Reverse Engineering

2. Analysis of a Program's Flow

Ing. Tomáš Zahradnický, EUR ING, Ph.D.
Ing. Josef Kokeš

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Czech Technical University in Prague
Faculty of Information Technology
Department of Information Security

Version 2020-10-08

Table of Contents I

Breaking a Function into Basic Blocks

With the stack frame analyzed and local variables identified, we can start analyzing the control flow to reconstruct at first a Control Flow Graph (CFG) and later higher level constructs such as **if-then-else** and **do-while/for/while** loops.

Creating a CFG starts with breaking the code into basic blocks.

Basic Block [MIE-GEN]

Basic Block (BB) is the maximal sequence of consecutive instructions where the flow of control can only enter and can only leave the block through the first instruction and the last instruction of the block, respectively.

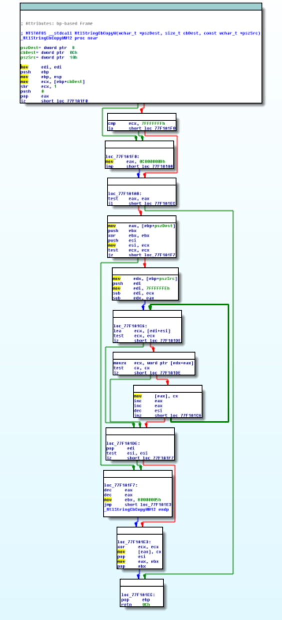
Constructing a Control Flow Graph

Once we have a function divided into basic blocks, we can construct a Control Flow Graph (CFG), where:

- each BB is a vertex;
- control flow is denoted by oriented edges between BBs.

CFG gives us an initial higher-level insight into the function being analyzed.

CFG serves as one of the inputs into a decompiler, which tries to recognize known patterns and reconstruct high-level code. Creating CFG is not always a straightforward task, especially if obfuscation techniques are used (e.g. Opaque Predicates).



If-then statements I

With a CMP/TEST Instruction

This is the simplest case of a conditional jump. The statement being analyzed looks like this:

C

```
if( condition )
  expression;
```

Assembly Pseudo Code

```
  comparison op testing the condition
  jxx skip_away
  expression
skip_away:
```

```
77e3c798 kernel32!StringCbPrintfW
...
77e3c7a0  shr  ecx, 1
77e3c7a2  push 0
77e3c7a4  pop  eax
77e3c7a5  jz   short loc_77e3c7e5

// The cmp instruction sets flags, based on the comparison result
77e3c7a7  cmp  ecx, 7fffffffh
77e3c7ad  ja   short loc_77e3c7e5
...
77e3c7e5  mov  eax, 80070057h   // HRESULT_FROM_WIN32( ERROR_INVALID_PARAMETER )
77e3c7ea  jmp  short loc_77e3c7af
```

If-then statements II

Without a CMP/TEST Instruction

This statement is the same as the previous one, except that is uses a non-comparison (e.g. arithmetic) instruction to set flags in the EFL register.

C

```
if( condition )
  expression;
```

Assembly Pseudo Code

```
  arithmetic op calculating the condition
  jxx skip_away
  expression
skip_away:
```

```
77e3c798 kernel32!StringCbPrintfW
77e3c798  mov  edi, edi         // A 2-byte NOP for hot patching
77e3c79a  push ebp
77e3c79b  mov  ebp, esp
77e3c79d  mov  ecx, [ebp+c]     // Load 2nd parameter into ECX
77e3c7a0  shr  ecx, 1           // Unsigned divide by 2, set ZF and CF
77e3c7a2  push 0
77e3c7a4  pop  eax

// The shr instruction sets/clears ZF in EFL if the result is/isn't zero
// Push and pop do not manipulate flags in EFL
77e3c7a5  jz   short loc_77e3c7e5
...
77e3c7e5  mov  eax, 80070057h   // HRESULT_FROM_WIN32( ERROR_INVALID_PARAMETER )
77e3c7ea  jmp  short loc_77e3c7af
```

If-then-else statements

This statement is the same as the previous one, except that it adds an alternative branch.

C

```c
if( condition )
  expression1;
else
  expression2;
```

Assembly Code

```
77e2a8cd kernel32!UIntPtrToInt
77e2a8cd  mov  edi, edi
77e2a8cf  push ebp
77e2a8d0  mov  ebp, esp
77e2a8d2  mov  eax, [ebp+8]
77e2a8d5  cmp  eax, 7fffffffh
77e2a8da  ja   loc_77e5547c
77e2a8e0  mov  ecx, [ebp+c]
77e2a8e3  mov  [ecx], eax
77e2a8e5  xor  eax, eax // Fall thru

77e2a8e7 loc_77e2a8e7:
77e2a8e7  pop  ebp
77e2a8e8  retn 8

77e5547c loc_77e5547c:
77e5547c  mov  eax, [ebp+c]
77e5547f  or   dword ptr [eax], 0ffffffffh
77e55482  mov  eax, 80070216h // HRESULT_FROM_WIN32( ERROR_ARITHMETIC_OVERFLOW )
77e55487  jmp  loc_77e2a8e7   // Jump back
```

Loops I

The while loop

C

```c
while( condition )
  body;
```

Assembly Code

```
00401561   push   ebp
00401562   mov    ebp,esp
00401564 loc_0401564:
00401564   mov    eax,dword ptr [ebp+c]
00401567   movsx  ecx,byte ptr [eax]
0040156a   test   ecx,ecx // Test the condition
0040156c   je     loc_040158c // Jump after the loop if failed
0040156e   mov    edx,dword ptr [ebp+8]
00401571   mov    eax,dword ptr [ebp+c]
00401574   mov    cl,byte ptr [eax]
00401576   mov    byte ptr [edx],cl
00401578   mov    edx,dword ptr [ebp+8]
0040157b   add    edx,1
0040157e   mov    dword ptr [ebp+8],edx
00401581   mov    eax,dword ptr [ebp+c]
00401584   add    eax,1
00401587   mov    dword ptr [ebp+c],eax
0040158a   jmp    loc_0401564 // Perform next iteration
0040158c loc_040158c:
0040158c   pop    ebp
0040158d   ret
```

Loops II

The do-while loop

C

```
do {
  body;
} while(condition);
```

Assembly Code

```
00401340    mov     edx,dword ptr [esp+4]
00401344    mov     eax,dword ptr [esp+8]
00401348 loc_0401348:
00401348    movzx   ecx,byte ptr [eax]
0040134b    lea     eax,[eax+1]
0040134e    mov     byte ptr [edx],cl
00401350    movzx   ecx,byte ptr [eax-1]
00401354    test    cl,cl          // Test the condition
00401356    jne     loc_0401348h // Perform next iteration
00401358    ret
```

Note that this function does not create a stack frame. The parameters are referred to directly using the `ESP` register. This approach produces a code which is both smaller and faster, but note that each `push` instruction changes the offset of parameters and local variables on the stack.

This omission of stack frames can be achieved at compile time by setting `-fomit-frame-pointer` GCC flag, or `/Oy` MSVC flag.

Loops III

For loop

C

```
for(
  initialization;
  condition;
  increment
)
{
    body;
}
```

Assembly Code

```
00401280   mov    ecx,dword ptr [esp+4]
00401284   xor    eax,eax
00401286   test   ecx,ecx            // Enter the loop?
00401288   jle    loc_040129c        // No - jump after it
0040128a   lea    ebx,[ebx]

00401290 loc_0401290:
00401290   mov    dword ptr [eax*4+403020h],eax
00401297   inc    eax                // Perform the increment
00401298   cmp    eax,ecx            // Test the condition
0040129a   jl     loc_0401290        // Perform next iteration

0040129c loc_040129c:
c0040129c  ret
```

The `for` statement can be used with any part empty. This effectively allows us to construct a `while` loop by omitting or moving both the initialization and the increment. The generated code would then be the same as in case of the `while` loop; for this reason it is not possible to determine whether the original code used `for` or `while`. The decompiler usually sticks to only one of them.

Switch I

using the cmp instruction

C

```
switch( arg ) {
  case 1:
    statement_1;
    break;

  case 2:
    statement_2;
    break;

  case 3:
    statement_3;
    break;

  case 4:
    statement_4;
    break;

  default:
    statement_n;
    break;
}
```

Assembly Code

```
8048389   mov    0x8(%ebp),%eax    // Load arg. into EAX
804838c   cmp    $0x2,%eax
804838f   je     0x80483df
8048391   jle    0x80483c0
8048393   cmp    $0x3,%eax
8048396   je     0x80483b2
8048398   cmp    $0x4,%eax
804839b   nop
804839c   lea    0x0(%esi,%eiz,1),%esi
80483a0   jne    0x80483d1
80483a2   movl   $0x8048595,(%esp)
80483a9   call   0x8048350 <puts@plt>
80483ae   xor    %eax,%eax         // Return 0;
80483b0   leave
80483b1   ret
80483b2   movl   $0x8048589,(%esp)
80483b9   call   0x8048350 <puts@plt>
80483be   jmp    0x80483ae
80483c0   dec    %eax              // Dec/sub instruction commonly used!
80483c1   jne    0x80483d1
80483c3   movl   $0x8048570,(%esp)
80483ca   call   0x8048350 <puts@plt>
80483cf   jmp    0x80483ae
80483d1   movl   $0x80485a1,(%esp)
80483d8   call   0x8048350 <puts@plt>
80483dd   jmp    0x80483ae
80483df   movl   $0x804857e,(%esp)
80483e6   call   0x8048350 <puts@plt>
80483eb   jmp    0x80483ae
```

Switch II

using sub/dec

C

```
switch( arg ) {
  case 1:
    statement_1;
    break;

  case 2:
    statement_2;
    break;

  case 4:
    statement_4;
    break;

  default:
    statement_n;
    break;
}
```

Assembly Code

```
00401000  mov   eax, [esp+4]
00401004  dec   eax
00401005  jz    short loc_401042
00401007  dec   eax
00401008  jz    short loc_401031
0040100A  sub   eax, 2
0040100D  jz    short loc_401020
0040100F  push  offset defaultCase
00401014  call  ds:printf
0040101A  add   esp, 4
0040101D  xor   eax, eax          // return 0, duplicated
0040101F  retn                    // Function exit, duplicated
00401020  push  offset threeParameters
00401025  call  ds:printf
0040102B  add   esp, 4
0040102E  xor   eax, eax          // return 0, duplicated
00401030  retn                    // Function exit, duplicated
00401031  push  offset oneParameter
00401036  call  ds:printf
0040103C  add   esp, 4
0040103F  xor   eax, eax          // return 0, duplicated
00401041  retn                    // Function exit, duplicated
00401042  push  offset noParameter
00401047  call  ds:printf
0040104D  add   esp, 4
00401050  xor   eax, eax          // return 0, duplicated
00401052  retn                    // Function exit, duplicated
```

Switch III

using a jump table

C

```
switch( arg ) {
  case 1:
    statement_1;
    break;

  case 2:
    statement_2;
    break;

  case 3:
    statement_3;
    break;

  case 4:
    statement_4;
    break;

  default:
    statement_n;
    break;
}
```

Jump Table

```
00401068 dd offset loc_401011
0040106C dd offset loc_401022
00401070 dd offset loc_401033
00401074 dd offset loc_401044
```

Assembly Code

```
00401000  mov   eax, [esp+4]
00401004  dec   eax
00401005  cmp   eax, 3
00401008  ja    short loc_401055
0040100A  jmp   ds:off_401068[eax*4]   // Jump to a table item
00401011  push  offset zeroParameters
00401016  call  ds:printf
0040101C  add   esp, 4
0040101F  xor   eax, eax           // return 0, duplicated
00401021  retn                     // Function exit, duplicated
00401022  push  offset oneParameter
00401027  call  ds:printf
0040102D  add   esp, 4
00401030  xor   eax, eax           // return 0, duplicated
00401032  retn                     // Function exit, duplicated
00401033  push  offset twoParameters
00401038  call  ds:printf
0040103E  add   esp, 4
00401041  xor   eax, eax           // return 0, duplicated
00401043  retn                     // Function exit, duplicated
00401044  push  offset threeParameters
00401049  call  ds:printf
0040104F  add   esp, 4
00401052  xor   eax, eax           // return 0, duplicated
00401054  retn                     // Function exit, duplicated
00401055  push  offset defaultCase
0040105A  call  ds:printf
00401060  add   esp, 4
00401063  xor   eax, eax           // return 0, duplicated
00401065  retn                     // Function exit, duplicated
```

Summary

Now we should:

- understand the prologue and epilogue;
- understand the stack frame and its structure;
- understand what a Basic Block is;
- be able to construct a Control Flow Graph of a function, and
- understand how C constructs are compiled into assembly and be able to translate them back into a human-readable code at a higher level of abstraction.

Now, let's explore what the runtime does when a program is run. This includes:

- what the entry point is and what it does;
- calling the initializer functions;
- calling the `main` function;
- calling the terminator functions.

The Main Entry Point

The `main`/`wmain`/`_tmain` functions **are not** the real entry points. The real entry point is the function whose Relative Virtual Address (RVA) is specified in the `AddressOfEntryPoint` field in the PE optional header.

Entry point, where are you?

```
// Get HMODULE of the current process
HMODULE            hSelf = GetModuleHandle(NULL); // = LoadLibrary, ...

// Cast HMODULE to a pointer to the PE image start.  Points to MZ... - the DOS header
PIMAGE_DOS_HEADER   pDosHeader = (PIMAGE_DOS_HEADER)hSelf;

// New header is located e_lfanew bytes from the start of the image
PIMAGE_NT_HEADERS32 pNTHeaders = (PIMAGE_NT_HEADERS)(((BYTE*)pDosHeader) + pDosHeader->e_lfanew);

// Ultimately find the entry point relative virtual address and add it the the image base
void*               pfnEntryPoint = (void*)(((BYTE*)pDosHeader)
                                    + pNTHeaders->OptionalHeader.AddressOfEntryPoint);

printf("Entry point address:  %p\n", pfnEntryPoint);
```

```
Entry point address:  00402390
```

Now we should ask **who** provides the main entry point, **where is it called from**, and **why** this is not the `main` function?

Who calls the Main Entry Point?

We have already seen this in Lec. 1.

```
Entry point address:  00402390
```

Main entry point Call Stack

```
004136C0  Tokens.exe!main(int argc, const char* * argv) // The C entry point
00402259    Tokens.exe!__tmainCRTStartup()
0040239D      Tokens.exe!mainCRTStartup()                // The main entry point
75C4EE0A        kernel32.dll!@BaseThreadInitThunk@12()
775A37C4          ntdll.dll!___RtlUserThreadStart@8()
775A37A3            ntdll.dll!__RtlUserThreadStart@8()
```

Who provides the main entry point?

It's the Runtime who provides the main entry point. It can be found in `crtexe.c`:

```
int mainCRTStartup( void )
{
  /*
   * The /GS security cookie must be initialized before any exception
   * handling targetting the current image is registered.  No function
   * using exception handling can be called in the current image until
   * after __security_init_cookie has been called.
   */
  __security_init_cookie();

  return __tmainCRTStartup();
}
```

Notes

This function can have various names depending on the setup (whether the "Use Unicode Character Set" is specified aka. `#define UNICODE 1`) or whether `main` or `WinMain` are used. These names include:

1. mainCRTStartup
2. wmainCRTStartup
3. WinMainCRTStartup
4. wWinMainCRTStartup

__tmainCRTStartup()

Each MSVC-compiled program starts with this code:

```
__declspec(noinline) int __tmainCRTStartup( void )
{
  __try {
    ...
    // Run initializers placed into .crt$xia ... .crt$xiz segs (merged into .rdata)
    // __xi_a and __xi_z bound initializer data start and end
    // calls pre_c_init(), by default initialize C, sets default FPU mode,
    // sets the unhandled exception filter to __CxxUnhandledExceptionFilter
    if( _initterm_e( __xi_a, __xi_z ) != 0 )
      return 255;

    // Run initializers placed into .crt$xca ... .crt$xcz segs (placed into .rdata)
    // calls pre_cpp_init(), sets atexit(_RTC_Terminate), prepares parameters for main,
    // calls all constructors of static objects and registers a stub calling appropriate
    // destructors using the atexit function.
    _initterm( __xc_a, __xc_z );

    // Call whichever main function we have!
    mainret = main(argc, argv, envp);
    ...
    exit(mainret);
  }
  __except( _XcptFilter( GetExceptionCode(), GetExceptionInformation() ) ) {
    // _XcptFilter terminates, inaccessible
    mainret = GetExceptionCode();
    ExitProcess(mainret);
  }
  return mainret;
}
```

Initialization Code

The compiler can do a lot for us without involving us in runtime details. In order to use a global C++ class, all we have to do is:

A Sample Initializer Code

```
// A globally initialized class
Initializer g_InitializerClassInstance;

class Initializer {
  public:
    Initializer() {
      printf("Hello pre-main code.\n");
    }

    ~Initializer() {
      printf("Hello post-main code.\n");
    }
};
```

Note: GCC uses special keywords `__attribute__((constructor))` and `__attribute__((destructor))`; this allows initializers to be used from C and get a function called during the initialization or termination phase.

Initializing the Hard Way

If we need a fine-grained control, we need to use `#pragma`s.

Fine Grained Initialization

```
#pragma section(".CRT$XIB")
__declspec(allocate(".CRT$XIB")) int (*g_MyInit_PreC)(void) = MyInit_PreC;

#pragma section(".CRT$XIY")
__declspec(allocate(".CRT$XIY")) int (*g_MyInit_PostC)(void) = MyInit_PostC;

// XCT = pre static objects constructors
// XCU = post static objects constructors
#pragma section(".CRT$XCB")
__declspec(allocate(".CRT$XCB")) void (*g_MyInit_PreCPP)(void) = MyInit_PreCPP;

#pragma section(".CRT$XCZ")
__declspec(allocate(".CRT$XCZ")) void (*g_MyInit_PostCPP)(void) = MyInit_PostCPP;
```

Initializers in `.CRT$XIxxx` sections can return a value. Returning a non-zero value causes the program initialization to abort with error 255.

Where are the Initializers? I

The `_initterm_e` and `_initterm` functions take the beginning and end pointer of a portion of the `.rdata` section of the image (following the IAT). Let's inspect this section:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000540	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000550	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000610	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000620	00	00	00	00	00	00	00	00	40	39	41	00	00	00	00	00	........@9A.....
00000630	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000710	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000720	00	00	00	00	00	00	00	00	00	00	00	00	8C	10	41	00	............▌+A.
00000730	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000820	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000830	C3	10	41	00	00	00	00	00	00	00	00	00	00	00	00	00	Ã+A.............
00000840	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000920	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000930	00	00	00	00	B3	11	41	00	22	11	41	00	00	00	00	00	....³◄A."◄A.....
00000940	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000A30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000A40																	

Figure: `__xi_a` through `__xi_z` in debug version.

Each entry in this list is a pointer to a C-style initialization function:

413940	`pre_c_init`
41108c	`jmp MyInit_PreC` in debug, respectively `MyInit_PreC` in release
4110c3	`jmp __atonexitinit` in debug, resp. `__atonexitinit` in release
4111b3	`jmp MyInit_PostC` in debug, resp. `MyInit_PostC` in release
411122	`jmp ___CxxSetUnhandledExceptionFilter` in debug, resp. direct jump in release

Where are the Initializers? II

```
int _initterm_e(int (**pSegStart)(void), int (**pSegEnd)(void))
{
  int initResult = 0;

  while ( pSegStart < pSegEnd && !initResult )
  {
    if ( *pSegStart )
      initResult = (*pSegStart)();

    ++pSegStart;
  }

  return initResult;
}
```

The array holds pointers to initialization functions; if non-null, the function is called. Note that this C-initializer function aborts as soon as any initializer returns a non-zero value (unlike the C++ initializer).

