Practical Reverse Engineering

Chapter 1 Exercises Walkthrough

Set 1 (Page 11)

1. This function uses a combination of SCAS and STOS to do its work. First, explain what is the type of the [EBP+8] and [EBP+C] in line 1 and 8, respectively. Next, explain what this snippet does.

```
01: 8B 7D 08   mov   edi, [ebp+8]
02: 8B D7      mov   edx, edi
03: 33 C0      xor   eax, eax
04: 83 C9 FF   or    ecx, 0xFFFFFFFF
05: F2 AE      repne scasb
06: 83 C1 02   add   ecx, 2
07: F7 D9      neg   ecx
08: 8A 45 0C   mov   al, [ebp+0Ch]
09: 8B FA      mov   edi, edx
10: F3 AA      rep stosb
11: 8B C2      mov   eax, edx
```

I am going to assume cdecl is the calling convention used.

The cdecl calling convention pushes function arguments on the stack from right to left. Given an example function of...

```
void foo(int x, int y, int z);
```

This will generate assembly that looks like the following pseudo-code:

```
push arg_z
push arg_y
push arg_x
call foo
```

`call` pushes the address of the next instruction onto the stack, and the function prologue saves the current frame pointer by pushing it onto the stack before setting up a new stack frame.

The stack layout shortly after `foo` has been called looks like this:

ESP-relative	EBP-relative	Data
[esp]	[ebp]	saved_ebp
[esp+0x4]	[ebp+0x4]	return_address
[esp+0x8]	[ebp+0x8]	arg_x
[esp+0xC]	[ebp+0xC]	arg_y
[esp+0x10]	[ebp+0x10]	arg_z
[esp+0x14]	[ebp+0x14]	...

*Note: ESP-relative and EBP-relative are the same because this function has no local variables.

Looking at that assembly again, we see two function arguments: [ebp+8], and [ebp+0Ch]. Going by the example table above, it is easy to deduce that [ebp+8] is the function's first argument, and [ebp+0Ch] is the function's second argument.

This is what we know so far about the function's signature:

```
unknown_type mystery_func(void *arg1, void *arg2);
```

There is some preparation for `repne scasb`. Before going through that, let's understand how `repne scasb` works.

`repne` means “repeat while not equal” and uses the value of eax to test for equality.
`scasb` means “scan string byte” and uses edi as a pointer to a string of bytes.
`repne` also repeatedly decrements ecx by the size in bytes of the data being operated on.

This means `repne scasb` will keep on stepping through a string of bytes one byte at a time while decrementing ecx by one until ecx reaches zero or the first byte matches eax. (or, more specifically, the lowest 8 bits of eax known as al)

Let's look at the preparation to `repne scasb` now.

First, since `repne scasb` will modify the data in edi, edi's data is copied to edx for safe-keeping. Next, eax is set to 0 by xor-ing with itself. Every bit in ecx is switched on. Once `repne scasb` finishes, ecx will be equal to the the result of this equation:

ECX = (non_zero_byte_length + 2) * -1

This is because ecx started at -1 and counted down for each byte including the null byte.

Add 2 to ecx to account for the fact that it started at -1 instead of 0 as well as how it counted the terminating null byte. The `neg` instruction switches ecx from negative to positive.

Ecx now has the length of a string. The function's signature can be rewritten.

```
unknown_type mystery_func(char *str, void *arg2);
```

With the string length already in ecx, another `rep`-style instruction is coming up. Instead of zero, eax now has a byte which comes from the function's second argument.

`rep` means “repeat while ecx != 0”
`stosb` means “store string byte” and uses edi as a pointer to a string of bytes.
`stosb` also uses the data in eax to write to edi.

Each byte in the string is set to the byte in the second argument. The function therefore looks more like this:

```
unknown_type mystery_func(char *str, char c)
{
      memset(str, c, strlen(str));
}
```

There is one thing left, and that is the function's return type. The register typically responsible for storing the return type is eax. The last thing that happens to eax is the preserved value of edi, the string parameter which was kept in edx way back at the start, is copied into eax. This means the function is simply returning its first argument.

```
char* mystery_func(char *str, char c)
{
      memset(str, c, strlen(str));
      return str;
}
```

Set 2 (Page 17)

1. Given what you learned about CALL and RET, explain how you would read the value of EIP? Why can't you just do MOV EAX, EIP?

There are no such bytes to represent a `MOV EAX, EIP` or `LEA EAX, [EIP]` instruction, so the compiler will fail to produce a valid binary. 64-bit machines support RIP-relative addressing, so you can at least do `LEA RAX, [RIP]` to read the instruction pointer. `MOV RAX, RIP` is still invalid on x86_64, however.

Two functions to get the value of EIP are provided below.

```
/* 32 bit */
__attribute__((always_inline))
inline uint32_t read_eip()
{
    uint32_t eip;
    __asm( // 0xE8 == `call`
    ".byte 0xE8,0x00,0x00,0x00,0x00\n"
    "pop eax\n"
    : "=X"(eip));
    return eip;
}
```

```
/* 64 bit */
__attribute__((always_inline))
inline uint64_t read_rip()
{
    uint64_t rip;
    __asm( // 0xE8 == `call`
    ".byte 0xE8,0x00,0x00,0x00,0x00\n"
    "pop rax\n"
    : "=X"(rip));
    return rip;
}
```

2. Come up with at least two code sequences to set EIP to 0xAABBCCDD.

```
jmp 0xAABBCCDD

call 0xAABBCCDD

push 0xAABBCCDD
ret
```

3. In the example function, addme, what would happen if the stack pointer were not properly restored before executing RET?

Nothing, because this function does not move the stack pointer. If you meant to ask about the frame pointer, then what happens is the saved frame pointer remains on the stack and the `ret` instruction will pop it into EIP. The program will now be trying to interpret bytes on the stack as x86 instructions and the program will exhibit undefined behaviour if it has not already crashed.

4. In all of the calling conventions explained, the return value is stored in a 32-bit register (EAX). What happens when the return value does not fit in a 32-bit register? Write a program to experiment and evaluate your answer. Does the mechanism change from compiler to compiler?

If the return value is too large for the return value register, space will be allocated on the stack before the function is called and a pointer to the start of that allocated space is passed to the function. The function uses that pointer to write data to the buffer on the stack so when the function returns, the data to return to the caller is already within the caller's stack frame.

Set 3 (Page 35)

1. Repeat the walk-through by yourself. Draw the stack layout, including parameters and local variables.

Address	Name	Type
EBP-130h	PROCESSENTRY32 procentry	Local variable (struct)
EBP-128h	procentry.th32ProcessID	Local variable (field)
EBP-118h	procentry.th32ParentProcessID	Local variable (field)
EBP-10Ch	procentry.szExeFile	Local variable (field)
EBP-8	IDTR idtr	Local variable (struct)
EBP-6	idtr.base	Local variable (field)
EBP	<Saved Frame Pointer>	x86 control data
EBP+4	<Saved Return Address>	x86 control data
EBP+8	LPVOID lpvReserved	Function parameter
EBP+Ch	DWORD fdwReason	Function parameter
EBP+10h	HINSTANCE hinstDLL	Function parameter

*Note: There is memory accessed in the instruction `lea edi,[ebp-12Ch]` not accounted for in the stack diagram. Memory is accessed here to use `rep stosd` which fills out 0x49 * 4 = 0x124 bytes of the 0x128 bytes of the struct. The remaining 4 bytes are manually written to in the instruction `mov dword ptr [ebp-130h], 0`. I don't know why this separate `mov` is used, seeing as `rep stosd` could have filled out these bytes if only the starting values in `ecx` and `edi` had been slightly adjusted.

