Exploring the x64

（株）フォティーンフォティ技術研究所

はじめに　& 目標

- ~~IA64~~
- ~~Linux, *BSD, Mac, etc.~~

✓ x86で利用される様々なテクニックがx64ではどのようになっているかを明確にする

環境

- Windows 7 x64 Edition
- Visual Studio 2008
- Windbg
- IDA Pro Advanced
  - *Standard Editionは、x64未対応*

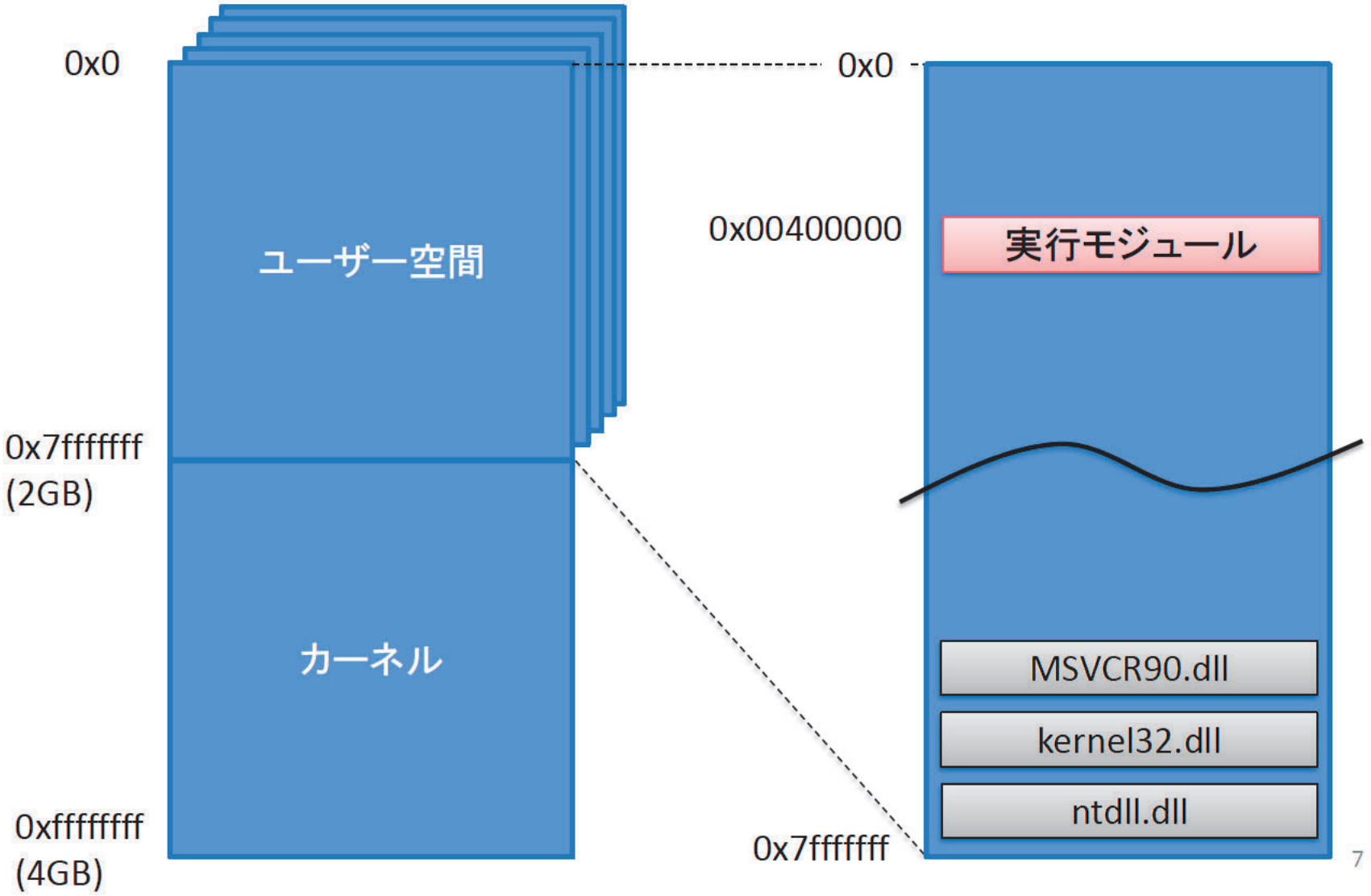
アジェンダ

- Windows x64
- ABI(Application Binary Interface)
- API Hooking
- Code Injection

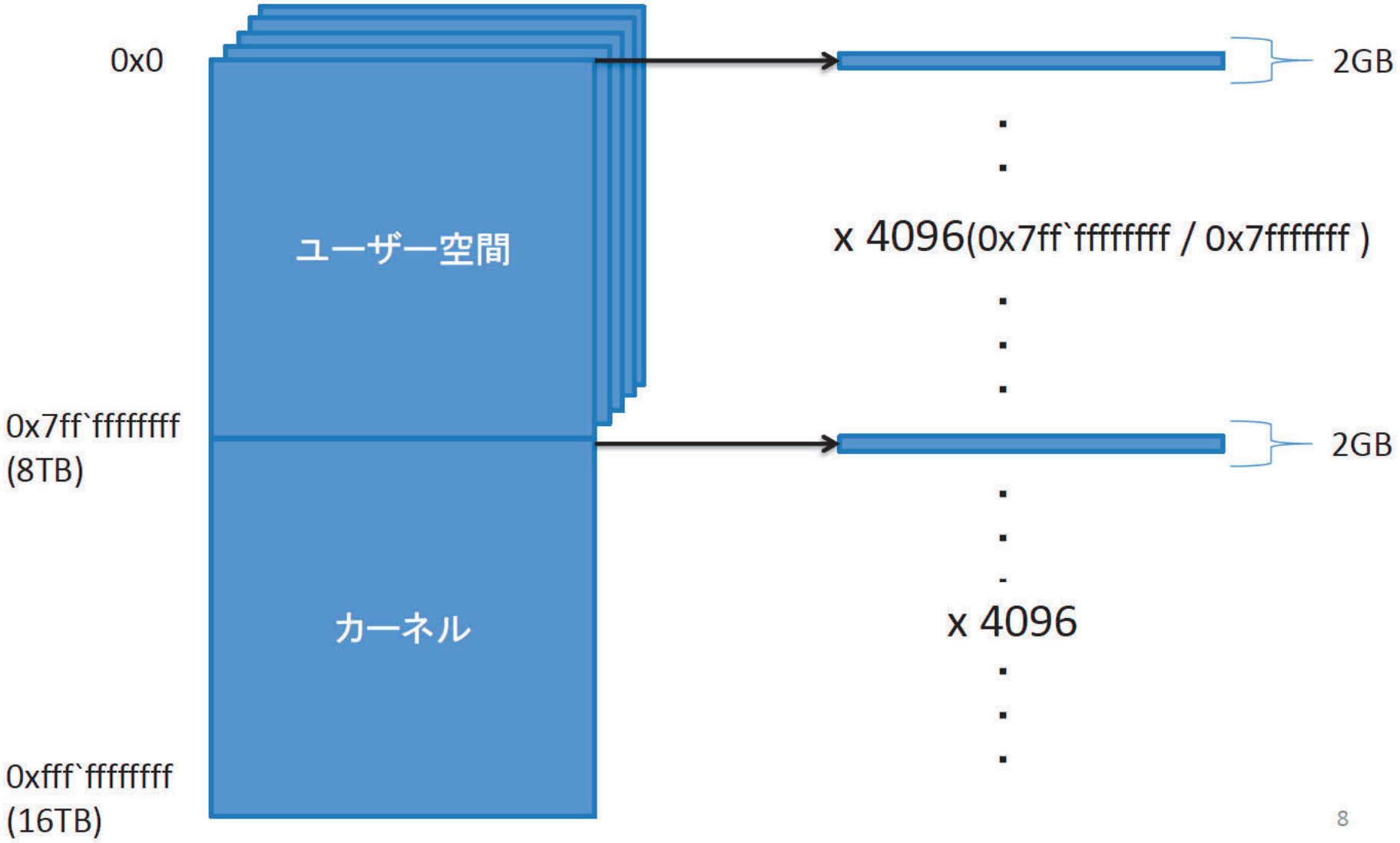
Windows x64

- Native x64 およびWoW64
- 仮想メモリ空間
  - 2^64 = 16 Exa Byte (Exa: 10^18)
  - 実際には、最大16TBに制限されている
- ファイル・レジストリリフレクション
- 64-bit用の追加API
  - IsWow64Process, GetNativeSystemInfo, etc.

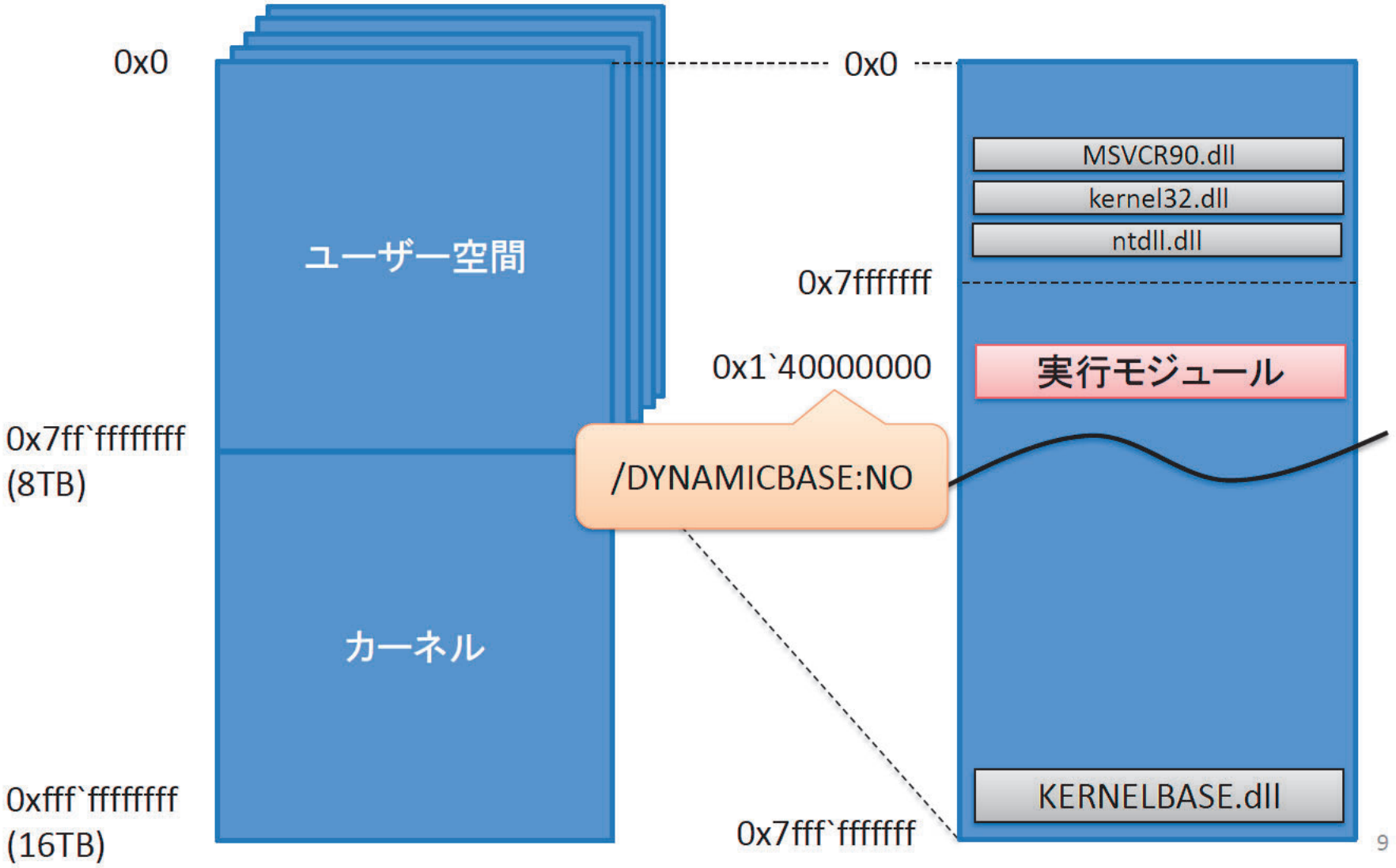
x86 – プロセスメモリ空間



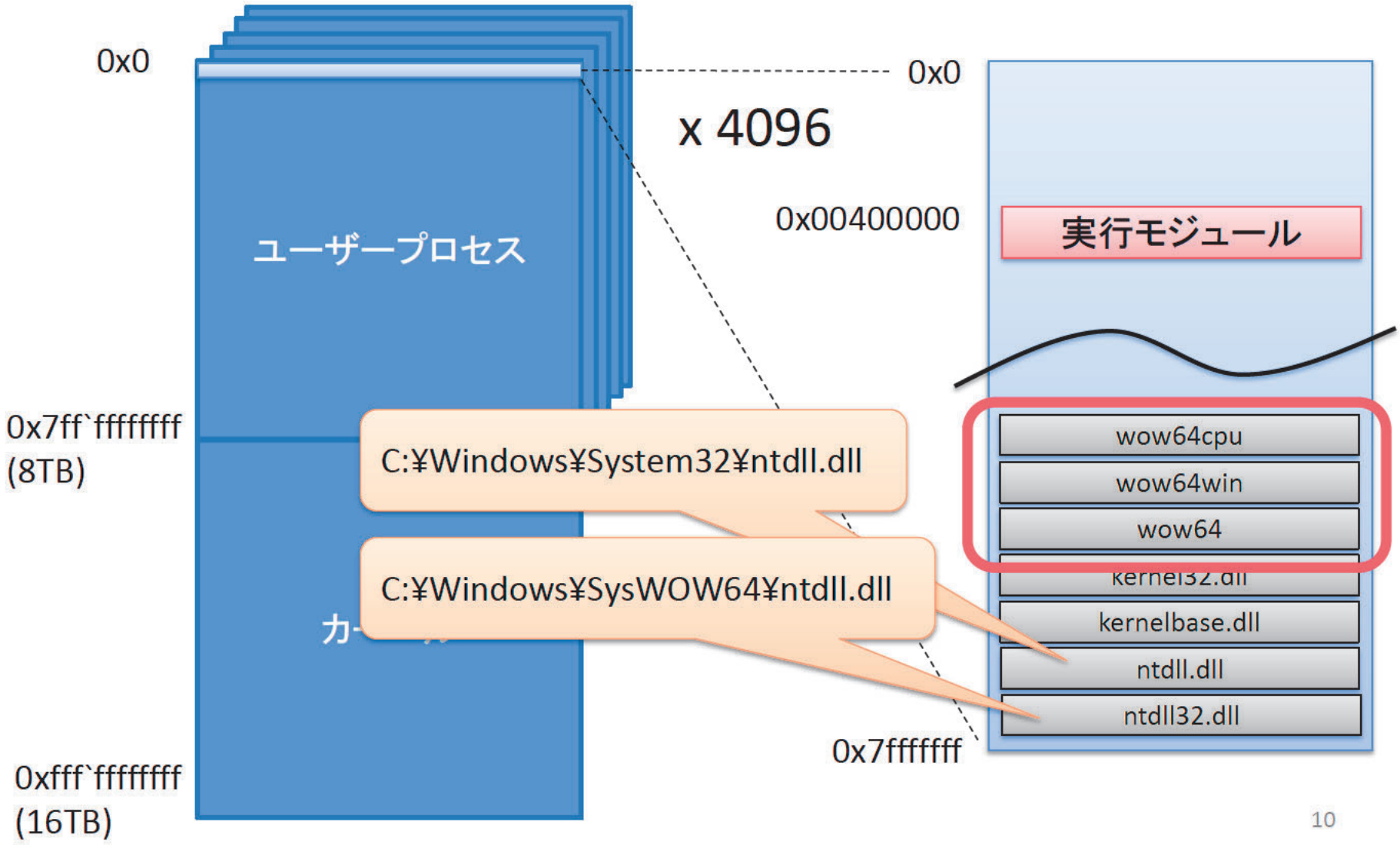
x64 – プロセスメモリ空間



x64 – プロセスメモリ空間



WoW64 – プロセスメモリ空間

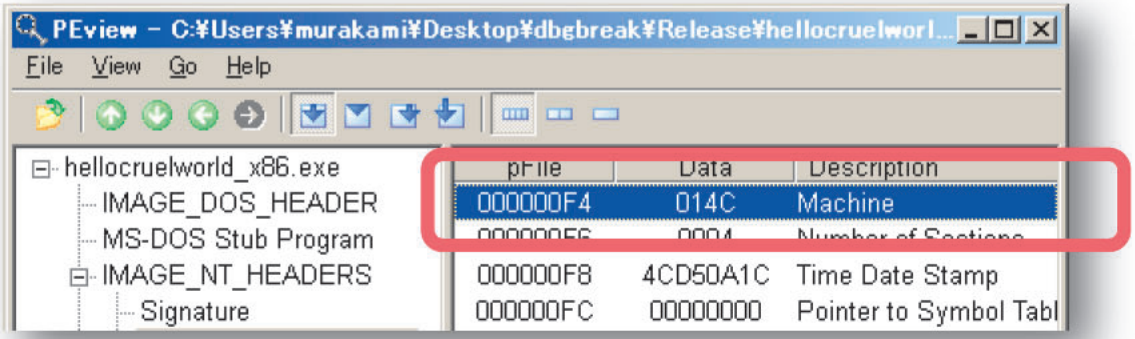


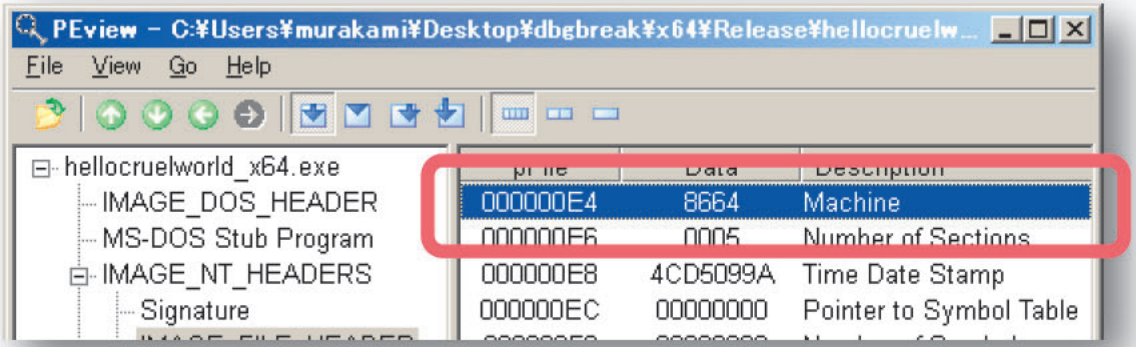
ABI

- バイナリ形式
- レジスタ
- 呼び出し規約
- 例外処理
- システムコール(x64, WoW64)

バイナリ形式 = PE32+

- 基本的な構造は、概ねPE32と同様
- IMAGE_NT_HEADERS.FileHeader.Machine
  - 0x014c => x86
  - 0x8664 => x64





バイナリ形式

- 次のフィールドサイズが64-bitsに拡張されている
  - IMAGE_NT_HEADERS.IMAGE_OPTIONAL_HEADER
    - ImageBase
    - SizeOfStackReserve
    - SizeOfStackCommit
    - SizeOfHeapReserve
    - SizeOfHeapCommit

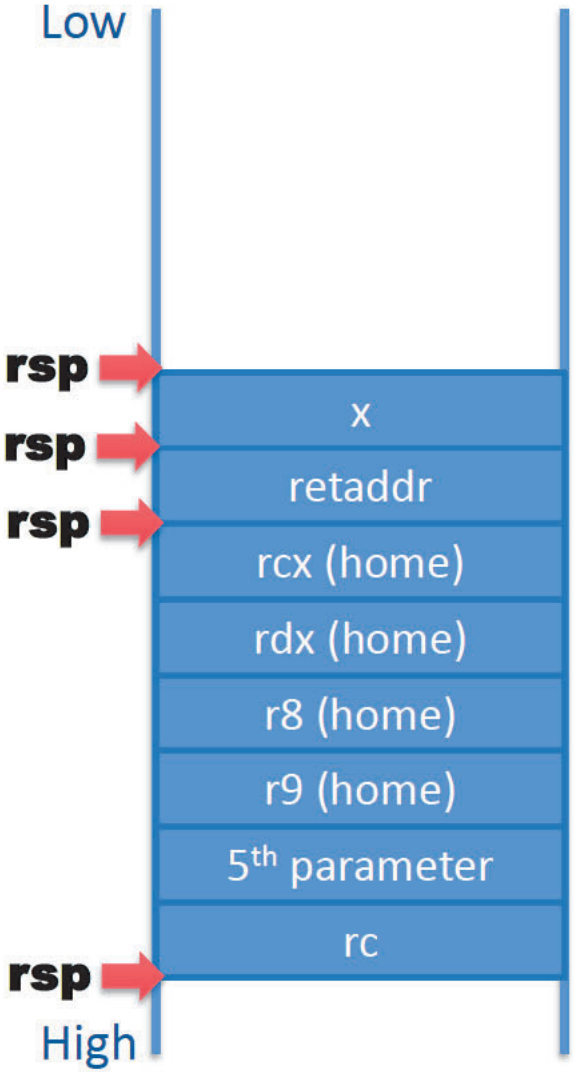
レジスタ

x86(32-bits)	x64(64-bits)	
EAX	RAX	R8
ECX	RCX	R9
EDX	RDX	R10
EBX	RBX	R11
ESI	RSI	R12
EDI	RDI	R13
ESP	RSP	R14
EBP	RBP	R15
EIP	RIP	

呼び出し規約

- 始めの4つの引数は、RCX, RDX, R8, R9レジスタを利用して渡される（5つ目以降は、スタック経由）
- 呼び出し側がスタック上にレジスタ・ホーム・スペースを確保
- 返り値は、x86同様RAXレジスタ経由で返却される
- リーフ関数・非リーフ関数
  - リーフ関数：スタックを一切利用しない関数
  - PE32+は、非リーフ関数の情報を例外ディレクトリに保存している
- レジスタの揮発性
  - 揮発レジスタ：RAX, RCX, RDX, R8-R11
  - 不揮発レジスタを関数内で変更する場合は、スタックを利用して保存・復元を行う必要がある

呼び出し規約




```
int foo(int a, int b, int c, int d, int e)
{
        int x = 0;

}

int
{
        call foo

        rc = foo(1, 2, 3, 4, 5);
        printf("%d¥n", rc);

        return rc;
}
```

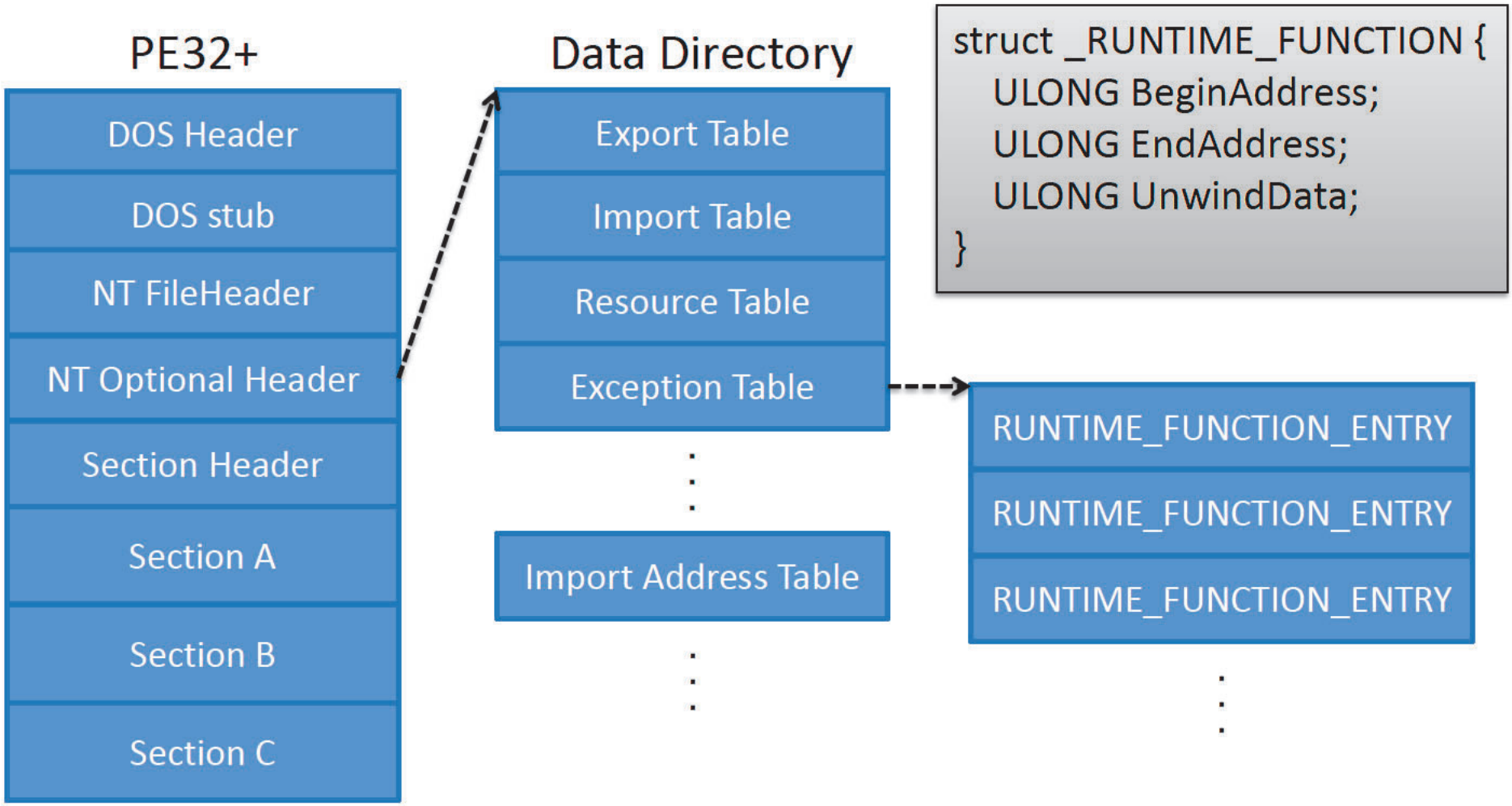
```
mov dword ptr [rsp+20h],r9d
mov dword ptr [rsp+18h],r8d
mov dword ptr [rsp+10h],edx
mov dword ptr [rsp+8h],ecx
sub  rsp,8h
```

例外処理

- テーブルベースでの処理
  - x86で利用されていたリンクリストでの管理は廃止

参考）x86での例外処理

例外ディレクトリ及びRUNTIME_FUNCTION構造体



dumpbin /unwindinfo

```
Begin    End      Info     Function Name

 00000000 00001000 00001041 000022D4  foo
   Unwind version: 1
   Unwind flags: None
   Size of prologue: 0x16
   Count of codes: 1
   Unwind codes:
     16: ALLOC_SMALL, size=0x18

 0000000C 00001050 00001095 000022DC  main
   Unwind version: 1
   Unwind flags: None
   Size of prologue: 0x04
   Count of codes: 1
   Unwind codes:
     04: ALLOC_SMALL, size=0x48
```

RUNTIME_FUNCTION.UnwindData

```
typedef struct _UNWIND_INFO {
    UBYTE Version         : 3;
    UBYTE Flags           : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister   : 4;
    UBYTE FrameOffset     : 4;
    UNWIND_CODE UnwindCode[1];
    union {
        // If (Flags & UNW_FLAG_EHANDLER)
        OPTIONAL ULONG ExceptionHandler;
        // Else if (Flags & UNW_FLAG_CHAININFO)
        OPTIONAL ULONG FunctionEntry;
    };
    // If (Flags & UNW_FLAG_EHANDLER)
    OPTIONAL ULONG ExceptionData[];
} UNWIND_INFO, *PUNWIND_INFO;
```

```
#define UNW_FLAG_NHANDLER 0x0
#define UNW_FLAG_EHANDLER 0x1
#define UNW_FLAG_UHANDLER 0x2
#define UNW_FLAG_CHAININFO 0x4
```

cf. http://www.osronline.com/article.cfm?article=469

ExceptionData

```
typedef struct _SCOPE_TABLE {
    ULONG Count;
    struct
    {
        ULONG BeginAddress;
        ULONG EndAddress;
        ULONG HandlerAddress;
        ULONG JumpTarget;
    } ScopeRecord[1];
} SCOPE_TABLE, *PSCOPE_TABLE;
```

cf. http://www.osronline.com/article.cfm?article=469

try/except

```
int main(void)
{
  int x = 0;

  __try {
    printf("%d¥n", 100/x);
    printf("foo¥n");
    printf("bar¥n");
    printf("baz¥n");
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    printf("catch!¥n");
  }

  return 0;
}
```

try/except

```
Name  main
Unwind version: 1
Unwind flags: EHANDLER
Size of prologue: 0x04
Count of codes: 1

Unwind codes
04: ALLOC_SMALL, size=0x28
Handler:0000165C __C_specific_handler

Count of scope table entries: 1
Begin   00001004
End     00001046
Handler 00000001
Target  00001046
```

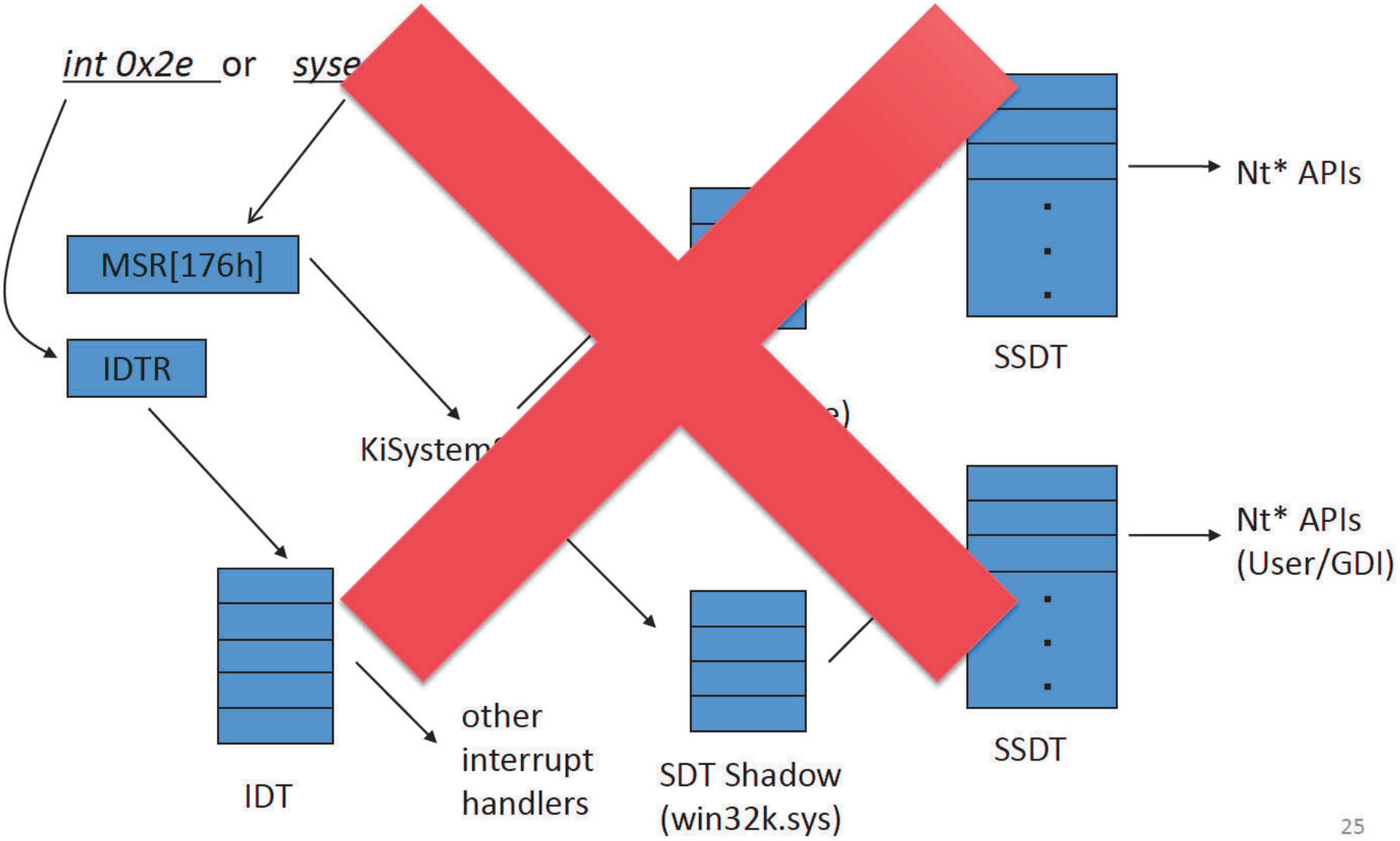
```
140001000  sub  rsp,28h
140001004  mov  eax,64h
140001009  cdq
14000100A  xor  ecx,ecx
14000100C  idiv eax,ecx
14000100E  mov  edx,eax
140001010  lea  rcx,[400021B0h]
140001017  call qword ptr [40002130h]
14000101D  lea  rcx,[400021B4h]
140001024  call qword ptr [40002130h]
14000102A  lea  rcx,[400021BCh]
140001031  call qword ptr [40002130h]
140001037  lea  rcx,[400021C4h]
14000103E  call qword ptr [40002130h]
140001044  jmp  0000000140001054

140001046  lea  rcx,[400021D0h]
14000104D  call qword ptr [40002130h]
140001053  nop
140001054  xor  eax,eax
140001056  add  rsp,28h
14000105A  ret
```

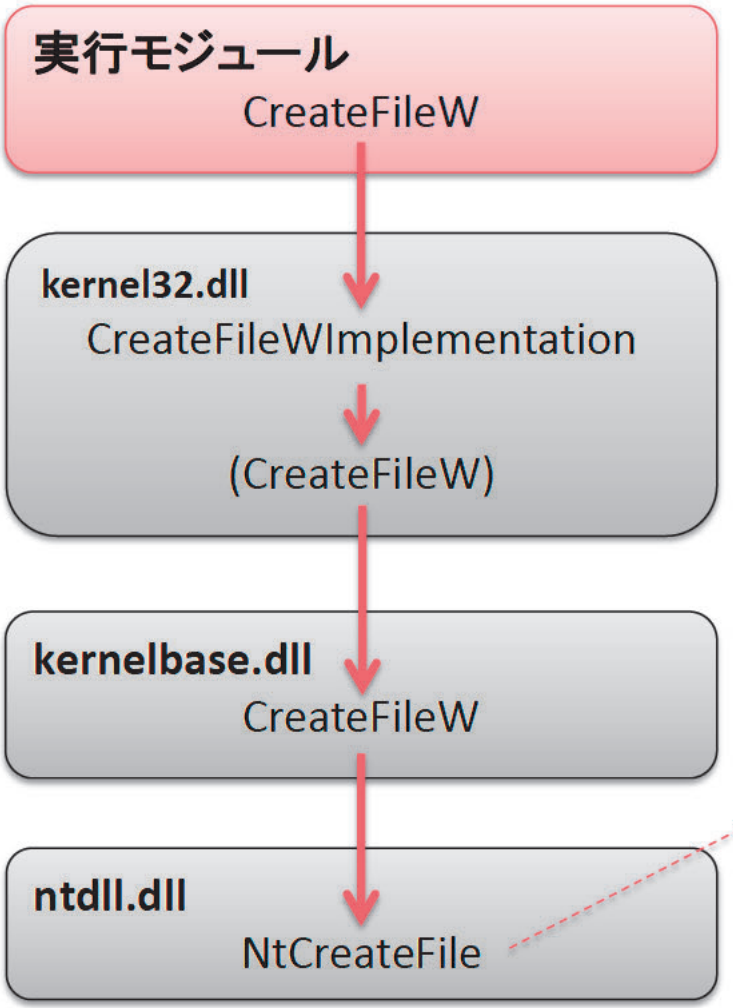
例外ディレクトリの効用

- 実行ファイル内の非リーフ関数を列挙可能
- 非リーフ関数それぞれについて:
  - 例外処理の情報を取得可能
  - スタック及び不揮発レジスタの用途を把握可能

システムコール x86

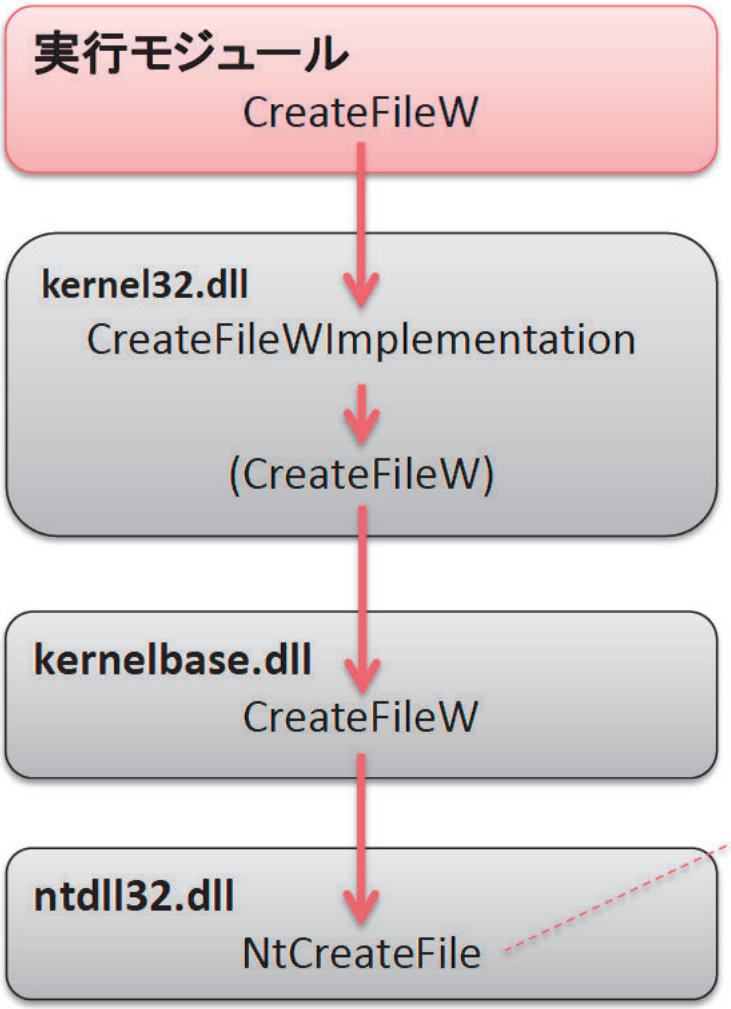

int 0x2e or syse
MSR[176h]
IDTR
KiSystem
SSDT
Nt* APIs
Nt* APIs
(User/GDI)
SSDT
IDT
other
interrupt
handlers
SDT Shadow
(win32k.sys)

システムコール x64




```
mov     r10,rcx
mov     eax,52h
syscall
ret
nop     dword ptr [rax+rax]
```

システムコール WoW64




```
mov     eax,52h
xor     ecx,ecx
lea     edx,[esp+4]
call    dword ptr fs:[0C0h]
add     esp,4
ret     2Ch
```

fs:[0C0h]

- FSレジスタは、TEB(Thread Environment Block) のアドレスを保持

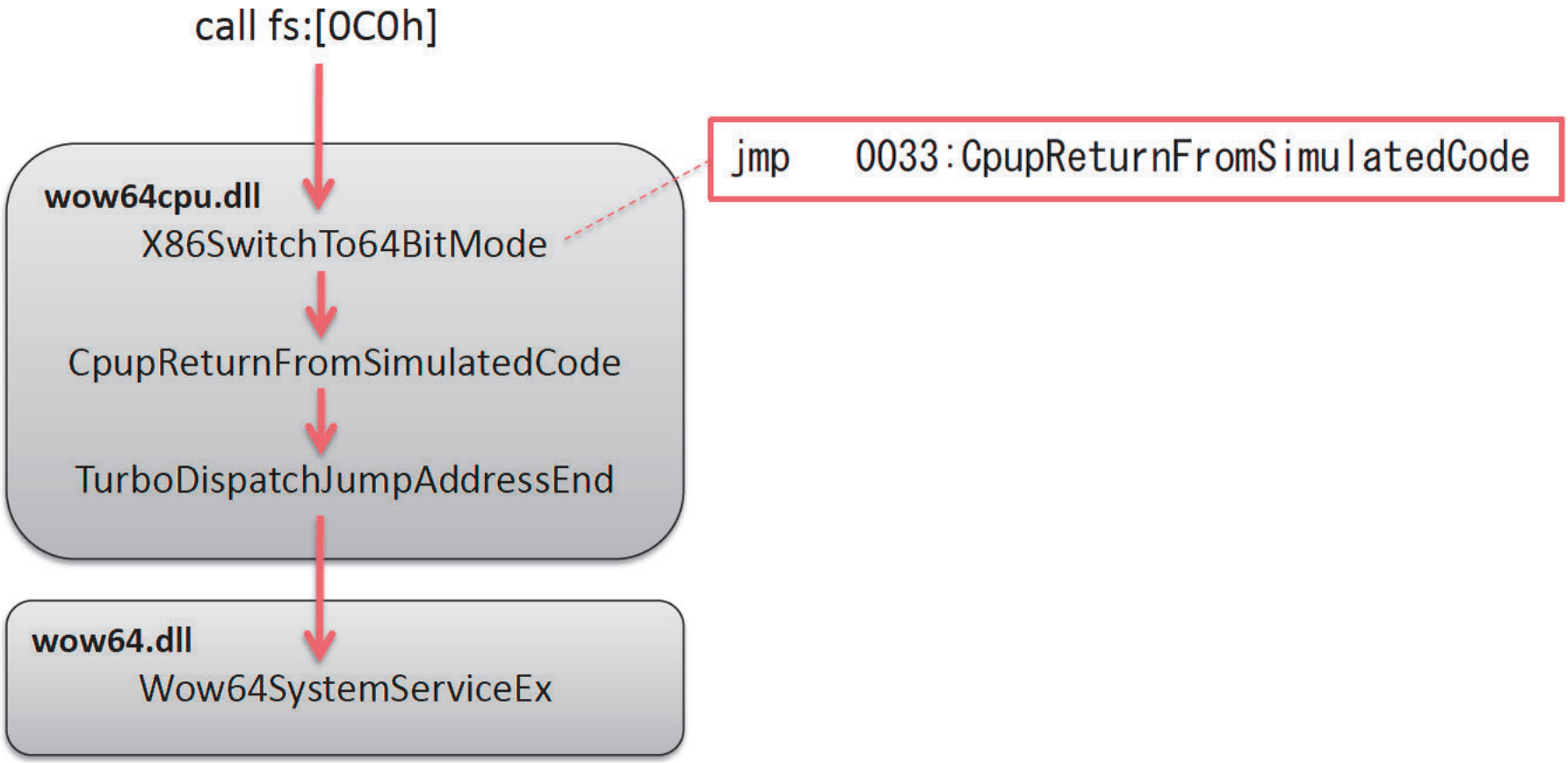
```
0:000:x86> dt _TEB
dbgbreak!_TEB
   +0x000 NtTib            : _NT_TIB
   ( ... )
```

```
0:000:x86> dd fs:[0C0h]
0053:000000c0  738c2320 00000411 00000000 00000000
```

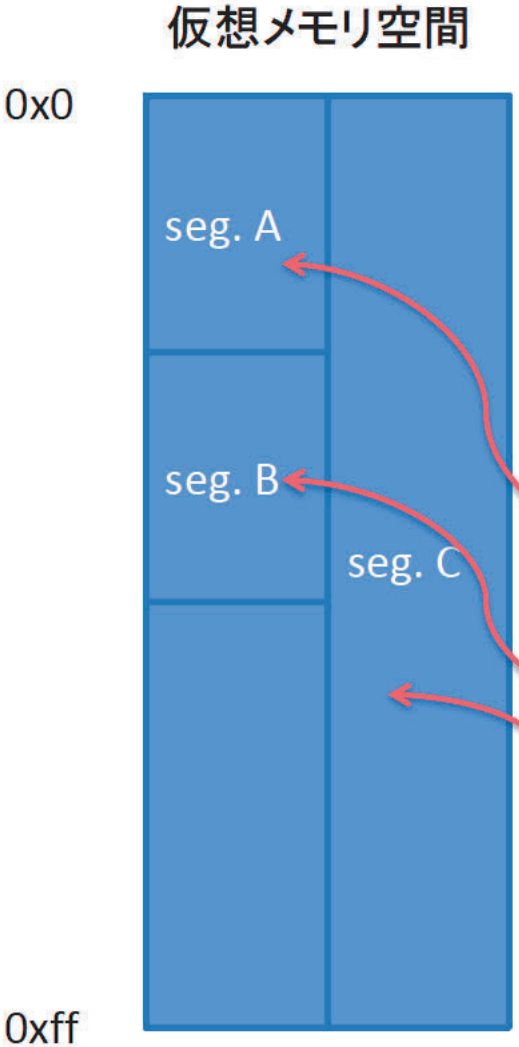
↑ X86SwitchTo64BitMode

```
   +0x040 Win32ThreadInfo  : Ptr32 Void
   +0x044 User32Reserved   : [26] Uint4B
   +0x0ac UserReserved     : [5] Uint4B
   +0x0c0 WOW32Reserved    : Ptr32 Void
```

Systemcall WoW64



GDT *(超約)*



- 仮想メモリ空間をセグメントとして管理
  - カーネルコード、カーネルデータ
  - ユーザコード、ユーザデータ等

GDT

ID	seg.	base	limit	type
0x00	A	0x0	0x3f	RW
0x08	B	0x40	0x7f	RE
0x10	C	0x0	0xff	RE

セグメントセレクタ

Windows 7(x64)でのGDTダンプ結果

```
kd> dg 0x00 0x60
                                                     P Si Gr Pr Lo
Sel        Base              Limit             Type    l ze an es ng Flags
---- ----------------- ----------------- ---------- - -- -- -- -- --------
0000 00000000`00000000 00000000`00000000 <Reserved> 0 Nb By Np Nl 00000000
0008 00000000`00000000 00000000`00000000 <Reserved> 0 Nb By Np Nl 00000000
0010 00000000`00000000 00000000`00000000 Code RE Ac 0 Nb By P  Lo 0000029b
0018 00000000`00000000 00000000`ffffffff Data RW Ac 0 Bg Pg P  Nl 00000c93
0020 00000000`00000000 00000000`ffffffff Code RE    3 Bg Pg P  Nl 00000cfa
0028 00000000`00000000 00000000`ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
0030 00000000`00000000 00000000`00000000 Code RE Ac 3 Nb By P  Lo 000002fb
0038 00000000`00000000 00000000`00000000 <Reserved> 0 Nb By Np Nl 00000000
0040 00000000`00b9b080 00000000`00000067 TSS32 Busy 0 Nb By P  Nl 0000008b
0048 00000000`0000ffff 00000000`0000f800 <Reserved> 0 Nb By Np Nl 00000000
0050 ffffffff`fffe0000 00000000`00003c00 Data RW Ac 3 Bg By P  Nl 000004f3
0058 00000000`00000000 00000000`00000000 <Reserved> 0 Nb By Np Nl 00000000
0060 00000000`00000000 00000000`ffffffff Code RE    0 Bg Pg P  Nl 00000c9a
```

Kernel CS(x64), Kernel DS

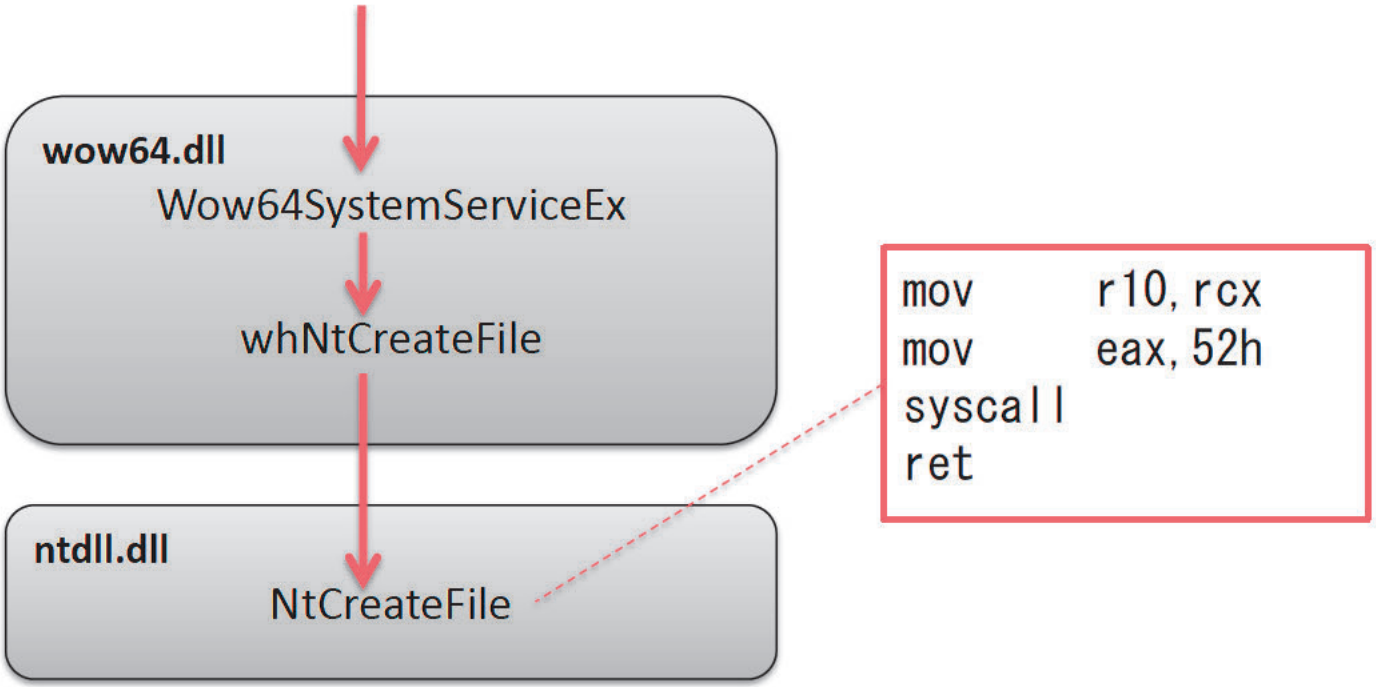
User CS(x86), User DS

User CS(x64)

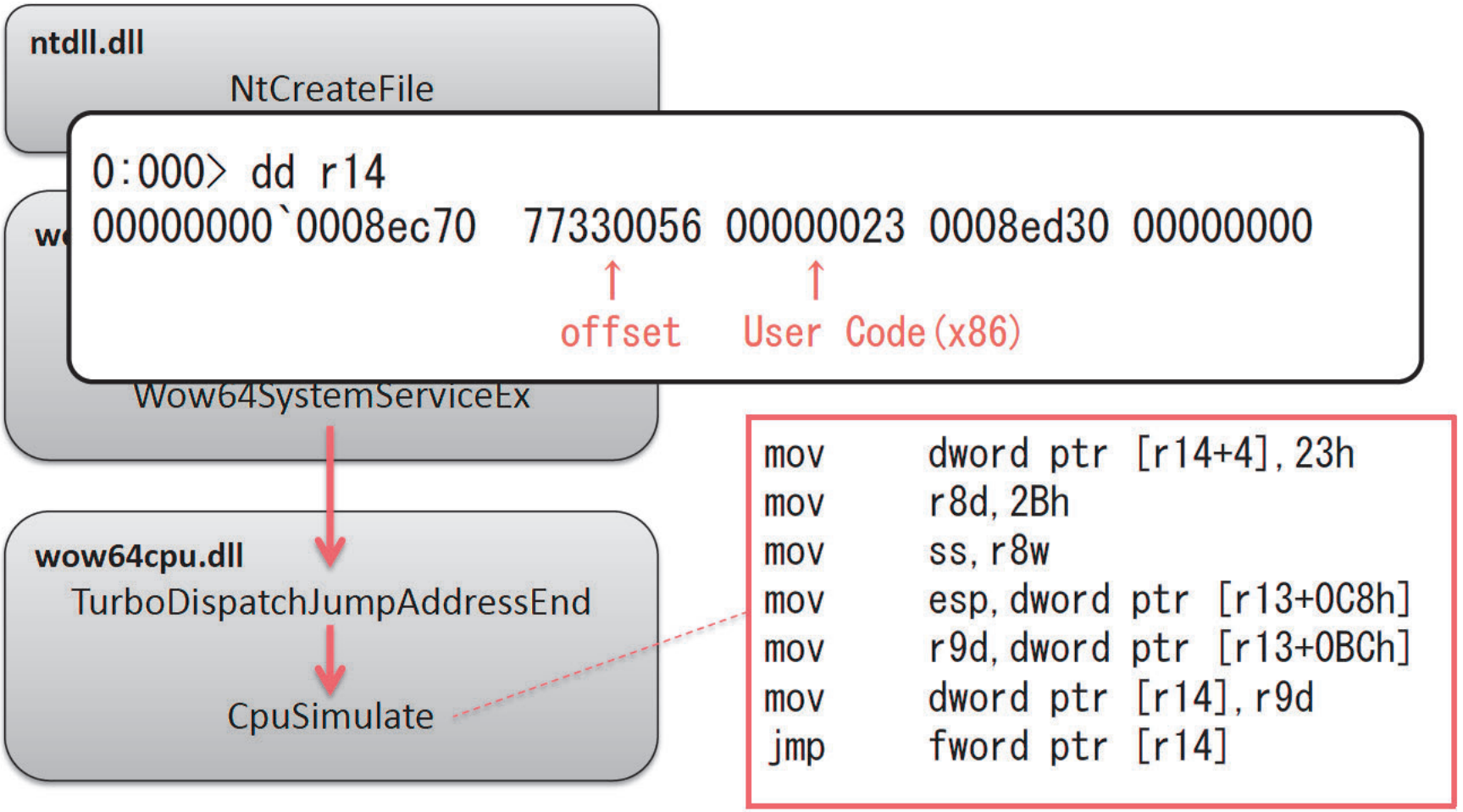
GDT[0x30]の内容

- ベース: 0x000`00000000
- リミット: 0x000`00000000
- タイプ: CODE, Read, Execute and Accessed
- 特権レベル: 3(ユーザーモード)
- L (64-bitコードセグメント) flag: 1

システムコール WoW64



システムコール WoW64 (return to x86)



デモ: WoW64からのx64 APIの直接呼出し

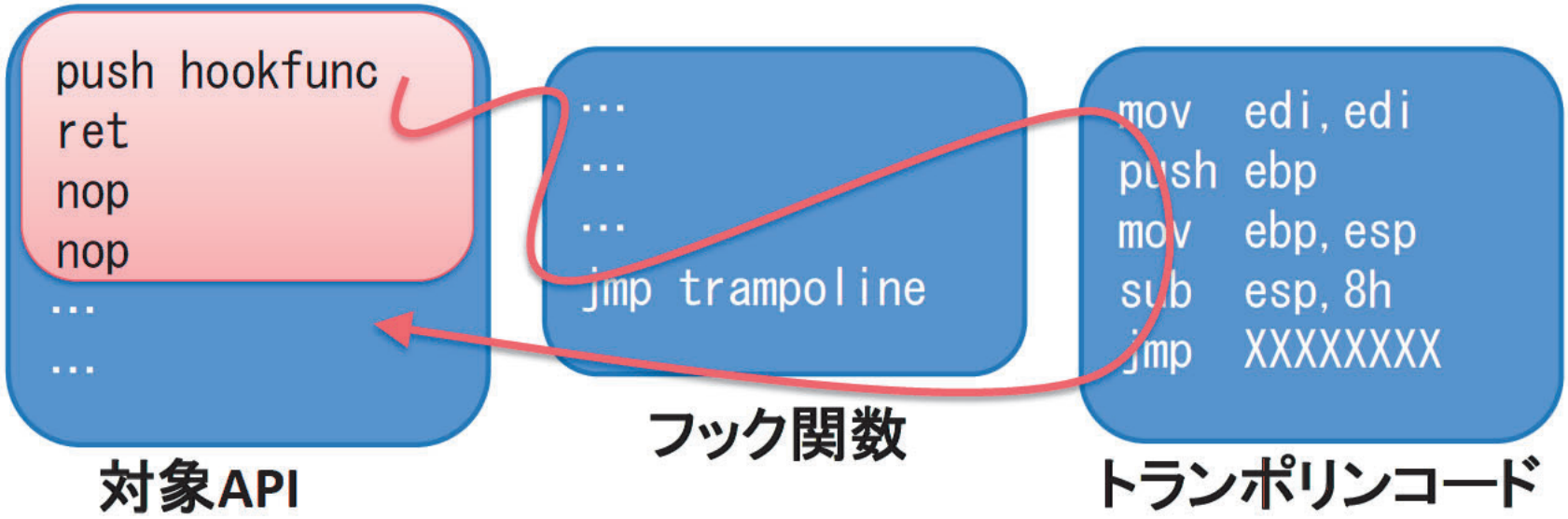
- x86からx64への遷移
  - jmp 0033:XXXXXXXX
- API呼び出し
  - rax: システムコール番号
  - rdx: 引数リストのアドレス
  - syscall
- x64からx86への遷移
  - call 0023:XXXXXXXX

API Hooking

- IAT Hooking
  - x86と同様の手法で実現可能
- Code Hooking

Code Hooking

- 基本的な方法は、x86と同じ
- ただし実装の詳細に若干の差異有り



REXプレフィックス

- 0x40～0x4E
  - x86: INC 、DEC 命令
  - x64: REX プレフィックス(レジスタ拡張)
- 例) 0x48,0xB8,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88

x86:

```
48                  dec     eax
B8 11 22 33 44      mov     eax, 44332211h
55                  push    ebp
66 77 88            ja      00004E9F
```

x64:

```
48 B8 11 22 33 44 55 66 77 88  mov     rax, 8877665544332211h
```

Code Hooking

```
00000000779811E4  mov     rax,7FFFFFA0028h
00000000779811EE  push    rax
00000000779811EF  ret
00000000779811F2  …
```

フック関数

```
000007FFFFFA0034  sub     rsp,38h
000007FFFFFA0038  xor     r11d,r11d
000007FFFFFA003B  cmp     dword ptr [7FFFFFC0F8Ch],r11d
000007FFFFFA0042  push    rax
000007FFFFFA0043  mov     rax,779811F2h
000007FFFFFA004D  xchg    rax,qword ptr [rsp]
000007FFFFFA0051  ret
```

書換えた元のコード

フックした関数への復帰先

Code Injection

- WoW64 からWoW64への
- x64からx64へのインジェ
- ~~WoW64 から x64 へのイン~~
- x64 からWoW64へのイン

(Fail CreateRemoteThead API)

x64環境は、x86マルウェアへの魔除けになるか?

まとめ

- Windows x64
- ABI(Application Binary Interface)
- API Hooking
- Code Injection