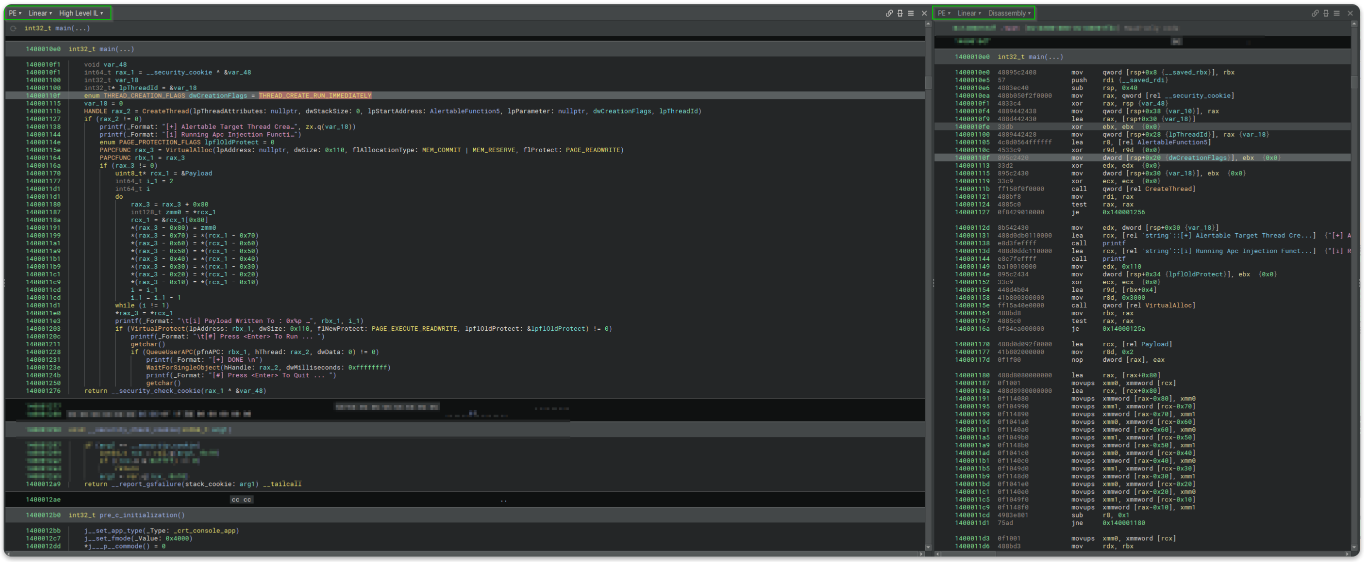
We are going to analyse APC-Injection technique, we will do it by using Binary Ninja and IDA PRO since we want to learn both decompilers usage.

Binary Ninja

Upon loading the sample into Binary Ninja, it shows us the main function of the program. We have opened 2 views, 1 HLIL and 1 Disassembly view. This is the view that I personally like the most, since the left view is meant for readability and the right view is to understand what is happening at a lower level. (ASM)



```
int32_t main(...)

1400010e0  int32_t main(...)

1400010f1      void var_48
1400010f1      int64_t rax_1 = __security_cookie ^ &var_48
140001100      int32_t var_18
140001100      int32_t* lpThreadId = &var_18
14000110f      enum THREAD_CREATION_FLAGS dwCreationFlags = THREAD_CREATE_RUN_IMMEDIATELY
140001115      var_18 = 0
14000111b      HANDLE rax_2 = CreateThread(lpThreadAttributes: nullptr, dwStackSize: 0, lpStartAddress: AlertableFunction5, lpParameter: nullptr, dwCreationFlags, lpThreadId)
140001127      if (rax_2 != 0)
140001138          printf(_Format: "[+] Alertable Target Thread Crea…", zx.q(var_18))
140001144          printf(_Format: "[i] Running Apc Injection Functi…")
14000114e          enum PAGE_PROTECTION_FLAGS lpflOldProtect = 0
14000115e          PAPCFUNC rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: 0x110, flAllocationType: MEM_COMMIT | MEM_RESERVE, flProtect: PAGE_READWRITE)
140001164          PAPCFUNC rbx_1 = rax_3
14000116a          if (rax_3 != 0)
140001170              uint8_t* rcx_1 = &Payload
140001177              int64_t i_1 = 2
1400011d1              int64_t i
1400011d1              do
140001180                  rax_3 = rax_3 + 0x80
140001187                  int128_t zmm0 = *rcx_1
14000118a                  rcx_1 = &rcx_1[0x80]
140001191                  *(rax_3 - 0x80) = zmm0
140001199                  *(rax_3 - 0x70) = *(rcx_1 - 0x70)
1400011a1                  *(rax_3 - 0x60) = *(rcx_1 - 0x60)
1400011a9                  *(rax_3 - 0x50) = *(rcx_1 - 0x50)
1400011b1                  *(rax_3 - 0x40) = *(rcx_1 - 0x40)
1400011b9                  *(rax_3 - 0x30) = *(rcx_1 - 0x30)
1400011c1                  *(rax_3 - 0x20) = *(rcx_1 - 0x20)
1400011c9                  *(rax_3 - 0x10) = *(rcx_1 - 0x10)
1400011cd                  i = i_1
1400011cd                  i_1 = i_1 - 1
1400011d1              while (i != 1)
1400011e0              *rax_3 = *rcx_1
1400011e3              printf(_Format: "\t[i] Payload Written To : 0x%p …", rbx_1, i_1)
140001203              if (VirtualProtect(lpAddress: rbx_1, dwSize: 0x110, flNewProtect: PAGE_EXECUTE_READWRITE, lpflOldProtect: &lpflOldProtect) != 0)
14000120c                  printf(_Format: "\t[#] Press <Enter> To Run ... ")
140001211                  getchar()
140001228                  if (QueueUserAPC(pfnAPC: rbx_1, hThread: rax_2, dwData: 0) != 0)
140001231                      printf(_Format: "[+] DONE \n")
14000123e                      WaitForSingleObject(hHandle: rax_2, dwMilliseconds: 0xffffffff)
14000124b                      printf(_Format: "[#] Press <Enter> To Quit ... ")
140001250                      getchar()
140001276      return __security_check_cookie(rax_1 ^ &var_48)
```

[redacted]

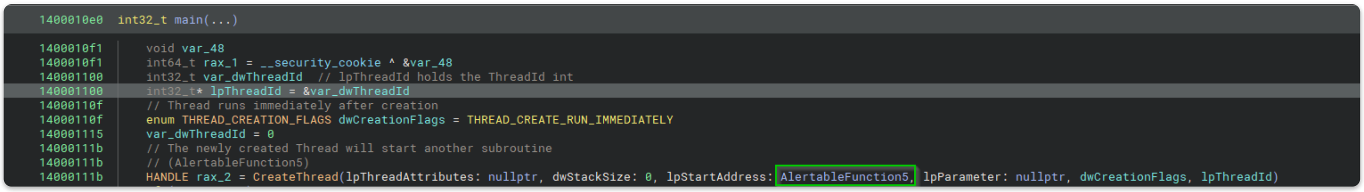
```
1400012a9      return __report_gsfailure(stack_cookie: arg1) __tailcall

1400012ae                                    cc cc                              ..

1400012b0  int32_t pre_c_initialization()

1400012bb      j__set_app_type(_Type: _crt_console_app)
1400012c7      j__set_fmode(_Value: 0x4000)
1400012dd      *j___p__commode() = 0
```

[redacted]

```
1400010e0  int32_t main(...)

1400010e0  48895c2408          mov     qword [rsp+0x8 {__saved_rbx}], rbx
1400010e5  57                  push    rdi {__saved_rdi}
1400010e6  4883ec40            sub     rsp, 0x40
1400010ea  488b050f2f0000      mov     rax, qword [rel __security_cookie]
1400010f1  4833c4              xor     rax, rsp {var_48}
1400010f4  4889442438          mov     qword [rsp+0x38 {var_10}], rax
1400010f9  488d442430          lea     rax, [rsp+0x30 {var_18}]
1400010fe  33db                xor     ebx, ebx  {0x0}
140001100  4889442428          mov     qword [rsp+0x28 {lpThreadId}], rax {var_18}
140001105  4c8d0564ffffff      lea     r8, [rel AlertableFunction5]
14000110c  4533c9              xor     r9d, r9d  {0x0}
14000110f  895c2420            mov     dword [rsp+0x20 {dwCreationFlags}], ebx  {0x0}
140001113  33d2                xor     edx, edx  {0x0}
140001115  895c2430            mov     dword [rsp+0x30 {var_18}], ebx  {0x0}
140001119  33c9                xor     ecx, ecx  {0x0}
14000111b  ff150f0f0000        call    qword [rel CreateThread]
140001121  488bf8              mov     rdi, rax
140001124  4885c0              test    rax, rax
140001127  0f8429010000        je      0x140001256

14000112d  8b542430            mov     edx, dword [rsp+0x30 {var_18}]
140001131  488d0db0110000      lea     rcx, [rel `string'::[+] Alertable Target Thread Cre...]  {"[+] A
140001138  e8d3feffff          call    printf
14000113d  488d0ddc110000      lea     rcx, [rel `string'::[i] Running Apc Injection Funct...]  {"[i] R
140001144  e8c7feffff          call    printf
140001149  ba10010000          mov     edx, 0x110
14000114e  895c2434            mov     dword [rsp+0x34 {lpflOldProtect}], ebx  {0x0}
140001152  33c9                xor     ecx, ecx  {0x0}
140001154  448d4b04            lea     r9d, [rbx+0x4]
140001158  41b800300000        mov     r8d, 0x3000
14000115e  ff15a40e0000        call    qword [rel VirtualAlloc]
140001164  488bd8              mov     rbx, rax
140001167  4885c0              test    rax, rax
14000116a  0f84ea000000        je      0x14000125a

140001170  488d0d092f0000      lea     rcx, [rel Payload]
140001177  41b802000000        mov     r8d, 0x2
14000117d  0f1f00              nop     dword [rax], eax

140001180  488d8080000000      lea     rax, [rax+0x80]
140001187  0f1001              movups  xmm0, xmmword [rcx]
14000118a  488d8980000000      lea     rcx, [rcx+0x80]
140001191  0f114080            movups  xmmword [rax-0x80], xmm0
140001195  0f104990            movups  xmm1, xmmword [rcx-0x70]
140001199  0f114890            movups  xmmword [rax-0x70], xmm1
14000119d  0f1041a0            movups  xmm0, xmmword [rcx-0x60]
1400011a1  0f1140a0            movups  xmmword [rax-0x60], xmm0
1400011a5  0f1049b0            movups  xmm1, xmmword [rcx-0x50]
1400011a9  0f1148b0            movups  xmmword [rax-0x50], xmm1
1400011ad  0f1041c0            movups  xmm0, xmmword [rcx-0x40]
1400011b1  0f1140c0            movups  xmmword [rax-0x40], xmm0
1400011b5  0f1049d0            movups  xmm1, xmmword [rcx-0x30]
1400011b9  0f1148d0            movups  xmmword [rax-0x30], xmm1
1400011bd  0f1041e0            movups  xmm0, xmmword [rcx-0x20]
1400011c1  0f1140e0            movups  xmmword [rax-0x20], xmm0
1400011c5  0f1049f0            movups  xmm1, xmmword [rcx-0x10]
1400011c9  0f1148f0            movups  xmmword [rax-0x10], xmm1
1400011cd  4983e801            sub     r8, 0x1
1400011d1  75ad                jne     0x140001180

1400011d3  0f1001              movups  xmm0, xmmword [rcx]
1400011d6  488bd3              mov     rdx, rbx
```

Analyse Main Function Part 1

By analysing the main subroutine, we are able to conclude several key points. `zq.q`, zero-extended qword (8 bytes) this means it will add the unsigned int (number) to the variable `var_dwThreadId` before we are going to dive into that subroutine, we first want to get a better understanding what `AlterableFunction5(`) does and why it's being called.



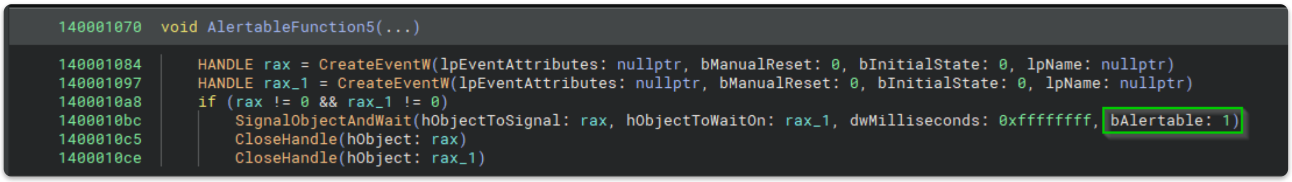
```
1400010e0  int32_t main(...)

1400010f1      void var_48
1400010f1      int64_t rax_1 = __security_cookie ^ &var_48
140001100      int32_t var_dwThreadId  // lpThreadId holds the ThreadId int
140001100      int32_t* lpThreadId = &var_dwThreadId
14000110f      // Thread runs immediately after creation
14000110f      enum THREAD_CREATION_FLAGS dwCreationFlags = THREAD_CREATE_RUN_IMMEDIATELY
140001115      var_dwThreadId = 0
14000111b      // The newly created Thread will start another subroutine
14000111b      // (AlertableFunction5)
14000111b      HANDLE rax_2 = CreateThread(lpThreadAttributes: nullptr, dwStackSize: 0, lpStartAddress: AlertableFunction5, lpParameter: nullptr, dwCreationFlags, lpThreadId)
```

Analyse AlertableFuntion5

- CreateEventW() Function Calls: The subroutine calls `CreateEventW()` twice, each time creating an event object. This function is indeed responsible for creating event objects in

Windows. The parameters passed (`lpEventAttributes`, `bManualReset`, `bInitialState`, `lpName`) specify the attributes of the event to be created.

- Comparison with 0: After the `CreateEventW()` calls, the return values (handles to the event objects) are stored in the `rax` and `rax_1` variables. The subsequent code then checks if both handles are not equal to 0. This check ensures that both `CreateEventW()` calls were successful in creating valid event objects. If either of them failed, the handle returned would be `NULL` (0), indicating failure.
- RAX : The comparison `if (rax != 0 && rax_1 != 0)` indeed checks whether the `rax` and `rax_1` registers hold valid handles to the event objects.
- SingleObjectAndWait() : This WinAPI is responsible for waiting on the event infinitely and make the event in an alterable state.

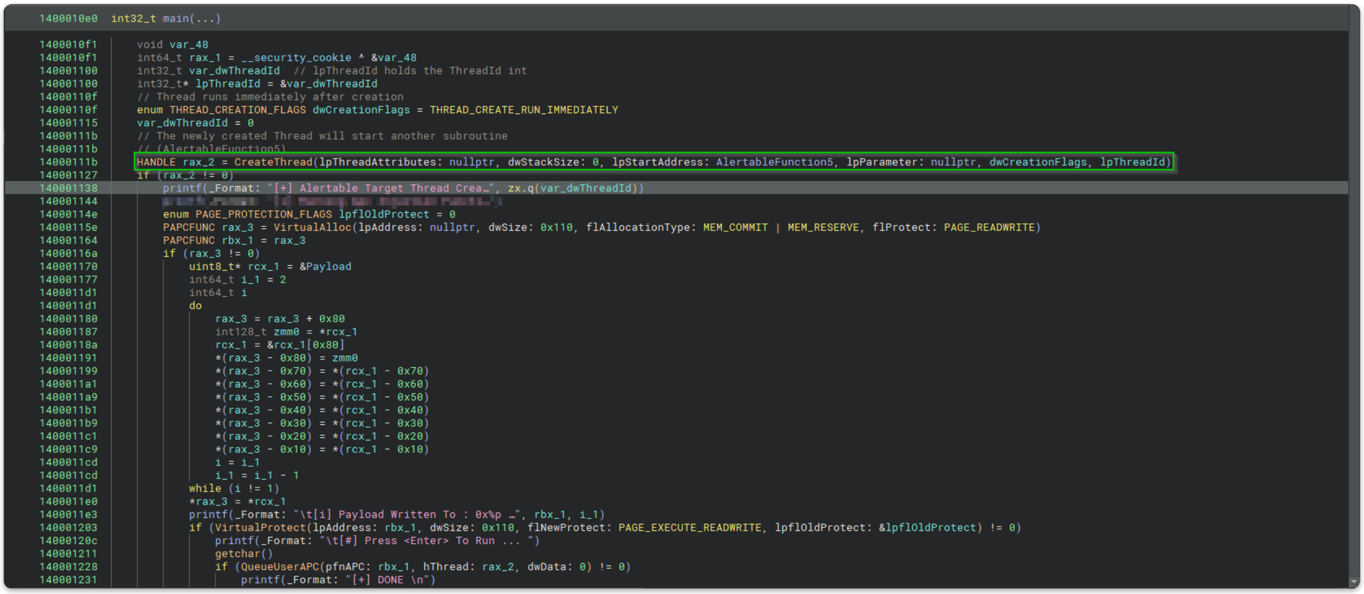


```
140001070  void AlertableFunction5(...)

140001084      HANDLE rax = CreateEventW(lpEventAttributes: nullptr, bManualReset: 0, bInitialState: 0, lpName: nullptr)
140001097      HANDLE rax_1 = CreateEventW(lpEventAttributes: nullptr, bManualReset: 0, bInitialState: 0, lpName: nullptr)
1400010a8      if (rax != 0 && rax_1 != 0)
1400010bc          SignalObjectAndWait(hObjectToSignal: rax, hObjectToWaitOn: rax_1, dwMilliseconds: 0xffffffff, bAlertable: 1)
1400010c5          CloseHandle(hObject: rax)
1400010ce          CloseHandle(hObject: rax_1)
```

We now know, that this function is responsible for creating an event in alterable mode. Since we have gained this information, we can go back to the main subroutine and further analyze it's purpose.

Analyse Main Function Part 2

Since we have analysed `AlterableFunction5`, we can now say that there will be a thread created in an alterable state.



```
1400010e0  int32_t main(...)

1400010f1      void var_48
1400010f1      int64_t rax_1 = __security_cookie ^ &var_48
140001100      int32_t var_dwThreadId  // lpThreadId holds the ThreadId int
140001100      int32_t* lpThreadId = &var_dwThreadId
14000110f      // Thread runs immediately after creation
14000110f      enum THREAD_CREATION_FLAGS dwCreationFlags = THREAD_CREATE_RUN_IMMEDIATELY
140001115      var_dwThreadId = 0
14000111b      // The newly created Thread will start another subroutine
14000111b      // (AlertableFunction5)
14000111b      HANDLE rax_2 = CreateThread(lpThreadAttributes: nullptr, dwStackSize: 0, lpStartAddress: AlertableFunction5, lpParameter: nullptr, dwCreationFlags, lpThreadId)
140001127      if (rax_2 != 0)
140001138          printf(_Format: "[+] Alertable Target Thread Crea…", zx.q(var_dwThreadId))
140001144          [redacted]
14000114e          enum PAGE_PROTECTION_FLAGS lpflOldProtect = 0
14000115e          PAPCFUNC rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: 0x110, flAllocationType: MEM_COMMIT | MEM_RESERVE, flProtect: PAGE_READWRITE)
140001164          PAPCFUNC rbx_1 = rax_3
14000116a          if (rax_3 != 0)
140001170              uint8_t* rcx_1 = &Payload
140001177              int64_t i_1 = 2
1400011d1              int64_t i
1400011d1              do
140001180                  rax_3 = rax_3 + 0x80
140001187                  int128_t zmm0 = *rcx_1
14000118a                  rcx_1 = &rcx_1[0x80]
140001191                  *(rax_3 - 0x80) = zmm0
140001199                  *(rax_3 - 0x70) = *(rcx_1 - 0x70)
1400011a1                  *(rax_3 - 0x60) = *(rcx_1 - 0x60)
1400011a9                  *(rax_3 - 0x50) = *(rcx_1 - 0x50)
1400011b1                  *(rax_3 - 0x40) = *(rcx_1 - 0x40)
1400011b9                  *(rax_3 - 0x30) = *(rcx_1 - 0x30)
1400011c1                  *(rax_3 - 0x20) = *(rcx_1 - 0x20)
1400011c9                  *(rax_3 - 0x10) = *(rcx_1 - 0x10)
1400011cd                  i = i_1
1400011cd                  i_1 = i_1 - 1
1400011d1              while (i != 1)
1400011e0              *rax_3 = *rcx_1
1400011e3              printf(_Format: "\t[i] Payload Written To : 0x%p …", rbx_1, i_1)
140001203              if (VirtualProtect(lpAddress: rbx_1, dwSize: 0x110, flNewProtect: PAGE_EXECUTE_READWRITE, lpflOldProtect: &lpflOldProtect) != 0)
14000120c                  printf(_Format: "\t[#] Press <Enter> To Run ... ")
140001211                  getchar()
140001228                  if (QueueUserAPC(pfnAPC: rbx_1, hThread: rax_2, dwData: 0) != 0)
140001231                      printf(_Format: "[+] DONE \n")
```

Thread Creation:

- A thread is created using `CreateThread`.

- The function `AlertableFunction5` is designated as the entry point.
- If thread creation is successful, the Thread ID is printed.

Memory Allocation and Payload Initialization:

- Memory is allocated using `VirtualAlloc`.
- The payload is copied into the allocated memory in blocks of 128 bytes.
- Details of payload copying and memory address are printed.

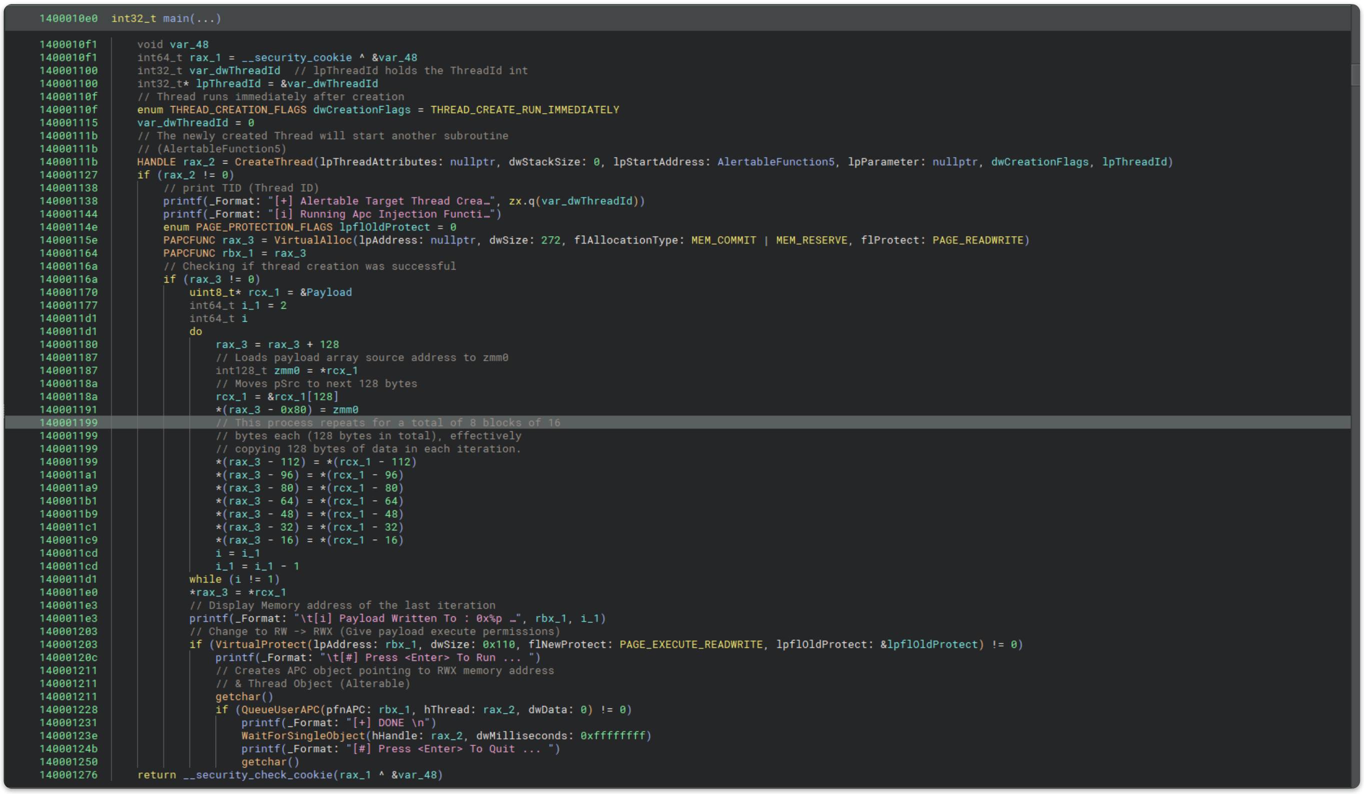
Memory Protection and Execution:

- Memory protection is changed to allow payload execution.
- User input is prompted before payload execution.
- APC object is created to execute the payload.
- Successful execution confirmation is printed.

Thread Synchronization and Completion:

- The main thread waits for the APC execution to finish.
- User prompt to exit is displayed.

The code involves creating a thread, allocating memory for payload, copying payload to memory, changing memory protection for payload execution, executing payload using APC, and waiting for execution to complete.

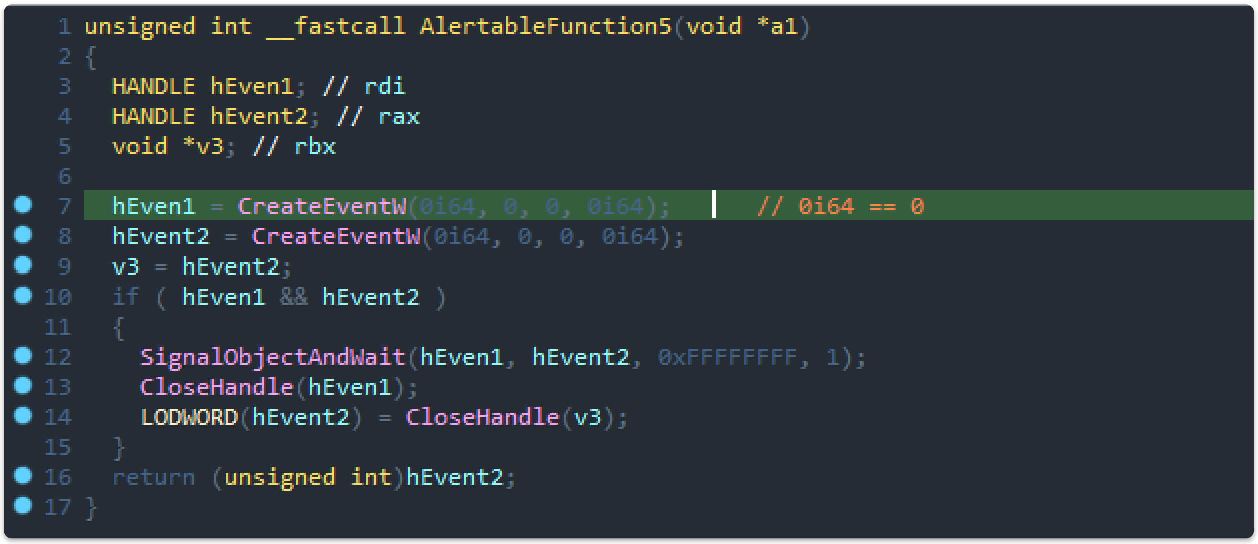


```
1400010e0  int32_t main(...)

1400010f1      void var_48
1400010f1      int64_t rax_1 = __security_cookie ^ &var_48
140001100      int32_t var_dwThreadId  // lpThreadId holds the ThreadId int
140001100      int32_t* lpThreadId = &var_dwThreadId
14000110f      // Thread runs immediately after creation
14000110f      enum THREAD_CREATION_FLAGS dwCreationFlags = THREAD_CREATE_RUN_IMMEDIATELY
140001115      var_dwThreadId = 0
14000111b      // The newly created Thread will start another subroutine
14000111b      // (AlertableFunction5)
14000111b      HANDLE rax_2 = CreateThread(lpThreadAttributes: nullptr, dwStackSize: 0, lpStartAddress: AlertableFunction5, lpParameter: nullptr, dwCreationFlags, lpThreadId)
140001127      if (rax_2 != 0)
140001138          // print TID (Thread ID)
140001138          printf(_Format: "[+] Alertable Target Thread Crea…", zx.q(var_dwThreadId))
140001144          printf(_Format: "[i] Running Apc Injection Functi…")
14000114e          enum PAGE_PROTECTION_FLAGS lpflOldProtect = 0
14000115e          PAPCFUNC rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: 272, flAllocationType: MEM_COMMIT | MEM_RESERVE, flProtect: PAGE_READWRITE)
140001164          PAPCFUNC rbx_1 = rax_3
14000116a          // Checking if thread creation was successful
14000116a          if (rax_3 != 0)
140001170              uint8_t* rcx_1 = &Payload
140001177              int64_t i_1 = 2
1400011d1              int64_t i
1400011d1              do
140001180                  rax_3 = rax_3 + 128
140001187                  // Loads payload array source address to zmm0
140001187                  int128_t zmm0 = *rcx_1
14000118a                  // Moves pSrc to next 128 bytes
14000118a                  rcx_1 = &rcx_1[128]
140001191                  *(rax_3 - 0x80) = zmm0
140001199                  // This process repeats for a total of 8 blocks of 16
140001199                  // bytes each (128 bytes in total), effectively
140001199                  // copying 128 bytes of data in each iteration.
140001199                  *(rax_3 - 112) = *(rcx_1 - 112)
1400011a1                  *(rax_3 - 96) = *(rcx_1 - 96)
1400011a9                  *(rax_3 - 80) = *(rcx_1 - 80)
1400011b1                  *(rax_3 - 64) = *(rcx_1 - 64)
1400011b9                  *(rax_3 - 48) = *(rcx_1 - 48)
1400011c1                  *(rax_3 - 32) = *(rcx_1 - 32)
1400011c9                  *(rax_3 - 16) = *(rcx_1 - 16)
1400011cd                  i = i_1
1400011cd                  i_1 = i_1 - 1
1400011d1              while (i != 1)
1400011e0              *rax_3 = *rcx_1
1400011e3              // Display Memory address of the last iteration
1400011e3              printf(_Format: "\t[i] Payload Written To : 0x%p …", rbx_1, i_1)
140001203              // Change to RW -> RWX (Give payload execute permissions)
140001203              if (VirtualProtect(lpAddress: rbx_1, dwSize: 0x110, flNewProtect: PAGE_EXECUTE_READWRITE, lpflOldProtect: &lpflOldProtect) != 0)
14000120c                  printf(_Format: "\t[#] Press <Enter> To Run ... ")
140001211                  // Creates APC object pointing to RWX memory address
140001211                  // & Thread Object (Alterable)
140001211                  getchar()
140001228                  if (QueueUserAPC(pfnAPC: rbx_1, hThread: rax_2, dwData: 0) != 0)
140001231                      printf(_Format: "[+] DONE \n")
14000123e                      WaitForSingleObject(hHandle: rax_2, dwMilliseconds: 0xffffffff)
14000124b                      printf(_Format: "[#] Press <Enter> To Quit ... ")
140001250                      getchar()
140001276      return __security_check_cookie(rax_1 ^ &var_48)
```

IDA

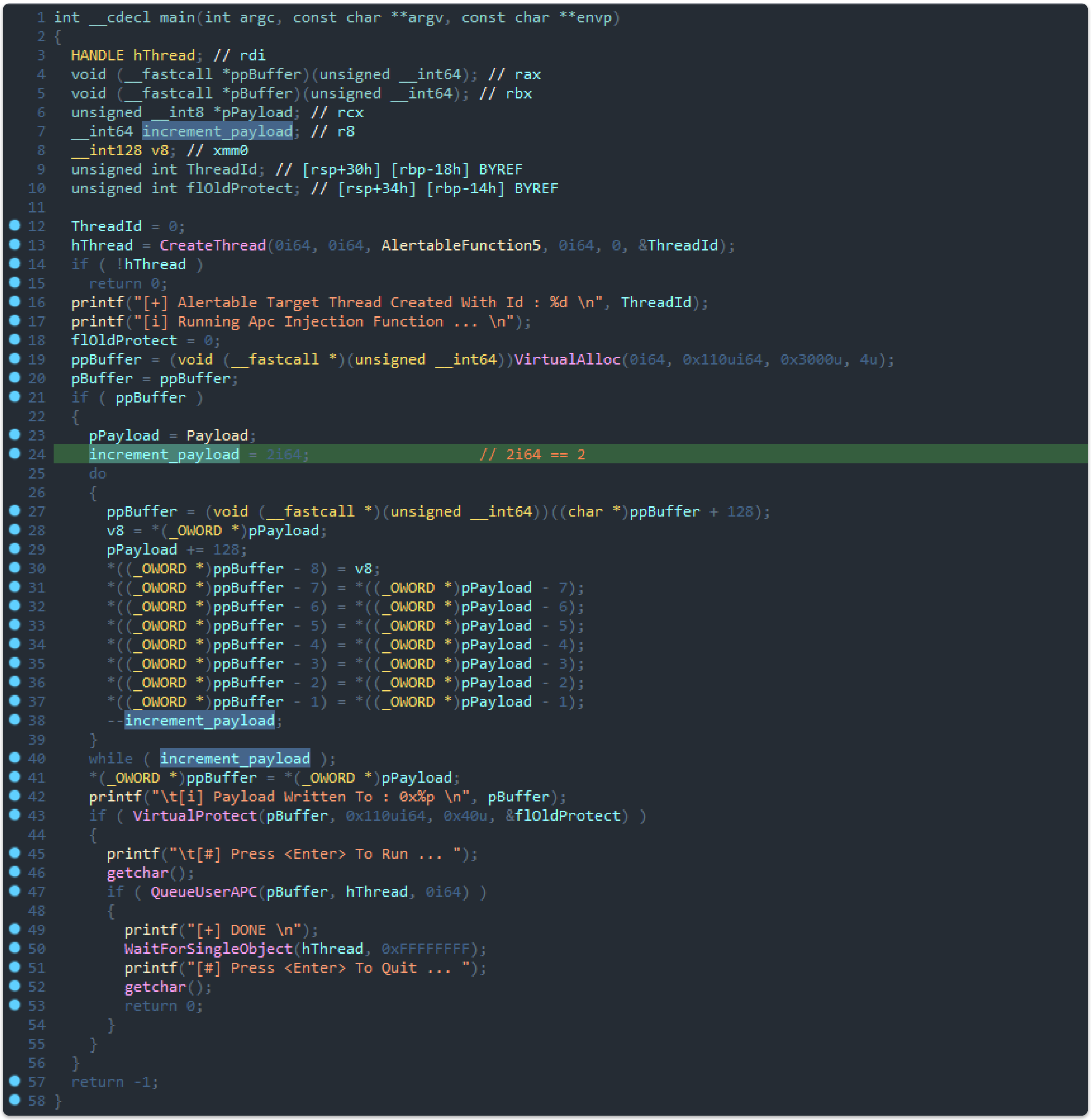
I only will show the steps without to much explanation.

AlterableFunction5



```
unsigned int __fastcall AlertableFunction5(void *a1)
{
  HANDLE hEven1; // rdi
  HANDLE hEvent2; // rax
  void *v3; // rbx

  hEven1 = CreateEventW(0i64, 0, 0, 0i64);      // 0i64 == 0
  hEvent2 = CreateEventW(0i64, 0, 0, 0i64);
  v3 = hEvent2;
  if ( hEven1 && hEvent2 )
  {
    SignalObjectAndWait(hEven1, hEvent2, 0xFFFFFFFF, 1);
    CloseHandle(hEven1);
    LODWORD(hEvent2) = CloseHandle(v3);
  }
  return (unsigned int)hEvent2;
}
```

Main Function



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HANDLE hThread; // rdi
  void (__fastcall *ppBuffer)(unsigned __int64); // rax
  void (__fastcall *pBuffer)(unsigned __int64); // rbx
  unsigned __int8 *pPayload; // rcx
  __int64 increment_payload; // r8
  __int128 v8; // xmm0
  unsigned int ThreadId; // [rsp+30h] [rbp-18h] BYREF
  unsigned int flOldProtect; // [rsp+34h] [rbp-14h] BYREF

  ThreadId = 0;
  hThread = CreateThread(0i64, 0i64, AlertableFunction5, 0i64, 0, &ThreadId);
  if ( !hThread )
    return 0;
  printf("[+] Alertable Target Thread Created With Id : %d \n", ThreadId);
  printf("[i] Running Apc Injection Function ... \n");
  flOldProtect = 0;
  ppBuffer = (void (__fastcall *)(unsigned __int64))VirtualAlloc(0i64, 0x110ui64, 0x3000u, 4u);
  pBuffer = ppBuffer;
  if ( ppBuffer )
  {
    pPayload = Payload;
    increment_payload = 2i64;                   // 2i64 == 2
    do
    {
      ppBuffer = (void (__fastcall *)(unsigned __int64))((char *)ppBuffer + 128);
      v8 = *(_OWORD *)pPayload;
      pPayload += 128;
      *((_OWORD *)ppBuffer - 8) = v8;
      *((_OWORD *)ppBuffer - 7) = *((_OWORD *)pPayload - 7);
      *((_OWORD *)ppBuffer - 6) = *((_OWORD *)pPayload - 6);
      *((_OWORD *)ppBuffer - 5) = *((_OWORD *)pPayload - 5);
      *((_OWORD *)ppBuffer - 4) = *((_OWORD *)pPayload - 4);
      *((_OWORD *)ppBuffer - 3) = *((_OWORD *)pPayload - 3);
      *((_OWORD *)ppBuffer - 2) = *((_OWORD *)pPayload - 2);
      *((_OWORD *)ppBuffer - 1) = *((_OWORD *)pPayload - 1);
      --increment_payload;
    }
    while ( increment_payload );
    *(_OWORD *)ppBuffer = *(_OWORD *)pPayload;
    printf("\t[i] Payload Written To : 0x%p \n", pBuffer);
    if ( VirtualProtect(pBuffer, 0x110ui64, 0x40u, &flOldProtect) )
    {
      printf("\t[#] Press <Enter> To Run ... ");
      getchar();
      if ( QueueUserAPC(pBuffer, hThread, 0i64) )
      {
        printf("[+] DONE \n");
        WaitForSingleObject(hThread, 0xFFFFFFFF);
        printf("[#] Press <Enter> To Quit ... ");
        getchar();
        return 0;
      }
    }
  }
  return -1;
}
```