

利用符号执行去除控制流平坦化

2017年01月22日

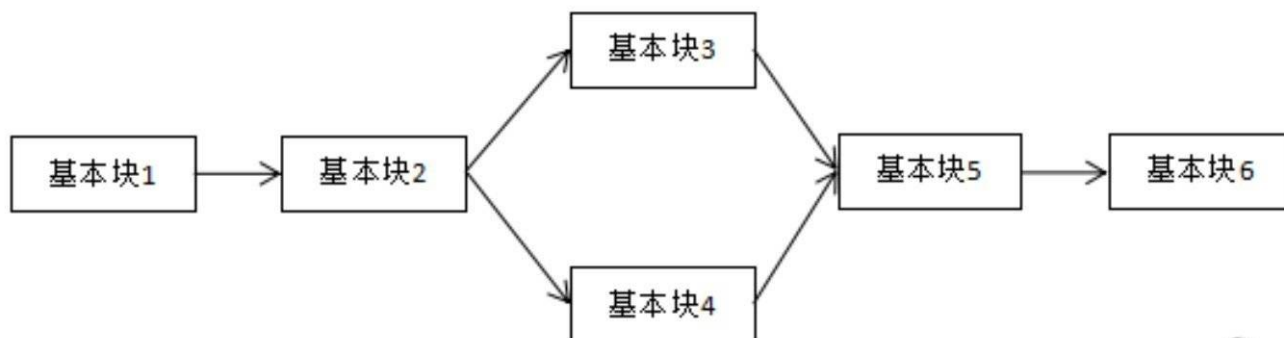
经验心得 (/category/experience/) · 二进制安全 (/category/bin-security/)

作者: **bird@tsrc** (<https://security.tencent.com/index.php/blog/msg/112>)

1. 背景

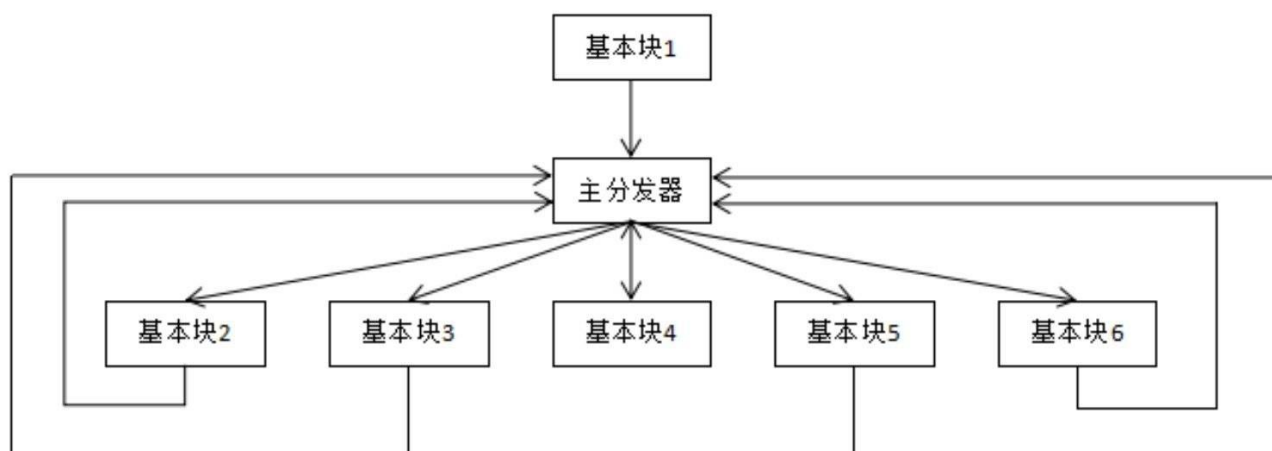
1.1 控制流平坦化

控制流平坦化(control flow flattening)的基本思想主要是通过一个主分发器来控制程序基本块的执行流程，例如下图是正常的执行流程



Seebug

经过控制流平坦化后的执行流程就如下图



Seebug

这样可以模糊基本块之间的前后关系，增加程序分析的难度，同时这个流程也很像VM的执行流程。更多控制流平坦化的细节可以看Obfuscating C++ programs via control flow flattening (http://ac.inf.elte.hu/Vol_030_2009/003.pdf)，本文以 Obfuscator-LLVM (<https://github.com/obfuscator-llvm/obfuscator/tree/llvm-3.6.1>) 的控制流平坦化为例。

1.2 符号执行

符号执行

(<https://pdfs.semanticscholar.org/a29f/c90b207befb42f67a040c6a07ea6699f6bad.pdf>) 是一种重要的形式化方法和软件分析技术，通过使用符号执行技术，将程序中变量的值表示为符号值和常量组成的计算表达式，符号是指取值集合的记号，程序计算的输出被表示为输入符号值的函数，其在软件测试和程序验证中发挥着重要作用，并可以应用于程序漏洞的检测。

符号执行的发展是从静态符号执行到动态符号执行到选择性符号执行

(<http://dslab.epfl.ch/pubs/selsymbex.pdf>)，动态符号执行会以具体数值作为输入来模拟执行程序，是混合执行

(<http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf>)(concolic execution)的典型代表，有很高的精确度，目前较新的符号执行工具有Triton (<https://github.com/JonathanSalwan/Triton>)和angr (<https://github.com/angr/angr>)，本文是以angr为例。

2. 分析

首先写一个简单的示例程序

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_password(char *passwd) {
    int i, sum = 0;
    for (i = 0; ; i++) {
        if (!passwd[i]) {
            break;
        }
        sum += passwd[i];
    }
    if (i == 4) {
        if (sum == 0x1a1 && passwd[3] > 'c' && passwd[3] < 'e' && passwd[0] ==
            if ((passwd[3] ^ 0xd) == passwd[1]) {
                return 1;
            }
            puts("Orz...");
        }
    }
    else{
        puts("len error");
    }
    return 0;
}

int main(int argc, char **argv) {
    if (argc != 2){
        puts("error");
        return 1;
    }
    if (check_password(argv[1])){
        puts("Congratulation!");
    }
    else{
        puts("error");
    }
    return 0;
}

```

编译

```
gcc check_passwd.c -o check_passwd
```

用IDA查看未经过控制流平坦化的控制流程图(CFG)



```

00000000000000520 var_18= quword ptr -18h
00000000000000520 var_8= dword ptr -8
00000000000000520 var_4= dword ptr -4
00000000000000520
00000000000000520 push rbp
00000000000000520 mov rbp, rsp
00000000000000520 sub rsp, 20h
00000000000000531 mov [rbp+var_18], rdi
00000000000000535 mov [rbp+var_4], 0
00000000000000540 mov [rbp+var_8], 0

```

```

00000000000000547 loc_400547:
00000000000000547 mov eax, [rbp+var_8]
00000000000000548 movsxd rdx, eax
00000000000000548 mov rax, [rbp+var_18]
00000000000000551 add rax, rdx
00000000000000554 movzx eax, byte ptr [rax]
00000000000000557 test al, al
00000000000000559 jnz short loc_400568

```

```

00000000000000558 nop
0000000000000055C cmp [rbp+var_8], 4
00000000000000560 jnz loc_4005E8

```

```

00000000000000568 loc_400568:
00000000000000568 mov eax, [rbp+var_8]
00000000000000568 movsxd rdx, eax
0000000000000056E mov rax, [rbp+var_18]
00000000000000572 add rax, rdx
00000000000000575 movzx eax, byte ptr [rax]
00000000000000578 movsx eax, al
0000000000000057B add [rbp+var_4], eax
0000000000000057E add [rbp+var_8], 1
00000000000000582 jmp short loc_400547

```

```

00000000000000566 jmp short loc_400584

```

```

00000000000000584 loc_400584:
00000000000000584 cmp [rbp+var_4], 101h
00000000000000588 jnz short loc_4005F2

```

```

0000000000000058D mov rax, [rbp+var_18]
00000000000000591 add rax, 3
00000000000000595 movzx eax, byte ptr [rax]
00000000000000598 cmp al, 63h
0000000000000059A jle short loc_4005F2

```

```

0000000000000059C mov rax, [rbp+var_18]
000000000000005A0 add rax, 3
000000000000005A4 movzx eax, byte ptr [rax]
000000000000005A7 cmp al, 64h
000000000000005A9 jg short loc_4005F2

```

```

000000000000005A8 mov rax, [rbp+var_18]
000000000000005AF movzx eax, byte ptr [rax]
000000000000005B2 cmp al, 62h
000000000000005B4 jnz short loc_4005F2

```

```

000000000000005B6 mov rax, [rbp+var_18]
000000000000005BA add rax, 3
000000000000005BE movzx eax, byte ptr [rax]
000000000000005C1 xor eax, 00h
000000000000005C4 mov edx, eax
000000000000005C6 mov rax, [rbp+var_18]
000000000000005CA add rax, 1
000000000000005CE movzx eax, byte ptr [rax]
000000000000005D1 cmp dl, al
000000000000005D3 jnz short loc_4005DC

```

```

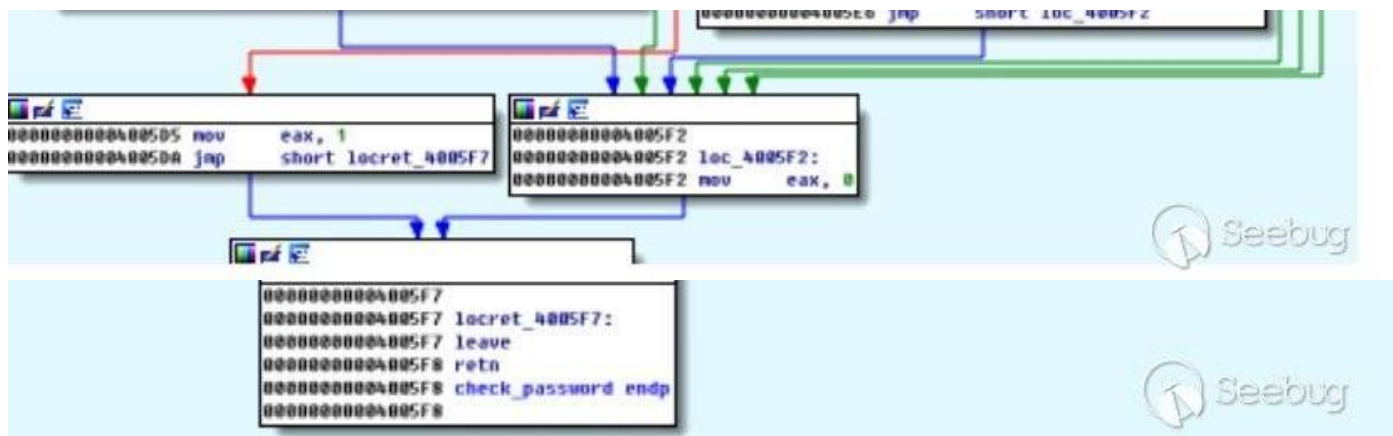
000000000000005E8 loc_4005E8: ; "len error"
000000000000005E8 mov edi, offset alenError
000000000000005EB call _puts

```

```

000000000000005DC loc_4005DC: ; "0x2..."
000000000000005DC mov edi, offset s
000000000000005E1 call _puts

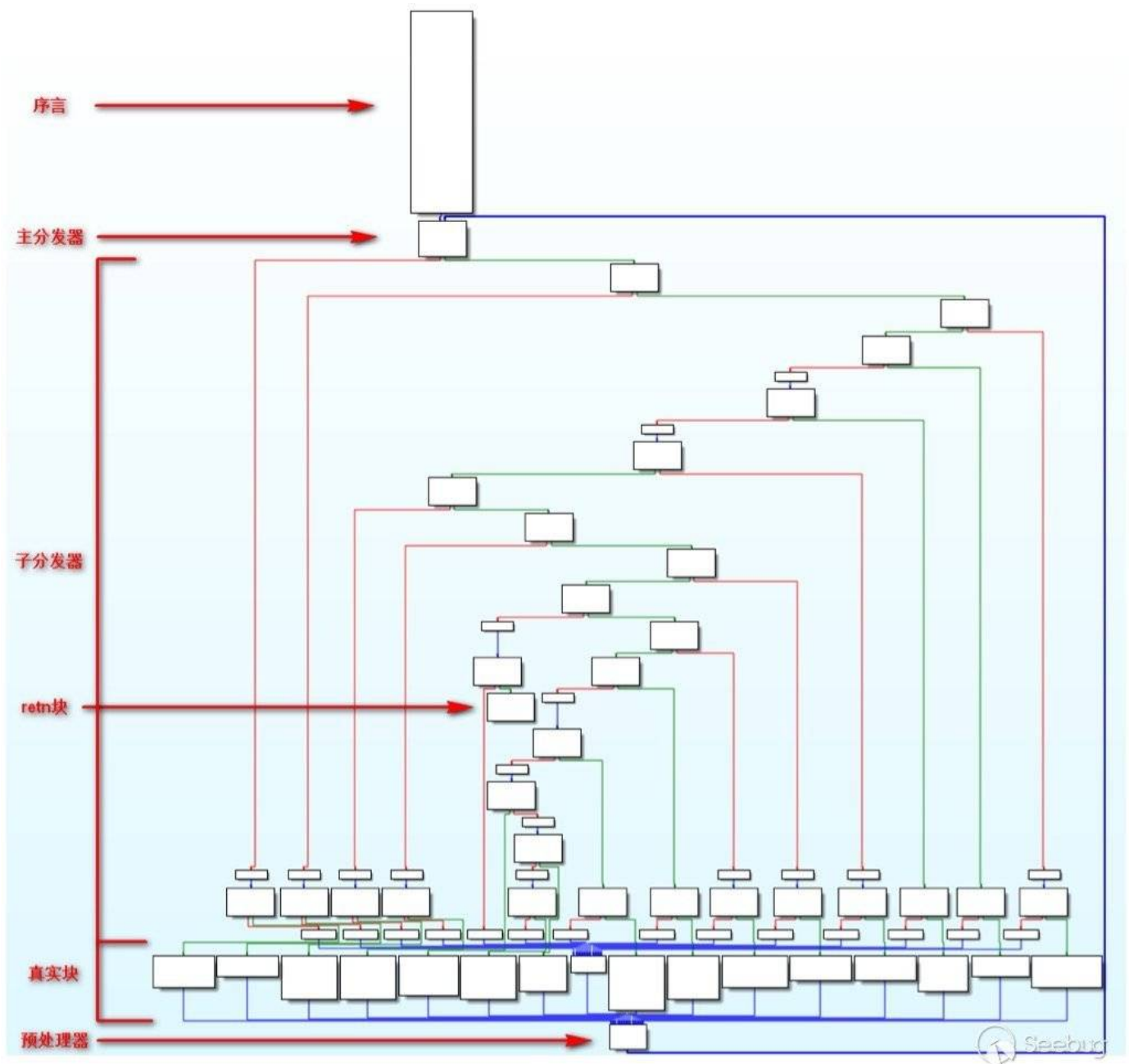
```



添加控制流平坦化

```
build/bin/clang check_passwd.c -o check_passwd_flat -mllvm -fla
```

可以看到控制流平坦化后的CFG非常漂亮



通过分析可以发现原始的执行逻辑只在真实块(自己想的名称...)以及序言和retn块中，其中会

产生分支的真实块中主要是通过CMOV指令来控制跳转到哪一个分支，因此只要确定这些块的前后关系就可以恢复出原始的CFG，这个思路主要是参考Deobfuscation: recovering an OLLVM-protected program (<http://blog.quarkslab.com/deobfuscation-recovering-an-llvm-protected-program.html>)。

3. 实现

3.1 获取真实块、序言、retn块和无用块

由于angr的CFG跟IDA的有点不同，因此本文使用BARF (<https://github.com/programa-stic/barf-project>)来获取，后来问了Fish Wang可以用angr-management (<https://github.com/angr/angr-management/blob/master/angrmanagement/utils/graph.py>) 下的to_supergraph来获取。主要思路：

1. 函数的开始地址为序言的地址
2. 序言的后继为主分发器
3. 后继为主分发器的块为预处理器
4. 后继为预处理器的块为真实块
5. 无后继的块为retn块
6. 剩下的为无用块

主要代码：

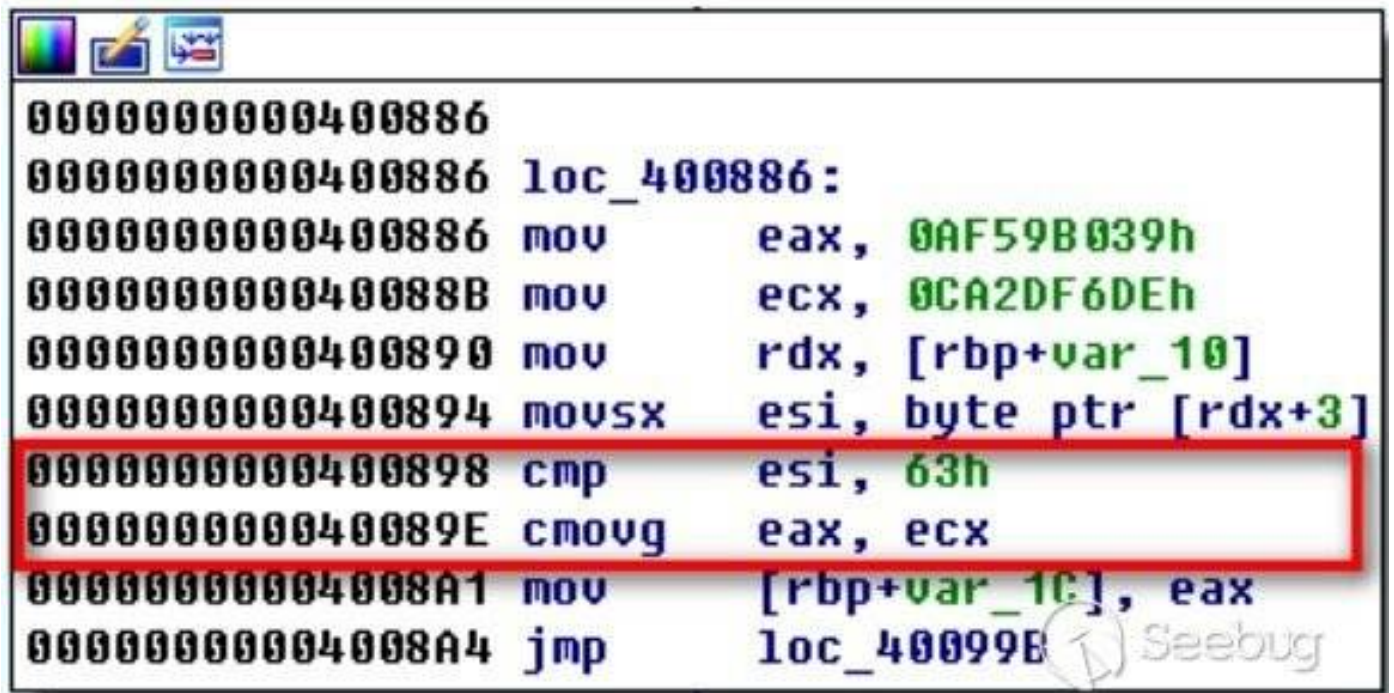
```
def get_retn_predispatcher(cfg):
    for block in cfg.basic_blocks:
        if len(block.branches) == 0 and block.direct_branch == None:
            retn = block.start_address
        elif block.direct_branch == main_dispatcher:
            pre_dispatcher = block.start_address
    return retn, pre_dispatcher

def get_relevant_nop_blocks(cfg):
    relevant_blocks = []
    nop_blocks = []
    for block in cfg.basic_blocks:
        if block.direct_branch == pre_dispatcher and len(block.instrs)
!= 1:
            relevant_blocks.append(block.start_address)
        elif block.start_address != prologue and block.start_address !=
retn:
            nop_blocks.append(block)
    return relevant_blocks, nop_blocks
```

3.2 确定真实块、序言和retn块的前后关系

这个步骤主要是使用符号执行，为了方便，这里把真实块、序言和retn块统称为真实块，符号执行从每个真实块的起始地址开始，直到执行到下一个真实块。如果遇到分支，就改变判断值 执行两次来获取分支的地址，这里用angr的inspect在遇到类型为ITE

的IR表达式时，改变临时变量的值来实现，例如下面这个块



```
00000000000400886
00000000000400886 loc_400886:
00000000000400886 mov     eax, 0AF59B039h
0000000000040088B mov     ecx, 0CA2DF6DEh
00000000000400890 mov     rdx, [rbp+var_10]
00000000000400894 movsx   esi, byte ptr [rdx+3]
00000000000400898 cmp     esi, 63h
0000000000040089E cmovg   eax, ecx
000000000004008A1 mov     [rbp+var_10], eax
000000000004008A4 jmp     loc_40099B
```

使用**statement before**类型的inspect:

```
1. def statement_hook(inspect):
2.     global modify_value
3.     expressions =
4.         state.scratch.irsb.statements[state.inspect.statement].expressions
5.         if len(expressions) != 0 and isinstance(expressions[0], pyvex.expr.
6.             ITE):
7.                 state.scratch.temps[expressions[0].cond.tmp] = modify_value
8.                 state.inspect._breakpoints['statement'] = []
9.
10. state.inspect.b('statement', when=simuvex.BP_BEFORE,
11.                 action=statement_inspect)
```

修改临时变量28为false或true再执行就可以得到分支的地址

```
t48 = ITE(t28,0xca2df6de,0xaf59b039)
```

如果遇到call指令，使用hook的方式直接返回

```

1. def retn_procedure(state):
2.     global b
3.     ip = state.se.any_int(state.regs.ip)
4.     b.unhook(ip)
5.     return
6.
7. b.hook(hook_addr, retn_procedure, length=5)

```



主要代码:

```

1. for relevant in relevants_without_retn:
2.     block = cfg.find_basic_block(relevant)
3.     has_branches = False
4.     hook_addr = None
5.     for ins in block.instrs:
6.         if ins.asm_instr.mnemonic.startswith('cmov'):
7.             patch_instrs[relevant] = ins.asm_instr
8.             has_branches = True
9.         elif ins.asm_instr.mnemonic.startswith('call'):
10.            hook_addr = ins.address
11.    if has_branches:
12.        flow[relevant].append(symbolic_execution(relevant, hook_addr, c
laripy.BVV(1, 1), True))
13.        flow[relevant].append(symbolic_execution(relevant, hook_addr, c
laripy.BVV(0, 1), True))
14.    else:
15.        flow[relevant].append(symbolic_execution(relevant))

```



3.3 Patch二进制程序

首先把无用块都改成nop指令

```

1. for nop_block in nop_blocks:
2.     for i in range(nop_block.start_address - base_addr, nop_block.end_a
ddress - base_addr + 1):
3.         origin_data[i] = '\x90'

```



然后针对没有产生分支的真实块把最后一条指令改成jmp指令跳转到下一真实块

```

1. last_instr = cfg.find_basic_block(parent).instrs[-1].asm_instr
2. file_offset = last_instr.address - base_addr
3. origin_data[file_offset] = opcode['jmp']
4. file_offset += 1
5. fill_nop(origin_data, file_offset, file_offset + last_instr.size - 1)
6. fill_jump_offset(origin_data, file_offset, child[0] - last_instr.address - 5)

```

针对产生分支的真实块把CMOV指令改成相应的条件跳转指令跳向符合条件的分支，例如 CMOVZ 改成 JZ，再在这条之后添加 JMP 指令跳向另一分支

```

1. instr = patch_instrs[parent]
2. file_offset = instr.address - base_addr
3. fill_nop(origin_data, file_offset,
  cfg.find_basic_block(parent).end_address - base_addr + 1)
4. origin_data[file_offset] = opcode['j']
5. origin_data[file_offset + 1] = opcode[instr.mnemonic[4:]]
6. fill_jump_offset(origin_data, file_offset + 2, child[0] - instr.address - 6)
7. file_offset += 6
8. origin_data[file_offset] = opcode['jmp']
9. fill_jump_offset(origin_data, file_offset + 1, child[1] - (instr.address + 6) - 5)

```

上述就是去除控制流平坦化的总体实现思路。

4. 演示

去除制定函数的控制流平坦化

```
python deflat.py check_passwd_flat 0x400530
```

用IDA查看恢复后的CFG






```

__int64 __fastcall check_password(__int64 a1)
{
    int v2; // [sp+88h] [bp-18h]@1
    int i; // [sp+8Ch] [bp-14h]@1

    v2 = 0;
    for ( i = 0; *(a1 + i); ++i )
        v2 += *(a1 + i);
    if ( i != 4 )
    {
        puts("len error");
LABEL_14:
        return 0;
    }
    if ( v2 != 0x1A1 || *(a1 + 3) <= 'c' || *(a1 + 3) >= 'e' || *a1 != 'b' )
        goto LABEL_14;
    if ( (*(a1 + 3) ^ 0xD) == *(a1 + 1) )
        return 1;
    puts("0rz...");
    goto LABEL_14;
}

```



5. 总结

本文主要针对 x86 架构下 Obfuscator-LLVM 的控制流平坦化，但最重要的是去除控制流平坦化 过程中的思路，同时当函数比较复杂时可能速度会有点慢。有时间会以此为基础尝试分析伪造 控制流、指令替换和 VM 等软件保护手段，另外符号执行也可以应用于漏洞挖掘领域，例如借助符号执行生成覆盖率更高的 Fuzzing 测试集以及求解达到漏洞点的路径等。由于小弟刚学习 符号执行，可能有理解错误的地方，欢迎研究符号执行或者认为有更好思路的师傅们吐槽。。。最后，感谢 angr 主要开发者 Fish Wang 在这期间的耐心帮助。

6. 参考

1. Obfuscating C++ programs via control flow flattening
(http://ac.inf.elte.hu/Vol_030_2009/003.pdf)
2. <https://github.com/obfuscator-llvm/obfuscator/tree/llvm-3.6.1>
3. Symbolic Execution and Program Testing
(<https://pdfs.semanticscholar.org/a29f/c90b207befb42f67a040c6a07ea6699f6bad.pdf>)
4. Selective Symbolic Execution (<http://dslab.epfl.ch/pubs/selsymbex.pdf>)
5. CUTE: A Concolic Unit Testing Engine for C
(<http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf>)
6. <https://github.com/JonathanSalwan/Triton>
7. <https://github.com/angr/angr>

8. <http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html>
 9. <https://github.com/programa-stic/barf-project>
 10. <https://github.com/angr/angr-management>
-



本文由 Seebug Paper 发布，如需转载请注明来源。本文地址：
<https://paper.seebug.org/192/> (<https://paper.seebug.org/192/>)