

# Reverzní inženýrství

## 1. Úvod do reverzního inženýrství, analýza zásobníku

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

Ing. Josef Kokeš



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

České vysoké učení technické v Praze  
Fakulta informačních technologií  
Katedra informační bezpečnosti

Verze 2020-09-04

# Obsah I

## 1 Úvod

- Reverzní inženýrství a jeho použití
- Etické a právní aspekty

## 2 Klíčové vlastnosti reverzního inženýrství

- Disassemblování a dekompilace
- Analýza mrtvého kódu
- Analýza živého kódu

## 3 Aplikační binární rozhraní

- Zarovnání zásobníku a dat
- Name mangling/Dekorace jmen
- Volací konvence

## 4 Reverzování funkcí

- Prolog
- Strukturovaná obsluha výjimek (32-bit)
- Epilog

# Obsah II

- Tělo

## 5 Analýza rámce zásobníku

- Analýza rámce zásobníku

## 6 Další analýza

- Analýza znamének typů
- Analýza API volání

# Reverzní inženýrství

## Softwarové inženýrství [1]

Softwarovým inženýrstvím rozumíme studium a použití inženýrských přístupů při návrhu, vývoji a údržbě software.

- V softwarovém inženýrství začínáme se zdrojovým kódem. Ten je kompilován na objektový kód a následně linkován na kód spustitelný.
- Reverzní softwarové inženýrství provádí opačný proces.

## Reverzní (softwarové) inženýrství (RE) [2]

Reverzním inženýrstvím rozumíme proces analýzy předmětného systému, abychom mohli vytvořit reprezentaci tohoto systému na vyšší úrovni abstrakce.

- V průběhu tohoto procesu dokumentujeme a snažíme se pochopit, jak studovaný systém funguje.

# Využití reverzního inženýrství

RE zkoumá, jak softwarový produkt funguje:

- za účelem zjištění, jaké algoritmy, metody, souborové formáty a protokoly produkt používá;
- za účelem nalezení zranitelností v produktu;
- jako první krok v návrhu konkurenčního produktu;
- za účelem návrhu produktu, který se zkoumaným bude spolupracovat;
- aby bylo možné rozhodnout, zda je o produkt škodlivý nebo neškodný.

Všechny tyto kroky mají dva aspekty: **etický** a **právní**.

# Etické aspekty reverzního inženýrství

- Etická využití RE, máme-li platnou licenci v souladu s českým právem:
  - propojení s proprietárním produktem;
  - zjištění (neveřejných) rozhraní, aby mohla být funkcionality proprietárního produktu rozšířena;
  - oprava zranitelností v proprietárním produktu.
- Neetická použití RE zahrnují:
  - odstranění nebo obejití ochrany proti kopírování;
  - zjištění způsobu vytváření sériového čísla produktu a vytvoření keygenů;
  - tvorba konkurenčního produktu.

# Legální aspekty reverzního inženýrství

Licenční smlouva s koncovým uživatelem (EULA) často zakazuje provádění RE:

*You may not reverse engineer, decompile, or disassemble the Software, except and only to the extent that such activity is expressly permitted by this EULA or applicable law notwithstanding this limitation. [Microsoft EULA]*

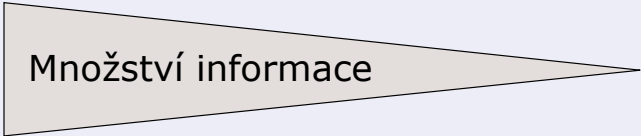
Český autorský zákon č. 121/2000 Sb. v platném znění v § 66 říká:

- (1) Do práva autorského **nezasahuje** oprávněný uživatel rozmnoženiny počítačového programu, jestliže:
  - a) **rozmnožuje, překládá, zpracovává, upravuje či jinak mění počítačový program, je-li to nezbytné k využití oprávněně nabytého rozmnoženiny počítačového programu, činí-li tak při zavedení a provozu počítačového programu nebo opravuje-li chyby počítačového programu,**
  - b) **jinak rozmnožuje, překládá, zpracovává, upravuje či jinak mění počítačový program, je-li to nezbytné k využití oprávněně nabytého rozmnoženiny počítačového programu v souladu s jeho určením, není-li dohodnuto jinak,**
  - d) **zkoumá, studuje nebo zkouší sám nebo jím pověřená osoba funkčnost počítačového programu za účelem zjištění myšlenek a principů, na nichž je založen kterýkoli prvek počítačového programu, činí-li tak při takovém zavedení, uložení počítačového programu do paměti počítače nebo při jeho zobrazení, provozu či přenosu, k němuž je oprávněn,**
  - e) **rozmnožuje kód nebo překládá jeho formu při rozmnožování počítačového programu nebo při jeho překladu či jiném zpracování, úpravě či jiné změně, je-li k ní oprávněn, a to samostatně nebo prostřednictvím jím pověřené osoby, jsou-li takové rozmnožování nebo překlad nezbytné k získání informací potřebných k dosažení vzájemného funkčního propojení** nezávisle vytvořeného počítačového programu s jinými počítačovými programy, **jestliže informace potřebné k dosažení vzájemného funkčního propojení nejsou pro takové osoby dříve jinak snadno a rychle dostupné a tato činnost se omezuje na ty části počítačového programu, které jsou potřebné k dosažení vzájemného funkčního propojení.**
- (4) **Informace získané při činnosti podle odstavce 1 písm. e) nesmějí být poskytnuty jiným osobám, ledaže je to nezbytné k dosažení vzájemného funkčního propojení nezávisle vytvořeného počítačového programu, ani využity k jiným účelům než k dosažení vzájemného funkčního propojení nezávisle vytvořeného počítačového programu.** Dále nesmějí být tyto informace využity ani k vývoji, zhotovení nebo k obchodnímu využití počítačového programu podobného tomuto počítačovému programu v jeho vyjádření nebo k jinému jednání ohrožujícímu nebo porušujícímu právo autorské.

→ Můžeme legálně RE software, abychom porozuměli, jak pracuje, a pro vytvoření funkčního propojení. Co o RE říká EULA **nemusí být vždy relevantní!** (zákon má přednost před smlouvou). Ale pozor na odst. 1 písmeno b!

# Množství informace

Zdrojový kód	Objektový kód	Spustitelný kód
zdrojový kód	debug informace	—
komentáře	(komentáře)	—
samostatné soubory	samostatné soubory	sloučeno
knihovny zvlášť	knihovny zvlášť	sloučeno



Množství informace

**Tabulka:** Množství informace ve zdrojovém, objektovém a spustitelném kódu.

- Objektový kód obsahuje méně informace než zdrojový kód.
- Spustitelný kód obsahuje ještě méně informace než objektový.

→ Vracet se zpět je obtížná úloha, protože informace chybí!



# Získání zdrojového kódu ze spustitelného programu

- Kvůli nedostatku informace ve spustitelných souborech není možné plně rekonstruovat zdrojový kód. Rekonstruovaný kód může být blízký původnímu, ale typicky není kompilovatelný. Plný převod zdroják-binárka-zdroják není realistický.
- Můžeme použít techniku:
  - **disassemblování** — transformace spustitelného kódu ze strojového jazyka do assembleru cílového procesoru. Výsledkem je obrovské množství kódu, lidské bytosti potřebují jazyk vyšší úrovně. Disassemblování může být prováděno na **mrtvém kódu** nebo na **živém kódu** v debuggeru. Disassemblování může být komplikováno pomocí obfuskací (e.g. neprůhledné predikáty, maskování API volání, ...), kódováním, kompresí, atd.
  - **dekompilece** — transformace spustitelného kódu do vyššího programovacího jazyka (např. C, Python, nebo Perl). Není stoprocentní, není možná rekompilece. Obfuskace, packery [Přednáška 4] a polymorfní kód mohou celý proces velmi ztížit.

# Analýza mrtvého kódu

## Analýza mrtvého kódu (též statická analýza)

Analýzu mrtvého kódu provádíme na neběžícím spustitelném kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci disassembleru.

- 1 Mějme disassemblovaný kód [Přednáška 4].
- 2 Měli bychom ho rovnou začít zkoumat?

# Analýza mrtvého kódu

## Analýza mrtvého kódu (též statická analýza)

Analýzu mrtvého kódu provádíme na neběžícím spustitelném kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci disassembleru.

- ➊ Mějme disassemblovaný kód [Přednáška 4].
- ➋ Měli bychom ho rovnou začít zkoumat?
- ➌ Když obsahuje tolik řádků kódu (LOC)? **Redukujme ho!**

# Analýza mrtvého kódu

## Analýza mrtvého kódu (též statická analýza)

Analýzu mrtvého kódu provádíme na neběžícím spustitelném kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci disassembleru.

- 1 Mějme disassemblovaný kód [Přednáška 4].
- 2 Měli bychom ho rovnou začít zkoumat?
- 3 Když obsahuje tolik řádků kódu (LOC)? **Redukujme ho!**
- 4 Redukujme LOC identifikováním kompilátoru [Přednáška 5] a staticky linkovaného knihovního kódu [Přednáška 5] a dat → zbavme se jich!

# Analýza mrtvého kódu

## Analýza mrtvého kódu (též statická analýza)

Analýzu mrtvého kódu provádíme na neběžícím spustitelném kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci disassembleru.

- 1 Mějme disassemblovaný kód [Přednáška 4].
- 2 Měli bychom ho rovnou začít zkoumat?
- 3 Když obsahuje tolik řádků kódu (LOC)? **Redukujme ho!**
- 4 Redukujme LOC identifikováním kompilátoru [Přednáška 5] a staticky linkovaného knihovního kódu [Přednáška 5] a dat → zbavme se jich!
- 5 Vyextrahujme veškerou informaci, která ve spustitelném kódu zůstává, a využijme ji k okomentování assembleru (kódové/datové/bss segmenty, jména tříd, řetězce, RTTI informace, informace o výjimkách, rámce zásobníku, informace o typech parametrů ze známých API volání, atd.)
- 6 Teprve na to, co zbyde, nastoupí člověk!

# Analýza mrtvého kódu

## Analýza mrtvého kódu (též statická analýza)

Analýzu mrtvého kódu provádíme na neběžícím spustitelném kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci disassembleru.

- 1 Mějme disassemblovaný kód [Přednáška 4].
- 2 Měli bychom ho rovnou začít zkoumat?
- 3 Když obsahuje tolik řádků kódu (LOC)? **Redukujme ho!**
- 4 Redukujme LOC identifikováním kompilátoru [Přednáška 5] a staticky linkovaného knihovního kódu [Přednáška 5] a dat → zbavme se jich!
- 5 Vyextrahujme veškerou informaci, která ve spustitelném kódu zůstává, a využijme ji k okomentování assembleru (kódové/datové/bss segmenty, jména tříd, řetězce, RTTI informace, informace o výjimkách, rámce zásobníku, informace o typech parametrů ze známých API volání, atd.)
- 6 Teprve na to, co zbyde, nastoupí člověk!
- 7 Pozor, i disassembler lze zmást [Přednáška 4]!

# Analýza živého kódu

## Analýza živého kódu (též dynamická analýza)

Analýzu živého kódu provádíme na běžícím kódu s cílem prostudovat a zdokumentovat chování programu. Obvykle ji provádíme za pomoci debuggeru.

- 1 Spustitelný program načteme do debuggeru a studujeme ho v něm.
- 2 Můžeme nastavit breakpointy a watchpointy a sledovat skutečné hodnoty registrů a paměti v libovolném okamžiku života programu.
- 3 Program se může debugování bránit (pomocí zabíjáčkových vláken, detekováním debuggeru, odmítnutím jeho připojení, atd.), používat CRC pro zjištění softwarových breakpointů, ...

# Aplikační binární rozhraní (ABI)/Volací konvence

Aplikační binární rozhraní (též volací konvence) [3] je dokument popisující, jak by se měl binární kód na cílové platformě chovat, aby byl kompatibilní s binárním kódem jiných tvůrců a s operačním systémem. To zahrnuje:

- jak jsou předávány parametry funkcím;
- jak je zarovnán zásobník a data;
- použití registrů CPU (které lze měnit, které musí být zachovány);
- jak jsou měněna jména symbolů;
- jak jsou v paměti vytvářeny struktury/třídy;
- jak jsou volány virtuální funkce v C++;
- jak se identifikuje typová informace za běhu (RTTI);
- jak se zpracovávají výjimky;
- ...



## Zarovnání dat I

Přístup k nezarovnaným datům je pomalejší (BI-APS), na některých CPU dokonce nemožný (např. MC68000 nemůže číst long word [4B] z liché adresy) → data musí být zarovnána. Dnešní procesory nezarovnaná data zpracují, ale přístup je pomalý → data zarovnáváme vždy, když to jde.

datový typ	i386		x86_64	
	MSVC	GCC	MSVC	GCC
1 byte char	1	1	1	1
2 byte int	4	2	4	2
4 byte int	4	4	4	4
8 byte int	8	8	8	8
float	4	4	4	4
double	8	8	8	8
pointer	4	4	8	8

**Tabulka:** Zarovnání statických dat v kompilátorech MSVC a GCC 3.x [3].

# Zarovnání dat II

datový typ	i386		x86_64	
	MSVC	GCC	MSVC	GCC
1 byte char	1	1	1	1
2 byte int	2	2	2	2
4 byte int	4	4	4	4
8 byte int	8	4/8	8	8
float	4	4	4	4
double	8	8	8	8
pointer	4	4	8	8

**Tabulka:** Zarovnání elementů struktur/tříd v kompilátorech MSVC a GCC 3.x [3].

- Zarovnání může působit problémy, pokud používáme několik různých kompilátorů.
- Z tohoto důvodu můžeme potřebovat zarovnání řídit.
- V assembleru používáme pseudoinstrukci `.align num_bytes`.
- C/C++ nabízí pro řízení zarovnání direktivu `#pragma`:

# Zarovnání dat III

```
#pragma pack(push)
// Uložit aktuální stav zarovnávání
#pragma pack(1)
// Od teď bude vše zarovnáno na 1 bajt
typedef struct _MYSTRUCT {
    char f_Field1;      // Pole f_Filler1 používáme k zarovnání
    char f_Filler1[1]; // f_Int na hranici 2 bajtů
    int  f_Int;         // Zarovnáno na hranici 2 bajtů!
} MYSTRUCT;
#pragma pack(pop)
// Obnovit původní stav zarovnávání
```

# Zarovnání zásobníku I

Také zásobník může vyžadovat zarovnání. Historicky, 32bitové systémy zarovnávají zásobník na 4bajtovou hranici, zatímco 64bitové systémy zarovnávají na 16bajtovou hranici. Zarovnání se typicky poruší použitím instrukce `call`, což způsobí, že vrchol zásobníku leží na adrese  $A \equiv 12 \pmod{16}$  (při 16bajtovém zarovnání v 32bitovém režimu) nebo  $A \equiv 8 \pmod{16}$  v 64bitovém režimu.

```
08048528 <main>:
8048528: 55                push    %ebp
8048529: 89 e5             mov     %esp,%ebp
804852b: 83 e4 f0          and     $0xffffffff0,%esp    // Obnova zarovnání zásobníku
804852e: 83 ec 20          sub     $0x20,%esp
8048531: c7 44 24 1c 00 00 00 00 movl    $0x0,0x1c(%esp)
8048539: c7 44 24 08 03 00 00 00 movl    $0x3,0x8(%esp)
8048541: c7 44 24 04 02 00 00 00 movl    $0x2,0x4(%esp)
8048549: c7 04 24 01 00 00 00 00 movl    $0x1,(%esp)
8048550: e8 76 ff ff ff    call    80484cb <f>
8048555: 89 44 24 1c       mov     %eax,0x1c(%esp)
8048559: 8b 44 24 1c       mov     0x1c(%esp),%eax
804855d: 89 44 24 04       mov     %eax,0x4(%esp)
8048561: c7 04 24 10 86 04 08 movl    $0x8048610,(%esp)
8048568: e8 23 fe ff ff    call    8048390 <printf@plt>
804856d: b8 00 00 00 00    mov     $0x0,%eax
```

