程序开发环境安全

--连接器中的漏洞

余科技，赵伟
启明星辰积极防御实验室

X'con 2005

前言

- 这是一篇关于讨论程序开发环境中连接器漏洞相关安全的演讲。
- 这是一个我们可能平时忽视的问题，大家考虑过日常使用开发环境时可能会遇到的安全问题吗？

为什么要研究开发环境的安全

- 起因：

  在日常的使用中发现多个开发环境的安全漏洞，却从来没有真正关注过开发环境的安全问题。提供源程序的软件都被认为是安全的，没有人在重新编译连接前会想到编译或者连接过程本身将导致自己受到攻击。

环境

- 通常的程序开发环境可能包括（以**Intel x86**下的**Windows**与**Linux**平台为主）：
  - 编译器：**CL**，**gcc**
  - 连接器：**Link, ld**
  - 调试器：**VC**自带的调试器，**GDB**
  - **Etc.**

重点

- 连接器安全是这个**ppt**的重点
- 什么是连接器，连接器的作用
- 连接器存在的漏洞
- 下面我们讲一个**linux**下**ld**的漏洞实例

连接器

- 为了深入了解该漏洞，我们先来描述一下连接器流程
- 连接器工作的一般过程：

  当连接器运行时，它首先会扫描输入文件，找出每个段的大小，收集所有符号的定义和引用。连接器会建立一个段表用来记录输入文件中所有的段，一个符号表记录所有导入或导出的符号。
  - 第一遍扫描
  - 第二遍扫描

Linux平台下

- 大家可能都有过的经历：

  经常碰到**Gcc**在编译的时候出现崩溃**(Segfault)**的情况，去年发现一个相关漏洞，当时以为在**gcc**当中进行调试时很困惑。

- 真正的原因：**ld**使用了**bfd**库，**bfd**库中存在多个漏洞

ld连接流程

- **Ld**简介：通常是编译程序的最后一步
- **Ld**连接流程

抽象层bfd库简介

- **Bfd**库的作用：为**ld**提供一个通用界面，使**ld**可以对各种格式的目标文件进行相同的操作
- **Bfd**库工作流程概要：
  - 打开目标文件后，自动检测文件格式
  - 对应处理对象和例程
- **Bfd**库对**elf**文件的解析

Elf文件格式简介1 ELF文件头

- ELF文件头

```
char magic[4] = "\177ELF";// magic number
char class; // address size， 1 = 32 bit， 2 = 64 bit
char byteorder; // 1 = little-endian， 2 = big-endian
char hversion; // header version， always 1
char pad[9];
short filetype; // file type: 1 = relocatable， 2 = executable，
    // 3 = shared object， 4 = core image
short archtype; // 2 = SPARC， 3 = x86， 4 = 68K， etc.
int fversion; // file version， always 1
int entry; // entry point if executable
```

Elf文件格式简介2 ELF文件头

接上页

```
int phdrpos; // file position of program header or 0
int shdrpos; // file position of section header or 0
int flags; // architecture specific flags， usually 0
short hdrsize; // size of this ELF header
short phdrent; // size of an entry in program header
short phdrcnt; // number of entries in program header or 0
short shdrent; // size of an entry in section header
short shdrcnt; // number of entries in section header or 0
short strsec; // section number that contains section name
    strings
```

Elf文件格式简介3 Section头

- **Section头部**

```
int sh_name; // name， index into the string table
int sh_type; // section type
int sh_flags; // flag bits， below
int sh_addr; // base memory address， if loadable， or zero
int sh_offset; // file position of beginning of section
int sh_size; // size in bytes
int sh_link; // section number with related info or zero
int sh_info; // more section-specific info
int sh_align; // alignment granularity if section is moved
int sh_entsize; // size of entries if section is an array
```

Bfd漏洞背景

- **2004**年我们就发现了**ld**的崩溃，被忽略没有深入研究**L**（不要忽略你身边的小细节）。
- **2005**年**6**月**1**日，**gentoo**代码审核小组公布该漏洞，影响较大。

bfd解析elf漏洞 1

- 一个恶意构造的**elf**文件头

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  94 80 04 08 34 00 00 00  |............4...|
00000020  26 00 00 00 00 00 41 41  41 41 41 41 41 41 41 41  |&.....AAAAAAAAAA|
00000030  00 00 41 41 41 41 41 41  41 41 00 00 00 40 41 41  |..AAAAAAAA...@AA|
00000040  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
```

bfd解析elf漏洞 2

- Elfcode.h文件elf_object_p函数当中：

```
bfd_set_start_address (abfd, i_ehdrp->e_entry);
  if (i_ehdrp->e_shoff != 0)
    {
     if (bfd_seek (abfd, (file_ptr) i_ehdrp->e_shoff, SEEK_SET) != 0)  //通过
     shoff找到section表头
     goto got_no_match;
     if (bfd_bread (&x_shdr, sizeof x_shdr, abfd) != sizeof (x_shdr))// 读取第一
     个section头
     goto got_no_match;
      elf_swap_shdr_in (abfd, &x_shdr, &i_shdr); // 转化成internal
     if (i_ehdrp->e_shnum == SHN_UNDEF) // 如果为section数为0，使用第一个
     section里面的大小
     i_ehdrp->e_shnum = i_shdr.sh_size;
     if (i_ehdrp->e_shstrndx == SHN_XINDEX)
     i_ehdrp->e_shstrndx = i_shdr.sh_link;
    }
```

bfd解析elf漏洞 3

接上页

```
/* 为internal形式的section报头表申请空间*/
 if (i_ehdrp->e_shnum != 0)
   {
     Elf_Internal_Shdr *shdrp;
     unsigned int num_sec;
     amt = sizeof (*i_shdrp) * i_ehdrp->e_shnum; // 其实是i_shdr.sh_size的大小
     i_shdrp = bfd_alloc (abfd, amt); // 分配amt大小的内存
     if (!i_shdrp)
      goto got_no_match;
     num_sec = i_ehdrp->e_shnum; // i_shdr.sh_size
     if (num_sec > SHN_LORESERVE)
      num_sec += SHN_HIRESERVE + 1 - SHN_LORESERVE; //0xffff-0xff00+1
     elf_numsections (abfd) = num_sec;
     amt = sizeof (i_shdrp) * num_sec; // 整数溢出
     elf_elfsections (abfd) = bfd_alloc (abfd, amt); // 分配错误的内存大小，后面对该内存区域的
     操作导致溢出
```

威胁

- **Bfd**是一个抽象层,影响巨大：很多开发相关的工具都会使用到，如：**objdump, gdb…etc.**
- 因为该漏洞没有涉及到内核部分，不会直接被用来做本地权限提升（不排除有使用**bfd**库的**setuid**程序）。
- 可能会用作针对程序开发人员的攻击**J**

小结

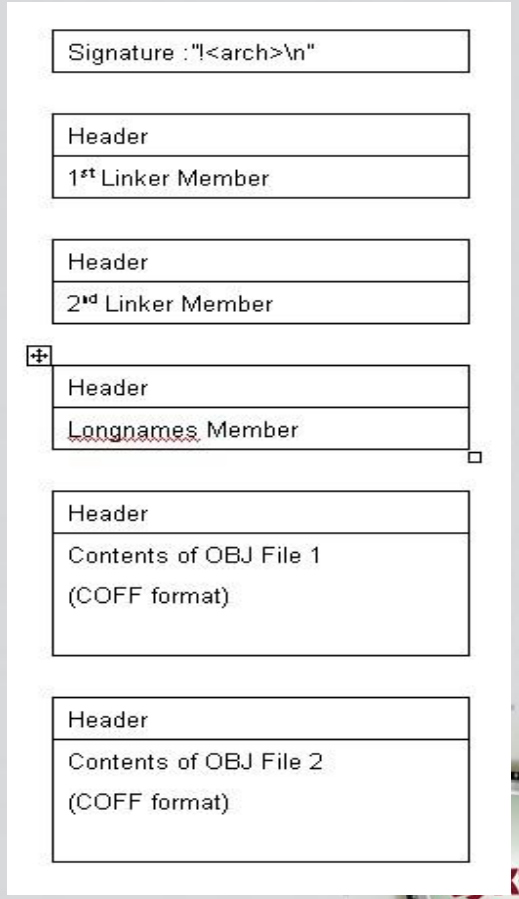
- **Reliable linkers never crash J**
- 这是一个典型的整数溢出攻击
- 修补方法建议
  - **Gentoo的patch**方案：检查逻辑正确性
  - **GUN的patch**方案：检查整数溢出

Windows平台

- **VC 6**中一个未公开的漏洞**J**（由**keji**发现）
- 与**ld**出现的问题类似
- 下面首先简单介绍**lib**文件格式

Lib文件格式简介

- **Lib**文件和**obj**文件的区别
- **Lib**文件的格式：
  - **Lib**文件头
  - 节（**section**）



Lib文件格式简介

- 第一节的结构如下：

```
typedef struct {
        unsigned long SymbolNum;          // 库中符号的数量
        unsigned long SymbolOffset[n];    // 符号所在目标节的偏移
        char StrTable[m];                 // 符号名称字符串表
 }FirstSec;
```

Link.exe漏洞

- 以下数据为文件头及节的数据，偏移**0x44**即为节的开始，前**4**个字节为符号数量**0x40000007**

```
000000  21 3C 61 72 63 68 3E 0A 2F 20 20 20 20 20 20 20  !<arch>./
000010  20 20 20 20 20 20 20 20 31 31 30 33 30 30 35 35          11030055
000020  34 35 20 20 20 20 20 20 20 3C 20 20 20 37 20 20  45       <   7
000030  30 20 20 20 20 20 20 20 32 34 33 20 20 20 20 20  0       243
000040  20 20 60 0A 40 00 00 07 00 00 02 62 00 00 02 62    `.@......b...b
000050  00 00 02 62 00 00 02 62 00 00 02 62 00 00 02 62  ...b...b...b...b
000060  00 00 02 62 3F 3F 5F 43 40 5F 30 42 41 43 48
    40  ...b??_C@_0BACH@
000070  4A 4B 49 43 40 43 43 AD 43 43 43 43 43 43 BD 43
    JKIC@CC.CCCCCC.C
000080  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43
    CCCCCCCCCCCCCCCC
000090  43 43 43 43 43 40 00 3F 3F 5F 5B 40 5F C5 48 41
    CCCCC@.??_[@_.HA
000040  48 40 45 42 42 43 40 41 41 41 55 41 41 41 41 41
    H@EBBC@AAAUAAAAA
```

Link.exe对section的处理

- **LINK.EXE** 处理节时，首先分配一段内存，用于保存符号名称字符串表，内存大小 **=** 符号数量 *** sizeof(char*)**代码如下：

```
0045FB54   mov      edx,[ebx+0x18] ; [EBX+18] 保存符号的数量
0045FB57   shl      edx,2          ; 符号数量 * 4
0045FB5A   push     edx
0045FB5B   call     00451B20       ; 分配内存 (malloc)
0045FB60   mov      edx,[ebx+0x18]
0045FB63   xor      ecx,ecx
0045FB65   mov      [ebx+0x28],eax ;保存分配到的内存地址
```

漏洞定位

- 以下代码计算所有的符号名称字符串地址，并把这些地址填充到上面分配到的内存中：

```
0045FB6F   mov      edx,[ebx+0x28]        ;分配的内存地址
0045FB72   mov      [edx+ecx*4],eax       ;保存符号字符串到内存
0045FB75   mov      dl,[eax]
0045FB77   inc      eax
0045FB78   test     dl,dl
0045FB7A   jz       0045FB83
0045FB7C   mov      dl,[eax]              ;定位下一个符号名称字符串
0045FB7E   inc      eax
0045FB7F   test     dl,dl
0045FB81   jnz      0045FB7C
0045FB83   mov      edx,[ebx+0x18]        ;[ebx+0x18]为符号数量
0045FB86   inc      ecx
0045FB87   cmp      ecx,edx               ;如果已计算的符号数量小于符号总数量，则继续进
    行计算
0045FB89   jb       0045FB6F
```

C语言描述

- 这是一个非常“清晰”的漏洞，是否还存在这种类型的漏洞呢？J
- 以C 语言进行描述以上的代码：

```
DWORD* pTable = (DWORD *)malloc( SymbolNum * 4);
//这里会出现整数溢出
for(int i=0; i<SymbolNum; i++)
{
    pTable[i] = 本次计算的符号字符串地址;
}
```

Bfd新漏洞

- 我们在**bfd**库里发现了一个新漏洞
- 漏洞原理与**VC6**的这个十分相似
- 漏洞代码在**bfd**库**Archive.c**文件中

Bfd新漏洞

在**do_slurp_coff_armap**函数中：

```
static bfd_boolean
do_slurp_coff_armap (bfd *abfd){
…

  carsym_size = (nsymz * sizeof (carsym));  //used the nsymz from file
  ptrsize = (4 * nsymz);  //integer overflow here
…
/* Allocate and read in the raw offsets.  */
  raw_armap = bfd_alloc (abfd, ptrsize);  // allocate wrong memory size here
  if (raw_armap == NULL)
    goto release_symdefs;
…
}
```

威胁

- 程序员仍是首要目标J，是否有人不写程序也用vcJ？
- 类似于其它文件格式漏洞，需要投递的方法。
- 可以隐藏在一些开源项目代码里，进行攻击（新口号：不要编译陌生人的代码哦J）。

疑问

- 疑问：为什么总是出现类似问题？
  - 是否我们忽略了一些本质问题？
  - 为什么是整数溢出漏洞？
  - 此类漏洞本质是什么？
  - 仍然此类存在漏洞？

真正的原因

- 通常安全编程强调：不要相信用户的输入
- 思维扩展：不要相信用户的输入，同样不要相信文件的输入，文件同样是用户的输入！
  - 包括开发工具中使用的配置文件
  - 包括开发工具中使用的工程文件
  - 包括开发工具中使用的**makefile**文件
  - 这些文件都有可能隐藏恶意程序

安全提示

- 我们提出两条全新的安全提示：
  - 不要相信用户输入，包括文件的输入。
  - 不要编译陌生人的源代码或者类似的工程文件，它们同样不安全。
- 希望大家可以进行更多的扩展。

总结

- 这是一个新的课题，会在未来遇到吗？
- 我的几点想法，希望与大家一同讨论：
  - 这些安全问题对程序开发人员会有什么威胁？
  - 如果程序员使用的开发工具存在漏洞会在程序开发过程中导致什么后果？
  - 可能会产生哪些攻击形式？能否利用这种攻击植入木马？

问题/讨论

Thanks

安全源自未雨绸缪

Venus Info Tec Inc.

Security

Trusted {Solution} Provider

Services