【译】反解码（反编译）技术举例

angel010 / 2017-11-20 09:30:00 / 浏览数 2507 技术文章 技术文章 顶(0) 踩(0)

程序编译的操作把人类可读的源码翻译成机器能识别的机器代码，反编译器是尝试恢复源码级表示的行为。

现在的编译工具变得越来越强大，一键就可以实现逆向工程，将机器代码变成人类可读的源码。本文介绍几种反反编译技术来干扰和误导依赖反编译工具的逆向工程师们。

Positive SP值

第一项技术是Positive SP值，这是一项古典的但是很有效果的方法，可以干扰Hex-Rays反编译器的行为。在IDA Pro中，如果不能清楚之前的栈分配，反编译器会拒绝反编译一个函数。

图2 在反编译的过程中如果检测到一个positive栈指针，IDA的报错消息
这是只有在IDA不能找出特定函数调用的type定义时才会产生。

作为反反编译技术，开发者可以找出在函数中想要隐藏在函数中的行为，使用的opaque predicate技术，这种技术可以破坏栈指针的平衡。

```
//
// compiled on Ubuntu 16.04 with:
//    gcc -o predicate predicate.c -masm=intel
//

#include <stdio.h>

#define positive_sp_predicate \
   __asm__ ("  push     rax      \n"\
            "  xor      eax, eax \n"\
            "  jz       opaque   \n"\
            "  add      rsp, 4   \n"\
            "opaque:             \n"\
            "  pop      rax      \n");

void protected()
{
   positive_sp_predicate;
   puts("Can't decompile this function");
}

void main()
{
   protected();
}
```

在运行时，positive_sp_predicate宏中定义的add rsp, 4永远不会执行，但会诱发IDA执行反编译的静态分析。尝试去反编译protected()函数产生了下面的结果：

图3 使用opaque predicates来扰乱栈指针，做为一种反反编译的方法

这种技术很著名，可以通过补丁修复或者手动纠正栈的偏移量来解决。因此，这种技术可以干扰那些跳过反装载（assembly）直接进行反编译的初级逆向工程师，比如学生

Return Hijacking

现代的反编译器的一个希望就是精确地识别并提取编辑器生成的低层的记账逻辑，比如函数序言、尾记和控制流元数据。

图4 编译器生成函数结束会保存寄存器、分配给栈结构
反编译器会从输出中省去这种信息，因为保存寄存器或管理栈结构分配的概念不存在于源码级。这些省去的一个有意思的地方是我们可以在反编译器没有报错的情况下在从函

图5 在ROP链上旋转栈指针
Stack
pivoting是一种二进制利用的常用方法，可以用来获取任意ROP。开发者用它作为一种从逆向工程师处劫持执行的机制。那些只关注反编译器的输出可以保证会错过这些信息

图6 反编译main，一个假的函数以stack pivot结束
我们在一个小的已经编译为二进制文件的ROP链上进行了栈旋转，这起着误导的作用。最终的结果是一个对反编译器不可见的函数调用。分开调用的函数简单地打印出“evil code”来证明代码执行了。

图7 使用return hijacking反反编译技术执行编译过的二进制文件
证明从反编译器技术中隐藏代码技术的代码在下面可以找到。

```
// compiled on Ubuntu 16.04 with:
//    gcc -o return return.c -masm=intel
//#include <stdio.h>void evil() {
   puts("Evil Code");}
extern void gadget();__asm__ (".global gadget        \n"
        "gadget:               \n"
        "  pop       rax       \n"
        "  mov       rsp, rbp  \n"
        "  call      rax       \n"
        "  pop       rbp       \n"
        "  ret                 \n");
void * gadgets[] = {gadget, evil};
void deceptive() {
   puts("Hello World!");
   __asm__("mov rsp, %0;\n"
           "ret"
           :
           :"i" (gadgets));}
void main() {
   deceptive();}
```

滥用‘noreturn’函数

还有一项可以利用IDA的技术是感知函数（perception of functions），这些函数会被自动标记为noreturn。一些常见的noreturn函数的例子有exit()，abort()等。
在为给定的函数生成伪代码的时候，反编译器会丢弃所有调用noreturn函数之后的代码。正常的逻辑是exit() 这样的函数之后不应该再有代码执行。

图8 noreturn函数之后的代码对反编译器是不可见的
如果程序员可以欺骗IDA让IDA相信某个不是noreturn函数的函数是noreturn函数，那么就可以隐藏代码了。下面的例子证明了我们的设想。

```
//
// compiled on Ubuntu 16.04 with:
//    gcc -o noreturn noreturn.c
//#include <stdio.h>
#include <stdlib.h>void ignore() {
   exit(0);                 // force a PLT/GOT entry for exit()
   }
void deceptive() {
   puts("Hello World!");
   srand(0);                // post-processing will swap srand() <--> exit()    puts("Evil Code");
   }
void main() {
   deceptive();
   }
```

通过编译上面的代码，并对生成的二进制文件运行基于Binary Ninja的post-processing script脚本，我们可以在Procedure Linkage Table表中交换进栈的序数。这些索引在runtime状态下解析库引用时使用。

图9 在ELF头部交换PLT序数

在上面的例子中，我们交换了srand() 和 exit()的序数，并且修改了一些调用。IDA相信修改后的二进制文件的deceptive()函数调用的是noreturn函数exit(),而不是应该调用的srand()。

图10 隐藏在noreturn调用后的欺骗函数
我们在IDA runtime中看到的 exit()调用实际上是srand() 调用。这对反编译器的效果等同于return hijacking技术。对二进制文件的运行证明了‘Evil Code’在反编译器不知道的情况下被运行了。

图11 使用noreturn技术执行编译的二进制文件

结论

反编译器是一种非常有用但是不完美的技术。利用不完整的信息为用户提供尽可能完整的估计。恶意用户就可以利用这种不对称进行欺骗。随着业界越来越依赖反编译器，
https://blog.ret2.io/2017/11/16/dangers-of-the-decompiler/

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板