2. In the example walk-through, we did a nearly one-to-one translation of the assembly code to C. As an exercise, re-decompile this whole function so that it looks more natural. What can you say about the developer's skill level/experience? Explain your reasons. Can you do a better job?

```
typedef struct _IDTR {
    DWORD base;
    SHORT limit;
} IDTR, *PIDTR;

BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    IDTR idtr;
    __sidt(&idtr);
    if (idtr.base <= 0x80047400 || idtr.base > 0x8003F400)
        return 0;

    PROCESSENTRY32 procentry;
    memset(&procentry, 0x00, sizeof(procentry));
```

```
        HANDLE h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
        if (h == -1)
            return 0;

        procentry.dwSize = sizeof(procentry); // 0x128
        int ret = Process32First(h, &procentry);
        while (ret) {
            if (_stricmp(procentry.szExeFile, "explorer.exe"))
                break;
            ret = Process32Next(h, &procentry);
        }

        if (ret) {
            if (procentry.th32ParentProcessID == procentry.th32ProcessID)
                return 0;
        } else
            return 0;

        if (fdwReason == DLL_PROCESS_ATTACH)
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)0x100032D0, 0, 0, NULL);

        return 1;
    }
```

3. In some of the assembly listings, the function name has a @ prefix followed by a number. Explain when and why this decoration exists.

Functions that start with "_" and have a "@n" postfix indicate the stdcall calling convention. The "n" in "@n" tells you how many bytes are used for the function's parameters.

4. Implement the following functions in x86 assembly: strlen, strchr, memcpy, memset, strcmp, strset.

All solutions are implemented using the cdecl calling convention on a 32bit architecture.

strlen:

```
        mov   edi, [ebp+8] ; const char *s
        xor   eax, eax
        or    ecx, 0xFFFFFFFF
        cld
        repne scasb
        add   ecx, 2
        neg   ecx
        mov   eax, ecx
```

strchr:

```
        mov   eax, [ebp+8] ; int c
        mov   edi, [ebp+c] ; const char *s
        or    ecx, 0xFFFFFFFF
        cld
        repne scasb
        mov eax, edi
```

memcpy:

```
        mov   edi, [ebp+10] ; void *dest;
        mov   esi, [ebp+c]  ; const void *src;
        mov   ecx, [ebp+8]  ; size_t n;
        cld
        rep movsb
```

memset:

```
        mov   edi, [ebp+10] ; void *s;
        mov   al,  [ebp+c]  ; int c;
        mov   ecx, [ebp+8]  ; size_t n
        cld
        rep stosb
```

strcmp:

```
        mov   edi, [ebp+c] ; const char *s1
        mov   esi, [ebp+8] ; const char *s2
        cld
        xor   eax, eax
        repne cmpsb
```

strset:

```
      ; deprecated
```

5. Decompile the following kernel routines in Windows:

KeInitializeDpc:

The `KeInitializeDpc` function prototype is given here.

Get the definition of a KDPC structure by executing `dt _KDPC` in a windbg kernel debugging session.

```
struct _KDPC {
    +0x000 Type             : UChar
    +0x001 Importance       : UChar
```

```
    +0x002 Number           : Uint2B
    +0x008 DpcListEntry     : _LIST_ENTRY
    +0x018 DeferredRoutine  : Ptr64 void
    +0x020 DeferredContext  : Ptr64 Void
    +0x028 SystemArgument1  : Ptr64 Void
    +0x030 SystemArgument2  : Ptr64 Void
    +0x038 DpcData          : Ptr64 Void
};
```

This struct contains a _LIST_ENTRY struct. For completeness and because we will see this structure often, here is the definition of a _LIST_ENTRY:

```
struct _LIST_ENTRY {
    +0x000 Flink  : Ptr64 _LIST_ENTRY
    +0x008 Blink  : Ptr64 _LIST_ENTRY
};
```

Get the disassembly by executing `u KeInitializeDpc` in a windbg kernel debugging session.

```
    fffff800`026ba358 33c0            xor     eax,eax
    fffff800`026ba35a c60113          mov     byte ptr [rcx],13h
    fffff800`026ba35d c6410101        mov     byte ptr [rcx+1],1
    fffff800`026ba361 48895118        mov     qword ptr [rcx+18h],rdx
    fffff800`026ba365 4c894120        mov     qword ptr [rcx+20h],r8
    fffff800`026ba369 66894102        mov     word ptr [rcx+2],ax
    fffff800`026ba36d 48894138        mov     qword ptr [rcx+38h],rax
    fffff800`026ba371 c3              ret
```

With the knowledge of the assembly, the structure definition, and the function prototype, we can now see the decompiled code is as follows:

```
void KeInitializeDpc(
  __drv_aliasesMem PRKDPC Dpc,
  PKDEFERRED_ROUTINE      DeferredRoutine,
  __drv_aliasesMem PVOID  DeferredContext
)
{
    Dpc->Type = 19;
    Dpc->Importance = 1;
    Dpc->DeferredRoutine = DeferredRoutine;
    Dpc->DeferredContext = DeferredContext;
    Dpc->Number = 0;
    Dpc->DpcData = NULL;
}
```

KeInitializeApc:

This function prototype is not in the Microsoft documentation, but you can find it in “include/ddk/winddk.h” which is located in your compiler's installation directory.

```
Void KeInitializeApc(
  PKAPC              Apc,
  PKTHREAD           Thread,
  UCHAR              StateIndex,
  PKKERNEL_ROUTINE   KernelRoutine,
  PKRUNDOWN_ROUTINE  RundownRoutine,
  PKNORMAL_ROUTINE   NormalRoutine,
  UCHAR              Mode,
  PVOID              Context
);
```

Here are the definitions of the KAPC and KTHREAD structures used in this function.

```
struct _KAPC {
    +0x000 Type              : UChar
    +0x001 SpareByte0        : UChar
    +0x002 Size              : UChar
    +0x003 SpareByte1        : UChar
    +0x004 SpareLong0        : Uint4B
    +0x008 Thread            : Ptr64 _KTHREAD
    +0x010 ApcListEntry      : _LIST_ENTRY
    +0x020 KernelRoutine     : Ptr64 void
    +0x028 RundownRoutine    : Ptr64 void
    +0x030 NormalRoutine     : Ptr64 void
    +0x038 NormalContext     : Ptr64 Void
    +0x040 SystemArgument1   : Ptr64 Void
    +0x048 SystemArgument2   : Ptr64 Void
    +0x050 ApcStateIndex     : Char
    +0x051 ApcMode           : Char
    +0x052 Inserted          : Uchar
};
```

```
struct _KTHREAD {
    +0x000 Header            : _DISPATCHER_HEADER
    +0x018 CycleTime         : Uint8B
    +0x020 QuantumTarget     : Uint8B
    +0x028 InitialStack      : Ptr64 Void
    +0x030 StackLimit        : Ptr64 Void
    +0x038 KernelStack       : Ptr64 Void
    +0x040 ThreadLock        : Uint8B
    +0x048 WaitRegister      : _KWAIT_STATUS_REGISTER
    +0x049 Running           : UChar
    +0x04a Alerted           : [2] UChar
    +0x04c KernelStackResident : Pos 0, 1 Bit
    +0x04c ReadyTransition   : Pos 1, 1 Bit
    +0x04c ProcessReadyQueue : Pos 2, 1 Bit
    +0x04c WaitNext          : Pos 3, 1 Bit
```

```
   +0x04c SystemAffinityActive : Pos 4, 1 Bit
   +0x04c Alertable        : Pos 5, 1 Bit
   +0x04c GdiFlushActive   : Pos 6, 1 Bit
   +0x04c UserStackWalkActive : Pos 7, 1 Bit
   +0x04c ApcInterruptRequest : Pos 8, 1 Bit
   +0x04c ForceDeferSchedule : Pos 9, 1 Bit
   +0x04c QuantumEndMigrate : Pos 10, 1 Bit
   +0x04c UmsDirectedSwitchEnable : Pos 11, 1 Bit
   +0x04c TimerActive      : Pos 12, 1 Bit
   +0x04c SystemThread     : Pos 13, 1 Bit
   +0x04c Reserved         : Pos 14, 18 Bits
   +0x04c MiscFlags        : Int4B
   +0x050 ApcState         : _KAPC_STATE
   +0x050 ApcStateFill     : [43] UChar
   +0x07b Priority         : Char
   +0x07c NextProcessor    : Uint4B
   +0x080 DeferredProcessor : Uint4B
   +0x088 ApcQueueLock     : Uint8B
   +0x090 WaitStatus       : Int8B
   +0x098 WaitBlockList    : Ptr64 _KWAIT_BLOCK
   +0x0a0 WaitListEntry    : _LIST_ENTRY
   +0x0a0 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x0b0 Queue            : Ptr64 _KQUEUE
   +0x0b8 Teb              : Ptr64 Void
   +0x0c0 Timer            : _KTIMER
   +0x100 AutoAlignment    : Pos 0, 1 Bit
   +0x100 DisableBoost     : Pos 1, 1 Bit
   +0x100 EtwStackTraceApc1Inserted : Pos 2, 1 Bit
   +0x100 EtwStackTraceApc2Inserted : Pos 3, 1 Bit
   +0x100 CalloutActive    : Pos 4, 1 Bit
   +0x100 ApcQueueable     : Pos 5, 1 Bit
   +0x100 EnableStackSwap  : Pos 6, 1 Bit
   +0x100 GuiThread        : Pos 7, 1 Bit
   +0x100 UmsPerformingSyscall : Pos 8, 1 Bit
   +0x100 VdmSafe          : Pos 9, 1 Bit
   +0x100 UmsDispatched    : Pos 10, 1 Bit
   +0x100 ReservedFlags    : Pos 11, 21 Bits
   +0x100 ThreadFlags      : Int4B
   +0x104 Spare0           : Uint4B
   +0x108 WaitBlock        : [4] _KWAIT_BLOCK
   +0x108 WaitBlockFill4   : [44] UChar
   +0x134 ContextSwitches  : Uint4B
   +0x108 WaitBlockFill5   : [92] UChar
   +0x164 State            : UChar
   +0x165 NpxState         : Char
   +0x166 WaitIrql         : UChar
   +0x167 WaitMode         : Char
   +0x108 WaitBlockFill6   : [140] UChar
   +0x194 WaitTime         : Uint4B
```

```
    +0x108 WaitBlockFill7   : [168] UChar
    +0x1b0 TebMappedLowVa   : Ptr64 Void
    +0x1b8 Ucb              : Ptr64 _UMS_CONTROL_BLOCK
    +0x108 WaitBlockFill8   : [188] UChar
    +0x1c4 KernelApcDisable : Int2B
    +0x1c6 SpecialApcDisable : Int2B
    +0x1c4 CombinedApcDisable : Uint4B
    +0x1c8 QueueListEntry   : _LIST_ENTRY
    +0x1d8 TrapFrame        : Ptr64 _KTRAP_FRAME
    +0x1e0 FirstArgument    : Ptr64 Void
    +0x1e8 CallbackStack    : Ptr64 Void
    +0x1e8 CallbackDepth    : Uint8B
    +0x1f0 ApcStateIndex    : UChar
    +0x1f1 BasePriority     : Char
    +0x1f2 PriorityDecrement : Char
    +0x1f2 ForegroundBoost  : Pos 0, 4 Bits
    +0x1f2 UnusualBoost     : Pos 4, 4 Bits
    +0x1f3 Preempted        : UChar
    +0x1f4 AdjustReason     : UChar
    +0x1f5 AdjustIncrement  : Char
    +0x1f6 PreviousMode     : Char
    +0x1f7 Saturation       : Char
    +0x1f8 SystemCallNumber : Uint4B
    +0x1fc FreezeCount      : Uint4B
    +0x200 UserAffinity     : _GROUP_AFFINITY
    +0x210 Process          : Ptr64 _KPROCESS
    +0x218 Affinity         : _GROUP_AFFINITY
    +0x228 IdealProcessor   : Uint4B
    +0x22c UserIdealProcessor : Uint4B
    +0x230 ApcStatePointer  : [2] Ptr64 _KAPC_STATE
    +0x240 SavedApcState    : _KAPC_STATE
    +0x240 SavedApcStateFill : [43] UChar
    +0x26b WaitReason       : UChar
    +0x26c SuspendCount     : Char
    +0x26d Spare1           : Char
    +0x26e CodePatchInProgress : UChar
    +0x270 Win32Thread      : Ptr64 Void
    +0x278 StackBase        : Ptr64 Void
    +0x280 SuspendApc       : _KAPC
    +0x280 SuspendApcFill0  : [1] UChar
    +0x281 ResourceIndex    : UChar
    +0x280 SuspendApcFill1  : [3] UChar
    +0x283 QuantumReset     : UChar
    +0x280 SuspendApcFill2  : [4] UChar
    +0x284 KernelTime       : Uint4B
    +0x280 SuspendApcFill3  : [64] UChar
    +0x2c0 WaitPrcb         : Ptr64 _KPRCB
    +0x280 SuspendApcFill4  : [72] UChar
    +0x2c8 LegoData         : Ptr64 Void
```

```
    +0x280 SuspendApcFill5  : [83] UChar
    +0x2d3 LargeStack       : UChar
    +0x2d4 UserTime         : Uint4B
    +0x2d8 SuspendSemaphore : _KSEMAPHORE
    +0x2d8 SuspendSemaphorefill : [28] UChar
    +0x2f4 SListFaultCount  : Uint4B
    +0x2f8 ThreadListEntry  : _LIST_ENTRY
    +0x308 MutantListHead   : _LIST_ENTRY
    +0x318 SListFaultAddress : Ptr64 Void
    +0x320 ReadOperationCount : Int8B
    +0x328 WriteOperationCount : Int8B
    +0x330 OtherOperationCount : Int8B
    +0x338 ReadTransferCount : Int8B
    +0x340 WriteTransferCount : Int8B
    +0x348 OtherTransferCount : Int8B
    +0x350 ThreadCounters   : Ptr64 _KTHREAD_COUNTERS
    +0x358 StateSaveArea    : Ptr64 _XSAVE_FORMAT
    +0x360 XStateSave       : Ptr64 _XSTATE_SAVE
};
```

This is a longer function than KeInitializeDpc, so execute `uf KeInitializeApc` to get the full disassembly.

```
    fffff800`02671474 c60112          mov     byte ptr [rcx],12h
    fffff800`02671477 c6410258        mov     byte ptr [rcx+2],58h
    fffff800`0267147b 4183f802        cmp     r8d,2
    fffff800`0267147f 7439            je      nt!KeInitializeApc+0x46
(fffff800`026714ba)  Branch

    nt!KeInitializeApc+0xd:
    fffff800`02671481 44884150        mov     byte ptr [rcx+50h],r8b

    nt!KeInitializeApc+0x11:
    fffff800`02671485 488b442428      mov     rax,qword ptr [rsp+28h]
    fffff800`0267148a 48895108        mov     qword ptr [rcx+8],rdx
    fffff800`0267148e 33d2            xor     edx,edx
    fffff800`02671490 48894128        mov     qword ptr [rcx+28h],rax
    fffff800`02671494 488b442430      mov     rax,qword ptr [rsp+30h]
    fffff800`02671499 4c894920        mov     qword ptr [rcx+20h],r9
    fffff800`0267149d 48894130        mov     qword ptr [rcx+30h],rax
    fffff800`026714a1 483bc2          cmp     rax,rdx
    fffff800`026714a4 741f            je      nt!KeInitializeApc+0x51
(fffff800`026714c5)  Branch

    nt!KeInitializeApc+0x32:
    fffff800`026714a6 8a442438        mov     al,byte ptr [rsp+38h]
    fffff800`026714aa 884151          mov     byte ptr [rcx+51h],al
    fffff800`026714ad 488b442440      mov     rax,qword ptr [rsp+40h]
    fffff800`026714b2 48894138        mov     qword ptr [rcx+38h],rax
```

```
    nt!KeInitializeApc+0x42:
    fffff800`026714b6 885152          mov     byte ptr [rcx+52h],dl
    fffff800`026714b9 c3              ret

    nt!KeInitializeApc+0x46:
    fffff800`026714ba 8a82f0010000    mov     al,byte ptr [rdx+1F0h]
    fffff800`026714c0 884150          mov     byte ptr [rcx+50h],al
    fffff800`026714c3 ebc0            jmp     nt!KeInitializeApc+0x11
(fffff800`02671485)  Branch

    nt!KeInitializeApc+0x51:
    fffff800`026714c5 885151          mov     byte ptr [rcx+51h],dl
    fffff800`026714c8 48895138        mov     qword ptr [rcx+38h],rdx
    fffff800`026714cc ebe8            jmp     nt!KeInitializeApc+0x42
(fffff800`026714b6)  Branch
```

We have the function prototype, the definitions for _KAPC and _KTHREAD, and we know the calling convention is Microsoft x64 Calling Convention (RCX, RDX, R8, R9, stack_RtL). The decompiled routine is…

```
void KeInitializeApc(
  PKAPC              Apc,
  PKTHREAD           Thread,
  UCHAR              StateIndex,
  PKKERNEL_ROUTINE   KernelRoutine,
  PKRUNDOWN_ROUTINE RundownRoutine,
  PKNORMAL_ROUTINE   NormalRoutine,
  UCHAR              Mode,
  PVOID              Context)
{
    Apc->Type = 18;
    Apc->Size = sizeof(*Apc);

    Apc->ApcStateIndex = (StateIndex == 2)
    ? Thread->ApcStateIndex
    : StateIndex;

    Apc->Thread = Thread;
    Apc->RundownRoutine = RundownRoutine;
    Apc->KernelRoutine = KernelRoutine;
    Apc->NormalRoutine = NormalRoutine;

    if (NormalRoutine != NULL) {
        Apc->ApcMode = Mode;
        Apc->NormalContext = Context;
    } else {
        Apc->ApcMode = 0;
```

```
        Apc->NormalContext = NULL;
    }

    Apc->Inserted = 0;
}
```

It might be easy to look at the last two blocks of instructions in the assembly, observe the backward jump, and think there is some kind of loop. Loops can usually be entered by naturally following the instructions without taking any jumps. Also, notice how the two blocks exist after `ret` and they both unconditionally jump back to only a few instructions after the place where you originally jumped to them. For example, loc+0xb jumps to loc+0x46 then jumps to loc+0x11 which is almost immediately after loc+0xb.

I am not sure why the arguments that were pushed on to the stack begin at `[rsp+28h]` rather than at `[rsp+0h]`. If the arguments really do start at `[rsp+28h]`, then I must also wonder what the deal is with the 40 (0x28) bytes starting at the top of the stack because this space does not appear to be used for local variables.

ObFastDereferenceObject (and explain its calling convention):

This function is undocumented and doesn't seem to be in any header file. Thankfully, I found it here after a few searches.

```
void ObFastDereferenceObject (
  PEX_FAST_REF FastRef,
  PVOID        Object
);
```

A new structure is used in this function, so here is the definition.

```
struct _EX_FAST_REF {
    +0x000 Object           : Ptr64 Void
    +0x000 RefCnt           : Pos 0, 4 Bits
    +0x000 Value            : Uint8B
};
```

Running `u ObFastDereferenceObject` gives too little assembly, and `uf ObFastDereferenceObject` gives too much. This is a snippet of the `uf` output only of the blocks identified by `nt!ObFastDereferenceObject+0x`.

```
    fffff800`02685d80 0f0d09          prefetchw [rcx]
    fffff800`02685d83 488b01          mov     rax,qword ptr [rcx]
    fffff800`02685d86 4c8bc0          mov     r8,rax
    fffff800`02685d89 4c33c2          xor     r8,rdx
    fffff800`02685d8c 4983f80f        cmp     r8,0Fh
```

```
    fffff800`02685d90 730f            jae     nt!ObFastDereferenceObject+0x21
(fffff800`02685da1)  Branch

    nt!ObFastDereferenceObject+0x12:
    fffff800`02685d92 4c8bc9          mov     r9,rcx

    nt!ObFastDereferenceObject+0x15:
    fffff800`02685d95 4c8d4001        lea     r8,[rax+1]
    fffff800`02685d99 f04d0fb101      lock cmpxchg qword ptr [r9],r8
    fffff800`02685d9e 7509            jne     nt!ObFastDereferenceObject+0x29
(fffff800`02685da9)  Branch

    nt!ObFastDereferenceObject+0x20:
    fffff800`02685da0 c3              ret  Branch

    nt!ObFastDereferenceObject+0x21:
    fffff800`02685da1 488bca          mov     rcx,rdx
    fffff800`02685da4 e9873d0000      jmp     nt!ObfDereferenceObject
(fffff800`02689b30)  Branch

    nt!ObFastDereferenceObject+0x29:
    fffff800`02685da9 488bc8          mov     rcx,rax
    fffff800`02685dac 4833ca          xor     rcx,rdx
    fffff800`02685daf 4883f90f        cmp     rcx,0Fh
    fffff800`02685db3 72e0            jb      nt!ObFastDereferenceObject+0x15
(fffff800`02685d95)  Branch

    nt!ObFastDereferenceObject+0x35:
    fffff800`02685db5 488bca          mov     rcx,rdx
    fffff800`02685db8 e9733d0000      jmp     nt!ObfDereferenceObject
(fffff800`02689b30)  Branch
```

```
void ObFastDereferenceObject (
  PEX_FAST_REF FastRef,
  PVOID        Object)
{
    void *var1 = FastRef->Object, *var2 = FastRef->Object;
    if (var2 ^ Object >= 15)
        ObfDereferenceObject(Object); /* does not return */

    do {
        var2 = var1+1;
        /*
        // not sure what source code makes this
        if (`lock cmpxchg FastRef->Object, var2` == 0)
            return;
        */
        FastRef->Object = var1 ^ Object;
```

```
    } while (FastRef->Object < 15);

    ObfDereferenceObject(Object); /* does not return */
}
```

The calling convention may be fastcall, because RCX and RDX are used as source registers very early on. Though, it may also just be Microsoft x64 Calling Convention seeing as this function takes exactly two arguments and the first two arguments are passed through RCX and RDX anyway.

KeInitializeQueue:

The function prototype for `KeInitializeQueue` is given here.

Here is the definition of a KQUEUE structure

```
struct _KQUEUE {
    +0x000 Header           : _DISPATCHER_HEADER
    +0x018 EntryListHead    : _LIST_ENTRY
    +0x028 CurrentCount     : Uint4B
    +0x02c MaximumCount     : Uint4B
    +0x030 ThreadListHead   : _LIST_ENTRY
};
```

A KQUEUE contains a DISPATCHER_HEADER structure.

```
struct _DISPATCHER_HEADER {
    +0x000 Type             : UChar
    +0x001 TimerControlFlags : UChar
    +0x001 Absolute         : Pos 0, 1 Bit
    +0x001 Coalescable      : Pos 1, 1 Bit
    +0x001 KeepShifting     : Pos 2, 1 Bit
    +0x001 EncodedTolerableDelay : Pos 3, 5 Bits
    +0x001 Abandoned        : UChar
    +0x001 Signalling       : UChar
    +0x002 ThreadControlFlags : UChar
    +0x002 CpuThrottled     : Pos 0, 1 Bit
    +0x002 CycleProfiling   : Pos 1, 1 Bit
    +0x002 CounterProfiling : Pos 2, 1 Bit
    +0x002 Reserved         : Pos 3, 5 Bits
    +0x002 Hand             : UChar
    +0x002 Size             : UChar
    +0x003 TimerMiscFlags   : UChar
    +0x003 Index            : Pos 0, 6 Bits
    +0x003 Inserted         : Pos 6, 1 Bit
    +0x003 Expired          : Pos 7, 1 Bit
    +0x003 DebugActive      : UChar
```

```
    +0x003 ActiveDR7        : Pos 0, 1 Bit
    +0x003 Instrumented     : Pos 1, 1 Bit
    +0x003 Reserved2        : Pos 2, 4 Bits
    +0x003 UmsScheduled     : Pos 6, 1 Bit
    +0x003 UmsPrimary       : Pos 7, 1 Bit
    +0x003 DpcActive        : UChar
    +0x000 Lock             : Int4B
    +0x004 SignalState      : Int4B
    +0x008 WaitListHead     : _LIST_ENTRY
};
```

```
    fffff800`0266b298 c60104          mov     byte ptr [rcx],4
    fffff800`0266b29b c6410210        mov     byte ptr [rcx+2],10h
    fffff800`0266b29f 4533c0          xor     r8d,r8d
    fffff800`0266b2a2 44884101        mov     byte ptr [rcx+1],r8b
    fffff800`0266b2a6 44894104        mov     dword ptr [rcx+4],r8d
    fffff800`0266b2aa 488d4108        lea     rax,[rcx+8]
    fffff800`0266b2ae 48894008        mov     qword ptr [rax+8],rax
    fffff800`0266b2b2 488900          mov     qword ptr [rax],rax
    fffff800`0266b2b5 488d4118        lea     rax,[rcx+18h]
    fffff800`0266b2b9 48894008        mov     qword ptr [rax+8],rax
    fffff800`0266b2bd 488900          mov     qword ptr [rax],rax
    fffff800`0266b2c0 488d4130        lea     rax,[rcx+30h]
    fffff800`0266b2c4 48894008        mov     qword ptr [rax+8],rax
    fffff800`0266b2c8 488900          mov     qword ptr [rax],rax
    fffff800`0266b2cb 44894128        mov     dword ptr [rcx+28h],r8d
    fffff800`0266b2cf 413bd0          cmp     edx,r8d
    fffff800`0266b2d2 7404            je      nt!KeInitializeQueue+0x40
(fffff800`0266b2d8)  Branch

    nt!KeInitializeQueue+0x3c:
    fffff800`0266b2d4 89512c          mov     dword ptr [rcx+2Ch],edx
    fffff800`0266b2d7 c3              ret

    nt!KeInitializeQueue+0x40:
    fffff800`0266b2d8 8b053a7d2800    mov     eax,dword ptr [nt!KeNumberProcessors
(fffff800`028f3018)]
    fffff800`0266b2de 89412c          mov     dword ptr [rcx+2Ch],eax
    fffff800`0266b2e1 c3              ret
```

```
void KeInitializeQueue(
  PRKQUEUE Queue,
  ULONG    Count)
{
    Queue->Header.Type = 4;
    Queue->Header.ThreadControlFlags = 16;
    Queue->Header.TimerControlFlags = 0;
    Queue->Header.SignalState = 0;
```

```
    Queue->Header.WaitListHead.Blink = &Queue->Header.WaitListHead;
    Queue->Header.WaitListHead.Flink = &Queue->Header.WaitListHead;

    Queue->EntryListHead.Blink = &Queue->EntryListHead;
    Queue->EntryListHead.Flink = &Queue->EntryListHead;

    Queue->ThreadListHead.Blink = &Queue->ThreadListHead;
    Queue->ThreadListHead.Flink = &Queue->ThreadListHead;

    Queue->CurrentCount = 0;

    Queue->MaximumCount = (Count == 0)
    ? KeNumberProcessors
    : Count;
}
```

This function defaults to `KeNumberProcessors` if the `Count` argument is zero. You can inspect the `KeNumberProcessors` kernel variable yourself by executing ` dd nt!KeNumberProcessors L 1` in windbg.

KxWaitForLockChainValid:

This is another undocumented function that does not appear to be in any header file. Its prototype can be found here.

Here is the definition of a KSPIN_LOCK_QUEUE structure.

```
struct _KSPIN_LOCK_QUEUE {
    +0x000 Next             : Ptr64 _KSPIN_LOCK_QUEUE
    +0x008 Lock             : Ptr64 Uint8B
};
```

```
    fffff800`02699758 48895c2408      mov     qword ptr [rsp+8],rbx
    fffff800`0269975d 57              push    rdi
    fffff800`0269975e 4883ec20        sub     rsp,20h
    fffff800`02699762 488bf9          mov     rdi,rcx
    fffff800`02699765 33db            xor     ebx,ebx

    nt!KxWaitForLockChainValid+0xf:
    fffff800`02699767 ffc3            inc     ebx
    fffff800`02699769 851ddd9b2500    test    dword ptr [nt!HvlLongSpinCountMask
(fffff800`028f334c)],ebx
    fffff800`0269976f 0f844b870600    je      nt! ?? ::FNODOBFM::`string'+0x57c0
```

```
(fffff800`02701ec0)  Branch

    nt!KxWaitForLockChainValid+0x1d:
    fffff800`02699775 f390            pause

    nt!KxWaitForLockChainValid+0x1f:
    fffff800`02699777 488b07          mov     rax,qword ptr [rdi]
    fffff800`0269977a 4885c0          test    rax,rax
    fffff800`0269977d 74e8            je      nt!KxWaitForLockChainValid+0xf
(fffff800`02699767)  Branch

    nt!KxWaitForLockChainValid+0x27:
    fffff800`0269977f 488b5c2430      mov     rbx,qword ptr [rsp+30h]
    fffff800`02699784 4883c420        add     rsp,20h
    fffff800`02699788 5f              pop     rdi
    fffff800`02699789 c3              ret

    nt! ?? ::FNODOBFM::`string'+0x57c0:
    fffff800`02701ec0 f6057d111f0040  test    byte ptr [nt!HvlEnlightenments
(fffff800`028f3044)],40h
    fffff800`02701ec7 0f84a878f9ff    je      nt!KxWaitForLockChainValid+0x1d
(fffff800`02699775)  Branch

    nt! ?? ::FNODOBFM::`string'+0x57cd:
    fffff800`02701ecd 8bcb            mov     ecx,ebx
    fffff800`02701ecf e8dc170400      call    nt!HvlNotifyLongSpinWait
(fffff800`027436b0)
    fffff800`02701ed4 90              nop
    fffff800`02701ed5 e99d78f9ff      jmp     nt!KxWaitForLockChainValid+0x1f
(fffff800`02699777)  Branch
```

```
PKSPIN_LOCK_QUEUE KxWaitForLockChainValid(
  PKSPIN_LOCK_QUEUE LockQueue)
{
    int counter = 0;
    do {
        counter++;
        if (counter & HvlLongSpinCountMask != 0 && HvlEnlightenments & 0x40 != 0) {
            HvlNotifyLongSpinWait(counter);
        } else {
    //      `pause` /* don't know what source code makes this instruction */
        }
    } while (LockQueue->Next == NULL);

    return LockQueue->Next;
}
```

Identifying the compound `if` statement may be tricky, but nesting singular `if` statements mimics multiple `&&` conditions. Remember to keep the `else` condition on the outermost singular `if` statement.

KeReadyThread:

This is another undocumented function that is not in the header files. You can refer to the prototype here.

Here is the definition of a KAPC_STATE structure.

```
struct _KAPC_STATE {
   +0x000 ApcListHead         : [2] _LIST_ENTRY
   +0x020 Process             : Ptr64 _KPROCESS
   +0x028 KernelApcInProgress : UChar
   +0x029 KernelApcPending    : UChar
   +0x02a UserApcPending      : UChar
};
```

Since it exists in the KAPC_STATE structure, here is the definition of a KPROCESS structure.

```
struct _KPROCESS {
   +0x000 Header                    : _DISPATCHER_HEADER
   +0x018 ProfileListHead           : _LIST_ENTRY
   +0x028 DirectoryTableBase        : Uint8B
   +0x030 ThreadListHead            : _LIST_ENTRY
   +0x040 ProcessLock               : Uint8B
   +0x048 Affinity                  : _KAFFINITY_EX
   +0x070 ReadyListHead             : _LIST_ENTRY
   +0x080 SwapListEntry             : _SINGLE_LIST_ENTRY
   +0x088 ActiveProcessors          : _KAFFINITY_EX
   +0x0b0 AutoAlignment             : Pos 0, 1 Bit
   +0x0b0 DisableBoost              : Pos 1, 1 Bit
   +0x0b0 DisableQuantum            : Pos 2, 1 Bit
   +0x0b0 ActiveGroupsMask          : Pos 3, 4 Bits
   +0x0b0 ReservedFlags             : Pos 7, 25 Bits
   +0x0b0 ProcessFlags              : Int4B
   +0x0b4 BasePriority              : Char
   +0x0b5 QuantumReset              : Char
   +0x0b6 Visited                   : UChar
   +0x0b7 Unused3                   : UChar
   +0x0b8 ThreadSeed                : [4] Uint4B
   +0x0c8 IdealNode                 : [4] Uint2B
   +0x0d0 IdealGlobalNode           : Uint2B
   +0x0d2 Flags                     : _KEXECUTE_OPTIONS
   +0x0d3 Unused1                   : UChar
   +0x0d4 Unused2                   : Uint4B
   +0x0d8 Unused4                   : Uint4B
```

```
   +0x0dc StackCount              : _KSTACK_COUNT
   +0x0e0 ProcessListEntry        : _LIST_ENTRY
   +0x0f0 CycleTime               : Uint8B
   +0x0f8 KernelTime              : Uint4B
   +0x0fc UserTime                : Uint4B
   +0x100 InstrumentationCallback : Ptr64 Void
   +0x108 LdtFreeSelectorHint     : Uint2B
   +0x10a LdtTableLength          : Uint2B
   +0x108 Spare2                  : [4] UChar
   +0x110 LdtSystemDescriptor     : _KGDTENTRY64
   +0x120 LdtBaseAddress          : Ptr64 Void
   +0x128 LdtProcessLock          : _KGUARDED_MUTEX
   +0x110 UserDirectoryTableBase  : Uint8B
   +0x118 AddressPolicy           : UChar
   +0x119 Spare3                  : [71] UChar
};
```

```
    fffff800`026aaf54 4053            push    rbx
    fffff800`026aaf56 4883ec20        sub     rsp,20h
    fffff800`026aaf5a 488b5170        mov     rdx,qword ptr [rcx+70h]
    fffff800`026aaf5e 488bd9          mov     rbx,rcx
    fffff800`026aaf61 8b82dc000000    mov     eax,dword ptr [rdx+0DCh]
    fffff800`026aaf67 a807            test    al,7
    fffff800`026aaf69 0f85f1790600    jne     nt! ?? ::FNODOBFM::`string'+0x16260
(fffff800`02712960)  Branch

    nt!KeReadyThread+0x1b:
    fffff800`026aaf6f 488bcb          mov     rcx,rbx
    fffff800`026aaf72 e88dfeffff      call    nt!KiFastReadyThread
(fffff800`026aae04)

    nt!KeReadyThread+0x23:
    fffff800`026aaf77 4883c420        add     rsp,20h
    fffff800`026aaf7b 5b              pop     rbx
    fffff800`026aaf7c c3              ret

    nt! ?? ::FNODOBFM::`string'+0x16260:
    fffff800`02712960 e8c7d1faff      call    nt!KiInSwapSingleProcess
(fffff800`026bfb2c)
    fffff800`02712965 84c0            test    al,al
    fffff800`02712967 0f850a86f9ff    jne     nt!KeReadyThread+0x23
(fffff800`026aaf77)  Branch

    nt! ?? ::FNODOBFM::`string'+0x1626d:
    fffff800`0271296d e9fd85f9ff      jmp     nt!KeReadyThread+0x1b
(fffff800`026aaf6f)  Branch
```

```
void KeReadyThread(
```

```
  PKTHREAD Thread)
{
    if (Thread->ApcState.Process->StackCount & 7 == 0
    || KiInSwapSingleProcess() == 0) {
        KiFastReadyThread(Thread);
    }
}
```

KiInitializeTSS:

```
    Couldn't resolve error at 'nt!kiinitializetss'
```

RtlValidateUnicodeString:

Undocumented and not in the header files, but I found a function prototype here.

This function takes a PCUNICODE_STRING as one of its arguments, so here is the definition of the _UNICODE_STRING structure.

```
struct _UNICODE_STRING {
    +0x000 Length           : Uint2B
    +0x002 MaximumLength    : Uint2B
    +0x008 Buffer           : Ptr64 Uint2B
};
```

```
nt!RtlValidateUnicodeString:
fffff800`026c1418 4883ec28        sub     rsp,28h
fffff800`026c141c 488bc2          mov     rax,rdx
fffff800`026c141f 85c9            test    ecx,ecx
fffff800`026c1421 0f8509840300    jne     nt! ?? ::FNODOBFM::`string'+0x3130
(fffff800`026f9830)  Branch

nt!RtlValidateUnicodeString+0xf:
fffff800`026c1427 baff7f0000      mov     edx,7FFFh
fffff800`026c142c 41b800010000    mov     r8d,100h
fffff800`026c1432 488bc8          mov     rcx,rax
fffff800`026c1435 e80e000000      call    nt!RtlValidateUnicodeString+0x30
(fffff800`026c1448)

nt!RtlValidateUnicodeString+0x22:
fffff800`026c143a 4883c428        add     rsp,28h
fffff800`026c143e c3              ret

nt!RtlValidateUnicodeString+0x30:
fffff800`026c1448 33c0            xor     eax,eax
fffff800`026c144a 483bc8          cmp     rcx,rax
```

```
fffff800`026c144d 7424            je      nt!RtlValidateUnicodeString+0x5b
(fffff800`026c1473)  Branch

nt!RtlValidateUnicodeString+0x37:
fffff800`026c144f f60101          test    byte ptr [rcx],1
fffff800`026c1452 7521            jne     nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch

nt!RtlValidateUnicodeString+0x3c:
fffff800`026c1454 0fb75102        movzx   edx,word ptr [rcx+2]
fffff800`026c1458 f6c201          test    dl,1
fffff800`026c145b 7518            jne     nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch

nt!RtlValidateUnicodeString+0x45:
fffff800`026c145d 663911          cmp     word ptr [rcx],dx
fffff800`026c1460 7713            ja      nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch

nt!RtlValidateUnicodeString+0x4a:
fffff800`026c1462 6681fafeff      cmp     dx,0FFFEh
fffff800`026c1467 770c            ja      nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch

nt!RtlValidateUnicodeString+0x51:
fffff800`026c1469 48394108        cmp     qword ptr [rcx+8],rax
fffff800`026c146d 0f843d750300    je      nt! ?? ::FNODOBFM::`string'+0x22b0
(fffff800`026f89b0)  Branch

nt!RtlValidateUnicodeString+0x5b:
fffff800`026c1473 f3c3            rep ret  Branch

nt!RtlValidateUnicodeString+0x5d:
fffff800`026c1475 b80d0000c0      mov     eax,0C000000Dh
fffff800`026c147a c3              ret

nt! ?? ::FNODOBFM::`string'+0x22b0:
fffff800`026f89b0 663901          cmp     word ptr [rcx],ax
fffff800`026f89b3 0f85bc8afcff    jne     nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch

nt! ?? ::FNODOBFM::`string'+0x22b9:
fffff800`026f89b9 663bd0          cmp     dx,ax
fffff800`026f89bc 0f84b18afcff    je      nt!RtlValidateUnicodeString+0x5b
(fffff800`026c1473)  Branch

nt! ?? ::FNODOBFM::`string'+0x22c2:
fffff800`026f89c2 e9ae8afcff      jmp     nt!RtlValidateUnicodeString+0x5d
(fffff800`026c1475)  Branch
```

```
nt! ?? ::FNODOBFM::`string'+0x3130:
fffff800`026f9830 b80d0000c0      mov     eax,0C000000Dh
fffff800`026f9835 e9007cfcff      jmp     nt!RtlValidateUnicodeString+0x22
(fffff800`026c143a)  Branch
```

```
NTSTATUS RtlValidateUnicodeString(
  ULONG Flags,
  PCUNICODE_STRING UnicodeString)
{
    if (Flags != 0)
        return 0x0C000000D;

    if (UnicodeString == NULL)
        return 0;

    if (UnicodeString->Length & 1 != 0)
        return 0x0C000000D;

    if (UnicodeString->MaximumLength & 1 != 0)
        return 0x0C000000D;

    if (UnicodeString->Length > UnicodeString->MaximumLength)
        return 0x0C000000D;

    if (UnicodeString->MaximumLength > 0xFFFE)
        return 0x0C000000D;

    if (UnicodeString->Buffer == NULL) {
        if (UnicodeString->Length != 0)
            return 0x0C000000D;

        if (UnicodeString->MaximumLength != 0)
            return 0x0C000000D;
    }

    return 0;
}
```

It is pretty easy to produce equivalent C that does not use a nested `if` block at the end, but this is fine.

---

6. Sample H. The function sub_13846 references several structures whose types are not entirely clear. Your task is to first recover the function prototype and then try to reconstruct the structure fields. After reading Chapter 3, return to this exercise to see if your understanding has changed. (Note: This sample is targeting Windows XP x86.)

```
00013842 8b 41 60     mov    eax,dword ptr [ecx + 0x60]
00013845 56           push   esi
00013846 8b 72 08     mov    esi,dword ptr [edx + 0x8]
00013849 fe 49 23     dec    byte ptr [ecx + 0x23]
0001384c 83 e8 24     sub    eax,0x24
0001384f 89 41 60     mov    dword ptr [ecx + 0x60],eax
00013852 89 50 14     mov    dword ptr [eax + 0x14],edx
00013855 0f b6 00     movzx  eax,byte ptr [eax]
00013858 51           push   ecx
00013859 52           push   edx
0001385a ff 54 86 38  call   dword ptr [esi + eax*0x4 + 0x38]
0001385e 5e           pop    esi
0001385f c3           ret
```

This function most likely uses the fastcall calling convention because ECX and EDX are being read from at the beginning.

```
void* sub_13842(void *arg1, void *arg2)
{
    var1 = arg1->fld_60;
    arg1->fld_32--;
    var1 -= 36;
    arg1->fld_60 = var1;
    var1->fld_14 = arg2;
    return arg2->fld_8->fld_38[var1->fld_0](arg1, arg2);
}
```

It isn't clear if the function being called takes its arguments through the stack or through ECX and EDX. I assumed the function still uses fastcall, but if the arguments are passed on the stack, then the function's parameters will be reversed.

---

7. Sample H. The function sub_10BB6 has a loop searching for something. First recover the function prototype and then infer the types based on the context. Hint: You should probably have a copy of the PE specification nearby.

```
00010bb2 8b 44 24 04        mov    eax,dword ptr [esp + param_1]
00010bb6 53                 push   ebx
00010bb7 56                 push   esi
00010bb8 8b 70 3c           mov    esi,dword ptr [eax + 0x3c]
00010bbb 03 f0              add    esi,eax
00010bbd 0f b7 46 14        movzx  eax,word ptr [esi + 0x14]
00010bc1 33 db              xor    ebx,ebx
00010bc3 66 39 5e 06        cmp    word ptr [esi + 0x6],bx
00010bc7 57                 push   edi
00010bc8 8d 7c 30 18        lea    edi,[eax + esi*0x1 + 0x18]
00010bcc 76 1d              jbe    LAB_00010beb
```

```
     LAB_00010bce
00010bce ff 74 24 14        push   dword ptr [esp + param_2]
00010bd2 57                 push   edi
00010bd3 ff 15 a4 69 01 00  call   dword ptr [LAB_000169a4]
00010bd9 85 c0              test   eax,eax
00010bdb 59                 pop    ecx
00010bdc 59                 pop    ecx
00010bdd 74 14              jz     LAB_00010bf3
00010bdf 0f b7 46 06        movzx  eax,word ptr [esi + 0x6]
00010be3 83 c7 28           add    edi,0x28
00010be6 43                 inc    ebx
00010be7 3b d8              cmp    ebx,eax
00010be9 72 e3              jc     LAB_00010bce
     LAB_00010beb
00010beb 33 c0              xor    eax,eax
     LAB_00010bed
00010bed 5f                 pop    edi
00010bee 5e                 pop    esi
00010bef 5b                 pop    ebx
00010bf0 c2 08 00           ret    0x8
     LAB_00010bf3
00010bf3 8b c7              mov    eax,edi
00010bf5 eb f6              jmp    LAB_00010bed
```

Much of this function becomes obvious after you define PE data structures in your favorite reverse engineering tool and look at a few places where this function is called. Some callers pass a hard-coded string as the second argument which helps in recovering the prototype.

```
/* Get section by name (Sample_H!sub_10BB6)
 * Returns a pointer to the section if that name exists, else NULL
 */
PIMAGE_SECTION_HEADER get_section_by_name(PIMAGE_DOS_HEADER pDos, char
*section_name)
{
    PIMAGE_NT_HEADER pNt = pDos->e_lfanew;
    PIMAGE_SECTION_HEADER pSect = (PIMAGE_SECTION_HEADER)((char*)pNt + pNt-
>FileHeader.SizeOfOptionalHeader);
    DWORD index = 0;

    if (pNt->FileHeader.NumberOfSections == 0)
        return NULL;

    do {
        if (section_name_compare(pSect, section_name) == 0)
            return pSect;
        ++index;
        ++pSect;
    } while (index < pNt->FileHeader.NumberOfSections);
```

```
    return NULL;
}
```

---

8. Sample H. Decompile sub_11732 and explain the most likely programming construct used in the original code.

```
0001172e 56           push  esi
0001172f 8b 74 24 08  mov   esi,dword ptr [esp + param_1]
00011733 4e           dec   esi
00011734 74 29        jz    LAB_0001175f
00011736 4e           dec   esi
00011737 74 1c        jz    LAB_00011755
00011739 4e           dec   esi
0001173a 74 0f        jz    LAB_0001174b
0001173c 83 ee 09     sub   esi,0x9
0001173f 75 2a        jnz   LAB_0001176b
00011741 8b 70 08     mov   esi,dword ptr [eax + 0x8]
00011744 d1 ee        shr   esi,1
00011746 83 c0 0c     add   eax,0xc
00011749 eb 1c        jmp   LAB_00011767
     LAB_0001174b
0001174b 8b 70 3c     mov   esi,dword ptr [eax + 0x3c]
0001174e d1 ee        shr   esi,1
00011750 83 c0 5e     add   eax,0x5e
00011753 eb 12        jmp   LAB_00011767
     LAB_00011755
00011755 8b 70 3c     mov   esi,dword ptr [eax + 0x3c]
00011758 d1 ee        shr   esi,1
0001175a 83 c0 44     add   eax,0x44
0001175d eb 08        jmp   LAB_00011767
     LAB_0001175f
0001175f 8b 70 3c     mov   esi,dword ptr [eax + 0x3c]
00011762 d1 ee        shr   esi,1
00011764 83 c0 40     add   eax,0x40
     LAB_00011767
00011767 89 31        mov   dword ptr [ecx],esi
00011769 89 02        mov   dword ptr [edx],eax
     LAB_0001176b
0001176b 5e           pop   esi
0001176c c2 04 00     ret   0x4
```

This function reads from EAX and the stack, and writes to ECX and EDX. The calling convention is unusual and makes it hard to tell this function's prototype. It at least appears to take four arguments which I'll say go in this order: ECX, EDX, EAX, [ESP+?].

```
void sub_11732(void *arg1, void *arg2, void *arg3, int arg4)
{
```

```
    DWORD var1;
    switch (arg4) {
        case 1:
            var1 = arg3->fld_3c;
            var1 /= 2;
            arg3 += 64;
            break;
        case 2:
            var1 = arg3->fld_3c;
            var1 /= 2;
            arg3 += 68;
            break;
        case 3:
            var1 = arg3->fld_3c;
            var1 /= 2;
            arg3 += 94;
            break;
        case 12:
            var1 = arg3->fld_8;
            var1 /= 2;
            arg3 += 12;
            break;
        default:
            return;
    }
    *arg1 = var1;
    *arg2 = arg3;
}
```

The most likely programming construct used in the original function is a `switch` block. The stack argument, which I've called `arg4` is the one being `switch`ed.

