Kernel Exploit Sample Hunting and Mining

Prepared by

Broderick Aquilino

Wayne Low

25 May 2016

Contents

Abstract

In the era of cyberwarfare, it becomes a norm to see cyber criminals use multi-level attacks to penetrate a multi-layered protected network infrastructure. We often see APT attackers manipulate 0-day or N-day Windows kernel vulnerabilities in order to guarantee a successful full system compromise. It would be a surprise if we do not see Windows kernel exploit involved in such targeted attacks.

It is also worth noting that beside APT attackers, the botnet operators also seize the opportunity to integrate these publicly, or sometime undisclosed, kernel exploits in their piece of work. One notable example is the CVE-2015-0057 Win32k exploit seen integrated into Dridex, a notorious banking Trojan spreading in the wild. This was first spotted by F-Secure's proprietary dynamic-analysis system in 20th April 2015 however no information was provided by F-Secure during that time. The exploit was also seen to be removed from newer version of Dridex distributed after late June 2015 since after the disclosure of FireEye. Apart from that, there are a lot more malware families manipulating kernel exploit, to name a few:

- Turla
- Necurs
- Carperb/Rovnix
- Evotoob
- Discpy
- Ramnit

This topic will focus on how to proactively discover the effective samples with kernel exploits, or potentially 0-day kernel exploits, through a dynamic-analysis system. This paper will also detail the analysis of some kernel exploit that could bypass kernel exploit detection and prevention methodology used in Host Instruction Prevention System (HIPS).

Introduction

Finding a previously undisclosed Windows kernel vulnerability, which can be done via various methods like automation fuzzing, black box analysis or static code analysis, is a lengthy process and not an easy task. In addition, reliable kernel vulnerability exploitation it is still questionable even though one is capable to discover new vulnerability. This is the main reason why malware authors are often quick in response to the elevation of privilege (EoP) exploits published on the Internet, as these exploits are often reliable. Besides that it also allows the malware authors to merge this open source EoP exploits into their malware code with least amount of effort. On the other hand this essentially enables malware researchers to effectively tell if the sample is malicious by finding the existence of the EoP exploits, which is one of the goals of this paper. Furthermore this paper also aims to dissect some of the interesting EoP payloads as well as the generic EoP prevention technology.

Elevation of privilege vs User Account Control

Some might wonder why on earth using EoP exploits to elevate a process privilege since there are some simpler privilege elevation methods like User Access Control (UAC) bypass techniques publicly available already. One of the advantages of manipulating UAC bypass over EoP exploit is its reliability and stability since the UAC bypass tricks are purely implemented under user mode code and it won't cause system crashes in case of program error. However UAC is working for Administrator account only. However, Administrator will always receive UAC dialog when the notification setting in User Account Control Settings was set to "Always notify" even if UAC bypass technique was deployed. As such it might alert the users the existence of suspicious activities when the UAC dialog appeared from nowhere. Hence, from the attacker's perspective, bypassing UAC might not be a perfect way for privilege escalation. As a result, in order to take advantage of full system privilege via EoP exploit while maintaining its resiliency, we often see the malware authors combine both the UAC bypass as well as the EoP exploit. The Figure 1 depicts the logic typically used by malware authors in the implementation of a standard privilege elevation routine.

Nonetheless, there is still lots of malware families do not follow this standard but instead they attempt to include multiple EoP exploits exploiting different Windows kernel vulnerabilities, which has no restrictions similar to UAC bypass techniques. Apparently the only caveat of exploiting Windows kernel vulnerabilities is its reliability as any unsuccessful attempt of EoP exploit might cause Blue Screen of Death (BSOD) on the machine. This is the reason why we regularly see the similar kernel vulnerabilities, for example CVE-2013-3660, CVE-2014-4113, CVE-2015-0057 and CVE-2015-1701 being manipulated by malware authors, particularly those exploits published by security researchers which are proven to be reliable most of the time. Certainly the exploits based on the CVEs mentioned above have been ported by some researchers to support 64-bit platform and were open source, which is another reason why they are popular to the malware authors.

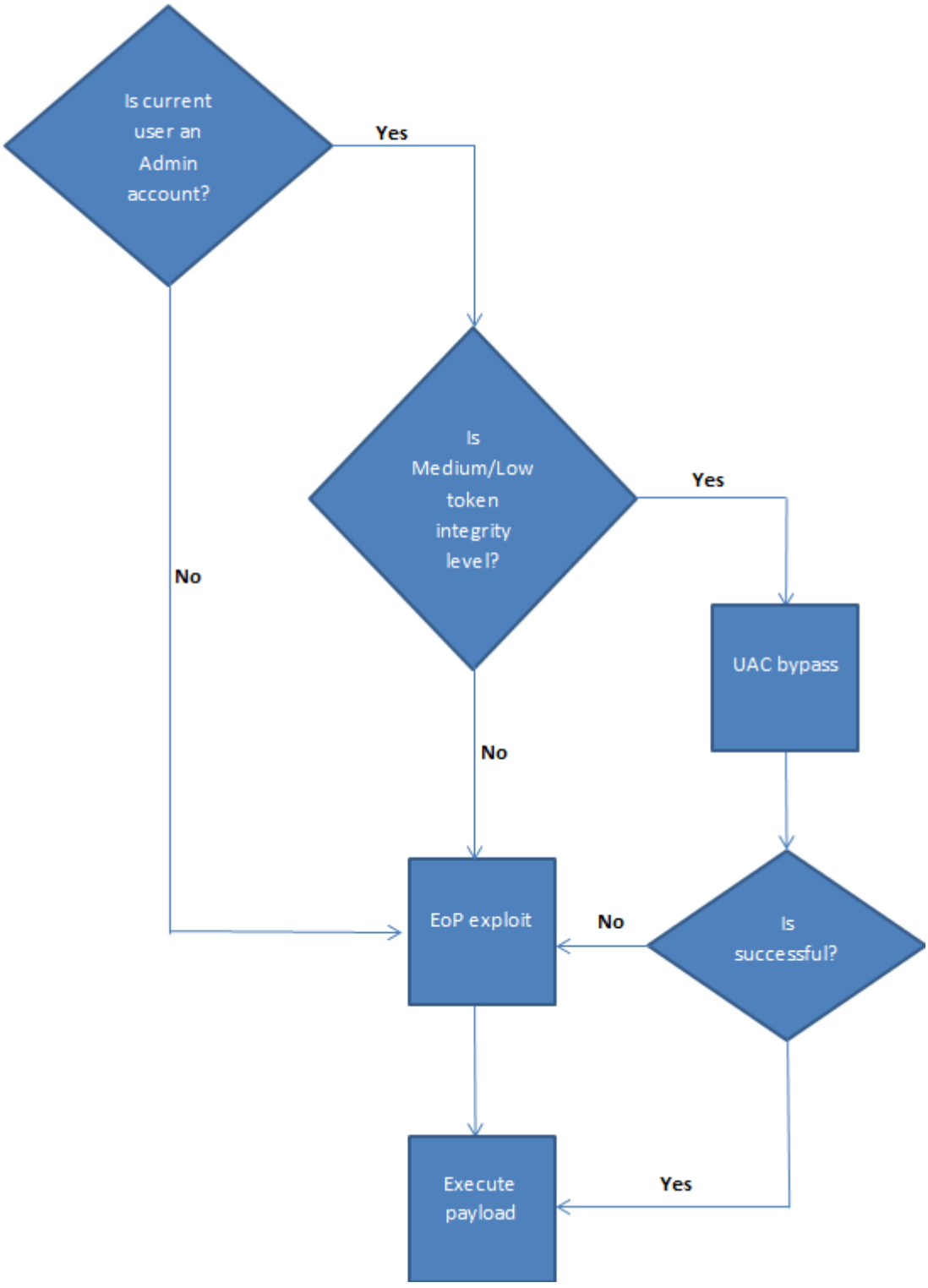

Figure 1: Standard elevation of privilege decision

Write-what-where primitive kernel exploit

While discussing kernel exploitation, write-what-where primitive is a most commonly used vector. It is powerful and most straightforward kernel exploitation technique, which is also widely used by exploit writers. In a nutshell, write-what-where primitive lets an attacker store (write) a specific value (what) to a specific kernel pointer address (where). A classic target for write-what-where scenario is *HalDispatchTable* but the description of this target is out of the scope of this paper. Hereafter, we will refer to this write-what-where primitive approach as writing arbitrary address to *HalDispatchTable*. Generally, an exploit can be constructed as following:

1. Prepares a user mode buffer to store the shellcode
2. Uses write-what-where approach to overwrite *HalDispatchTable+sizeof(void*)* with shellcode address
3. Redirects code execution to the prepared shellcode using *NtQueryIntervalProfile*

This approach is the simplest form of exploitation but it has some limitations as well. Due to the enhanced mitigation techniques introduced in modern Windows operation system, user-mode shellcode execution in step 3 will not work on Windows 8 and above - thanks to the Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP). Even though SMEP is useful to stop user-mode shellcode execution, there are various known techniques can be deployed to circumvent SMEP [1]. A researcher, named n3phos, has shown a working example on how to create a reliable exploit leveraging primitive approach to bypass SMEP [2]. This is just one of the real world examples to show how prominent is this approach to the exploit writers. In fact this approach usually depends on certain vulnerability context but it works in most of the vulnerability classes. For example, some of the exploits that are not based on the primitive approach:

1. CVE-2014-4113 – The lack of sanitization in win32k code allows arbitrary value returned from user-mode to kernel mode. This allows the attacker to send a crafted *win32k!tagWND* structure define at a NULL memory address allocated via *NtAllocateVirtualMemory*, which will be used later in the win32k code that result in arbitrary code execution from the field crafted in *tagWND->lpfnWndProc.*
2. CVE-2015-1701 – The nature of this vulnerability resides in the infamous user-mode callback functions in win32k code. Essentially, an application is not restricted to hook these user-mode callback functions where the initial address of these callback functions can be dereferenced by *PEB->KernelCallbackTable*. The root cause of this vulnerability is that a callback function, in this case *user32!__ClientCopyImage*, is hooked by the exploit so that the hooked function can modify the state *tagWND->bServerSideWindowProc* of the server-side windows procedure that could result in a user-mode windows procedure stored in *tagWND->lpfnWndProc*, to be executed under kernel-mode context.

As you can see, some exploits don't necessary leverage *NtQueryIntervalProfile* to achieve kernel exploitation. So, this type of kernel exploits might not be able to be spotted directly using our sample

hunting methodology, however the use of native API *NtQueryIntervalProfile* derived from *HalDispatchTable* is still favorable to us in the event of mining kernel exploit samples and it has been proven to be helpful in discovering 0-day kernel exploits.