Where are the Initializers? III

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000080	86	25	00	00	98	25	00	00	DE	23	00	00	F0	23	00	00	▌%..▌%..Þ#..ð#..
00000090	E8	23	00	00	86	24	00	00	00	00	00	00	00	00	00	00	è#..▌$..........
000000A0	DD	12	40	00	30	10	40	00	00	10	40	00	30	10	40	00	Ý↕@.0┼@..┼@.0┼@.
000000B0	00	00	00	00	00	00	00	00	24	12	40	00	20	10	40	00	........$↕@. ┼@.

Figure: `__xc_a` through `__xc_z` in release version.

Each entry in this list is a pointer to a C++-style initialization function:

```
000000    -
4012dd    pre_cpp_init
401030    MyInit_PreCPP
401000    Initializer ctor stub
401020    MyInit_PostCPP
```

```
00401000  mov   ecx,4031B8h // This pointer
00401005  call  Initializer::Initializer
0040100A  push  401930h     // Address of the destructor stub
0040100F  call  atexit      // Register to be called at exit
00401014  pop   ecx         // Stack cleanup
00401015  ret
...
00401930  mov   ecx,4031B8h // This pointer
00401935  jmp   Initializer::~Initializer
```

Terminators I

As we have seen, the `atexit` API was used to register stubs to call static object's destructors at the program's exit time. `Atexit` uses a dynamically allocated array pointed to by two encoded pointer globals `__onexitbegin` and `__onexitend`. Encoded function pointers are registered in this array. When the C API `exit` (but not `TerminateProcess`) is called, `atexit`-registered functions are called. Moreover, if we are using a statically linked runtime library, pre-termination and termination functions registered in `.crt$xpXXX` and `.crt$xtXXX` segments are called in the same manner as the initializers.

Terminators II

atexit code (atonexit.c)

```
int __cdecl atexit(void (__cdecl *func)()) {
  return _onexit(func) == NULL ? -1 :  0;
}

_onexit_t __cdecl _onexit(_onexit_t Func) {
  int (__cdecl *pfnFunc)();

  _lockexit();
  pfnFunc = _onexit_nolock(Func);
  _unlockexit();
  return pfnFunc;
}
```

exit code (crt0dat.c)

```
void __cdecl exit( int code ) {
  doexit(code, 0, 0); /* full term, kill process */
}
```

Terminators III

doexit pseudocode (crt0dat.c)

```
void __cdecl doexit(int code, int quick, int retcaller) {
  if ( !retcaller && check_managed_app() ) // If the process is managed, call CorExitProcess
    __crtCorExitProcess(uExitCode);
  ...
  if ( !quick ) {
    onexitbegin = DecodePointer(__onexitbegin); onexitend = DecodePointer(__onexitend);
    while ( 1 ) { // Iterate over all exit functions
      // Find the first "non-zero" func
      while ( --onexitend >= onexitbegin && *onexitend == EncodePointer(NULL) );
      // Ending condition
      if ( onexitend < pfn__onexitbegin ) break;
      // Decode, call, and remove the atexit registered function from the list
      pfnExitProc = (void (*)(void))DecodePointer(*onexitend);
      *onexitend = EncodePointer(0);
      pfnExitProc();
      ...
    }
    #ifndef CRTDLL
      _initterm(__xp_a, __xp_z); // Call pre-terminators
    #endif
  }
  #ifndef CRTDLL
    _initterm(__xt_a, __xt_z); // Call terminators
  #endif
  ...
  if ( ret ) return;
  if ( !ret ) __crtExitProcess(code);
}
```

Pointer Encoding

Pointers on the stack/heap could be overwritten and used to run exploit code. `EncodePointer` and `DecodePointer` APIs are used to make this difficult. These calls are internally mapped to `RtlEncodePointer` and `RtlDecodePointer` APIs in NTDLL.DLL:

```
77F1A290    db 5 dup(90h)
77F1A295 RtlEncodePointer:
77F1A295    mov  edi, edi
77F1A297    push ebp
77F1A298    mov  ebp, esp
77F1A29A    push ecx
77F1A29B    push 0                 ; ReturnLength
77F1A29D    push 4                 ; ProcessInformationLength
77F1A29F    lea  eax, [ebp+ProcessInformation]
77F1A2A2    push eax               ; ProcessInformation
77F1A2A3    push 24h               ; ProcessInformationClass = process cookie
77F1A2A5    push 0FFFFFFFFh        ; ProcessHandle = GetCurrentProcess()
77F1A2A7    call _ZwQueryInformationProcess@20
77F1A2AC    test eax, eax
77F1A2AE    jl   loc_77F4276F
77F1A2B4    mov  eax, [ebp+ProcessInformation]
77F1A2B7    mov  cl, al
77F1A2B9    xor  eax, [ebp+arg_0]
77F1A2BC    and  cl, 1Fh
77F1A2BF    ror  eax, cl
77F1A2C1    leave
77F1A2C2    retn 4
```

Hot Patching Support

You might have noticed there's a `mov edi,edi` instruction at the beginning of the previous function. Moreover there are 5 `nop` instructions before the function's start. These serve for the purpose of hot patching, a mechanism allowing us to easily replace the function at runtime without having to restart the application. The instructions provide 7 bytes of free space which we can use to replace the function. The `mov edi,edi` instruction is replaced by a `jmp short` instruction to the start of nops, where `jmp [addr]` instruction is placed. The code then looks like this:

Non-patched code

```
     90   nop
     90   nop
     90   nop
     90   nop
     90   nop
function_start:
  8B FF   mov edi,edi
     55   push ebp
  8B EC   mov ebp,esp
```

Patched code

```
jump_to_patched_function:
  E9 xx xx xx xx   jmp dword ptr [&patched_function]
function_start:
           EB F9   jmp short jump_to_patched_function
              55   push ebp     // Inaccessible
           8B EC   mov ebp,esp // Inaccessible
```

Call to a jmp? I

In debug version all functions were called via an extra level of indirection. This indirection was common to all functions. What was the point of that?

```
main:
00413130  push  ebp
00413131  mov   ebp,esp
00413133  sub   esp,48h
00413136  push  ebx
00413137  push  esi
00413138  push  edi
00413139  mov   dword ptr [result],0
00413140  call  PEDUMP (4110BEh)

...
PEDUMP_real:
004125F0  push  ebp
004125F1  mov   ebp,esp
004125F3  sub   esp,50h
...
```

```
_GetCurrentProcess@0:
004110B4  jmp   GetCurrentProcess (4131D0h)

___report_securityfailure:
004110B9  jmp   __report_securityfailure (413640h)

PEDUMP:
004110BE  jmp   PEDUMP_real (4125F0h)

___atonexitinit:
004110C3  jmp   __atonexitinit (413220h)

___report_securityfailureEx:
004110C8  jmp   __report_securityfailureEx (413750h)

__FindPESection:
004110CD  jmp   _FindPESection (413FC0h)

MyInit_PreCPP:
004110D2  jmp   MyInit_PreCPP (412340h)

_LoadLibraryW@4:
004110D7  jmp   LoadLibraryW (4131EEh)

__configthreadlocale:
004110DC  jmp   _configthreadlocale (4143EEh)

Initializer:: Initializer:
004110E1  jmp   Initializer:: Initializer (412390h)
```

Call to a jmp? II

When incremental linking is enabled, all calls in the program are made via an extra level of indirection. This allows us to **replace any function at program run-time**. New code is simply compiled, copied into the process's memory and the redirection address is updated so that it points to the new implementation. MSVC uses "Apply Code Changes", Apple uses "Fix and Continue", but the principle is the same.

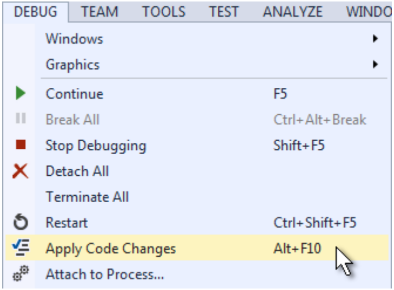


Figure: MSVC Apply Code Changes.

Import Address Table I

A module can depend on other modules. The modules depended on and symbol names and/or their ordinal numbers can be found in the Import Directory (ID) of the PE file.

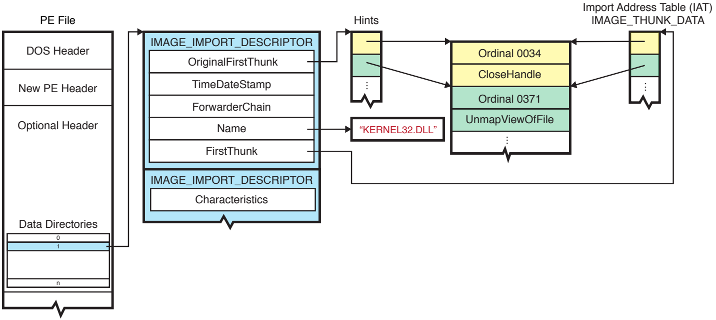


Figure: The PE Import Directory and Import Address Table.

Import Address Table II

Traversing the Import Directory

```
PIMAGE_IMPORT_DESCRIPTOR pImports = (PIMAGE_IMPORT_DESCRIPTOR)(((BYTE*)pDosHeader)
  + pNTHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);

PIMAGE_IMPORT_DESCRIPTOR pImportsEnd = (PIMAGE_IMPORT_DESCRIPTOR)(((BYTE*)pImports)
  + pNTHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size);

for (; pImports < pImportsEnd && pImports->OriginalFirstThunk != NULL; ++pImports)
{
  char* pszDLLName = (char*)((BYTE*)pDosHeader + pImports->Name);
  printf("DLL: %s\n", pszDLLName );

  if (pImports->Characteristics != NULL)
  {
    PIMAGE_THUNK_DATA pSymbolData = (PIMAGE_THUNK_DATA)(((BYTE*)pDosHeader) + pImports->OriginalFirstThunk);

    for (; pSymbolData->u1.AddressOfData != NULL; ++pSymbolData )
    {
      // note:  we should check the import type first...
      PIMAGE_IMPORT_BY_NAME pImport = (PIMAGE_IMPORT_BY_NAME)(((BYTE*)pDosHeader)
        + pSymbolData->u1.AddressOfData);

      printf("%04hx %s\n", pImport->Hint, pImport->Name);
    }
  }
}
```

Import Address Table III

Once an image is loaded, the loader replaces all IAT references (RVAs of `IMAGE_THUNK_DATA32` in a 32-bit PE file) (on the left) pointing to `IMAGE_IMPORT_BY_NAME` (bottom right) with real function pointers (on the right).

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00001200  8A 24 00 00 78 24 00 00 68 24 00 00 58 24 00 00  Š$..x$..h$..X$..
00001210  1C 27 00 00 06 27 00 00 EC 26 00 00 D0 26 00 00  .'...'..ì&..Ð&..
00001220  BC 26 00 00 AC 26 00 00 9C 26 00 00 32 27 00 00  ¼&..¬&..œ&..2'..
00001230  00 00 00 00 DC 24 00 00 E4 24 00 00 EE 24 00 00  ....Ü$..ä$..î$..
00001240  FC 24 00 00 0A 25 00 00 22 25 00 00 3C 25 00 00  ü$...%.."%..<%..
00001250  D2 24 00 00 58 25 00 00 68 25 00 00 7A 25 00 00  Ò$..X%..h%..z%..
00001260  82 25 00 00 8A 25 00 00 94 25 00 00 AA 25 00 00  ‚%..Š%..”%..ª%..
00001270  BE 25 00 00 CC 25 00 00 D8 25 00 00 E4 25 00 00  ¾%..Ì%..Ø%..ä%..
00001280  EE 25 00 00 FA 25 00 00 10 26 00 00 2A 26 00 00  î%..ú%...&..*&..
00001290  42 26 00 00 56 26 00 00 7A 26 00 00 8C 26 00 00  B&..V&..z&..Œ&..
000012A0  AC 24 00 00 C8 24 00 00 C0 24 00 00 B6 24 00 00  ¬$..È$..À$..¶$..
000012B0  4A 25 00 00 00 00 00 00 00 00 00 00 76 16 40 00  J%..........v.@.
000012C0  30 10 40 00 00 10 40 00 30 10 40 00 00 00 00 00  0.@...@.0.@.....
000012D0  00 00 00 00 BD 15 40 00 20 10 40 00 91 14 40 00  ....½.@. .@.‘.@.
000012E0  20 10 40 00 29 1A 40 00 00 00 00 00 00 00 00 00   .@.).@.........
000012F0  00 00 00 00 10 A2 FC 54 00 00 00 00 02 00 00 00  .....¢üT........
00001300  73 00 00 00 68 22 00 00 68 14 00 00 00 00 00 00  s...h"..h.......
00001310  10 A2 FC 54 00 00 00 00 0C 00 00 00 14 00 00 00  .¢üT............
00001320  DC 22 00 00 DC 14 00 00 48 65 6C 6C 6F 20 70 72  Ü"..Ü...Hello pr
00001330  65 2D 6D 61 69 6E 20 63 6F 64 65 2E 0A 00 00 00  e-main code.....
```

Figure: IAT on disk.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00402000  5C CD C4 75 DD 2C C4 75 62 DD C4 75 90 CE C4 75  \ÍÄuÝ,ÄubÝÄu.ÎÄu
00402010  C0 C4 C4 75 65 D8 C4 75 D2 C4 C4 75 D6 6C C5 75  ÀÄÄueØÄuÒÄÄuÖlÅu
00402020  E2 7E C4 75 10 CD 59 77 95 A2 59 77 C6 D8 C4 75  â~Äu.ÍYw•¢YwÆØÄu
00402030  00 00 00 00 D7 ED A5 6B FC ED A5 6B 08 12 A6 6B  ....×í¥küí¥k..¦k
00402040  46 CA A6 6B 6B BE A7 6B AA 2A A6 6B ED 4C AF 6B  FÊ¦kk¾§kª*¦kíL¯k
00402050  73 22 A6 6B 5F E2 A7 6B CE C7 A7 6B 93 42 A8 6B  s"¦k_â§kÎÇ§k“B¨k
00402060  B8 BB AC 6B 04 41 A8 6B EB 55 AF 6B E9 B9 AC 6B  ¸»¬k.A¨këU¯ké¹¬k
00402070  86 CC A6 6B 50 CC A6 6B 2C F6 B2 6B 40 F7 B2 6B  †Ì¦kPÌ¦k,ö²k@÷²k
00402080  38 F6 B2 6B B8 6B AF 6B 0C 48 AF 6B F7 47 AF 6B  8ö²k¸k¯k.H¯k÷G¯k
00402090  2C DC AE 6B DB C7 A7 6B 9B 46 AF 6B B5 C9 A7 6B  ,Ü®kÛÇ§k›F¯kµÉ§k
004020A0  D9 2F AD 6B 30 ED A5 6B E0 EC A5 6B FF CB A6 6B  Ù/.k0í¥kàì¥kÿË¦k
004020B0  8D BB AC 6B 00 00 00 00 00 00 00 00 76 16 40 00  .»¬k........v.@.
004020C0  30 10 40 00 00 10 40 00 30 10 40 00 00 00 00 00  0.@...@.0.@.....
004020D0  00 00 00 00 BD 15 40 00 20 10 40 00 91 14 40 00  ....½.@. .@.‘.@.
004020E0  20 10 40 00 29 1A 40 00 00 00 00 00 00 00 00 00   .@.).@.........
004020F0  00 00 00 00 10 A2 FC 54 00 00 00 00 02 00 00 00  .....¢üT........
00402100  73 00 00 00 68 22 00 00 68 14 00 00 00 00 00 00  s...h"..h.......
00402110  10 A2 FC 54 00 00 00 00 0C 00 00 00 14 00 00 00  .¢üT............
00402120  DC 22 00 00 DC 14 00 00 48 65 6C 6C 6F 20 70 72  Ü"..Ü...Hello pr
00402130  65 2D 6D 61 69 6E 20 63 6F 64 65 2E 0A 00 00 00  e-main code.....
```

