CTRL-INJECT

by Rotem Kerner on May 08, 2018 - enSilo Corporate and Product

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 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 –

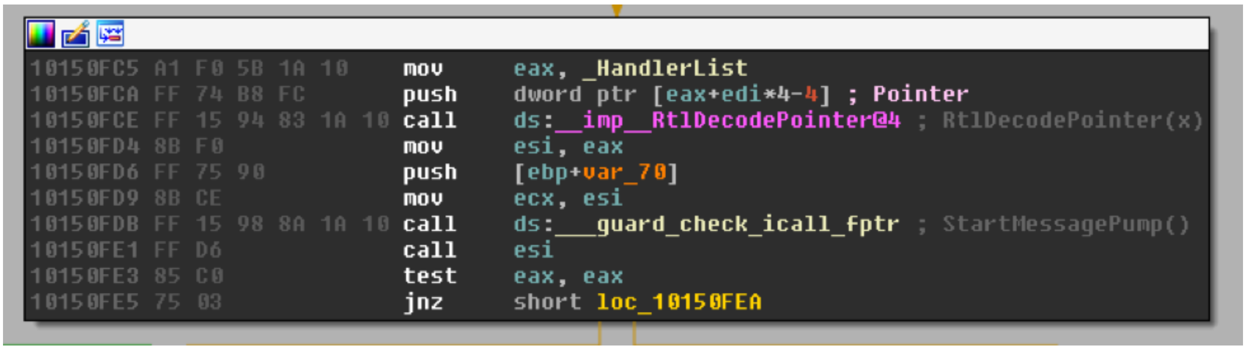


Figure 1: Decoding pointer before running and CFG check

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.

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.

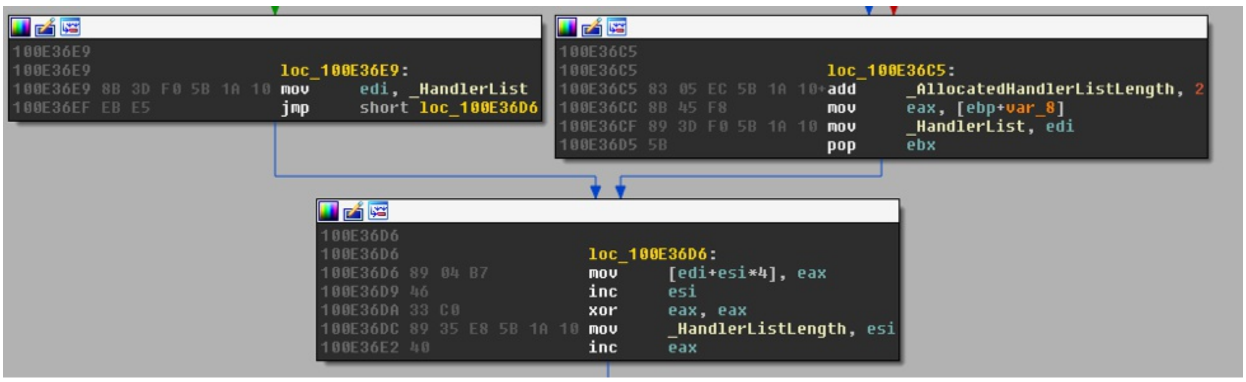


Figure 3: Updating the HandlerList and increasing HandlerListLength

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.

