*TECHNICAL*

*WHITE PAPER*

# Easy local Windows Kernel exploitation

**Abstract**

In this paper I detail how to easily exploit some kind of windows kernel vulnerabilities. This is about 3 really easy tricks that can be used in different situations depending what you want to do and what you can do.



## By Cesar Cerrudo

IOActive™

COMPREHENSIVE COMPUTER SECURITY SERVICES

# Contents

# Introduction

There was so many things I wanted to do when researching this, but I didn't have enough time. I wanted to do some statistics about the amount of drivers installed in a regular PC and perform quick security audits on them, but who cares about statistics if we all know that most drivers are full of vulnerabilities and will crash the system if you stare at them for 2 seconds. There will be always vulnerabilities in kernel code.

Windows kernel exploitation is still kind of dark art. There are just few papers about Windows kernel exploitation techniques and there are few public good and reliable exploits available.

When talking about write "what" "where" exploitation, when you can write some controlled value to a controlled address, there are almost no generic techniques that works across different Windows versions and service pack level. Also some techniques are not reliable and/or complicated. There is no documented easy way to exploit vulnerabilities when "what" is a fixed value, it's null, or when you can just write one or two bytes. Also no easy way to exploit when you can only increment or decrement the value on "where" or other restrictions. Basically no generic technique for hard to exploit vulnerabilities. Another common thing between most known techniques is that you always end up running code in kernel mode, this could be cool sometimes but it's not safe and any little mistake will cause a BSOD.

# All started with a good paper

On January 2010 Matthew "j00ru" Jurczyk and Gynvael Coldwind published "GDT and LDT in Windows kernel vulnerability exploitation" [1]. This is a good paper detailing a technique for Windows kernel exploitation. The technique they described in the paper is cool but only works if you can control the "what" value and it's a little bit complicated. What it got my attention was the use of NtQuerySystemInforamtion() API for getting the kernel address of the KPROCESS structure of processes, this inspired me. I already had seen NtQuerySystemInformation() API long time ago when researching how Process Explorer from Sysinternals worked but I never thought that this API would be useful for kernel exploitation.

When calling NtQuerySystemInformation() with SystemHandleInformation parameter it will return an array of the following structure:

```
typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO {
   USHORT UniqueProcessId;          //Process id of process with open handle to the object
   USHORT CreatorBackTraceIndex;
   UCHAR ObjectTypeIndex;           //Object type: thread, process, token, etc.
   UCHAR HandleAttributes;
   USHORT HandleValue;
   PVOID Object;                    //Kernel address of object
```

```
    ULONG GrantedAccess;
}SYSTEM_HANDLE_TABLE_ENTRY_INFO,
*PSYSTEM_HANDLE_TABLE_ENTRY_INFO;
```

With NtQuerySystemInformation() API you can get the kernel addresses of all Windows object handles, also you get the object type, the process that has an open handle to the object, etc.

When I saw this API in this paper the first thing I wondered was why they didn't try to overwrite other values in KPROCESS or on other kernel object structures. I thought it would be possible to find some easy way to exploit kernel vulnerabilities since you can use this API to get almost any kernel object address and know exactly where to write in order to elevate privileges. After that I started to think and try some ideas from time to time until I came up with the tricks I'm going to describe.

# Making exploitation easier

Maybe I'm lazy, but I like easy things, to do the least effort, I know that's not very good but you find ways to make things faster and easier. I wanted an easy way to exploit kernel vulnerabilities, I didn't want to run code in kernel, etc.

I started to wonder what could be done by doing just one write of a controlled (or not sometimes) value to the kernel in order to elevate privileges without running any code in kernel:

- Could I remove ACLs of Windows objects?
- Could I set any privileges in a process token?
- Could I replace a process token?

If we can remove ACLs then we can bypass security controls. If we can set arbitrary privileges to a process then we can elevate privileges and if we can replace a token with a chosen one we can elevate privileges too.

After some research I could confirm that It's possible to do all of the above with just one write to kernel and without running code in kernel mode as I'm going to detail.

I also wondered  why does people want a System shell?

