Windows Kernel Exploitation Part 2: Type Confusion

BY HIMANSHU KHOKHAR / ON APRIL 3, 2019

Introduction

Welcome to the second part of Windows Kernel Exploitation series. In the second part, we are taking a detour from usual memory corruption vulnerabilities (which are a majority in case of the driver we are exploiting). I was quite confused whether to make it the first part because how easy it is to exploit, but here we are, once we have tasted blood in kernel land.

What is Type Confusion?

Type confusion is a vulnerability where the application doesn't verify the type of an object (function, data type, etc.) and then processes it as it expects but the passed object is some other object.

Vulnerability

Now we have got it cleared, let us have a look at the vulnerable code (function TriggerTypeConfusion located in *TypeConfusion.c*).

The kernel first checks if the buffer resides in user land and then it allocates memory for it in Non-Paged Pool. Once that has been done, the kernel assigns ObjectID from user mode buffer to kernel mode buffer and does the same for object type.

```
132         KernelTypeConfusionObject->ObjectID = UserTypeConfusionObject->ObjectID;
133         KernelTypeConfusionObject->ObjectType = UserTypeConfusionObject->ObjectType;
```

Assigning ObjectID and ObjectType

After done that, the kernel calls *TypeConfusionInitializer* function on the object (kernel mode and not on the user mode).

```
154         Status = TypeConfusionObjectInitializer(KernelTypeConfusionObject);
```

Calling TypeConfusionInitializer on Object

Let us have a look at the function:

```
76  NTSTATUS TypeConfusionObjectInitializer(PKERNEL_TYPE_CONFUSION_OBJECT KernelTypeConfusionObject) {
77      NTSTATUS Status = STATUS_SUCCESS;
78
79      PAGED_CODE();
80
81      DbgPrint("[+] KernelTypeConfusionObject->Callback: 0x%p\n", KernelTypeConfusionObject->Callback);
82      DbgPrint("[+] Calling Callback\n");
83
84      KernelTypeConfusionObject->Callback();
85
86      DbgPrint("[+] Kernel Type Confusion Object Initialized\n");
87
88      return Status;
89  }
```

Function TypeConfusionObjectInitializer

This function receives the object and calls a function pointer present inside the object.

Let us have a look at the structure of KERNEL_TYPE_CONFUSION_OBJECT (which is essentially a structure), present in *TypeConfusion.h* header file. This header file

structure), present in *TypeConfusion.h* header file. This header file contains definitions for user mode object as well as kernel mode object, which makes exploitation of this exploit easier than that of stack overflow.

```
57      typedef struct _USER_TYPE_CONFUSION_OBJECT {
58          ULONG_PTR ObjectID;
59          ULONG_PTR ObjectType;
60      } USER_TYPE_CONFUSION_OBJECT, *PUSER_TYPE_CONFUSION_OBJECT;
61
62      typedef struct _KERNEL_TYPE_CONFUSION_OBJECT {
63          ULONG_PTR ObjectID;
64          union {
65              ULONG_PTR ObjectType;
66              FunctionPointer Callback;
67          };
68      } KERNEL_TYPE_CONFUSION_OBJECT, *PKERNEL_TYPE_CONFUSION_OBJECT;
```

Object Prototypes

First, let us see what the user mode object contains. The user mode object is a structure that holds 2 members:

1. Object ID
2. Object Type

In case of kernel mode object, it is also a structure that holds 2 members:

1. Object ID
2. The second member is a UNION which can either hold:
    a. Object Type
    b. Callback (A function pointer)

Now, if you remember, a UNION can hold one member at a time, and here it can either be an Object Type or a pointer to a function which will be called by *TypeConfusionInitializer* function.

The confusion occurs when the function *TriggerTypeConfusion* function does not validate whether the second member is

function does not validate whether the second member is *ObjectType* or *Callback*.

Exploiting the Confusion

To exploit this confusion, all we need to do is to pass a structure whose second member is the address of the function we want to call from kernel land.

In case of our exploit, it is going to be the address of our Token Stealing Shellcode and replace the token of our process so when a new process is created, it will be created with that token.

But there is a problem, the shellcode provided with the HEVD (*TokenStealingPayloadWin7* does not work and crashes the machine).

Modifying the shellcode

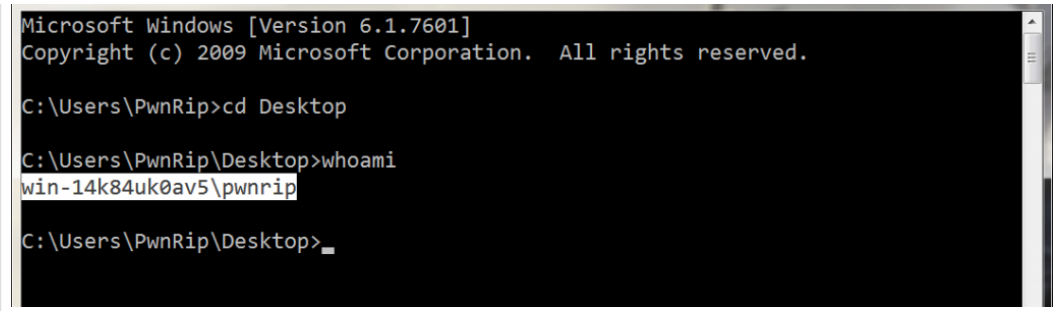
Since the function *TypeConfusionInitializer* calls the *Callback* pointer as it is a function, we need to setup function prologue and epilogue, and change the **ret 8** to **ret**.

Note: I compiled my shellcode functions as **naked** but if you don't do that, the provided shellcode can be used as it is. I just don't like extra code to be added to my shellcode by the compiler.

My code for the exploit is located here

Getting the Shell

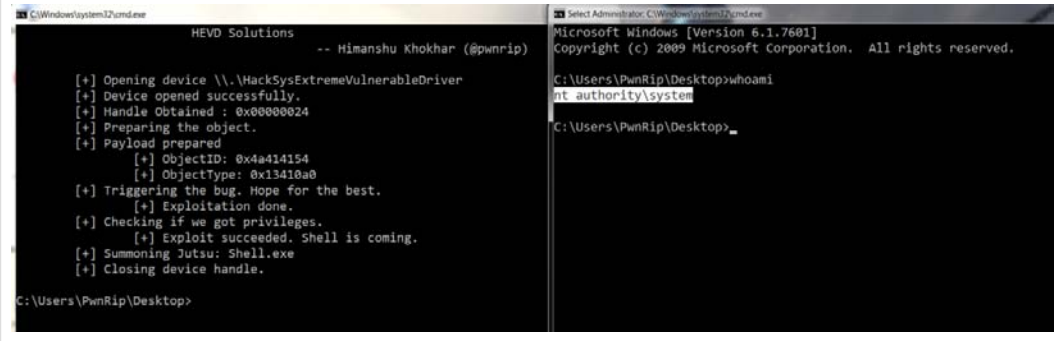
Let us first verify whether I am a regular user or not.



Regular User

As it can be seen, I am just a regular user.

After we run our exploit, I become **nt authority/system**



SYSTEM Shell via exploitation of Type Confusion

That's for this this part folks, see you in next part.

HEVD SHELLCODING TYPE CONFUSION WINDOWS KERNEL EXPLOITATION

PREVIOUS
Windows Kernel Exploitation Part 1: Stack Buffer Overflows

NEXT
Windows Kernel Exploitation Part 3: Integer Overflow

1 Comment

Himanshu Khokhar

No idea about that.

JUNE 13, 2019 REPLY

Leave a Reply

COMMENT

NAME *

EMAIL *

WEBSITE

Save my name, email, and website in this browser for the next time I comment.

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT