Reverse Engineer 2

By 2019

Self Modifying Code

自修改代码，顾名思义，是在程序执行时修改自己的代码

然后执行新生成的代码

可以把结果放在原位置，也可以新建一片内存存放结果

Example

WHCTF 2017 BabyRE

Facebook CTF 2019 matryoshka

可以用IDA脚本解密

https://github.com/opensource-apple/xnu/blob/master/bsd/kern/syscalls.master

Packing

加壳可以理解为一个加强版的SMC

不仅代码被加密，整个可执行文件都被加密了

运行时，会把可执行文件映像还原于内存，然后再执行

原本OS加载器(Loader)做的事情，现在需要这个壳来做

IAT and Original Entry Point

IAT需要被替换成实际在DLL中的函数地址

OEP需要变成壳的入口点(Entry Point)

如果有ASLR，那么还需要根据重定位表的数据重写代码

Unpacking

脱壳的核心思路，就是把内存映像(Image)给dump下来

变成一个PE文件

但是需要修复IAT、OEP和重定位相关的代码(若有ASLR)

Example: UPX

UPX是一个压缩壳，目的是压缩PE文件大小

不是用来混淆的，所以脱起来比较简单

Example: UPX

1. 找到OEP，在OEP处断下
2. 使用工具dump成PE文件
3. 使用IAT修复工具修复IAT

De1CTF 2019 Re_Sign

Anti-debug Technique

IsDebuggerPresent

```
BOOL MyIsDebuggerPresent(VOID)
{
    __asm {
        mov eax, fs:[0x30]              //在 TEB 偏移 30h 处获得 PEB 地址
        movzx eax, byte ptr [eax+2]     //获得 PEB 偏移 2h 处 BeingDebugged 的值
    }
}
```

Heap Magic

```
LPVOID GetHeap(SIZE_T nSize)
{
    return HeapAlloc(GetProcessHeap(), NULL, nSize);
}
 HeapPtr = GetHeap(0x100);

 ScanPtr = (PDWORD)HeapPtr;
 try {
     for(;;) {
         switch (*ScanPtr++) {
             case 0xABABABAB:
             case 0xBAADF00D:
             case 0xFEEEFEEE:
                 nMagic++;
                 break;
         }
 catch(...) {
     return (nMagic > 10) ? TRUE : FALSE;
 }
```

Anti-anti-debug?

使用StringOD！

Code Integrity Check

下断点时，指令的第一个字节会被改写成0xcc

这个时候如果检查codes的hash，会不一致

可以利用这个来检测是否在被调试

同时也可以检测代码是否被patch

Code Integrity Check

最简单的方法 : patch掉check的函数

call calc_hash

cmp eax,ebx ; ebx是原本的hash

jnz being_debugged ; -> patch成nop

Time Difference

利用时间差检测调试

```
uint64_t t0 = time64(0);
func(); // 如果在这里下了断点调试，time64差会很大
if (time64(0) - t0 > 3)
    IsDebugged();
```

Time Difference

同理还有rdtsc

解决方法：patch

Many, many others

https://www.symantec.com/connect/articles/windows-anti-debug-reference

Example: CXK壳

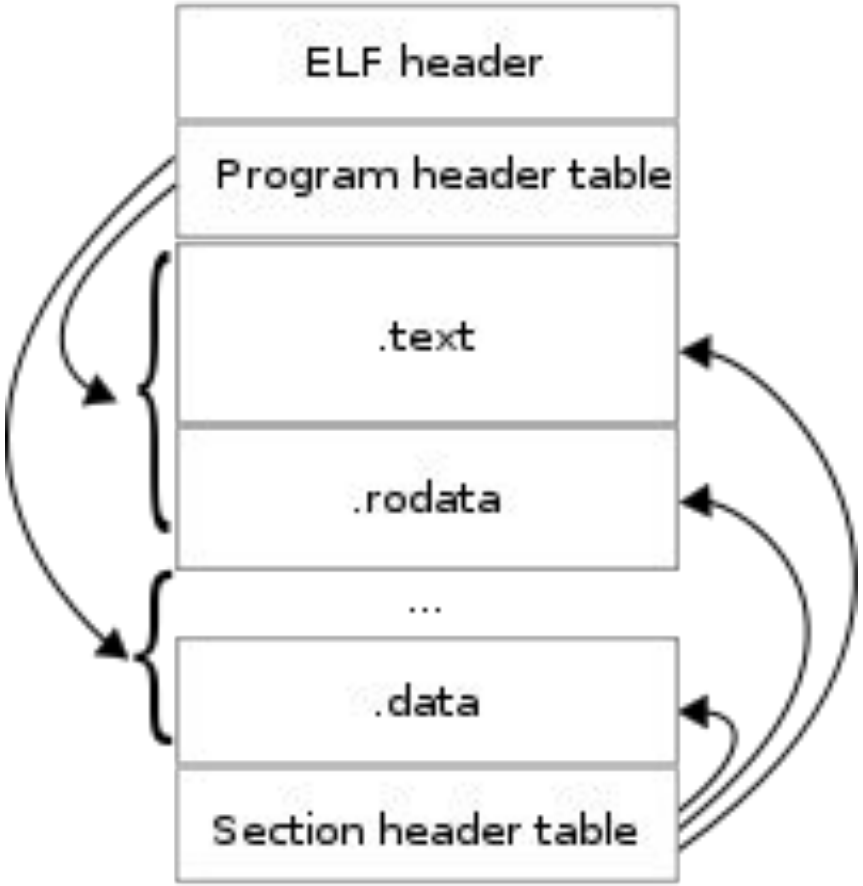
看雪CTF Q2 第七题

ELF File

Program Header Table 存放segments的信息

Section Header Table 存放sections的信息

ELF的一个segment由一个或多个section(s)组成



ELF Header

```
00 7F 45 4C 46              dword_0         dd 464C457Fh                ; DATA XREF: LOAD:00000000000002E0↓o
00                                                                      ; LOAD:00000000000002F8↓o ...
00                                                                      ; File format: \x7FELF
04 02                                       db 2                        ; File class: 64-bit
05 01                                       db 1                        ; Data encoding: little-endian
06 01                                       db 1                        ; File version
07 00                                       db 0                        ; OS/ABI: UNIX System V ABI
08 00                                       db 0                        ; ABI Version
09 00 00 00 00 00 00 00                     db 7 dup(0)                 ; Padding
10 03 00                                    dw 3                        ; File type: Shared object
12 3E 00                                    dw 3Eh                      ; Machine: x86-64
14 01 00 00 00                              dd 1                        ; File version
18 A0 07 00 00 00 00 00 00                  dq offset start             ; Entry point
20 40 00 00 00 00 00 00 00                  dq 40h                      ; PHT file offset
28 30 11 00 00 00 00 00 00                  dq 1130h                    ; SHT file offset
30 00 00 00 00                              dd 0                        ; Processor-specific flags
34 40 00                                    dw 40h                      ; ELF header size
36 38 00                                    dw 38h                      ; PHT entry size
38 09 00                                    dw 9                        ; Number of entries in PHT
3A 40 00                                    dw 40h                      ; SHT entry size
3C 1B 00                                    dw 1Bh                      ; Number of entries in SHT
3E 1A 00                                    dw 1Ah                      ; SHT entry index for string table
40                          ; ELF64 Program Header
```

ELF Segment and Section

在PE文件中，每个section都会被映射到不同页的内存

但是在ELF的Segment中，只有属性为LOAD的Segment才会被映射到内存。

ELF的Segment不需要对其0x200或0x1000，但是FO & 0xFFF要等于VA & 0xFFF（还是0xFFFFF？）

ELF Program Header

```
typedef struct
{
    Elf32_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_phdr;
```

ELF Section Header

sh_name is the index to .shstrtab

sh_addr is the address in memory

sh_offset is the offset in the file

```
typedef struct
{
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
}Elf32_Shdr;
```

.strtab

一个用来存放meta字符串的区域（不是程序用的字符串）

e.g. “\x00malloc\x00free\x00printf…”

ELF结构通过这个字符串表的下标来指定字符串

如1表示"malloc"，8表示"free"

.rel.plt

起到PE中输入表的作用

有一个指向GOT表VA的指针（相当于IAT）

有一个.dynsym的下标，

这个section即能用来输入函数也能用来输出函数

导入函数不指定特定.so

ELF Unpacking

一些关键数据不会被映射到内存中

dump下来的文件不完整

没有成熟好用的工具（如果有告诉我）

释放出来的ELF image可能放在mmap出的page中

Example: *CTF 2019

dump binary memory result.bin start_mem end_mem

先把ELF头dump下来

再找PT_LOAD的segment，把他们dump下来

然后拼接到一起

Virtual Machine

VMP是不可能讲的，这辈子不可能讲的

现在CTF中作者自己写的VM居多

Virtual Machine

识别寄存器访问，内存访问

识别基本运算

识别跳转与条件跳转

特殊指令(比较复杂的操作)

Example

HCTF 2017 ez_crackme

TSG CTF 2019 EsoVM (very hard!)

C++ Reverse Engineering

虚函数

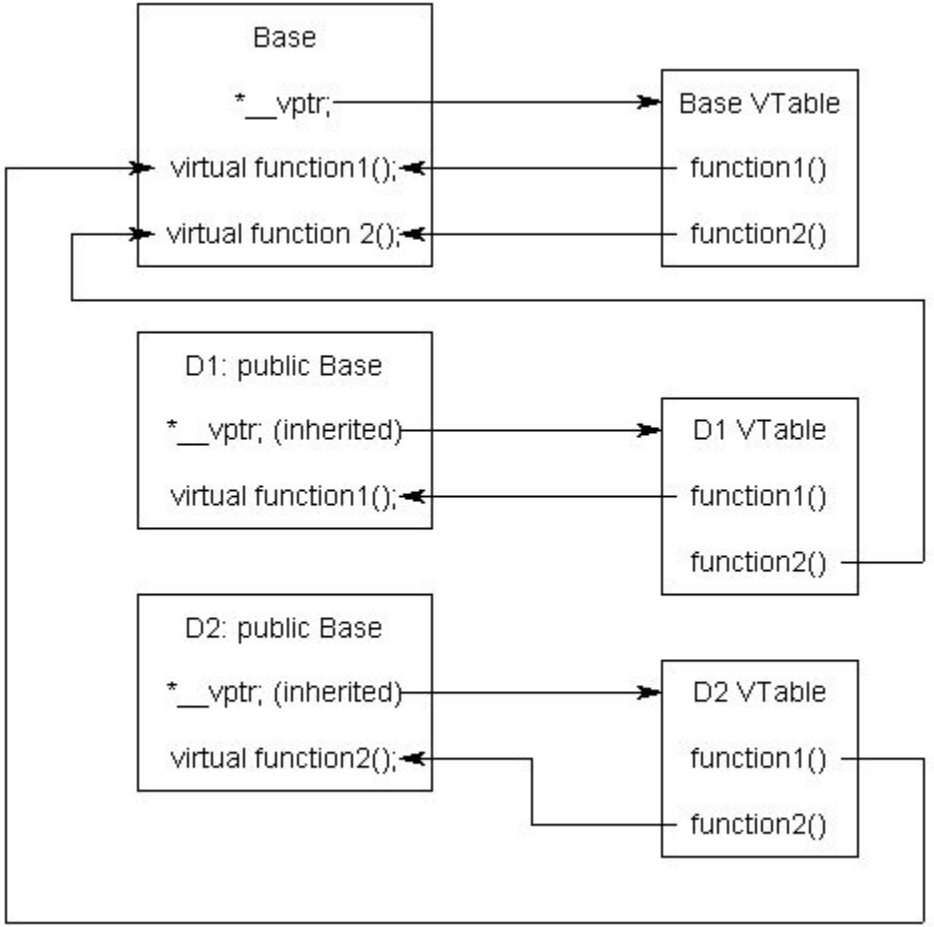
STL识别

继承关系（Class Informer插件）

Virtual Function

mov eax, [ecx]

call dword ptr [eax+4]



std::string

有一个指针，指向字符串

如果长度<0x10，指针指向(char*)this+0x10处

否则，会通过std::allocator来分配内存，底层用malloc

释放的时候判断长度，如果>=0x10就free(指针)

std::string

```
union str_maxlen
{
  char str[0x10];
  size_t maxlen;
}
```

```
struct string
{
  char *pointer;
  size_t len;
  str_maxlen data;
};
```

std::vector

用std::allocator分配array

beg指向头

end指向尾

real_end指向实际末尾

```
template<typename T>
struct vector
{
  T *beg;
  T *end;
  T *real_end;
};
```

std::vector

sizeof(T) == 0x10

p = malloc(0x40)的内存

beg = p

end = p + 0x30

real_end = p + 0x40

push_back一次，end+=0x10

再push_back一次，需要重新分配

不能只分配0x50，需要更多，

因为拷贝数组的效率问题，

比如0x80

Example

ASIS CTF 2019 Mind Space

Indentify Template STL Function

错误信息

通过用法猜+调试并且观察行为

切忌硬着头皮啃F5！

STL的算法给你源码你都不一定看得懂！

逆向的时候需要快速判断是不是STL函数

Identify Inline STL Function

有些库函数在编译时会被内联(inline)

这个时候需要快速通过大致形状判断是否是inline STL

同样，切忌硬着头皮啃！

Compiler Optimization

v5 = (((((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 4) | ((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 8;

v6 = ((v5 | ((((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 4) | ((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 16) | v5 | ((((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 4) | ((((unsigned int)(size - 1) >> 1) | ((_DWORD)size - 1)) >> 2) | ((unsigned int)(size - 1) >> 1) | (size - 1);

WTF?

Example

VolgaCTF 2019 PyTFM

SSE Instruction Set

1. 浮点数运算（现在编译器基本不用FPU了）
2. SIMD（Single Instruction, Multiple Data）

xmm0-xmm7，128位寄存器

ymm0-ymm7，256位寄存器

zmm0-zmm7，512位寄存器

SSE Instruction Set

MOVSS/MOVSD: A=Aligned, S=Single, D=double

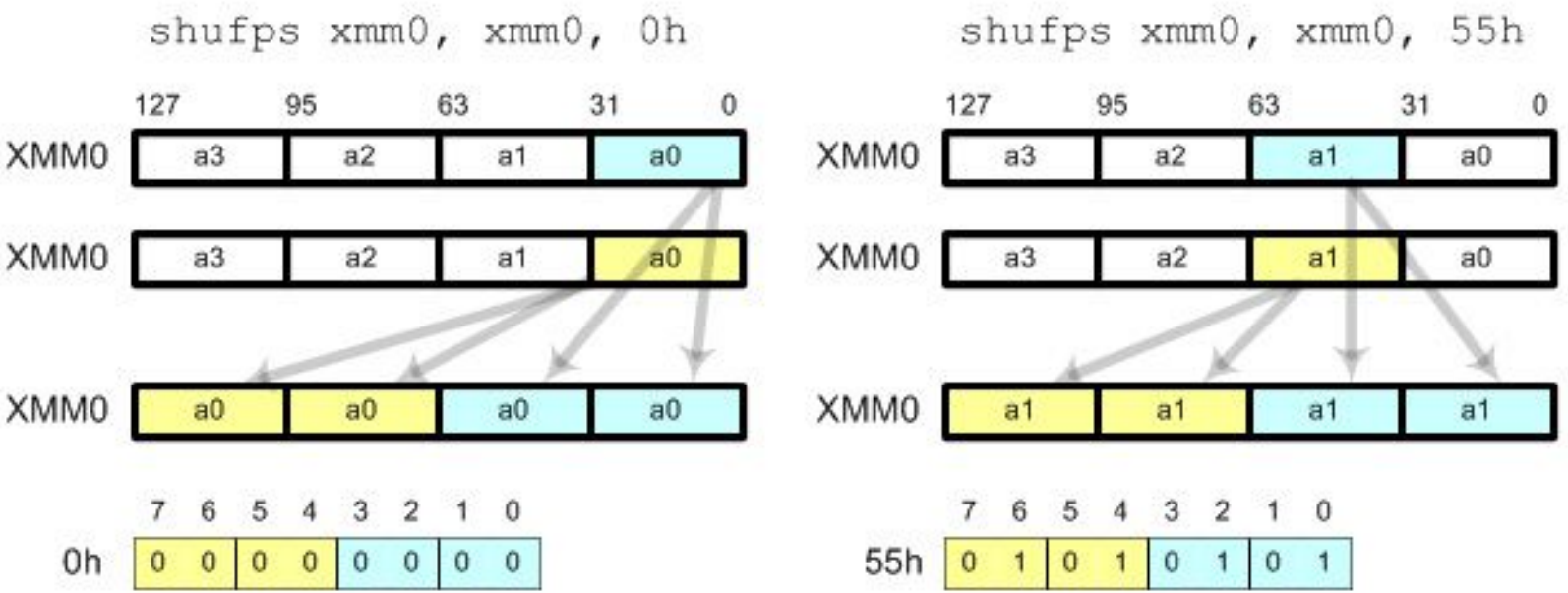
MOVAPS/MOVDQA: move aligned 128 bits at one time

ADDSS/ADDPS/XORPS: S=Single, P=Packed

CVTSI2SS: convert single integer to single single-presision

CVTPI2PD: convert packed integer to packed double-presision

SSE Instruction Set



Example

0CTF 2019 Elements

Loop Unrolling

用SSE展开循环，提高速度

学习逆向的一个好网站：https://godbolt.org/

A Save/Load Add new... CppInsights C++

```cpp
// Type your code here, or load an example
#include <stdio.h>
char keys[0x20] = "qwertyihjfndkgpansjemgmskiegnsa";
int main()
{
    char input[0x20];
    fgets(input, 0x20, stdin);
    for (int i = 0; i < 0x20; i++)
    {
        input[i] ^= keys[i];
    }
    puts(input);
}
```

x86-64 gcc 9.1 -O2

A 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

```
1   main:
2           sub     rsp, 40
3           mov     rdx, QWORD PTR stdin[rip]
4           mov     esi, 32
5           mov     rdi, rsp
6           call    fgets
7           xor     eax, eax
8   .L2:
9           movzx   edx, BYTE PTR [rsp+rax]
10          xor     dl, BYTE PTR keys[rax]
11          mov     BYTE PTR [rsp+rax], dl
12          add     rax, 1
13          cmp     rax, 32
14          jne     .L2
15          mov     rdi, rsp
16          call    puts
17          xor     eax, eax
18          add     rsp, 40
19          ret
```

x86-64 gcc 9.1 -O3

A 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

```
1   main:
2           sub     rsp, 40
3           mov     rdx, QWORD PTR stdin[rip]
4           mov     esi, 32
5           mov     rdi, rsp
6           call    fgets
7           movdqa  xmm0, XMMWORD PTR [rsp]
8           mov     rdi, rsp
9           pxor    xmm0, XMMWORD PTR keys[rip]
10          movaps  XMMWORD PTR [rsp], xmm0
11          movdqa  xmm0, XMMWORD PTR [rsp+16]
12          pxor    xmm0, XMMWORD PTR keys[rip+16]
13          movaps  XMMWORD PTR [rsp+16], xmm0
14          call    puts
15          xor     eax, eax
16          add     rsp, 40
17          ret
18  keys:
19          .string "qwertyihjfndkgpansjemgmskiegnsa"
```

Example

RCTF 2018 babyre2

My Methodology

有时候巧妙的思路比扎实的逆向功底更重要

PlaidCTF 2019 Plaid Party Planning III

My Methodology

静态看一个函数，分析其功能

1. 它执行了什么
2. 它做了什么
3. 它是用来做什么的

My Methodology

做修改变量类型和名字的好习惯，尽管可能一开始错了

但这会让代码更加清晰，

所以能解放大脑去处理一些更重要的信息，

能让代码更容易看懂

My Methodology

猜+调试验证，有时候比静态分析更有用

动静结合，比如通过调试查看变量状态

e.g. how index changes in a loop

My Methodology

搜索常量、字符串寻找开源代码

bindiff工具

My Methodology

遇到没见过的东西怎么办？

e.g. API/SSE指令

e.g. 非PE、非ELF、其他非常规逆向

e.g. 没见过的指令集

答：善用Google，**快速定位**到所需要的东西