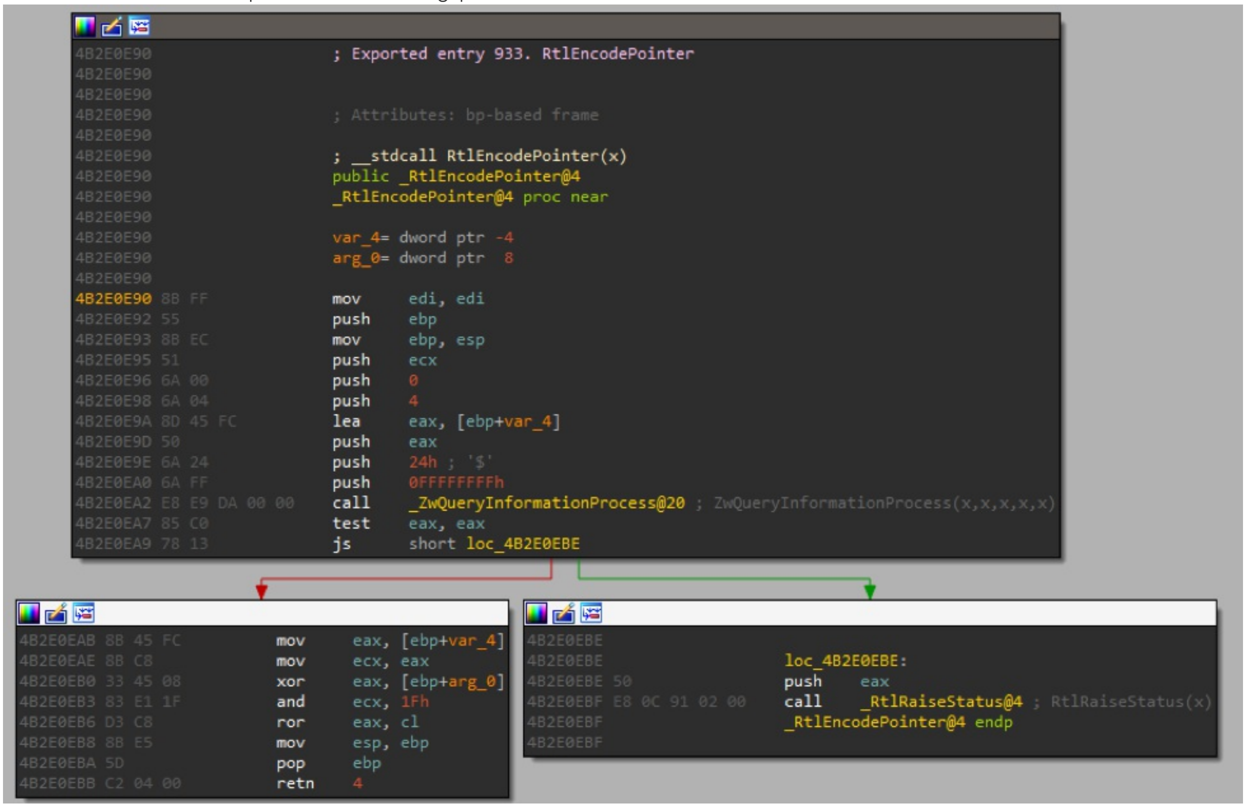


Figure 4: Inner workings of RtlEncodePointer

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:

EncodedPointer = (OriginalPointer ^ SecretCookie) >> (SecretCookie & 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

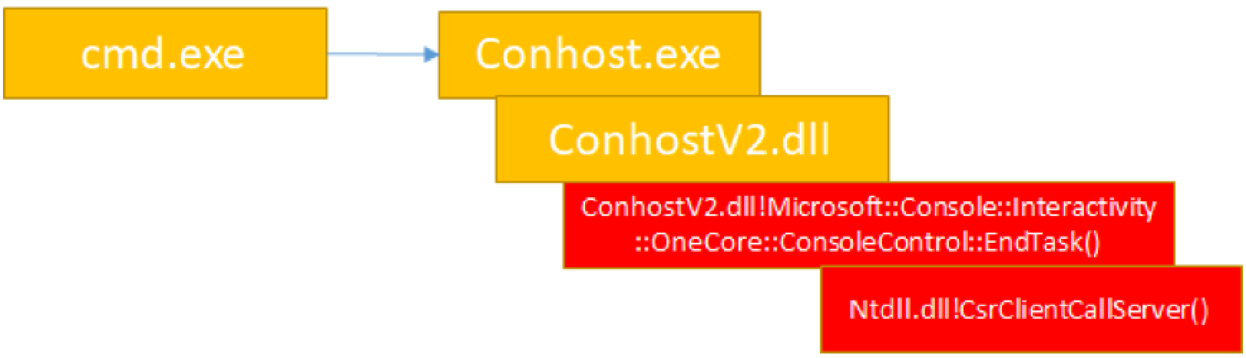
Now that everything is in place, all we need to do is to trigger Ctrl+C on the target process in order to invoke our code. There are several ways in which we can trigger it. In this case, we used a combination of SendInput to trigger a system wide Ctrl key-press, together with a PostMessage for sending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

```
void TriggerCtrlC(HWND hWindow) {

    INPUT ip;
    //ShowWindow(hWindow, 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(hWindow, WM_KEYDOWN, 0x43, 0);
}
```

BEHIND THE SCENES

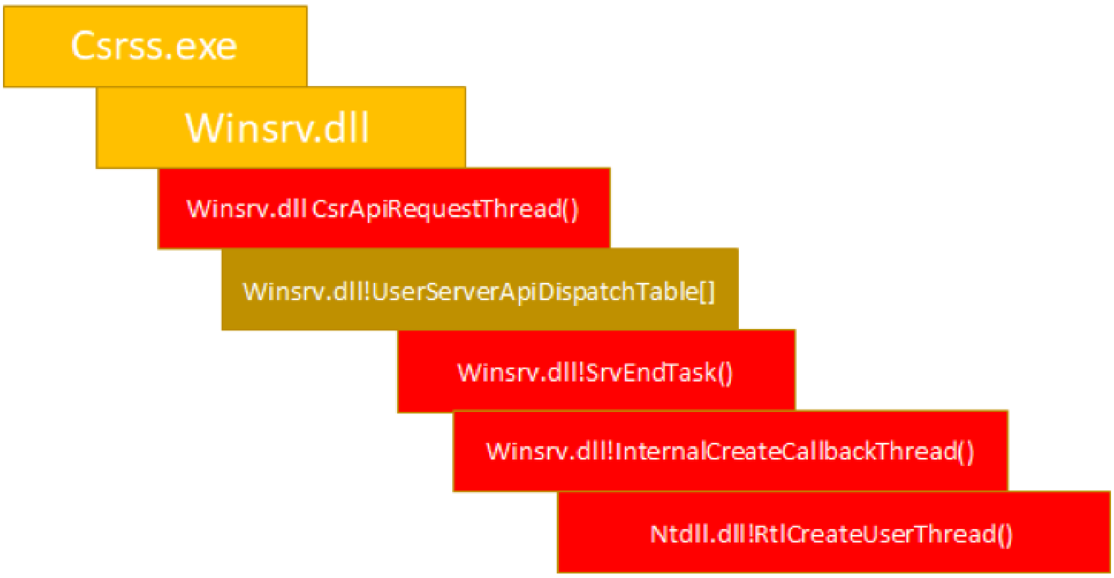
Essentially, in this process injection technique, we inject our code to the target process, but we never invoke it directly, that is, we never call CreateRemoteThread ourselves or alter execution flow using SetThreadContext. Instead, we are making csrss.exe invoke it for us which is far less suspicious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

- Attach to a console process using OpenProcess.
- Allocate a new buffer for the malicious payload by calling VirtualAllocEx.
- Write the data into the allocated buffer using WriteProcessMemory.
- 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.
- Letting the remote process know that our new pointer is a valid pointer using SetProcessValidCallTargets.
- Finally, triggering Ctrl+C signal using a combination of PostMessage and SendInput.
- Restore the original handlers list.

Related Blog Posts

How To Handle The Increase In Powershell Attacks ? one
Read More

Customers Say It Best - Managed Security Service Provider one
Read More

5 Ways to Tackle Ransomware Attacks One
Read More

enSilo Blocks New Variant of Adwind RAT one
Read More