Figure: IAT in memory.

The first `IMAGE_THUNK_DATA` (4 bytes) contains a RVA of `0000248A` (left). This address, when mapped into the executable's memory,

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00402480  6C 50 72 6F 74 65 63 74 00 00 67 02 47 65 74 4D  lProtect..g.GetM
00402490  6F 64 75 6C 65 48 61 6E 64 6C 65 57 00 00 4B 45  oduleHandleW..KE
004024A0  52 4E 45 4C 33 32 2E 64 6C 6C 00 00 FD 06 70 72  RNEL32.dll..ý.pr
```

points to an `IMAGE_IMPORT_BY_NAME` structure containing an ordinal number (`0267`) followed by a zero-terminated function name (`GetModuleHandleW`) (bottom right). The function is from `kernel32.dll` and resolves to address `75C4CD5C`, which is written back to the IAT (right). The IAT is found in the `.rdata` section, thus it is **read-only**.

Import Address Table IV

When an external symbol is bound, its address is resolved and all references to that symbol are updated according to the relocation table (see BIE-BEK, lec. 2). This approach is used for object files; with executables, we would have to relocate every external call during the application's startup. Instead, indirection is used and only **a single entry** per symbol is modified — the one in the IAT!

Calling a function through the IAT

```
// A function call
00401072  push  0
00401074  call  dword ptr [__imp__GetModuleHandleW@4 (402000h)]

// IAT
00402000  .dd 075C4CD5Ch     // KERNEL32.DLL!GetModuleHandleW
00402004  .dd 075C42CDDh     // KERNEL32.DLL!VirtualProtect
...
004020B0  .dd 06BACBB8Dh     // MSVCR120.DLL!__amsg_exit
004020B4  .dd 0              // - end -

// Implementation
_GetModuleHandleWStub@4:
75C4CD5C  mov   edi,edi
75C4CD5E  push  ebp
75C4CD5F  mov   ebp,esp
```

IAT Hacking I

As we have seen, calls to the same external API are made through a single point in the program — its entry in the IAT. What happens if we change that entry in the IAT?

IAT Hacking II

As we have seen, calls to the same external API are made through a single point in the program — its entry in the IAT. What happens if we change that entry in the IAT?

All calls to that API throughout the entire program would be redirected!

IAT Hacking III

As we have seen, calls to the same external API are made through a single point in the program — its entry in the IAT. What happens if we change that entry in the IAT?

All calls to that API throughout the entire program would be redirected!

The IAT is read-only by default, but it is possible to use e.g. `VirtualProtect` to make it writable and redirect all calls to that function throughout the entire program to our function.

```
BOOL WINAPI VirtualProtect(
  LPVOID lpAddress,      // IAT_start
  SIZE_T dwSize,         // ( IAT_size + page_size - 1 ) & (  page_size - 1 )
  DWORD flNewProtect,    // PAGE_READWRITE
  PDWORD lpflOldProtect
);
```

IAT Hacking IV

Can we get beyond the process boundary? The answer is a definite YES. We can either use functions such as `VirtualAllocEx`, `ReadProcessMemory`, `WriteProcessMemory`, `VirtualProtectEx`, `CreateRemoteThread` to hack another process(es) and to inject code/DLL into them, or use the Native API!

Win32 API

```
BOOL WINAPI VirtualProtectEx(
  _In_ HANDLE hProcess,
  _In_ LPVOID lpAddress,
  _In_ SIZE_T dwSize,
  _In_ DWORD flNewProtect,
  _Out_ PDWORD lpflOldProtect
);
```

NT API

```
NTSYSAPI NTSTATUS NTAPI
NtProtectVirtualMemory(
  IN HANDLE ProcessHandle,
  IN OUT PVOID *BaseAddress,
  IN OUT PULONG NumOfBytesToProtect,
  IN ULONG NewAccessProtection,
  OUT PULONG OldAccessProtection
);
```

Bibliography

Russinovich M., Solomon D. A., Ionescu A.: *Windows Internals Part 1*, 6$^{\mathrm{th}}$ ed., 2012.

Russinovich M., Solomon D. A., Ionescu A.: *Windows Internals Part 2*, 6$^{\mathrm{th}}$ ed., 2012.