

# Reverzní inženýrství

## 2. Analýza toku kódu

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

Ing. Josef Kokeš, Ph.D.



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra informační bezpečnosti

Verze 2. září 2023

# Obsah I

## 1 Analýza kódu

- Analýza toku kódu

## 2 Vstupní bod, inicializace a ukončení

- Hlavní vstupní bod
- Inicializace
- Ukončení

## 3 Další aspekty

- Kódování ukazatelů
- Podpora pro Hot Patching
- Call na instrukci jmp

## 4 Import Address Table

- Importní adresář a Import Address Table
- Hackování Import Address Table

## Rozklad funkce do základních bloků

Nyní, když máme zanalyzovaný zásobník a identifikovány lokální proměnné, začneme s analýzou toku kódu. Nejprve vytvoříme graf toku kódu (Control Flow Graph, CFG), a ten použijeme pro analýzu jazykových konstruktů jako **if-then-else** nebo smyček **do-while/for/while**.

Vytváření CFG začíná rozložením kódu na základní bloky.

### Základní blok [NI-GEN]

Základní blok (Basic Block, BB) je maximální posloupnost po sobě jdoucích instrukcí, do které lze vstoupit jedinečně první instrukcí posloupnosti a opustit ji jedinečně poslední instrukcí.

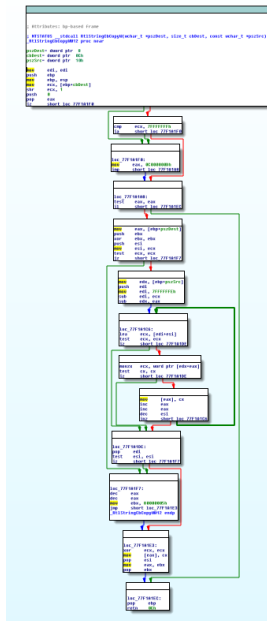
## Tvorba grafu toku kódu

Jakmile máme funkci rozdělenou na základní bloky, můžeme vytvořit graf toku kódu (CFG), kde:

- každý základní blok je vrcholem;
- tok kódu vytváří orientované hrany z jednoho BB do druhého.

CFG nám dává prvotní vysokoúrovňový  
náhled na funkci, kterou analyzujeme.

CFG slouží jako jeden ze vstupů do dekompilátoru, který se v něm snaží rozpoznat známé vzorce a rekonstruovat vysokoúrovňový kód. Vytvoření CFG nemusí být vždy zcela přímočaré, zejména pokud jsou použity obfuskační techniky (např. neprůhledné predikáty).



# Příkaz if-then I

## S instrukcí CMP/TEST

Toto je nejjednodušší varianta podmíněného skoku. Příkaz má tuto strukturu:

**C**

```
if( podmínka )  
    výraz;
```

**Pseudokód v assembleru**

```
porovnávací operace ověřující podmínku  
jxx přeskočit  
výraz  
přeskočit:
```

```
77e3c798 kernel32!StringCbPrintfW
```

```
...
```

```
77e3c7a0 shr ecx, 1
```

```
77e3c7a2 push 0
```

```
77e3c7a4 pop eax
```

```
77e3c7a5 jz short loc_77e3c7e5
```

```
// Instrukce cmp nastaví příznaky podle výsledku porovnání
```

```
77e3c7a7 cmp ecx, 7fffffffh
```

```
77e3c7ad ja short loc_77e3c7e5
```

```
...
```

```
77e3c7e5 mov eax, 80070057h // HRESULT_FROM_WIN32( ERROR_INVALID_PARAMETER )
```

```
77e3c7ea jmp short loc_77e3c7af
```

# Příkaz if-then II

## Bez instrukce CMP/TEST

Příkaz má stejnou strukturu jako minule, ale pro nastavení příznaků v registru EFL používá aritmetické nebo jiné instrukce..

