AcidBox: Rare Malware Repurposing Turla Group Exploit Targeted Russian Organizations

By Dominik Reichel, Esmid Idrizovic

Published: 2020-06-17 · Archived: 2026-04-05 12:39:55 UTC

Executive Summary

When the news broke in 2014 about a new sophisticated threat actor dubbed the Turla Group, which the Estonian foreign intelligence service believes has Russian origins and operates on behalf of the FSB, its kernelmode malware also became the first publicly-described case that abused a third-party device driver to disable Driver Signature Enforcement (DSE). This security mechanism was introduced in Windows Vista to prevent unsigned drivers from loading into kernel space. Turla exploited the signed VirtualBox driver, VBoxDrv.sys v1.6.2, to deactivate DSE and load its unsigned payload drivers afterward.

There is some confusion about this exploit, however, as it's often generally referred to as CVE-2008-3431. The exploit used by Turla actually abuses two vulnerabilities -- of which, only one was ever fixed in the aforementioned CVE. The other vulnerability was found by Turla and is used in the first version of their exploit, along with CVE-2008-3431. The second version of their exploit, presumably introduced in 2014 of their kernelmode malware, only uses the unpatched vulnerability, which we discuss in greater detail later.

In February 2019, Unit 42 found that a yet-to-be-known threat actor -- unbeknownst to the infosec community -- discovered that the second unpatched vulnerability can not only exploit VirtualBox VBoxDrv.sys driver v1.6.2, but also all other versions up to v3.0.0. Furthermore, our research shows that this unknown actor exploited VirtualBox driver version 2.2.0 to target at least two different Russian organizations in 2017, which we are revealing for the first time. We anticipate this was done because the driver version 2.2.0 wasn't known to be vulnerable and thus most likely is not on the radar of security companies being exploited. Since no other victims have been found, we believe this is a very rare malware used in targeted attacks only.

The actors used a previously unknown malware family that we have dubbed AcidBox due to the first part being an anagram of the malware's driver device name and the second part taken from VirtualBox. Because of the malware's complexity, rarity, and the fact that it's part of a bigger toolset, we believe it was used by an advanced threat actor for targeted attacks and it's likely that this malware is still being used today if the attacker is still active. However, we anticipate that it was rewritten to a certain extent. Based on the information we have, we don't believe this unknown threat actor is tied to Turla, except for the used exploit.

Palo Alto Networks customers are protected from this threat. Our threat prevention platform with WildFire identifies this malware as malicious. AutoFocus customers can track malware activity by using the AcidBox tag. We also created an Adversary Playbook for this attack, which can be found here.

The Unknown Threat Actor

In February 2019, we discovered a sample of AcidBox (SHA256: eb30a1822bd6f503f8151cb04bfd315a62fa67dbfe1f573e6fcfd74636ecedd5) uploaded to VirusTotal, which contained a string known to be used in Turla's VirtualBox exploit. A deeper analysis of the sample revealed it's the main worker module as part of a sophisticated malware that we couldn't tie to any known threat actor.

In collaborating with our colleagues at Dr.Web, we learned that this sample was used in a targeted attack on an unspecified entity in Russia back in 2017. Thankfully, they shared three additional samples of the same malware family. Two of those usermode samples are modules that load the main worker from the Windows registry and one is the kernelmode payload driver embedded in the main worker sample. Moreover, we contacted Kaspersky since the company is headquartered in Russia, which found only one additional sample in their databases that was also the usermode loader version. We also contacted ESET, which didn't find any victims infected with this malware, just like in our own case. For this reason, we conclude it's a very rare malware used in targeted attacks only.

What all the samples have in common are the compilation timestamps of May 9, 2017. This date seems legitimate, as the sample found by Kaspersky appeared in June 2017 in their databases. Therefore, we conclude that the campaign which involved this malware took place in 2017. We couldn't find any newer samples and thus it's unknown if AcidBox is still in use or has been further developed.

We compared the specific characteristics of the samples to all publicly-known malware, but couldn't find any clear overlaps. There are some very loose similarities to Remsec malware attributed to ProjectSauron, like:

- DWORD size data marker values
- Export function names made of 2/3 words
- MS Visual C/C++ compiler used
- Various import API functions overlaps
- Use of vulnerable 3rd party driver to load own unsigned driver
- Zlib compression used
- Sensitive data encrypted in the resource section

However, based on these facts alone, it's not possible to attribute the samples to the ProjectSauron threat actor. We think it's more likely this is a yet unknown threat actor.

The VirtualBox Exploit and Turla's Versions

The original vulnerability described in CVE-2008-3431 was found by Core Security in 2008 and affected VBoxDrv.sys less or equal version 1.6.2. It was fixed in version 1.6.4 and thus couldn't be exploited anymore.

VirtualBox 1.6.2

```
1  __int64 __fastcall VBoxDrvNtDeviceControl(DEVICE_OBJECT *pDevObj, IRP *pIrp)
2  {
3    IRP *irp; // rdi
4    struct SUPDRVDEVEXT *pDevExt; // r10
5    _IO_STACK_LOCATION *pStack; // r9
6    __int64 pSession; // rdx
7    __int64 IoControlCode; // rax
8    unsigned __int8 currentIrql; // bl
9    int rc; // eax
10
11   irp = pIrp;
12   pDevExt = (struct SUPDRVDEVEXT *)pDevObj->DeviceExtension;
13   pStack = pIrp->Tail.Overlay.CurrentStackLocation;
14   pSession = (__int64)pStack->FileObject->FsContext;
15   IoControlCode = pStack->Parameters.DeviceIoControl.IoControlCode;
16   if ( (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_RAW_RUN
17     && (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_HM_RUN
18     && (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_NOP )
19   {
20     return VBoxDrvNtDeviceControlSlow(pDevObj->DeviceExtension, pSession, irp);
21   }
22   currentIrql = KeGetCurrentIrql();
23   __writecr8(2ui64);
24   rc = supdrvIOCtlFast(IoControlCode, pDevExt, pSession);
25   __writecr8(currentIrql);
26   irp->IoStatus.Status = 0;
27   irp->IoStatus.Information = 4i64;
28   *(_DWORD *)irp->UserBuffer = rc;
29   IofCompleteRequest(irp, 0);
30   return 0i64;
31 }
```

VirtualBox 1.6.4

```
1  __int64 __fastcall VBoxDrvNtDeviceControl(DEVICE_OBJECT *pDevObj, IRP *pIrp)
2  {
3    _IO_STACK_LOCATION *pStack; // r9
4    PVOID pDevExt; // r10
5    IRP *irp; // rdi
6    __int64 pSession; // rdx
7    __int64 IoControlCode; // rax
8    unsigned __int8 currentIrql; // bl
9    int rc; // eax
10   unsigned int v10; // ebx
11
12   pStack = pIrp->Tail.Overlay.CurrentStackLocation;
13   pDevExt = pDevObj->DeviceExtension;
14   irp = pIrp;
15   pSession = (__int64)pStack->FileObject->FsContext;
16   IoControlCode = pStack->Parameters.DeviceIoControl.IoControlCode;
17   if ( (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_RAW_RUN
18     && (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_HM_RUN
19     && (_DWORD)IoControlCode != SUP_IOCTL_FAST_DO_NOP )
20   {
21     return ((__int64 (__fastcall *)(PVOID, __int64, IRP *))VBoxDrvNtDeviceControlSlow)(
22               pDevObj->DeviceExtension,
23               pSession,
24               irp);
25   }
26   currentIrql = KeGetCurrentIrql();
27   __writecr8(2ui64);
28   rc = supdrvIOCtlFast(IoControlCode, (__int64)pDevExt, pSession);
29   __writecr8(currentIrql);
30   v10 = 0xC000000D;
31   if ( rc >= 0 )
32     v10 = 0;
33   irp->IoStatus.Status = v10;
34   IofCompleteRequest(irp, 0);
35   return v10;
36 }
```

Figure 1. Vulnerable and patched VirtualBox device dispatch handler routines on the left and right, respectively

The vulnerability is located in the dispatch device control routine called VBoxDrvNtDeviceControl. On versions prior to 1.6.4, you can call the usermode DeviceIoControl API function and send one of the following control codes together with a kernel address you want to overwrite as the input/output buffer:

- SUP_IOCTL_FAST_DO_RAW_RUN
- SUP_IOCTL_FAST_DO_HM_RUN
- SUP_IOCTL_FAST_DO_NOP

The kernel address is passed down to the control handler (see Figure 1 left, line 28) without any check or validation and is filled with the return value of supdrvIOCtlFast (see Figure 1 left, line 24). This is where the vulnerability digging from CoreSecurity stops and where Turla continues. In the original exploit, the return value from supdrvIOCtlFast isn't controlled and thus it will be a random value written to your kernel address. Turla's exploit controls the return value by overwriting the function pointer of supdrvIOCtlFast to redirect execution to a small shellcode, which returns the needed value. This was described in great detail in a couple of articles and the complete reverse-engineered exploit code is also available.

The patched version 1.6.4 (see Figure 1 right) doesn't use the UserBuffer pointer anymore, which could be abused by passing a kernel address. Additionally, it checks if the rc variable is equal or bigger than zero -- which isn't needed for the patch, but more like a sanity check.

With this patch, the original vulnerability to overwrite a kernel address was fixed. The other vulnerability that lets you control the function pointer of supdrvIOCtlFast remained unpatched. Of course, that's because it wasn't discovered by Core Security at the time, but only a few years later by the Turla threat actor.

While Turla still uses the vulnerable VirtualBox driver v.1.6.2 to date, it only makes use of the unpatched vulnerability. The reason why and how it uses it was described by Lastline, and also the reverse-engineered exploit code is available in a project named Turla Driver Loader.

The secret is the exact same exploit, with only a small modification -- which we won't disclose here -- can be used on all VBoxDrv.sys versions up to 3.0.0. This was also figured out by an unknown threat actor. While VirtualBox versions smaller than 4.0 aren't available on the official website anymore, they can still be found on some software download sites.

Starting from version 3.0.0, some structures and routines have changed so the exploit does not work anymore. However, it can't be ruled out that in later versions it's still possible to exploit the same vulnerability with some more adjustments.

What's also interesting is that not even the Turla authors themself seem to have realized that. They still use the old VBoxDrv.sys v.1.6.2 in their otherwise stealthy exploit. This driver is widely known to be used for malicious or various otherwise dubious purposes, for example, in-game cheats.

Technical Analysis of AcidBox

The malware is a complex modular toolkit of which we have only a part of it. In total, we have found four 64-bit usermode DLLs and an unsigned kernelmode driver (SHA256: 3ef071e0327e7014dd374d96bed023e6c434df6f98cce88a1e7335a667f6749d). Three out of the four usermode samples have identical functionality and are loaders for the main worker module. They only differ in their file descriptions and the embedded and encrypted registry path. These loaders are created as security support providers (further SSP). A SSP is a DLL that exports at least the function SpLsaModeInitialize and usually provides security mechanisms such as authentication between client/server applications. There are a couple of standard SSPs provided in Windows such as Kerberos (kerberos.dll) or NTLM (msv1_0.dll). You can abuse the SSP interface for malware persistency and also for injection purposes. In order to maintain persistency, you have to put your SSP DLL into the Windows system directory and add the name of your DLL to a certain registry value. Upon a system restart, your DLL gets loaded into the Windows lsass.exe process and is executed. If you just want your SSP DLL to be injected into lsass.exe, you can call the API function AddSecurityPackage which triggers immediate loading. Of course, you need at least admin privileges for both of these methods. The first case of a malware abusing the SSP interface was mentioned by Matt Graeber in 2014. Since then, this persistence and injection trick has become wider known, but it's still rarely used in malware.

