2021/04/07_shellcode_第2课_GS选项、DEP、ROP链

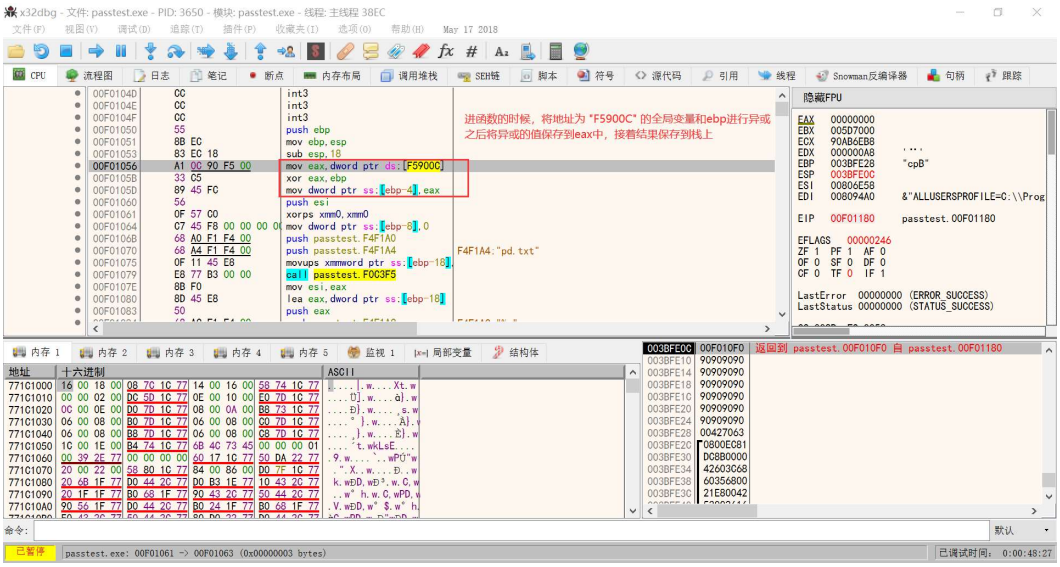
笔记本: shellcode
创建时间: 2021/4/7 星期三 10:31
作者: ileemi

GS 安全检查

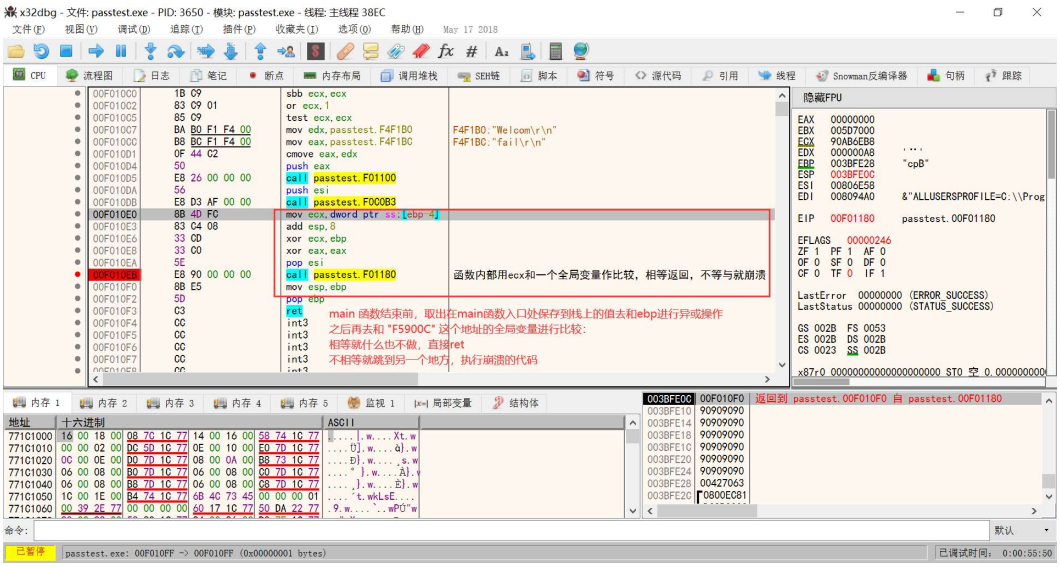
高版本的VS编译昨天的 passtest 程序时(源码如下)，在main函数入口位置有以下的操作:

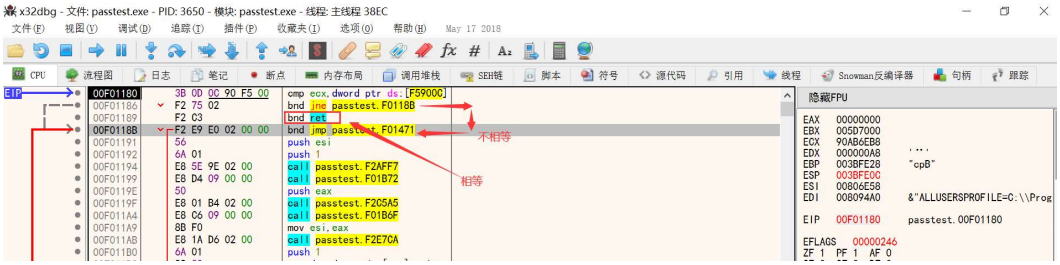
```
#include <stdio.h>
#include <stdlib.h>
char Password[]="crtest";
int main(void) {
  char cPas[20]={20 *0};
  int iResult;
  FILE* pFile = NULL;
  pFile = fopen("pd.txt", "r");
  fscanf(pFile, "%s",cPas); // %s 会导致溢出
  iResult=strcmp(Password, cPas);
  if(iResult == 0) {
    printf("Welcom\r\n");
  }else {
    printf("fail\r\n");
  }
  //system("pause");
  fclose(pFile);
  return 0;
}
```

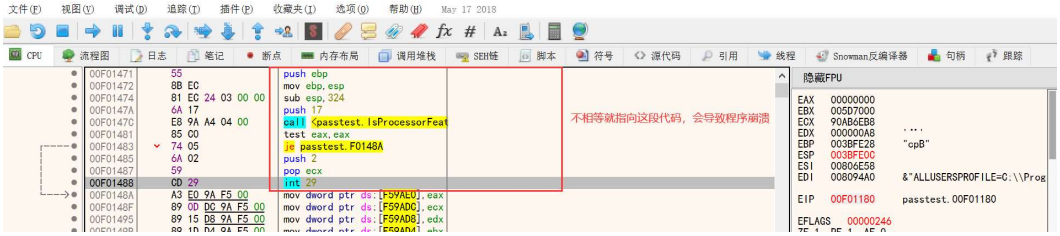
> 进函数的时候，将地址为 "F5900C" 的全局变量和ebp进行异或。之后将异或的值保存到eax中，接着再将结果保存到栈上。



> main 函数结束前，取出在main函数入口处保存到栈上的值去和ebp进行异或操作之后再去和 "F5900C" 这个地址的全局变量进行比较：相等就什么也不做，直接ret（相等，ebp再进函数前和出函数时的值一样，栈空间没有被破坏）。不相等就跳到另一个地方，执行崩溃的代码（不相等，栈针ebp被破坏）。







对应的主要汇编代码：

```
00F01056 | A1 0C 90 F5 00 | mov eax,dword ptr ds:[F5900C] |
00F0105B | 33 C5          | xor eax,ebp                   |
```

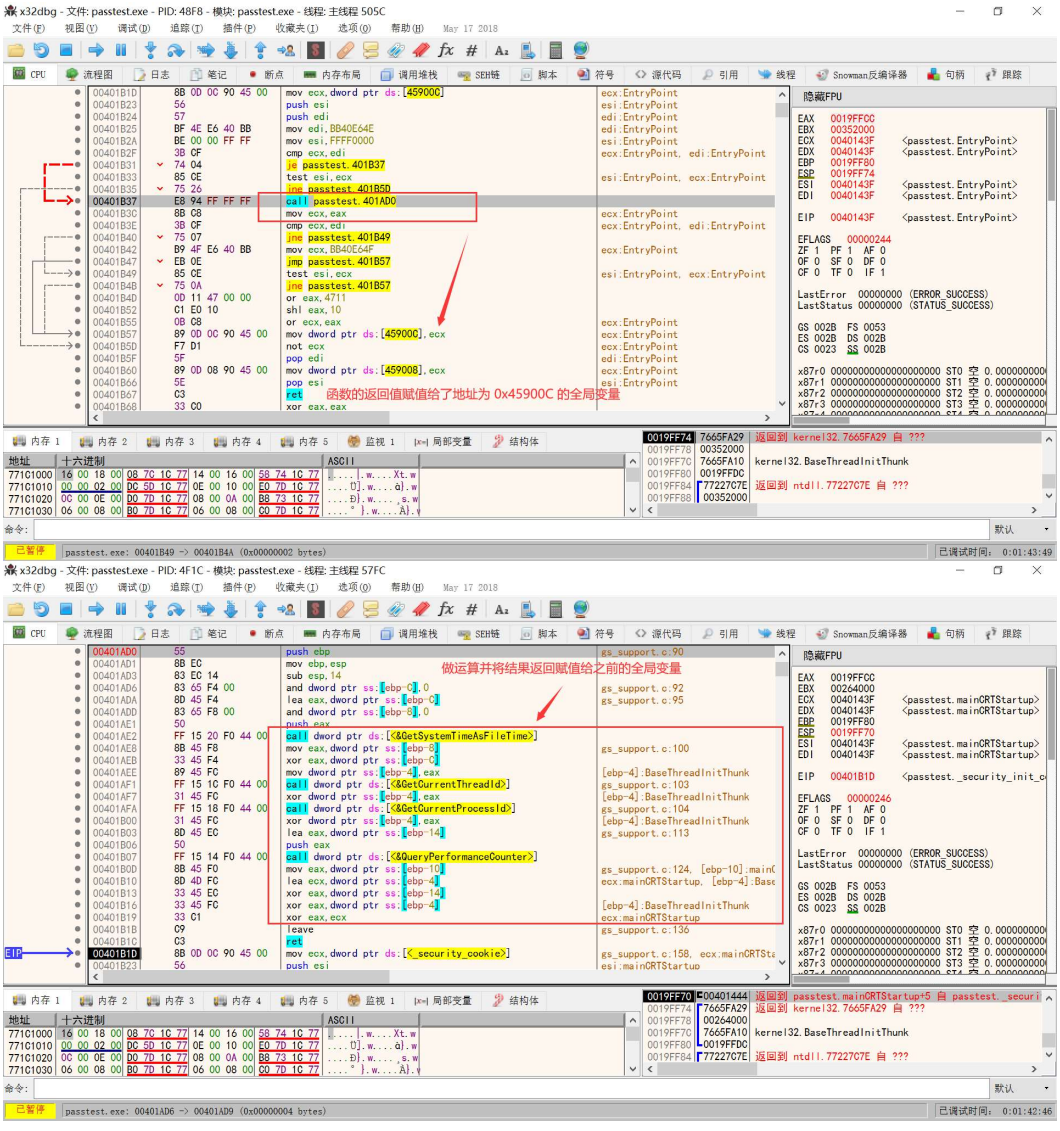
```
00F0105D | 89 45 FC        | mov dword ptr ss:[ebp-4],eax |

00F010E0 | 8B 4D FC        | mov ecx,dword ptr ss:[ebp-4] |
00F010E6 | 33 CD           | xor ecx,ebp                  |
00F010EB | E8 90 00 00 00  | call passtest.F01180         |

00F01180 | 3B 0D 0C 90 F5 00 | cmp ecx,dword ptr ds:[F5900C] |
00F01186 | F2 75 02          | bnd jne passtest.F0118B       |
00F01189 | F2 C3             | bnd ret                       |
00F0118B | F2 E9 E0 02 00 00 | bnd jmp passtest.F01471       |
```

> 上述的情况就是VS高版本编译器编译的程序默认启用了 "安全检查（/GS）"，主要为了防止栈溢出。

上述全局变量地址 "F5900C" 上的值是不确定的，所以再编写shellcode时就无法猜测，在Windows10操作系统环境下，该地址上的值（**security cookie**）由 "ntdll.dll" 中进行赋值（和系统时间、线程ID、进程ID、系统环境等信息做运算，会调用 NTQueryPerformanceCounter 等函数）。而在Windows7环境下，其值通过OEP的第一个 "call" 函数内部进行赋值操作，如下图所示：



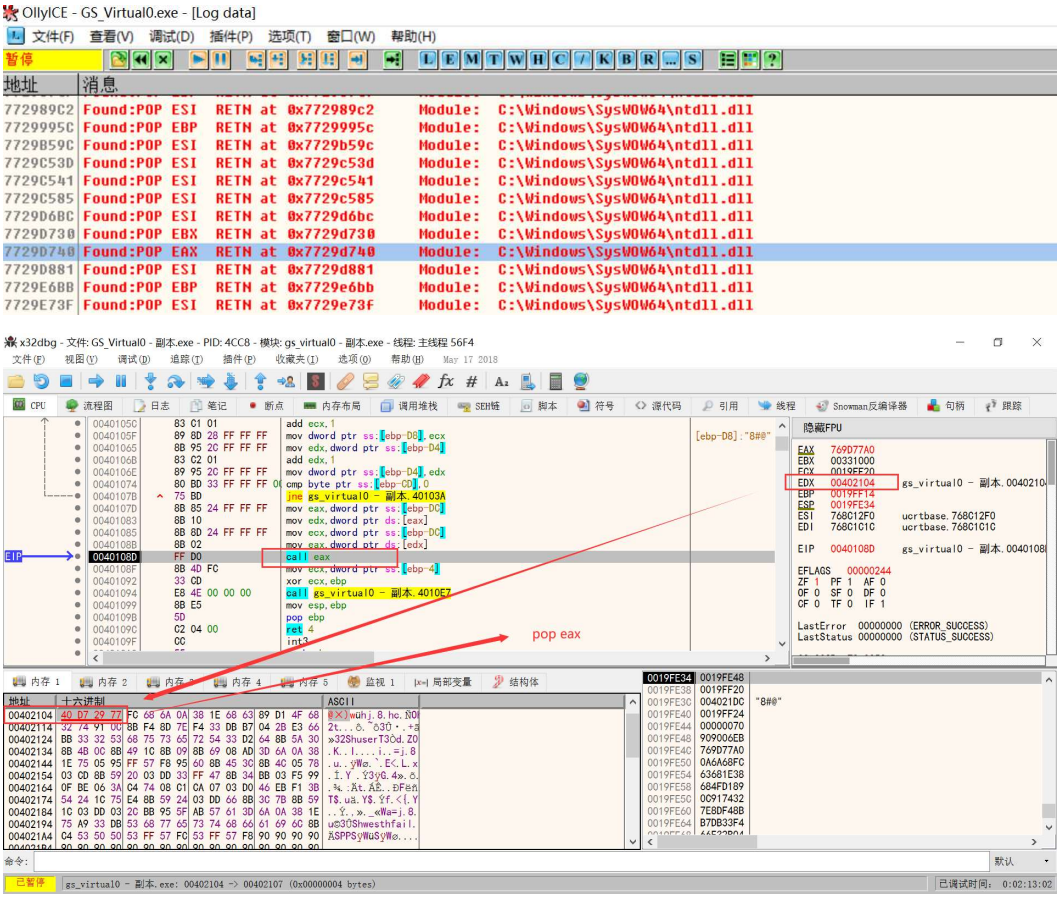
> 程序中关闭 "安全检查（/GS）" 选项后，防止栈溢出的检测代码就不存在了。

突破开启 "安全检查（/GS）" 选项后，存在检查栈溢出的代码：

- 虚表指针：程序由C++类开发，可通过虚表指针突破 "/GS" 选项。通过溢出数据（构造一个虚表）到类对象的虚表指针，通过虚表指针去调用到自己的虚表（改写虚表指针），由此获取执行权限。this指针在类对象的首四个字节。
- SEH：移动栈针，往栈上存放SEH链的节点，每个节点有两个值，一个是异常的回调函数，另一个是上一个节点的地址。溢出的值去覆盖异常回调函数的地址，异常来的时候，就会调用异常回调，此时的回调地址已经被替换，shellcode代码也就可以执行。

覆盖虚函数突破 GS

> 《0day安全:软件漏洞分析技术》 --> 273页。



修改二进制数据后，需要对其文件进行打补丁（保存内存数据到可执行文件）。

DEP

> DEP（data excution protect）：数据执行保护。开启数据执行保护，内存对应的属性必须有执行权限才可以执行代码，此时栈上不能执行代码。

没有可执行属性的内存中按道理是不能执行代码的（会触发 0xC05 异常），但是如果给予了可读属性，其内存中就可以执行代码。

- **软件DEP**：通过PE文件中的选项头的 "DllCharacteristics" 的 "Image is NX compatible" 决定，关闭选项后，栈上的地址就可以执行代码。在 VS 中可在项

目属性页中进行关闭（链接器 --> 高级 --> 数据执行保护(DEP) --> 关闭）。

- **硬件DEP**：通过在 此电脑 --> 属性 --> 高级系统设置 --> 系统属性页 系统保护 --> 数据执行保护 页面进行修改。

绕过DEP：使用 "ROP 链"，想办法首先调用一次 "VirtualProtect" 进行修改内存属性（使其内存具有可执行权限），然后跳转到执行代码位置。

> 构造 "VirtualProtect" 相关的参数的代码，在栈上不保存代码，填写上述指令的返回地址。栈溢出，溢出到返回地址，执行栈上保存的地址上对应的代码，最后执行 "VirtualProtect" 函数修改栈上对应地址的内存内存属性，之后就可以在栈上执行自己的shellcode。主要是为了避免一开始在栈上执行代码，没有可执行权限而导致程序崩溃。

mona 构造 ROP 链

使用 "mona" 去构造 ROP 链。"mona" 是一个python的脚本，需要在 "ImmunityDebugger" 调试器中运行。可通过命令 "!mona rop -m * " 搜索对应的ROP链。

使用python的 "ROpgadget" 也可对可执行文件进行操作，命令如下：

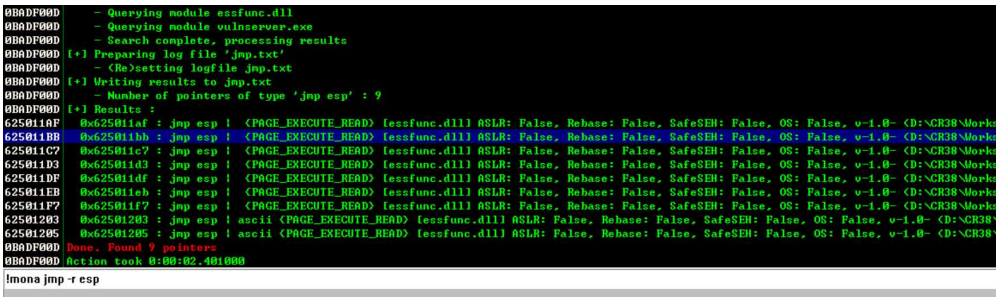
```
python ROpgadget --binary xxx.exe --ropchain >> 1.txt
```

命令示例：

```
!mona rop -cpb '\x00' -m
essfunc.dll,ntdll.dll,kernel32.dll,kernelbase.dll,msvcrt.dll,CRYPTBASE.dll,RPCRT4.dll
```

mona 相关指令：

- !mona jmp -r esp：查找 "jmp reg" 指令，后加 "-m * "，在所有模块中进行查询。



- -cqb '\x00'：将地址中含有 '00' 的进行过滤，排除坏字节。使用示例：`!mona jmp -r esp -m * -cqb '\x00\x62'