### C

```
if( podmínka )
    výraz;
```

### Pseudokód v assembleru

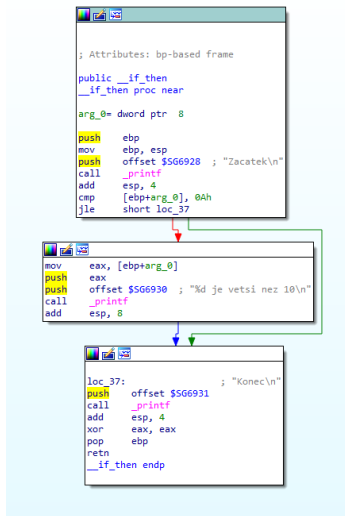
```
aritmetická operace počítající podmínku
jxx přeskočit
výraz
přeskočit:
```

```
77e3c798 kernel32!StringCbPrintfW
77e3c798 mov edi, edi          // 2-byte NOP pro hot patching
77e3c79a push ebp
77e3c79b mov ebp, esp
77e3c79d mov ecx, [ebp+c]      // Načti 2. parametr do ECX
77e3c7a0 shr ecx, 1          // Neznaménkové dělení 2, nastaví ZF a CF
77e3c7a2 push 0
77e3c7a4 pop eax

// Instrukce shr nastaví/smaže ZF, pokud výsledek je/není nula
// Push a pop nemění příznaky
77e3c7a5 jz short loc_77e3c7e5
...
77e3c7e5 mov eax, 80070057h    // HRESULT_FROM_WIN32( ERROR_INVALID_PARAMETER )
77e3c7ea jmp short loc_77e3c7af
```

# Příkaz if-then III

## Graf toku kódu



# Příkaz if-then-else I

Tento příkaz přidává k předchozímu alternativní větev.

C

```
if( podmínka )  
    výraz1;  
else  
    výraz2;
```

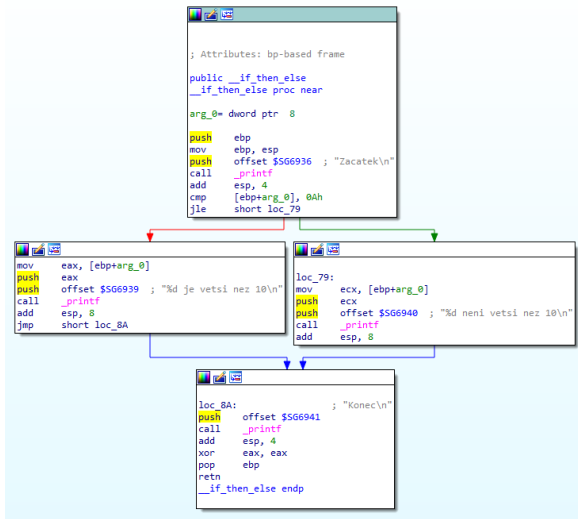
kód v assembleru

```
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 // Propadne do společné části  
  
77e2a8e7 loc_77e2a8e7:  
77e2a8e7 pop ebp  
77e2a8e8 ret 8  
  
77e5547c loc_77e5547c:  
77e5547c mov eax, [ebp+c]  
77e5547f or dword ptr [eax], 0fffffffh  
77e55482 mov eax, 80070216h // HRESULT_FROM_WIN32( ERROR_ARITHMETIC_OVERFLOW )  
77e55487 jmp loc_77e2a8e7 // Skok zpět
```



# Příkaz if-then-else II

## Graf toku kódu



# Smyčky I

## Smyčka while

**C**

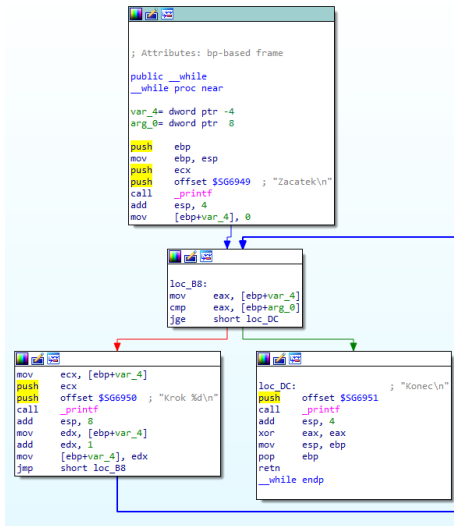
```
while( podmínka )  
    tělo;
```

**Kód v assembleru**

```
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 podmínky  
0040156c  je      loc_040158c // Nesplněno, skok za smyčku  
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 // Skok na další iteraci  
0040158c  loc_040158c:  
0040158c  pop     ebp  
0040158d  ret
```

# Smyčky II

## Graf toku kódu — smyčka while



# Smyčky III

## Smyčka do-while

**C**

```
do {  
    tělo;  
} while(podmínka);
```

**Kód v assembleru**

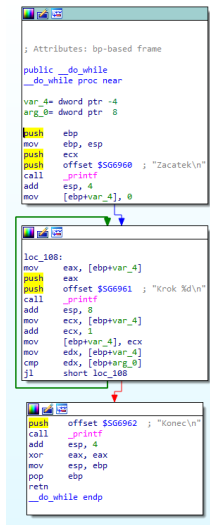
```
00401340  mov     edx,dword ptr [esp+4]  
00401344  mov     eax,dword ptr [esp+8]  
00401348  loc_0401348:  
00401348  movzx   ecx,byte ptr [eax]  
0040134b  lea     eax,[eax+1]  
0040134e  mov     byte ptr [edx],cl  
00401350  movzx   ecx,byte ptr [eax-1]  
00401354  test    cl,cl           // Test podmínky  
00401356  jne     loc_0401348h    // Skok na další iteraci  
00401358  ret
```

Všimněte si, že tato funkce nevytvořila rámec zásobníku. Na parametry je odkazováno přímo přes registr ESP. Tento přístup vede na kratší a rychlejší kód, ale každá instrukce `push` mění offset parametrů a lokálních proměnných na zásobníku.

Vynechání rámce zásobníku si lze vyžádat během kompilace pomocí `-fomit-frame-pointer` v GCC nebo `/Oy` v MSVC.

# Smyčky IV

## Graf toku kódu — smyčka do-while



# Smyčky V

## Smyčka for

**C**

```
for(  
    inicializace;  
    podmínka;  
    inkrement  
)  
{  
    tělo;  
}
```

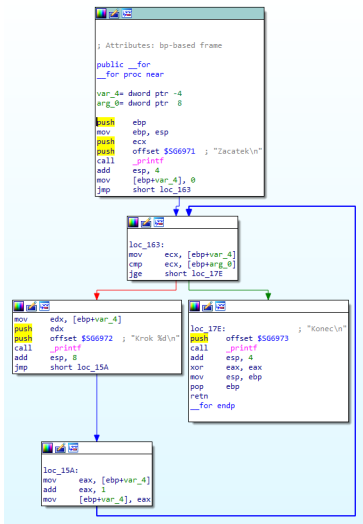
**Kód v assembleru**

```
00401280  mov     ecx,dword ptr [esp+4]  
00401284  xor     eax,eax  
00401286  test    ecx,ecx           // Vstoupit do smyčky?  
00401288  jle     loc_040129c       // Ne - skok za smyčku  
0040128a  lea     ebx,[ebx]  
00401290  loc_0401290:  
00401290  mov     dword ptr [eax*4+403020h],eax  
00401297  inc     eax               // Provést inkrementaci  
00401298  cmp     eax,ecx           // Test podmínky  
0040129a  jl      loc_0401290       // Skok na další iteraci  
0040129c  loc_040129c:  
c0040129c  ret
```

V příkazu **for** může být kterákoliv z částí prázdná. To nám dovolí zkonstruovat **while** smyčku vynecháním inicializace a inkrementu nebo jejich přesunutím. Generovaný kód by pak byl totožný jako u smyčky **while**. Z kódu nelze rozpoznat, jestli původní kód používal **for** nebo **while**. Dekompilátor obvykle preferuje jeden z nich a ten vždy použije.

# Smyčky VI

## Graf toku kódu — smyčka for



# Příkaz switch I

s použitím instrukce cmp

C

```
switch( arg ) {
    case 1:
        výraz_1;
        break;

    case 2:
        výraz_2;
        break;

    case 3:
        výraz_3;
        break;

    case 4:
        výraz_4;
        break;

    default:
        výraz_n;
        break;
}
```

## Kód v assembleru

```
8048389 mov    0x8(%ebp),%eax // Načíst arg. do 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 // Vrátit 0;
80483b0 leave
80483b1 ret
80483b2 movl   $0x8048589,(%esp)
80483b9 call   0x8048350 <puts@plt>
80483be jmp    0x80483ae
80483c0 dec    %eax // I zde se běžně používá instr. dec/sub!
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
```



# Příkaz switch II

s použitím sub/dec

C

```
switch( arg ) {
    case 1:
        výraz_1;
        break;

    case 2:
        výraz_2;
        break;

    case 4:
        výraz_4;
        break;

    default:
        výraz_n;
        break;
}
```

## Kód v assembleru

```
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           // vrátit 0, duplikováno
0040101F retn                    // konec funkce, duplikováno
00401020 push    offset třiParametry
00401025 call    ds:printf
0040102B add     esp, 4
0040102E xor     eax, eax           // vrátit 0, duplikováno
00401030 retn                    // konec funkce, duplikováno
00401031 push    offset jedenParametr
00401036 call    ds:printf
0040103C add     esp, 4
0040103F xor     eax, eax           // vrátit 0, duplikováno
00401041 retn                    // konec funkce, duplikováno
00401042 push    offset bezParametrů
00401047 call    ds:printf
0040104D add     esp, 4
00401050 xor     eax, eax           // vrátit 0, duplikováno
00401052 retn                    // konec funkce, duplikováno
```

