看雪逆向-Security Documentation

原创

amingMM 已于 2022-04-07 10:28:06 修改 506 收藏

分类专栏： 渗透测试 # 逆向分析 文章标签： cpu 逆向

于 2022-01-07 19:54:12 首次发布

本文链接：https://blog.csdn.net/qq_33608000/article/details/122370389

渗透测试 同时被 2 个专栏收录

74 篇文章 15 订阅

订阅专栏

逆向分析

18 篇文章 0 订阅

订阅专栏

看雪逆向-Security Documentation

push ebp // 实现压入操作的指令

POP //实现弹出操作的指令

// ESP是堆栈指针 总是指向栈顶位置。一般堆栈的栈底不能动 无法暂借使用

MOV指令 //数据传送指令

push -0x1

F12暂停法 Alt+K

F8单步

fstsw 可以把状态寄存器读取到一个双字节内存位置或者AX寄存器中

fstcw 指令获取当前控制寄存器的值

fldcw 指令把 值加载到控制寄存器

fstcw 指令检查当前控制寄存器的值

ALT+M 内存镜像

ALT+E 调用了那些系统模块 该模块的存放地址和文件名

程序领空 OD载入程序 程序本身的代码

系统领空 USE32 程序调用的咱们系统的一个函数模块

AX、BX、CX、DX 用于存放数据 数据寄存器

EIP CPU下次要执行的指令的地址

EBP 栈的栈底指针 栈基址 （ESP传递给）

ESP 栈的栈顶。并且始终指向栈顶

一 、基础知识

1.1 cpu体系架构

1.1-1 RISC和CISC

RISC(精简指令集计算机)Reduced Instruction Set Computer
CISC(复杂指令集计算机)Complex Instruction Set Computer
当前CPU的两种架构。

区别在于不同的CPU设计理念和方法。

早期的CPU全部是CISC架构，
它的设计目的是要用最少的机器语言指令来完成所需的计算任务。

比如对于乘法运算，
在CISC架构的CPU上，
您可能需要这样一条指令：MUL ADDRA, ADDRB
就可以将ADDRA和ADDRB中的数相乘并将结果储存在ADDRA中。

将ADDRA, ADDRB中的数据读入寄存器，相乘和将结果写回内存的操作
全部依赖于CPU中设计的逻辑来实现。

这种架构会增加CPU结构的复杂性和对CPU工艺的要求，但对于编译器的开发十分有利。
比如上面的例子，C程序中的a*=b就可以直接编译为一条乘法指令。
今天只有Intel及其兼容CPU还在使用CISC架构。

RISC架构要求软件来指定各个操作步骤。
上面的例子如果要在RISC架构上实现，

将ADDRA, ADDRB中的数据读入寄存器，相乘和将结果写回内存的操作
都必须由软件来实现，

比如：

```
MOV A, ADDRA;
MOV B, ADDRB;

MUL A, B;

STR ADDRA, A。
```

这种架构可以降低CPU的复杂性
以及允许在同样的工艺水平下生产出功能更强大的CPU，
但对于编译器的设计有更高的要求。

1.1-2 CPU工作的基本原理

- 如何设计一个简单的16位CPU模型
  中央处理器（central processing unit，简称CPU）

计算机的组成结构
数字电路基础
编程的基础

实现 一个RISC指令集的CPU
要自己为这个CPU设计指令并且编码

英特尔的 叫特德·霍夫（Ted Hoff）的工程师
设计和生产专用集成电路芯片
用于实现桌面计算器
各实现一种特定的功能
想法：
为什么不能用一块通用的芯片加上程序来实现几块芯片的功能呢？
当需要某种功能时，只需要把实现该功能的一段程序代码（称为子程序）加载到通用芯片上，其功能与专用芯片会完全一样。

计算器的新的体系结构图，
其中包含4块芯片：
一块通用处理器芯片，实现所有的计算和控制功能；
一块可读写内存（RAM）芯片，用来存放数据；
一块只读内存（ROM）芯片，用来存放程序；
一块输入输出芯片，实现键入数据和操作命令、打印结果等等功能

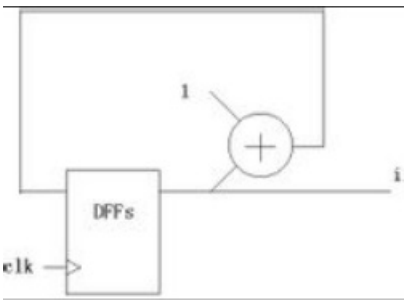
例1-1:

```
mov eax,0
repeat:inc eax
jmp repeat
```

例1-2:

```
 int main()
{
unsigned int i = 0;
while(1)
i++;
}
```

例1-3:



以上三个例子都产生了一个从0不断增加的序列
而且前两个例子会一直加到溢出又从0开始
（这个取决于计算机的字长也就是多少位的CPU，
eax是32位寄存器所以必然是加到4294967295然后回0，
而后面那个c程序则看不同编译器和不同平台不一样）

后面那个例子则看你用的是什么样的加法器和多少个D触发器

假设要一个递减的序列

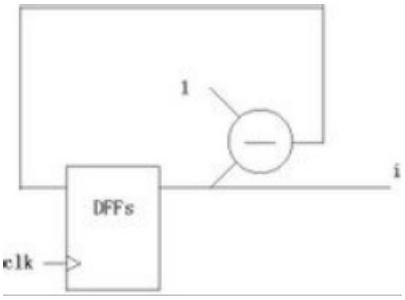
例2-1:

```
 mov eax,0
repeat:dec eax
jmp repeat
```

例2-2:

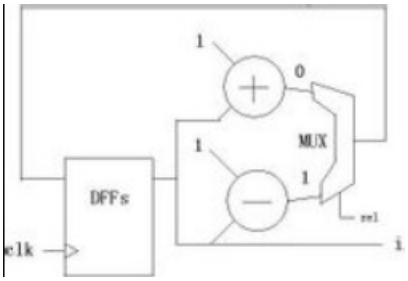
```
 int main()
{
unsigned int i = 0;
while(1)
i--;
}
```

例2-3:



上面
（例1-3）中是个加法器，
减法器（例2-3）！

例3：



使用了一个加法器一个减法器，没比上面的电路省
（加上一个负数的补码确实就是减去一个数）

多了一组多路器，少了一组D触发器
而sel信号就是用来选择的（0是递增，1是递减）。

push ebp // 实现压入操作的指令

POP //实现弹出操作的指令

//缺省对堆栈操作的寄存器 ESP 和 EBP 扩展基址指针寄存器(extended base pointer)

// ESP是堆栈指针 总是指向栈顶位置。一般堆栈的栈底不能动 无法暂借使用

一般使用 EBP 来存取堆栈

调用中有两个参数

在 push 第一个参数前的堆栈指针 ESP 为 X，那么压入两个参数后的 ESP 为 X-8

计算机转移到调用的 子程序 call指令

//把返回地址压入堆栈 ESP 为 X-C 这时已经在子程序中了

可以开始使用 EBP 来存取参数

为了在返回时恢复 EBP 的值，我们还是再需要一句 push ebp

来先保存 EBP 的值，这时 ESP 为 X-10

再执行一句 mov ebp,esp

实际上这时候 [ebp + 8] 就是参数1，[ebp + c]就是参数2

MOV指令 //数据传送指令

//用于将一个数据从源地址传送到目标地址

（寄存器间的数据传送本质上也是一样的）。

其特点是不破坏源地址单元的内容

push -0x1

-1=FFFFFFFF

1=00000000

F12暂停法 Alt+K

显示调用 下断点

F8单步

FPU :(Float Point Unit,浮点运算单元)

状态寄存器

控制寄存器

fstsw 可以把状态寄存器读取到一个双字节内存位置或者AX寄存器中

fstcw 指令获取当前控制寄存器的值

fldcw 指令把 值加载到控制寄存器

fstcw 指令检查当前控制寄存器的值

ALT+M 内存镜像

ALT+E 调用了那些系统模块 该模块的存放地址和文件名

程序领空 OD载入程序 程序本身的代码

系统领空 USE32 程序调用的咱们系统的一个函数模块

放在WINDOWS目录下SYSTEM32文件夹里的一个DLL文件

esp定律 向 堆栈 中压入下一行程序的地址

逆向破解 暴力破解 绕过注册机制 追踪注册码

混淆器 壳 侦壳程序进行识别 伪装技术来混淆侦壳程序

压缩壳 程序进行体积缩小化处理

保护壳混淆或加密代码防止他人进行逆向程序、破解程序

EP段

AX 8086CPU微处理器中8个 通用寄存器之一

AX、BX、CX、DX 用于存放数据 数据寄存器

按16位使用时主要用于存放数据 临时用于存放地址

每一个都可以拆开成为两个独立的8位寄存器使用

分别用高字节和低字节表示

AH，AL等，按8位使用时只能用于存放数据

系统 地址 寄存器

EIP CPU下次要执行的指令的地址

EBP 栈的栈底指针 栈基址 （ESP传递给）

调用前 ESP存储的是栈顶地址，也是栈底地址

ESP 栈的栈顶。并且始终指向栈顶

三个指针 系统中栈 栈的数据结构 后进先处

1.栈是用来存储临时变量，函数传递的中间结果。
2.操作系统维护的，对于程序员是透明的。
函数调用 栈实现 原理

函数压栈 再出栈