# Zarovnání zásobníku II

```
8048572: c9          leave
8048573: c3          ret
8048574: 66 90      xchg    %ax,%ax
8048576: 66 90      xchg    %ax,%ax
8048578: 66 90      xchg    %ax,%ax
804857a: 66 90      xchg    %ax,%ax
804857c: 66 90      xchg    %ax,%ax
804857e: 66 90      xchg    %ax,%ax

// Funkce končí na adrese 8048573
// Tyto ne-operace jsou použity
// jako výplň, aby další funkce
// začala opět na 16-bajtové
// hranici 8048580.
```

# Name Mangling I

Některé jazyky podporují přetěžování funkcí/operátorů/metod. Tím pádem není prostý název symbolu jedinečný a nestačí k rozlišení dvou symbolů. Z tohoto důvodu se do názvu symbolu přidává další informace. Tento proces se nazývá **name mangling** (Microsoft mu říká **dekorace jmen**). V C++ se obvykle kódují následující informace:

- jméno symbolu;
- všechny jmenné prostory symbolu;
- jestli je symbol `const` a/nebo `volatile`;
- jestli je symbol `public`, `protected`, nebo `private`;
- pokud je symbol funkcí nebo metodou, tak jeho argumenty (některé kompilátory kódují i návratový typ);
- jestli je symbol skalárním nebo vektorovým datovým typem.

Ozdobená jména poskytují velké množství informace.

# Name Mangling II

Podívejme se na toto ozdobené jméno:

```
?_GetOutputTechnologyString@CDisplayData@@AAEXPAGHH@Z
```

Jde o jméno ozdobené dle Microsoftí konvence, která začíná jména znakem ? následovaným jménem metody `_GetOutputTechnologyString`.

Vidíme, že metoda je součástí třídy (nebo struktury) `CDisplayData`.

Všimněme si také, že @ slouží jako oddělovač. Následuje část `AAEXPAGHH`, která podle [3] znamená:

A	A	E	X	P
private	ani <code>const</code> ani <code>volatile</code>	<code>__thiscall</code>	vrací <code>void</code>	ukazatel na
A	G	H	H	@Z
ani <code>const</code> ani <code>volatile</code>	<code>unsigned short</code>	<code>int</code>	<code>int</code>	konec

Tabulka: Demangling jména symbolu.

# Name Mangling III

Získali jsme tedy tento prototyp funkce:

```
private __thiscall void CDisplayData::_GetOutputTechnologyString(  
    CDisplayData* this, unsigned short*, int, int  
);
```

Metody instance automaticky dostávají ukazatel `this` jako svůj první (implicitní) parametr. Pod Windows je v 32bitovém režimu použita volací konvence `__thiscall`, podle níž se ukazatel `this` předává v registru ECX. V 64bitovém režimu se ukazatel `this` předává jako první (implicitní) parametr.

Skutečný prototyp funkce by byl:

```
private void CDisplayData::_GetOutputTechnologyString( LPWSTR, int, int );
```

## Poznámka

Metody deklarované s klíčovým slovem `static` (třídní metody) **nezískávají** ukazatel `this` vůbec!



## Použití registrů

ABI také určuje způsob, jak se používají registry CPU: které se mohou měnit (volající může jejich hodnotu zničit), které slouží jako ukazatel na zásobník, které jako ukazatel na rámec zásobníku, kterými se předávají parametry při volání funkcí a které nesou návratovou hodnotu funkce.

Použití	Win32	32b Linux/BSD/OS X	Win64	64b Linux/BSD/OS X
<b>lze měnit</b>	EAX, ECX, EDX, ST(0)–ST(7)	EAX, ECX, EDX, ST(0)–ST(7)	RAX, RCX, RDX, R8–R11, ST(0)–ST(7)	RAX, RCX, RDX, RSI, RDI, R8–R11, ST(0)–ST(7)
<b>nutno zachovat</b>	EBX, ESI, EDI, EBP	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12–R15	RBX, RBP, R12–R15
<b>předávání parametrů</b>	speciální	speciální	RCX, RDX, R8, R9	RDI, RSI, RDX, RCX, R8, R9
<b>návratové hodnoty</b>	EAX, EDX, ST(0), XMM0, YMM0, ZMM0	EAX, ST(0), XMM0, YMM0, ZMM0	RAX, ST(0), XMM0, YMM0, ZMM0	RAX, RDX, ST(0), XMM0, YMM0, ZMM0

**Tabulka:** Použití registrů [3] (zjednodušeno).

# Volací konvence I

Volací konvence [3] určují, jak jsou funkce volány, kdo a kam umísťuje parametry, jak se vrací výsledek, a kdo – volající nebo volaný – odstraňuje použité parametry. V C je implicitní volací konvencí pro funkce `__cdecl` a pro metody `__thiscall`.

Konvence	Par. v reg.	Pořadí	Úklid
<code>__cdecl</code>	—	C	volající
<code>__stdcall</code>	—	C	volaný
<code>__pascal</code>	—	Pascal	volaný
GNU	—	C	oba
<code>__fastcall</code>	ECX, EDX	C	volaný
<code>__thiscall</code>	ECX	C	volaný

Tabulka: Volací konvence pro 32bitové architektury.

## Volací konvence II

64bitové volací konvence jsou mnohem jednodušší:

Konvence	Par. v reg.	Pořadí	Úklid
Windows	RCX, RDX, R8, R9	C	volající
Linux/BSD/OS X	RDI, RSI RDX, RCX, R8, R9	C	volaný

**Tabulka:** Volací konvence pro 64bitové architektury.

# Volací konvence III

Konvence je vždy součástí deklarace funkce/metody:

```
BOOL WINAPI MessageBeep( UINT uType );
```

je ve skutečnosti:

```
BOOL __stdcall MessageBeep( UINT uType );
```

a i ukazatel na tuto funkci musí obsahovat volací konvenci:

```
BOOL (__stdcall *pfnMB)(UINT) = (BOOL (__stdcall*)(UINT))GetProcAddress(...);
```

**Pozor!**

Vynechání nebo zaměnění volací konvence způsobuje velmi vážné chyby.

# Začínáme

Zatím jsem se zabývali tím:

- co je to reverzní inženýrství počítačového programu;
- co jsou aplikační binární rozhraní;
- co jsou volací konvence;
- k čemu potřebujeme name mangling.

Pojďme se teď pustit do skutečné práce — analýzy skutečné funkce<sup>1</sup>! Co budeme dělat:

- analyzovat rámec zásobníku funkce;
- analyzovat instrukce podmíněného skoku, abychom zjistili znaménka položek na zásobníku;
- analyzovat volání funkcí, abychom zjistili datové typy;
- rekonstruovat lokální proměnné funkce.

---

<sup>1</sup>pokud si v assembleru nejste jistí, запиšte si také předmět BI-SOJ!

# Reverzování funkcí

Každou funkci tvoří tři hlavní části:

- 1 prolog (volitelně),
- 2 tělo, a
- 3 epilog (volitelně).

**Prologem** rozumíme několik prvních instrukcí každé funkce, které vytvoří rámec zásobníku, alokují prostor pro lokální proměnné, zajistí zarovnání zásobníku a uloží všechny registry, které je nutno zachovat. Volitelně mohou na zásobník uložit kanárka a v případě použití strukturované obsluhy výjimek také strukturu `EXCEPTION_REGISTRATION`, včetně specifických rozšíření MSVC.

**Epilogem** rozumíme poslední instrukce funkce těsně před instrukcí `ret`. Jejich úkolem je zrušit rámec zásobníku. Pokud byl použit kanárek, je navíc ověřena jeho hodnota a v případě nesouladu je program ukončen.

V RE můžeme prolog rychle přejít a zaměřit se na tělo funkce.

# Prolog I

Při vstupu do funkce je na vrcholu zásobníku návratová adresa. Na ni ukazuje registr ESP. V tu chvíli začíná prolog:

## Typický prolog (MSVC/IA-32)

```
push    %ebp           // EBP musí být zachováno!  
mov     %esp, %ebp     // Položky na zásobníku odkazujeme relativně vůči EBP  
sub     $0x20, %esp    // Vytvoříme prostor 32 B pro lokální proměnné  
push    %ebx           // Pokud ho funkce přepisuje, uložíme EBX  
push    %esi           // Pokud ho funkce přepisuje, uložíme ESI  
push    %edi           // Pokud ho funkce přepisuje, uložíme EDI
```

Nyní můžeme používat EBP k odkazování na položky na zásobníku:

- EBP-20...EBP-1 odkazuje na lokální proměnné;
- EBP+ 0...EBP+3 odkazuje na uloženou hodnotu EBP;
- EBP+ 4...EBP+7 ukazuje na návratovou adresu;
- EBP+ 8...EBP+B ukazuje na první 4bajtový argument (pokud existuje);
- EBP+ C...EBP+F ukazuje na druhý 4bajtový argument (pokud existuje);

EBP+offset $\geq$ 8 odkazuje na argumenty, zatímco EBP-offset odkazuje na lokální proměnné.

## Prolog II

Architektura x86 má instrukci `enter`, která také vytváří rámec zásobníku. Následující dva prology jsou funkčně ekvivalentní. Počty  $\mu\text{op}$  platí pro architekturu Intel Haswell [3]:

### Standardní prolog pro architekturu IA-32

<code>push</code>	<code>%ebp</code>	<code>// 2 <math>\mu\text{ops}</math></code>
<code>mov</code>	<code>%esp, %ebp</code>	<code>// 1 <math>\mu\text{op}</math></code>
<code>sub</code>	<code>\$0x20, %esp</code>	<code>// 1 <math>\mu\text{op}</math></code>

### Prolog s instrukcí `enter`

<code>enter</code>	<code>\$0x20, \$0x0</code>	<code>// 12 <math>\mu\text{ops}</math></code>
--------------------	----------------------------	---

### Poznámka

Funkce `enter` se pro vytvoření rámce zásobníku používá jen výjimečně, protože je významně pomalejší než posloupnost `push/mov/sub`.



# Kanárky

Aby se ztížily útoky typu stack smashing (úmyslné přepsání dat na zásobníku jako důsledek speciálně zkonstruovaného vstupu do programu), vkládá se v prologu na zásobník náhodné slovo zvané **kanárek**. Před ukončením funkce je hodnota kanárku ověřována a v případě neshody není programu povoleno vrátit se z funkce — místo toho je okamžitě ukončen. To je nezbytné, protože útočník mohl přepsat návratovou adresu a provedením návratu z funkce by se zmocnil kontroly nad programem.

**Kanárek** je při startu náhodně vygenerován běhovým prostředím; může být jeden pro všechny funkce nebo pro každou funkci jiný. To je závislé na implementaci.

# Kanárky — GNU

Linuxový **Stack Protector** (`-fstack-protector`) používá slovo uložené v `[gs:0x14]`:

## Vložení kanárka

```
mov %gs:0x14, %eax  
mov %eax, -0xc(%ebp)
```

## Ověření kanárka

```
mov    -0xc(%ebp), %ecx  
xor     %gs:0x14, %ecx  
je      canary_check_ok  
call    <__stack_chk_fail@plt>  
canary_check_ok:  
// Program pokračuje
```

# Kanárky — MSVC Windows I

Windows nazývají mechanismus ochrany zásobníku **Buffer Security Check**. Na začátku programu je náhodně vygenerována globální hodnota `DWORD __security_cookie` (details na dalším slajdu) a ta je před uložením na zásobník v prologu funkce XORována s hodnotou EBP. V epilogu je ověřováno, že se hodnota na zásobníku nezměnila.

## Vložení kanárka

```
// Načtení cookie do EAX
mov  eax, [__security_cookie]

// XOR s EBP, získáme kanárka
xor  eax, ebp

// Uložení kanárka na zásobník
mov  [ebp-4], eax
```

## Ověření kanárka

```
// Načtení kanárka do ECX
mov  ecx, [ebp-4]

// Dopočítání __security_cookie
xor  ecx, ebp

// Ukončit program, pokud nesouhlasí
call j_@__security_check_cookie@4
```

# Kanárky — MSVC Windows II

```
unsigned int __security_init_cookie() {
    unsigned int result;
    LARGE_INTEGER perfctr;
    unsigned int cookie;
    FILETIME      systime;

    systime.ft_scalar = 0;
    if( __security_cookie != 0xBB40E64E // Jde o výchozí hodnotu?
        && (result = __security_cookie & 0xFFFF0000) != 0 )
        __security_cookie_complement = ~__security_cookie;
    else {
        GetSystemTimeAsFileTime((LPFILETIME)&systime);
        cookie = systime.ft_struct.dwHighDateTime;
        cookie ^= systime.ft_struct.dwLowDateTime;
        cookie ^= GetCurrentProcessId();
        cookie ^= GetCurrentThreadId();
        cookie ^= GetTickCount();
        QueryPerformanceCounter(&perfctr);
        result = perfctr.LowPart ^ cookie;
        cookie ^= perfctr.LowPart;
        cookie ^= perfctr.HighPart;
        if ( cookie == 0xBB40E64E ) // Cookie == výchozí?
```

# Kanárky — MSVC Windows III

```
    cookie = 0xBB40E64F;
    else if ( !(cookie & 0xFFFF0000) ) {
        result = cookie | (cookie << 16);
        cookie |= cookie << 16;
    }

    __security_cookie = cookie;
    __security_cookie_complement = ~cookie;
}

return result;
}
```

## Strukturovaná obsluha výjimek (32-bit) I

Strukturovaná obsluha výjimek (SEH) je specifikem Windows. MSVC C SEH (`__try`/`__except`/`__finally`) a C++ výjimky (`try`/`catch`) jsou nadstavbou nad SEH. Když dojde k výjimce na systémové úrovni (např. dělení nulou nebo dereference NULL ukazatele), je zavolán handler výjimky. Ten je definován v následující struktuře (uložené na zásobníku):

```
struct EXCEPTION_REGISTRATION {  
    EXCEPTION_REGISTRATION* prev;    // Ukazatel na předchozí E. R.  
    LPVOID handler; // Ukazatel na handler výjimky  
};
```

Ukazatel na nejvyšší `EXCEPTION_REGISTRATION` se nachází v prvním poli (FS: [0]) Thread Information Blocku (TIB), na který odkazuje registr FS.

## Strukturovaná obsluha výjimek (32-bit) II

Když dojde k výjimce, OS zavolá handler zjištěný v poli handler nejvyššího `EXCEPTION_REGISTRATION`. Handler se rozhodne, zda chce výjimku zpracovat. Pokud to odmítne, obrátí se OS na předcházející (nadřazený) `EXCEPTION_REGISTRATION`, na který ukazuje pole `prev`, a zavolá jeho handler. Tento postup se opakuje, dokud jeden z handlerů výjimku nezpracuje nebo dokud systém nenarazí na konec řetězce handlerů — v takovém případě použije OS “handler poslední záchrany”, který ukončí program nebo k němu připojí debugger, pokud je nějaký nainstalován.

### Instalace raw `EXCEPTION_REGISTRATION`

```
push ebp                // Začátek standardního prologu
mov  ebp, esp           // Zajistit ukazatel na rámec zásobníku
push MyExceptionHandler // Push EXCEPTION_REGISTRATION.handler
push fs:[0]             // Push EXCEPTION_REGISTRATION.prev
mov  fs:[0], esp        // Nastavit nejvyšší EXCEPTION_REGISTRATION
```

# Strukturovaná obsluha výjimek (32-bit) III

## Odstranění EXCEPTION\_REGISTRATION

```
mov eax, [ebp-8]    // Načíst adresu předchozího EXCEPTION_REGISTRATION do EAX
mov fs:[0], eax     // Nastavit tento EXCEPTION_REGISTRATION jako nejvyšší
```

Handler výjimky má následující prototyp:

```
EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD* ExceptionRecord,
    void* EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void* DispatcherContext
);
```

ExceptionRecord obsahuje informaci o výjimce, např. její kód, adresu, příznaky atd. Druhý parametr, EstablisherFrame, ukazuje na [EXCEPTION\\_REGISTRATION](#) v tom rámci, ve kterém výjimka nastala. ContextRecord obsahuje stav vlákna.



## Strukturovaná obsluha výjimek (32-bit) IV

Poté, co byl nalezen vyhovující handler, se řetězec handlerů prochází znovu až do tohoto místa, až na to, že `ExceptionRecord->ExceptionFlags` má nastavený příznak `EH_UNWINDING`. To znamená, že se zásobník bude čistit od všech rámců, až k rámcu použitého handleru. Handlers na cestě mají možnost provést svůj úklid (v C++ se volají destruktory, provádí se kód v bloku `__finally`, atd.).

Protože jde o značně náročnou úlohu, obsahuje runtime MSVC (nikoliv operační systém!) funkci `_except_handler3` a novější `_except_handler4`, které udělají většinu práce za vás! Struktura `EXCEPTION_REGISTRATION` pak ukazuje na MSVC handler `_except_handler3/4` a je rozšířena o dvě dodatečná pole (úroveň try a ukazatel na scope tabulku), která umožňují zpracovat vnořené `__try` bloky jediným handlerem. Struktura `EXCEPTION_REGISTRATION` je v tomto případě rozšířena:[6]

# Strukturovaná obsluha výjimek (32-bit) V

```
struct EH3_EXCEPTION_REGISTRATION {  
    EH3_EXCEPTION_REGISTRATION* prev;           // Ukazatel na předchozí E. R.  
    LPVOID handler;                             // Ukazatel na handler výjimky  
    SCOPETABLE_ENTRY* ScopeTable;               // Ukazatel na scope tabulku  
    DWORD TryLevel;                             // Identifikace try bloku  
};
```

Scope tabulka typicky leží v paměti jen pro čtení a v základní variantě (SEH3) má strukturu posloupnosti záznamů:

```
struct _SCOPETABLE_ENTRY {  
    DWORD EnclosingLevel;                       // Identifikace nadřazeného try bloku  
    LPVOID* FilterFunc;                         // Filtrovací funkce  
    LPVOID* HandlerFunc;                       // Zpracovací funkce  
};
```

SEH4 přidává další funkcionality, mimo jiné maskování ukazatele na scope tabulku pomocí security cookie.

## Strukturovaná obsluha výjimek (32-bit) VI

V tomto případě má celá funkce jediný handler výjimky dle SEH. Pro vstup do nového `try` bloku stačí zapsat jeho identifikátor to proměnné `TryLevel`. Naopak při zpracování výjimky funkce handleru přečte hodnotu proměnné `TryLevel` a adresu scope tabulky. `TryLevel` se použije jako index do tabulky a získáme jeden její řádek. Zavolá se funkce `FilterFunc`, která určí, zda má handler zájem o zpracovávanou výjimku. Pokud ano, vyvolá se `HandlerFunc`, která ji zpracuje, a nastaví se `TryLevel` na hodnotu `EnclosingLevel`. Pokud ne, použijeme `EnclosingLevel` jako index nadřazeného bloku, který je požádán o zpracování výjimky.

Nejvyšší záznam scope tabulky funkce má index -2 (`__try` z C) nebo -1 (`try` z C++). Pokud ani takto identifikovaný záznam nevede k vyřešení výjimky, obsluha SEH3 končí a vrací se ke standardnímu SEH z operačního systému.

Kompletní příprava obsluhy výjimek pak vypadá takto:

# Strukturovaná obsluha výjimek (32-bit) VII

```
00412AB0  push    ebp
00412AB1  mov     ebp,esp
00412AB3  push    0FFFFFFFh                // Aktuální try level je -2
00412AB5  push    418540h                  // Uložit ukazatel scope tabulky
00412ABA  push    411069h                  // Uložit adresu __except_handler4
00412ABF  mov     eax,dword ptr fs:[00000000h] // Načíst nejvyšší EXC_REG z TIB
00412AC5  push    eax                      // Uložit předchozí EXC_REG
00412AC6  sub     esp,10h
00412ACC  mov     eax,dword ptr ds:[__security_cookie]
00412AD1  xor     dword ptr [ebp-8],eax     // Zamaskovat ukazatel scope tab.
00412AD4  xor     eax,ebp
00412AD6  mov     dword ptr [ebp-1Ch],eax   // Kanárek
00412AD9  push    ebx
00412ADA  push    esi
00412ADB  push    edi
00412ADC  push    eax
00412ADD  lea     eax,[ebp-10h]             // EAX = & EXCEPTION_REGISTRATION
00412AE0  mov     dword ptr fs:[00000000h],eax // Nastavit nejvyšší EXC_REG v TIB
00412AE6  mov     dword ptr [ebp-18h],esp   // Uložit vrchol zásobníku
```

# Strukturovaná obsluha výjimek (32-bit) VIII

```
// Zde začíná blok __try
00412C74 mov     dword ptr [ebp-4],0           // Vstupujeme do nového try blooku
00412C7B mov     dword ptr ds:[0],0          // Dereference NULL ukazatele - crash!
00412C85 mov     dword ptr [ebp-4],0FFFFFFEh // Obnova úrovně try
00412C8C jmp     $LN6+18h (0412CC1h)         // Konec __try, pokračuj za ním

// Kód pro volání filtru výjimek
// MyExceptionFilter( GetExceptionCode(), GetExceptionInformation())
00412C8E mov     eax,dword ptr [ebp-14h]      // GetExceptionInformation()
00412C91 mov     ecx,dword ptr [eax]         //-> ExceptionRecord
00412C93 mov     edx,dword ptr [ecx]         //-> ExceptionCode
00412C95 mov     dword ptr [ebp-5Ch],edx     // Uložit kód výjimky do prac. prom.
00412C98 mov     eax,dword ptr [ebp-14h]     // GetExceptionInformation()
00412C9B push    eax                        // Uložit na zásobník
00412C9C mov     ecx,dword ptr [ebp-5Ch]     // GetExceptionCode()
00412C9F push    ecx                       // Uložit na zásobník
00412CA0 call    MyExceptionFilter (0411104h) // Zavolat filtr
00412CA5 add     esp,8                     // Vyčistit parametry ze zásobníku
00412CA8 ret                               // Návrat do handleru výjimky
```

# Strukturovaná obsluha výjimek (32-bit) IX

```
// Zde začíná blok __except
00412CA9 mov  esp,dword ptr [ebp-18h]
00412CAC push 417A50h
00412CB1 call dword ptr ds:[41A120h]
00412CB7 add  esp,4
00412CBA mov  dword ptr [ebp-4],0FFFFFFEh

// Obnovit nejvyšší EXCEPTION_REGISTRATION
00412CC1 mov  ecx,dword ptr [ebp-10h]
00412CC4 mov  dword ptr fs:[0],ecx

// Standardní epilog
00412CCB pop  ecx
00412CCC pop  edi
00412CCD pop  esi
00412CCE pop  ebx
00412CCF mov  esp,ebp
00412CD1 pop  ebp
00412CD2 ret

// Nastavit vrchol zásobníku
// Uložit parametry pro printf
// Zavolat printf
// Vyčistit parametry
// Návrat do top level try bloku

// Načíst EXC_REG.prev do ECX
// Nastavit nejvyšší EXC_REG v TIB
```

## Kdo instaluje handler poslední záchrany? I

Už jsme zmiňovali, že existuje handler operačního systému, který zpracuje jakoukoliv výjimku tím, že ukončí program. Odkud se tento handler bere? Podívejme se na posloupnost volání (call stack) v okamžiku, kdy bylo vytvořeno nové vlákno:

```
application!ThreadRoutine  // Toto je naše obsluha vlákna
    kernel32!BaseThreadInitThunk+0xe
        ntdll!__RtlUserThreadStart+0x70
            ntdll!_RtlUserThreadStart+0x1b

kernel32!BaseThreadInitThunk:
    761cee0a  mov     edi,edi
    761cee0c  push    ebp
    761cee0d  mov     ebp,esp
    761cee0f  test    ecx,ecx
    761cee11  jne     kernel32!BaseThreadInitThunk+0x15 (761cef64)
    761cee17  push    dword ptr [ebp+8]    // Uložit param. vlákna
    761cee1a  call    edx                  // Zavolat obsluhu vlákna
    761cee1c  push    eax                  // Uložit výsledek vlákna
    761cee1d  call    dword ptr [kernel32!_imp__RtlExitUserThread (7618170c)]
    // Nedosažitelný bod
```

Je zjevné, že BaseThreadInitThunk handler neinstaloval!