# Příkaz switch III

se skokovou tabulkou

C

```
switch( arg ) {
    case 1:
        výraz_1;
        break;

    case 2:
        výraz_2;
        break;

    case 3:
        výraz_3;
        break;

    case 4:
        výraz_4;
        break;

    default:
        výraz_n;
        break;
}
```

## Skoková tabulka

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

## Kód v assembleru

```
00401000 mov     eax, [esp+4]
00401004 dec     eax
00401005 cmp     eax, 3
00401008 ja     short loc_401055
0040100A jmp     ds:off_401068[eax*4] // Skok na adresu dle tab.
00401011 push    offset bezParametrů
00401016 call    ds:printf
0040101C add     esp, 4
0040101F xor     eax, eax // vrátit 0, duplikováno
00401021 retn    // konec funkce, duplikováno
00401022 push    offset jedenParametr
00401027 call    ds:printf
0040102D add     esp, 4
00401030 xor     eax, eax // vrátit 0, duplikováno
00401032 retn    // konec funkce, duplikováno
00401033 push    offset dvaParametry
00401038 call    ds:printf
0040103E add     esp, 4
00401041 xor     eax, eax // vrátit 0, duplikováno
00401043 retn    // konec funkce, duplikováno
00401044 push    offset třiParametry
00401049 call    ds:printf
0040104F add     esp, 4
00401052 xor     eax, eax // vrátit 0, duplikováno
00401054 retn    // konec funkce, duplikováno
00401055 push    offset defaultCase
0040105A call    ds:printf
00401060 add     esp, 4
00401063 xor     eax, eax // vrátit 0, duplikováno
00401065 retn    // konec funkce, duplikováno
```

## Graf toku kódu — skoková tabulka



# Shrnutí

Nyní bychom měli:

- rozumět prologu a epilogu;
- rozumět rámci zásobníku a jeho struktuře;
- chápat, co je to základní blok;
- umět zkonstruovat graf toku kódu funkce, a
- rozumět tomu, jak jsou jazykové konstrukty C kompilovány do assembleru, a dokázat je přeložit zpět do čitelného kódu s vyšší úrovní abstrakce.

Nyní se detailněji podíváme na to, co dělá běhové prostředí při spuštění programu. To zahrnuje:

- co je to vstupní bod a k čemu slouží;
- volání inicializačních funkcí;
- volání funkce `main`;
- volání ukončovacích funkcí.

# Hlavní vstupní bod

Funkce `main/wmain/_tmain` **nejsou** vstupním bodem do programu. Skutečným vstupním bodem je funkce, jejíž relativní virtuální adresa (RVA) je uvedena v poli `AddressOfEntryPoint` v Optional Header PE souboru.

## Vstupní bode, kdepak jsi?

```
// Získat HMODULE aktuálního procesu
HMODULE hSelf = GetModuleHandle(NULL); // = LoadLibrary, ...

// Přetypovat HMODULE na ukazatel na začátek PE image v paměti. Ukazuje na MZ... - DOSová hlavička
PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)hSelf;

// Nová hlavička se nachází e_lfanew bajtů od začátku image
PIMAGE_NT_HEADERS32 pNtHeaders = (PIMAGE_NT_HEADERS)((BYTE*)pDosHeader) + pDosHeader->e_lfanew;

// Nakonec přečteme relativní virtuální adresu vstupního bodu a přičteme ji k začátku image
void* pfnEntryPoint = (void*)((BYTE*)pDosHeader
                               + pNtHeaders->OptionalHeader.AddressOfEntryPoint);

printf("Adresa vstupního bodu: %p\n", pfnEntryPoint);
```

Adresa vstupního bodu: 00402390

Měli bychom se ptát, **kdo** vytváří vstupní bod, **odkud** je volán, a **proč** to není funkce `main`?

# Kdo volá vstupní bod?

To už jsme viděli v přednášce 1.

Adresa vstupního bodu: 00402390

## Call stack pro vstupní bod

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

# Kdo vytváří vstupní bod programu?

Vstupní bod programu poskytuje runtime knihovna jazyka. Nalezneme ho v `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();
}
```

## Poznámky

Jméno této funkce je závislé na nastavení programu (zda používáme „Use Unicode Character Set“, tj. máme `#define UNICODE` 1) a na tom, zda se používá `main` nebo `WinMain`. Možná jména jsou:

- 1 `mainCRTStartup`
- 2 `wmainCRTStartup`
- 3 `WinMainCRTStartup`
- 4 `wWinMainCRTStartup`

## \_\_tmainCRTStartup()

Každý program v zkompileovaný MSVC začíná tímto kódem:

```
__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;
}
```



# Inicializační kód

Kompilátor za nás odvede množství práce a my se nemusíme starat o runtime detaily. Pokud chceme použít globální C++ třídu, stačí napsat:

## Příklad kódu s inicializovanou třídou

```
// Globální inicializovaná třída
Initializer g_InitializerClassInstance;

class Initializer {
public:
    Initializer() {
        printf("Pozdrav z kódu před main.\n");
    }

    ~Initializer() {
        printf("Pozdrav z kódu za main.\n");
    }
};
```

**Pozn.:** GCC používá speciální klíčová slova `__attribute__((constructor))` a `__attribute__((destructor))`; to dovoluje použít inicializátory i z C a nechat zavolat funkci během inicializační nebo ukončovací fáze běhu programu.

# Ruční inicializace

Pokud potřebujeme jemnější kontrolu, musíme použít konstrukty `#pragma`.

## Detailní řízení inicializace

```
#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;
```

Inicializátory v sekcích `.CRT$XIxxx` mohou vracet hodnoty. Nenulová návratová hodnota způsobí, že se inicializace programu přeruší a program skončí s chybou 255.

# Kde jsou inicializátory umístěny? I

Funkce `_initterm_e` a `_initterm` mají na vstupu ukazatele na začátek a konec části sekce `.rdata`, následující po IAT. Podívejme se na obsah této sekce:

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	.....I+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	A+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	.....3A."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	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Obrázek: `__xi_a` až `__xi_z` v debug verzi programu.

Každá položka v tomto seznamu reprezentuje ukazatel na inicializační funkci v jazyku C:

```

413940 pre_c_init
41108c jmp MyInit_PreC pro debug, respektive MyInit_PreC pro release
4110c3 jmp __atoneexitinit pro debug, resp. __atoneexitinit pro release
4111b3 jmp MyInit_PostC pro debug, resp. MyInit_PostC pro release
411122 jmp ___CxxSetUnhandledExceptionFilter pro debug, resp. přímý skok v release

```

## Kde jsou inicializátory umístěny? II

```
int _initterm_e(int (**pSegStart)(void), int (**pSegEnd)(void))
{
    int initResult = 0;
    while ( pSegStart < pSegEnd && !initResult )
    {
        if ( *pSegStart )
            initResult = (*pSegStart)();
        ++pSegStart;
    }
    return initResult;
}
```

Pole obsahuje ukazatele na inicializační funkce. Pokud ukazatel není null, bude zavolán. Všimněte si, že se volání C-inicializátorů ukončí v okamžiku, kdy první inicializátor vrátí nenulovou hodnotu (na rozdíl od C++ inicializátorů).

# Kde jsou inicializátory umístěny? 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	FD	23	00	00	!%..!%..b#...š#...
00000090	E8	23	00	00	86	24	00	00	00	00	00	00	00	00	00	00	e#...!\$.....
000000A0	DD	12	40	00	30	10	40	00	00	10	40	00	30	10	40	00	Y:@.Q+@.  +@. Q+@.
000000B0	00	00	00	00	00	00	00	00	24	12	40	00	20	10	40	00	.....\$!@..+@.

Obrázek: \_\_xc\_a až \_\_xc\_z v release verzi.

Každá položka v tomto seznamu reprezentuje ukazatel na inicializační funkci v jazyku C++:

```

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

00401000    mov     ecx,4031B8h // Ukazatel na this
00401005    call    Initializer::Initializer
0040100A    push    401930h      // adresa volání destruktoru
0040100F    call    atexit        // zaregistrovat k volání před ukončením programu
00401014    pop     ecx          // vyčištění zásobníku
00401015    ret

...

00401930    mov     ecx,4031B8h // Ukazatel na this
00401935    jmp     Initializer::~Initializer

```

# Terminátory I

Viděli jsme, že byla používána funkce `atexit` k zaregistrování funkcí, které při ukončení programu zavolají destruktory. `Atexit` využívá dynamicky alokované pole, na které ukazují dva zakódované globální ukazatele `__onexitbegin` a `__onexitend`. Do tohoto pole jsou umístěny zakódované ukazatele na funkce. Když je zavolána C funkce `exit` (ale ne `TerminateProcess`), jsou postupně volány funkce zaregistrované pomocí `atexit`. Pokud navíc používáme staticky přilinkovanou runtime knihovnu, zavolají se zde i před-ukončovací a ukončovací funkce zaregistrované v segmentech `.crt$xpXXX` a `.crt$xtXXX`, obdobně jako u inicializátorů.

# Terminátory II

## Funkce atexit (atexit.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;  
}
```

## Funkce exit (crt0dat.c)

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

# Terminátory III

## Pseudokód funkce doexit (crt0dat.c)

```
void __cdecl doexit(int code, int quick, int retcaller) {
    if ( !retcaller && check_managed_app() ) // V managovaném procesu zavolej CorExitProcess
        __crtCorExitProcess(uExitCode);
    ...
    if ( !quick ) {
        onexitbegin = DecodePointer(__onexitbegin); onexitend = DecodePointer(__onexitend);
        while ( 1 ) { // Iteruj přes všechny ukončovací funkce
            // Najdi první "nenulovou" funkci
            while ( --onexitend >= onexitbegin && *onexitend == EncodePointer(NULL) );
            // Ukončovací podmínka
            if ( onexitend < pfn__onexitbegin ) break;
            // Dekóduj, zavolej, a odstraň ze seznamu
            pfnExitProc = (void (*)(void))DecodePointer(*onexitend);
            *onexitend = EncodePointer(0);
            pfnExitProc();
            ...
        }
        #ifndef CRTDLL
            _initterm(__xp_a, __xp_z); // Zavolej před-ukončovací funkce
        #endif
    }
    #ifndef CRTDLL
        _initterm(__xt_a, __xt_z); // Zavolej ukončovací funkce
    #endif
    ...
    if ( ret ) return;
    if ( !ret ) __crtExitProcess(code);
}
```



## Kódování ukazatelů

Ukazatele na zásobníku nebo na haldě by mohly být přepsány a zneužity ke spuštění útočnickova kódu. API EncodePointer a DecodePointer tomu brání. Jejich volání jsou interně mapována na RtlEncodePointer resp. RtlDecodePointer API z 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 0FFFFFFFh        ; 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
```

# Podpora pro Hot Patching

Na začátku poslední funkce jste si mohli všimnout instrukce `mov edi,edi`. Před začátkem funkce je navíc 5 instrukcí `nop`. Tyto instrukce zajišťují podporu pro hot patching, mechanismus, který nám dovoluje snadno upravit funkci za běhu bez nutnosti restartovat aplikaci. Poskytují 7 bajtů volného místa, které můžeme použít pro upravenou funkci. Instrukce `mov edi,edi` bude nahrazena instrukcí `jmp short`, která skočí na začátek `nopů`, kam bude vložena instrukce `jmp [addr]`. Kód pak vypadá takto:

## Neupravený kód

```

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

```

## Upravený kód

```

skok_na_upravenou_funkci:
E9 xx xx xx xx  jmp dword ptr [&upravená_funkce]
začátek_funkce:
EB F9  jmp short skok_na_upravenou_funkci
55  push ebp  // Nedosažitelné
8B EC  mov ebp,esp // Nedosažitelné

```

# Call na instrukci jmp? I

V debugovací verzi byly všechny funkce volány přes další úroveň nepřímého skoku. Co to mělo za smysl?

```
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)

__atosexitinit:
004110C3 jmp  __atosexitinit (413220h)

__report_securityfailureEx:
004110C8 jmp  __report_securityfailureEx (413750h)

_FindPESection:
004110CD jmp  _FindPESection (413FC0h)

MyInit_PreCPP:
004110D2 jmp  MyInit_PreCPP (412340h)

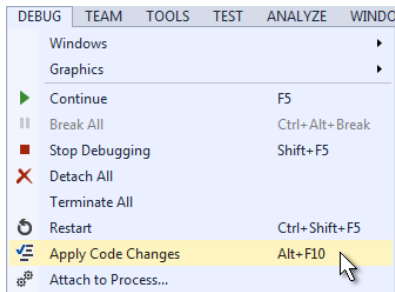
_LoadLibraryW@4:
004110D7 jmp  LoadLibraryW (4131EEh)

__configthreadlocale:
004110DC jmp  _configthreadlocale (4143EEh)

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

## Call na instrukci jmp? II

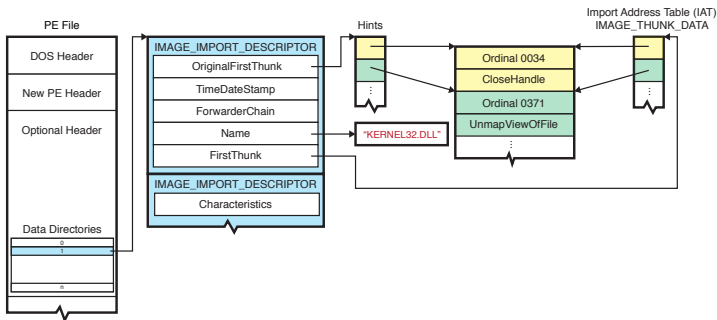
Pokud je povoleno inkrementální linkování, je do všech volání funkcí zabudován další nepřímý skok. To nám umožňuje **za běhu nahradit libovolnou funkci**. Jednoduše zkompilujeme nový kód, vložíme ho do paměti procesu a upravíme nepřímý skok tak, aby ukazoval na novou implementaci. MSVC k tomu používá „Apply Code Changes“, Apple používá „Fix and Continue“, ale princip je v obou případech stejný.