Discovering elevation of privilege exploit samples

To reiterate, the target of *HalDispatchTable* in write-what-where primitive has been the most common vector used in kernel exploitation. In the event of successfully exploiting kernel vulnerability, the attacker will need to call *NtQueryIntervalProfile* in order to transfer code execution to arbitrary address under the attacker's control. So it is important to keep in mind that these APIs string are native kernel APIs and they are not commonly used by user-mode application, hence it is safe to say that when these APIs string are found in a PE binary, the said PE binary could potentially contain a kernel exploit.

```
rule www_kernel_exploit

{

        meta:

        description = "Typical APIs used in Write-What-Where Windows kernel exploitation"


        strings:

        $NtQueryIntervalProfile = "NtQueryIntervalProfile" nocase

         $ZwQueryIntervalProfile = "ZwQueryIntervalProfile" nocase

         $HalDispatchTable = "HalDispatchTable" nocase


        condition:

        ($NtQueryIntervalProfile or $ZwQueryIntervalProfile) and $HalDispatchTable and

         not tags contains "native"

}
```

Listing 1: Yara rule to identify *NtQueryintervalProfile*

The Yara rules shown in Listing 1, 4 and 5 might not give a promising result in finding samples that are encrypted or encoded. So it is necessary to use these rules in a dynamic analysis system that allows you to effectively find the related API string from the stripped or unpacked binary exists in the process memory. While discussing elevation of privilege, the traditional kernel exploit payload involves the following steps:

1. Get the EPROCESS structure of the System (process id=4) and subsequently obtains its corresponding access token address.

2. Get the EPROCESS structure of the exploit process and replace its access token address with the System's access token.
3. As a result the exploit process possesses the same access token as the System which has the highest privilege on Windows environment.

Traditionally these steps are implemented in pure assembly code, which would use constant offsets to obtain the actual access token address from EPROCESS structure and then traversing the active process list and other types of hardcoded information. However, payloads these days would use the documented Windows kernel API to improve its resiliency and portability. Some of the example looks like the following code snippet:

```
int __stdcall elevate_system_privilege()

{

  int result;

  PEPROCESS currentEproc;

  PEPROCESS systemEproc;



  ptrPsLookupProcessByProcessId(g_dwCurrentPid, &currentEproc);

  ptrPsLookupProcessByProcessId(g_dwSystemPid, &systemEproc);

  result = g_dwOffsetEprocToken;

  *(_DWORD *)((char *)currentEproc + g_dwOffsetEprocToken) = *(_DWORD *)((char *)systemEproc +
g_dwOffsetEprocToken);

  return result;

}
```

Listing 2: A variant of elevation of privilege payload

```
int elevate_privilege()

{

  PACCESS_TOKEN currentToken;

  PACCESS_TOKEN SystemToken;

  PEPROCESS currentEproc;



  g_boolExploited = 1;

  *(_DWORD *)(g_pHalDispatchTable + 4) = g_origNtQueryIntervalProfile;

  if ( !ptrPsLookupProcessByProcessId(g_dwCurrentPid, &currentEproc) )
```

```
  {
    currentToken = pfnPsReferencePrimaryToken(currentEproc);
    SystemToken = pfnPsReferencePrimaryToken(*(_DWORD *)g_PsInitialSystemProcess);
    replace_token(currentToken, SystemToken);
  }
  return 0;
}
```

Listing 3: Another variant of elevation of privilege payload

The following rules are used in conjunction with *www_kernel_exploit* shown in Listing 1 to generically identify the other variants of kernel exploit payload that might not call the native API *NtQueryIntervalProfile* directly based on the fact that some kernel exploit that don't leverage primitive approach still utilize documented kernel API in token stealing operation.

```
rule generic_um_win32k_kernel_exploitation
{
        meta:
        description = "Typical APIs used in user-mode exploit to leverage win32k kernel mode
vulnerability"

        strings:
         $PsLookupProcessByProcId = "PsLookupProcessByProcessId" nocase
         $NtQuerySystemInformation = "NtQuerySystemInformation" nocase
         $ZwQuerySystemInformation = "ZwQuerySystemInformation" nocase

        condition:
         ($NtQuerySystemInformation or $ZwQuerySystemInformation) and
         $PsLookupProcessByProcId and (pe.imports("user32.dll") or pe.imports("gdi32.dll")) and
         tags contains "peexe" and
         not tags contains "native"
}
```

Listing 4: Yara rule to generically identify payload used in win32k.sys exploit

```
rule generic_um_kernel_exploitation

{

        meta:

        description = "Typical APIs used in user-mode exploit to leverage kernel mode
vulnerability"



        strings:

        $NtQuerySystemInformation = "NtQuerySystemInformation" nocase

         $ZwQuerySystemInformation = "ZwQuerySystemInformation" nocase

         $PsLookupProcessByProcId = "PsLookupProcessByProcessId" nocase

         $PsReferencePrimaryToken = "PsReferencePrimaryToken" nocase



        condition:

        ($NtQuerySystemInformation or $ZwQuerySystemInformation) and

         ($PsLookupProcessByProcId or $PsReferencePrimaryToken) and

         tags contains "peexe" and

         not tags contains "native"

}
```

Listing 5: Yara rule to generically identify payload used in exploit

Case study

Dridex/Dyre

One of the interesting findings from the result of rule 1, deployed in our in-house dynamic analysis system, in Listing 1 is the discovery of first malware on May 2015 exploiting CVE-2015-0057 Windows kernel vulnerability – 3 months after Microsoft patched the vulnerability as MS-15-010. After analyzing the exploit, we realized that it was the first malware exploiting vulnerability specified in CVE-2015-0057. It is worth to mention that there was no public exploit code available that time. However the EoP payload used by the exploit is similar to the code presented in Listing 2.