In case of the three AcidBox SSP DLLs, they don't make use of any security related operations, but purely abuse this interface for injection purposes and most likely also for persistence. The three SSPs have different file names that are similar to the standard packages provided in Windows (msv1_0.dll, pku2u.dll, wdigest.dll):

- msv1_1.dll (SHA256: b3166c417d49e94f3d9eab9b1e8ab853b58ba59f734f774b5de75ee631a9b66d)
- pku.dll (SHA256: 3ad20ca49a979e5ea4a5e154962e7caff17e4ca4f00bec7f3ab89275fcc8f58c)
- windigest.dll (SHA256: 003669761229d3e1db0f5a5b333ef62b3dffcc8e27c821ce9018362e0a2df7e9)

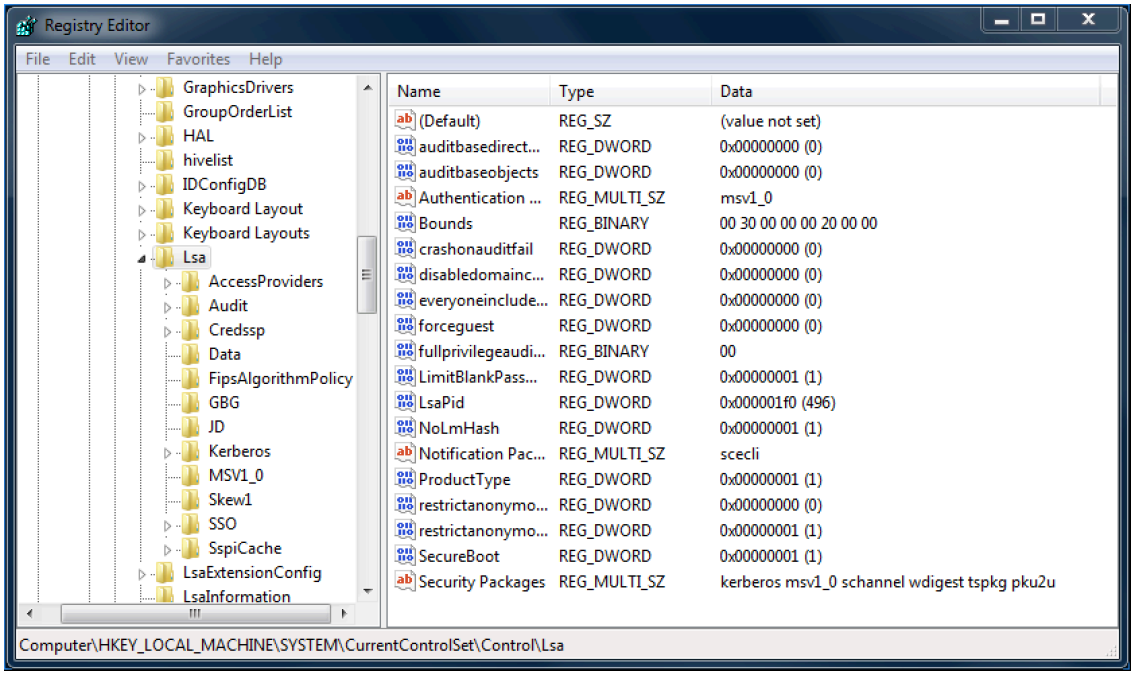

Figure 2. Standard SSP DLLs provided in Windows 7 present in the “Security Packages” value

For this reason, we conclude the AcidBox SSPs also abuse the interface for persistence. However, as we don’t have the component which installs the SSP DLLs, we don’t know for sure. What we know is that the SSP interface is used for injection into lsass.exe, as they check at the beginning whether the process path they're loaded into matches the one which is embedded into every sample in the resource section (C:\WINDOWS\SYSTEM32\lsass.exe). This process path is contained in the resource 4097 which we describe later how it is hidden via steganography.

The purpose of the AcidBox SSP DLLs is to load the main worker module from a registry value contained in resource 256 of each sample. We don’t know how the main worker DLL was stored in the registry, but we believe it was done by the same missing component which installed the SSP DLLs. We also assume that the three SSP DLLs are from three different systems as one of those samples has a different registry key embedded. Also, as these modules are the only visible part on a system -- with the main worker module being loaded remains encrypted in the registry -- they likely differ in some way like their chosen file name. The main worker is stored in the registry encrypted within a data blob that contains various other metadata such as the CRC32 hash of the data blob or magic byte sequences which indicate different types of contained data.

After the main worker DLL gets decrypted by a SSP DLL from the registry via simply XORing the data with key 0xCA, it gets prepared to be loaded from memory. It does so by creating a thread for the module and uses the exported function UpdateContext of the main worker as its start address. The main worker module then loads the unsigned malware driver via the VirtualBox exploit and waits for commands from one or more components that we don’t have. These commands include the loading of additional payloads from the registry from kernel space via the driver or the installation of new SSP DLLs.

The main worker has 2 export functions named InitMainStartup and UpdateContext. The following strings are present in cleartext:

1	%s\%s
2	%s\%s{%s}
3	%s\[[%s]]
4	%s.dll
5	%s%s%s.dll
6	\\.\PCIXA_CFGDEV
7	InitEntry
8	InitExit
9	The Magic Word!
10	ntoskrnl.exe
11	ntkrn
12	ntkrp
13	hal.dll
14	ntoskrnl
15	ntkrnlpa.exe
16	%s%s%s
17	Group
18	Count
19	NextInstance
20	Root\LEGACY_NULL\0000
21	
22	
23	
24	
25	
26	

The following additional strings are stack obfuscated:

1	
2	SeLoadDriverPrivilege
3	%s\%c*.dll
4	System\CurrentControlSet\Control\
5	NtQueryInformationThread
6	BFE_Notify_Event_
7	Microsoft\Cryptography
8	ntdll.dll
9	\Registry\Machine\
10	SOFTWARE
11	Global
12	%s\%s
13	Security Packages
14	kernel32.dll
15	SOFTWARE
16	\Registry\Machine\
17	MachineGuid
18	ntdll.dll
19	

There are also XOR-encoded DLL and function names, which later get dynamically resolved and used:

1	ntdll.RtlGetVersion
2	ntdll.ZwLoadDriver
3	ntdll.ZwUnloadDriver

4	ntdll.ZwQuerySystemInformation
5	kernel32.DeviceIoControl
6	kernel32.GetSystemDirectoryA
7	ntdll.RtlAnsiStringToUnicodeString
8	ntdll.ZwClose
9	ntdll.ZwCreateFile
10	ntdll.ZwQueryInformationFile
11	ntdll.ZwReadFile
12	ntdll.ZwWriteFile
13	kernel32.GetSystemDirectoryA
14	kernel32.GetSystemDirectoryW
15	kernel32.BaseThreadInitThunk
16	kernel32.LZDone
17	advapi32.CryptAcquireContextA
18	advapi32.CryptGenRandom
19	advapi32.CryptReleaseContext
20	ntdll.RtlRbInsertNodeEx
21	ntdll.RtlRbRemoveNode
22	ntdll.RtlAcquireSRWLockExclusive
23	ntdll.RtlReleaseSRWLockExclusive
24	ntdll.RtlEnterCriticalSection
25	ntdll.RtlPcToFileHeader
26	ntdll.RtlGetVersion
27	ntdll.RtlUpcaseUnicodeChar
28	ntdll.RtlAnsiStringToUnicodeString
29	ntdll.LdrLockLoaderLock

```
30  ntdll.LdrUnlockLoaderLock
31  ntdll.ZwClose
32  ntdll.ZwCreateSection
33  ntdll.ZwMapViewOfSection
34  ntdll.ZwUnmapViewOfSection
```

All the functionality is contained in the two export functions, while DllMain does not contain any relevant code. What stands out is the extensive use of custom DWORD-size status codes throughout the code. Decompiled code example with status codes in result variable:

```
1   …
2   if ( !a1 || !a2 || !(_DWORD)v3 )
3   {
4       result = 0xA0032B02;
5       goto LABEL_45;
6   }
7   if ( strlen(a1) <= 1 )
8       goto LABEL_65;
9   result = get_kernel32_path(a1, &v43, &v37, &v41);
10  if ( result )
11      goto LABEL_45;
12  if ( (unsigned int)v3 < 0x1C )
13  {
14      LABEL_65:
15      result = 0xA0032B02;
16      goto LABEL_45;
17  }
18  pBuffer = allocate_buffer(v13, (unsigned int)v3, 0i64, v11, 0, 1);
```

```
19  buffer = pBuffer;
20  if ( !pBuffer )
21  {
22      LABEL_9:
23      result = 0xA0032B04;
24      goto LABEL_45;
25  }
26  if ( memcpy_s(pBuffer, v3, a2, v3) )
27  {
28      LABEL_11:
29      result = 0xA0032B06;
30      goto LABEL_45;
31  }
32  …
```

The main worker sample contains five icon resources with valid icons named 16, 256, 4097, 8193 and 12289. The names indicate different icon resolutions, but the icons only differ in the encrypted data appended to them which can be considered as a form of steganography. This data is encrypted with a custom algorithm and additionally zlib compressed. The same method is used within the SSP DLLs. A Python script for decryption and decompression can be found in the Appendix. After decryption, the data blob has the following structure:

```
struct data_blob {
    DWORD marker;    // Marker bytes (0x9A65659A)
    DWORD crc32;    // CRC32 value of decrypted or zlib uncompressed
bytes
    DWORD size;     // Size of decrypted or zlib uncompressed bytes
    DWORD option;    // Information if data is encrypted or zlib
compressed; 0x1 = encrypted, 0x2 = zlib compressed
```

```
    char data[];    // Encrypted or zlib compressed data
};
```

The decrypted data is as follows.

Resource 16:

```
System\CurrentControlSet\Control\Class\{4D36E97D-E325-11CE-BFC1-08002B
E10318}\0003\DriverData
```

Resource 256:

```
System\CurrentControlSet\Control\Class\{4D36E96A-E325-11CE-BFC1-08002B
E10318}\0000\DriverData
```

Resources 16 and 256 are the Windows registry keys that contain the decryption key for the embedded driver in resource 8193 and additional payloads that are likely going to be injected by the AcidBox driver.

Resource 4097:

```
C:\WINDOWS\SYSTEM32\lsass.exe
```

This resource contains the path of the process each sample uses to verify it is being loaded into the correct process. Resource 8193 contains the unsigned kernelmode payload driver, which is also encrypted with RSA. The driver is realized as a kernelmode DLL with two export functions InitEntry and InitExit. It contains the following cleartext strings:

```
ntoskrnl.exe
ntkrn
ntkrp
hal.dll
ntoskrnl
ntkrnlpa.exe
```

csrss.exe

PsCreateSystemThread

\Device\VBoxDrv

\DosDevices\PCIXA_CFGDEV

\Windows\ApiPort

\Sessions\%u\Windows\ApiPort

\Sessions\xxxxxxxx\Windows\ApiPort

\Device\PCIXA_CFG

\DosDevices\PCIXA_CFGDEV

Resource 12289 contains the VirtualBox VBoxDrv.sys driver v2.2.0.0 signed by Sun Microsystems, which we previously described is also vulnerable.

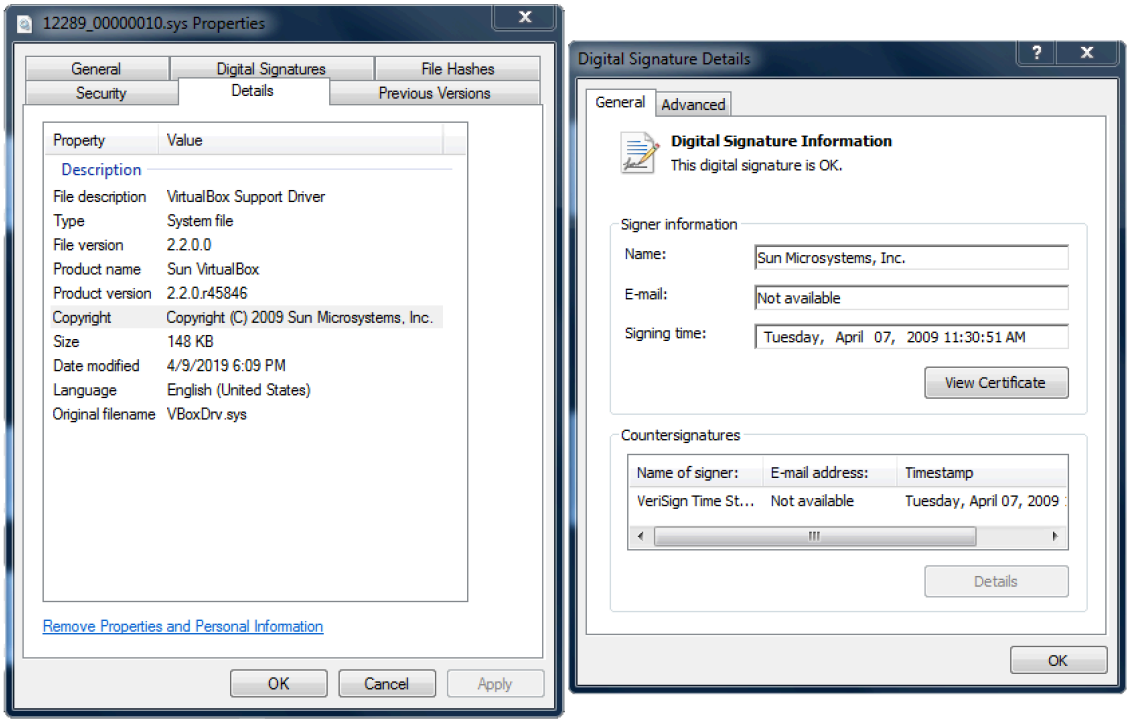

Figure 3. Signed vulnerable VBoxDrv.sys driver version 2.2.0.0

A Little Forensic Tidbit in the PE header

While studying the samples, PE header characteristics -- an often overseen forensic indicator -- caught our attention. This little known fact can be found in the export directory and can be helpful for attributing malware samples. All of the AcidBox samples contain gaps between the single exported function entries:

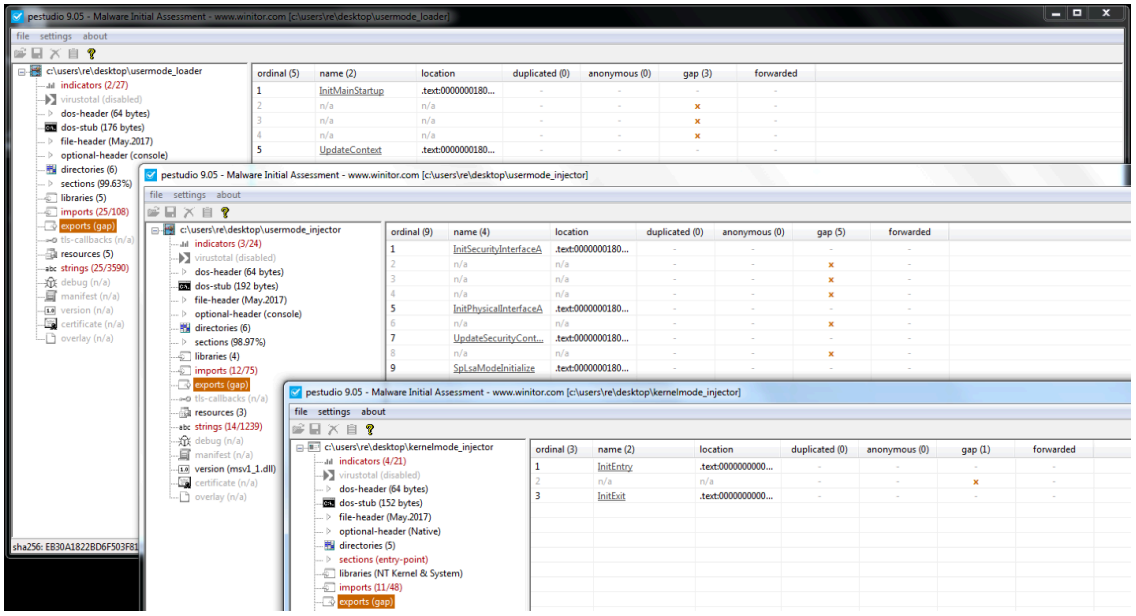

Figure 4. Export directories of AcidBox samples with gaps between the export function entries

Every AcidBox sample has a NumberOfFunctions value in the export directory that is bigger than the NumberOfNames value. This isn't something unusual, as not every exported function has to have a name too. Unnamed functions can be also called by their ordinal values. What is uncommon, however, is that the function entries which are unnamed are also zeroed out, thus not used.

This is the result when you use your own DEF file instead of __declspec(dllexport) to describe the attributes of your DLL file. When you use a DEF file, you can choose which ordinal your export function will have. This is not possible with __declspec(dllexport) as the Visual Studio compiler always counts your functions starting from one.

Using a DEF file instead of __declspec(dllexport) has some advantages. You are able to export functions by ordinals and you can also redirect functions among other things. The disadvantage is that you have to maintain an additional file within your project.

In the case of the AcidBox samples, we can conclude a couple of things. First, the author uses a DEF file, although he doesn't make use of its advantages. This could indicate it's a habit of the author to use DEF files. Second, the function ordinals seem to be chosen in steps of two integers. A possible explanation could be that the unused ordinals were once used for functions too. And last, if we assume the author really has chosen to make two integer steps, then in the usermode DLLs, one export function was removed. We can see that ordinal 3 is unused, leaving a bigger gap than one integer. All this information can be useful for malware attribution.

Conclusion

A new advanced malware, dubbed AcidBox, was used by an unknown threat actor in 2017 that went undetected until now. It uses a known VirtualBox exploit to disable Driver Signature Enforcement in Windows, but with a new twist: While it's publicly known that VirtualBox driver VBoxDrv.sys v1.6.2 is vulnerable and used by Turla, this new malware uses the same exploit but with a slightly newer VirtualBox version.

Sometimes, you are still able to find a technically interesting Windows malware that uses a new technique. This has become quite a rarity in today's threat landscape where everything is either a copy of a copy of a copy or

technically underwhelming. While AcidBox doesn't use any fundamentally new methods, it breaks the myth that only VirtualBox VBoxDrv.sys 1.6.2 can be used for Turla's exploit. Appending sensitive data as an overlay in icon resources, abusing the SSP interface for persistence and injection and payload storage in the Windows registry puts it into the category of interesting malware.

The samples we dubbed AcidBox are only part of a bigger toolkit which we, unfortunately, could not identify. However, we provide two Yara rules for detection and threat hunting. Additionally, if you happen to find an additional sample, or are even infected, you can use the provided Python script to extract the sensitive data appended to the icon resources. All of these can be found here at Unit 42's GitHub repository.

If you have any further information about this threat, don't hesitate to contact us.

Palo Alto Networks customers are protected from this malware. Our threat prevention platform with WildFire identifies this malware as malicious. AutoFocus customers can investigate this activity with the tag AcidBox.

We would like to thank Dr.Web, Kaspersky, ESET and hFireF0X for their collaboration.

IOCs

Files in Windows system32 directory

msv1_1.dll

pku.dll

windigest.dll

Mutexes

Global\BFE_Event_{xxxxxxxxxxxx--xxxx-xxxxxxxx-xxxxxxxx}

Global\{xxxxxxxxxxxx--xxxx-xxxxxxxx-xxxxxxxx}

The malware takes the MachineGuid stored in the registry and reshuffles the single characters alternating from the end to the beginning and vice versa in steps of two. For example, the MachineGuid string a9982d3e-c859-4702-c761-df7eea468ade gets transferred into e9a86daeecf5--67c2-07419d87-e34289da and appended to the above templates.

Windows Registry

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E97D-E325-11CE-BFC1-08002BE10318}\0003\DriverData (REG_BINARY type)

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E96A-E325-11CE-BFC1-08002BE10318}\0000\DriverData (REG_BINARY type)

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E969-E325-11CE-BFC1-08002BE10318}\0000\DriverData (REG_BINARY type)

Sample Hashes

Main worker DLL:

eb30a1822bd6f503f8151cb04bfd315a62fa67dbfe1f573e6fcfd74636ecedd5

Kernelmode driver:

3ef071e0327e7014dd374d96bed023e6c434df6f98cce88a1e7335a667f6749d

SSP DLL modules:

003669761229d3e1db0f5a5b333ef62b3dffcc8e27c821ce9018362e0a2df7e9

b3166c417d49e94f3d9eab9b1e8ab853b58ba59f734f774b5de75ee631a9b66d

3ad20ca49a979e5ea4a5e154962e7caff17e4ca4f00bec7f3ab89275fcc8f58c

Benign VirtualBox VBoxDrv.sys driver v2.2.0 (signed by “Sun Microsystems, Inc.”):

78827fa00ea48d96ac9af8d1c1e317d02ce11793e7f7f6e4c7aac7b5d7dd490f

---