addresses or response/return value buffer addresses. The determination of parameter types involves analyzing server-side methods within the IPC call. During Phase 2, we recover server method signatures (e.g., `BaseRegQueryValue` in Figure 9), which are tagged with `in` for request parameters and `out` for response parameters. As shown in Figure 9, there is a direct one-to-one mapping between server-side method parameters and IPC API parameters. This mapping enables the identification of parameters that store return values on the client side. In the case of Winsock, the return value address is stored in the second parameter, `buf`, of the `recv` method.

An important consideration is that an IPC call may contain multiple parameters pointing to different return value buffers. To address this complexity, we implement a two-phase approach. First, we consolidate these separate buffers into a single byte array to enable unified mutation handling. Then, during the propagation phase, the hook program randomly segments the mutated value into n portions, where n corresponds to the number of original buffers. These segments are systematically distributed to their respective target buffers.

4) Coverage Monitor: This monitor utilizes Intel's processor trace hardware [31] to accurately collect code coverage data, encompassing both basic block and edge coverage metrics. After each test case execution, the monitor aggregates the coverage results into a comprehensive fuzzing coverage profile. If the overall coverage does not increase within a predetermined period, defined by a user-specified threshold (default is 4 hours), the monitor alerts the fuzzing controller to terminate the current fuzzing process. The fuzzing controller then initiates testing of the next IPC call.

5) Dirty-Page Monitor: This monitor, based on Intel page modification logging [32], tracks the number of dirty pages in memory during non-memory exception tests. When the count of dirty pages exceeds 1,000, the fuzzing controller is notified to stop the current test. This adaptive termination strategy is based on a key observation from our testing process: in non-memory exception tests, the number of dirty pages typically remains below 300 before the client processes the IPC return values. However, once the IPC return values are fully handled, the number of dirty pages on the client side rapidly increases to over 1,000 due to activities such as UI maintenance. Performance experiments [33] conducted using Nyx indicates that when the number of modified pages exceeds 1,000, the execution speed drastically decreases from 1,000 operations per second to 100, significantly affecting testing efficiency. Consequently, our strategy for non-memory exception tests is to instruct the fuzzing controller to terminate the test if the number of dirty pages exceeds 1,000.

6) Exception Monitor: This monitor performs two essential functions within GLEIPNIR. Once an exception is detected, it first dumps detailed information about the exception. It then notifies the fuzzing controller to terminate the current test. In this section, we primarily discuss the first function.

The reason for dumping the exception details is that we rely on software exceptions thrown by the client to determine whether the mutated value has triggered a vulnerability. However, Windows has numerous exception types, and not all of them lead to security issues. For example, an `ArithmeticException` caused by division by zero does not pose a security threat. Although this exception might cause a crash if not caught by the client, it is generally not considered a Denial-of-Service (DoS) attack, and Microsoft does not recognize it as a security issue. Only one exception type, the Access Violation Exception, which occurs due to invalid memory access, is considered a security risk. This exception comprises three specific types:

- **ACCESS_VIOLATION_READ**: Occurs when reading from unauthorized memory, often due to invalid pointers or out-of-bounds access.
- **ACCESS_VIOLATION_WRITE**: Occurs when writing to unauthorized memory, usually due to pointer errors or exceeding array boundaries.
- **ACCESS_VIOLATION_EXECUTE**: Occurs when executing code from unauthorized memory, typically in code injection attacks or invalid addresses.

Thus, the monitor needs to distinguish between above three specific types of exceptions and other general exceptions.

The exception monitor is implemented based on the Windows independent exception-handling mechanism, Structured

TABLE IV
NUMBER OF INFERRED IPC CLIENTS

Client	Number	Matched Server	Expected Server	IPC Call			
				RPC	COM	Winsock2	Total
Built-in App	18	73	84	1,182	324	180	1,686
Public API	145	56	64	874	212	62	1,148
Total	163	129	148	2,056	536	242	2,834

Exception Handling (SEH) [34]. When an exception occurs, it is first caught by the kernel. If it originates from a user-space process, the kernel passes it back to the throwing process, invoking the registered exception handling code on the SEH. Based on this mechanism, the monitor intercepts all exceptions from the `RtlDispatchException` method, which belongs to SEH and primarily handles exception dispatching. If the exception is identified as an `Access Violation Exception` based on the exception code, the monitor tags it as vulnerability-induced. Otherwise, it is tagged as a bug. The monitor then extracts the call stack, key parameters and exception type, from the `EXCEPTION_RECORD` structure and dumps them along with our tags. Subsequently, we conduct a manual analysis of these exceptions to determine if they represent vulnerabilities and evaluate their attack vectors for possible exploitation. Although our primary goal is to identify vulnerabilities, we also record bugs because they may affect the client's functionality, and we responsibly report them to Microsoft.

IV. EVALUATION

We implemented GLEIPNIR on Windows. The implementation comprises 11.2k lines of C++ code and 5.0k lines of Python code. In Phase 1, the algorithms for identifying clients are mainly implemented in Python using an IDA plugin. In Phase 2, we developed our *UI Automator* based on Microsoft UI Automation [12]. We leverage GPT-4 as our large language model. We implemented the process in Figure 7 in Python, interacting with GPT-4 via its official API [35]. In Phase 3, we developed the controller in C++ and implemented hooks using Microsoft Detours [36].

We evaluated GLEIPNIR by analyzing all binaries on the Windows 11 and Windows Server 2025 platforms, ranging from Windows Insider Build 26040.1 to Build 26080.1. These analyses were conducted on systems running their default configurations on an i9-13900HX desktop (featuring 32 cores) equipped with 64GB of memory and a 2.0 TB SSD. For each IPC call, we concurrently tested 32 mutated values, with each test pinned to a dedicated core.

Our evaluation addresses the following research questions:

- RQ1: How effective is GLEIPNIR in identifying clients and servers?
- RQ2: How effective is GLEIPNIR in preparing the context?
- RQ3: How effective is GLEIPNIR in detecting client vulnerabilities?
- RQ4: How efficient is GLEIPNIR in fuzzing clients?

- RQ5: How does GLEIPNIR compare with other fuzzing approaches?
- RQ6: How effective are the strategies adopted by GLEIPNIR (ablation study)?

A. RQ1: Identifying clients and servers

Table IV shows the identification results on Windows Insider Build 26080.1. GLEIPNIR identified 18 built-in applications, 145 Public APIs, and 129 servers. However, the expected number of corresponding servers on the client side is 148. The missing servers are due to some legacy clients whose corresponding servers have been removed by Microsoft. We manually verified these clients and servers and found no false positives. These clients include a rich set of IPC calls totaling 2,834. Among these, the number of RPC calls accounts for 72.54%, COM IPC accounts for 18.91%, and Winsock IPC accounts for 8.54%. This result indicates that RPC and COM are the primary means of IPC. Although the number of built-in applications is only 18, these built-in applications contain a wealth of remote IPC calls. Therefore, the remote servers they support are more numerous than public APIs.

We aimed to assess the false negatives in our IPC client and server identification approach. Establishing ground truth for these entities proved challenging, so we focused our analysis on identifying potential sources of false negatives. One significant factor is the incomplete coverage of IPC APIs, as some are not included in Figure I. Additionally, GLEIPNIR 's reliance on existing tools [29], [1] for extracting semantic information from binaries introduces another potential source of error. The inherent limitations of these tools likely contribute to false negatives in our analysis.

B. RQ2: Preparing Context

Table V shows the results of GLEIPNIR preparing the context. GLEIPNIR prepared the context for IPC calls at 2,834 different call sites identified through static analysis. These call sites targeted 537 unique remote methods. We attempted to trigger 1,686 IPC calls from BuiltIn applications and 1,148 IPC calls from Public APIs using methods such as user clicks, user input, and constraint solving. Additionally, for each CLI program, we utilized an average of 12.85 queries to the LLM. For testing Public APIs, which do not involve GUI, we focused on LLM-generated programs. Initially, LLM generated 78 programs. However, 16 of these programs proved non-executable due to complex parameter dependencies that remained unresolvable even after manual intervention attempts. This resulted in a final test set of 58 programs, which contained 145 APIs generated from 58 LLM queries. Overall, we successfully triggered 2,169 IPC requests, using 846 user clicks and 410 user inputs. Constraint solving was able to bring about an 11.28% improvement (from 1,949 (the fourth column) to 2,169 (the third column)). The time required to create fuzzing contexts was 8.53 hours, with UI testing taking 1.24 hours, and constraint solving consuming 2.10 hours.

TABLE V
RESULT OF PREPARE FUZZING CONTEXTS. RIC STANDS FOR RECOVERED IPC CALL, TIC STANDS FOR TRIGGERED IPC CALL, AND TW/O STANDS FOR TRIGGERED WITHOUT CONSTRAINTS RESOLVE.

ID	Application	RIC	TIC	Tw/oCS	Events	
					User Click	User Input
01	Performance Monitor	78	72	66	36	1
02	Eventlog Viewer	102	88	82	76	2
03	Device Manager	32	26	24	27	1
04	Windows Server Backup	60	45	42	32	1
05	Windows Disk Management	86	70	65	26	2
06	Service Management	70	59	52	68	1
07	Routing and Remote Access	116	92	86	77	6
08	Task Scheduler	80	64	59	84	12
09	Shared Folders	48	37	37	52	3
10	File Server Resource Manager	73	56	54	49	14
11	DFS Management	56	40	33	39	4
12	Group Policy Management	84	54	50	96	22
13	dfs replication	54	42	41	0	24
14	wmic	92	76	64	0	66
15	nfsadmin	60	44	40	0	32
16	mount	62	45	38	0	28
17	ftp	42	32	29	0	42
18	Windows Admin Center	491	365	300	184	77
	Builtin Total	1,686	1,307	1,162	846	338
	Public APIs	1,148	862	787	0	58
	Total	2,834	2,169	1,949	846	410

Furthermore, We manually analyzed the reasons why some IPC calls are failed to trigger :

- 350 IPC calls are located within exception handling logic, which is only invoked when an error or exception occurs in the previous IPC request. Since we did not insert steps to trigger exceptions during the server runtime, these IPC calls could not be triggered.
- 186 IPC calls fail to trigger due to the complexity of constraint solving. The complexity of constraints or the need to unroll loops more than 10 times prevented determining the necessary return values for reaching a path containing the IPC call.
- 129 IPC calls fail to trigger when the built-in GUI program requires user input other than the IP address. This user input must meet complex format requirements.

C. RQ3: Detecting vulnerabilities

Based on the prepared context, GLEIPNIR performed fuzz testing on the actual IPC calls that could be triggered. The results of detecting vulnerabilities are shown in Table VI. GLEIPNIR successfully discovered 25 vulnerabilities in 7 built-in applications and 8 Public APIs (column 2). The third column lists the functions containing IPC calls that led to vulnerabilities in the built-in applications, while the fourth column shows the Windows versions.

Currently, 14 vulnerabilities have received CVE numbers (column 5), and 19 vulnerabilities have been confirmed by Microsoft. These 25 vulnerabilities led to memory corruption, 20 of them could lead to remote code execution, and 5 could result in information leakage (column 6).

1) Security Impact: Currently, 14 vulnerabilities have received CVE numbers, and a total of 19 vulnerabilities have been confirmed by Microsoft. These 25 vulnerabilities led to memory corruption, and 20 of them could lead to remote code execution, and 5 could result in information leakage.

The cause of sensitive information leakage is due to a vulnerability modifying the address of the next IPC call's sent

TABLE VI
LIST OF VULNERABILITIES DISCOVERED BY GLEIPNIR (CONFIRMED BY MSRC).

ID	IPC Client Name	Function Name	Windows Version	Status	Security Impact
1	Routing and Remote Access	DeleteProtocolFromRouterConfig	Windows 11 Insider Build 26063.1	CVE-2024-30014	Remote code execution
2	Routing and Remote Access	CDhcpRelayComponent::QueryDataObject	Windows 11 Insider Build 26063.1	CVE-2024-30015	Remote code execution
3	Routing and Remote Access	TFSComponent::Construct	Windows 11 Insider Build 26063.1	CVE-2024-30022	Remote code execution
4	Windows Performance Monitor	GetSystemPerfData	Windows 11 Insider Build 26040.1	CVE-2024-38019	Remote code execution
5	Windows Performance Monitor	PERF_MACHINE::BuildNameTable	Windows 11 Insider Build 26040.1	CVE-2024-38025	Remote code execution
6	Windows Performance Monitor	UpdateMultiCounterV2CounterValue	Windows 11 Insider Build 26040.1	CVE-2024-38028	Remote code execution
7	Windows Performance Monitor	CollectServerQueueObjectData	Windows 11 Insider Build 26040.1	confirmed	Remote code execution
8	Windows Performance Monitor	PerflibV2QueryCounterData	Windows 11 Insider Build 26040.1	confirmed	Remote code execution
9	Windows Eventlog Viewer	Event::SetData	Windows 11 Insider Build 26040.1	confirmed	Information Disclosure
10	Windows Eventlog Viewer	Event::SetDataEx	Windows 11 Insider Build 26040.1	confirmed	Information Disclosure
11	Windows Eventlog Viewer	Event::ProcessData	Windows 11 Insider Build 26040.1	pending	Information Disclosure
12	Windows Admin Center	ApplicationServer	Windows 11 Insider Build 26040.1	confirmed	Remote code execution
13	Windows Admin Center	CategorySample	Windows 11 Insider Build 26040.1	CVE-2024-43475	Information Disclosure
14	Windows Disk Management	ActivationUser	Windows 11 Insider Build 26040.1	pending	Information Disclosure
15	Windows Disk Management	AppExtension	Windows 11 Insider Build 26040.1	pending	Remote code execution
16	Windows Task scheduler	TaskSchedulerProcess	Windows 11 Insider Build 26040.1	pending	Remote code execution
17	Windows DFS Replicator	BundlePackage	Windows 11 Insider Build 26040.1	pending	Remote code execution
18	PublicAPI	MSMQManagement.BytesInQueue	Windows 11 Insider Build 26040.1	CVE-2024-20680	Remote code execution
19	PublicAPI	CollectDiskObjectData	Windows 11 Insider Build 26040.1	pending	Remote code execution
20	PublicAPI	MprAdminPortEnum	Windows 11 Insider Build 26080.1	CVE-2024-38114	Remote code execution
21	PublicAPI	MprAdminConnectionEnum	Windows 11 Insider Build 26080.1	CVE-2024-38115	Remote code execution
22	PublicAPI	MprAdminDeviceEnum	Windows 11 Insider Build 26080.1	CVE-2024-38116	Remote code execution
23	PublicAPI	MprConfigTransportEnum	Windows 11 Insider Build 26080.1	CVE-2024-30023	Remote code execution
24	PublicAPI	MprAdminInterfaceEnum	Windows 11 Insider Build 26080.1	CVE-2024-30024	Remote code execution
25	PublicAPI	MprConfigInterfaceEnum	Windows 11 Insider Build 26080.1	CVE-2024-30029	Remote code execution

```
1  System.Diagnostics.CategorySample.CategorySample(...){
2      ...
3      long value = GetCategorySample(...); // First RPC Query
4      IntPtr intPtr = new IntPtr(value); // value is
       ↪  controlled data from remote IPC server
5      Marshal.PtrToStructure(intPtr, perf_DATA_BLOCK); //
       ↪  Interpreted as C# Structure
6      ref perf_INSTANCE_DEFINITION perfInstance =
       ↪  MemoryMarshal.AsRef(perf_DATA_BLOCK.member1);
7      ...
8      array3 = this.GetInstanceNamesFromIndex(
9      perfInstance.ParentObjectTitleIndex); // second RPC
       ↪  Query, leak triggered
10 }
```

Fig. 10. CVE-2024-43475: A vulnerability causing information disclosure in Windows Admin Center.

data points to the memory location containing sensitive information. Figure 10 gives such a case in Windows Admin Center, a contemporary system administration tool primarily developed using C#. It is widely believed that C# is a memory-safe language, and thus, programs written in C# should not introduce memory corruption causing information leakage. However, GLEIPNIR has breached this assumption through rigorous fuzzing. In line 3, variable `value` receives a value controlled by the malicious IPC server, the `value` is then used to create a structure in C# language context: `perf_DATA_BLOCK`. Then `perf_DATA_BLOCK` member is used to create another C# structure `perf_INSTANCE_DEFINITION`, and the structure is then used in line 9, where part of its memory will be sent to the malicious server through the second RPC call. The server can force variable `perfInstance` pointing at arbitrary memory locations in the process of Windows Admin Center via `value`. If `perfInstance` points to sensitive memory sections, the data in that area will be exposed to the remote malicious server, causing serious information leakage.

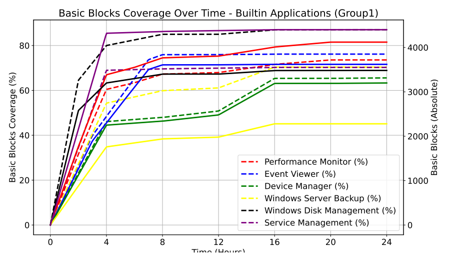
2) Responsible Disclosure : As Client vulnerabilities can lead to severe consequences, we took the responsibility of disclosing all the 25 vulnerabilities we identified in 76 clients(18 built-in applications and 58 programs generated by GPT-4 for public APIs) with a detailed report and Proof of Concept (PoC) via MSRC [10]. As per responsible disclosure practices, we will not publicly release any unfixed vulnerabilities until the developers address them. It is worth noting that all vulnerabilities with detailed information in our paper have been fixed by developers. At present, 19 of the identified vulnerabilities have been either confirmed or fixed, and 14 CVE identifiers have been assigned to these issues.

D. RQ4: Efficiency

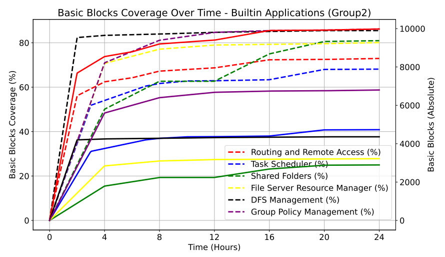
Figure 11 illustrates the testing efficiency of GLEIPNIR across 18 built-in applications. As shown in the figure, within four hours, the test coverage increased rapidly, with four applications reaching a stable coverage rate. Within 24 hours, the coverage rate for all applications stabilized, and 14 vulnerabilities were detected during this period. Over the subsequent six days of testing, the coverage rate for 7 applications increased slowly due to the triggering of deeper code layers, resulting in the detection of 3 additional vulnerabilities.

E. RQ5: Comparison

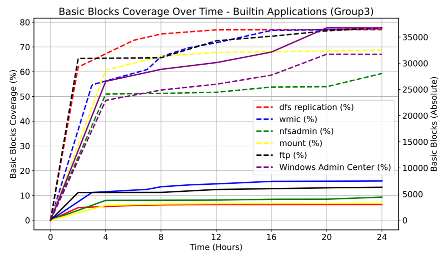
We compared GLEIPNIR with two popular Windows platform fuzzing tools, WinAFL [37] and WINNIE [38]. WinAFL is a Windows port of AFL, and WINNIE adds fork mode support to WinAFL. We further modified these tools to test server-side return values by replacing the original return values as mutated values. The server then returns these mutated values to the client. Our testing targets exclude built-in Apps, as these tools cannot create harnesses for them and thus cannot be tested. For Public APIs, GLEIPNIR has generated harnesses to drive IPC call execution. We selected a specific number of public APIs known to have vulnerabilities and compared the performance of the three tools over a 7-day



(a) Performance of fuzzing app-group1



(b) Performance of fuzzing app-group2



(c) Performance of fuzzing app-group3

Fig. 11. Performance of testing built-in applications. The left y-axis represents the percentage of basic block coverage relative to IPC-related code, while the right y-axis displays the absolute values of covered basic blocks. Absolute values are shown with solid lines, and percentages are depicted with dashed lines.

period. Each fuzzer was fixed to a single core, utilizing the same harness programs. For statistical robustness, we repeated the entire fuzzing process five times for each tool and used the average values in our final results. We also considered comparing GLEIPNIR with Winfuzz-Winnie [39]. However, Winfuzz-Winnie merely redirects socket traffic to file inputs and cannot handle complex network behaviors such as IPC, resulting in malfunctions across all our test subjects.

Table VII presents the experimental results. We observed that the execution speed(i.e., tested mutations per second) and coverage of WinAFL and WINNIE were significantly lower(achieved speeds of 2.88 and 3.24 executions per second, alongside coverage of 289.7 and 323.5 new basic blocks (BBs)) than those(with a speed of 234.5 executions per second and a coverage of 1,718 new BBs) of GLEIPNIR. Our analysis revealed that network request speed was the primary performance bottleneck for these tools. Our snapshot design mitigated network-related issues, improving execution speed and coverage. All 10 test cases reached stable coverage within a week. Consequently, our tool detected 20/4 and 20/3 more bugs/vulnerabilities than WinAFL and WINNIE, respectively.

F. RQ6: Ablation study

We conducted ablation studies by removing or replacing different strategies of GLEIPNIR to evaluate their individual contributions. The experimental results are presented in Table VIII.

During GUI testing, we employed a depth-first strategy and IP-based input testing. Each of these strategies was individually replaced with a random strategy. The results showed that the random input strategy failed to trigger any IPC calls, leading to zero vulnerability detections. The random click strategy triggered 665 IPC calls and detected 8 vulnerabilities, which was 403 fewer IPC calls and 8 fewer vulnerabilities compared to our original strategy. For CLI applications, the random input strategy was unable to trigger any IPC calls or detect vulnerabilities.

Furthermore, the experimental results indicated that our constraint-solving strategy improved IPC call triggering and vulnerability detection across all testing scenarios. In GUI testing, it increased IPC calls from 950 to 1,068 (+12.4%) and vulnerabilities from 13 to 16 (+23.1%). In CLI testing, it enhanced IPC calls from 220 to 239 (+8.6%) and doubled the vulnerabilities from 2 to 4 (+100%). In public API testing, it increased IPC calls from 750 to 862 (+14.9%) and raised vulnerability detection from 32 to 40 (+25%).

Table VIII also reveals that without enabling the snapshot feature, our testing efficiency was significantly reduced, achieving only 5.3 executions per second (2.2% of our tool's capacity with Stop-1000) and detecting only 11 vulnerabilities over 7 days, missing 14 potential vulnerabilities. Additionally, when we set the dirty page threshold to 2,000 and 3,000, both the efficiency and effectiveness of vulnerability testing declined. Conversely, when the threshold was set to 500, testing efficiency improved to 282 executions per second, but 5 vulnerabilities were missed compared to our Stop-1000 strategy.

V. DISCUSSIONS

A. What is the likelihood of this type of vulnerability appearing on other platforms, such as Linux and IOT devices?

Our proposed threat model is not limited by platform and can be extended to software and devices across various operating systems, including Linux and Internet of Things (IoT) devices. For instance, Linux users often use remote management software like xrdp or VNC to access and manage potentially untrusted remote machines. Complex IoT devices frequently incorporate various software applications to monitor device operational status and network traffic, mirroring the functionality of the Windows Performance Monitor. For example, we identified a vulnerability in the widely-used ThingsBoard IoT platform [40]. This vulnerability occurs when a monitored device transmits a large packet via RPC, triggering an Out-of-Memory error and resulting in a DoS. We have reported this vulnerability, and it has been assigned CVE-2024-9358. Hence such software and IoT devices could also become targets for malicious attackers.

While Windows offers a broader variety of IPC channels, vulnerabilities of the kind discussed directly impact the security of both domains and personal computers, thus drawing our primary focus. It is crucial to acknowledge that while the threat model is universally relevant, its specific expressions and consequences may differ across platforms and device types. For instance, software providing performance monitoring for IoT devices might be implemented as web services or using

TABLE VII
COMPARISON OF GLEIPNIR SNAPSHOT FUZZING AGAINST WINAFL AND WINNIE. FOR "SPEED" AND "COVERAGE," THE LAST ROW IN THE TABLE INDICATES THE AVERAGE VALUES ACROSS ALL APPLICATIONS. FOR "BUGS/VULNERABILITIES FOUND," THE LAST ROW DENOTES THE TOTAL NUMBER ACROSS ALL APPLICATIONS.

Application	Speed(exec/sec)			Coverage(# of new BBs)			Bug/Vuln Found		
	WinAFL	WINNIE	GLEIPNIR	WinAFL	WINNIE	GLEIPNIR	WinAFL	WINNIE	GLEIPNIR
RemoteQMStartReceive2	1.8	2.2	238	320	365	1,026	1/0	1/1	3/1
QuerySnapshotsByVolume	2.1	2.3	215	336	352	2,072	2/0	2/0	5/2
QMMgmgGetInfo	1.8	2.4	176	291	310	1,872	2/1	2/1	6/2
CollectDiskObjectData	3.5	3.6	252	190	252	2,417	1/1	2/1	4/1
MprAdminPortEnum	3.2	3.1	222	224	265	1,644	3/1	3/1	5/1
MprAdminConnectionEnum	3.5	4.2	275	230	280	1,571	1/0	2/0	3/0
MprAdminDeviceEnum	3.1	3.8	185	298	365	1,820	1/0	1/0	2/0
MprConfigTransportEnum	3.0	3.3	218	318	327	1,440	4/1	2/1	5/1
MprAdminInterfaceEnum	3.5	3.6	289	320	330	1,520	2/0	2/0	3/0
MprConfigInterfaceEnum	3.3	3.9	275	370	389	1,798	3/0	3/0	4/0
Average/Total	2.88	3.24	234.5	289.7	323.5	1,718	20/4	20/5	40/8

TABLE VIII
TESTING STRATEGIES AND THEIR RESULTS. CS STANDS FOR CONSTRAINT SOLVING, WHILE PUB REPRESENTS PUBLIC API.

Strategies	**TIPC**	**Vulns**	**AvgSpeed (exec/sec)**
GUI-RandomInput	0	0	-
GUI-RandomClick	665	8	-
GUI-W/O CS	950	13	-
GUI-GLEIPNIR	1068	16	-
CLI-RandomInput	0	0	-
CLI-W/O CS	220	2	-
CLI-GLEIPNIR	239	4	-
PUB-W/O CS	750	32	-
PUB-GLEIPNIR	862	40	
W/O Snapshot	-	11	5.3
Stop-500	-	20	282
Stop-1000	-	25	240
Stop-2000	-	22	186
Stop-3000	-	16	120

scripting languages. As a result, the likelihood of memory corruption vulnerabilities within such software is comparatively lower. However, these implementations may still be vulnerable to issues such as cross-site scripting and SQL injection. The security risks posed by these vulnerabilities remain consistent with our proposed threat model.

B. What are the limitations of GLEIPNIR*?*

GLEIPNIR utilizes static analysis algorithms to identify clients and servers but does not guarantee soundness. Some clients do not match with corresponding servers, which does not imply that these clients are free from the threat model discussed in this paper. On the contrary, due to their inability to be successfully matched, they are more likely to have untested and potentially more severe security issues. Our tool does not support the automated creation of servers for testing in such cases. Additionally, some IPC calls within client internals are not fully triggered, and GLEIPNIR cannot detect vulnerabilities caused by these IPC calls. The fuzzing methods currently employed by GLEIPNIR do not support all vulnerabilities in client software, such as those related to access control and credential handling.

C. Where can we make improvements to GLEIPNIR

1) More platforms: To migrate GLEIPNIR to other platforms, such as Linux and IoT platforms, we need to undertake the following efforts across three phases:

- In the first phase, our bottom-up client identification algorithm is cross-platform. However, we still need to manually specify IPC APIs on other platforms and analyze the code characteristics of these IPCs to identify the SIDs and MIDs.
- In the second phase, considerable effort is required to adapt the UI operations for different platforms. For instance, while we currently use Microsoft UI Automation on Windows, we need to replace it with the Linux Desktop Testing Project on Linux systems. Other techniques in this phase, such as LLM-powered automated testing for CLI programs, are cross-platform compatible.
- In the third phase, our snapshot-based approach can be migrated to other platforms. For snapshot-based fuzzing techniques, we can use technologies from other platforms, such as nyt-net on Linux. We can apply our IPC emulation and dirty-page-based adaptive termination techniques to this technology for acceleration, but this requires some effort. Additionally, the exception monitor requires significant modifications to identify vulnerabilities based on the exception handling mechanisms of other platforms.

2) More Vulnerability patterns: Client applications are diverse. For example, an IoT monitoring platform essentially functions as a web server. Consequently, it is susceptible to common vulnerability patterns such as file inclusion, injection, Server-Side Request Forgery (SSRF), and others. Currently, extensive work has been done to fuzz these types of vulnerabilities. For instance, SSRFuzz [41] focuses on fuzzing SSRF vulnerabilities, while Atropos [42] targets SQL injection and file inclusion vulnerabilities. GLEIPNIR can integrate these fuzzing tools into its internal fuzzing processes to detect other common types of vulnerabilities effectively.

3) More Servers: Due to our tool's limitations, we were unable to identify corresponding servers for all clients. To test clients lacking servers, a feasible approach is to mock a server that provides the APIs requested by the client. The specific functionalities of these APIs may need to be manually analyzed based on the context of the client's calls and then implemented in the API code.

4) More Triggered IPC Calls: As discussed in Section IV, there are three reasons why IPC calls may fail to trigger. For exceptions, instrumentation techniques can be used to insert exceptions into the program, thus triggering the execution of exception handling code and the IPC requests located within that logic. For constraint solving, the loop threshold in constraint solving can be increased, although this might reduce runtime efficiency. For complex user inputs, integrating the user input part of GUI testing with an LLM can enhance the validity of user inputs.

VI. Related Work

Fuzz. In recent years, fuzzing [43], [38], [37] has become very popular due to its efficiency in discovering vulnerabilities. Various fuzzing tools have been developed for different testing targets, such as operating systems [44], [45], virtual machines [46], [33], [47], [48], and web applications [42], [49], [41]. Among them, several fuzzing techniques for IPC server-side [2], [18], [3], [1] have been proposed to detect server-side issues like data race vulnerabilities. While some researches [20], [21], [18] have attempted to apply fuzzing techniques to client-side applications, these approaches often oversimplify the problem by treating clients as servers and directly applying server fuzzing technology. This paper highlights a key difference between client and server fuzzing: the importance of context in client fuzzing. To address this gap, we introduce Gleipnir, a novel fuzzing tool designed specifically for client-side applications. Our approach excels in context building and leverages enhanced snapshot-based technology to achieve efficient client fuzzing.

Snapshot. Snapshot technology provides a restore point for fuzz testing, allowing the testing process to quickly revert to a previous state, thereby avoiding the overhead of restarting the program each time. Snapshot-based fuzzing is widely applied in scenarios where harness construction is challenging, such as operating system kernel testing [44], [39], [50], virtual machine testing [33] and Windows binaries testing [39], [51]. In this paper, we introduce snapshot technology for preserving execution contexts of IPC calls, complemented by adaptive termination and direct IPC API simulation, to achieve efficient and accelerated client fuzzing.

Binary Analysis. Binary analysis serves as the foundational approach for examining Windows and other closed-source systems. Numerous binary analysis techniques have been proposed and developed in recent years [52], [53], [54], [55], [56]. BAP [52] and BinCAT [53] are binary analysis platforms providing fundamental support for static analysis on binary files. Angr [56], an advanced binary analysis framework, extends the capabilities of such platforms by incorporating sophisticated features such as control-flow analysis, data-dependency analysis, and symbolic execution. Gleipnir leverages these tools to identify IPC clients and servers and employs data flow analysis techniques from these works to aid in constructing the fuzzing context.

Harness. Harness generation has become a very popular topic recently. Tools like WINNIE [38], [57], [58], [59], [60] have introduced powerful harness generation techniques to bypass GUI restrictions and directly test critical code. The implementation of these techniques often requires the testing target to meet certain conditions, such as having critical code accessible via public APIs. Additionally, in particularly complex targets, harness generation can significantly reduce the cost of manual construction. In our scenario, based on snapshot technology, we transform the harness generation task into preparing the testing context for snapshots, achieving the goal of testing critical code without the need to bypass the GUI, thus providing a more realistic testing environment.

LLM powered test. Considering the powerful performance of LLM, researchers have successfully leveraged it for software testing. OSS-Fuzz has attempted to use LLM for completely automatic fuzz target generation [61]. TitanFuzz [62] has used LLM to generate input programs for fuzzing deep learning libraries. QTypist [63] leveraged LLM to generate text inputs for passing a GUI page, aiming to improve the testing coverage of mobile testing. GPTDroid [64] asks LLM to propose different actions to interact with the target app. Unlike these works, this paper utilizes LLM to explore the execution parameters of CLI programs and generate programs to drive the execution of public APIs, thereby preparing the fuzzing context to test the return values of an IPC server.

VII. Conclusion

This paper introduces Gleipnir, a novle vulnerability detection tool specifically designed for Windows remote IPC clients. Unlike previous research that primarily targets server-side vulnerabilities, Gleipnir addresses the critical and often overlooked client-side vulnerabilities. By fuzzing the return values of IPC calls and leveraging snapshot technology to enhance testing efficiency, Gleipnir effectively identifies vulnerabilities in client applications. Our experiments on 76 client applications revealed 25 vulnerabilities within 7 days, resulting in 14 CVEs and a total bounty of $36,000.

Acknowledgement

We thank all reviewers for their valuable feedback. This work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing Key Laboratory of Network Security and Protection Technology, the National Natural Science Foundation of China (62202452).

References

[1] 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

[2] K. Yang, H. Zhao, C. Zhang, J. Zhuge, and H. Duan, "Fuzzing ipc with knowledge inference," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 11–1109.

[3] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 307–323. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/liu

[4] H. Feng and K. G. Shin, "Understanding and defending the binder attack surface in android," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 398–409.

[5] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013, pp. 68–74.

[6] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1165–1178.

[7] G. Gong, "Fuzzing android system services by binder call to escalate privilege," *BlackHat USA*, vol. 2015, 2015.

[8] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 361–370.

[9] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework." in *NDSS*, 2016.

[10] MSRC, *Microsoft Security Response Center*. [Online]. Available: https://msrc.microsoft.com

[11] ITarian, *Remote Management Benefits to IT Professionals*. [Online]. Available: https://www.itarian.com/what-is-remote-management/#:~:text=Remote%20management%20is%20an%20innovative,and%20networks%20from%20any%20location.

[12] MSDN, *Windows Performance Monitor Overview*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/ui-automation/ui-automation-overview

[13] MicroSoft, *Remote Desktop Services overview in Windows Server*. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/remote/remote-desktop-services/remote-desktop-services-overview

[14] MicroSoft 2024, *Windows Admin Center*. [Online]. Available: https://www.microsoft.com/en-us/windows-server/windows-admin-center

[15] MSDN, *Overview of Disk Management*. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/storage/disk-management/overview-of-disk-management

[16] TeamViewer GmbH, "Teamviewer." [Online]. Available: https://www.teamviewer.com/

[17] VMware, Inc., "Vmware vcenter." [Online]. Available: https://www.vmware.com/products/vcenter-server.html

[18] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 166–180. [Online]. Available: https://doi.org/10.1145/3492321.3519591

[19] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[20] N. Bars, M. Schloegel, N. Schiller, L. Bernhard, and T. Holz, "No peer, no cry: Network application fuzzing via fault injection," *arXiv preprint arXiv:2409.01059*, 2024.

[21] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4481–4498.

[22] MicroSoft, "Remote procedure call (rpc)," 2022, mSDN document for RPC. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/Rpc/rpc-start-page

[23] Microsoft, "Component object model (com)," 2020, cOM MSDN. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal

[24] C. Yu, Y. Xiao, J. Lu, Y. Li, Y. Li, L. Li, Y. Dong, J. Wang, J. Shi, D. Bo *et al.*, "File hijacking vulnerability: The elephant in the room," in *Proceedings of the Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society*, 2024.

[25] A. Lee, I. Ariq, Y. Kim, and M. Kim, "Power: Program option-aware fuzzer for high bug detection ability," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 220–231.

[26] MSDN, *Win32 interprocess-communications*. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/ipc/interprocess-communications/

[27] cwe, *Trust Boundary Violation*. [Online]. Available: https://cwe.mitre.org/data/definitions/501.html

[28] J. Lu, H. Li, C. Liu, L. Li, and K. Cheng, "Detecting missing-permission-check vulnerabilities in distributed cloud systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2145–2158.

[29] lucasg, *findrpc : Ida script to extract RPC interface from binaries*. [Online]. Available: https://github.com/lucasg/findrpc

[30] J. Forshaw, *oleviewdotnet*. [Online]. Available: https://github.com/tyranid/oleviewdotnet

[31] Intel, *Intel® Processors Support*. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000056730/processors.html

[32] S. Bitchebe, D. Mvondo, A. Tchana, L. Réveillère, and N. De Palma, "Intel page modification logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation," *arXiv preprint arXiv:2001.09991*, 2020.

[33] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[34] MSDN, "https://learn.microsoft.com/en-us/windows/win32/debug/structured-exception-handling," 2021, structured Exception Handling.

[35] OpenAPI, *Batch API*. [Online]. Available: https://platform.openai.com/docs/guides/batch/overview

[36] Microsoft, *Microsoft Research Detours Package*. [Online]. Available: https://github.com/microsoft/Detours

[37] I. Fratric, *WinAFL*. [Online]. Available: https://github.com/googleprojectzero/winafl

[38] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie : Fuzzing windows applications with harness synthesis and fast cloning," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:231854623

[39] 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)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4913–4929. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/stone

[40] "Open-source iot platform," 2021, https://thingsboard.io/.

[41] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 216–216.

[42] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for server-side vulnerabilities," in *USENIX Security Symposium*, 2024.

[43] M. Zalewski, "american fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[44] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2541–2557. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/song

[45] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 818–834.

[46] Y. Liu, S. Chen, Y. Xie, Y. Wang, L. Chen, B. Wang, Y. Zeng, Z. Xue, and P. Su, "Vd-guard: Dma guided fuzzing for hypervisor virtual device," *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1676–1687, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265055952

[47] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, “Videzzo: Dependency-aware virtual device fuzzing,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 3228–3245.

[48] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, “V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing,” *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:244077725

[49] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, “Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2658–2675.

[50] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, “No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions,” *Proceedings 2023 Network and Distributed System Security Symposium*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257501158

[51] Axel Souchet, *what the fuzz*. [Online]. Available: https://github.com/0vercl0k/wtf

[52] *BAP*, https://github.com/BinaryAnalysisPlatform, 2021.

[53] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger, “Bincat bincat : purrfecting binary static analysis,” 2017.

[54] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” 2016.

[55] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware,” 2015.

[56] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” 2016.

[57] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, “Fudge: fuzz driver generation at scale,” *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:199501660

[58] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “Fuzzgen: Automatic fuzzer generation,” in *USENIX Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:207905588

[59] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, “Utopia: Automatic generation of fuzz driver using unit tests,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2676–2692.

[60] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, “Apicraft: Fuzz driver generation for closed-source sdk libraries,” in *USENIX Security Symposium*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:237522074

[61] OSS, *Fuzz target generation using LLMs*. [Online]. Available: https://google.github.io/oss-fuzz/research/llms/target_generation/

[62] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[63] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, “Fill in the blank: Context-aware automated text input generation for mobile gui testing,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1355–1367.

[64] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, “Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

APPENDIX

A. GPT Generated Testcase.

The complete code corresponding to Figure 6 is shown in Figure A, which was obtained by querying GPT-4 with the queries from Table III.

```cpp
#include <windows.h>
#include <mprapi.h>
#include <iostream>
#pragma comment(lib, "mprapi.lib")
int main() {
    HANDLE hMprServer = NULL;
    WCHAR serverName[] = L"192.168.1.5";
    DWORD dwError;
    std::cout << "Attempting to connect to remote server: " << "192.168.1.5" << std::endl;
    // Test Case 1: Connect to remote server
    dwError = MprAdminServerConnect(serverName, &hMprServer);
    if (dwError == NO_ERROR) {
        std::cout << "Successfully connected to remote server." << std::endl;
    } else {
        std::cerr << "Failed to connect to remote server. Error code: " << dwError << std::endl;
        return 1;
    }
    // Test Case 2: Get server information
    MPR_SERVER_0 serverInfo;
    dwError = MprAdminServerGetInfo(hMprServer, 0, (LPBYTE*)&serverInfo);
    if (dwError == NO_ERROR) {
        std::wcout << L"Server name: " << serverInfo.lpwsServerName << std::endl;
        std::wcout << L"Server version: " << serverInfo.dwServerVersion << std::endl;
    } else {
        std::cerr << "Failed to get server information. Error code: " << dwError << std::endl;
    }
    // Test Case 3: Enumerate interfaces
    LPBYTE pBuffer = NULL;
    DWORD dwEntriesRead = 0;
    DWORD dwTotalEntries = 0;
    dwError = MprAdminInterfaceEnum(hMprServer, 0, &pBuffer, (DWORD)-1, &dwEntriesRead, &dwTotalEntries, NULL);
    if (dwError == NO_ERROR) {
        std::cout << "Number of interfaces: " << dwEntriesRead << std::endl;
        MPR_INTERFACE_0* pInfo = (MPR_INTERFACE_0*)pBuffer;
        for (DWORD i = 0; i < dwEntriesRead; i++) {
            std::wcout << L"Interface name: " << pInfo[i].wszInterfaceName << std::endl;
        }
        MprAdminBufferFree(pBuffer);
    } else {
        std::cerr << "Failed to enumerate interfaces. Error code: " << dwError << std::endl;
    }
    // Test Case 4: Disconnect from remote server
    if (hMprServer != NULL) {
        MprAdminServerDisconnect(hMprServer);
        std::cout << "Disconnected from remote server." << std::endl;
    }
    return 0;
}
```

Fig. 12. The complete code in Figure 6.