After the blog post of Dridex leveraging the 1-day exploit has been disclosed by FireEye around July 2015, we noticed that the EoP exploit has been disappeared in later variants of Dridex. The author has probably learnt the lesson from FireEye's disclosure of its 1-day exploit, so it started to push the EoP exploit to the infected machine downloadable from the remote server only if the infected machine has not installed Microsoft's patch for CVE-2015-0057 and the current user is not running as administrator [4]. It also started to introduce UAC bypass techniques to elevate the malware process integrity level when the user account is local administrator group. One of the UAC bypass techniques used by Dridex was AppCompat whitelisting vulnerability that allows malware whitelist their executable into Application Compatibility Database which can effectively suppress UAC prompt when the malware's executable is triggered. Microsoft has addressed this vulnerability in October 2015 update [5]. As we constantly keep track of the development of Dridex, we realized that the recent variants of Dridex have obfuscated all readable string and the strings will only be de-obfuscated in run-time. Some of the de-obfuscated string, as shown in Figure 2, leads us to believe that Dridex intended to remove unwanted Microsoft's patches like KB3045645 and KB3048097 if it found installed on the infected machine before executing any UAC bypass routines. However we couldn't find any routine that actually trigger the command. So we assumed that this feature could be implemented in the future variants.

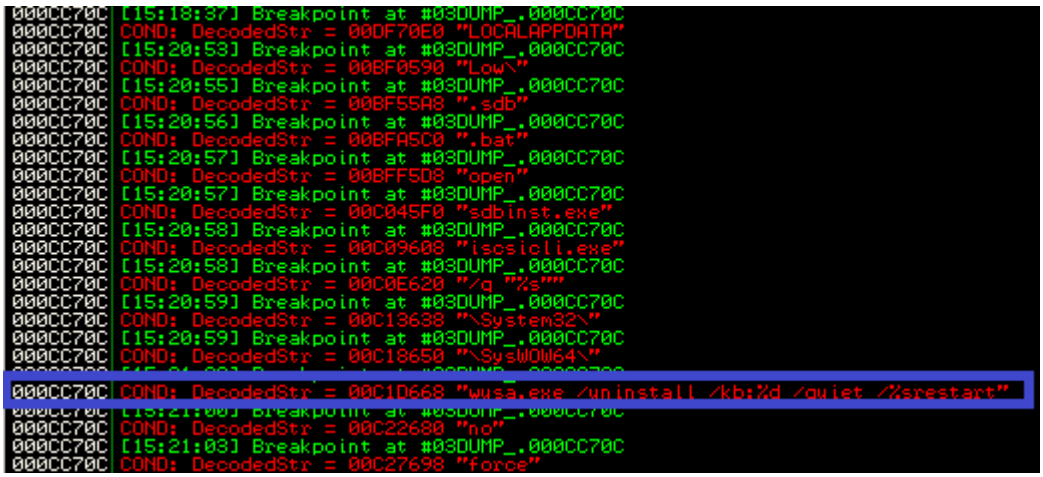

Figure 2: Dridex wanted to remove installed Microsoft patches

Carperb/Rovnix

Rovnix is one of the well-known malware families to adapt multiple publicly known kernel exploits. This is mainly because it uses bootkit component as its persistence mechanism to store fileless rootkit driver in the hidden disk sector to deliver and execute its payload before Windows boot up. Bootkit technique is very powerful to subvert most of the security products; however it is mandatory for one to have sufficient privilege to open physical disk in order to install bootkit components. Rovnix consists of a variety of open source kernel exploits like CVE-2015-1701, CVE-2014-4113 and CVE-2013-3660. Our analysis of the recent variant, sample's compilation timestamp suggested 11 January 2016, reveals that Rovnix started to integrate Hacking Team's exploit leveraging vulnerability CVE-2015-2487 – that was leaked around July 2015. These exploits payloads are identical to the proof-of-concept (POC) published on the Internet, as shown in Listing 1 and 2, which indicates that whoever created this sample just copy-paste the source code from the POC.

Before exploiting the kernel vulnerabilities, it performs some checks to determine if the machine is vulnerable or not:

1. Determine if the process is executed under Administrator account
2. Determine the integrity level of the process
3. Determine win32k.sys file's timestamp via *GetFileTime* and *FileTimeToSystemTime* APIs call.
4. Attempts from the latest exploit to the older ones, until one of them successfully elevate the privilege

When the kernel exploitation is not successful, it falls back to UAC bypass methods to elevate the privilege.

Ramnit & Evotob

Ramnit is a long-standing malware family that first emerged around 2011. At the same time, it has been evolving to counteract the improved mitigations introduced in the latest Windows operating system. It is also one of the very few malware families that adapt the new kernel exploits from the old one very quickly; while Evotob is another malware family doing that same thing.

Family	Timestamp	CVEs
Ramnit	07 October 2014	CVE-2013-0008 CVE-2013-3660
	05 February 2015	CVE-2013-3660 CVE-2014-4113
Evotob	29 April 2015	CVE-2013-3660 CVE-2014-4113
	10 June 2015	CVE-2013-3660 CVE-2015-0057
	14 July 2015	CVE-2015-0057 CVE-2015-2387

Table 1: Different kernel exploits used by Ramnit and Evotob

Another similarity between Ramnit and Evotob is that they share the same local privilege elevation exploit framework that we believed developed by the same author.

```
int __cdecl exploit_eop(LPSTR lpCommandLine)
{
  if ( !(unsigned __int8)is_NT6_2() && !(unsigned __int8)is_NT6_3() )
  {
    if ( (unsigned __int8)is_NT5() )
    {
      if ( !(unsigned __int8)is_patch_NT5("KB3000061") )
      {
        if ( check_sid_isadmin() )
          return 1;
LABEL_6:
        exploit_cve_2014_4113(lpCommandLine);
        return 1;
      }
    }
    else if ( !(unsigned __int8)is_patch_NT6("KB3000061") )
    {
      if ( check_sid_isadmin() && check_token_il() > 1 )
        return 1;
      goto LABEL_6;
    }
    exploit_cve_2013_3660(lpCommandLine);
    return 1;
  }
  return 0;
}
```