Obrázek: MSVC Apply Code Changes.

# Import Address Table I

Modul může záviset na jiných modulech. Tyto moduly a jména symbolů nebo jejich pořadová čísla nalezneme v importním adresáři (Import Directory, ID) v PE souboru.



Obrázek: Importní adresář PE a Import Address Table.

# Import Address Table II

## Procházení importního adresáře

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

PIMAGE_IMPORT_DESCRIPTOR pImportsEnd = (PIMAGE_IMPORT_DESCRIPTOR)((BYTE*)pImports
+ pNTHHeaders->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 )
        {
            // pozor, napřed bychom měli otestovat typ importu...
            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

Poté, co byl image načten do paměti, nahradí loader všechny odkazy do IAT (RVA struktur `IMAGE_THUNK_DATA32` v 32-bitovém PE souboru) (vlevo) ukazující na `IMAGE_IMPORT_BY_NAME` (vpravo dole) skutečnými ukazateli na funkce (vpravo).

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00001200 8A 24 08 00 78 24 00 00 68 24 00 00 58 24 00 00 58.x6.h6.X6.
00001210 AC 27 00 00 06 27 00 00 EC 26 00 00 0D 26 00 00 .'....la.Ba.
00001220 BC 26 00 00 AC 26 00 00 9C 26 00 00 32 27 00 00 46...m..m..2'..
00001230 00 00 00 00 DC 24 00 00 E4 24 00 00 EE 24 00 00 ....US..AS..IS.
00001240 FC 24 00 00 0A 25 00 00 22 25 00 00 3C 25 00 00 45...%...%<x.
00001250 D2 24 00 00 58 25 00 00 68 25 00 00 7A 25 00 00 05.X6.h6.x6.
00001260 D2 25 00 00 8A 25 00 00 9A 25 00 00 AA 25 00 00 %..Š...%..*%.
00001270 BE 25 00 00 CC 25 00 00 D8 25 00 00 EA 25 00 00 06..I...0%..Š%.
00001280 EE 25 00 00 FA 25 00 00 10 26 00 00 2A 26 00 00 44..ú...%..*%.
00001290 A2 26 00 00 56 26 00 00 7A 26 00 00 8C 26 00 00 86..V...z...%..Š%.
000012A0 AC 24 00 00 C8 24 00 00 C0 24 00 00 B6 24 00 00 4F..Š..Š..Š..Š..Š..Š..
000012B0 4A 25 00 00 00 00 00 00 00 00 00 00 76 16 40 00 4A.....v.Š.
000012C0 30 10 40 00 00 10 40 00 00 30 10 40 00 00 00 00 0.Š...Š.0.Š.....
000012D0 00 00 00 00 BD 15 40 00 00 20 10 40 00 91 14 40 00 ....%..Š..Š..Š..
000012E0 20 10 40 00 29 1A 40 00 00 00 00 00 00 00 00 00 .Š...Š.....
000012F0 00 00 00 00 10 A2 FC 54 00 00 00 02 00 00 00 00 .Š...eU.....
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 00 00 00 14 00 00 00 00 .eU.....
00001320 DC 22 00 00 DC 14 00 00 48 65 6C 6F 20 70 72 U"...Ü...Hello pr
00001330 65 2D 6D 61 69 6E 20 63 6F 64 65 2E 0A 00 00 00 e-main code.....
```

Obrázek: IAT na disku.

První `IMAGE_THUNK_DATA` (4 bajty) obsahuje RVA 0000248A (vlevo). Tato adresa, když ji namapujeme do paměti procesu, ukazuje na strukturu `IMAGE_IMPORT_BY_NAME` obsahující pořadové číslo (0267) následované nulou ukončeným jménem funkce (`GetModuleHandleW`) (vpravo dole). Funkce pochází z `kernel32.dll` a je umístěna na adrese 75C4CD5C, která je zapsána zpět do IAT (vpravo). IAT je umístěna v sekci `.rdata`, tudíž je **pouze pro čtení**.

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00402000 3C DD C4 75 DD 2C C4 75 62 DD C4 75 90 CE C4 75 \IauY,AuBYAu.IAu
00402010 C0 C4 C4 75 65 D8 C4 75 D2 C4 C4 75 D6 C6 C5 75 AAuEsAu0A0Au0A0
00402020 E2 7E C4 75 10 CD 59 77 95 A2 59 77 C6 D8 C4 75 á-Au.Iy..yYwE0Au
00402030 00 00 00 00 D7 ED A5 6B FC ID A5 6B 08 12 A6 6B ....iVkuIYk...jK
00402040 46 CA A6 6B 6B BE A7 6B AA 2A A6 6B ED 4C AF 6B FÉ:kk&ks**;kil"K
00402050 73 22 A6 6B 5F E2 A7 6B CE C7 A7 6B 93 42 A8 6B s"iK_&skfcs"b"K
00402060 B8 BB AC 6B 04 A1 A8 6B EB 35 AF 6B E9 B9 AC 6B .w-k.A"keU"ke-k
00402070 86 CC A6 6B 50 CC A6 6B 2C F6 B2 6B 40 F7 B2 6B +iKPI"K,k"6"6s-k
00402080 38 F6 B2 6B B5 6B AF 6B 0C 48 AF 6B F7 47 AF 6B 80-k"K"K"K"K"K"K"K
00402090 2C DC AE 6B DB C7 A7 6B 9B 46 AF 6B B5 C9 A7 6B .0&k0C&k"K"K"K"K"K
004020A0 D9 2F AD 6B 30 ED A5 6B E0 IC A5 6B FF CB A6 6B U/.k0iVkuIYkYÉjK
004020B0 8D BB AC 6B 00 00 00 00 00 00 00 76 16 40 00 .w-k.....v.Š.
004020C0 30 10 40 00 00 10 40 00 00 30 10 40 00 00 00 00 0.Š...Š.0.Š.....
004020D0 00 00 00 00 BD 15 40 00 00 20 10 40 00 91 14 40 00 ....%..Š..Š..Š..
004020E0 20 10 40 00 29 1A 40 00 00 00 00 00 00 00 00 00 .Š...Š.....
004020F0 00 00 00 00 10 A2 FC 54 00 00 00 02 00 00 00 00 .Š...eU.....
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 00 00 00 14 00 00 00 00 .eU.....
00402120 DC 22 00 00 DC 14 00 00 48 65 6C 6F 20 70 72 U"...Ü...Hello pr
00402130 65 2D 6D 61 69 6E 20 63 6F 64 65 2E 0A 00 00 00 e-main code.....
```

Obrázek: IAT v paměti.

```
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 4E iProtect...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 RHEL32.dll..y.pr
```

# Import Address Table IV

Při „svázání“ (bind) externího symbolu je zjištěna jeho adresa a všechny odkazy na tento symbol jsou zaktualizovány prostřednictvím relokační tabulky (BI-BEK, přednáška 2). Tak to platí pro objektové soubory; u spustitelných souborů by to při každém startu vynucovalo relokační volání externích symbolů. Proto se používají nepřímé skoky a stačí změnit **jeden** údaj pro každý symbol – ten v IAT!

## Volání funkce skrze IAT

```
// Voláme funkci
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      // MSVCRT120.DLL!__amsg_exit
004020B4  .dd  0                // - konec -

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



# Hackování IAT I

Jak jsme viděli, všechna volání daného externího API jsou provedena skrz jediný bod v programu — záznam symbolu v IAT. Co by se stalo, kdybychom obsah tohoto údaje změnili?

## Hackování IAT II

Jak jsme viděli, všechna volání daného externího API jsou provedena skrz jediný bod v programu — záznam symbolu v IAT. Co by se stalo, kdybychom obsah tohoto údaje změnili?

Všetchna volání tohoto API napříč celým programem by byla přesměrována!

## Hackování IAT III

Jak jsme viděli, všechna volání daného externího API jsou provedena skrz jediný bod v programu — záznam symbolu v IAT. Co by se stalo, kdybychom obsah tohoto údaje změnili?

Všechna volání tohoto API napříč celým programem by byla přesměrována!

IAT je standardně jen pro čtení, ale můžeme použít API jako `VirtualProtect` k povolení zápisu a přesměrovat všechna volání napříč celým programem na naši funkci.

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

# Hackování IAT IV

Dokážeme se dostat za hranice našeho procesu? Odpověď je jednoznačně ANO. Můžeme použít buď funkce jako VirtualAllocEx, ReadProcessMemory, WriteProcessMemory, VirtualProtectEx, CreateRemoteThread pro hacknutí jiného procesu a injektování svého kódu nebo DLL, nebo využít tzv. 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

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

# Literatura



Russinovich M., Solomon D. A., Ionescu A.: *Windows Internals Part 1*, 6<sup>th</sup> ed., 2012.



Russinovich M., Solomon D. A., Ionescu A.: *Windows Internals Part 2*, 6<sup>th</sup> ed., 2012.