Most kernel exploits will end up providing a System shell, which is cool, it's really cool right?. But have you ever asked why do you want that? I think that what I really want is to be able to perform some privileged actions in a system (such as dumping hashes, reading/writing registry or files, injecting code in privileged processes, etc.) but if I can do that without having a System shell then there is no need for it, who cares, maybe it's

something interesting to show to girls (where?), look baby here I have a system shell!. The point is that you don't always need a System shell you just need a way to do stuff that requires elevated privileges. For instance, if you get the "Take ownership" privilege then you can change ACLs of any Windows object and then elevate privileges. If you get "Debug privileges" then you can debug any privileged process, inject code and elevate privileges, if you can remove the ACL of LSASS process then you can access the process and dump passwords hashes, etc.

# First trick

Let's start with the first trick. Looking at different kernel object structures I found the OBJECT_HEADER structure that is common to all Windows objects. Every Windows object (process, thread, token, event, etc.) as far as I know has an object header just before the main object body which its kernel address we can easily get with NtQuerySystemInformation() API.

```
kd> dt nt!_OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Lock             : _EX_PUSH_LOCK
   +0x00c TypeIndex        : UChar
   +0x00d TraceFlags       : UChar
   +0x00e InfoMask         : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void          //Body -0x4 x86  or -0x8 x64 >=Win2K
   +0x018 Body             : _QUAD                         //Here starts the object structure
```

The interesting part is the Security descriptor pointer which points to all the ACL related information. What is cool is that if you just NULL out the pointer then the object becomes ACL free, meaning that it can be accessed by any user process since it doesn't have any ACL. For instance if you NULL out the security descriptor of the LSASS process object then any process will be able to access LSASS process and perform any actions on it. Cool isn't it?. Since we can get Windows objects addresses from user mode, in order to NULL out ACLs we just need to write NULL to the object address -0x4 on x86 or -0x8 on x64 and that's it. The best part is that this structure doesn't change, it is the same across all Windows versions at least since Windows 2000 maybe since Windows NT!

Let's see step by step how this trick works.

### Nulling out ACLs

1. Get target object (process, thread, etc.) kernel address by calling NtQuerySystemInformation(SystemHandleInformation) API
2. Write NULL to [object address–0x4] on x86 or [object address – 0x8)] on x64
3. Manipulate the target object (if it's a process then inject code, read memory, etc.) to escalate privileges from user mode.

Simple and easy, isn't it?

## Second trick

I hope you liked the first trick now let's continue with the second trick. Another interesting kernel structure I found is SEP_TOKEN_PRIVILEGES structure which is part the TOKEN object structure. Here is the TOKEN structure for Windows >= Vista:

```
typedef struct _TOKEN
   {
...   //same offset and structure on Vista, Win7, Win2008 R1 & R2 x86 x64)
/*0x040*/     typedef struct _SEP_TOKEN_PRIVILEGES
                        {
                            UINT64 Present;
/*0x048*/                   UINT64 Enabled;
                            UINT64 EnabledByDefault;
                        } SEP_TOKEN_PRIVILEGES, *PSEP_TOKEN_PRIVILEGES;
...
  }TOKEN, *PTOKEN;
```

Those three fields in the SEP_TOKEN_PRIVILEGES structure are bitmasks, every bit set on Present field means a privilege present in the token, in the Enabled field means the enabled privileges in the token and in the later one the privileges enabled by default. I started to play with this and I found something interesting, it doesn't matter what values are in Present and EnableByDefault field, what is checked by Windows when performing actions on the system are the bits on Enabled field. I guess you are realizing how this could be abused, just write values to [TOKEN+0x48] to add privileges. One cool thing is that there is no integrity check performed so you can just write there and instantly elevate privileges in a very reliable way. Also the SEP_TOKEN_PRIVILEGES structure is the same since Vista and it's the same offset inside the TOKEN structure! This is different before Windows Vista so let's see it.

On Windows XP and 2003 the Token structure and privileges are different:

```
lkd> dt _TOKEN
   ...
   +0x074 Privileges   Ptr32 _LUID_AND_ATTRIBUTES   //points to  VariablePart
   ...
   +0x0a0 VariablePart     : Uint4B

lkd> dt _LUID_AND_ATTRIBUTES
   +0x000 Luid           : _LUID
   +0x008 Attributes      : Uint4B          //0x0 disabled, 0x1 enabled by default, 0x2 enabled

lkd> dt _LUID
   +0x000 LowPart         : Uint4B          //number identifying a privilege
   +0x004 HighPart        : Int4B
```

At offset TOKEN+0x74 there is a pointer that points to VariablePart at the end of TOKEN structure, there, depending on the amount of privileges on the token there will be one or more LUID_AND_ATTRIBUTES structures. This structure contains also a LUID structure where the value present there indicates a privilege identified by a number, for instance privilege 0x14 is “Debug privilege”. VariablePart seems to be always at 0xA0 offset in TOKEN structure allowing us to easily get that address and write a value there in order to elevate privileges.

Basically with this trick we will be able to instantly elevate privileges on all Windows versions by setting the privileges we want in the token by written a value to the already described places in the TOKEN structure. We could set all privileges if we can fully control the written value or we could enable specific privileges depending on the vulnerability and the operating system version.

Let’s see now some Windows powerful privileges we would like to get when elevating privileges:

- Debug programs : allow us to debug any program.
- Take ownership: allow us to change any Windows object ACL
- Restore files and directories: allow us to overwrite any file in the system
- Impersonate a client after authentication: allow us to impersonate any user
- Load and unload device drivers: allow us to load drivers
- Create a token object: allow us to create tokens
- Act as part of the operating system: allow us to authenticate as any user
- Etc: there are other privileges that could be used to perform privileged actions on the system

All those privileges will allow us to perform privileged actions on Windows, if we can set all the privileges then it would be almost the same as running code under System account. But

like I said we usually don't need all the privileges just the necessary to perform some desired actions.

In order to use this trick we just need to get the kernel address of the primary token of our exploit process. With this kernel address we can know exactly where we need to write a value to set arbitrary privileges to the primary token of our exploit and instantly elevate privileges. Elevating privileges has never been so easy!.

Let's see this second trick step by step.

## Enabling privileges

1. Get process primary token calling OpenProcessToken() API and then search its kernel address using NtQuerySystemInformation(SystemHandleInformation) API

2. Write 0xffffffff or the value you can to [TOKEN+0x48] to enable privileges in the process primary token on Windows>=Vista

   Or write some value (0x14 to enable debug privilege) to [TOKEN+0xA0] on WinXP or 2003

3. Perform privileged actions depending on enabled privileges

I think you can see how easy is this since it can be described with three easy steps and we are talking about kernel vulnerability exploitation.

I successfully used this trick in a difficult to exploit use after free vulnerability found by Tarjei Mandt. I won't go into the details of the vulnerability I will just detail how to exploit it. In this vulnerability we could only control eax when landing in a dec dword ptr [eax+4] instruction. As you can see exploiting this is not straight forward and usually it would require some reverse engineering efforts, finding a reliable way across different Windows version, etc. Luckily with this last trick I just described it becomes very easy the exploitation. Let see first how to do it on Windows>=Vista:

By default on Windows 7 at TOKEN+0x48 offset it's the value 0x800000 which correspond to Enabled field in SEP_TOKEN_PRIVILEGES structure representing the enabled privileges on the token. If this value would be different then we can easily get that value there since Windows allows to enable/disable privileges in a token. If we convert this value to binary: 0x800000==100000000000000000000000b we can see that there is just one privilege enabled, SeChangeNotifyPrivilege which is a common privilege most process primary tokens have . This is the only one enabled by default on low privileged processes but if you don't want to get any problems just disable all the privileges in the token and enable SeChangeNotifyPrivilege. Since we have 0x800000 at TOKEN+0x48 offset and we can easily get that kernel address, we just put that address – 0x4 in eax and then when dec dword ptr [eax+4] executes we get 0x800000-0x1==0x7fffff== 11111111111111111111111b this means lots of privileges enabled and instant elevation of privileges.

Now let's see how to exploit the same vulnerability on Windows XP and 2003 because the TOKEN structure is different and we need to exploit the vulnerability a bit different. By default on non privileged tokens at TOKEN+0xA0 offset it's always the value 0x15 which is located in LUID structure, this value represents "Audit privilege" (if you want to be safe just remove all privileges and leave only this one). If we set eax to TOKEN+0xA0-0x4 kernel address then when dec dword ptr [eax+4] executes we get 0x15-0x1==0x14==Debug privilege this means we enable the "Debug privilege" allowing the exploit to debug any process and elevate privileges.

A difficult to exploit vulnerability became easy and simple to exploit.

## Third trick

Lastly we have the third trick, being the last trick doesn't mean it's the least interesting.

Below we can see an extract of the EPROCESS structure which corresponds to a Windows process object:

```
typedef struct _EPROCESS  (Win7 RTM x86)
{
… // this offset changes in some Windows versions but stable on different service pack level)
/*0x0F8*/     struct _EX_FAST_REF Token;  // 0x4 bytes x86  or 0x8 bytes x64
...
 }EPROCESS, *PEPROCESS;
```

Inside EPROCESS structure there is an structure with a pointer to the primary token of the process, this is an structure because the three less significant bytes are used for fast reference counting (tests show that ignoring this fast reference counting seems to not affect the system stability). Replacing the primary token has been a known technique for elevating privileges after executing code in kernel mode. Most people using this technique doesn't care much about reference counting and most of the time people just set the pointer to the System process primary token in the exploit without increasing the reference counter. If you run the exploit many times Windows will crash causing BSOD because every time the exploit process terminates the reference counter of the System process primary token is decremented and when the count reaches 0 the object is freed but there will be still references to the object that will make the system crash when the object is tried to access.

Since my idea is to do a write to change the token replacing it with a privileged one without running code in kernel, I need to be sure that there won't be any BSOD so I need to deal with the reference counting issue. It seems the reference count problem can be solved by incrementing the PointerCount field on the OBJECT_HEADER structure (described on first trick) but in order to do this you should be able to make two writes. First, write a big value

on the PointerCount field so the object is never freed and then overwrite the primary token of the exploit process with a System process primary token to elevate privileges.

Another option for token replacement without problems is to replace the token with first write and then restore original token with second write just before the exploit finishes, but this approach could be a bit dangerous because if the exploit crashes then the system could crash too if the token is not restored properly.

If you can do just one write then after elevating privileges and before the exploit finishes duplicate the System token twice in other process that never terminates like LSASS, etc. so the token object won't be freed and there will be no BSOD.

Something interesting I found is that we can use any kind of token for this trick, no matter it is not a Primary token, any kind of token will work. It seems Windows doesn't check the token type which is good because it's really simple to get an identity token we just need to abuse some inter-process communication (IPC) mechanisms and we can get a System identity token. Another approach but a little bit difficult is to guess a token account name from different processes because you just get the address, handle, etc. but not the name. You can't know for sure from which account is the token but you can do some good and high probability guessing by looking at some system processes and the tokens that are present there most of the time.

Let's see this last trick step by step.

## Replacing process token

1. Get System Identity token by hooking NtOpenThreadToken() and calling MsiInstallProduct()*, then get object kernel address using NtQuerySystemInformation(SystemHandleInformation)

- If multiple writes

    2. Increase ref count with first write to PointerCount [TOKEN – 0x18] on x32 or [TOKEN – 0x30] on x64 >= Win 2000

    3. Second write to replace Token on EPROCESS with System token

- If one write

    2. Replace Token on EPROCESS

    3. After elevation and before exploit finishes duplicate the System token twice in other process that never terminates like LSASS, etc.

*MsiInstallProduct() API uses LPC for inter-process communication (IPC) to communicate with Windows Installer service which runs under System account. After some IPC calls the calling process ends up with a System identity token. We need to hook

NtOpenThreadToken() API in order to detect when the token is got and to duplicate it to avoid the token handle being closed. If we are running the exploit from a low integrity process we will need to use another API or use another method to get a System identity token since MsiInstallProduct() API won't help to get a System identity token due to low integrity process restriction.

# Conclusions

Exploiting some kernel vulnerabilities is really easy with help of NtQuerySystemInformation() API.

These tricks allow to quickly and easily build reliable and multi Windows version kernel exploits even when the vulnerability is "difficult" to exploit.

These are just some ways I found and researched but there should be other ones that maybe allow even more easier exploitation.

Finally, you don't always need a System shell.

# Thanks

***To Ruben Santamarta and Tarjei Mandt for feedback and PoCs***


# References

[1] - http://j00ru.vexillium.org/?p=290#more-290

# Contact

**IOActive, Inc.**

Global Headquarters:  701 5th Avenue, Suite 6850, Seattle, WA 98104

Toll free: (866) 760-0222
Office: (206) 784-4313
Fax: (206) 784-4367

EMEA Headquarters:  One New Change, London EC4M 9AF

Toll Free: +44 (0) 8081.012678
Direct: +44 (0) 2032.873421