Ramnit

```
int __cdecl exploit_eop(LPSTR lpCommandLine)
{
  if ( !(unsigned __int8)is_NT6_2() && !is_NT6_3() )
  {
    if ( (unsigned __int8)is_NT5() )
    {
      if ( !(unsigned __int8)is_NT5_patch_installed("3036220") )
      {
        if ( is_admin() )
          return 1;
LABEL_6:
        exploit_eop_cve_2015_0057(a1);
        return 1;
      }
      goto LABEL_13;
    }
    if ( (unsigned __int8)is_wow64() )
    {
      if ( !(unsigned __int8)is_NT6_patch_installed("3036220") )
      {
        if ( is_admin() && get_access_token_IL() > 1 )
          return 1;
        goto LABEL_6;
      }
LABEL_13:
      execute_payload();
      return 1;
    }
  }
  exploit_eop_cve_2015_2387(a1);
  return 3;
}
```

Evotob

Figure 3: Similar local privilege escalation framework used by Ramnit and Evotob

Discpy

Although Discpy is not a popular family, it uses some out of ordinary exploit payload that we haven't seen from other malware families. The Discpy sample was first seen on May 2014 exploiting CVE-2013-3660. Though it was a common vulnerability exploited by many malware families, what caught our attention was its post kernel exploitation payload.

When a machine was exploited by Discpy successfully, the exploit first allocates kernel mode buffer via *ExAllocatePool* to hold the Asynchronous Procedure Call (APC) injector routine. In short, this APC injector is responsible to inject the final payload as an APC to a remote process. The target remote process would be any active privileged Windows processes, which is very unlikely to be monitored by HIPS. In normal circumstances, an unprivileged process is unable to access to privileged Windows processes because of the lack of security permission. This security permission is neglected when the unprivileged process gain arbitrary code execution in kernel mode context due to successful kernel exploitation. In other words, this process running under kernel mode context can perform all operations to any privileged processes without the constraints of a restricted user account. In this particular exploit's payload, the APC code injector executed in kernel mode will traverse a list of active threads from svchost.exe and find alertable thread (*ETHREAD.Tcb.Alertable*) and non-system thread (*ETHREAD.Tcb.SystemThread*). One mandatory and important step when implementing this kind of payload is that the payload code must reside in kernel mode buffer to avoid any system crashes when injecting APC routine code from the current process context to remote process context via native API *NtWriteVirtualMemory*. After the APC routine code is in place in the remote process, an APC object can be initialized from *KeIntiailizeApc* and assigned this APC object the alertable thread that it found earlier via *KeInsertQueueApc*. When in the APC queue, the APC thread will be triggered and executed only after *ETHREAD.Tcb.ApcState.UserApcPending* has been set accordingly.

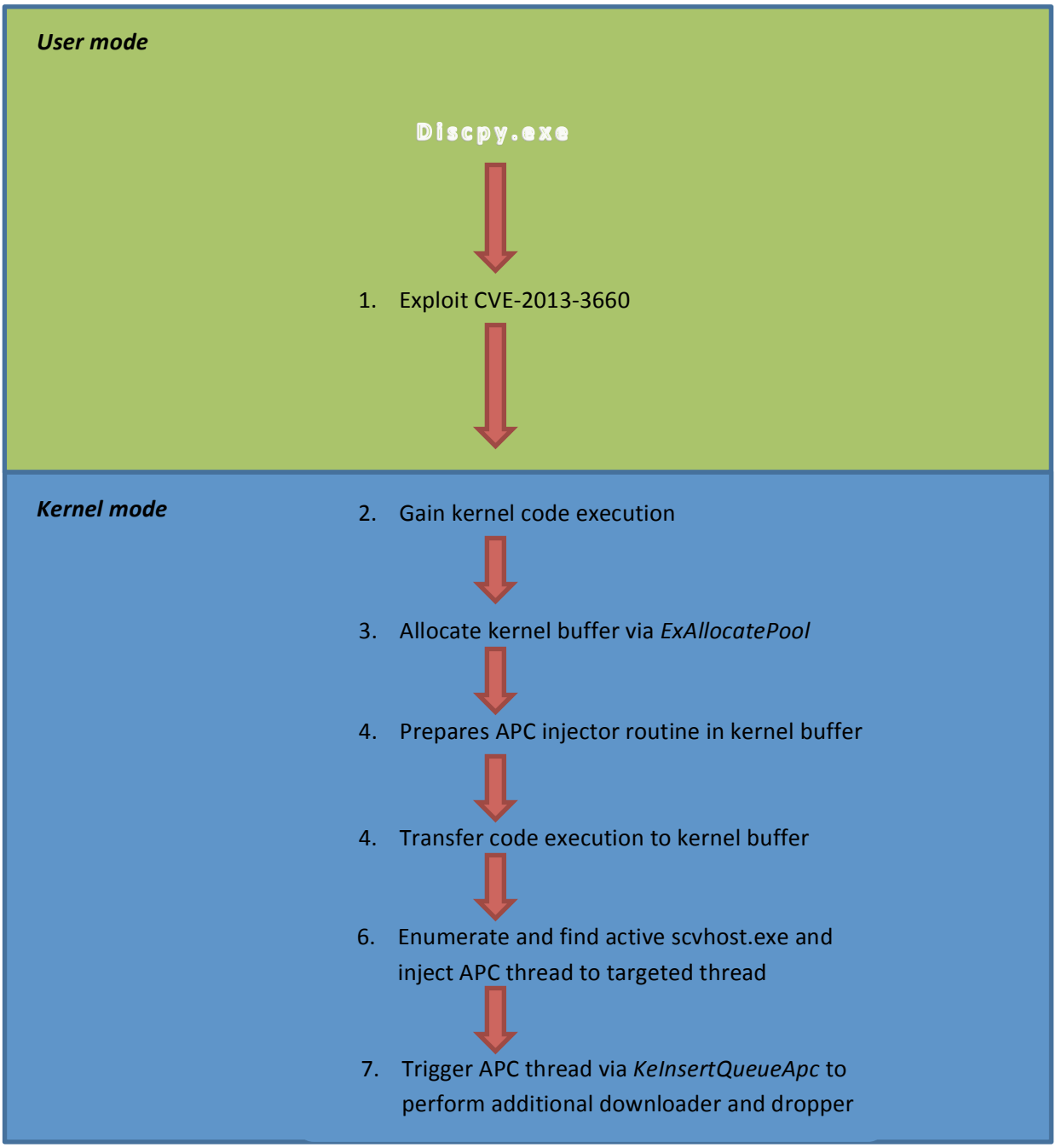

Figure 4: Discpy post kernel exploit payload

Anomalies

There is a famous saying “knowing is half the battle”. For example, we need to know that a process is misbehaving first before we would know that there is a need to terminate it. In this section, we share some anomalies that we observed from the real-world kernel exploit sample that can be used to search for elevated processes.

Non-system Services Having System Integrity Level

In normal circumstances, there should not be any user processes having system integrity level. According on MSDN [6], “Processes you start and objects you create receive your integrity level (medium or high) or low if the executable file's level is low; system services receive system integrity”. Something is wrong when there is a process having system integrity level but is not a system service.

The problem is that there is no definite way to determine whether a process is a system service or not. However, some checks might be possible. For example, system services should have been launched by services.exe. This means that there should not be any processes having system integrity level that are not under the process tree of services.exe unless they are Windows system services.

Another check would be that there should not be any processes having system integrity level that are child of applications like explorer.exe, which interacts with the desktop. This leads us to our next approach.

Processes Having System Integrity Level with Non-system Integrity Level Parent Process

The previous anomaly might require some sort of whitelist process names of Windows system services. However, a more generic anomaly to look for is if the parent of a process having system integrity level but the former has a lower integrity level (Figure 5).

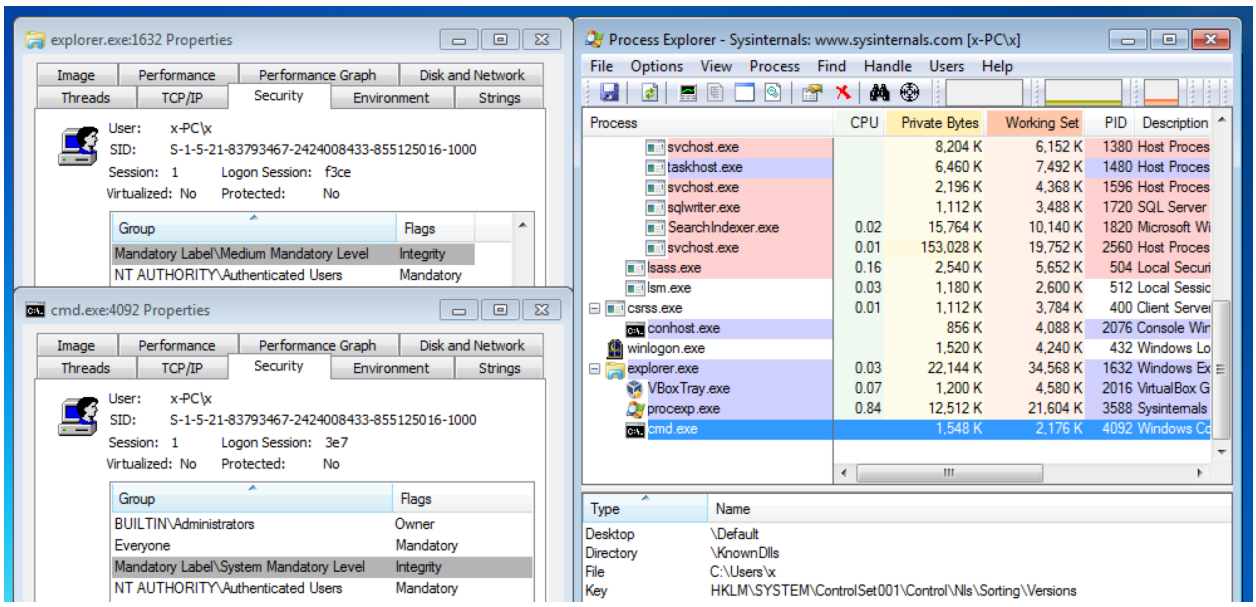

Figure 5: Child process (cmd.exe) has system integrity level but parent (explorer.exe) only have medium integrity level

Processes Having Integrity Levels Lower than High with Administrative Windows Privileges

So far we have been discussing how to identify anomalies based on the integrity level. Malwares want to elevate their integrity levels to the highest level to be able to access the resources not available to them at lower levels. However, it is also possible for them to directly acquire the necessary privileges without elevating to a higher integrity level. For example, an exploit process could enable SE_DEBUG_PRIVILEGE to allow itself to inject its routines into any processes regardless of the integrity levels or SE_RESTORE_PRIVILEGE allows them to write to any files regardless of the DACL. It is important to take note that none of the Windows applications running medium integrity level would be able to adjust the access token privileges as some of the privileges are not present by default in standard user account, for example SE_DEBUG_PRIVILEGE. Unless the malware is able to do self-elevation to administrator account through UAC elevation and then adjust the access token privileges via Windows API *AdjustTokenPrivileges* (Figure 7) or by enabling the disabled privileges on lower-integrity malware process through kernel exploit, which is the anomaly that we are trying to detect.

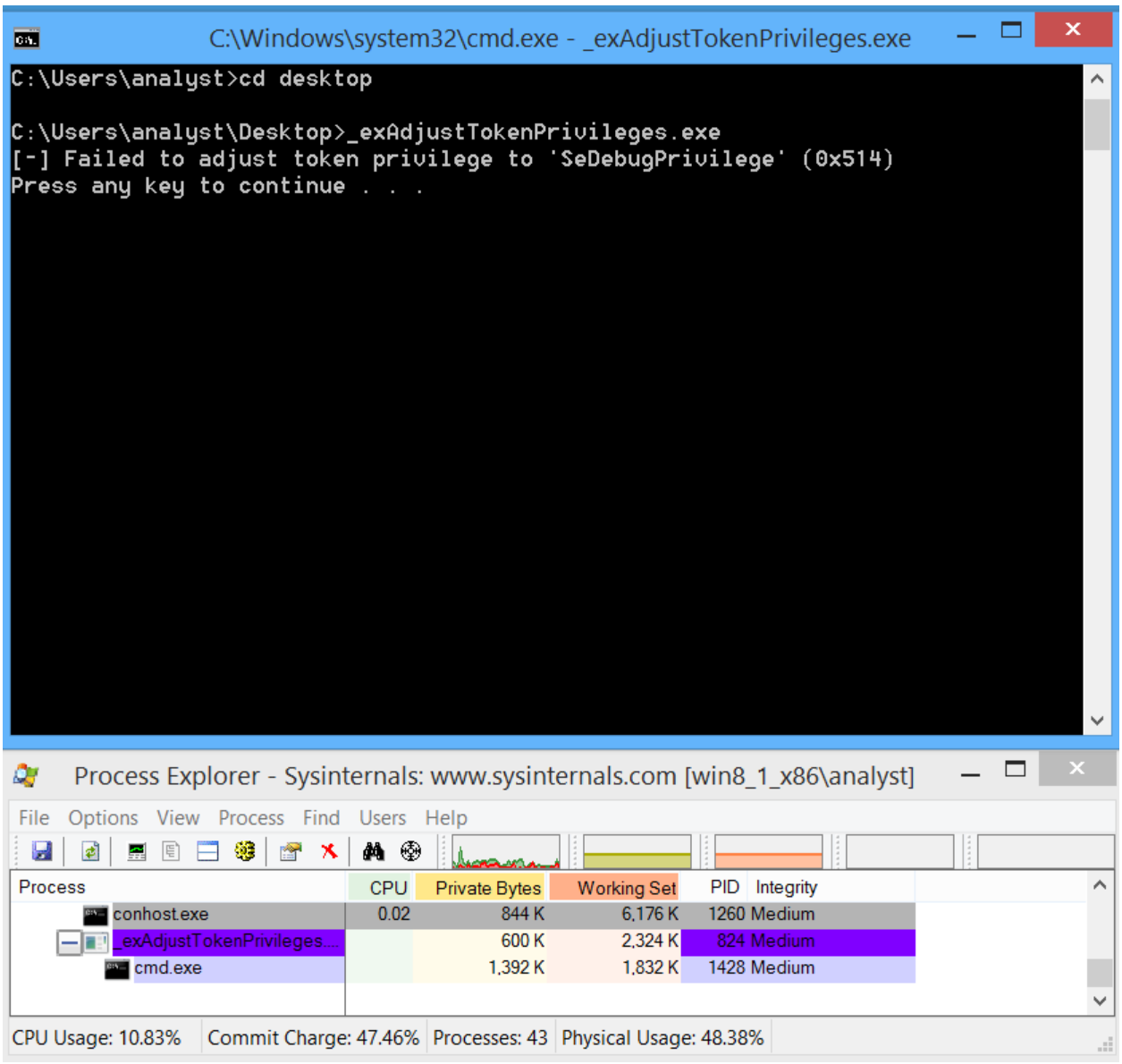

Figure 6: Normal user account is not allowed to adjust privileges that are not assigned implicitly by operating system

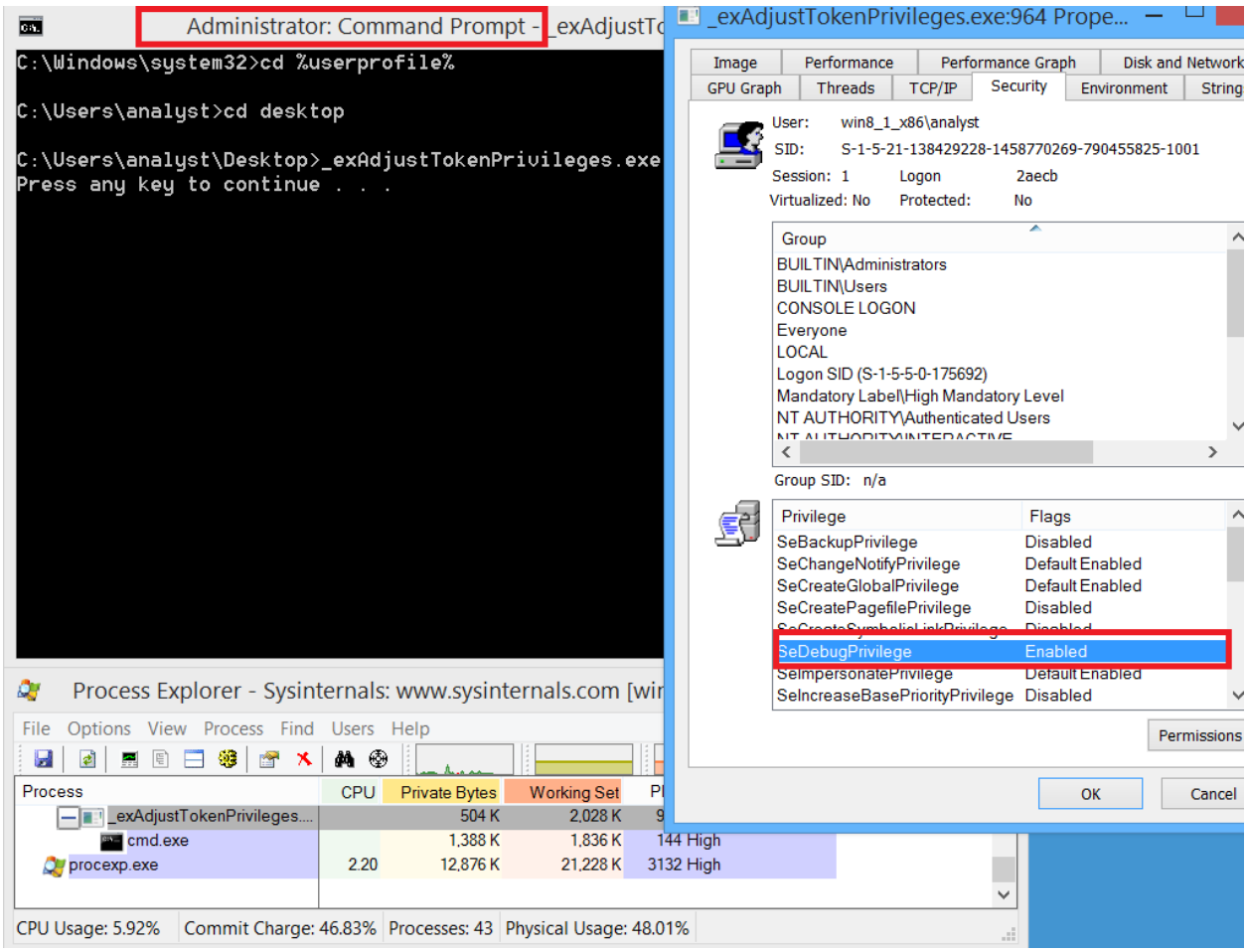

Figure 7: An example of administrative privilege like SE_DEBUG_PRIVILEGE can be enabled only by a local administrator account running in high integrity level

According on MSDN [7], "certain administrative Windows privileges can be assigned to an access token only with at least a high integrity level" (Listing 7). This means that there is something wrong when they are found in processes having lower integrity levels.

SE_CREATE_TOKEN_PRIVILEGE	**SE_DEBUG_PRIVILEGE**
SE_TCB_PRIVILEGE	SE_IMPERSONATE_PRIVILEGE
SE_TAKE_OWNERSHIP_PRIVILEGE	SE_RELABEL_PRIVILEGE
SE_BACKUP_PRIVILEGE	SE_LOAD_DRIVER_PRIVILEGE
SE_RESTORE_PRIVILEGE	

Listing 7: Administrative privileges associated with a high integrity level

Processes Accessing Objects with Higher Integrity Level

Another technique that malware may use without elevating to higher integrity level is to remove the protection of its target object. This may be achieved by tampering with the security descriptor of the target object. According on MSDN [8], “A security descriptor with no DACL (also known as a NULL DACL) gives unconditional access to everyone”. In our experiment, it seems that setting the SACL and DACL field of the security descriptor or the security descriptor itself to NULL has the same effect [9].

There may be other techniques for processes having lower integrity level to manipulate objects with higher integrity level therefore a more generic approach is to simply check if a process is trying to access an object with a higher integrity level. The idea of the Windows integrity mechanism is to “restrict the level of access that is available to lower-integrity subjects” [7]. Something fishy is going on; let us say for example, if a process having medium integrity level is opening a handle to another process having high integrity level to inject code into it.

Conclusion

Samples exploiting kernel vulnerabilities will eventually use kernel mode APIs because it is not trivial to implement those routines themselves. This means that we can use the names of those APIs to hunt for such samples because user mode samples should not have them in normal circumstance. However, expect the names to be encrypted so the approach presented in this paper is best done in dynamic analysis systems.

This paper has also presented some anomalies that may be used for discovering unauthorized processes having elevated privileges. The usual payload of samples exploiting kernel vulnerabilities is EoP therefore being able to discover such processes can also support the hunting process. It also provides information that may help in mitigating such attacks. The latter is particularly challenging because it is usually game over when a malware is able to reach the kernel mode. The malware can practically do anything to remove all the anomalies. Checking for the presented anomalies also comes with a performance overhead.

However, this does not mean that we should make it easy for them. Hope lies in the fact that it is not trivial to tamper with kernel objects directly and that can be changed between Windows releases. This might discourage attackers to iron out all the anomalies and just focus on attaining an EoP.

References

[1] http://j00ru.vexillium.org/?p=783

[2] https://github.com/n3phos/zdi-15-030

[3] https://dl.packetstormsecurity.net/papers/attack/CVE-2014-4113.pdf

[4] http://blog.fortinet.com/post/what-s-cooking-dridex-s-new-and-undiscovered-recipes

[5] https://support.microsoft.com/en-us/kb/3045645

[6] https://msdn.microsoft.com/en-us/library/windows/desktop/bb648648%28v=vs.85%29.aspx

[7] https://msdn.microsoft.com/en-us/library/bb625963.aspx

[8] https://technet.microsoft.com/en-us/library/cc781716%28v=ws.10%29.aspx#w2k3tr_acls_how_ctlz

[9] https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf