软件漏洞的发现与利用

赵磊，博士，教授

leizhao@whu.edu.cn

武汉大学国家网络安全学院

提纲

- 漏洞分析与检测
  - 漏洞分析
  - 常见的漏洞挖掘技术
- 漏洞利用
  - 漏洞利用及Shellcode
  - 透明化的漏洞利用缓解机制

漏洞研究

- 漏洞挖掘
  - 人工代码审计、工具分析挖掘
- 漏洞分析
  - 漏洞机理、触发条件、漏洞危害
- 漏洞利用
  - 编制触发漏洞的POC，或者攻击者实施攻击目的的程序（Exploit）
- 漏洞防御
  - 从缺陷修补、热补丁、IDS升级
  - 通过系统自身机制阻断/缓解

漏洞类型

- 白帽子/黑客挖掘漏洞
  - 0-Day VUL
- 从公开发布的POC得到的漏洞
  - 1-Day VUL
- 从已发布的漏洞公告和漏洞补丁获得的漏洞
  - n-Day VUL

软件漏洞挖掘技术

- 源代码审计
  - 人工审计
  - 源代码分析工具
- Fuzzing
- 二进制对比

源代码安全审计

- 词法分析和语法分析
- 构建控制流图、数据流图审计危险函数
- 字符串操作函数
  - 内存拷贝函数
  - 文件操作
  - 数据库操作
  - 进程操作
  - 安全检查函数
  - ……

静态分析工具

- Fortify SCA- 多语言支持
  - 数据流引擎：跟踪,记录并分析程序中的数据传递过程所产生的安全问题
  - 语义引擎：分析程序中不安全的函数,方法的使用的安全问题
  - 结构引擎：分析程序上下文环境,结构中的安全问题
  - 控制流引擎：分析程序特定时间,状态下执行操作指令的安全问题
  - 配置引擎：分析项目配置文件中的敏感信息和配置缺失的安全问题
  - 特有的X-Tier™跟踪器：跨跃项目的上下层次,贯穿程序来综合分析问题

静态分析工具

- Findbugs-Java Bug pattern
  - 正确性：这种归类下的问题在某种情况下会导致bug，比如错误的强制类型转换等。
  - 最佳实践反例：这种类别下的代码违反了公认的最佳实践标准，比如某个类实现了equals方法但未实现hashCode方法等。
  - 多线程正确性：关注于同步和多线程问题。
  - 性能：潜在的性能问题。
  - 安全：安全相关。

软件漏洞挖掘技术

- 源代码审计
  - 人工审计
  - 源代码分析工具
- **Fuzzing**
- 二进制对比

Fuzzing

- 1989年Barton Miller在威斯康星大学提出
- 主要原理
  - 随机生成大量测试用例
    - 畸形用例
  - 监控被测程序的行为，检测Crash
- 工业界应用最广、贡献最大的自动化安全测试方法
  - 缓冲区溢出、跨站点脚本，格式漏洞、SQL 注入

Fuzzing

- 标准的 HTTP GET request
  - GET /index.html HTTP/1.1

- Fuzzing生成的畸形报文
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1.1

随机测试 vs. Fuzzing

- 随机测试
  - 随机生成大量正常的输入来测试程序
  - 主要目标是测试程序的正常逻辑与功能

- 模糊测试
  - 畸形输入
  - 主要目标：防止出现非预期的Crash或可利用漏洞

Fuzzing的分类

- 基于变异的模糊测试用例生成
  - 给定一点数量的初始用例（种子）
  - 通过随机变异种子里的某些字节来生成新的测试用例
- 基于规则的模糊测试用例生成
  - 目标：满足高结构化要求的输入
    - 大部分的输入都是结构化/高结构化的，格式复杂
    - 大部分的程序都会校验格式的合法性

基于变异的测试用例生成

- 优势
  - 所依赖的前期知识少、自动化程度高
  - 简单、性能高、Fuzzer的吞吐量高
    - AFL： PC吞吐量可以上千

- 劣势
  - 频率非常高的“随机”
  - 理论上，生成符合格式的结构化输入的概率很低
  - 初始种子很重要

基于生成的模糊测试

- 基于RFC/文档等定义的规则生成测试用例
  - 建立特定的协议/输入格式规则
  - 工业界：基于模板的用例生成
  - 代表性工具：SPIKE by Immunity、peach、boofuzz等
  - 定义可疑点，增加输入中的异常部分
  - 完整、能够应对复杂格式

模糊测试实践

- 阅读AFL的源代码:
  - AFL　http://lcamtuf.coredump.cx/afl/
  - 理解源代码插桩模式
    - 需要对被测程序对象做一定的修改
    - 在条件跳转等语句插入标记
  - 理解二进制工作模式
    - 直接工作在二进制程序上，无需源代码支持
    - 利用qemu模拟器来运行被测试的目标程序
  - 能够使用AFL对可执行文件进行测试
    - 针对ubuntu下的常见程序，如PowerDNS, Bind和dnsmasq

软件漏洞挖掘技术

- 源代码审计
  - 人工审计
  - 源代码分析工具
- Fuzzing
- **二进制对比**

BinDiff

- 比较两段二进制代码是否相似/逻辑等价 -> 找出不同
  - 两个二进制文件(可执行文件、目标文件、库文件等)比对
  - 粒度：函数，基本块，指令或自定义切分的粒度等
- 应用场景
  - 恶意代码/软件分析
  - 基于patch的漏洞发现（Nday），patch存在性检测
  - 著作权侵权(盗版)检查
  - 通过提取归纳常见的漏洞类型特征，进而来进行 0-day 漏洞挖掘

背景

- 软件补丁是修复漏洞的重要机制
- 安全补丁包含相关漏洞的重要信息
  - 分析安全影响
  - 构造热补丁
  - “N-day” 漏洞攻击

二进制对比工具PatchDiff

- PatchDiff
  - IDA Pro中的一个插件（不再维护）
  - 本质是bindiff

```
        if (argv[i][0] == '-') {
            if (!strncmp(argv[i], "-prestring", 10)) {
                nflags++;
-               ret = sscanf(argv[i] + 1, "prestring=%s", buf);
+               ret = sscanf(argv[i] + 1, "prestring=%490s", buf);
                if (ret != 1) {
                    fprintf(stderr, "parse failure for prestring\n");
                    return 1;
                }
```

CVE-2018-7186 补丁源代码

08048E3C	mov	eax, ss:[ebp+command]		08048E3C	mov	eax, ss:[ebp+command]
08048E3F	mov	ss:[esp+4], 0x80493A6		08048E3F	mov	ss:[esp+4], 0x80493A6
08048E47	mov	ss:[esp+8], eax		08048E47	mov	ss:[esp+8], eax
08048E4B	mov	eax, ss:[ebp+str]		08048E4B	mov	eax, ss:[ebp+str]
08048E4E	mov	eax, ds:[eax+ebx*4]		08048E4E	mov	eax, ds:[eax+ebx*4]
08048E51	add	eax, bl 1		08048E51	add	eax, bl 1
08048E54	mov	ss:[esp], eax		08048E54	mov	ss:[esp], eax
08048E57	call	.__isoc99_sscanf		08048E57	call	.__isoc99_sscanf

BinDiff 二进制对比结果

提纲

- 漏洞分析与检测
  - 漏洞分析
  - 常见的漏洞挖掘技术
- **漏洞利用**
  - 漏洞利用及Shellcode
  - 透明化的漏洞利用缓解机制

通用漏洞评分系统（CVSS）

- 通用漏洞评分系统（CVSS）
  - NIAC开发、FIRST维护
  - 开放并且能够被产品厂商免费采用的标准
  - 评测漏洞的严重程度
  - 帮助确定所需反应的紧急度和重要度。
- CVSS得分最大为10，最小为0
  - 得分7~10的漏洞通常被认为比较严重
  - 得分在4~6.9之间的是中级漏洞
  - 0~3.9的则是低级漏洞

漏洞利用的思路

- 利用目标:
  - 修改内存变量（邻接变量）
  - 修改代码逻辑（代码的任意跳转）
  - 修改函数的返回地址
  - 修改函数指针(++)[虚函数]
  - 修改异常处理函数指针（SEH）
  - 修改线程同步的函数指针

漏洞利用的思路

- 利用过程
  - 定位并分析漏洞
    - 利用静态分析和动态调试确定漏洞机理
    - 如堆溢出、栈溢出、整数溢出的数据结构，影响的范围
  - 按照利用目的，编写Shellcode
  - 溢出并绕过防御
  - 控制程序跳转或代码指针，使得Shellcode获得可执行权

漏洞利用（Exploit）结构

- Exploit
  - 利用漏洞实现Shellcode 的植入和触发的过程
  - Exploit ≈ Payload+Shellcode
- Payload：部署漏洞触发和利用的基本数据
  - 触发漏洞
  - 绕过一定的系统防御
  - 跳转到Shellcode
- Shellcode： 执行恶意功能的一段指令
- Payload与漏洞关联，Shellcode独立于漏洞

Shellcode设计

- 什么是shellcode
  - 缘起：1996年，出现在Aleph One的论文“Smashing the Stack for fun and profit”
  - 原义：the code to spawn a shell
  - 指能完成特殊任务的自包含的二进制代码，根据不同的任务可能是发出一条系统调用或建立一个高权限的Shell，Shellcode也就由此得名
  - 延伸：攻击者植入目标内存中的代码片段

Shellcode典型功能

- 正向连接（目标主机开一个服务端口）
- 反向连接（目标主机主动连接攻击者的控制端）
- 下载并执行程序
- 动态生成可执行程序并执行
- 执行一个程序
- 打开Shell
- 消息弹框
- ……

Shellcode设计流程

- 编写shellcode(高级语言)
- 反汇编该shellcode（调试)
- 从汇编级分析程序执行流程
- 生成完成的Shellcode(机器代码)
- 优化Shellcode（编码、长度)
- 适配漏洞

Shellcode编写语言

- 汇编语言
  - 代码短小，可控性强
  - 但耗时或对语言精通
- C语言
  - 效率高，便于入手
  - 但需要调整

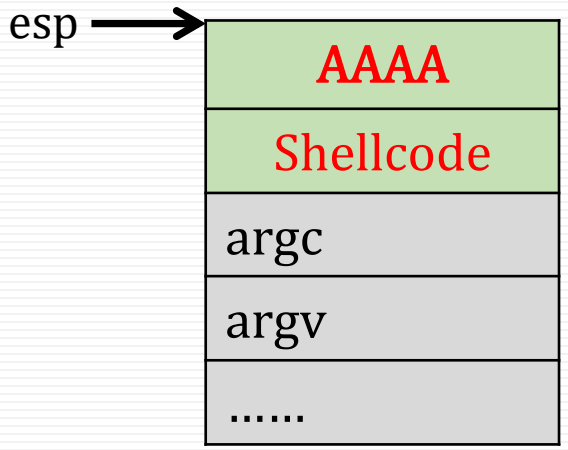
缓冲区溢出的漏洞利用

- 最典型的
  - 利用构造数据填充栈
  - 栈上插入指令，如exec(“/bin/sh”)
  - 修改返回地址
  - 跳转到shellcode

代码注入攻击

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



leave:

mov %ebp, %esp

pop %ebp

ret

%ebp = AAAA

%eip 指向shellcode

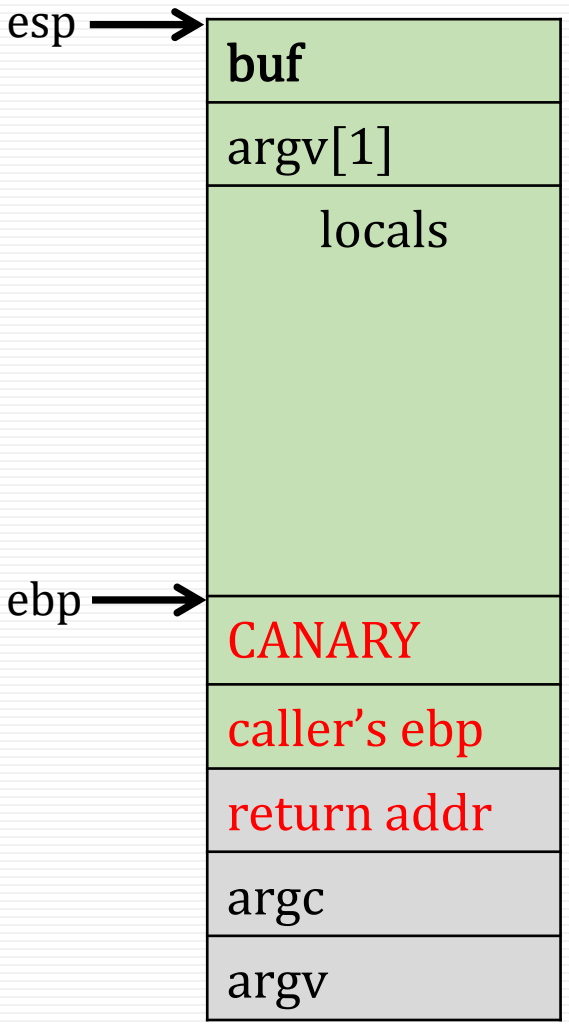
漏洞利用的缓解技术

- 攻击缓解技术
  对程序透明化的保护技术
- Canaries
- DEP(数据执行保护)/NX(禁止运行)
- ASLR(地址空间布局随机化)

Canary/Stack cookie

StackGuard [cowen etal 1998]

- 核心思想
- 将随机生成的canary(探测值)插入到返回地址和局部变量之间
- 在函数返回之前检查canary
- 错误的canary→溢出



Data Execution Provention(DEP)/ No eXecute(NX)

DEP的思想

- 最典型的缓冲区溢出漏洞是把shellcode注入到栈上
  - 栈其实是用来保存函数运行时的栈平衡、局部变量等数据的
  - 数据并不会作为代码

  - DEP：将栈所在的内存置为不可执行
  - 不会执行恶意代码
  - 会导致Crash

DEP

- DEP的全称是“Data Execution Prevention”，是微软随Windows XP SP2和Windows 2003 SP1的发布而引入的一种数据执行保护机制。
  - 2003.9：AMD CPU开始支持NX，即“No eXecute”
  - 随后：Intel CPU开始支持“Execute Disable”（或EDB）或“XD-bit”
  - 2004年8月6日：XP SP2推出， 支持DEP

数据执行保护-DEP

- DEP的实现机理是把堆栈的页属性设置为NX
- Windows中的DEP选项
  - Optin：默认仅将DEP保护应用于Windows系统组件和服务，具备NX标记的程序自动保护（个人版默认）
  - Optout：为排除列表程序外的所有程序和服务启用DEP（服务器版默认）
  - AlwaysOn：对所有进程启用DEP 的保护，不能关闭
  - AlwaysOff：对所有进程都禁用DEP，不能启动

思考

- 在实施DEP（数据执行保护）之后如何构造Shellcode
  - Ret2Libc
  - ROP （Return - Oriented Programming）
  - JOP
  - COP 等

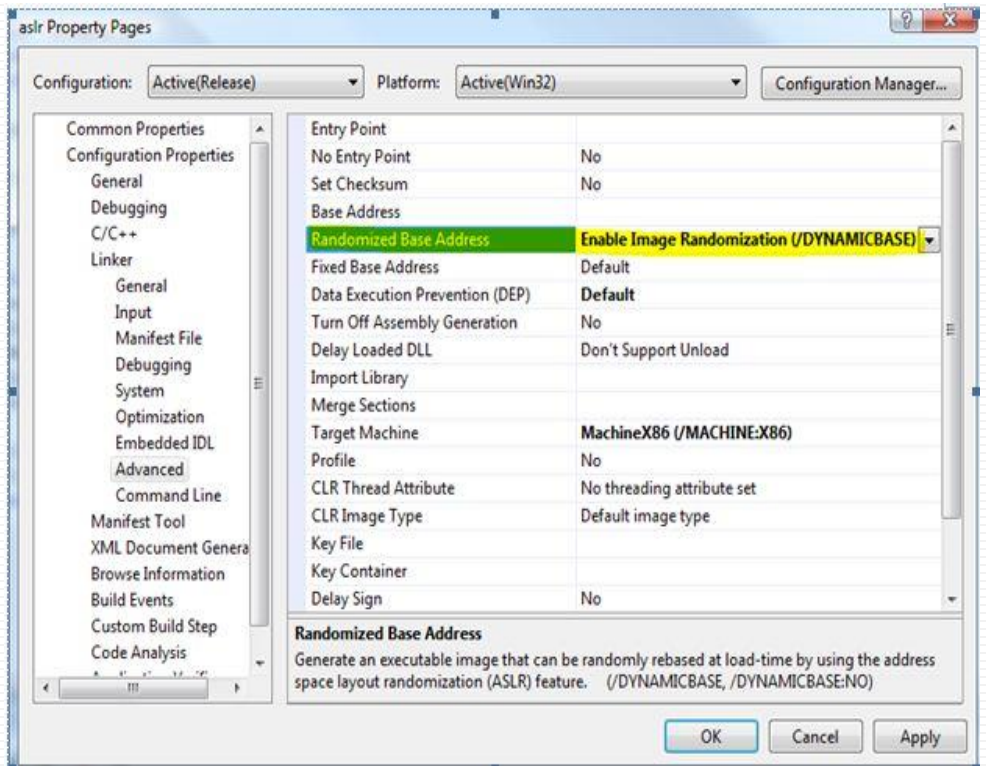
Address Space Layout Randomization (ASLR)

ASLR地址空间布局随机化

- 传统的利用方法需要准确的地址
  - 栈溢出：shellcode位置
  - return-to-libc： 库函数位置
  - ROP: gadget的位置
- 虚拟内存环境下，程序各段的布局是固定的
  - 堆、栈、库等
- 解决办法：将各个区域的地址随机化

地址空间布局随机化-ASLR

- 部署Shellcode
  - 知晓堆或者栈的地址
  - 借用程序自身或者库中的代码（如jmp ESP-ROP）
- ASLR的思想
  - 栈和堆的基址是加载时随机确定的
  - 程序自身和关联库的基址是加载时随机确定的



地址空间布局随机化-ASLR

- Windows中的ASLR
  - 从Visual Studio 2005 SP1开始，增加了/dynamicbase链接选项。
  - /dynamicbase选项设置
  - Project Property -> Configuration Properties -> Linker -> Advanced -> Randomized Base Address

ASLR示例

```
int main( int argc, char* argv[] )
{
   HMODULE hMod = LoadLibrary( L"Kernel32.dll" );
   char StackBuffer[256];

    void* pvAddress = GetProcAddress(hMod, "LoadLibraryW");
   printf( "Kernel32 loaded at %p\n", hMod );
   printf( "Address of LoadLibrary = %p\n", pvAddress );
   printf( "Address of main = %p\n", main );

   foo();
   printf( "Address of g_GlobalVar = %p\n", &g_GlobalVar );
   printf( "Address of StackBuffer = %p\n", StackBuffer );

   if( hMod )
     FreeLibrary( hMod );

   system(“pause”);
   return 0;
}
```

操作系统重启前后

	重启前	重启后
kernel32 基址	0x776D0000	0x76780000
Loadibrary 函数地址	0x777228D2	0x767D28D2
Main 函数地址	0x00DA1020	0x013B1020
Foo 函数地址	0x00DA1000	0x013B1000
全局变量 g_GlobalVar 的地址	0x00DA336C	0x013B336C
StackBuffer 数组地址	0x0021FC28	0x001BFDC0

ASLR的绕过

- 暴力猜解
- 并非全部代码都是随机化的
- 内存泄露+ROP链构造
- GOT表劫持

谢谢