# Kdo instaluje handler poslední záchrany? II

Pokračujme dál a prozkoumejme `__RtlUserThread`:

`ntdll!__RtlUserThreadStart:`

```
77f237c4 push 14h
77f237c6 push offset stru_77f11278 // Obsahuje info o filtru a handleru
77f237cb call __SEH_prolog4
77f237d0 and [ebp+ms_exc.registration.TryLevel], 0 // Vstupujeme do __try bloku
77f237d4 mov eax, _Kernel32ThreadInitThunkFunction
77f237d9 push [ebp+c]
77f237dc test eax, eax
77f237de jz loc_77ec5e77
77f237e4 mov edx, [ebp+8]
77f237e7 xor ecx, ecx
77f237e9 call eax // _Kernel32ThreadInitThunkFunction
77f237eb mov [ebp+ms_exc.registration.TryLevel], 0fffffffh // Opouštíme __try blok
77f237f2 call __SEH_epilog4
77f237f7 retn 8
```

`77ec5e77 loc_77ec5e77:`

```
77ec5e77 call [ebp+8]
77ec5e7a push eax
77ec5e7b call _RtlExitUserThread@4
```



# Kdo instaluje handler poslední záchrany? III

Když tento kód přepíšeme do C, dostaneme následující:

```
void __stdcall __RtlUserThreadStart(  
    DWORD (__stdcall* pfnThreadProc)(LPVOID),  
    LPVOID pThreadArg  
)  
{  
    __try  
    {  
        DWORD dwResult;  
  
        if( Kernel32ThreadInitThunkFunction )  
            dwResult = Kernel32ThreadInitThunkFunction( NULL, pfnThreadProc, pThreadArg );  
        else  
        {  
            dwResult = pfnThreadProc( pThreadArg );  
            RtlExitUserThread(dwResult);  
        }  
    }  
    __except( RtlpGetExceptionFilter( GetExceptionInformation() ) )  
    {  
        ZwTerminateProcess( GetCurrentProcess(), GetExceptionCode() );  
    }  
}
```

# Strukturovaná obsluha výjimek — zneužití I

Zneužití SEH lze tak, že útočník přepíše pole handler v `EXCEPTION_REGISTRATION.handler` a způsobí vyvolání výjimky. Bude zavolán handler z `EXCEPTION_REGISTRATION` odkazované z `FS:[0]`, což je obvykle právě ten přepsaný, aby určil, jestli chce vyvolanou výjimku zpracovat. Útočník ho obvykle přepíše adresou posloupnosti instrukcí `pop-pop-ret`:

```
pop libovolný_registr // Odstraní návratovou adresu
pop libovolný_registr // Odstraní první parametr, t.j. ExceptionRegistration
ret                  // Vrátí se na adresu odkazovanou EstablisherFrame
```

**Proč pop-pop-ret a kam přesně se kód vrátí?**

# Strukturovaná obsluha výjimek — zneužití II

Call stack na vstupu do handleru obsahuje: (Windows 7)

```

application.exe!_except_handler
ntdll.dll!ExecuteHandler2
ntdll.dll!ExecuteHandler
ntdll.dll!_RtlDispatchException
ntdll.dll!_KiUserExceptionDispatcher
application.exe!FUNKCE_CO_VYVOLALA_VYJIMKU

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD* ExceptionRecord,
    void* EstablisherFrame,
    struct _CONTEXT* ContextRecord,
    void* DispatcherContext
);

```

Zásobník na vstupu do \_except\_handler obsahuje:

	Návratová adresa do ExecuteHandler2	ExceptionRecord*	EstablisherFrame* t.j. ukazatel na EXC_REG.*	ContextRecord*
0x0012F998	f9 71 92 77	80 fa 12 00	c8 fe 12 00	9c fa 12 00
0x0012F9A8	54 fa 12 00	c8 fe 12 00	0d 72 92 77	c8 fe 12 00
	Disp...Context*			

Instrukce pop-pop-ret (opkódy 5x 5y c3,  $x, y \in \{8, \dots, f\} \setminus \{c\}^2$ ) napřed odstraní z vrcholu zásobníku první dvě položky (návratovou adresu a `ExceptionRecord*`) a instrukce ret skočí na začátek `EXCEPTION_REGISTRATION`, kde leží pole prev. Toto pole útočník naplnil prvními 4 bajty kódu exploitu.

**Pozn.:** Tento přístup vyžaduje spustitelný zásobník.

<sup>2</sup>5c je pop esp a pro toto použití se nehodí.

# Epilog

Epilog je standardní zakončení funkce. Jeho účelem je zrušit rámec zásobníku, obnovit registry a vrátit řízení volajícímu. Volitelně (dle volací konvence) mohou být vyčištěny parametry na zásobníku.

Pokud se používají kanárky, tak je zde navíc kód ověřující hodnotu kanárka. Pokud hodnota nesouhlasí, je program ukončen.

## Typický epilog(MSVC/IA-32) v konvenci `__cdecl`

```
mov %ebp, %esp
pop %ebp
ret
```

## Typický epilog (MSVC/IA-32) v konvenci `__stdcall`

```
mov %ebp, %esp
pop %ebp
ret $0x8
```

## Typický epilog (GNU/IA-32)

```
leave
ret
```

## Typický epilog (GNU/x86\_64)

```
leaveq
retq
```

# Shrnutí

Když se nyní podíváme na funkci, měli bychom být schopni rychle poznat:

- prolog
  - kde začíná?
  - kolik bajtů zabírají lokální proměnné?
  - je zásobník zarovnaný?
  - používá funkce rámec zásobníku?
  - používá funkce kanárky?
  - používá funkce SEH?
  - pokud funkce očekává parametr v ECX, jde pravděpodobně o `__thiscall` (metoda) nebo `__fastcall` (další parametr může být v EDX).
- tělo funkce, bezprostředně následující za prologem, budeme zkoumat později :-)
- epilog
  - kde funkce končí?
  - testuje funkce kanárka?
  - obnovuje funkce `EXCEPTION_REGISTRATION`?
  - pokud funkce používá `ret XXX`, používá pravděpodobně konvenci `__stdcall` nebo `__thiscall`, jinak konvenci `__cdecl`.

# Počáteční analýza rámce zásobníku I

CFG [Přednáška 2] nám dá základní představu o logice kódu. My se nyní pokusíme lépe porozumět rámci zásobníku tím, že se podíváme na všechny instrukce, které s ním pracují (např. `mov`, `push`, `lea`, ...). Touto analýzou rozdělíme prostor vytvořený instrukcí `sub esp, __LOCAL_SIZE`<sup>3</sup>. V každé zkoumané instrukci se zaměřujeme na:

- ① offset — pozice položky v rámci zásobníku;
- ② velikostní modifikátor (byte, word, dword, ...) — nese informaci o velikosti položky.

Tímto rozčleníme rámec zásobníku na položky, u nichž **zatím neznáme datový typ a použití znaménka**, t.j. měli bychom ke značení používat pouze typy `UINT64`, `DWORD`, `WORD`, a `BYTE`. Protože neznáme jména proměnných, používáme obvykle generické názvy jako `local_offset` pro lokální proměnné a `arg_offset` pro parametry. V dalším testu budeme psát `l_off` a `a_off`, abychom ušetřili místo.

<sup>3</sup>MSVC zpřístupňuje symbol `__LOCAL_SIZE`, celkovou velikost lokálních proměnných v aktuální funkci. To je užitečné, pokud vytváříme funkci včetně jejího prologu a epilogu pomocí `__declspec(naked)`.

# Počáteční analýza rámce zásobníku II

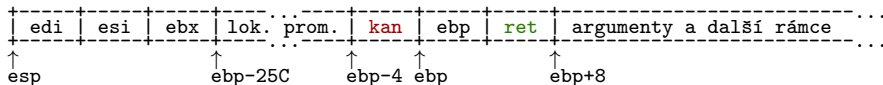
Podívejme se na ukázkový rámec zásobníku:

```

00412ad0  push    ebp
00412ad1  mov     ebp,esp
00412ad3  sub     esp,25Ch           // Alokuj 25C bajtů pro lok. prom.
00412ad9  mov     eax,dword ptr [00419000] // Načti __security_cookie do EAX
00412ade  xor     eax,ebp
00412ae0  mov     dword ptr [ebp-4],eax // Ulož kanárka
00412ae3  push    ebx
00412ae4  push    esi
00412ae5  push    edi                // Konec prologu

```

Po provedení prologu bude rámec zásobníku obsahovat:



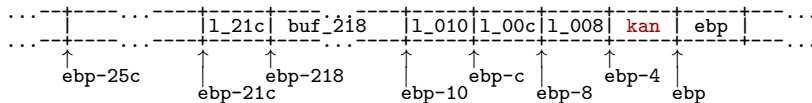
kde **kan** je kanárek a **ret** je návratová adresa.

## Počáteční analýza rámce zásobníku III

Pokračujeme dalším kouskem kódu:

```
00412ae6  mov     dword ptr [ebp-8],0
00412aed  mov     dword ptr [ebp-0Ch],0
00412af4  mov     dword ptr [ebp-10h],0
00412afb  mov     dword ptr [ebp-21Ch],0
00412b05  lea     eax,[ebp-218h]
```

První instrukce ukládá **DWORD** nulu do [ebp-8]. Toto místo je tedy **DWORD**. Totéž platí pro [ebp-C], [ebp-10], a [ebp-21C].



Vyznačili jsme 4 lokální proměnné `1_21c`, `1_010`, `1_00c`, a `1_008`, každou typu **DWORD**. Instrukce `lea` spočítá adresu bufferu neznámé délky začínajícího na `ebp-218`; označíme ho jako `buf_218`.



## Počáteční analýza rámce zásobníku IV

Pokračujeme dalším kouskem kódu:

```
00412b0b  push  eax
00412b0c  push  0
00412b0e  push  0
00412b10  push  25h
00412b12  push  0
00412b14  call  dword ptr [Tokens!_imp__SHGetFolderPathW (0041a17c)]
00412b1a  mov   dword ptr [ebp-21Ch],eax
00412b20  cmp   dword ptr [ebp-21Ch],0
00412b27  jge   loc_00412b2d
00412b29  jmp   loc_00412b97
```

Vidíme, že program volá API SHGetFolderPathW. Jde o veřejné Win32 API z knihovny shell32.dll. Můžeme z něj snadno odvodit velikost a typ bufferu, protože z předchozí instrukce víme, že EAX ukazuje na začátek buf\_218.

# Počáteční analýza rámce zásobníku V

Pokračujeme dalším kouskem kódu:

```
00412b0b  push  eax
00412b0c  push  0
00412b0e  push  0
00412b10  push  25h
00412b12  push  0
00412b14  call  dword ptr [Tokens!_imp__SHGetFolderPathW@412b14]
00412b1a  mov   dword ptr [ebp-21Ch],eax
00412b20  cmp   dword ptr [ebp-21Ch],0
00412b27  jge   loc_00412b2d
00412b29  jmp   loc_00412b97
```

```
SHFOLDERAPI SHGetFolderPathW(
    HWND hwnd,
    int csidl,
    HANDLE hToken,
    DWORD dwFlags,
    LPWSTR pszPath
);
// pszPath MAX_PATH=260×2 B long
```

Vidíme, že program volá API `SHGetFolderPathW`. Jde o veřejné Win32 API z knihovny `shell32.dll`. Můžeme z něj snadno odvodit velikost a typ bufferu, protože z předchozí instrukce víme, že `EAX` ukazuje na začátek `buf_218`. Funkce `SHGetFolderPathW` má následující prototyp, ale prozatím se jí nebudeme zabývat a API volání zanalyzujeme později.

## Počáteční analýza rámce zásobníku VI

```
00412b2d  push  offset Tokens!'string' (00417a18)
00412b32  lea    eax,[ebp-218h]
00412b38  push  eax
00412b39  call  dword ptr [Tokens!_imp__PathAppendW (0041a1ac)]
00412b3f  lea    eax,[ebp-218h]
00412b45  push  eax
00412b46  call  dword ptr [Tokens!_imp__LoadLibraryW (0041a054)]
00412b4c  mov    dword ptr [ebp-8],eax
00412b4f  cmp    dword ptr [ebp-8],0
00412b53  jne    loc_00412b64
```

Vidíme volání PathAppendW, která opět dostane náš buffer, a poté volání LoadLibraryW s naším bufferem. Návrátová hodnota API LoadLibraryW v registru EAX je uložena do 1\_008. Výsledek je otestován na nulu a blok končí.

## Počáteční analýza rámce zásobníku VII

```
00412b55  mov     dword ptr [ebp-10h],0
00412b5c  jmp     loc_00412b97
```

Zde vidíme větev **else** příkazu **if**, který jsme našli na konci předchozího bloku kódu. Jednoduše nastaví proměnnou `l_010` na nulu.

```
00412b64  push    offset Tokens!'string' (00417a38)
00412b69  mov     eax,dword ptr [ebp-8]
00412b6c  push    eax
00412b6d  call    dword ptr [Tokens!_imp__GetProcAddress (0041a044)]
00412b73  mov     dword ptr [ebp-0Ch],eax
00412b76  cmp     dword ptr [ebp-0Ch],0
00412b7a  je      loc_00412b86
```

Tento blok kódu volá `GetProcAddress` a ukládá výsledek do `l_00c`. Tento výsledek je opět porovnán na nulu.

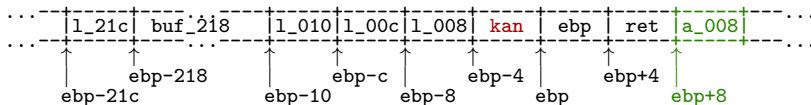
## Počáteční analýza rámce zásobníku VIII

```

00412b7c  mov     eax,dword ptr [ebp+8]
00412b7f  push    eax
00412b80  call    dword ptr [ebp-0Ch]
00412b83  mov     dword ptr [ebp-10h],eax
00412b86  mov     eax,dword ptr [ebp-8]
00412b89  push    eax
00412b8a  call    dword ptr [Tokens!_imp__FreeLibrary (0041a048)]
00412b90  mov     dword ptr [ebp-8],0

```

Tento kód čte z adresy `ebp+8`, kde je uložen první parametr, a to typu `DWORD`. Dále je volána funkce, jejíž adresa je uložena v `1_00c`. Nakonec je zavolána `FreeLibrary` a proměnná `1_008` vynulována.



## Počáteční analýza rámce zásobníku IX

Poslední část těla funkce nastaví registr EAX na hodnotu 1\_010. Jde o `return 1_010;`.