---

9. Sample L. Explain what function sub_1000CEA0 does and then decompile it back to C.

```
1000cea0 55                                push            ebp
1000cea1 8b ec                             mov             ebp,esp
1000cea3 57                                push            edi
1000cea4 8b 7d 08                          mov             edi,dword ptr [ebp + _Str]
1000cea7 33 c0                             xor             eax,eax
1000cea9 83 c9 ff                          or              ecx,0xffffffff
1000ceac f2 ae                             repne scasb     es:edi
1000ceae 83 c1 01                          add             ecx,0x1
1000ceb1 f7 d9                             neg             ecx
1000ceb3 83 ef 01                          sub             edi,0x1
1000ceb6 8a 45 0c                          mov             al,byte ptr [ebp + _Ch]
1000ceb9 fd                                std
1000ceba f2 ae                             repne scasb     es:edi
```

```
1000cebc 83 c7 01                    add             edi,0x1
1000cebf 38 07                       cmp             byte ptr [edi],al
1000cec1 74 04                       jz              LAB_1000cec7
1000cec3 33 c0                       xor             eax,eax
1000cec5 eb 02                       jmp             LAB_1000cec9
     LAB_1000cec7
1000cec7 8b c7                       mov             eax,edi
     LAB_1000cec9
1000cec9 fc                          cld
1000ceca 5f                          pop             edi
1000cecb c9                          leave
1000cecc c3                          ret
```

sub_1000CEA0 returns a pointer to the last occurrence of a character in a string, or NULL if the character does not exist. In other words, it's `strrchr`. Here is the C source code:

```
/* Sample_L!strrchr (Sample_L!sub_1000CEA0)
 */
char* _strrchr(char *str, char letter)
{
    size_t i = 0, end = 0;
    while (str[end] != 0) ++end;
    --end; // `sub edi,0x1` step back by 1 off null terminator
    for (; i < end && str[end-i] != letter; ++i);
    return (str[end-i] == letter) ? &str[end-i] : NULL;
}
```

---

10. If the current privilege level is encoded in CS, which is modifiable by user-mode code, why can’t user-mode code modify CS to change CPL?

The CPU architecture was never designed to allow user-mode code to write to the CS register. Here is a sample of what happens if you try to write to the CS register.

```
struct segment_register {
    union {
        uint16_t value;
        struct {
            uint16_t rpl   : 2;
            uint16_t table : 1;
            uint16_t index : 13;
        };
    };
};

int main()
{
    struct segment_register seg_reg = {0};
```

```
    __asm(
    "mov %0, cs\n"
    : "=X"(seg_reg));

    printf("cs => 0x%04x : index = 0x%x | table = %d | RPL = %d\n",
        seg_reg.value, seg_reg.index, seg_reg.table, seg_reg.rpl);

    seg_reg.rpl = 0;

    puts("Trying to set CS RPL to 0...");

    __asm(
    "mov cs, %0\n"
    :: "X"(seg_reg));

    puts("Success!");

    return 0;
}
```

Then here is the result of running this code.

```
cs => 0x0033 : index = 0x6 | table = 0 | RPL = 3
Trying to set CS RPL to 0...
Illegal instruction (core dumped)
```

11. Read the Virtual Memory chapter in Intel Software Developer Manual, Volume 3 and AMD64 Architecture Programmer's Manual, Volume 2: System Programming. Perform a few virtual address to physical address translations yourself and verify the result with a kernel debugger. Explain how data execution prevention (DEP) works.

I don't have a 32-bit system to do a virtual address to physical address conversion on, but there is a helpful resource here showing how to do it in windbg.

Data Execution Prevention only exists in 64-bit systems, and it works by setting the most significant bit (bit 63) of a Page Table Entry. When this bit is set, any attempt to execute instructions on the associated page of memory will result in a memory access violation.

12. Bruce's favorite x86/x64 disassembly library is BeaEngine by BeatriX (www.beaengine.org). Experiment with it by writing a program to disassemble a binary at its entry point.

I wrote a straight-forward disassembler called "disasm_artist" that takes a path to a binary and a file offset of where to begin disassembling. The source code is provided below.

```
#include <stdio.h>
#include <stdlib.h>
#include "BeaEngine.h"

```

```c
static const int RET = 0xC3;

/* Print message to stderr and exit with the passed error code
 */
void fatal(const char *str, int err_code)
{
    fprintf(stderr, str);
    printf("\n");
    exit(err_code);
}

/* Read file content into allocated buffer
 */
void* load_file(const char *str, size_t offset, size_t *fsize)
{
    FILE *fp = fopen(str, "rb");
    if (fp == NULL)
        fatal("Unable to open the file", -1);

    // get file size
    fseek(fp, 0, SEEK_END);
    *fsize = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    if (offset >= *fsize)
        fatal("Offset is greater than file size", -1);

    // allocate memory
    void *bytes = malloc(*fsize);
    if (bytes == NULL)
        fatal("Memory allocation failed", -1);

    // read file content
    if (*fsize != fread(bytes, 1, *fsize, fp))
        fatal("Failed to read file", -1);

    return bytes;
}

/* Disassemble at `offset` until EOF or a `ret` instruction
 */
void disassemble(void *bytes, size_t offset, size_t fsize)
{
    DISASM infos = {.EIP=(uint64_t)(bytes + offset)};

    while (infos.Error == 0 && offset < fsize) {
        int len = Disasm(&infos);
        offset += len;
        if (infos.Error != UNKNOWN_OPCODE) {
```

```
            puts(infos.CompleteInstr);
            infos.EIP += len;
            if (infos.Instruction.Opcode == RET)
                break;
        }
    }
}

int main(int argc, char *argv[])
{
    if (argc < 3) {
        printf("Usage: %s <FILE> <OFFSET>\n", argv[0]);
        return 0;
    }

    size_t offset = 0;
    if (sscanf(argv[2], "%lu", &offset) != 1)
        fatal("Offset must be a number", -1);

    size_t fsize = 0;
    void *bytes = load_file(argv[1], offset, &fsize);
    disassemble(bytes, offset, fsize);
    free(bytes);
    return 0;
}
```

Set 4 (Page 38)

1. Explain two methods to get the instruction pointer on x64. At least one of the methods must use RIP addressing.

This first method is recycled from set 2 question 1. It uses a relative call instruction offset zero from the current value of the instruction pointer. The result is it simply pushes the address of `pop rax` onto the stack, and the instruction pointer gets popped into `rax`.

```
__attribute__((always_inline))
inline uint64_t read_rip()
{
    uint64_t rip;
    __asm( // 0xE8 == `call`
    ".byte 0xE8,0x00,0x00,0x00,0x00\n"
    "pop rax\n"
    : "=X"(rip));
    return rip;
}
```

The second method uses RIP addressing, and all it does is `lea` the instruction pointer into a variable.

```
__attribute__((always_inline))
inline uint64_t read_rip()
{
    uint64_t rip;
    __asm(
    "lea %0,[rip]\n"
    : "=X"(rip));
    return rip;
}
```

2. Perform a virtual-to-physical address translation on x64. Were there any major differences compared to x86?

Based on the resource from earlier, the method is nearly the same. The only difference I encountered is the first parameter to `!vtop` must be the exact DirBase value, as you no longer need to do a bitwise right shift by 12.

```
0: kd> !process b9c 0
Searching for Process with Cid == b9c
PROCESS fffffa8005505b00
    SessionId: 1  Cid: 0b9c    Peb: 7efdf000  ParentCid: 0818
    DirBase: 48545000  ObjectTable: fffff8a00b143b00  HandleCount:  11.
    Image: virt_to_phys.exe

0: kd> !vtop 48545000 28ff06
Amd64VtoP: Virt 00000000`0028ff06, pagedir 48545000
Amd64VtoP: PML4E 48545000
Amd64VtoP: PDPE 479df000
Amd64VtoP: PDE 49463008
Amd64VtoP: PTE 4822f478
Amd64VtoP: Mapped phys 484d2f06
Virtual address 28ff06 translates to physical address 484d2f06.

0: kd> dc 28ff06
00000000`0028ff06  646e6966 20796d20 73796870 6c616369  find my physical
00000000`0028ff16  64646120 73736572 ff280021 9e340028   address!.(.(.4.

0: kd> !dc 484d2f06
#484d2f04 69661930 6d20646e 68702079 63697379 0.find my physic
#484d2f14 61206c61 65726464 00217373 0028ff28 al address!.(.(.
```