

C++ Summit 2020

谈静国  
华为ICT基础软件  
渗透测试专家

# 渗透视角下的 C/C++安全编码 实践

# 目录

1. 从一个漏洞说起 (CVE-2020-8597)

2. 安全函数

3. 静态扫描

4. fuzzing

5. 安全编译选项

6. 总结

## CVE-2020-8597

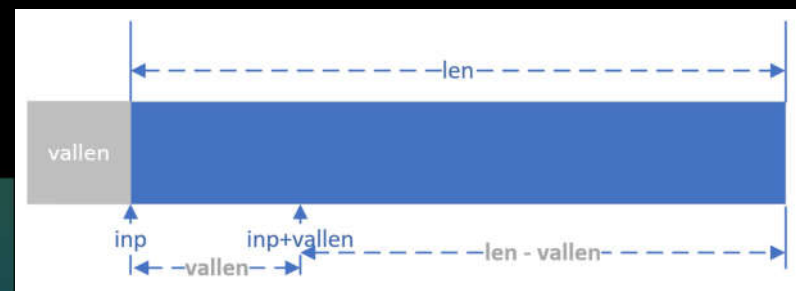
CVE-2020-5987 是pppd软件中存在17年之久的**远程代码执行**漏洞，CVSS评分为**9.8分(严重程度:Critical)**。

Ubuntu/Debian/Fedora等系统均受影响。

漏洞发生在处理Extensible Authentication Protocol (EAP)消息的eap.c:eap\_request函数中。

如右图所示，inp中是网络传输的ppp数据，len和vallen均可控。

代码中存在逻辑错误，未判断 **len - vallen** 与 **sizeof(rhostname)** 大小，导致1429行发生缓冲区溢出。



```

1407: case EAPT_MD5CHAP:
1408:     if (len < 1) {
1409:         error("EAP: received MD5-Challenge with no data");
1410:         /* Bogus request; wait for something real. */
1411:         return;
1412:     }
1413:     GETCHAR(vallen, inp);
1414:     len--;
1415:     if (vallen < 8 || vallen > len) {
1416:         error("EAP: MD5-Challenge with bad length %d (8..%d)",
1417:             vallen, len);
1418:         /* Try something better. */
1419:         eap_send_nak(esp, id, EAPT_SRP);
1420:         break;
1421:     }
1422:
1423:     /* Not so likely to happen. */
1424:     if (vallen >= len + sizeof(rhostname)) {
1425:         dbglog("EAP: trimming really long peer name down");
1426:         BCOPY(inp + vallen, rhostname, sizeof(rhostname) - 1);
1427:         rhostname[sizeof(rhostname) - 1] = '\0';
1428:     } else {
1429:         BCOPY(inp + vallen, rhostname, len - vallen);
1430:         rhostname[len - vallen] = '\0';
1431:     }

```

判断逻辑错误，永远为 false

rhostname 为栈变量，栈溢出

## CVE-2020-8597

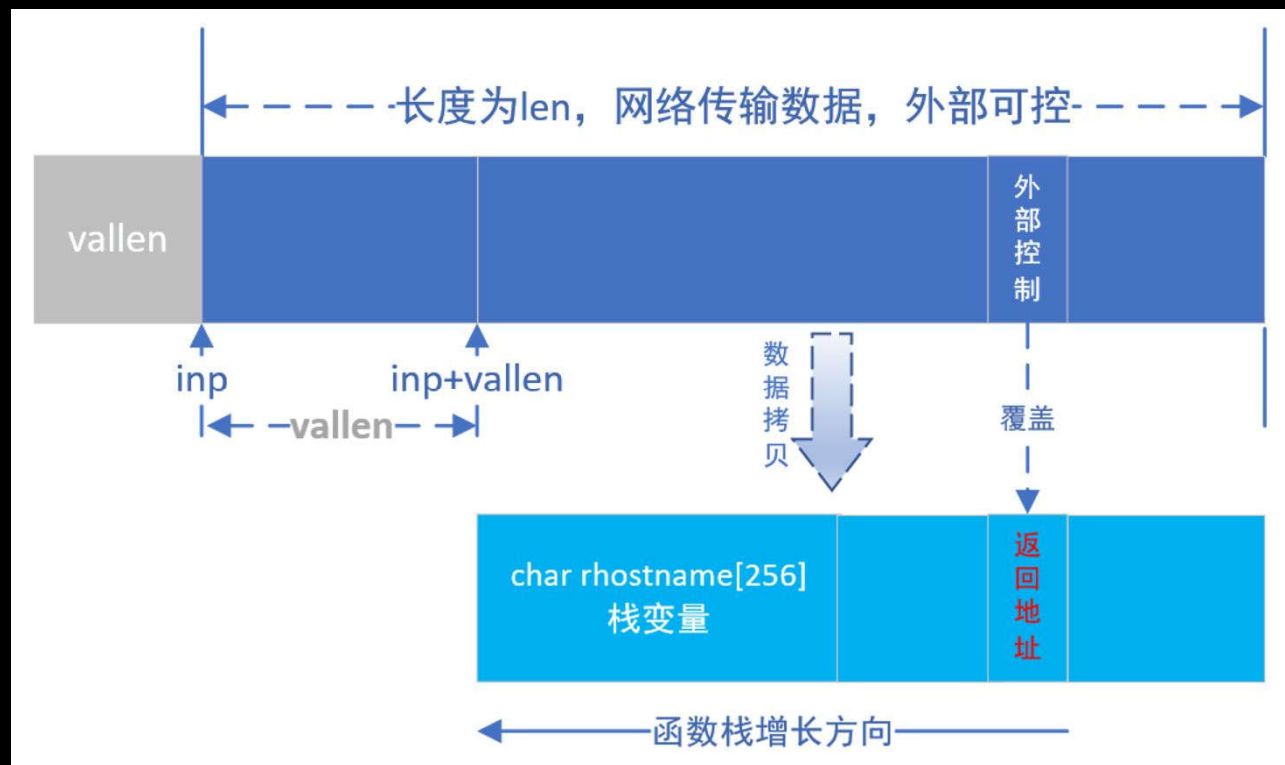
通过精心EAP数据包中的内容，  
在调用到BCOPY时覆盖函数**返回地址**：

`BCOPY(inp + vallen, rhostname, len - vallen);`

如右图示意图所示：

`char rhostname[256];` //栈上的变量

函数返回地址覆盖之后，可以  
通过执行**ShellCode**或**ROP**等方式  
控制程序执行流，让pppd进程执行  
恶意代码。



# CVE-2020-8597

漏洞修复patch:

pppd/eap.c	
@@ -1420,7 +1420,7 @@ int len;	
1420 }	1420 }
1421	1421
1422 /* Not so likely to happen. */	1422 /* Not so likely to happen. */
1423 - if (vallen >= len + sizeof (rhostname)) {	1423 + if (len - vallen >= sizeof (rhostname)) {
1424 dbglog("EAP: trimming really long peer name down");	1424 dbglog("EAP: trimming really long peer name down");
1425 BCOPY(inp + vallen, rhostname, sizeof (rhostname) - 1);	1425 BCOPY(inp + vallen, rhostname, sizeof (rhostname) - 1);
1426 rhostname[sizeof (rhostname) - 1] = '\0';	1426 rhostname[sizeof (rhostname) - 1] = '\0';
@@ -1846,7 +1846,7 @@ int len;	
1846 }	1846 }
1847	1847
1848 /* Not so likely to happen. */	1848 /* Not so likely to happen. */
1849 - if (vallen >= len + sizeof (rhostname)) {	1849 + if (len - vallen >= sizeof (rhostname)) {
1850 dbglog("EAP: trimming really long peer name down");	1850 dbglog("EAP: trimming really long peer name down");
1851 BCOPY(inp + vallen, rhostname, sizeof (rhostname) - 1);	1851 BCOPY(inp + vallen, rhostname, sizeof (rhostname) - 1);
1852 rhostname[sizeof (rhostname) - 1] = '\0';	1852 rhostname[sizeof (rhostname) - 1] = '\0';

修改错误的逻辑，将if (vallen >= len + sizeof (rhostname)) 改为：  
if (len - vallen >= sizeof (rhostname))

思考：这种方法是针对漏洞的专门修复，工程上有没有更普适的方法？

## 安全函数

漏洞关键代码:

```
BCOPY(inp + vallen, rhostname, len - vallen);
```

BCOPY是一个宏定义, 没有长度判断, 无保护机制:

```
#define BCOPY(s, d, l) memcpy(d, s, l)
```

使用安全函数memcpy\_s:

```
memcpy_s(rhostname, sizeof(rhostname),  
         inp + vallen, len - vallen);  
//sizeof(rhostname) = 256
```

**安全函数使用效果:**

- 1) 无论len-vallen的值为多大, 最多只往rhostname拷贝256字节, 不会发生缓冲区溢出。
- 2) 逻辑漏洞未patch, 也不会导致栈溢出。

## 安全函数

华为安全函数库（随openark等项目开源，mulan开源协议）：

[https://gitee.com/openarkcompiler/OpenArkCompiler/tree/master/src/mapleall/huawei\\_secure\\_c/src](https://gitee.com/openarkcompiler/OpenArkCompiler/tree/master/src/mapleall/huawei_secure_c/src)

常见的安全函数包括：





- ✓ memcpy\_s
- ✓ memmove\_s
- ✓ memset\_s
- ✓ scanf\_s
- ✓ snprintf\_s
- ✓ strcpy\_s
- ✓ strncat\_s
- ✓ strncpy\_s
- ✓ .....

跨平台支持：Windows/Linux

fscanf_s.c	mv to mapleall folder
fwscanf_s.c	mv to mapleall folder
gets_s.c	mv to mapleall folder
input.inl	mv to mapleall folder
memcpy_s.c	mv to mapleall folder
memmove_s.c	mv to mapleall folder
memset_s.c	mv to mapleall folder
output.inl	mv to mapleall folder
scanf_s.c	mv to mapleall folder
secinput.h	mv to mapleall folder
secureutil.c	mv to mapleall folder

## 安全函数

防止不正确的使用memcpy\_s:

- 1) BCOPY(inp + vallen, rhostname, len - vallen); 
- 2) memcpy\_s(rhostname, sizeof(rhostname),  
inp + vallen, len-vallen); 
- 3) memcpy\_s(rhostname, len-vallen,  
inp + vallen, len- vallen);   
memcpy\_s 2,4参数相同, 未起到保护效果。
- 4) #define SAFE\_COPY(d, s, l) memcpy\_s(d, l, s, l)   
SAFE\_COPY(rhostname, inp + vallen, len-vallen)  
封装安全函数, 导致memcpy\_s 2,4参数相同, 未起到保护效果。

可通过制定安全编程规范, 禁止错误使用安全函数的行为。



## 整形溢出漏洞

## Sample Code 1:

```
p = malloc(len + 1);  
if(p != NULL) {  
    memset(p,0,len);  
}
```

漏洞在哪?

## Sample Code 2:

```
p = malloc(nblocks * block_size);  
if(p != NULL) {  
    memset(p,0,blocks);  
}
```

漏洞在哪?

## 整形溢出漏洞

### Sample Code 1:

```
p = malloc(len + 1);    //len=0xffffffff len+1=0, len is unsigned int
if(p != NULL) {
    memset(p,0,len);    //heap overflow
}
溢出导致heap overflow.
```

### Sample Code 2:

```
p = malloc(nblocks * block_size); //nblocks,block_size可控
                                   //nblocks = 0x10000 block_size = 0x10000
                                   //32位 nblocks*block_size = 0

if(p != NULL) {
    memset(p,0,blocks); //heap overflow
}
溢出导致heap overflow.
```

## OOB 漏洞

## Sample Code 3:

```
int g_array[0x100];  
  
int func(int idx) {  
    if(idx > 0x100) {  
        return -1;  
    } else {  
        return g_array[idx];  
    }  
}
```

漏洞在哪?

## OOB 漏洞

## Sample Code 3:

```
int g_array[0x100];

int func(int idx) {
    if(idx > 0x100) {
        return -1;
    } else {
        return g_array[idx];    //OOB if idx = -0x1000
    }
}

idx = -0x1000时, 发生OOB(Out Of Bound).
```

使用安全函数无法避免整形溢出/OOB漏洞, 工程上可采用什么方法防止?

## 静态代码扫描

常用静态代码扫描工具:

- ✓ Fortify
- ✓ CodeMars
- ✓ Coverity
- ✓ PinPoint
- ✓ CodeChecker
- ✓ LLVM Clang Static Analyzer (开源, 可定制checker插件)

通常使用静态代码扫描工具 + 定制分析规则通常能检测出漏洞:

Sample Code 1(整形溢出)

Sample Code 2(整形溢出)

Sample Code 3(OOB)

# LLVM CSA

官方文档: <https://clang.llvm.org/docs/ClangStaticAnalyzer.html>

静态检测原理:

Call Graph + CFG + 符号执行约束求解

Sample Code 1:

```
int func(unsigned int len) {    //符号化:reg_$0<unsigned int len>
    p = malloc(len + 1);      //reg_$0<unsigned int len> + 1
    if(p != NULL) {
        memset(p,0,len);
    }
}
```

对加法操作设置溢出检测条件 (在checker中实现) :

$\text{reg\_}\$0\langle\text{unsigned int len}\rangle > (\text{reg\_}\$0\langle\text{unsigned int len}\rangle + 1)$

进行约束求解, 若有解表示可以触发溢出; 若无解, 说明Call graph上有其他限制条件, 无法触发溢出。

# LLVM CSA

LLVM CSA中已实现一些checker, 如检查overflow的checker:

```
$ ./build/bin/clang -cc1 -analyzer-checker-help | grep overflow
alpha.security.ArrayBound      Warn about buffer overflows (older checker)
alpha.security.ArrayBoundV2    Warn about buffer overflows (newer checker)
alpha.security.MallocOverflow  Check for overflows in the arguments to malloc()
alpha.unix.Overflow            Check for overflow.
```

检查越界读、越界写的checker:

```
$ ./build/bin/clang -cc1 -analyzer-checker-help | grep Bound
alpha.security.ArrayBound      Warn about buffer overflows (older checker)
alpha.security.ArrayBoundV2    Warn about buffer overflows (newer checker)
alpha.unix.cstring.OutOfBounds Check for out-of-bounds access in string functions
osx.coreFoundation.containers.OutOfBounds
```

问题: 成因比较复杂的漏洞无法检测, 如race condition, 需要自定义实现checker插件。

## Race Condition-> Double Free

### Sample Code 4: (Race condition 导致 Double Free)

```
int delete_data(int index) {
    DATA_INFO* pinfo;
    pthread_mutex_lock(&g_mutex);
    pinfo = search_for_data(index);    //A,B两个线程均通过相同index找到相同pinfo
    pthread_mutex_unlock(&g_mutex);

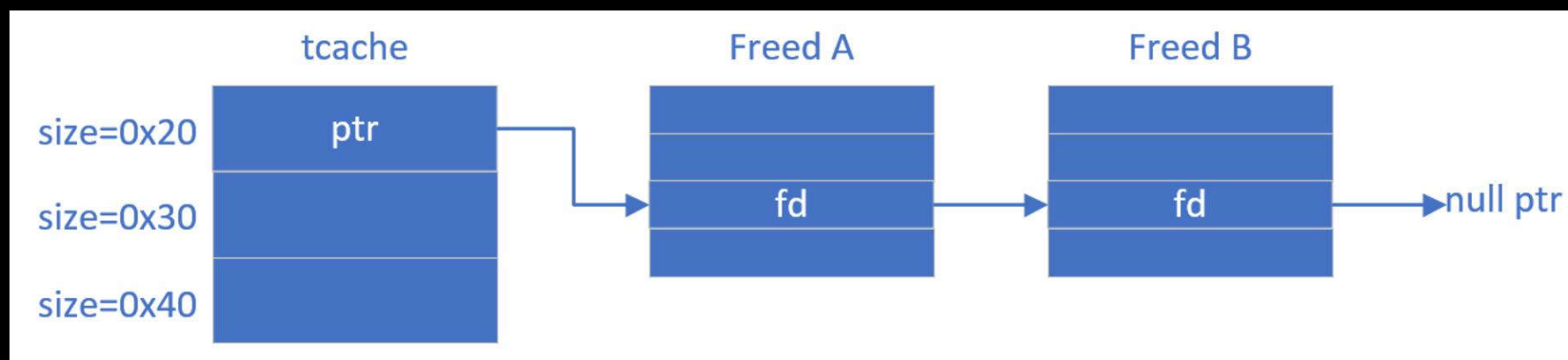
    pthread_mutex_lock(&g_mutex);
    if(pinfo!= NULL) {
        free_data(pinfo);              //A,B两个线程先后调用free_data(pinfo),Double Free
        pinfo = NULL;
    }
    pthread_mutex_unlock(&g_mutex);
}
```

Race Condition --> Double Free造成的危害是怎样?

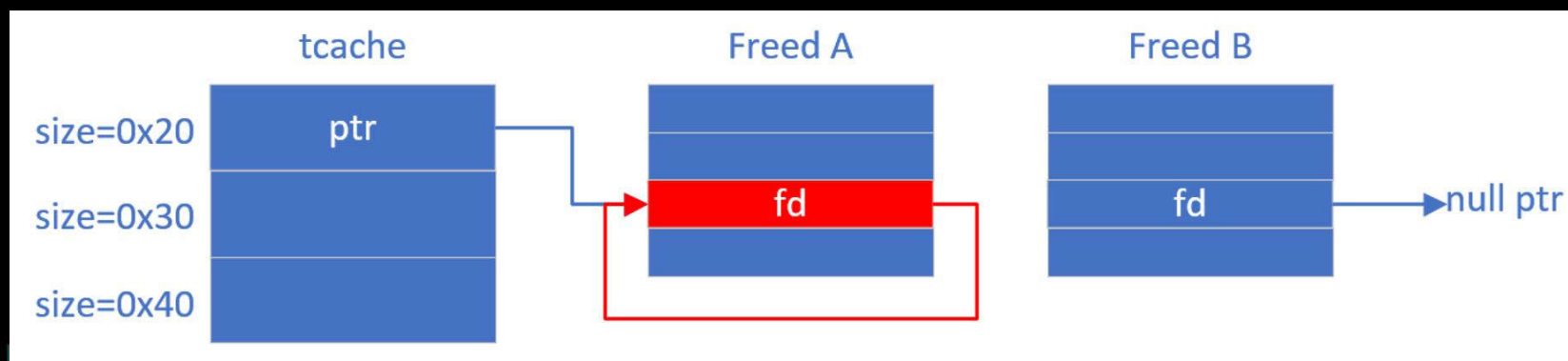


## Double Free的危害

Double Free可以转换为任意地址写任意值，进而可执行恶意代码，获取Shell。  
glibc较新版本中引入tcache，free释放的A B两块内存挂在tcache链表上：

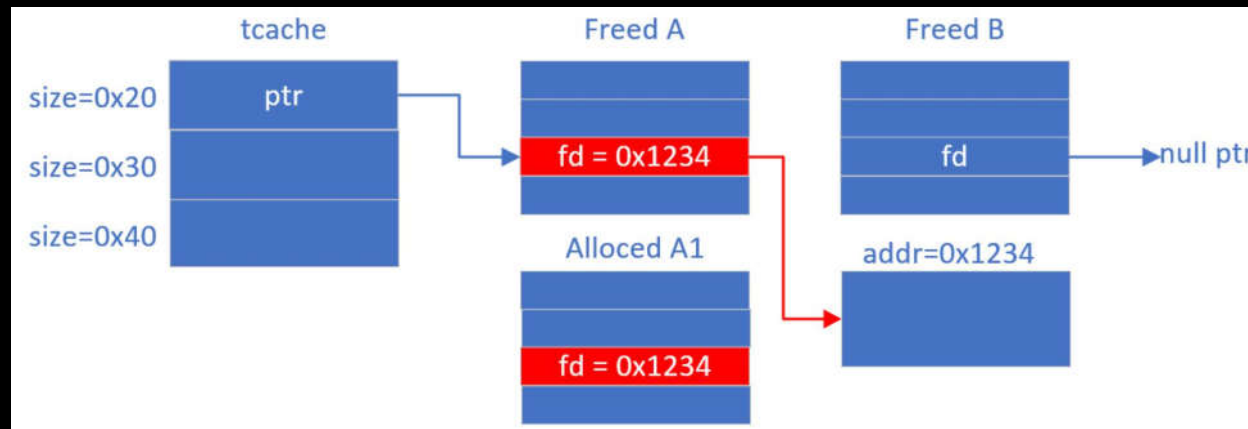


Double Free发生时，再次free A，链表结构被破坏，成为自循环：

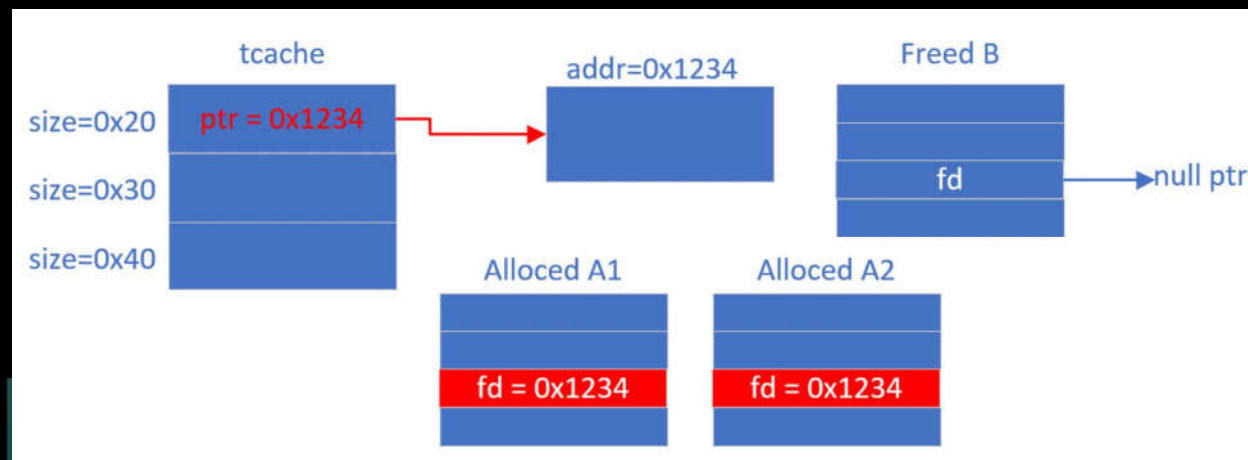


# Double Free的危害

申请一块内存Alloced A1后，tcache的ptr仍然指向Freed A，Alloced A1 = Freed A  
写A1内存fd = 0x1234(或其他任意地址)

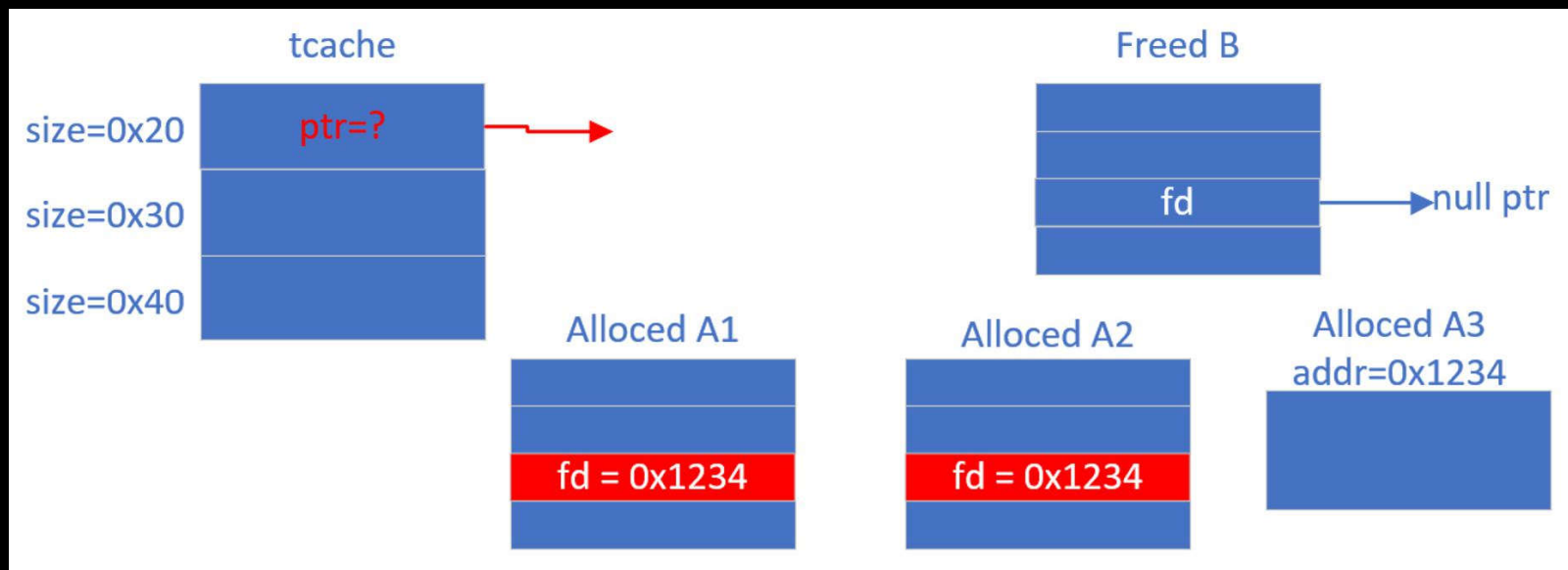


再次申请一块内存A2之后，tcache ptr指向0x1234 (Alloced A1 = Alloced A2)



## Double Free的危害

再申请一块内存，申请到的Alloced A3地址为0x1234（或其他任意地址）。



此时往Alloced A3中写入数据，形成任意地址写任意值。

## Double Free的危害

具备任意地址写任意值能力之后，向全局变量\_\_malloc\_hook地址写入OneGadget地址，再次调用malloc时，glibc中\_\_libc\_malloc将执行OneGadget地址的指令，如下所示：

```
3034: void *
3035: __libc_malloc (size_t bytes)
3036: {
3037:     mstate ar_ptr;
3038:     void *victim;
3039: |
3040:     void>(*hook) (size_t, const void *)
3041:         = atomic_forced_read (__malloc_hook);
3042:     if (__builtin_expect (hook != NULL, 0))
3043:         return (*hook)(bytes, RETURN_ADDRESS (0));
3044: #if USE_TCACHE
```

OneGadget地址执行execve("/bin/bash")，如下所示：

```
→ test one_gadget /lib/x86_64-linux-gnu/libc.so.6
0x4f3d5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL
```

## Race Condition检测

### Race Condition LLVM CSA漏洞检测建模:

- 1) 识别访问共享资源。定义访问共享资源操作，如访问链表 list\_head等可认为是访问共享资源。
- 2) 识别使用锁。定义锁操作，如pthread\_mutex\_lock等函数。
- 3) 识别存在Race Condition的行为。

规则：Call graph中一条路径执行过程中，多组lock/unlock访问共享资源，且多组访问之间无额外锁保护，则认为存在Race Condition问题。

```
int delete_data(int index) {
    DATA_INFO* pinfo;
    pthread_mutex_lock(&g_mutex);    //1.1 识别到第一组lock/unlock
    pinfo = search_for_data(index);    //2 search_for_data函数中链表操作，识别到访问共享资源
    pthread_mutex_unlock(&g_mutex);    //1.2 识别到第一组lock/unlock

    pthread_mutex_lock(&g_mutex);    //3.1 识别到第二组lock/unlock
    if(pinfo!= NULL) {
        free_data(pinfo);    //4 free_data函数中链表操作，识别到访问共享资源
        pinfo = NULL;
    }
    pthread_mutex_unlock(&g_mutex);    //3.2 识别到第三组lock/unlock
}
```

# LLVM CSA

测试CSA checker:

存在问题的函数delete\_data, 在62行查找pinfo, 在67行释放。

两个线程并发时, 62行均找到pinfo, 两次调用67行释放pinfo, Double Free。

运行结果提示delete\_data -> free\_data这条调用路径存在Double Free, 如下所示:

```
#0 Calling free_data at line 67
#1 Calling delete_data
Double Free Detected...
./sample/race_double_free.c:53:5: warning: Call Stack:#0 Calling free_data at line 67#1 Calling delete_data
    list_del(&(pinfo->list));
    ~~~~~
2 warnings generated.
```

其他类型的漏洞如UAF等也可以通过建模方式检测。

```
32 DATA_INFO* search_for_data(int index)
33 {
34     DATA_INFO* pinfo;
35     list_for_each_entry(pinfo,&g_data_head,list)
36     {
37         if(index == pinfo->index)
38         {
39             return pinfo;
40         }
41     }
42     return NULL;
43 }
51 int free_data(DATA_INFO* pinfo)
52 {
53     list_del(&(pinfo->list));
54     free(pinfo);
55     return 0;
56 }
57
58 int delete_data(int index)
59 {
60     DATA_INFO* pinfo;
61     pthread_mutex_lock(&g_mutex);
62     pinfo = search_for_data(index);
63     pthread_mutex_unlock(&g_mutex);
64
65     pthread_mutex_lock(&g_mutex);
66     if(pinfo != NULL) {
67         free_data(pinfo);
68         pinfo = NULL;
69     }
70     pthread_mutex_unlock(&g_mutex);
71     return 0;
72 }
```

## Fuzzing

Fuzzing能够发现更多的安全编码漏洞。  
Fuzzing也是一门博大精深的学问。  
常用的fuzzer有：

- ✓ libfuzzer (基于代码覆盖率变异, 开发阶段使用)
- ✓ honggfuzz
- ✓ SecCodeFuzz
- ✓ Trinity
- ✓ AFL
- ✓ Peach
- ✓ SecDive
- ✓ syzkaller
- ✓ ...

Fuzzing时加上 ASAN, 更好的探测到内存错误。

```
int stack_overflow(const uint8_t *Data, size_t Size) {  
    char buf[32];  
    memcpy(buf, Data, Size);  
    return 0;  
}  
  
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {  
    stack_overflow(Data, Size);  
    return 0;  
}
```

libfuzzer函数级fuzz用例



## 安全编译

通过安全编译，可使漏洞无法利用，或大幅提升漏洞利用的难度。

几个常用的安全编译选项：

### ➤ NX 堆栈不可执行

防止注入shellcode到堆栈并执行

### ➤ SP 栈保护

生成stack canary，探测到栈溢出时crash

### ➤ PIE 地址无关

系统支持ASLR时程序各segment地址随机化

地址随机化之后，攻击者无法获取程序代码段地址，无法获取libc加载地址增加攻击难度。

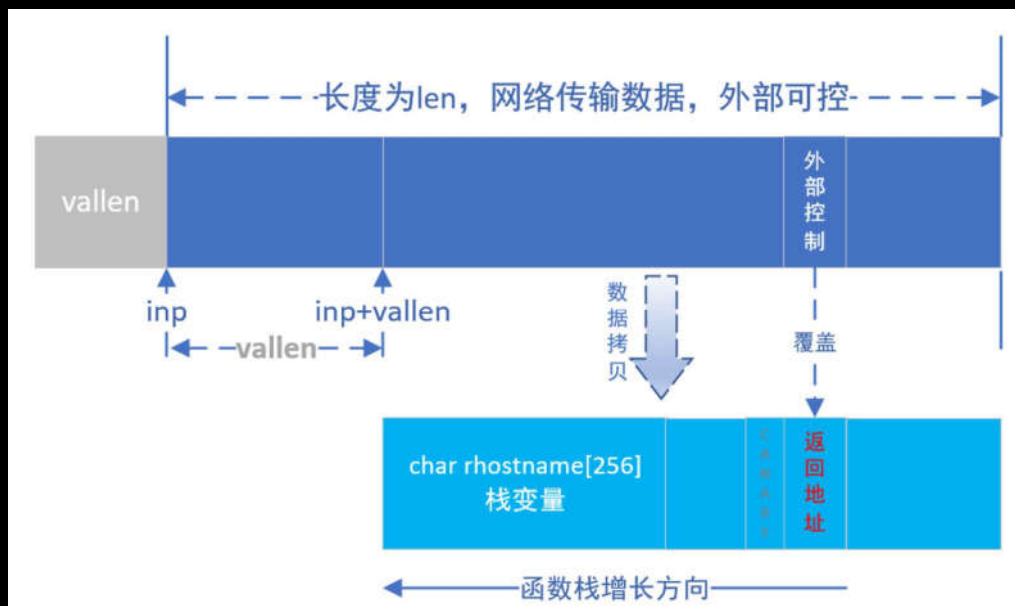
例如:Double Free利用成功必须先获取libc的加载地址。

### ➤ Strip

删除符号表，增大程序被逆向分析的难度。（对开源软件无影响）

### ➤ RELRO

GOT表保护，Full RELRO时无法修改GOT表。





## 总结

开发安全可信的代码，工程上可采用的方法：

- ✓ 安全函数库

推荐!在不改变程序逻辑的情况下避免漏洞，但要防止使用方法错误和错误封装

- ✓ 安全编程规范

约束员工编码行为，编写可信的代码

- ✓ 商用静态代码扫描工具

商用工具 + 定制规则 解决一部分问题

- ✓ 定制静态代码扫描工具

基于LLVM + CSA，通过漏洞建模，编写自定义扫描工具增强发现能力

- ✓ Fuzzing + ASAN

通过模糊测试发现漏洞

- ✓ 安全编译选项

在漏洞尚未修复的情况下可使漏洞无法利用或增大漏洞利用的难度

# Q&A

