### **PHẦN GIỚI THIỆU:**

Những kiến thức mà chúng ta phải chuẩn bị trước khi đọc quyển sách này và những quyển sách về kiến thức đó:

**1.Học lập trình**( đặc biệt là C):*The C Programming Language*, by Brian Kernighan and  Dennis Ritchie (Prentice Hall, 1988) and *C: A Reference Manual*, by Samuel Harbison (Prentice Hall, 2002)

        +Sau khi đã biết viết, compiling, debugging một chương trình cơ bản thì có thể đọc quyển: *Expert C Programming:* *Deep C Secrets*, by Peter van der Linden (Prentice Hall, 1994)

        Để thực sự hiểu cách một chương trình thực sự được kết hợp với nhau:*Compilers: Principles,Techniques, and Tools*, by Alfred Aho, Ravi Sethi, and Jeffrey Ullman, (Prentice Hall, 1994) and*Linkers and Loaders*, by John Levine (Morgan Kaufmann, 1999)  
    *(Mục tiêu của đọc những quyển sách trên là làm quen với những khái niệm cơ bản, không cần hiểu tất cả mọi thứ bây giờ, sẽ có thời gian cho việc đó sau này)*

Hơn nữa có thể cân nhắc đọc quyển *Advanced Compiler Design and Implementation*, by Steven Muchnick (Morgan Kaufmann, 1997)

**2.Môi trường**

        Khi đã hiểu cách một chương trình được viết, thực thi, gỡ lỗi như thế nào, chúng ta nên bắt đầu khám phá môi trường thực thi của chương trình:  
          +Đầu tiên ta nên học về Bộ xử lý Intel: *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture* by Intel( Tập trung vào Chương 2-7, những chương này sẽ giải thích những yếu tố cơ bản của một máy tính hiện đại)

          +Những ai hứng thú với ARM : *Cortex-A Series Programmer’s Guide* and *ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition* by ARM  
       (Sau khi đọc lướt (skimming) qua những quyển sách trên bạn sẽ có một đánh giá cơ bản về các khối xây dựng kĩ thuật của một hệ thống máy tính)

        Hoặc hơn nữa có thể cân nhắc đọc  
            + *Structured Computer Organization* by Andrew Tanenbaum (Prentice Hall, 1998)  
            +*Microsoft PE and COFF Specification*

(Tại thời điểm này các bạn đã có đủ kiến thức để đọc Chương 1: x86 and x64  và Chương 2:ARM)

**3.Hệ điều hành.**

Có rất nhiều hệ điều hành khác nhau,nhưng chúng có rất nhiều điểm chung bao gồm quy trình (processes), luồng(threads), bộ nhớ ảo (vitual memory), phân tách quyền (privilege separation), đa tác vụ( multi-tasking), ..... Cách tốt nhất để hiểu về những thứ đó là đọc: *Modern Operating Systems*, by Andrew Tanenbaum (Prentice Hall, 2005)

Về window : *Windows NT Device Driver Development*, by Peter Viscarola and Anthony Mason (New Riders Press, 1998)

*What Makes It Page? The Windows 7 (x64) Virtual Memory Manager*, by Enrico Martignetti (CreateSpace Independent Publishing Platform, 2012).

        (Sau đó bạn đã đủ những kiến thức cần thiết để đọc Chương 3: The Windows Kernel)

        Bạn cũng nên cân nhắc học lập trình Win32: *Windows System Programming*, by Johnson Hart (Addison-Wesley Professional, 2010), and *Windows via C/C++*, by Jeffrey Richter and Christophe Nasarre (Microsoft Press, 2007)

       Chương 4, “Debugging and Automation,” Đọc*Inside Windows Debugging: A Practical Guide to Debugging and Tracing Strategies in Windows*, by Tarik Soulami (Microsoft Press, 2012), and *Advanced Windows Debugging*, by Mario Hewardt and Daniel Pravat (Addison-Wesley Professional, 2007).

    Chapter 5, “Obfuscation,” yêu cầu một sự hiểu biết rõ ràng về assembly và có thể đọc sau Chương 1. Đối với kiến thức nền có thể đọc  *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, by Christian Collberg and Jasvir Nagra (Addison-Wesley Professional, 2009).

**Chú Ý:**

Có thể tìm những mẫu malware ở :  [http://kernelmode.info](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

### **CHƯƠNG 1: x86 AND x64**

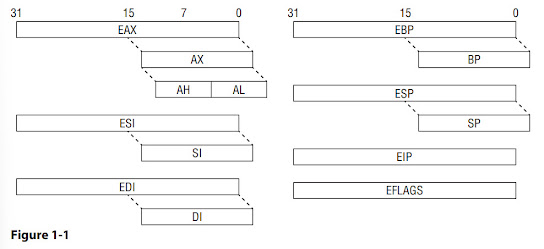
      x86 là kiến trúc little-endian dựa trên bộ xử lý Intel 8086. x86 là 32-bit của kiến trúc Intel(IA-32) được định nghĩa trong Intel Software Development Manual. Nó có thể hoạt động trong 2 chế độ: real và được bảo vệ(real and protected). Chế độ đọc(Real mode) là trạng thái bộ xử lý khi nó được chạy lần đầu tiên và chỉ hỗ trợ tập lệnh 16-bit. Chế độ được bảo vệ ( Protected mode) là trạng thái bộ xử lý hỗ trợ bộ nhớ ảo, phân trang bộ nhớ(paging), và các tính năng khác; đó là trạng tháo của hệ điều hành hiện đại. Mở rộng 64 bit của kiến trúc được gọi là x64 hoặc x86-x64.Chương này thảo luận về kiến trúc x86 hoạt động ở chế độ bảo vệ.

        x86 hỗ trợ khái niệm tách đặc quyền (privilege separation) được gọi là *ring level.*Bộ xử lý cũng cấp 4 cấp độ, được đánh số từ 0 đến 3.( Vòng 1 và 2 không thường được dùng nên không thảo luận ở đây). Vòng 0 là mức đặc quyền cao nhất và có thể sửa đổi tất cả các cài đặt hệ thống. Vòng 3 là mức đặc quyền thấp nhất và chỉ có để đọc/sửa đổi một tập hợp các cài đặt của hệ thống. Do đó, hệ điều hành hiện đại thường tách biệt đặc quyền của người dùng. kernel ( user/kernel) bởi có chế độ người dùng applications chạy trong vòng 3, và Kernel chạy ở vòng 0. Ring Level được mã hóa trong thanh ghi *cs*và đôi khi được gọi *current privilege level (CPL)*trong tài liệu chính thức.

        Chương này bàn luận về  kiến trúc x86/IA-32 như được định nghĩa trong Intel 64 and IA-32 Architectures Software Developer’s Manual, Volumes 1–3 ([www.intel.com/content/www/us/en/processors/architectures-software-developermanuals.html](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)).

#### THANH GHI VÀ KIỂU DỮ LIỆU

       Khi hệ điều hành trong chế độ được bảo vệ( protected mode), kiến trúc x86 có 8 thanh ghi 32- bit mục đích chung (eight 32-bit generalpurpose registers)(GPRs): EAX, EBX, ECX, EDX, ESI, EBP, and ESP. Một trong số những thanh ghi có thể chia thành những thanh ghi 8-bit và 16-bit. Con trỏ lệnh( instruction pointer) được lưu trong  thanh ghi. EIP. Những thanh ghi được minh họa trong hình 1-1.

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Một số GPRs và cách chúng được sử dụng:

**ECX**: Dùng để đếm trong vòng lặp ( COUNT)

**ESI**: Nguồn trong các thao tác chuỗi/bộ nhớ (Source in string/memory operations)

**EDI**: Đích trong các thao tác  chỗi/bộ nhớ (Destination in string/memory operations)

**EBP**: Con trỏ cơ sở (Base frame pointer)

**ESP**: Con trỏ ngăn xếp( Stack pointer)

        Những kiểu dữ liệu chung như sau:

        Bytes—8 bits. Examples: AL, BL, CL  
        Word—16 bits. Examples: AX, BX, CX  
        Double word—32 bits. Examples: EAX, EBX, ECX  
        Quad word—64 bits.  Khi x86 không có 64-bit GPRs, nó có thể kết hợp 2 thanh ghi,thường thì EDX:EAX, và coi chúng như là một giá trị 64-bit trong 1 số trường hợp, ví dụ lệnh  RDTSC viết giá trị 64-bit là EDX:EAX.

        Thanh ghi 32-bit EFLAGS được sử dụng để lưu trữ trạng thái của phép toán và những trạng thái thực thi khác. VÍ dụ , nếu là phép toán ADD có kết quả là zero, ZF flag sẽ được đặt là 1. Các cờ trong EFLAGS chủ yếu được dùng để phân nhánh câu điều kiện.

        Ngoài GPRs,EIP, and EFLAGS, cũng có những thanh khi điều khiển cơ chế hệ thống cấp thấp như là bộ nhớ ảo, ngắt( interrupts), gỡ lỗi(debugging). Ví dụ, CR0 điều khiển cho dù paging bật hay tắt, CR2 bao gồm các địa chỉ tuyến tính điều đó gây ra lỗi page ( page fault), CR3 là địa chỉ cơ sở của cấu trúc dữ liệu paging, và CR4 điều khiển ảo hóa phần cứng, DR0–DR7 được sử dụng để đặt memory breakpoints. Chính ta sẽ trở lại những thanh ghi này trong phần "System Mechanism".

**CHÚ Ý:**Mặc dù có 7 thanh ghi gỡ lỗi( debug registers), hệ thống chỉ cho phép 4 memory breakpoints(DR0-DR3). Những cái còn lại được dùng cho trạng thái( status)

     Cũng có những thanh ghi mô hình cụ thể ( model-specific registers (MSRs)). Như cái tên của nó, những thanh ghi này có thể khác nhau giữa các bộ xử lý bởi Intel và AMD. Mỗi MSR được định nghĩa mởi tên và 1 số 32-bit, và được đọc/viết thông qua RDMSR/WRMSR. Chúng chỉ có thể truy cập vào code đang được chạy trong ring 0 và thường được sử dụng để lưu trữ các bộ đếm đặc biệt và thực hiện chức năng cấp thấp. Ví dụ, lệnh SYSENTER chuyển lệnh thự thi đến địa chỉ lưu tại IA32\_SYSENTER\_EIP MSR (0x176), thường là của hệ điều hành xử lý lệnh gọi hệ thống (operating system’s

system call handler). MSRs được thảo luận qua cuốn sách này khi chúng xuất hiện.

### Instruction Set(Tập Lệnh)

        Tập lệnh x86 cho phép mức độ linh hoạt cao trong việc di chuyển dữ liệu giữa các thanh ghi và bộ nhớ. Việt di chuyển dữ liệu có thể chia thành 5 phương pháp :

            - Biến tức thời đến Thanh ghi

            -Thanh ghi đến thanh ghi

            -Biến tức thời đến bộ nhớ

            - Thanh ghi đến bộ nhớ và ngược lại

            -Bộ nhớ đến bộ nhớ

            (■ Immediate to register

            ■ Register to register  
            ■ Immediate to memory  
            ■ Register to memory and vice versa  
            ■ Memory to memory)

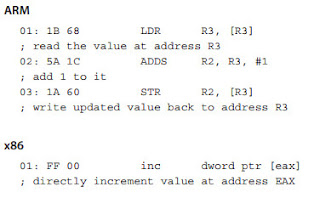
        Bốn phương pháp đầu tiên được hỗ trợ bởi tất cả các kiến trúc hiện đại, nhưng cái cuối đặc biệt chỉ cho x86. Một kiến trúc cổ điển RISC như là ARM chỉ có thể đọc/viết dữ liệu từ/đến bộ nhớ với câu lệnh tải/lưu trữ ( LDR và STR, tương tự); ví dụ, một phép toán đơn giản như tăng một giá trị trong bộ nhớ yêu cầu 3 câu lệnh:

        1. Đọc dữ liệu từ bộ nhớ từ bộ nhớ rồi chuyển đến Thanh ghi (LDR)

        2.Thêm một đơn vị vào Thanh ghi( ADD)

        3.Viết thanh ghi vào bộ nhớ( STR)

        Trong x86 như phép toán trên chỉ yêu cầu chỉ một lệnh( có thể là INC hoặc ADD) bởi vì nó có thể truy cập trực tiếp thẳng vào bộ nhớ. Lệnh MOVS có thể đọc và viết bộ nhớ cùng một thời điểm

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Một đặc điểm quan trọng là x96 sử dụng độ dài câu lệnh khác nhau: Lệnh có thể trong khoảng 1 đến 15 bytes. trong ARM, lệnh chỉ có thể 2 hoặc 4 bytes

        CÚ PHÁP(Syntax)

        Phụ thuộc vào assembler/disassembler, có 2 kí hiệu cú pháp cho code assembly x86, Intel và AT&T:

**Intel**  
        mov ecx, AABBCCDDh  
        mov ecx, [eax]  
        mov ecx, eax  
        **AT&T**  
        movl $0xAABBCCDD, %ecx  
        movl (%eax), %ecx  
        movl %eax, %ecx

        Điều quan trọng cần lưu ý là lệnh giống nhau nhưng viết khác nhau. Có một vài điểm khác giữa ký hiệu Intel và AT&T, nhưng điều đáng chú ý nhất là:

        - Tiền tố thanh ghi của AT&T là %, của giá trị tức thời (immediates) là $. Intel không làm điều đó

        -AT&T thêm một tiền tố cào lệnh để cho biết chiều rộng hoạt động. Ví dụ MOVL(Long), MOVB( BYTE), v.v  Intel không làm thế

        -AT&T đặt toán tử nguồn trước đích, Intel thì ngược lại.

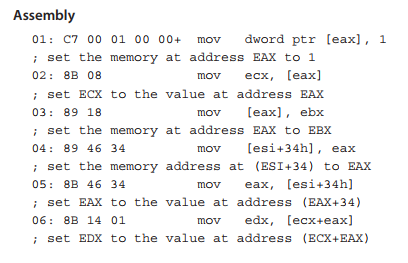
        Dissassemblers/Assemblers và những công cụ dịch ngược( IDA Pro, OllyDbg, MASM, vv) trên Windows thường sử dụng kí hiệu Intel, Trong khi trên UNIX thường sử dụng kí hiệu AT&T(GCC). Trong thực tế kí hiệu Intel được sử dụng chủ yếu và được sử dụng xuyên suốt quyển sách này.

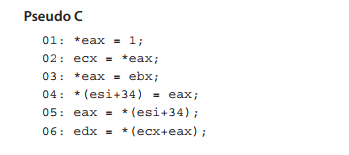
### Di chuyển Dữ liệu(Data Movement)

        Lệnh hoạt động trên giá trị từ thanh ghi hoặc bộ nhớ chính. Các lệnh phổ biến để di chuyển dữ liệu là MOV. Cách sử dụng đơn giản nhất là di chuyển một thanh ghi hoặc một giá trị tức thời đến thanh ghi:

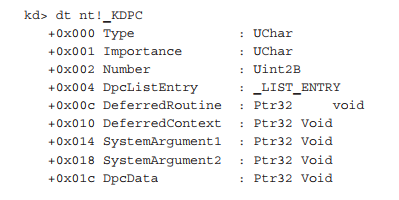
01: BE 3F 00 0F 00       mov esi, 0F003Fh               ;set ESI = 0xF003  
02: 8B F1                       mov esi, ecx                        ;set ESI = ECX

        Cách sử dụng phổ biến tiếp theo là di chuyển dữ liêu đến/từ bộ nhớ. Giống như những quy ước ngôn ngữ assembly, x86 sử dụng ngoặc vuông [] để chỉ bộ nhớ truy cập. ( Ngoại lệ duy nhất là LEA, lệnh sử dụng [] nhưng không thực sự tham chiếu bộ nhớ). Quyền truy cập bộ nhớ có thể theo một số cách khác nhau, chúng ta bắt đầu với trường hợp đơn giản nhất:

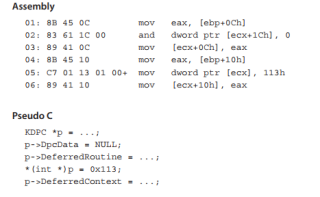
[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Ví dụ này chứng minh khả năng truy cập bộ nhớ thông qua một thanh ghi cơ sở và giá trị, trong đó giá trị có thể là một thanh ghi hay một giá trị tức thời. Hình thức này thường được sử dụng để truy cập các thanh viên cấu trúc hoặc bộ đệm dữ liệu tại một vụ trí được tính toán trong thời gian chạy. Ví dụ, Giả sử ECX trỏ đến cấu trúc KDPC với bố cục

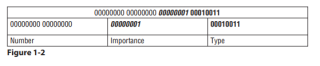
[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        và sử dụng:

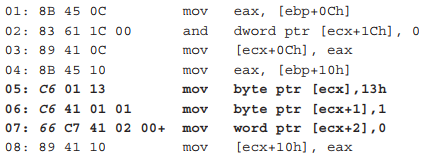
[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Dòng 1 đọc giá trị từ bộ nhớ và lưu vào EAX. Biến  DeferredRoutine được đặt giá trị đó ở dòng 3. Dòng 2 xóa giá trị DpcData bằng cách ADD nó với 0, Dòng 4 đọc một giá trị khác từ bộ nhớ vào lưu nó vòa EAX. Biến DeferredContext được đặt giá trị đó ở dòng 6.

        Dòng 5 viết Double-Word giá trị 0x113 vào cơ sở cấu trúc(Base of structure). Tại sao nó lại viết một giá trị Double-Word vào base mà nó chỉ có kích cỡ 1 byte.Điều đó sẽ không ngầm định đặt các trường Importance và Number ? Câu trả lời là có. Hình 1-2 cho thấy kết quả chuyển đổi 0x113 thành nhị phân

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Type được đặt thành 0x13( bits in đậm), Importance đặt thành 0x, và Number được đặt thành 0x0.Bằng cách viết một giá trị, mã quản lý để khởi tạo 3 trường với 1 lệnh duy nhất. Code có thể như sau:

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Trình biên dịch quyết định gập 3 lệnh vào 1 bởi vì nó biết các hằng số ahead of time( AOT) và muốn tiết kiệm dung lượng. Phiên bản 3 lệnh chiếm 13 bytes( byte thêm ở dòng 7 không được hiển thị), trong khi đó phiên bản 1 lệnh chỉ chiếm 6 bytes. Một cách nhìn thú vị khác là truy cập bộ nhớ có thể hoàn thành ở 3 cấp độ quy mô: byte( dòng 5-6), word( dòng 7), và double word( dòng 1-4,8). quy mô mặc định là 4 bytes, nó có thể chuyển thành 1 hoặc 2 byte cũng với tiền tố viết đè. Trong ví dụ,byte tiền tố viết đè là C6 và 66( in nghiêng). những tiền tố khác sẽ được thảo luận khi ta gặp chúng.

        Mẫu truy cập bộ nhớ tiếp theo thường được sử dụng để truy cập các đối tượng kiểu mạng. Nói chung, có định dạng như sau: [ Base + Index\*scale]

01: 8B 34 B5 40 05+ mov esi, \_KdLogBuffer[esi\*4]  
; luôn được viết là mov esi, [\_KdLogBuffer + esi \* 4]  
; \_KdLogBuffer là địa chỉ cơ sở của mảng toàn cục  
; ESI là chỉ số (index); chúng ta biết rằng mỗi phần tử trong mảng   
; là 4 bytes chiều dài (do đó scale = 4)

02: 89 04 F7 mov [edi+esi\*8], eax  
; ở đây EDI là địa chỉ cơ sở; ESI is là chỉ số mảng(array index)

; kích cỡ phần tử là 8.

   Trong thực tế, điều này được quan sát thấy trong code vòng lặp trên một mảng. Ví dụ:

01: loop\_start:  
02: 8B 47 04 mov eax, [edi+4]  
03: 8B 04 98 mov eax, [eax+ebx\*4]  
04: 85 C0 test eax, eax  
...  
05: 74 14 jz short loc\_7F627F  
06: loc\_7F627F:  
07: 43 inc ebx  
08: 3B 1F cmp ebx, [edi]  
09: 7C DD jl short loop\_

        Dòng 2 đọc một double word tù giá trị +4 từ EDI và sau đó sử dụng nó như một địa chỉ cơ sở vào mảng trong dòng 3; do đó, bạn biết rằng EDI có thể là một cấu trúc có một mảng +4. Dòng 7 tăng chỉ số. Dòng 8 so sánh chỉ số với một giá trị của mảng ở +0 trong cùng một cấu trúc. Với những thông tin đó, vòng lặp nhỏ này có thể decompiled như sau:

typedef struct \_FOO  
{  
DWORD size; // +0x00  
DWORD array[...]; // +0x04  
} FOO, \*PFOO;  
PFOO bar = ...;  
for (i = ...; i < bar->size; i++) {  
if (bar->array[i] != 0) {  
...  
}  
}

        Lệnh MOVSB(byte move)/MOVSW(word move)/MOVSD(double word move) di chuyển dữ liệu với quy mô(độ lớn phần tử) 1-,2-or 4- byte giữa 2 địa chỉ bộ nhớ.  Chúng ngầm hiểu EDI/ESI như là địa chỉ đích/nguồn. Thêm nữa, chúng còn tự động cập nhật địa chỉ nguồn/đích phụ thuốc vào cờ chỉ đạo( Direction flag( DF)) ở trong EFLAGS. Nếu DF = 0, những địa chỉ sẽ bị giảm xuống, ngược là chúng tăng lên. Những lệnh này thường được sử dụng để triển khai chuỗi hoặc bộ nhớ sao chép khi biết độ dài tại thời điểm biên dịch. Trong một vài trường hợp, chúng đi kèm với tiền tố REP., Tiền tố này lặp lại một lệnh tối đa ECX lần. Hãy xem ví dụ sau:

01: BE 28 B5 41 00     mov esi, offset \_RamdiskBootDiskGuid  
; ESI = pointer to RamdiskBootDiskGuid  
02: 8D BD 40 FF FF+    lea edi, [ebp-0C0h]  
; EDI là một địa chỉ đâu đó trên stack  
03: A5                 movsd  
; copies 4 bytes từ EDI đến ESI; tăng từng 4 bytes   
04: A5                 movsd  
; như trên          
05: A5                 movsd  
; như trên  
06: A5                 movsd  
; như trên

Pseudo C  
/\* a GUID là cấu trúc 16-byte  \*/  
GUID RamDiskBootDiskGuid = ...; // toàn cục  
...  
GUID foo;  
memcpy(&foo, &RamdiskBootDiskGuid, sizeof(GUID));

        Dòng 2 đáng được quan tâm đặc biệt. Mặc dù lệnh LEA sử dụng [], nó thực sự không đọc từ địa chỉ bộ nhớ; nó chỉ đơn giản là đánh giá biểu thức trong dấu ngoặc vuông và đặt kết quả vào thanh ghi đích.

Ví dụ nếu EBP là 0x1000, sau đó  EDI sẽ là 0xF40(=0x1000-0xC0) sau đó thực thi dòng 2. Vấn đề là LEA không truy cập vào bộ nhớ, mặc dù cú pháp sai lầm.

        Ví dụ tiếp theo, từ nt!KiInitSystem, sử dụng tiền tố REP:

01: 6A 08          push 8 ;push 8 vào stack( sẽ giải thích stack sau)   
02: ...  
03: 59             pop ecx ; pop the stack. Về cơ bản là đặt ECX bằng 8  
04: ...  
05: BE 00 44 61 00 mov esi, offset \_KeServiceDescriptorTable  
06: BF C0 43 61 00 mov edi, offset \_KeServiceDescriptorTableShadow  
**07: F3 A5          rep movsd ;** sao chép 32 bytes (movsd lặp lại 8 lần (8 lần là giá trị của ECX( count) )  
; từ đây chúng ta có thể suy ra rằng bất kể 2 đối tượng này là gì, chúng đều có thể có kích thước 32 byte

        Mã C tương đương sẽ như sau:

memcpy(&KeServiceDescriptorTableShadow,&KeServiceDescriptorTable,32);

        Ví dụ cuối cùng, nt!MmInitializeProcessAddressSpace, sử dụng một sự kết hợp của những lệnh này bởi vì kích thước bản sao không phải là bội của 4:

01: 8D B0 70 01 00+ lea esi, [eax+170h]  
; EAX có lẽ giống cấu trúc của một địa chỉ cơ sở. nhưng Hãy nhớ những gì chúng ta nói về LEA   
02: 8D BB 70 01 00+ lea edi, [ebx+170h]  
03: A5              movsd  
04: A5              movsd

05: A5              movsd   (doube word move) ( 4byte)

**06: 66 A5           movsw  (word move)(2byte)**  
07: A4              movsb (byte move) (1 byte)

        Sau dòng 1-2, bạn biết rằng EAX và EBX có lẽ cùng kiểu với nhau bời vì chúng đang được sử dụng như source/destination và giá trị bù giống hệt nhau( offset = 170h). Đoạn code này đơn giản sao chép 15 bytes từ một trường cấu trúc này sang một trường cấu trúc khác. Chú ý rằng đoạn code cũng có thể được viết bằng lệnh MOVSB với tiền tố REP và ECX được đặt là 15; tuy nhiên điều đó không hiệu quả vì kết quả là 15 lần đọc thay vì 5

        Một lớp các lệnh di chuyển dữ liệu khác với nguồn và đích bao gồm các lệnh SCAS và STOS. Giống như MOVS, những lệnh này hoạt động với các quy mô 1- 2- hoặc 4-byte. SCAS ngầm hiểu so sánh AL/AX/EAX với dữ diệu bắt đầu từ địa chỉ bộ nhớ EDI; EDI được tự động tăng lên/giảm đi phụ thuộc vào DF bit trong EFFLAGS. Với ngữ nghĩa của nó, SACS thường được sử dụng cùng với tiền tố REP để tìm một byte, word hoặc double-word trong bộ nhớ đệm. Ví dụng hàm strlen() có thể được triển khai như sau:

**01: 30 C0     xor al, al**  
; Đặt AL thành 0 (NULL byte). bạn sẽ thường xuyên quan sát thấy

;XOR reg,reg( register,register)  
**02: 89 FB     mov ebx, edi**  
; save the original pointer to the string( lưu con trỏ gốc vào chuỗi)  
**03: F2 AE     repne scasb**

; liên tục quét về phía trước từng byte một miễn là AL không khớp với ;byte ở EDI khi lệnh này kết thúc, nó có nghĩa là chúng ta chạm đến ;NUL byte trong bộ nhớ đệm  
**04: 29 DF     sub edi, ebx**

;edi bây giờ là vị trí NUL byte. trừ nó cho con trỏ ban đầu ta được độ dài của chuỗi

        STOS giống SCAS ngoại trừ rằng nó viết giá trị AL/AX/EAX vào EDI. Nó thường được sử dụng để khởi tạo bộ nhớ đệm cho một giá trị hằng( như hàm memset()).Đây là ví dụ:

**01: 33 C0     xor eax, eax**  
; Đặt EAX thành 0  
**02: 6A 09     push 9**  
; push 9 vào stack  
**03: 59        pop ecx**  
; pop nó lại vào ECX. Now ECX = 9.  
**04: 8B FE     mov edi, esi**  
; Đặt địa chỉ đích  
**05: F3 AB     rep stosd**

;viết 36 byte 0 vào bộ nhớ đệm đích( STOSD lặp lại 9 lần)

;điều này tương đương với memset(edi,0,36)

    LODS là một lệnh khác giống các lệnh trên. nó đọc một giá trị 1-, 2-, or 4-byte từ ESI và lưu nó vào AL,AX or EAX

#### **Bài tập**

**1.**Hàm này sử dụng kết hợp SCAS và STOS để làm việc của nó. Đầu tiên, giải thích kiểu của [EBP+8] và [EBP+C] ở dòng 1 và 8. Tiếp theo giải thích đoạn code này làm gì

**01: 8B 7D 08        mov edi, [ebp+8]**

;ebp+8 là kí tự đầu tiên của chuỗi

;Đặt edi = địa chỉ của stack base pointer +8   
**02: 8B D7           mov edx, edi**

; đặt edx = edi  
**03: 33 C0           xor eax, eax**

; đặt eax = 0

**04: 83 C9 FF        or ecx, 0FFFFFFFFh**

; đặt ecx = 0FFFFFFFFh = -1   
**05: F2 AE           repne scasb**

;hàm scas so sánh eax vs edi đến khi bằng nhau thì dừng lại( đang tìm null của chuỗi)

;cho nên sẽ scan đến khi nào giá trị ở địa chỉ của edi = null  
**06: 83 C1 02        add ecx, 2**

;ecx=ecx+2 ( phải trừ đi 2 đơn vị vì 1 của null 1 của giá trị -1 lúc đầu ta cho vào)  
**07: F7 D9           neg ecx (neg = negation = phủ định)**

;ecx=-ecx đổi ecx lại thành dương ta được độ dài của string  
**08: 8A 45 0C        mov al, [ebp+0Ch]**

;al là 1 byte, nên [ebp+0Ch] có thể là một ký tự  
**09: 8B FA           mov edi, edx**

;đặt edi lại thành vị trí ban đầu của chuỗi  
**10: F3 AA           rep stosb**

; viết giá trị của eax( chúng ta đang không biết eax = bao nhiêu sau khi đặt al = 1 kí tự nào đó) vào edi cho đến khi ecx =0( ecx đang bằng độ dài chuỗi)  
**11: 8B C2           mov eax, edx**

;ta chuyển lại edx( vị trí đầu của chuỗi vào lại eax)

vì vậy đoạn code trên là ghi đè 1 kí tự ở trong **al** lên chuỗi ban đầu của chúng ta

Video giải thích: [https://www.youtube.com/watch?v=1zY7lbcbBZQ](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

#### Các phép tính toán học( Arithmetic Operations)

        Các phép toán cơ bản như là cộng, trừ nhận và chia được hỗ trợ bởi tập lệnh. Hoạt động ở lớp Bit như là AND, OR, XOR, NOT, và dịch trái dịch phải cũng có lệnh. Ngoại trừ phép nhân và phép chia, các lệnh còn lại rất đơn giản về cách sử dụng. Các phép tính được giải thích bằng ví dụ sau:

01: 83 C4 14     add esp, 14h ; esp = esp + 0x14  
02: 2B C8        sub ecx, eax ; ecx = ecx - eax  
03: 83 EC 0C     sub esp, 0Ch ; esp = esp - 0xC  
04: 41           inc ecx ; ecx = ecx + 1  
05: 4F           dec edi ; edi = edi - 1  
06: 83 C8 FF     or eax, 0FFFFFFFFh ; eax = eax | 0xFFFFFFFF  
07: 83 E1 07     and ecx, 7 ; ecx = ecx & 7  
08: 33 C0        xor eax, eax ; eax = eax ^ eax  
09: F7 D7        not edi ; edi = ~edi  
10: C0 E1 04     shl cl, 4 ; cl = cl << 4  
11: D1 E9        shr ecx, 1 ; ecx = ecx >> 1  
12: C0 C0 03     rol al, 3 ; rotate AL left 3 positions  
13: D0 C8        ror al, 1 ; rotate AL right 1 position

    Lệnh dịch trái và dịch phải bit ( dòng 11- 12) có một số giải thích, như chúng thường được quan sát trong code thực tế. Lệnh này thường được sử dụng để tối ưu hóa các phép toán nhân và chia trong đó phép nhân và phép chia là 1 lũy thừa bậc 2.Loại tối ưu hóa náy đôi khi được biết đến như giảm sức mạnh vì nó thay thế một hoạt động tính toán tốn kém với một cái rẻ hơn. Ví dụ: phép chia số nguyên hoạt động tương đối chậm, nhưng khi số chia là lũy thừa của 2, nó có thể được giảm xuống chuyển thành các bit bên phải; 100/2 giống với 100>>1. Tương tự, nhân lũy thừa 2 có thể được giảm để dịch chuyển các bit sang trái 100\*2 giống với 100<<1.

    Phép nhân không dấu và có dấu được thực hiện lần lượt thông qua các lệnh MUL và IMUL. Lệnh MUL có dạng chung sau MUL reg/memory. Có nghĩa là, nó chỉ có thể hoạt động  trên các thanh ghi hoặc bộ nhớ. Thanh ghi được nhân với AL, AX hoặc EAX và kết quả được lưu trữ trong AX, DX:AX hoặc EDX:EAX, phụ thuộc vào độ dài của toán hạng. Ví dụ

**01: F7 E1        mul ecx** ; EDX:EAX = EAX \* ECX  
**02: F7 66 04     mul dword ptr [esi+4]** ; EDX:EAX = EAX\*dword\_at(ESI+4)  
**03: F6 E1        mul cl ; AX = AL \* CL**  
**04: 66 F7 E2     mul dx ; DX:AX = AX \* DX**  
Xem một số ví dụ khác:   
**01: B8 03 00 00 00 mov eax,3** ; set EAX=3  
**02: B9 22 22 22 22 mov ecx,22222222h** ; set ECX=0x22222222  
**03: F7 E1          mul ecx** ; EDX:EAX = 3 \* 0x22222222 =  
; 0x66666666  
; hence, EDX=0, EAX=0x66666666  
**04: B8 03 00 00 00 mov eax,3** ; set EAX=3  
**05: B9 00 00 00 80 mov ecx,80000000h** ; set ECX=0x80000000  
**06: F7 E1          mul ecx** ; EDX:EAX = 3 \* 0x80000000 =  
                           ; 0x180000000  
                           ; hence, EDX=1, EAX=0x80000000

  Lý do tại sao kết quả được lưu trong EAX cho phép nhân 32-bit là bởi vì kết quả có thể không phù hợp với một thanh ghi 32 bit( như đã chứng mình ở dòng 4 -6)

IMUL có 3 dạng:

■ IMUL reg/mem — Same as MUL  
■ IMUL reg1, reg2/mem — reg1 = reg1 \* reg2/mem  
■ IMUL reg1, reg2/mem, imm — reg1 = reg2 \* imm

Some disassemblers shorten the parameters. For example:  
01: F7 E9           imul ecx ; EDX:EAX = EAX \* ECX  
02: 69 F6 A0 01 00+ imul esi, 1A0h ; ESI = ESI \* 0x1A0

03: 0F AF CE        imul ecx, esi ; ECX = ECX \* ESI

Việc chia có dấu và không dấu được thực hiện thông qua các lệnh DIV và IDIV. chúng chỉ nhận một tham số( số chia) và có các dạng sau: DIV/IDIV reg/mem. Phụ thuộc vào kích cỡ số chia, DIV sẽ sử dụng AX, DX:AX hoặc EDX: EAX là số chia, còn thương/phần dư thu được sẽ được lưu trong AL/AH, AX/DX, or EAX/EDX. Ví dụ:

**01: F7 F1          div ecx** ; EDX:EAX / ECX, thương in EAX,  
**02: F6 F1          div cl** ; AX / CL, thương in AL, số dư in AH  
**03: F7 76 24       div dword ptr [esi+24h]** ; see line 1  
**04: B1 02          mov cl,2** ; Đặt CL = 2  
**05: B8 0A 00 00 00 mov eax,0Ah** ; Đặt EAX = 0xA  
**06: F6 F1          div cl** ; AX/CL = A/2 = 5 in AL (thương),  
; AH = 0 (số dư)  
**07: B1 02          mov cl,2** ; Đặt CL = 2  
**08: B8 09 00 00 00 mov eax,09h** ; đặt EAX = 0x9  
**09: F6 F1          div cl** ; AX/CL = 9/2 = 4 in AL (thương),  
; AH = 1 (số dư)

#### Hoạt Động Ngăn Xếp(Stack) và lệnh gọi hàm

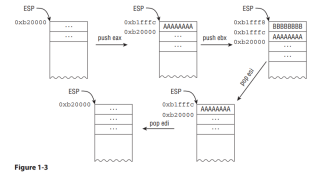
    Ngăn xếp(stack) là cấu trúc dữ liệu cơ bản trong ngôn ngữ lập trình và hệ điều hành. Ví dụ, biến cục bộ trong C được lưu trữ trên ngăn xếp của hàm. Khi hệ điều hành chuyển đổi từ ring 3 đến ring 0, nó lưu trạng thái thông tin vào stack. Về mặt khái niệm, ngăn xếp là cấu trúc dữ liệu nhập trước xuất sau( first in last out) hỗ trợ 2 haojt động: push and pop. Push có nghĩa là cho thứ gì đó lên trên đầu của stack, pop nghĩa là lấy một thứ từ trên đầu của stack. Một cách cụ thể, trong x86, Stack là một vùng bộ nhớ liền kề được trỏ tới bởi ESP(stack pointer) và nó phát triển xuống dưới. Các hoạt động push/pop được thực hiện thông qua lệnh PUSH/POP và chúng hoàn toàn sửa đỏi ESP. Lệnh PUSH giảm ESP xuống và sau đó ghi dữ liệu tại vị trí được chỉ ra bởi ESP; POP đọc dữ liệu và tăng ESP lên. Giá trị tăng giảm tự động mặc định là 4, nhưng nó có thể thay đổi thành 1 or 2  với đi đè tiền tố. Trong thực tế, giá trị hầu như luôn luôn là 4 bởi vì hệ điều hành yêu câù stack phải được căn chỉnh theo double-word.

    Giải sử rằng ESP trỏ tới 0xb20000 và bạn có đoạn code dưới đây:

; khởi tạo ESP = 0xb20000  
**01: B8 AA AA AA AA mov eax,0AAAAAAAAh  
02: BB BB BB BB BB mov ebx,0BBBBBBBBh  
03: B9 CC CC CC CC mov ecx,0CCCCCCCCh  
04: BA DD DD DD DD mov edx,0DDDDDDDDh  
05: 50             push eax**  
; địa chỉ 0xb1fffc sẽ chữa giá trị 0xAAAAAAAA and ESP  
; sẽ là 0xb1fffc (=0xb20000-4)

**06: 53             push ebx**  
; address 0xb1fff8 sẽ chứa giá trị 0xBBBBBBBB and ESP  
; sẽ là 0xb1fff8 (=0xb1fffc-4)  
**07: 5E             pop esi**  
; ESI sẽ chứa giá trị 0xBBBBBBBB and ESP sẽ là 0xb1fffc  
; (=0xb1fff8+4)  
**08: 5F             pop edi**  
; EDI sẽ chứa giá trị 0xAAAAAAAA and ESP sẽ là 0xb20000  
; (=0xb1fffc+4)

Hình 1-3 sẽ minh họa stack layout

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

    ESP cũng có thể sửa đổi trực tiếp bằng các lệnh khác chẳng hạn như ADD và SUB.

          Trong khi các ngôn ngữ lập trình cấp cao có khái niệm về hàm có thể được gọi và trả về từ đó, bộ xử lý không cung cấp tính trừu tượng như vậy. Ở mức thấp nhất, bộ xử lý chỉ hoạt động trên các đối tượng cụ thể, chẳng hạn như dưới dạng thanh ghi hoặc dữ liệu đến từ bộ nhwos. Các chức năng được dịch như thế nào tại machine level? Chúng được thực hiện qua cấu trúc dữ liệu ngăn xếp( stack data structure) Xem ví dụ sau   
**C**  
int  
\_\_cdecl addme(short a, short b)  
{  
return a+b;  
}  
**Assembly**  
01: 004113A0 55          push ebp

02: 004113A1 8B EC       mov ebp, esp  
03: ...  
04: 004113BE 0F BF 45 08 movsx eax, word ptr [ebp+8]  
05: 004113C2 0F BF 4D 0C movsx ecx, word ptr [ebp+0Ch]  
06: 004113C6 03 C1       add eax, ecx  
07: ...  
08: 004113CB 8B E5       mov esp, ebp  
09: 004113CD 5D          pop ebp  
10: 004113CE C3          retn

Hàm được gọi bằng đoạn mã sau:

**C**  
sum = addme(x, y);  
**Assembly**  
01: 004129F3 50             push eax  
02: ...  
03: 004129F8 51             push ecx  
04: 004129F9 E8 F1 E7 FF FF call addme  
05: 004129FE 83 C4 08       add esp, 8

        Trước khi đi vào chi tiết, trước tiên hãy xem xét lệnh CALL/RET và quy ước gọi( calling conventions). Lệnh CALL thực hiện 2 hành động:

        1. Nó đẩy địa chỉ trả về( địa chỉ ngay sau lệnh CALL) lên ngăn xếp

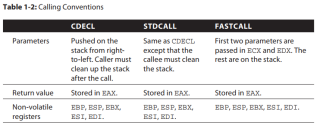
        2.Nó thay đổi EIP thành call destination. Điều này có hiệu quả chuyển quyền kiểm soát cho mục tiêu CALL và bắt đầu thực hiện ở đó.

        RET đơn giản chỉ cần đưa địa chỉ được lưu trữ trên cùng của stack vào EIP và chuyển quyền điều khiển đến nó (nghĩa đen giống như POP EIP nhưng chuỗi lệnh như vậy không tồn tại trên x86). Ví dụ: nếu bạn muốn bắt đầu thực thi ở 0x12345678, bạn chỉ có thể làm như sau:

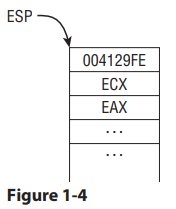
01: 68 78 56 34 12 push 0x12345678  
02: C3             ret

   Một calling convention là một tập hợp các quy tắt chỉ định cách hoạt động của lệnh CALL tại machine level. Nó được xác định bởi Application Binary Interface (ABI) cho một hệ thống cụ thể.Ví dụ: các tham số được chuyển qua ngăn xếp, trong thanh ghi, hay là cả hai? các tham số được chuyển từ trái sang phải hay phải sang trái? Giá trị trả về có nên được lưu trữ trên stack, thanh ghi hay cả hai không? có nhiều quy ước, nhưng những quy ước phổ biến nhất là CDECL,STDALL,THISCALL, and FASTCALL.( Trình biên dịch cũng có thể tạo quy ước tùy chỉnh của riêng nó, nhưng những quy đước đó sẽ không được thảo luận ở đây)

Bảng 1-2 tóm tắt nghĩa của chúng:

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Bây giờ chúng ta quay lại đoạn mã để thảo luận về cách hàm addme được gọi như thế nào. Trong dòng 1 và 3, hai tham sổ được đẩy lên ngăn xếp; ECX và EAX lần lượt là tham số đầu tiên và tham số thứ 2. Dòng 4 gọi hàm addme với lệnh CALL. Điều này ngay lập tức đẩy địa chỉ trả lại 0x4129FE, trên ngắn xếp và bắt đầu thực thi ở 0x113A0. Hình 1-4 minh họa bố cục ngăn xếp sau khi dòng 4 được thực thi

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Sau khi  dòng 4 thực thi, bây giờ chúng ta đang ở trong phần thân hàm addme. Dòng 1 đẩy EBP lên ngăn xếp. Dòng 2 đặt EBP thành con trỏ ngăn xếp hiện tại. Chuỗi 2 lệnh này thường được gọi là phần mở đầu hàm vì nó thiết lập một khung hàm mới. Dòng 4 đọc giá trị tại địa chỉ EBP +8, là tham số đầu tiên trên ngăn xếp; dòng 5 đọc tham số thứu 2. Lưu ý rằng các tham số được truy cập bằng EBP làm thanh ghi cơ sở. Khi được sử dụng trong ngữ cảnh này, EBP được gọi là con trỏ khung cơ sở( xem dòng 2) vì nó trỏ đến khung ngăn xếp cho hàm hiện tại, và tham số/ local có thể được truy cập liên quan đến nó. Trình biên dịch cũng có thể được lệnh tạo mã không sử dụng EBP làm con trỏ khung cơ sở thông qua một số tối ưu hóa được gọi là bỏ qua con trỏ khung. với sự tối ưu hóa như vậy, quyền truy cập vào các biến và tham số cục bộ được thực hiện liên quan đến ESP và EBP có thể được sử dụng như một thanh ghi như EAX,EBX,ECX, v,v Dòng 6 thêm các số và thên kết quả vào EAX. DÒng 8 đặt con trỏ ngăn xếp thành con trỏ khung cơ sở.Dòng 9 đưa EBP đã lưu từ dòng 1 vào EBP. Chuỗi 2 lệnh này thường được gọi là phần kết thức của hàm vì nó nằm ở cuối hàm và khôi phục là khung hàm trước đó. Tại thời điểm này, trên cùng của ngăn xếp chứa địa chỉ trả về được lưu bởi lệnh CALL tại 0x4129F9. Dòng 10 thực hiện một RET, pops ngăn xếp và tiếp tục thực thi ở 0x4129FE. Dòng 5 trong đoạn mã thu nhỏ ngăn xếp đi 8 vì người gọi mải dọn dẹp ngăn xếp theo quy ước CDECL.

        Nếu hàm addme có các biến cục bộ, thì mã sẽ cần phải phát triển ngăn xếp bằng các trừ ESP sau dòng 2. Tất cả các biến cục bộ sẽ có thể truy cập được thông qua phần bù âm của EBP.

#### Bài tập

1.Với những gì bạn học về CALL và RET, giải thích cách bạn sẽ đọc giá trị EIP? Tại sao bạn không thể chỉ làm MOV EAX,EIP?

2.Đưa ra ít nhất 2 chuỗi mã để đặt EIP thành 0xAABBCCDD

3.Trong hàm ví dụ, addme, điều gì sẽ xảy ra nếu con trỏ ngăn xếp không được khôi phục đúng cách trước khi thực hiện lệnh RET

4.Trong quy ước gọi hàm đã được  giải thích, giá trị trả về được lưu trong thanh ghi 32-bit(EAX). Điều gì xảy ra khi giá trị trả về không vừa với thanh ghi 32-bit? Viết trường trình để thử nghiệm và đánh giá câu trả lời của bạn. Cơ chế có thay đổi từ trình biên dịch sang trình biên idchj không ?

Bài làm:

    1.

|  |
| --- |
| **call f** |
|  |  |
|  | ; f() sau khi call hàm f() địa chỉ của dòng ngay sau lệnh call (EIP) sẽ được đẩy ;vào stack nằm trên cùng của stack, lúc này eip = esp |
|  | **mov eax, [esp]** ;sau đó ra lấy bằng cách lấy giá trị của địa tại ESP( địa chỉ EIP) |
|  | **ret** |

        không thể MOV EAX,EIP vì EIP không thể là source operand of mov.( xem ở phần intruction set)

    2.  cách 1    push  0xAABBCCDD

          ret

   cách 2

     mov eax,0xAABBCCDD

     jmp eax (call eax)

    cách 3

    jmp 0xAABBCCDD

|  |
| --- |
|  |

3.thì nó sẽ trả giá trị ESP về địa chỉ EIP sai và dẫn đến nhảy sai đến địa chỉ không mong muốn

4.[https://tekwizz123.blogspot.com/2017/11/practical-reverse-engineering-chapter-1\_9.html](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

#### Luồng Điều Kiện( Control Flow)

        Phần này môi tả cách hệ thống triển khai các cấu trúc cấp cao hơn thực thi có điều kiện như if/else, switch/case và while/for. Tất cả những điều này là được thực hiện thông qua các lệnh CMP, TEST, JMP, và Jcc và thanh ghi EFLAGS. Danh sách sau đây tóm tắt các cờ phổ biến trong EFLAGS:

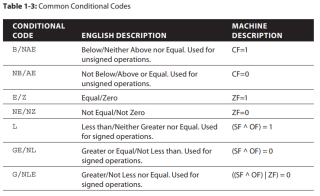
            ZF/Zero flag - Được bật nếu kết quả của phép toán trước =0

            SF/Sign flag - Được đặt thành các bit có nghĩa của kết quả

            CF/Carry flag- Được bật khi kết quả yêu câù thực hiện. Nó áp dụng cho số không dấu

            OF/Overflow flag- Được bật nếu kết quả vượt quá kích thước tối đa. Nó áp dụng cho số có dấu

    Lệnh toán học sẽ cập nhật những flags này dựa vào kết quả. Ví dụ, Lệnh SUB EAX, EAX sẽ bật cờ ZF. Lệnh Jcc, trong đó 'cc' là mã có điều kiện, thay đổi luồng điều khiển tùy thuộc vào cờ. Có tới 16 mã điều kiện, nhưng những mã phổ biến nhất được mô tả trong bảngA 1-3

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

   Bởi vì ngôn ngữ assembly không có một hệ thống kiểu xác định, một trong số ít cách để nhận ra kiểu có dấu/không dấu là thông qua các mã điều kiện này.

        Lệnh CMP so sánh 2 toán hạng và đặt mã điều kiện thích hợp vào EFLAGS; nó so sánh 2 số bằng sách trừ một số cho 1 số khác mà không cập nhật kết quả. Còn lệnh TEST thực hiện điều tương tự ngoại trừ nó thực hiện một lệnh AND giữa 2 toán hạng

#### if-else

        Cấu trúc If-else khá đơn giản để nhận ra vì chúng liên quan đến so sánh/test theo sau đó là Jcc. Ví dụ:

**Assembly**

01: mov esi, [ebp+8]  
02: mov edx, [esi]  
03: test edx, edx  
04: jz short loc\_4E31F9  
05: mov ecx, offset \_FsRtlFastMutexLookasideList  
06: call \_ExFreeToNPagedLookasideList@8  
07: and dword ptr [esi], 0  
08: lea eax, [esi+4]  
09: push eax  
10: call \_FsRtlUninitializeBaseMcb@4  
11: loc\_4E31F9:

12: pop esi  
13: pop ebp  
14: retn 4  
15: \_FsRtlUninitializeLargeMcb@4 endp

**Pseudo C**  
if (\*esi == 0) {  
return;  
}  
ExFreeToNPagedLookasideList(...);  
\*esi = 0;  
...  
return;  
OR  
if (\*esi != 0) {  
...  
ExFreeToNPagedLookasideList(...);  
\*esi = 0;  
...  
}  
return;

        Dòng 2 đọc giá trị tại ví trị của ESI và lưu trữ nó vào EDX. Dòng 3 ADNs EDX với chính nó và đặt cờ thích hợp trong EFLAGS. Chú ý rằng mẫu này thường được sử dụng để xác định xem một thanh ghi có bằng 0 hay 0. Dòng 4 ngảy đến loc\_4E31F9(dòng 12) nếu ZF =1. Nếu ZF=0, sau đó thực hiện dòng 5 và tiếp tục cho đến khi hàm trả về. Chú ý rằng có 2 bản dịch C hơi khác nhau nhưng tương đương về mặt logic cho đoạn mã này.

#### Switch-case

                Một khối switch-case là một chuỗi lệnh if/esle. ví dụ

#### Switch-case

switch(ch) {  
    case 'c':  
        handle\_C();  
        break;  
    case 'h':  
        handle\_H();  
        break;  
    default:  
        break;  
}  
domore();  
...

**If-Else**  
if (ch == 'c') {  
    handle\_C();  
} else  
if (ch == 'h') {  
    handle\_H();  
}  
domore();  
...  
         Do đó, mã máy chuyển đổi sã là một chuỗi if/else. Ví dụ minh họa:

**Assembly**  
01: push ebp  
02: mov ebp, esp  
03: mov eax, [ebp+8]  
04: sub eax, 41h  
05: jz short loc\_caseA  
06: dec eax  
07: jz short loc\_caseB  
08: dec eax  
09: jz short loc\_caseC  
10: mov al, 5Ah  
11: movzx eax, al  
12: pop ebp  
13: retn  
14: loc\_caseC:  
15: mov al, 43h  
16: movzx eax, al  
17: pop ebp  
18: retn  
19: loc\_caseB:  
20: mov al, 42h  
21: movzx eax, al  
22: pop ebp  
23: retn  
24: loc\_caseA:  
25: mov al, 41h  
26: movzx eax, al  
27: pop ebp  
28: retn

**C**  
unsigned char switchme(int a)  
{  
unsigned char res;

switch(a) {  
    case 0x41:  
        res = 'A';  
        break;  
    case 0x42:  
        res = 'B';  
        break;  
    case 0x43:  
        res = 'C';  
        break;  
    default:  
        res = 'Z';  
        break;  
}  
return res;  
}

     Switch-case thực sự có thể phức tạp hơn, và trình biên dịch thường xây dựng một *jump table*giảm thiểu số lầm so sánh và các bước nhảy có điều kiện. *jump table* là một mảng địa chỉ, mỗi địa chỉ trỏ đến một trường hợp cụ thể. Mẫu này có thể quan sát thấy trong Sample J trong sub\_10001110:

**Assembly**  
01: cmp edi, 5  
02: ja short loc\_10001141  
03: jmp ds:off\_100011A4[edi\*4]  
04: loc\_10001125:  
05: mov esi, 40h  
06: jmp short loc\_10001145  
07: loc\_1000112C:  
08: mov esi, 20h  
09: jmp short loc\_10001145  
10: loc\_10001133:  
11: mov esi, 38h  
12: jmp short loc\_10001145  
13: loc\_1000113A:  
14: mov esi, 30h  
15: jmp short loc\_10001145  
16: loc\_10001141:  
17: mov esi, [esp+0Ch]  
18: ...  
19: off\_100011A4 dd offset loc\_10001125  
20: dd offset loc\_10001125  
21: dd offset loc\_1000113A  
22: dd offset loc\_1000112C  
23: dd offset loc\_10001133  
24: dd offset loc\_1000113A

**Pseudo C**  
switch(edi) {  
case 0:  
case 1:  
// goto loc\_10001125;  
esi = 0x40;  
break;  
case 2:  
case 5:  
// goto loc\_1000113A;  
esi = 0x30;  
break;  
case 3:  
// goto loc\_1000112C;  
esi = 0x20;  
break;  
case 4:  
// goto loc\_10001133;  
esi = 0x38;  
break;  
default:  
// goto loc\_10001141;  
esi = \*(esp+0xC)  
break;  
}  
...

        Ở đây, trình biên dịch hiểu rằng có 5 trường hợp và giá trị trường hợp là liên tiếp; do đó, nó có thể xây dựng bảng nhảy và lập chỉ mục trực tiếp vào đó( Dòng 3). Nếu không có bảng nhảy, sẽ có thêm 10 lệnh để test mỗi trường hợp và phân nhánh để xử lý( có các hình thức chuyển đổi switch/case tối ưu hóa, nhưng tôi sẽ không đề cập ở đây)

#### Loop

        Ở cấp độ mã máy, vòng lặp được thực hiện bằng cách kết hợp Jcc và JMP. Nói cách khác, chúng được triển khai bằng cách sử dụng if/else và goto. Cách tốt nhất để hiểu điều này là viết là một vòng lặp chỉ dùng mỗi if/else và goto. Xem ví dụ sau:

**Using for**

for (int i=0; i<10; i++) {  
printf("%d\n", i);  
}  
printf("done!\n");

**Using if/else and goto**  
int i = 0;  
loop\_start:  
if (i < 10) {  
printf("%d\n", i);  
i++;  
goto loop\_start;  
}  
printf("done!n");

Khi được biên dịch, cả 2 phiên bản giống nhau ở lớp mã máy

01: 00401002 mov edi, ds:\_\_imp\_\_printf  
02: 00401008 xor esi, esi  
03: 0040100A lea ebx, [ebx+0]  
04: 00401010 loc\_401010:  
05: 00401010 push esi  
06: 00401011 push offset Format         ; "%d\n"  
07: 00401016 call edi     ; \_\_imp\_\_printf  
08: 00401018 inc esi  
09: 00401019 add esp, 8     ;esp = esp +8

10: 0040101C cmp esi, 0Ah  
11: 0040101F jl short loc\_401010  
12: 00401021 push offset aDone     ; "done!\n"  
13: 00401026 call edi     ; \_\_imp\_\_printf  
14: 00401028 add esp, 4

        Dòng 1 đặt EDI thành hàm printf. Dòng 2 đặt ESI về 0. Dòng 4 bắt đầu vòng lặp; tuy nhiên, chú ý rằng nó không bằng đầu với một so sánh. Không có phép so sánh ở đây bởi vì trình biên dịch biết rằng biến đến được khởi tạo =0 ( dòng 2) và hiển nhiên nhỏ hơn 10 nên bỏ qua check. Dòng 5-7 gọi hàm printf với đúng tham số( mã định dạng và số của chúng ta). Dòng 8 tăng biến đếm lên. Dòng 9 làm sạch stack bởi vì printf sử dụng CDECL (calling convention). Dòng 10 kiểm tra xem biếm đếm có nhỏ hơn 0xA không. nếu nhỏ hơn, nhảy trở lại loc\_401010.Nếu không nhỏ hơn 0xA, nó tiếp tục thực hiện dòng 12 và kết thúc với printf.

        Một quan sát quan trọng cần thực hiện là việc disassembly cho phép chúng ta suy ra rằng biến đếm là một số có dấu. DÒng 11 sử dụng câu điều kiện ''less than" ( JL), vì thế chúng ta biết ngay rằng phép so sánh được thực hiện trên các số nguyên có dấu. Hãy nhớ Nếu"below/above", nó là không dấu, nếu "Less than/greater than" có dấu. Ví dụ L có một hàm, sub\_100AE3B, với vòng lặp thú vị

**Assembly**  
01: sub\_1000AE3B proc near  
02: push edi

03: push esi  
04: call ds:lstrlenA  
05: mov edi, eax  
06: xor ecx, ecx  
07: xor edx, edx  
08: test edi, edi  
09: jle short loc\_1000AE5B  
10: loc\_1000AE4D:  
11: mov al, [edx+esi]  
12: mov [ecx+esi], al  
13: add edx, 3  
14: inc ecx  
15: cmp edx, edi  
16: jl short loc\_1000AE4D  
17: loc\_1000AE5B:  
18: mov byte ptr [ecx+esi], 0  
19: mov eax, esi  
20: pop edi  
21: retn  
22: sub\_1000AE3B endp

**C**  
char \*sub\_1000AE3B (char \*str)  
{  
int len, i=0, j=0;  
len = lstrlenA(str);  
if (len <= 0) {  
str[j] = 0;  
return str;  
}  
while (j < len) {  
str[i] = str[j];  
j = j+3;  
i = i+1;  
}  
str[i] = 0;  
return str;  
}

        Hàm sub\_1000AE3B có một tham số được truyền bằng cách sử dụng quy ước gọi tùy chỉnh( ESI giữ tham số). Dòng 2 lưu EDI. Dòng 3 gọi lstrlenA với tham số; do đó, bạn hiểu ngay rằng ESI là kiểu char \*. Dòng 5 lưu giá trị trả về( string length) vào EDI. Dòng 6-7 clear ECX và EDX. Dòng 8-9 kiểm tra xem nếu độ dài string bé hơn hay bằng 0. Nếu nó bé hơn hoặc bằng 0, sẽ chuyển đến dòng 18, nó sẽ đặt trá trị tại ECX+ESI thành 0. Nếu nó không, sau đó tiếp tục thực hiện dòng 11, điều đó bằng đầu vòng lặp. Đầu tiên, nó đọc kí tự ở trong ESI+EDX( dòng 11), và sau đố nó lưu tại ESI+ECX( dòng 12)

Tiếp theo, nó tăng EDX lên 3 và ECX lên 1. Dòng 15 16 kiểm tra xem nếu EDX ít hơn độ dài string; nếu thế nhảy lại về đầu vòng lặp, nếu không thì thực hiện tiếp dòng 18.

        Thoạt đầu, nó có vẻ phức tạp, nhưng hàm này nhận một chuỗi bị xáo trộn có giá trị  được giải mã là mỗi kí tự thứ 3. Ví dụ chuỗi SX]*O*TYFKPTY^W\aAFKRW\\E thực sự là SOFTWARE. Mục đích của hàm này là ngăn chặn các máy quét chuỗi ngây thơ và trốn tránh sự phát hiện. Như một bài tập, bạn nên dịch ngược hàm này đến nó trông "tự nhiên" hơn( trái ngược bới bản dịch theo nghĩa đen của tôi)

        Bên cạnh cấu trúc Jcc thông thường, một số vòng lặp nhất định có thể được triển khai bằng cách sử dụng lệnh LOOP thực thi một khối mã ECX lần. Ví dụ

        Assembly

01: 8B CA     mov ecx, edx  
02:         loc\_CFB8F:  
03: AD         lodsd  
04: F7 D0      not eax  
05: AB         stosd  
06: E2 FA      loop loc\_CFB8F  
        Rough C  
while (ecx != 0) {  
    eax = \*edi;  
    edi++;  
    \*esi = ~eax;  
    esi++;  
    ecx--;  
}

        Dòng 1 đọc biến đếm từ EDX. Dòng 3 bắt đầu vòng lặp, nó đọc một giá trị double-word ở  địa chỉ EDI và lưu nó vào EAX; nó cũng đồng thời tăng EDI lên 4. Dòng 4 thực hiện toán tử NOT trên giá trị vừa đọc. Dòng 5 viết giá trị sửa đổi vào trong bộ nhớ địa chỉ ESI và tăng ESI lên 4. Dòng 6 kiểm tra xem nếu ECX =0; nếu không thì thực hiện lại vòng lặp.

#### System Mechanism ( Cơ chế hệ thống)

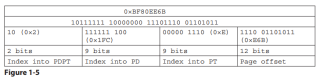
        Phần trước giải thích các cơ chế và lệnh có sẵn để mã chạy ở tất cả các cấp đặc quyền. Để đánh giá tốt hơn về kiến trúc, phần này bàn luận về 2 cơ chế cấp phát hệ thống cơ bản: virtual address translation (dịch địa chỉ ảo) và excetion/interrupt handding( xử lý lỗi và ngắt). Bạn có thể bỏ qua phần này trong lần đọc đầu tiên.

#### Address Translation( Dịch địa chỉ)

        Bộ nhớ vật lý trên hệ thống máy tính được chia thành các đơn vị 4KB gọi là *pages*( một page có thể hơn 4KB, nhưng chúng ta sẽ không bàn luận đến những kích thước khác ở đây). Địa chỉ bộ nhớ được chia thành 2 loại: ảo và vật lý.Địa chỉ ảo là những địa chỉ đc sử dụng bởi các lệnh được thực thi trong bộ xử lý khi tính tăng paging được bật. Ví dụ

01: A1 78 56 34 12 mov eax, [0x12345678]; đọc bộ nhớ tại địa chỉ ảo   
                                        ; 0x12345678  
01: 89 08 mov [eax], ecx ; viết ECX vào địa chỉ ảo EAX

        Địa chỉ vật lý là vị trí bộ nhớ thực tế được sử dụng bởi bộ xử lý khi truy cập bộ nhớ. Đơn vị quản lý bộ nhớ của bộ xử lý ( Memory management unit( MMU)) chuyển mọi địa chỉ ảo thành địa chỉ vật lý trước khi truy cập. Mặc dù một địa chỉ ảo vó vẻ giống như một số khác đối với người dùng, nhưng có một cấu trúc đối với nó khi được xem bởi MMU. Trên x86 với phần mở rộng địa chỉ vậy lý ( physical address extention(PAE)) hỗ trợ, một địa chỉ bộ nhớ ảo có thể được chia thành các chỉ mục thành 3 bảng và giá trị: page directory pointer table(PDPT), page directory( PD), page table(PT), and page table entry (PTE).PDPT là một mảng của 4 phần tử 8 byte, mỗi cái trỏ đến một PD. PD là một mảng gầm 512 phần tử 8 byte, mỗi phần tử trỏ đến 1 PT. PT là một mảng gồm 512 phần tử 8 byte, mỗi phần tử chứa một PTE. Ví dụ địa chỉ ảo 0xBF80EE6B có thể được hiểu như trong hình  1-5

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

 Các phần tử 8 byte trong các bảng này chứa dữ liệu về bảng, quyền bộ nhớ, và những đặc điểm bộ nhớ khác. Ví dụ có các bit xác định xem trang ở chế độ chỉ đọc hay có thể đọc/ghi, thực thi hay không thực thi, người dùng có thể truy cập hay không. v.v

        Quá trình dịch địa chỉ xoay quanh 3 bảng này và Thanh ghi CR3. CR3 giữ địa chỉ cở sở vật lý của PDPT. Phần còn lại của phần này sẽ giới thiệu sơ lược về dịch địa chỉ ảo 0xBF80EE6B trên hệ thống thực( tham khảo hình 1-5)

kd> r @cr3     ; CR3 là địa chỉ vật lý cho cơ sở cảu PDPT  
cr3=085c01e0  
kd> !dq @cr3+2\*8 L1 ; đọc mục PDPT ở chỉ mục 2  
# 85c01f0 00000000`0d66e001

        Theo tài liệu, 12 bit dưới cùng của mục PDPT là flag/bit dành riêng, và những bit còn lại được sử dụng làm địa chỉ vật lý của cơ sở PD. Bit 63 là FLAG NX trong PAE, vì vậy bạn cũng sẽ cần phải xóa điều đó. Trong ví dụ cụ thể này, chúng tôi đã không xóa nó vì nó đã =0( CHúng tôi đang xem xét các trang mã có thể thực thi được ).

; 0x00000000`0d66e001 = **00001101 01100110 1110**0000 00000001  
; Sau khi clear 12 bits cuối, chúng ta được  
; 0x0d66e000 = 00001101 01100110 11100000 00000000  
; Điều này nói với chúng ta rằng cơ sở PD là ở địa chỉ vật lý 0x0d66e000

kd> !dq 0d66e000+0x1fc\*8 L1 ; Đọc mục PD ở chỉ mục 0x1FC  
# d66efe0 00000000`0964b063

        Một lần nữa, theo tài liệu, 12 bit dưới dùng của mục PD được sử dụng cho flag/bit dành riêng và những bit còn lại  được sử dụng làm cơ sở cho PT:

; 0x0964b063 = 00001001 01100100 10110000 01100011  
; Sau khi xóa 12 bit ở cuối, chúng ta được  
; 0x0964b000 = 00001001 01100100 10110000 00000000  
; Điều này cho chúng ta biết PT base ở 0x0964b000  
kd> !dq 0964b000+e\*8 L1 ; đọc mục PT ở chỉ mục 0xE  
# 964b070 00000000`06694021

        Một lần nữa , 12 bit cuối có thể được xóa để lấy cơ sở của mục page:

; 0x06694021 = 00000110 01101001 01000000 00100001  
;sau khi xóa 12 bit cuối ta được  
; 0x06694000 = 00000110 01101001 01000000 00000000  
; This tells us that the page entry base is at 0x06694000

( ĐIều đó cho chúng ta biết page entry base là ở 0x6694000)  
kd> !db 06694000+e6b L8 ; Đọc 8 byte từ page entry ở giá trị 0xE6B  
  
# 6694e6b 8b ff 55 8b ec 83 ec 0c ..U.....[).t.... ; dữ liệu của chúng                                                     ;ta ở trang vật lý  
  
kd> db bf80ee6b L8 ; Đọc 8 byte từ địa chỉ ảo  
bf80ee6b 8b ff 55 8b ec 83 ec ..U.....[).t.... ; same data!

    Sau toàn bộ quá trình, nó được xác định rằng địa chỉ ảo 0xBF80E6B chuyển thành địa chỉ vật lý 0x6694E6B.

     Các hệ điều hành điện đại thực hiện quy trình phân tách không gian địa chỉ bằng cơ chế này. Mọi quy trình được liên kết với một CR3 khác nhau, dẫn đến dịch địa chỉ ảo dành riêng cho quy trình. Điều kỳ diệu đằng sau  process's illusion đó là nó có không gian địa chỉ riêng. Hy vọng rằng bạn sẽ có nhiều đánh giá cao hơn với bộ xử lý trong lần tiếp theo chương trình của bạn truy cập bộ nhớ!

#### Interrupts and Exceptions(Ngắt và Ngoại lệ)

        Phần này sẽ bàn luận ngắn gọn về interrupts và exceptions, vì bạn có thể tìm thấy chi tiết triển khai đầy đủ trong chương 3"The Window Kernel"

         Trong  các hệ thống máy tính hiện đại, bộ xử lý thường được kết nối với các thiết bị ngoại vi thông qua một bus dữ liệu như PCI Express, Firewire, or USB. Khi thiết bị yêu cầu sự chú ý của bộ xử lý, nó sẽ gây ra gián đoạn(interrupt) buộc bộ xử lý phải tạm dừng bất cứ điều gì nó đang làm và xử lý yêu cầu của thiết bị. Làm thế nào để bộ xử lý biết cách xử lý yêu cầu? Ở mức cao nhất, người ta co thể nghĩ về một interrupt được kết hợp với một số sau đó được sử dụng để lập chỉ mục thành một mảng các con trỏ hàm. Khi bộ xử lý nhận 1 gián đoạn( interrupt), nó sẽ thực thi hàm ở trong chỉ mục kết hợp với interrupt và tiếp tục thực thi tại bất cứ đầu trước khi xảy ra ngắt quãng. Chúng được gọi là *hardware interrupts* bởi vì chúng được tạo bởi phần cứng. Chúng không đồng bộ về bản chất.

        Khi bộ xử lý đang thực hiện một lệnh, nó có thể gặp phải ngoại lệ( exceptions). Ví dụ, một lệnh có thể tạo ra lỗi chia cho 0, tham chiếu đến một địa chỉ không hợp lệ, hoặc kích hoạt chuyển đổi mức đặc quyền. Với mục đích của cuộc thảo luận này, các ngoại lệ(exceptions) có thể được chia thành 2 loại: lỗi và bẫy( faults and traps). Lỗi là một ngoại lệ có thể sửa chữa được.Ví dụ: khi bộ xử lý thực thi một lệnh tham chiếu đến một địa chỉ bộ nhớ không hợp lệ nhưng dữ liệu không có trong bộ nhớ chính( nó đã bị phân trang( paged out)), a page fault exception được tạo. Bộ xử lý xử lý điều này bằng cách lưu trạng thái thực thi hiện tại, gọi trình xử lý lỗi page để sửa lại ngoại lệ này ( bằng cách phân trang trong dữ liệu) và thực hiện lại cùng một lệnh( không gây lỗi trang). Một bẫy là một ngoại hệ do thực hiện các loại lệnh đặc biệt. Ví dụ: lệnh SYSENTER khiến bộ xử lý bắt đầu thực hiện trình xử lý system call; sau khi xử lý xong, thực thi được tiếp tục thoe lệnh ngay sau SYSENTER. Do đó sự khác biệt chính giữa lỗi và bẫy là nơi tiếp tục thực thi. Hệ điều hành thường triển khai system call thôi qua cơ chế interupt và exception

#### Walk-Through( Xem Qua)

        Chúng ta kết thúc chương này bằng cách xem qua một hàm có ít hơn 100 lệnh. Đó là thói quen của Sample J’s DllMain.Bài tập này có 2 mục tiêu. Đầu tiên, nó áp dụng hầu hết những nội dung có trong chương( ngoại trừ switchcase). Thứ 2, nó dạy một yêu cầu quan trọng trong thực hành kỹ thuật đảo ngược: đọc hướng dẫn ký thuật và tài liệu trực tuyến. Đây là hàm

01: ; BOOL \_\_stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason,  
; LPVOID lpvReserved)  
02: \_DllMain@12 proc near  
03: 55                  push ebp  
04: 8B EC mov           ebp, esp  
05: 81 EC 30 01 00+     sub esp, 130h  
06: 57                  push edi  
07: 0F 01 4D F8         sidt fword ptr [ebp-8]  
08: 8B 45 FA mov        eax, [ebp-6]  
09: 3D 00 F4 03 80      cmp eax, 8003F400h

10: 76 10               jbe short loc\_10001C88 (line 18)  
11: 3D 00 74 04 80      cmp eax, 80047400h  
12: 73 09               jnb short loc\_10001C88 (line 18)  
13: 33 C0               xor eax, eax  
14: 5F                  pop edi  
15: 8B E5               mov esp, ebp  
16: 5D                  pop ebp  
17: C2 0C 00            retn 0Ch  
18:     loc\_10001C88:  
19: 33 C0               xor eax, eax  
20: B9 49 00 00 00      mov ecx, 49h  
21: 8D BD D4 FE FF+     lea edi, [ebp-12Ch]  
22: C7 85 D0 FE FF+     mov dword ptr [ebp-130h], 0  
23: 50                  push eax  
24: 6A 02               push 2  
25: F3 AB               rep stosd  
26: E8 2D 2F 00 00      call CreateToolhelp32Snapshot  
27: 8B F8               mov edi, eax  
28: 83 FF FF            cmp edi, 0FFFFFFFFh  
29: 75 09               jnz short loc\_10001CB9 (line 35)  
30: 33 C0               xor eax, eax  
31: 5F                  pop edi  
32: 8B E5               mov esp, ebp  
33: 5D                  pop ebp  
34: C2 0C 00            retn 0Ch  
35:     loc\_10001CB9:  
36: 8D 85 D0 FE FF+     lea eax, [ebp-130h]  
37: 56                  push esi  
38: 50                  push eax  
39: 57                  push edi  
40: C7 85 D0 FE FF+     mov dword ptr [ebp-130h], 128h  
41: E8 FF 2E 00 00      call Process32First  
42: 85 C0               test eax, eax  
43: 74 4F               jz short loc\_10001D24 (line 70)  
44: 8B 35 C0 50 00+     mov esi, ds:\_stricmp  
45: 8D 8D F4 FE FF+     lea ecx, [ebp-10Ch]  
46: 68 50 7C 00 10      push 10007C50h  
47: 51                  push ecx  
48: FF D6               call esi ; \_stricmp  
49: 83 C4 08            add esp, 8  
50: 85 C0               test eax, eax  
51: 74 26               jz short loc\_10001D16 (line 66)  
52: loc\_10001CF0:  
53: 8D 95 D0            FE FF+ lea edx, [ebp-130h]  
54: 52                  push edx  
55: 57                  push edi  
56: E8 CD 2E 00 00      call Process32Next  
57: 85 C0               test eax, eax  
58: 74 23               jz short loc\_10001D24 (line 70)  
59: 8D 85 F4 FE FF+     lea eax, [ebp-10Ch]  
60: 68 50 7C 00 10      push 10007C50h  
61: 50                  push eax  
62: FF D6               call esi ; \_stricmp  
63: 83 C4 08            add esp, 8

64: 85 C0               test eax, eax  
65: 75 DA               jnz short loc\_10001CF0 (line 52)  
66:     loc\_10001D16:  
67: 8B 85 E8 FE FF+     mov eax, [ebp-118h]  
68: 8B 8D D8 FE FF+     mov ecx, [ebp-128h]  
69: EB 06 jmp short loc\_10001D2A (line 73)  
70:     loc\_10001D24:  
71: 8B 45 0C            mov eax, [ebp+0Ch]  
72: 8B 4D 0C            mov ecx, [ebp+0Ch]  
73:     loc\_10001D2A:  
74: 3B C1               cmp eax, ecx  
75: 5E                  pop esi  
76: 75 09               jnz short loc\_10001D38 (line 82)  
77: 33 C0               xor eax, eax  
78: 5F                  pop edi  
79: 8B E5               mov esp, ebp  
80: 5D                  pop ebp  
81: C2 0C 00            retn 0Ch  
82:     loc\_10001D38:  
83: 8B 45 0C            mov eax, [ebp+0Ch]  
84: 48                  dec eax  
85: 75 15               jnz short loc\_10001D53 (line 93)  
86: 6A 00               push 0  
87: 6A 00               push 0  
88: 6A 00               push 0  
89: 68 D0 32 00 10      push 100032D0h  
90: 6A 00               push 0  
91: 6A 00               push 0  
92: FF 15 20 50 00+     call ds:CreateThread  
93:     loc\_10001D53:  
94: B8 01 00 00 00      mov eax, 1  
95: 5F                  pop edi  
96: 8B E5               mov esp, ebp  
97: 5D                  pop ebp  
98: C2 0C 00            retn 0Ch  
99:                     \_DllMain@12 endp

        Dòng 3 dòng 4 thiết lập phần mở đầu hàm, phần mở đầu này sẽ lưu con trỏ khung cơ sở  trước đó và thiết lập một phần mở đầu mới. Dòng 5 dự trữ 0x130 bytes không gian stack. Dòng 6 lưu EDI. Dòng 7 thực thi lệnh SIDT, lệnh đó ghi thanh ghi IDT 6-byte vào một vùng bộ nhớ chỉ định. Dòng 8 đọc một giá trị double-word ở EBP-6 và lưu nó vào EAX. Dòng 9-10 kiểm tra nếu EAX bé hơn hoặc bằng 0x8003F400 không. Nếu đúng như vậy, việc thực thi chuyển sang dòng 18, nếu không tiếp tục ở dòng 11. Dòng 11-12 thực hiện kiểm tra tương tự ngoại trừ điều kiện là không thấp hơn 0x80047400. Nếu đúng, việc thực thi sẽ chuyển sang dòng 18, nếu không sẽ tiếp tục ở dòng 13. Dòng 13 xóa EAX. Dòng 14 khôi phục thanh ghi EDI ở dòng 6. Dòng 15-16 khôi phục khung cơ sở và con trỏ stack. Dòng 17 thêm 0xC bytes vào con trỏ stack và trả về caller.

        Trước khi bàn luận vầ đoạn tiếp theo, chú ý một vài điều về 17 dòng đầu tiên này. Lệnh SIDT( dòng 7) ghi nội dung của thanh ghi IDT vào 6 byte vị trí bộ nhớ. Thế thanh ghi IDT là gì? Sách hướng dẫn Intel/AMD refernce nói rằng IDT là một dãy gồm 256 mục 8 bytes, mỗi mục chứa một con trỏ trỏ tới interrupt handler, segment selector, và offset. Khi một interrupt hoặc exception xảy ra, bộ xử lý sử dụng interrupt number làm chỉ mục vào IDT và gọi trình xử lý được chỉ định của mục. Thanh ghi IDT là một thanh ghi 6-byte, 4 byte đầu bao gồm cơ sở của mảng/bảng IDT và 2 byte cuối chứa giới hạn của bảng. Với suy nghĩ này, bạn có thể hiểu rằng dòng 8 thực ra đang đọc địa chỉ cơ sở của IDT. Dòng 9 và 11 kiểm tra địa chỉ cơ sở có nằm trong dải (0x8003F400, 0x80047400) hay không. Điều gì đặt biệt về những hằng số dường như ngẫu nhiên này? Nếu bạn tìm kiếm trên internet, bạn sẽ thấy rằng 0x8003F400 là một địa chỉ cơ sở của IDT trên windows XP trên x86. Điều này được chứng minh trong debbuger kernel:

0: kd> vertarget  
Windows XP Kernel Version 2600 (Service Pack 3) MP (2 procs) Free x86 compatible  
Built by: 2600.xpsp.080413-2111  
…  
0: kd> r @idtr  
idtr=8003f400  
0: kd> ~1  
1: kd> r @idtr  
idtr=bab3c590

        Tại sao đoạn mã kiểm tra hành vi này? một cách có thể giải thích là nhà phát triển đã giả định rằng địa chỉ cơ sở IDT rơi vào khoảng  đó được coi là "không hợp lệ" hoặc có thể kết quả của việc bị ảo hóa. Hàm Tự động trả về 0 nếu IDTR là "invalid". Bạn có thể dịch ngược mã này thành C như sau

 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 > 0x8003F400 && idtr.base < 0x80047400h) { return FALSE; }  
//line 18  
...  
}

**Chú ý: Nếu bạn đọc hướng dẫn kỹ, bạn sẽ thấy rằng mỗi bộ xử lý có IDT riêng của nó và cũng như IDTR. Vì thế, trên một hệ thống đa lõi, IDTR sẽ khác nhau đối với mỗi lõi(core). Rõ ràng, 0x8003F400 chỉ hợp lý cho core 0 trên win XP. Nếu lệnh được định để chạy trên một lõi khác, IDTR sẽ là 0xBAB3C590. Trên các phiên bản window mới hơn, địa chỉ cơ sở IDT thay đổi giữa các lần khởi động lại; do đó, việc thực hành mã hóa cứng các địa chỉ cơ sở sẽ không hoạt động**

        Nếu cơ sở IDT có vẻ hợp lệ, chương trình sẽ tiếp tục thực hiện ở dòng 18.Dòng 19-20 clear EAX và đặt ECX thành 0x49. Dòng 21 sử dụng để đặt EDI thành bất kì EBP-0x12C là gì; bởi vì EBP là con trỏ khung cơ sở. EBP-12C là địa chỉ của biến cục bộ. Dòng 22 ghi 0 vào vị trí được EBP-130h chỉ đến. Dòng 23-24 đẩy EAX và 2 lên stack.Dòng 25 tạo bộ đệm 0x124-byte bắt đầu từ EBP-0x12C.Dòng 26 gọi CreateToolhelp32Snapshot:

HANDLE WINAPI CreateToolhelp32Snapshot(  
\_In\_ DWORD dwFlags,  
\_In\_ DWORD th32ProcessID  
);

       Hàm Win32 API này có 2 tham số integer. Nguyên tắc chung, Hàm Win32 API tuân theo STDCALL calling convention(tất cả mọi tham số của hàm phải đẩy vào stack trước khi gọi hàm). Do đó,tham số dwFlags và th32ProcessId là 0x2(dòng 24) và 0x0(dòng 23). Hàm này liệt kê tất cả tiến trình trên hệ thống và trả về một handle để sử dụng trong Process32Next. Dòng 27-28 lưu và trả về giá trị trong EDI và kiểm tra nó có bằng -1 k. Nếu có thì giá trị trả về được đặt là 0 và nó trả về( dòng 30-34); mặt khác, thực thi tiếp ở dòng 35. Dòng 36 đặt EAX thành địa chỉ cuả biến cục bộ được khởi tạo từ trước thành 0 ở dòng 22; dòng40 khởi tạo nó thành 0x128.Dòng 37.39 đẩy ESI,EAX, và EDI lên stack.Dòng 41 call Process32First:

    Function prototype(ngyên mẫu hàm)

BOOL WINAPI Process32First(  
\_In\_ HANDLE hSnapshot,  
\_Inout\_ LPPROCESSENTRY32 lppe  
);

Relevant structure definition( định nghĩa cấu trúc liên quan)  
typedef struct tagPROCESSENTRY32 {  
DWORD dwSize;  
DWORD cntUsage;  
DWORD th32ProcessID;  
ULONG\_PTR th32DefaultHeapID;  
DWORD th32ModuleID;  
DWORD cntThreads;  
DWORD th32ParentProcessID;  
LONG pcPriClassBase;  
DWORD dwFlags;  
TCHAR szExeFile[MAX\_PATH];  
} PROCESSENTRY32, \*PPROCESSENTRY32;  
00000000 PROCESSENTRY32 struc ; (sizeof=0x128)  
00000000 dwSize dd ?  
00000004 cntUsage dd ?  
00000008 th32ProcessID dd ?

0000000C th32DefaultHeapID dd ?  
00000010 th32ModuleID dd ?  
00000014 cntThreads dd ?  
00000018 th32ParentProcessID dd ?  
0000001C pcPriClassBase dd ?  
00000020 dwFlags dd ?  
00000024 szExeFile db 260 dup(?)  
00000128 PROCESSENTRY32 ends  
        Bởi về API này nhận 2 tham số, hSnapshot là EDI( dòng 39, trước đó được trả về từ CreateToolhelp32Snapshot in line 27), và lppe là địa chỉ của biến cục bộ(EBP-0x130). Bởi vì lppe trỏ tới cấu trúc PROCESSENTRY32, chúng ta ngay lập tức biết rằng biến cục bộ ở EBP-0x130 là cùng 1 loại. Nó cũng có ý nghĩa vì tài liệu cho Process32First nói rằng trước khi gọi hàm, trường dwSize phải đặt thành kích thước của cấy trúc PROCESSENTRY32( đó là 0x128). Bây giờ chúng ta biết rằng dòng 19-25 chỉ đơn giản là khởi tạo cấu trúc này 0. Ngoài ra, chúng ta có thể nói rằng biến cục bộ này bắt đầu ở EBP-130 và kết thúc ở EBP-0x8

        Dòng 42 kiểm tra giá trị trả về của Process32Next. Nếu là 0, thực thi dòng 70, ngược lại, tiếp tục ở dòng 43.Dòng 44 lưu địa chỉ của hàm stricmp trong ESI. Dòng 45 đặt ECX thành địa chỉ của biến cục bộ( EBP-0x10C), đây là trường hợp trong PROCESSENTRY32(xem đoạn trước). Dòng 46-48 push 0x1007C50/ECX trên stack và call stricmp. Bây giờ chúng ta biết rằng stricmp nhận 2 chuỗi kí tự làm đối số, do đó ECX phải là trường szExeFile trong PROCESSENTRY32 và 0x10007C50 là địa chỉ của 1 string.

         .data:10007C50 65 78 70 6C 6F+Str2 db 'explorer.exe',0

        Dòng 49 dọn sạch stack bởi vì stricmp sử dụng CDECL calling conventions. Dòng 50 check giá trị trả về của hàm stricmp. Nếu là 0, nghĩa là chuỗi khớp với "explorer.exe", quá trình thực hiện bắt đầu ở dòng 66, còn nếu  k thì nó sẽ tiếp tục ở dòng 52. Chúng ta cùng dịch ngược dòng 18-51 như sau

        HANDLE h;

PROCESSENTRY32 procentry;  
h = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS, 0);  
if (h == INVALID\_HANDLE\_VALUE) { return FALSE; }

memset(&procentry, 0, sizeof(PROCESSENTRY32));  
procentry.dwSize = sizeof(procentry); // 0x128  
if (Process32Next(h, &procentry) == FALSE) {  
// line 70  
...  
}  
if (stricmp(procentry.szExeFile, "explorer.exe") == 0) {  
// line 66  
...  
}  
// line 52  
        Dòng 52-65 gần giống khối trước đó ngoại trừ việc chúng tạo thành một vòng lặp với 2 điều kiện thoát. ĐIều kiện thoát đầu tiên là Process32Next trả về FALSE(dòng 58) và điều kiện thứ 2 là khi stricmp trả về 0. Chúng ta có thể dịch ngược dòng 52-65 như sau:

while (Process32Next(h, &procentry) != FALSE) {  
    if (stricmp(procentry.szExeFile, "explorer".exe") == 0)  
        break;  
}

        Sau khi thoát khỏi vòng lặp, thực hiệp tiếp ở dòng 66. Dòng 67-68 lưu th32ParentProcessID/th32ProcessID của PROCESSENTRY32 vào EAX/ECX và tiếp tục thực thi ở 37. Chú ý rằng Dòng 66 cũng là một mục tiêu nhảy của dòng 43.

        Dòng 70-74 đọc tham số fdwReason của DllMain(EBP+C) và kiểm tra có bằng 0 hay không(DLL\_PROCESS\_DETACH). Nếu có, giá trị trả về được đặt là 0 là nó return; ngược lại, nó đi đến dòng 82. Dòng 82-85 kiểm tra xem fdwReason có lớn hơn 1 không( DLL\_THREAD\_ATTACH, DLL\_THREAD\_DETACH). Nếu có giá trị trả về được đặt là 1 và return; nếu không, tiếp tục ở dòng 86. Dòng 86-92 gọi CreateThread:

        HANDLE WINAPI CreateThread(

\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpThreadAttributes,  
\_In\_ SIZE\_T dwStackSize,  
\_In\_ LPTHREAD\_START\_ROUTINE lpStartAddress,  
\_In\_opt\_ LPVOID lpParameter,  
\_In\_ DWORD dwCreationFlags,  
\_Out\_opt\_ LPDWORD lpThreadId  
);

với lpStartAddress là 0x100032D0. Khối này có thể dịch ngược như sao

if (fdwReason == DLL\_PROCESS\_DETACH) { return FALSE; }  
if (fdwReason == DLL\_THREAD\_ATTACH || fdwReason == DLL\_THREAD\_DETACH) {  
return TRUE; }  
CreateThread(0, 0, (LPTHREAD\_START\_ROUTINE) 0x100032D0, 0, 0, 0);  
return TRUE;

        Sau khi phân tích hàm, chúng ta có thể suy ra rằng ý định ban đầu của nhà phát triển là:

            1.Phát hiện xem máy mục tiêu có IDT "lành mạnh hay không"

            2.Kiểm tra xem"explorer.exe" có đang chạy trên hệ thống không-tức là ai đó đã đăng nhập

            3.Tạo 1 luồng chính lây nhiễm máy mục tiêu.

#### Bài tập

            1. Tự mình đọc lại hàm. Vẽ bố cục ngăn xếp bao gồm các tham số và biến cục bộ

            2. Trong phầm hướng dẫn ví dụ, chúng tôi đã thực hiện dịch gần như 1-1 của mã assembly sang C.Như một bài tập, hãy dịch ngược lại toàn bộ hàm này để nó trông tự nhiên hơn. Bạn có thể nói gì về cấp độ/kỹ năng của nhà phát triển? Giải thích lý do của bạn.Bạn có thể làm tốt hơn?

            3. Trong một số danh sách assembly, tên hàm có tiền tố @ theo sau là 1 số. Giải thích khi nào và tại sao nó lại như thế

            4.Triển khai các hàm sau trong assembly x86: strlen,strchr, memcpy, memset, strcmp, strset

            5.Giải mã các quy trình kernel trong Windows sau:

■ KeInitializeDpc

■ KeInitializeApc

■ ObFastDereferenceObject (and explain its calling convention)

■ KeInitializeQueue

■ KxWaitForLockChainValid

■ KeReadyThread

■ KiInitializeTSS

■ RtlValidateUnicodeString

            6. SampleH. Hàm sub\_13846 tham chiếu đến 1 số cấu trúc có kiểu không hoàn toàn rõ ràng. Nhiệm vụ của bạn trước tiên là khôi phục lại nguyên mẫu hàm và sau đó cố gắng tạo lại các trường cấu trúc. Sau khi đọc xong chương 3, hãy qua lại bài tập này để xem hiểu biết của bạn có thay đổi không.(Lưu ý: Mẫu này đang nhắm mục tiêu WIndows XP x86).

            7. Sample H. Hàm sub\_10BB6 có một vòng lặp tìm kiếm thứ gì đó. Đầu tiên khôi phục lại nguyên mẫu hàm và sau đó suy ra các kiểu dựa trên ngữ cảnh. Gợi ý: bạn có thể nen có một bản sao đặt ra  PE gần đó.

            8.SampleH. Giải mã sub\_11732 và giải thích cấu trúc lập trình giống nhất sử dụng trong mã gốc

            9.SampleH. GIải thích hàm sub\_1000CEA0 làm gì và sau đó dịch ngược nó trở lại C.

            10.Nếu mức đặc quyền hiện tại được mã hóa trong CS, có thể sử đổi mã chế độ người dùng, tại sao mã chế độ người dùng không thể sửa đổi CS để thay đổi CPL(current privilege level) ?

            11.Đọc chương Bộ nhớ ảo trong sách Intel Software Developer Manual, Volume 3 and AMD64 Architecture Programmer’s Manual, Volume 2: System Programming. Tự thực hiện một vài địa chỉ ảo sang bản dịch địa chỉ vật lý và xác minh kết quả bằng trình gỡ lỗi kernel. Giải thích như thế nào ngăn chặn thực thi dữ liệu( data execution prevention(DEP)) hoạt động.

            12. Thư viện giải mã x86/x64 yêu thích của Bruce là BeaEngine của BeatriX ( www.beaengine.org). Thử nghiệm nó bằng cách viết 1 chương trình để tháo rời 1 nhị phân tại điểm vào của nó.

Trả lời

1.2.[https://bin.re/blog/practical-reverse-engineering-solutions-page-35-part-i/](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

3. Tên có tiền tố \_ và tiền tố @h cho biết quy ước gọi \_stdcall. Nó mặc định cho dll's của Windows. Sau khi call phải dọn dẹp ngăn xếp. Số n trong hậu tố cho biết có bao nhiêu byte được sử dụng cho các tham số hàm. \_DllMain@12 do đó sử dụng 12 Byte làm tham số, tức là một byte cho mỗi tham số trong 3 tham số

4. [strlen.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417), [strchr.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417), [memcpy.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417), [memset.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417), [strcmp.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417), [strset.asm](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

5.[https://zerosum0x0.blogspot.com/2015/01/practical-reverse-engineering-p35-5.html](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

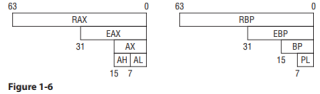
6.[https://bin.re/blog/practical-reverse-engineering-solutions-page-35-part-iii/](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

#### x64

        x64 là một phần mở rộng của x86, vì thế hầu hết các thuộc tính kiến trúc là giống nhau, với những khác biệt nhỏ giống như kích cỡ thanh ghi và một số lệnh không khả dụng ( như PUSHAD). Phần tiếp theo sẽ thảo luận về sự khác biệt có liên quan.

#### Tập Thanh ghi và các loại dữ liệu

        Bộ thanh ghi có 18 GPRs(generalpurpose registers) 64bit, và có thể minh hoạt như trong hình 1-6. Chú ý rằng thanh ghi 64-bit có tiền tố R ở đầu:

[](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

        Mặc dù RBP có thể được sử dụng như khung con trỏ cơ sở, nhưng nó hiếm khi được sử dụng cho mục đích đó trong mã do trình biên dịch tạo ra trong thực tế. Hầu hết trình biên dịch x64 chỉ đơn giản xử lý RBP như một GPR khác và tham chiếu các biến cục bộ liên quan đến RSP.

#### Di chuyển dữ liệu

        x64 hỗ trợ một khái niệm được gọi là RIP-Relative addressing(định địa chỉ tương đối), cho phép các lệnh tham chiếu dữ liệu ở một vị trí tương đối với RIP. Ví dụ

01: 0000000000000000 48 8B 05 00 00+ mov rax, qword ptr cs:loc\_A  
02:                                                                                                 ; originally written as "mov rax, [rip]"  
03: 0000000000000007                 loc\_A:  
04: 0000000000000007 48 31 C0        xor rax, rax  
05: 000000000000000A 90              nop

        Dòng 1 đọc địa chỉ của loc\_A(0x7) và lưu nó vào RAX. RIP -Định địa chỉ tương đối chủ yếu được sử dụng để tạo điều kiện cho mã không phụ thuộc vào vị trí.

        Hầu hết lệnh số học được tự đuunjg thăng cấp lên 64bits ngay cả khi các toán hạng chỉ là 32bits. Ví dụ:

48 B8 88 77 66+             mov rax, 1122334455667788h  
31 C0                       xor eax, eax ; will also clear the upper 32bits of RAX.  
                                         ; i.e., RAX=0 after this  
48 C7 C0 FF FF+             mov rax,0FFFFFFFFFFFFFFFFh  
FF C0                       inc eax ; RAX=0 after this

#### Địa chỉ chuẩn(Canonical Address)

        Trên x64, địa chỉ ảo có độ dài 64bits, nhưng hầu hết bộ xử lý không hỗ không gian  64bit địa chỉ ảo đầy đủ. Bộ xử lý Intel/AMD hiện tại chỉ sử dụng 48 bits cho không gian địa chỉ. Tất cả địa chỉ bộ nhớ ảo phải ở dạng chuẩn. Địa chỉ ảo ở dạng chuẩn nếu các các bít từ 63 đến bit được triển khai quan trọng nhất đều là 1s hoặc 0s. Về mặt thực tế, nó có khi là bits 48-63 cần phải khớp với bit 47. Ví dụ

Canonical Address

0xfffff801`c9c11000 = **11111111 11111111** 11111000 00000001 11001001 11000001 00010000 00000000 ; canonical  
0x000007f7`bdb67000 = **00000000 00000000** 00000111 11110111 10111101 10110110 01110000 00000000 ; canonical  
0xffff0800`00000000 = **11111111 11111111** 00001000 00000000 00000000 00000000 00000000 00000000 ; non-canonical  
0xffff8000`00000000 = **11111111 11111111** 10000000 00000000 00000000 00000000 00000000 00000000 ; canonical  
0xfffff960`000989f0 = **11111111 11111111** 11111001 01100000 00000000 00001001 10001001 11110000 ; canonical  
        Nếu code cố gắng bỏ qua một địa chỉ không hợp lệ, hệ thống sẽ gây ra một ngoại lệ.

#### Lời gọi hàm(Function Invocation)

        Nhớ lại một số quy ước gọi ( calling convention) yêu cầu các tham số để truyền vào ngăn xếp trong x86. Trên x64, hầu hết các quy ước gọi đều truyền các tham số thông qua các thanh ghi. Ví dụ, trên Window x64, chỉ có một số quy ước gọi và bốn tham số được truyền qua RCX, RDX, R8 và R9; phần còn lại được đẩy sang stack từ phải sang trái. Trên Linux, sau tham số đầu tiên được chuyển trên RDI,RSI, RDX, RCX, R8 và R9.

#### Chú ý: Để biết thêm thông tin về x64 ABI trên Windows, xem "x64 Software Conventions” trên MSDN [http://msdn.microsoft.com/en-us /library/7kcdt6fy.aspx](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

#### Bài Tập

1. Giải thích 2 phương pháp để lấy con trỏ lệnh trên x64. Ít nhất một trong số các phương pháp phải sử dụng địa chỉ RIP

2.Thực hiện dịch địa chỉ ảo sang địa chỉ thực trên x64. Có khác biệt gì lớn so với x86?

1[https://zerosum0x0.blogspot.com/2015/01/practical-reverse-engineering-p-38-1.html](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

2.[https://zerosum0x0.blogspot.com/2015/01/practical-reverse-engineering-p-38-2.html](https://www.blogger.com/blog/post/edit/6725528767662423643/8147565583797245417)

### Chapter 2: ARM

        Một công ty tên là Acorn Computers đã phát triển kiến trúc RISC 32-bit được đặt tên là Acorn RISC Machine (sau đó đổi tên thành Advanced RISC Machine) vào cuối những năm 1980. Kiến trúc này tỏ ra hữu ích ngoài dòng sản phẩm hạn chế của họ, vì vậy một công ty đặt tên là ARM Holdings đã được thành lập để cấp phép cho kiến trúc này được sử dụng trong nhiều loại sản phẩm. Nó thường được thấy trong các thiết bị nhúng như điện thoại di động, thiết bị điện tử ô tô, máy nghe nhạc MP3, TV, v.v. Phiên bản đầu tiên của kiến trúc được giới thiệu vào năm 1985, và tại thời điểm viết bài này, nó đang ở phiên bản 7 (ARMv7). ARM đã phát triển một số lõi cụ thể( ARM7, ARM7TDMI, ARM926EJS, Cortex) - không nên nhầm lẫn với các thông số kĩ thuật kiến trúc khác nhau, được đánh số là ARMv1-ARMv7. Mặc dù có một số phiên bản, hầu hết các thiết bị đều sử dụng ARMv4,5,6,or7. ARMc4 và c5 tương đối cũ, nhưng chúng cũng là phiên bản phổ biến và chiếm ưu thế nhất của bộ vi xử lý( hơn 10 tỷ lõi đang tồn tại, theo tiếp thị của ARM). Các sản phẩm điện tử tiêu dùng phổ biến thường sử dụng các phiên bản kiến trúc mới hơn. Ví dụ Apple iPod Touch và Iphone thế hệ thứ 3 đang chạy trên chip ARMv6 và các thiết bị Iphone/IPad và Window Phone 7 sau này đều chạy trên ARMv7. Trong khi các công ty như Intel và AMD thiết kế và sản xuất bộ vi xử lý của họ, ARM lại đi theo một mô hình hơi khác. ARM thiết kế kiến trúc và cấp phép cho các công ty khác, sau đó sản xuất và tích hợp bộ vi xử lý vào thiết bị của họ. Các công ty như Apple, NVIDIA, Qualcomm, và Texas Instruments tiếp thị bộ vi xử lý của riêng họ(A,Tegra, Snapdragon and OMAP, tương ứng), nhưng kiến trúc cốt lõi của chúng được cấp phép từ ARM. Tất cả chung đều triển khai tập lệnh cơ sở và mô hình bộ nhớ được định nghĩa trong sổ thay tham chiếu kiến trúc ARM(ARM architecture reference manual). Phần mở rộng có thể được thêm vào bộ xử lý; ví dụ, phần mở rộng Jazelle cho phép mã bytecode của Java được thực thi nguyên bản trên bộ xử lý. Phần mở rộng Thumb bổ sung các lệnh có thể rộng 16 hoặc 32 bit, do đó cho phép mật độ mã cao hơn(các lệnh ARM gốc luôn có chiều rộng 32 bit). Phần mở rộng Debug cho phép kĩ sư phân tích bộ xử lý vật lý sử dụng debugging hardware đặc biệt. Mỗi phần mở rộng thường được biểu thị bằng 1 chữ cái (J,T,D,v,v). Phụ thuộc vào yêu cầu của họ, nhà sản xuất có thể quyết định xem họ cần cấp phép cho các phầm mở rộng bổ sung này không. Đây là lý do tại sao ARMv6 trở về trước có các chữ cái đằng sau chúng ví dụ: ARM1156T2 nghĩa là ARMv6 với phần mở rộng Thumb-2). Các quy ước này không còn được sử dụng trong ARMv7, thay vào đó sử dụng 3 cấu hình(Application, Real-time, and Microcontroller) và tên mô hình( Cortex) với tính năng khác nhau. Ví dụ dòng ARMv7 Cortex-A là bộ xử lý có cấu hình ứng dụng; và Cortex-M dành cho vi điều khiển và chỉ hỗ trợ thực thi chế độ Thumb

        Chương này bao gồm kiến trúc ARMv7 như được định nghĩa trong*ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition(ARM DDI 0406B)*

#### Tính Năng Cơ Bản(Basic Features)

        Bởi vì ARM là một kiến trúc RISC, có một số khác biệt cơ bản giữa kiến trúc ARM và CISC(x86/64). (Từ góc độ thực tế, các phiên bản mới của bộ xử lý Intel cũng có một số tính năng RISC - tức là họ không thực sự là CISC “thuần túy”).Đầu tiên, tập lệnh ARM rất nhỏ so mới x86, nhưng nó cung cấp nhiều thanh ghi có mục đích chung hơn. Thứ 2, Chiều dài của lệnh là chiều dài cố định( 16 bits hoặc 32 bits, phụ thuộc vào trạng thái). Thứ 3, ARM sử dụng mô hình kho tải để truy cập bộ nhớ. Điều đó có nghĩa là dữ liệu phải được chuyển từ bộ nhớ vào thanh ghi trước khi được vận hành, và chỉ các lệnh tải/lưu có thể truy cập bộ nhớ. Trong ARM, điều này chuyển thành các lệnh LDR và STR. Nếu bạn muốn tăng một giá trị 32-bit tại một địa chỉ bộ nhớ cụ thể, bản đầu tiên phải tải giá trị ở địa chỉ đó vào một thanh ghi, tăng nó lên và lưu nó lại. Ngược lại với x86, nó cho phép hầu hết các lệnh hoạt động trực tiếp trên dữ liệu trong bộ nhớ, một hoạt động đơn giản như vậy trên ARM sẽ yêu cầu 3 câu lệnh(một lần tải, một lần tăng, một lần lưu trữ). Điều này có thể ngụ ý răng có nhiều mã hơn để đọc cho thiết kế dịch ngược, nhưng thực tế, nó không thực sự quan trọng lắm khi bạn đã quen với nó.

ARM cũng cung cấp một số mức đặc quyền khác nhau để thực hiện cách ly đặc quyền. Trong x86, Các đặc quyền được xác định bởi 4 vòng, với vòng 0 có đặc quyền cao nhất và vòng 3 có mức đặt quyền thấp nhất. Trong ARM, các đặc quyền được xác định bởi 8 chế độ khác nhau:

■ User (USR)  
■ Fast interrupt request (FIQ)  
■ Interrupt request (IRQ)  
■ Supervisor (SVC)  
■ Monitor (MON)  
■ Abort (ABT)  
■ Undefined (UND)  
■ System (SYS)

Mã chạy trong một chế độ nhất định có quyền truy cập vào các đặc quyền và thanh ghi nhất định mà những cái khác có thể không; ví dụ Mã chạy ở chế độ USR không được phép chỉnh sử thanh ghi hệ thống( thường chỉ được sử đổi trong chế độ SVC). USR là chế độ ít đặc quyền nhất. Mặc dù có nhiều khác biệt về mặc kỹ thuật, nhưng để đơn giản hơn, bạn có thể ví con rằng USR giống như vòng 3 và SVC giống như vòng 0. Hầu hết các hệ điều hành thực hiện chế độ kernel mode trong SVC và chế độ người dùng USR. Cả Windows và Linux đều làm điều này.

Nếu bạn nhớ lại từ Chương 1, bộ xử lý x64 có thể thực thi 32-bit, 64-bit hoặc cả 2 thay thế cho nhau. Bộ xử lý ARM tương tự ở chỗ chúng cũng có thể hoạt động ở 2 trạng thái ARM và Thumb. TRạng thái ARM/Thumb chỉ xác định taapj lệnh, không xác định mức đặc quyền. Ví dụ, code chạy ở SVC mod có thể ARM hoặc Thumb. Trong Trạng thái ARM, lệnh luôn có độ dài 32bit; ở trạng tháo Thumb, lệnh có thể là 16bits hoặc 32bits. Bộ xử lý thực ở trạng thái nào phụ thuộc vào 2 điều kiện:

■ Khi Phân nhánh với lệnh BX và BLX, nếu bit quan trọng nhất của thanh ghi đích là 1, thì nó sẽ chuyển sang trạng thái Thumb.(mặc dù các lệnh được căn chỉnh 2 hoặc 4 byte, nhưng bộ xử lý sẽ bỏ qua bit quan trọng nhất nên sẽ không có vấn đề về căn chỉnh.)

■Nếu bit T trong thanh ghi trạng thái chương trình hiện tại( current program status register (CPSR) ) được đặt, thì nó đang ở chế độ Thumb. Ngữ nghĩa của CPSR được giải thích trong phần sau, nhưng bây giờ bạn có thể coi nó như một thanh ghi EFLAGS mở rộng trong x86

Khi lõi ARM khởi động, phần lớn thời gian nó chuyển sang trạng thái ARM và giữ nguyên như vậy cho đến khi có sự thay đổi rõ ràng hoặc ngầm định đối với Thumb. Trong thực tế, rất nhiều mã hệ điều hành gần đây chủ yếu sử dụng mã Thumb bởi vì muốn mật độ mã cao hơn( Hỗn hợp các lệnh rộng 16/32bit có thể kích thước nhỏ hơn tất cả các lệnh 32 bits); ứng dụng có thể hoạt động ở bất kỳ chế độ nào họ muốn.

Mặc dù hầu hết lệnh Thumb và ARM đều có cùng cách ghi nhớ, nhưng các lệnh Thumb 32 bit có hậu tố .w

**CHÚ Ý: Nó là một quan niệm sai lầm khi nghĩa rằng Thumb giống như chế độ thực(real mode) và ARM giống như chế độ bảo vệ (protected mode) trên x86/x64. Đừng nghĩ về nó theo cách này. Hầu hết các hệ điều hành trên nền tảng x86/x64 chạy ở chế độ bảo vệ và hiếm khi chuyển về chế độ thực. Hệ điều hành và ứng dụng trên nền tảng ARMcó thể thực thi cả ở trạng thái ARM và Thumb thay thế cho nhau. Cũng lưu ý rằng những trạng thái này hoàn toàn khác với các chế độ đặc quyền được giải thích trong đoạn trước ( USR, SVC, etc…)**

**Có 2 phiên bản của Thumb: Thumb-1 và Thumb-2. Thumb-1 được sử dụng trong kiến trúc ARMv6 trở về trước, và lệnh của nó luôn luôn có độ dài 16 bit. Thumb-2 mở rộng điều đó bằng cách thêm nhiều lệnh hơn và cho phép chúng có chiều rộng là 16 bit hoặc 32 bit. ARMv7 yêu cầu Thumb-2, vì vậy bất cứ khi nào chúng ta nói về Thumb, chúng ta đang đề cập đến Thumb-2**

**Có một số khác biệt giữa ARM và Thumb nhưng chúng ta không thể đề cập hết ở đây. Ví dụ, một vài câu lệnh có thể sẵn ở trạng thái ARM và ngược lại. Bạn có thể tham khảo tài liệu chính thức của ARM để biết thêm chi tiết.**

Ngoài việc có nhiều trạng thái thực thi khác nhau, ARM còn hỗ trợ thực thi có điều kiện. Điều này có nghĩa là một lệnh mã hóa các điều kiện số học nhất định phải được đáp ứng để nó được thực thi. Ví dụ, một lệnh có thể chỉ định rằng nó sẽ chỉ được thực thi nếu kết quả của lệnh trước đó bằng 0. Ngược lại điều này với x86, hầu hết mọi lệnh đều được thi thi vô điều kiện.( Intel có một số lệnh trực tiếp hỗ trợ thực thi có điều kiện CMOV và SETNE.) Thực thi có điều kiện rất hữu dụng vì nó cắt giảm các lệnh rẽ nhanh( rất tốn kém) và giảm số lượng lệnh được thi thi( dẫn đến mật đỗ mã cao hơn). Tất cả các lệnh trong trạng thái ARM hỗ trợ thực thi có điều kiện, nhưng mặc định chúng thực thi vô điều kiện, Trong trạng thái Thumb, cần có một lệnh IT đặt biệt để cho phép thực thi có điều kiện.

Một tính năng độc đáo khác của ARM là bộ chuyển đổi thùng ( barrel shifter). Một số lệnh nhất định có thể “ chứa” một lệnh số học khác có thể thay đổi hoặc xoay một thanh ghi. Điều này rất hữu ích bởi vì nó có thể thu nhỏ nhiều lệnh thành một; ví dụ bạn muốn nhân một thanh ghi với 2 và sau đó lưu kết quả vào một thanh ghi khác. Thông thường nó sẽ yêu cầu 2 lệnh( nhân sau đó là di chuyển), nhưng với bộ dịch chuyển bạn có thể bao gồm nhân( dịch trái với 1) bên trong lệnh MOV. Lệnh sẽ giống như sau:

MOV R1, R0, LSL #1 ; R1 = R0 \* 2

**Kiểu Dữ Liệu Và Thanh Ghi ( Data Types and Registers)**

Giống như ngôn ngữ bậc cao, ARM hỗ trợ các hoạt động trên các kiểu dữ liệu khác nhau. Các kiểu dữ liệu được hỗ trợ là 8bit( byte), 16bit( half-word), 32-bit(word) và 64bit(double-word).

Kiến trúc ARM định nghĩa 16 thanh ghi mục đích chung 32 bit, được đánh số từ R0,R1,…R15. Trong khi tất cả chúng đều có sẵn cho lập trình viên, Trên thực tế, 12 thanh ghi đầu tiên dành cho mục đích sử dụng chung ( chặng hạn như EAX, EBX,.. trong x86) và 3 thanh ghi cuối cùng có ý nghĩa đặc biệt tọng kiến trúc

■ R13 được kí hiệu là con trỏ ngăn xếp (SP). Nó tương đương với ESP/RSP trong x86/64. Nó trỏ đến đầu ngăn xếp chương trình.

■ R14 được kí hiệu như là một thanh ghi liên kết. Nó thường giữ địa chỉ trả về trong khi gọi hàm. Một số lệnh nhất định ngầm sử dụng thanh ghi này. Ví dụ, BL luôn lưu trữ địa chỉ trả về trong LR trước khi phân nhánh về đích. X86/x64 không có thanh ghi tương đương vì nó lưu giá trị trả về trên ngăn xếp. Trong mã không sử dụng LR để lưu trữ địa chỉ trả về, nó có thể được sử dụng như một thanh ghi có mục đích chung.

■ R15 được kí hiệu là một bộ đếm chương trình (Program counter PC). Khi thực thi ở trạng thái ARM, PC là một địa chỉ có lệnh hiện tại cộng với 8 ( 2 lệnh ARM phía trước); ở trạng thái Thumb, nó là địa chỉ của lệnh hiện tại cộng với 4( 2 lệnh Thumb 16 bit phía trước). Nó tương đương với EIP/RIP trong x86/64 ngoại trừ việc chúng luôn trỏ đến địa chỉ của lệnh tiếp theo sẽ được thực thi. Một điểm khác biệt lớn nữa là mã có thể đọc và ghi trực tiếp vào thanh ghi PC. Việc ghi địa chỉ vào PC sẽ ngay lập tức khi qua trình thực thi bắt đầu tại đại chỉ đó. Điều này có thể giải thích kĩ hơn 1 chút để tránh nhầm lẫn. Hãy xem đoạn mã sau ở trạng thái Thumb:

1: 0x00008344 push {lr}  
2: 0x00008346 mov r0, pc  
3: 0x00008348 mov.w r2, r1, lsl #31  
4: 0x0000834c pop {pc}

Sau khi dòng 2 được thực thi, R0 sẽ lưu giá trị 0x0000834a (=0x00008346+4):

(gdb) br main  
Breakpoint 1 at 0x8348  
...  
Breakpoint 1, 0x00008348 in main ()  
(gdb) disas main  
Dump of assembler code for function main:  
0x00008344 <+0>: push {lr}  
0x00008346 <+2>: mov r0, pc  
=> 0x00008348 <+4>: mov.w r2, r1, lsl #31  
0x0000834c <+8>: pop {pc}  
0x0000834e <+10>: lsls r0, r0, #0  
End of assembler dump.  
(gdb) info register pc  
pc 0x8348 0x8348 <main+4>  
(gdb) info register r0  
r0 0x834a 33610

Ở đây chúng ta đặt breakpoint ở 0x00008348. Khi nó chạm vào, chúng tôi hiển thi thanh ghi PC và R0; như được hiển thị thanh ghi PC trỏ đến lệnh thứ 3 tại 0x00008348( sắp được thực thi) và R0 hiển thị giá trị PC đã đọc trước khó. Từ ví dụ này bạn có thể thấy rằng khi đọc trực tiếp PC, nó tuân theo định nghĩa, nhưng khi gỡ lỗi, PC trỏ đến lệnh sẽ được thực thi

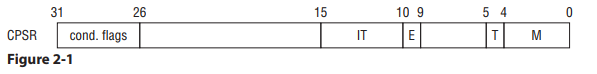
Lý do cho sự khác biệt này là do kết nối kế thừa từ các bộ xử lý ARM cũ hơn, luôn tìm nạp 2 lệnh trước lệnh hiện đang thực thi. Ngày nay, các pipelines phức tạp hơn nhiều nên điều này không thực sự quan trọng lắm, nhưng ARM vẫn giữ nguyên định nghĩa này để đảm bảo khả năng tương thích với các bộ vi xử lý trước đó.

Giống như các kiến trúc khác, ARM lưu thông tin về trạng thái thực thi hiện tại trong thanh ghi trạng thái chương trình hiện tại ( current program status register (CPSR)). Từ quan điểm của lập trình viên ứng dụng, CPSR tương dừng với EFLAGS/RFLAGS trong x86/x64. Một số tài liệu có thể thảo luận về Thanh ghi trạng thái chương trình ứng dụng (application program status register (APSR)), là bí danh cho một số trường nhất định trong CPSR. Có nhiều cờ trong CPSR, một số trong đó được minh họa trong hình 2-1( các cờ khác được đề cập trong các phần sau).

■ E (Endianness bit)— ARM có thể hoạt động ở chế độ big or little endian.Bit này được đặt tương ứng thành 0 hoặc 1 cho little endian hoặc big endian. Hầu hết, little endian được sử dụng, nên bit này sẽ là 0

■ T (Thumb bit)— Bit này được đặt nếu bạn ở trong trạng thái Thumb; mặt khác, nếu nó là trạng thái ARM.Một cách để chuyển dổi rõ ràng từ Thumb thành ARM( và ngược lại) là điều chỉnh bit này

■ M(Mode bits)—Những bit này chỉ chế độ đặc quyền hiện tại( USR,SVC,...)



(45/73/383)