

2019-2020学年秋季学期

漏洞利用与攻防实践

*Exploiting Software Vulnerability-
Techniques and Practice*

讲述人：曹旭栋、刘鹏

材料准备人：刘鹏、曹旭栋

漏洞利用与攻防实践

Exploiting Software Vulnerability-Techniques and Practice

[第2次课] 格式化字符串漏洞

提 纲

1. 背景介绍
 - 格式化字符串简介
 - 栈与格式化字符串的关系
2. Format String Vulnerability介绍
3. 如何利用Format String Vulnerability
 - 我们能控制什么?
 - 如何进行攻击?
4. 如何防范Format String Vulnerability

1、背景介绍

	<i>Buffer Overflow</i>	<i>Format String</i>
public since	mid 1980's	June 1999
danger realized	1990's	June 2000
number of exploits	a few thousand	a few dozen
considered as	security threat	programming bug
techniques	evolved and advanced	basic techniques
visibility	sometimes very difficult to spot	easy to find

表1: Buffer Overflows vs. Format String Vulnerabilities

格式化字符串漏洞和普通的缓冲溢出有相似之处,但又有所不同,它们都是利用了程序员的疏忽大意来改变程序运行的正常流程。格式化字符串漏洞的历史要比缓冲区溢出短得多,而且一般被认为是程序员的编程错误。但是格式化字符串漏洞攻击可以往任意地址写任意内容,它的危害也是非常致命的。

1、背景介绍

格式化字符串简介

○什么是格式化字符串？

- 格式化字符串就是按一定格式输出的字符串，字符串包含文本内容以及格式化符号。

格式化字符串中有一些常见的格式化符号，分别对应(C语言中)不同的参数类型

格式符	含义	含义 (英)	传
%d	十进制数 (int)	decimal	值
%u	无符号十进制数 (unsigned int)	unsigned decimal	值
%x	十六进制数 (unsigned int)	hexadecimal	值
%s	字符串 ((const) (unsigned) char *)	string	引用 (指针)
%n	%n 符号以前输入的字符数量 (* int)	number of bytes written so far	引用 (指针)

表2：格式符说明

格式化字符串简介

○什么是格式化函数？

- 格式化函数指的是一类特殊的ANSI C函数，这类函数的参数中含有格式化字符串，其功能是按照格式化字符串的形式将其他的C类型参数变为人可以阅读的表现形式输出。格式化函数调用时包括格式化字符串在内的参数会被压入栈中。

常用的格式化函数：

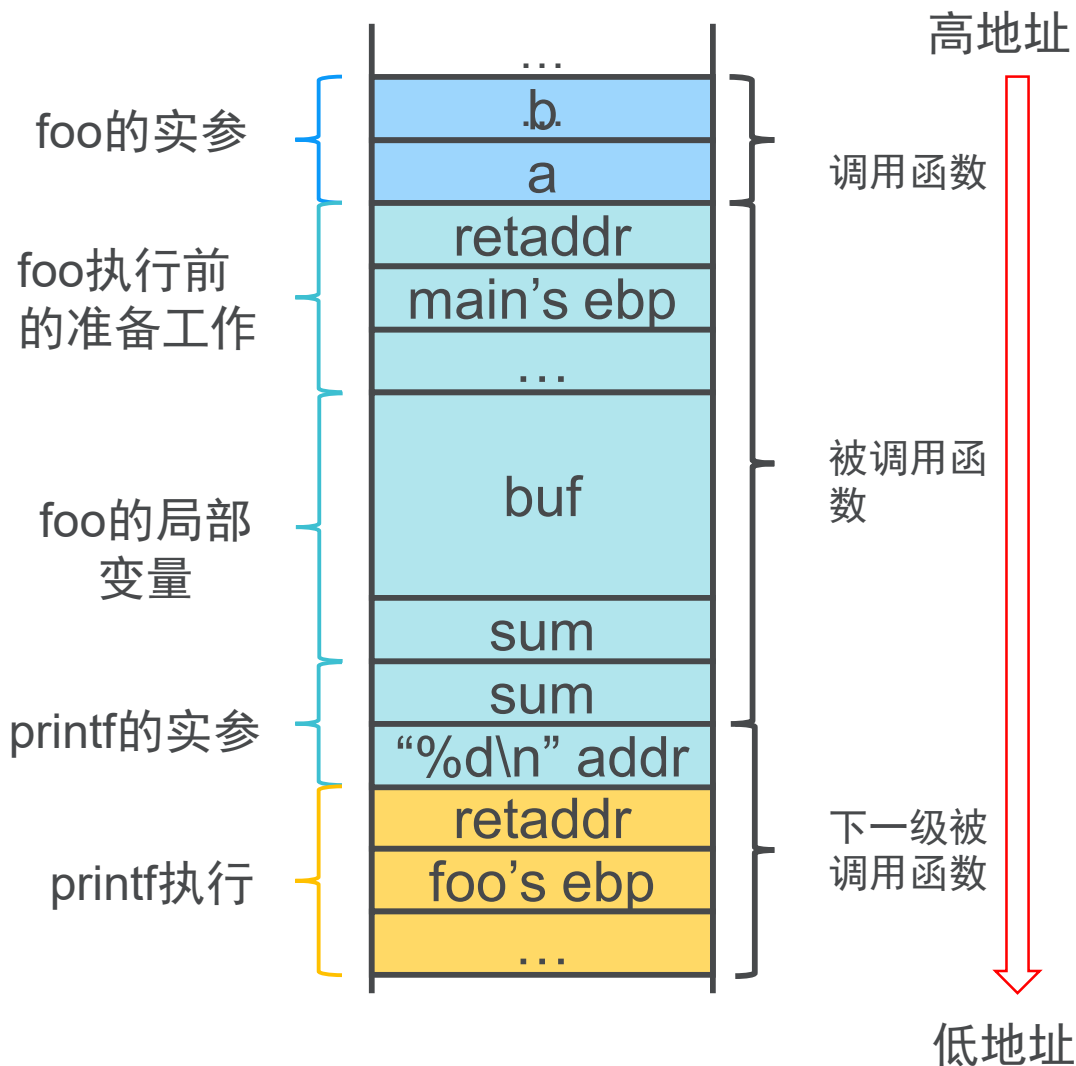
- printf —— 标准输出，在屏幕上打印出一段字符串来。
- fprintf —— 把格式化的数据写入到某个文件流中。
- sprintf —— 把格式化的数据写入到某个字符串中，返回值为字符串长度。
- snprintf —— 向一个字符串输出，并带有长度检查机制
- syslog —— 向系统记录输出

1、背景介绍

栈与格式化字符串的关系

```
void foo(int p1, int p2)
{
    char buf[32];
    int sum = p1 + p2;
    printf("%d\n", sum);
}

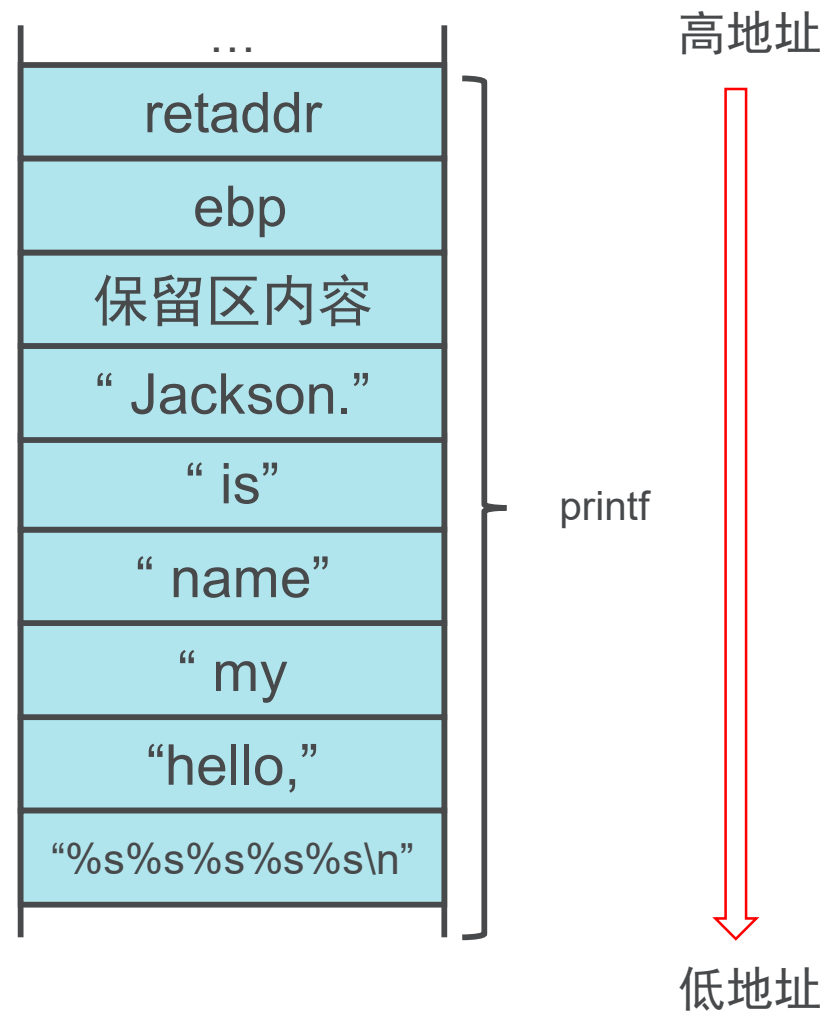
int main()
{
    int a = 1;
    int b = 2;
    foo(a, b);
    return 0;
}
```



1、背景介绍

- 栈与格式化字符串的关系
 - 一次格式化函数执行过程

```
4 int main()  
5 {  
6     printf(  
7         "%s%s%s%s%s\n",  
8         "hello,",  
9         " my",  
10        " name",  
11        " is",  
12        " Jackson."  
13    );  
14    return 0;  
15 }
```



2、背景介绍

○ 栈与格式化字符串的关系

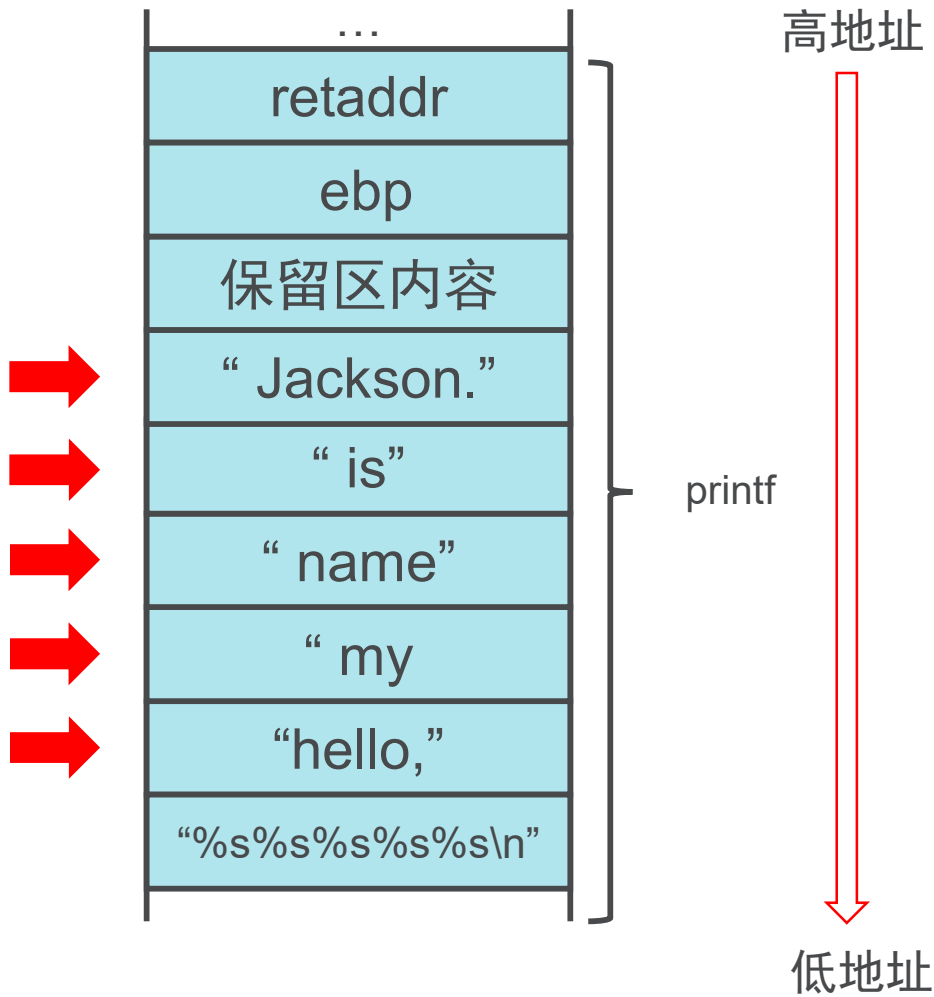
○ 一次格式化函数执行过程

```
4 int main()  
5 {  
6     printf(  
7         "%s%s%s%s%s\n",  
8         "hello,",  
9         " my",  
10        " name",  
11        " is",  
12        " Jackson."  
13    );  
14    return 0;  
15 }
```

↓ ↓ ↓ ↓ ↓

%s%s%s%s%s\n

输出: hello, my name
is Jackson



提 纲


1. 背景介绍
 - 格式化字符串简介
 - 栈与格式化字符串的关系
2. **Format String Vulnerability**介绍
3. 如何利用Format String Vulnerability
 - 我们能控制什么?
 - 如何进行攻击?
4. 如何防范Format String Vulnerability

2、Format String Vulnerability介绍

原理介绍

○格式化字符串漏洞

```
#include <stdio.h>
int main(void)
{
    printf("%d%d%d%d%s", 5, 6, 8, 0x21, "test");
    return 0;
}
```



-00000003	db ? ;
-00000002	db ? ;
-00000001	db ? ;
+00000000	s db 4 dup(?)
+00000004	r db 4 dup(?)
+00000008	format db 4 ; "%d%d%d%c"
+0000000c	%d db 4 ; 4
+00000010	%d db 4 ; 6
+00000014	%d db 4 ; 8
+00000018	%x db 4 ; 0x21
+0000001c	%s db 4 ; "test"
+0000001c	; end of stack variables

函数的调用者可以自由的指定函数参数的数量和类型，被调用者无法知道在函数调用之前到底有多少参数被压入栈帧当中。所以printf函数要求传入一个format参数用以指定到底有多少，怎么样的参数被传入其中。然后它就会忠实的按照函数的调用者传入的格式一个一个的打印出数据。

2、Format String Vulnerability介绍

原理介绍

○格式化字符串漏洞

- 如果我们无意或者有意，在format中，或者说我们要求printf打印的数据数量大于我们所给的数量会怎样？

```
#include <stdio.h>
int main(void)
{ printf("%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x");
  return 0; }
```



```
cxld@cxld-machine:~/UCAS$ sudo ./b
50242478,50242488,00000000,004005b0,69a32ac0,00400540,69678830,00000001,50242478,69c47ca0,00400526,00000000
```

这里我们只给了printf一个参数，却让其打印出12个int类型的数据。可以看到，printf按照我们意愿打印出了12个数值。这些数值不是我们输入的参数，而是保存在栈中的其他的数值。通过这个特性，黑客们就创造出了格式化字符串的漏洞。

2、Format String Vulnerability介绍

原理介绍

○格式化字符串攻击

○格式化字符串攻击原理是利用格式化函数（如printf()）的沿着堆栈指针向下打印的特性，通过**只提供格式化字符串但不提供对应的变量**，读取栈内空间的内容。更进一步，通过将某个要攻击的目标地址放入栈中，就可以利用格式化字符串读写里面的值。因此，它的攻击分为两步：

- (1) 第一步，将目标地址放入栈中；
- (2) 第二步，设计格式化字符串，读写目标地址里的值。

2、Format String Vulnerability介绍

格式化字符串漏洞的基本形式

形式上是编程

- 在程序开发过程中，开发者编程上的失误导致攻击者的输入可以作为格式化字符串，攻击者输入的格式化字符串可以控制格式化函数的运行，进而获得程序运行的控制权。

```
1  int main () {  
2      char user[256];  
3      scanf("%s", user);  
4      printf(user);  
5  }
```

```
1  int main () {  
2      char user[256];  
3      scanf("%s", user);  
4      printf("%s", user);  
5  }
```

图1：错误(左)以及正确(右)使用格式化函数printf的方法

2、Format String Vulnerability介绍

格式化字符串漏洞的基本形式

○本质上是通道问题(channeling problem)

- 对于一般的系统而言，会有两个通道，一个数据通道和一个控制通道。数据通道中传递的内容不被主动解析，只被复制。控制通道中传递的内容被主动解析，并据此影响系统的运行。许多系统中两个通道被合二为一，并通过特殊的转义字符和序列来区分活动的通道。

——通道问题本身不是漏洞，但是使得开发过程中的bug可被利用

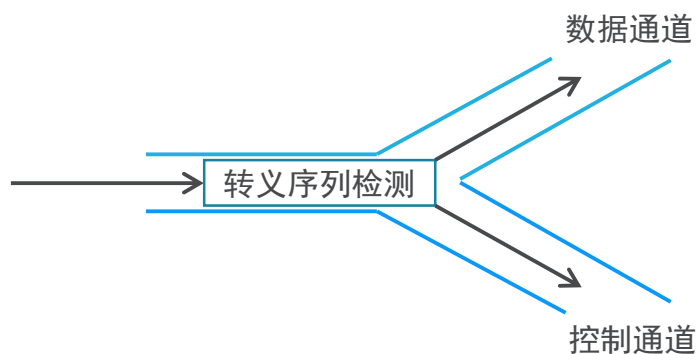


图2：通道问题示意图

通道问题	数据通道	控制通道	安全问题
电话系统	语音数据	线路控制音	影响线路控制
栈	栈中的数据	返回地址	控制返回地址
格式化字符串	输出字符串	格式化符号	控制格式化函数

表3：常见的通道问题

原理介绍

○*printf()系列函数的3条特殊性质

所谓格式化字符串,就是在*printf()系列函数中按照一定的格式对数据进行输出,可以输出到标准输出,即 printf(),也可以输出到文件句柄、字符串等。对应的函数有fprintf、sprintf、snprintf、vprintf、vfprintf、vsprintf等,能被黑客利用的地方也就出在这一系列的*printf()函数中。在正常情况下这些函数不会造成什么问题,但是*printf()系列函数有3条特殊的性质,这些特殊性质如果被攻击者结合起来利用,就会形成漏洞。

1、Format String Vulnerability介绍

原理介绍

○*printf()系列函数的3条特殊性质

○参数个数不固定造成访问越界数据

因为*printf()系列函数的参数的个数是不固定的，如果其第一个参数即格式化字符串是由用户来提供的话，那么用户就可以访问到格式化字符串前面的堆栈里的任何内容了。之所以会出现格式化串漏洞，就是因为程序员把printf()的第一个参数，即格式化字符串，交给用户来提供，如果用户提供特定数量的%x，就可以访问到特定地址的堆栈内容。

1、Format String Vulnerability介绍

原理介绍

○*printf()系列函数的3条特殊性质

○利用%n格式符写入跳转地址

%n是一个在编程中不经常用到的格式符，它的作用是把前面已经打印的长度写入某个内存地址。在实际利用某个漏洞时，并不是直接把跳转地址写入函数的返回地址，而是通过地址来间接的改写返回地址。现在已经可以利用提交格式化字符串访问格式化字符串前面堆栈里的内容，并且利用%n可以向一个内存单元中的地址去写入一个值。既然可以访问到提交的字符串，就可以在提交的字符串当中放上某个函数的返回地址的地址，这样就可以利用%n来改写这个返回地址。当然，%n向内存中写入的值并不是随意的，它只能写入前面打印过的字符数量。

1、Format String Vulnerability介绍

原理介绍

○*printf()系列函数的3条特殊性质

○利用附加格式符控制跳转地址的值

printf()系列函数有个性质是程序员可以定义打印字符的宽度。就是在格式符的中间加上一个整数，printf()就会把这个数值作为输出宽度，如果输出的实际大于指定宽度则仍按实际宽度输出；如果小于指定宽度，则按指定宽度输出。可以利用这个特性，用很少的格式符来输出一个很大的数值到%n，我们通过附加格式符来控制向函数返回地址中写入的值。

```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```

提 纲

1. 背景介绍

- 格式化字符串简介
- 栈与格式化字符串的关系

2. Format String Vulnerability介绍

3. 如何利用Format String Vulnerability

- 我们能控制什么?
- 如何进行攻击?

4. 如何防范Format String Vulnerability

1、Format String Vulnerability介绍

如何利用Format String Vulnerability

●我们能控制什么？

格式化字符串漏洞是由像printf(user_input)这样的代码引起的，其中user_input是用户输入的数据，具有Set-UID root权限的这类程序在运行的时候，printf语句将会变得非常危险，因为它可能会导致下面的结果：

○使程序崩溃（制造段错误）

[illegible]

利用格式化字符串函数printf ("%s%s%s%s%s%s%s%s%s%s%s")引发非法指针访问内存，引起程序崩溃

1、Format String Vulnerability介绍

如何利用Format String Vulnerability

○我们能控制什么？

○任意一块内存读取数据

- 我们需要得到一段数据的内存地址，但我们无法修改代码，供我们使用的只有格式字符串。
- 如果我们调用printf(%s)时没有指明内存地址，那么目标地址就可以通过printf函数，在栈上的任意位置获取。printf函数维护一个初始栈指针，所以能够得到所有参数在栈中的位置。
- 观察：格式字符串位于栈上。如果我们可以把目标地址编码进格式字符串，那样目标地址也会存在于栈上，在接下来的例子里，格式字符串将保存在栈上的缓冲区中。

1、Format String Vulnerability介绍

如何利用Format String Vulnerability

○我们能控制什么？

○任意一块内存读取数据

```
int main(int argc, char *argv[])
{
    char user_input[100];
    ... /* other variable definitions and statements */
    scanf("%s", user_input); /* getting a string from user */
    printf(user_input); /* Vulnerable place */
    return 0;
}
```

如果我们让printf函数得到格式字符串中的目标内存地址(该地址也存在于栈上)，我们就可以访问该地址。

1、Format String Vulnerability介绍

如何利用Format String Vulnerability

○我们能控制什么？

○任意一块内存读取数据

```
printf ("\x10\x01\x48\x08 %x %x %x %x %s");
```

- \x10\x01\x48\x08是目标地址的四个字节,在 C 语言中,\x10告诉编译器将一个16进制数0x10放于当前位置（占1字节）。如果去掉前缀\x10就相当于两个ascii 字符1和0了，这就不是我们所期望的结果了。
- %x导致栈指针向格式字符串的方向移动；
- 攻击的关键就是弄清楚user_input和传给printf的地址的距离。这个距离决定了在提供%s之前，你需要向格式化字符串插入多少个%x。

下图解释了攻击方式：

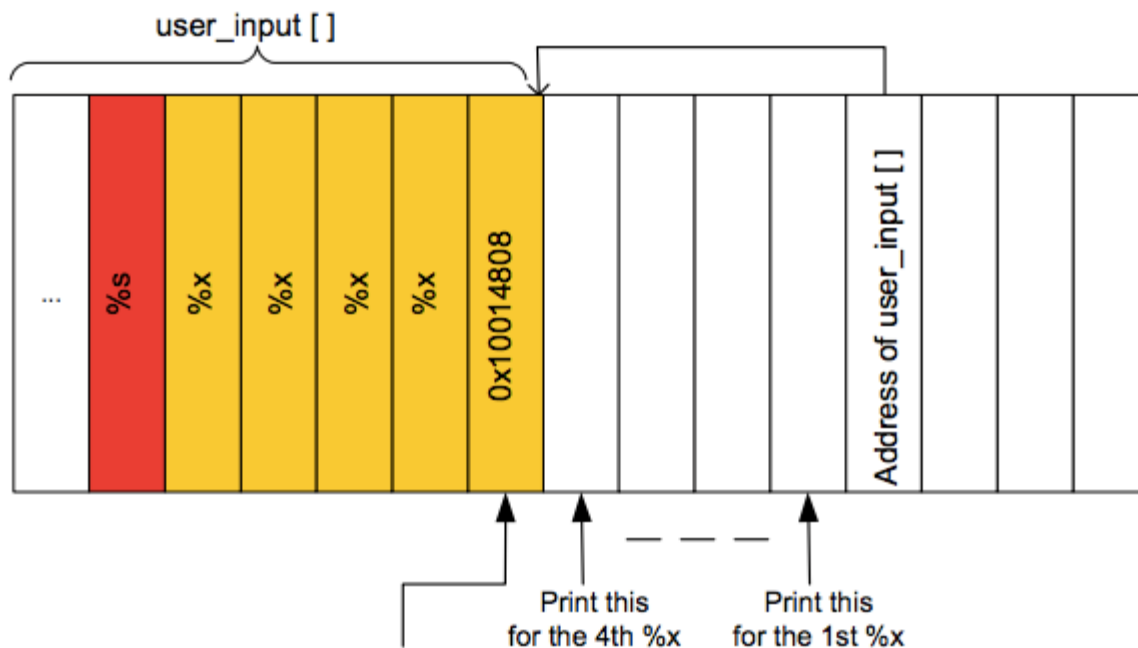
1、Format String Vulnerability介绍

如何利用Format String Vulnerability

○我们能控制什么？

○任意一块内存读取数据

Print out the contents at the address 0x10014808 using format-string vulnerability



For %s: print out the contents pointed by this address

1、Format String Vulnerability介绍

如何利用Format String Vulnerability

○我们能控制什么？

○修改任意一块内存里的数据

这个结果是非常危险的，因为它允许用户修改 set-UID root 程序内部变量的值，从而改变这些程序的行为。

%n: 该符号前输入的字符数量会被存储到对应的参数中去。

```
int i;  
printf ("12345%n", &i);
```

- 数字5（%n 前的字符数量）将会被写入i中；
- 运用同样的方法在访问任意地址内存的时候，我们可以将一个数字写入指定的内存中。只要将上一小节的 %s 替换成 %n 就能够覆盖 0x10014808 的内容；
- 利用这个方法，攻击者可以做以下事情：
 - 重写程序标识控制访问权限
 - 重写栈或者函数等等的返回地址。

提 纲

1. 背景介绍

- 格式化字符串简介
- 栈与格式化字符串的关系

2. Format String Vulnerability介绍

3. 如何利用Format String Vulnerability

- 我们能控制什么?
- 如何进行攻击?

4. 如何防范Format String Vulnerability

示例A：攻击要求

- 关闭内存的地址随机化（后面会讲到原因，不关也可以）
 - `cat /proc/sys/kernel/randomize_va_space`
- 使用32位 Ubuntu 16.04 系统进行演示

攻击目标A

改变 *const* 变量的值

const代码示例

```
#include <stdio.h>
#include <string.h>
int main( int argc, char *argv[] ) {
    const int N = 6; // static 变量是不可能在运行中更改的
    printf( "The addr of <const int N> is %08x\n", &N );
    printf( "The value of <const int N> is %d\n", N );
    N += 1;
    char str[100];
    strcpy( str, argv[1] );
    printf( str );
    printf( "\nThe value of <const int N> is %d\n", N );
    return 0;
}
```

通过合法的输入改变const变量的值

- Const修饰符在C/C++中是非常重要的，他是程序员用来保护某个变量的武器。但是通过合法的输入，可以“悄悄”使其失效，从而引发雪崩。

```
newton@Newton-rPC-1 ~/format
$ ./format.o "`printf "\xa4\xf5\xff\xbf"`.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x"
The addr of <const int N> is bffff5a4
The value of <const int N> is 6
.bffff7f5.b7fff918.00f0b5ff.bffff5ce.00000001.000000c2.bffff6c4.bffff5ce.00000006.bffff5a4
The value of <const int N> is 6

newton@Newton-rPC-1 ~/format
$ ./format.o "`printf "\xa4\xf5\xff\xbf"`.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08x.%.08n"
The addr of <const int N> is bffff5a4
The value of <const int N> is 6
.bffff7f5.b7fff918.00f0b5ff.bffff5ce.00000001.000000c2.bffff6c4.bffff5ce.00000006.
The value of <const int N> is 86
```

1 2 3 4 5 6 7 8 9 10

Hard coded pointer, 硬编码指针

图2: const修饰的变量被篡改

C/C++中的const修饰符

○C语言中的const修饰符

- 用在变量前，表示该变量是不能改变的

```
newton@Newton-rPC-1 ~/format
$ make
gcc format.c -o format.o -w
format.c: In function 'main':
format.c:14:7: error: assignment of read-only variable 'N'
    N += 1;
    ^
Makefile:3: recipe for target 'gen' failed
make: *** [gen] Error 1
```

图2: stdio.h中的printf函数

攻击目标2

2. 更进一步：**关闭**随机地址也行

如何向内存里写入地址

- 1. printf 命令生成二进制值
 - CLI 中的 printf (非本次课所提及的那个 printf)

```
PRINTF(1)                                BSD General Co  
  
NAME  
    printf -- formatted output  
  
SYNOPSIS  
    printf format [arguments ...]
```

图：BSD bash中的printf命令

如何向内存里写入地址2

- 2. 用 `scanf ("%u", &target)`
 - 相当于**直接**把地址以 16 进制（或者其他进位制）的形式**写进去**，要特别注意转换进位制以及变量的内存占用量。

提 纲

1. 背景介绍

- 格式化字符串简介
- 栈与格式化字符串的关系

2. Format String Vulnerability介绍

3. 如何利用Format String Vulnerability

- 我们能控制什么?
- 如何进行攻击?

4. 如何防范Format String Vulnerability

如何防范?

- **地址空间随机化**: 就像用于保护缓冲区溢出攻击的预防措施那样, 地址空间**随机化**, 攻击者难以找到他们想要读取或写入什么地址。

ASLR, 没有银弹 1

Address Space Layout Randomization

内存空间布局随机化

Address space layout randomization makes it more difficult to exploit existing vulnerabilities, instead of increasing security by removing them. Thus ASLR is not a replacement for insecure code, but it can offer protection from vulnerabilities that have not been fixed and even not been published yet. The aim of ASLR is to introduce randomness into the address space of a process. This lets a lot of common exploits fail by crashing the process with a high probability instead of executing malicious code.

攻击ASLR的对策

- 耗尽Entropy Pool（熵池），转换到64位系统是比较好的选择。
- 微软在Vista Beta 2系统里开启ASLR（2006）

攻击ASLR的对策

- 耗尽Entropy Pool（熵池），转换到64位系统是比较好的选择。
- 微软在Vista Beta 2系统里开启ASLR（2006）

有什么靠谱的可行方案吗？

改变母体

1. **Compiler extensions:** StackGuard, StackShield, /GS-Option, bounds checking, canary **(编译器角度，审查边界、栈防卫)**
2. 但是，这需要对现有的所有代码用新的编译器重编译
3. **结果：**不是很好推广。

有什么靠谱的可行方案吗？

改变生存环境

1. **PaX (complete ASLR), Openwall (non-executable stack) (操作系统角度, 完全的随机地址)**
2. 但是, 这会大大加重系统消耗, 降低性能
3. **结果:** 用户可能不太喜欢。

有什么靠谱的可行方案吗？

改变程序员

1. **Safe programming:** source code analyzer, tracer, fuzzer **(开发流程角度，严格的代码审查)**
2. 全靠自律
3. **结果：**不能一劳永逸。

如何一劳永逸，高枕无忧？

一个合格的程序员，不会把内存地址暴露出来！

C -> C++ -> Java -> C#

建议用Reference
不要用Pointer

彻底摒弃指针