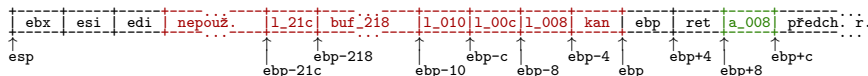
```
00412b97  mov     eax,dword ptr [ebp-10h]
```

Poslední částí funkce je epilog, který obnoví registry, ověří kanárka, a skončí. Funkce má nejméně jeden parametr a\_008. Nenarazili jsme na žádné odkazy na hodnoty vstupující v ECX ani EDX. Funkce používá pro návrat instrukci `ret` bez operandu. Z toho vyvodíme, že jde o funkci ve volací konvenci `__cdecl`.

```
00412b9a  pop     edi
00412b9b  pop     esi
00412b9c  pop     ebx
00412b9d  mov     ecx,dword ptr [ebp-4]
00412ba0  xor     ecx,ebp
00412ba2  call    Tokens!ILT+30(__security_check_cookie (00411023))
00412ba7  mov     esp,ebp
00412ba9  pop     ebp
00412baa  ret
```

# Počáteční analýza rámce zásobníku X

Konečná podoba rámce zásobníku je:



Zatím tedy máme:

```

DWORD __cdecl NeznámáFunkce( DWORD a_008 )
{
    DWORD l_21c;           // ebp-21c
    BYTE l_218[unknown];   // ebp-218
    DWORD l_010;           // ebp-010
    DWORD l_00c;           // ebp-00c
    DWORD l_008;           // ebp-008

    ...

    return l_010;
}

```

# Analýza znamének typů I

Informace, kterou jsme zatím získali, nám o datech neřekla o mnoho více než jejich délku. Nyní určíme, které proměnné jsou znaménkové.

Nejjednodušší je, podívat se na všechny aritmetické instrukce a jim odpovídající instrukce `cmp` a `jxx`. Pokud uvidíme instrukci `cmp/test` nebo aritmetickou operaci následovanou podmíněným skokem založeným na příznaku **CF**, víme, že datový typ je bez znaménka; pokud je založen na příznaku **SF** a/nebo **OF**, jde o znaménkový typ. Podívejme se na náš kód:

```
00412b20  cmp     dword ptr [ebp-21Ch],0
00412b27  jge     loc_00412b2d
00412b29  jmp     loc_00412b97
```

Protože `jge` skočí, pokud **SF=OF**, víme, že datový typ `1_21c` je se znaménkem a můžeme přepsat `DWORD` na `int`. Naneštěstí žádné další podobné instrukce nenacházíme, takže nemůžeme říci více.



# Analýza znamének typů II

Instrukce	CF	ZF	SF	OF	Význam	Signed/Unsigned
ja/jnbe	0	0			Jump if above	unsigned
jae/jnc	0				Jump if above or equal	unsigned
jb/jc/jnae	1				Jump if below	unsigned
jbe/jna	CF $\vee$ ZF				Jump if below or equal	unsigned
je/jz		1			Jump if equal	nelze určit
jne/jnz		0			Jump if not equal	nelze určit
jg/jnle		0	SF=OF		Jump if greater	signed
jge/jnl			SF=OF		Jump if greater or equal	signed
jl/jnge			SF $\neq$ OF		Jump if less	signed
jle/jng		ZF $\vee$ (SF $\neq$ OF)			Jump if less or equal	signed
jo				1	Jump if overflow	signed
jno				0	Jump if not overflow	signed
js			1		Jump if sign	signed
jns			0		Jump if not sign	signed

**Tabulka:** Instrukce podmíněného skoku a jejich použití při určení znaménkovosti typu.

# Analýza API volání I

Funkce, kterou analyzujeme, volá externí moduly. Tato volání vedou na funkce z dokumentovaného Windows API. Můžeme z nich odvodit typy, délky a **význam** parametrů použitých pro volání. To nám umožní opravit buffer `buf_218` a identifikovat datové typy všech zbylých proměnných.

- ❶ Vrátime-li se na slajd 57, vidíme, že poslední parametr funkce je buffer přesně `MAX_PATH` znaků dlouhý. Protože `MAX_PATH` má hodnotu 260 a každý znak má 2 bajty, víme, že buffer je 520 ( $(=208)_{16}$ ) bajtů dlouhý a zabírá tedy celé pole `buf_218`. Jeho deklarace je `WCHAR buf_218[MAX_PATH]`.
- ❷ Dále, funkce `SHGetFolderPathW` vrací hodnotu typu `HRESULT`. Tento typ je znaménkový (to už jsme také zjistili) a má 4 bajty (to také). Můžeme přepsat `int` na `HRESULT`.
- ❸ Volání `PathAppendW`: Toto API připojí řetězec za cestu s tím, že výsledek bude nejvýše `MAX_PATH` znaků dlouhý — nic nového.

## Analýza API volání II

- ④ Volání LoadLibraryW: Toto API přijme cestu k PE modulu, ten načte do paměti a vrátí jeho `HMODULE`. Výsledek je uložen v `1_008`, tato proměnná je tedy typu `HMODULE`.
- ⑤ Volání GetProcAddress: Toto API přijme `HMODULE` jako první argument a `char*`, jméno symbolu, jako druhý. Funkce vrátí ukazatel na symbol v daném modulu. Tento ukazatel je uložen do `1_00c`, a my už víme, že jde o ukazatel na funkci. Parametr předaný GetProcAddress nám říká jméno funkce (`SetProcessDEPPolicy`) a dále nám umožňuje zpřesnit typ `1_00c` tím, že určí volací konvenci (`__stdcall`) a typ návratové hodnoty `BOOL`.
- ⑥ Funkce, ukazatel na niž je uložen v `1_00c`, je volána a jejím argumentem je první argument `a_008` naší funkce. Funkce vrací `DWORD` (ale my víme, že to je 4bajtový `BOOL`), který je uložen do `1_010`.

# Analýza API volání III

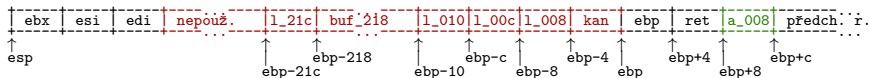
```

DWORD __cdecl NeznámáFunkce( DWORD a_008 )
{
    HRESULT l_21c; // ebp-21c, pol. 2
    WCHAR l_218[MAX_PATH]; // ebp-218, pol. 1
    DWORD l_010; // ebp-010, pol. 6
    BOOL (__stdcall *l_00c)(DWORD); // ebp-00c, pol. 5
    HMODULE l_008; // ebp-008, pol. 4

    ...

    return l_010;
}

```



## Analýza API volání IV

Po přejmenování proměnných a doplnění jejich inicializace na nulu získáváme:

```
DWORD __cdecl NeznámáFunkce( DWORD a_008 )
{
    HRESULT hr = S_OK;                // ebp-21c, pol. 2
    WCHAR wszLibraryPath[MAX_PATH];   // ebp-218, pol. 1
    DWORD dwResult = 0;               // ebp-010, pol. 6
    // ebp-00c, pol. 5, pol. 6
    BOOL (__stdcall *pfnSetProcessDEPPolicy)(DWORD) = NULL;
    HMODULE hLibrary = NULL;          // ebp-008, pol. 4

    ...

    return dwResult;
}
```

# Literatura I



Wikipedia Foundation, Inc.: *Software Engineering*, 2015,  
[https://en.wikipedia.org/wiki/Software\\_engineering](https://en.wikipedia.org/wiki/Software_engineering).



Chikofsky, E. J. and Cross, J. H.: *Reverse engineering and design recovery: A taxonomy*, IEEE Software 7: 13-17, 1990,  
<http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf>



Fog A.: *Calling conventions for different C++ compilers and operating systems*, 2014,  
[http://www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf).



Pietrek M.: *A Crash Course on the Depths of Win32™ Structured Exception Handling*, 1997,  
<http://www.microsoft.com/msj/0197/Exception/Exception.aspx>.



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

# Literatura II



Skochinsky, I.: *Compiler Internals: Exceptions and RTTI*, Recon, 2012,  
<https://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf>