反漏洞挖掘介绍

演讲人：俞科技

职务：华为网络安全技术专家

日期：2014.9.23

当前软件攻防对抗手段回顾

- 漏洞利用 vs 反漏洞利用
  - 编译器层：GS、Safeseh选项
  - 操作系统层：ASLR
  - 处理器层：DEP

- 逆向工程 vs 反逆向工程
  - 反调试/反虚拟机
  - 花指令
  - 垃圾指令
  - 指令虚拟化
  - 加壳
  - 代码抽取执行
  - …

- 调试 vs 反调试 vs 反反调试
  - IsDebuggerPresent
  - Heap flag
  - CheckRemoteDebuggerPresent（ProcessDebugPort）
  - NtSetInformationThread （ThreadHideFromDebugger）
  - Trap Flag
  - 程序执行时间
  - 父进程检查
  - ……

- 后门隐藏 vs 后门检测
  - 应用层
    - 感染程序
    - 注册表自启动项
    - Dll劫持/注入
  - 服务
    - 新建服务
    - 替换/劫持原服务
  - 内核rootkit
  - Bootkit
    - MBR
    - NTLDR
  - Chipkit

- 外挂 vs 反外挂
  - 防代码注入
  - 内核级反调试
  - 防止程序多开
  - 关键代码保护
  - …

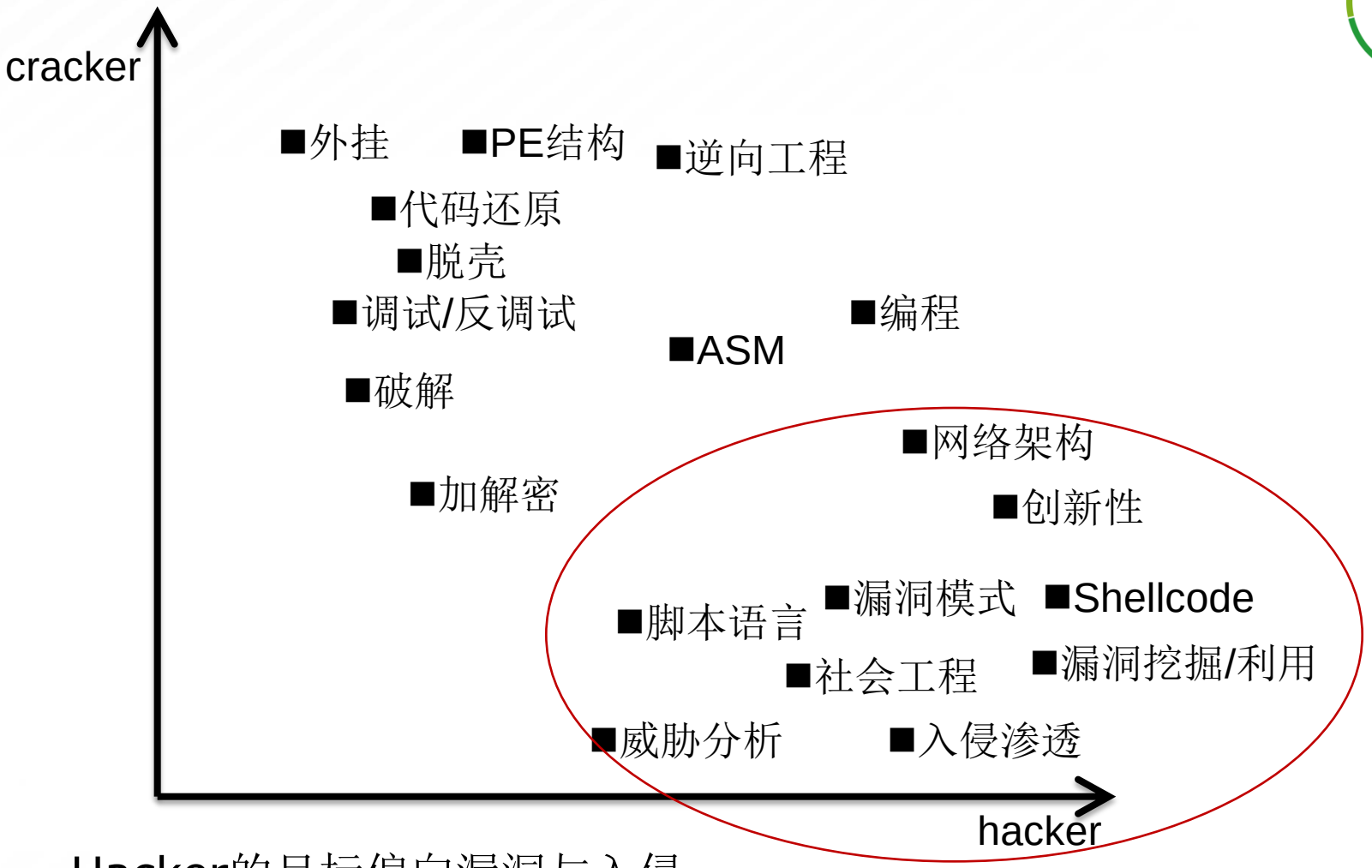
没有“漏洞挖掘 vs 反漏洞挖掘”

- 目前编译器/操作系统的重点放在了防止漏洞利用
- 反逆向工程可以阻止漏洞分析，但不能完全阻止漏洞挖掘
- SDL过程的目标是减少软件漏洞

反漏洞挖掘

- 反漏洞挖掘是在分析漏洞挖掘者（hacker）的能力与漏洞挖掘方法的基础上，采取的反制措施

软件攻击者的技能



- Hacker的目标偏向漏洞与入侵
- Cracker的目标偏向软件破解

比较

- Hacker：偏重技巧、思维和手段的巧妙
- Cracker：偏重扎实的逆向工程知识以及对系统内部运行机制的了解
- Cracker的技能更具有持久性，而Hacker的技术更容易失效
- Hacker转型到Cracker：较难
- Cracker转型到Hacker：较易

反漏洞挖掘设想

- 用反Cracker的手段对付Hacker
  - 反逆向工程
    - 对于Cracker，反逆向工程的目标是保护软件算法或完整性
    - 对于Hacker，反逆向工程的目标是防止漏洞挖掘
- 用Hacker自身的手段与技巧来对付Hacker
  - 迷惑、欺骗
  - 软件蜜罐

漏洞挖掘技术现状

- 设计架构审视
- 基于源代码
  - 源代码检测工具
  - 源代码人工检视
- 基于二进制
  - 人工分析
  - 补丁比对
  - 模糊测试
  - 动态符号执行与污点传播
  - 静态分析

反漏洞挖掘的对象是二进制程序

人工分析

- 漏洞假设法
  - 从外部数据输入入口处看后续的处理代码，结合漏洞触发模式，判断数据处理过程中的缺陷
  - 从潜在的漏洞代码（如strcpy、memcpy、rep movs、free）处向上回溯，检查导致漏洞的数据是否来自外部输入
- 盲测
  - 根据个人经验，篡改输入数据

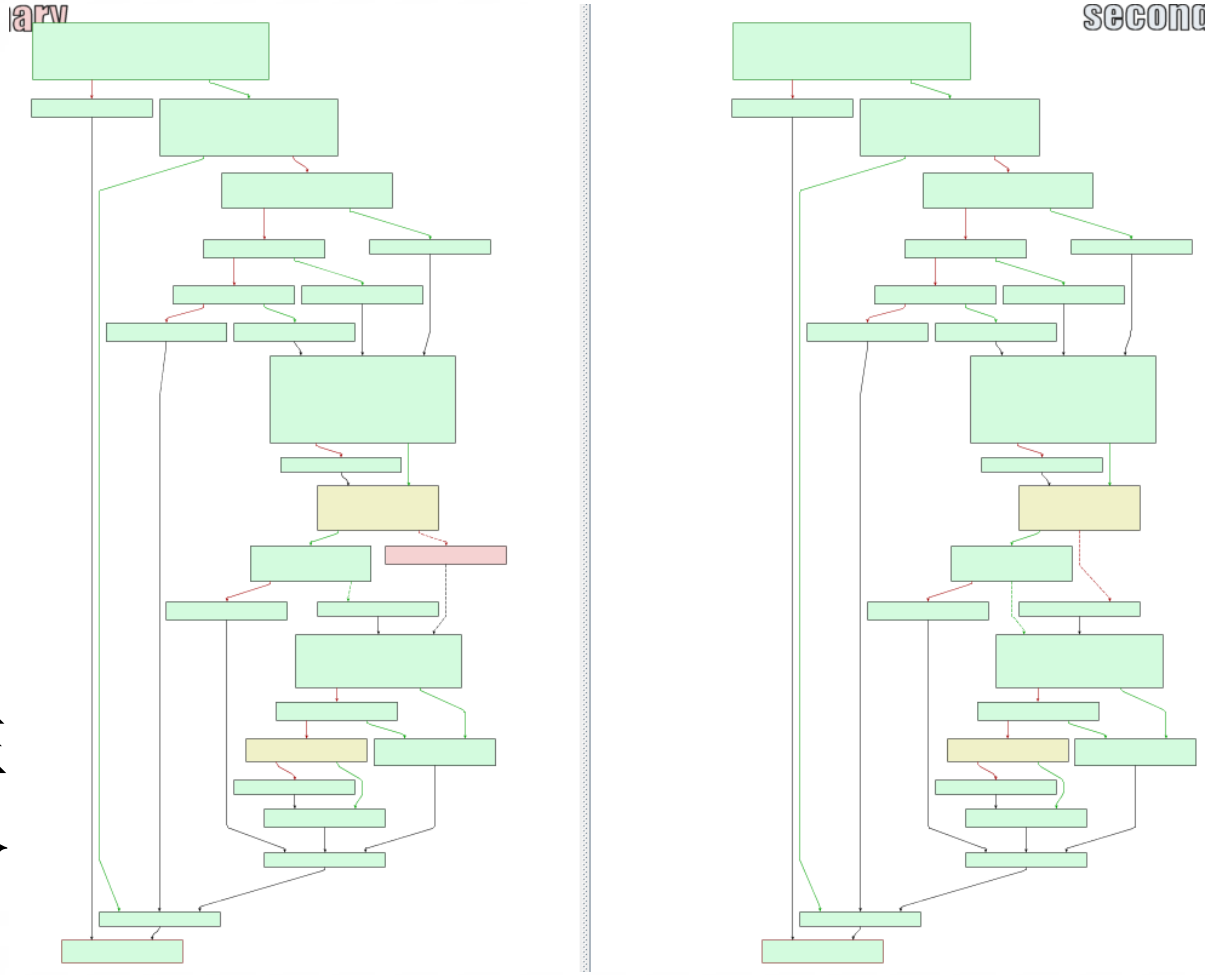
反人工分析

- 反逆向工程可以阻止绝大多数的人工二进制分析
- 对可执行文件进行处理，删除敏感函数，替换为等价函数，防止静态分析工具识别
- 编译器支持：编译过程中自动替换

补丁比对

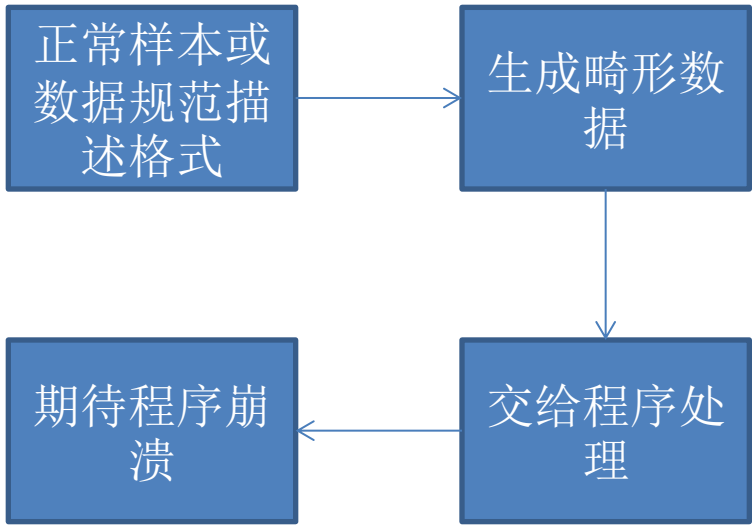
- 对补丁前后的二进制程序进行反编译，生成控制流图和函数调用图
- 根据文件结构、函数调用信息进行补丁前后函数间的对应
- 函数间的结构化比较，判断是否进行了修补

- 反补丁对比：在编译过程中插入垃圾分支



模糊测试

- 成本低，效果显著，受到黑客界普遍欢迎
- 模糊测试仍将被长期使用
- 瓶颈明显
- Codenomicon/MU/spike/peach/…




```
69 4D 53 47 00 0F 00 00
02 B1 00 55 00 00 00 00
00 50 2D 4A 38 39 FF 80
62 68 5F 70 65 61 63 68
5F 66 75 7A 7A 32 C0 80
35 39 C0 80 59 09 76 3D
31 26 6E 3E 64 31 74 38
75 69 70 38 6B 70 64 6C
6F 26 6C 3D 31 37 5F 66
34 30 32 37 5F 35 6B 70
70 73 2F 6F 26 70 3D 6D
32 54 63 7F 4D 44 55 2D
```

一段漏洞示例代码

```
struct Header
{
  ushort ver;
  ushort len; // 33
  char* data; // 33
}

int parse_msg(char* data)
{
…
char* msg_data = malloc(len);
strcpy(msg_data, data);
…
}
```

```
59 4D 00 21 41 4F 41 42
44 B1 45 55 56 0F 43 44
45 50 2D 4A 38 39 C0 80
62 68 5F 70 65 61 63 68
5F 66 75 7A 00 32 C0 80
35 39 C0 80 59 09 76 3D
31 26 6E 3D 64 31 74 38
75 69 70 38 6B 70 64 6C
6F 26 6C 3D 31 37 5F 66
34 30 32 37 5F 00 6B 70
70 73 2F 6F 26 70 3D 6D
32 54 63 78 4D 44 55 2D
```

Anti Fuzzing

- Fuzzing检测暗桩
  - 判断单位时间内程序重启次数
  - 父进程检测
  - 单位时间内同一IP连接次数
  - 函数返回值错误检测
- 在函数起始或结尾处加上判断函数

当Fuzzing被检测到...

- 引导到无关的代码分支上进行数据处理
- 耗时操作
- 给出虚假崩溃信息

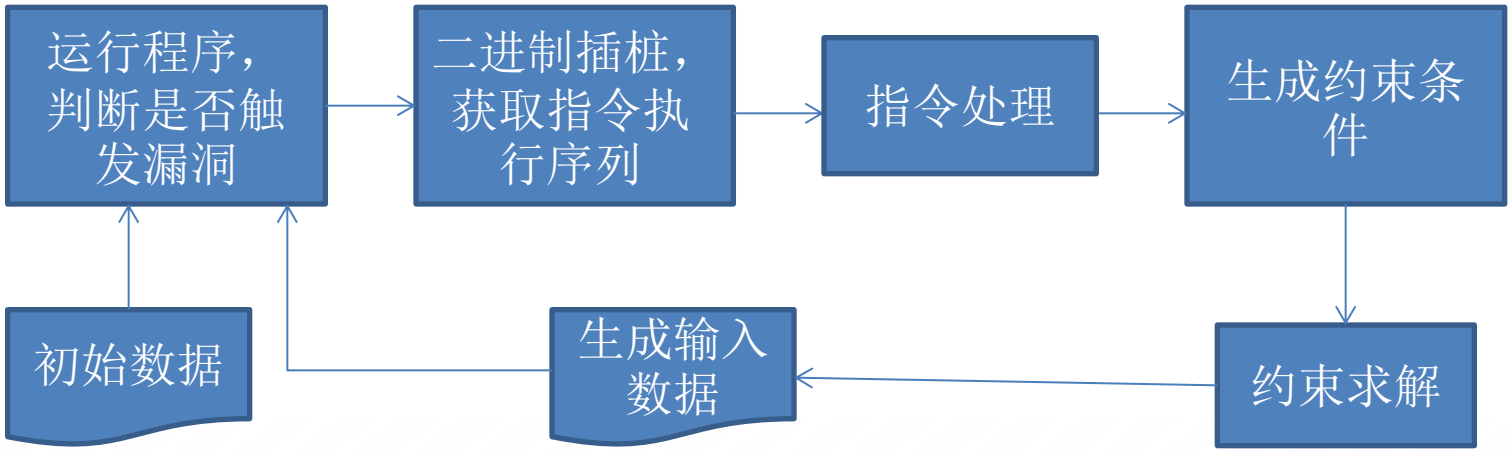
```
int parse_msg(char* data)
{
    if(check_fuzzing())
    {
        …
        char* msg_data = malloc(len);
        strcpy(msg_data, data);
        …
    }
    else
    {
        …
        fake code
        …
    }
}
```

编译过程中或直接在二进制文件中加入

智能动态分析

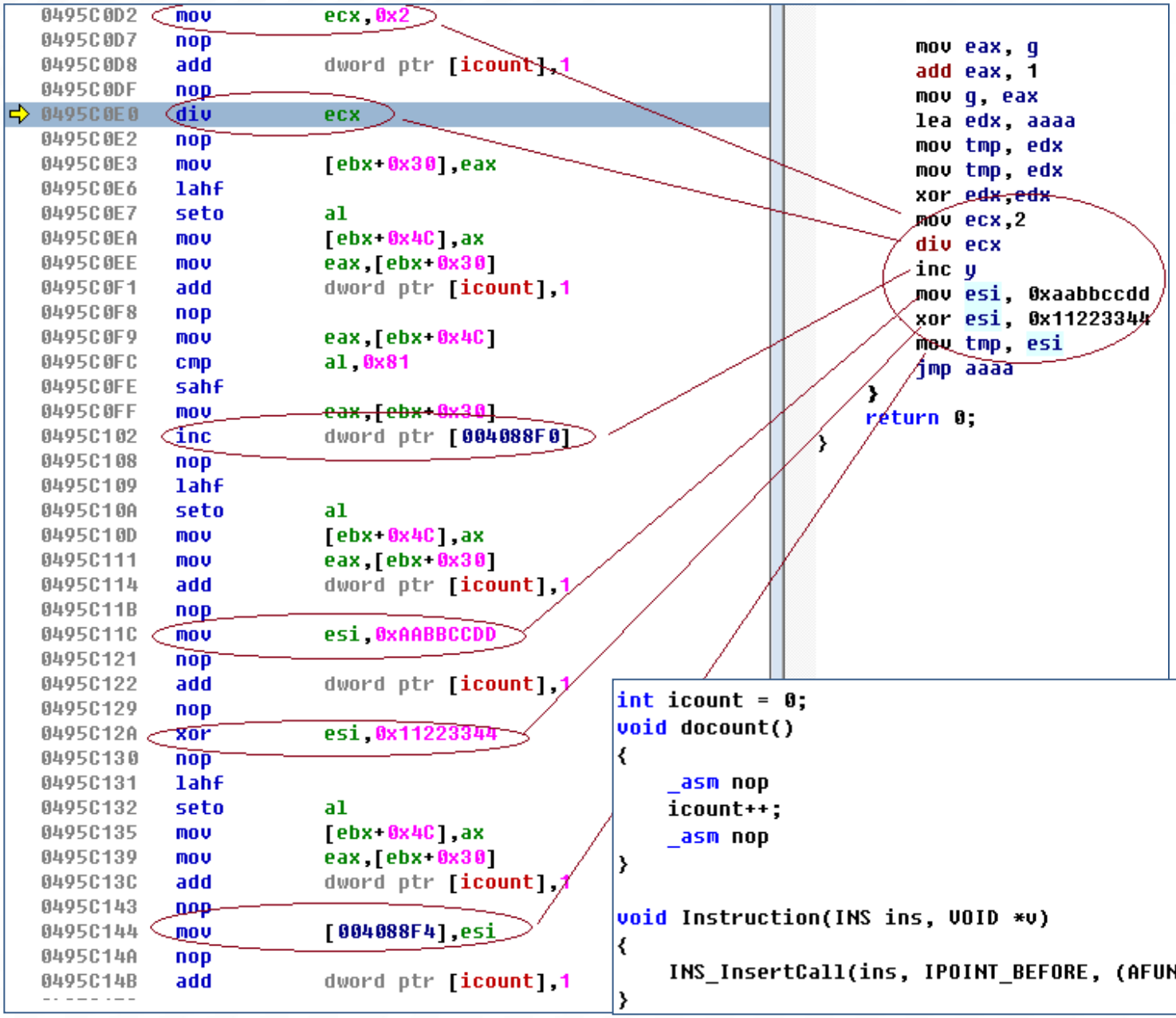
- 黑盒Fuzz效率低下，无法理解程序的运行，路径覆盖率低
- 白盒测试：计算输入数据之间的运算关系，构造输入数据，加大程序执行空间的路径覆盖度
- 二进制插桩(Dynamic binary instrumentation)
  - Pin
  - Valgrind
  - DynamoRIO
  - Nirvana
- 符号执行/约束求解
  - 以符号量作为程序输入，记录进入不同路径的条件，然后计算该条件并生成对应的数据
  - S2e
  - Klee
  - Fuzzball
  - Fuzzgrind
  - Sage

```
int parse_msg(int a, int b)
{
    …
    if(a == 600 && b == 800)
    {
        char* msg_data = malloc(len);
        strcpy(msg_data, data);
        …
    }
    else
    {
        …
    }
}
```



二进制插桩工具（Pin）

- 丰富的回调函数
  - 指令级
  - 基本块（Basic Block）
  - 函数调用
  - 模块加载
  - ……
- JIT
  - 在程序执行的第一条指令处开始插桩
  - 只对即将执行的基本块代码进行插桩
  - 代码修改检查
  - 半虚拟化执行，与目标程序处于同一个进程空间
- 高性能
  - 应用程序无感
  - 支持多线程/系统调用
  - 程序运行速度下降不明显



- 右侧：被插桩的代码
- 左侧：插桩以后的代码
- 下侧：安装插桩
- Ebx+30：保存当前线程的寄存器

```
int icount = 0;
void docount()
{
    _asm nop
    icount++;
    _asm nop
}

void Instruction(INS ins, VOID *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

Anti DBI

- 执行耗时
- 父进程检测（pin.exe）
- 模块枚举（pinvm.dll）
- 虚拟机内存检测/修改虚拟机内存
- Hook检测
  - KiUserApcDispatcher
  - KiUserCallbackDispatcher
  - KiUserExceptionDispatcher
  - LdrInitializeThunk
- EIP非代码区检测
- 代码自修改（导致虚拟机内存无限增加）
- 设置单步调试异常
- ……

Anti 符号执行 & 约束求解

- 符号执行的前提是代码未被反逆向工程保护
- 加入垃圾判断与分支
- 加入循环

总结

- 用黑客的手段对付黑客
- 增加漏洞挖掘成本,让挖掘者望而却步
- 在二进制层面完成,对开发者无感
- 可能影响性能和功能

Thanks!