I、glibc 漏洞简介

Google 的安全研究团队披露了 glibc getaddrinfo 溢出漏洞。经研究发现，所有 Debian、Red Hat 以及更多其它 Linux 发行版，只要 glibc 版本大于 2.9 就会受到该溢出漏洞影响。攻击者可以通过该漏洞直接批量获取大量主机权限。

在 getaddrinfo 函数中，若传递 AF_UNSPEC 参数进行 DNS 查询，调用 NSS 模块 libnss_dns.so.2 的 send_dg(UDP) 和 send_vc(TCP) 函数后会引发溢出。（如果是 AF_UNSPEC ，仅调用 gethostbyname4_r 函数）。

使用 AF_UNSPEC 参数会触发低级解析代码并发查询 A(ipv4) 和 AAAA(ipv4)。由于用于查询的缓冲区管理不当，返回响应结果时，超出 _nss_dns_gethostbyname4_r 中分配的缓冲区。

II、实验环境

① 操作系统和 glibc 版本

- 操作系统：Ubuntu 15.04
- 调试器：GDB
- glibc 版本：glibc2.20

Tips：在 Ubuntu 14.04 编译安装 glibc2.20 后，运行最基本的 helloWorld 程序，都会出现段错误。我现在还不太清楚原因，如果读者和我有相同的错误，建议使用 Ubuntu 15.04 。

② glibc 编译安装

(1)普通命令行安装

在 Ubuntu 系统下，只需要执行源码和调试符的命令之后就可以使用 gdb 对 glibc 的跟踪调试，安装指令如下：

```
1    sudo apt-get install libc6-dbg
2    sudo apt-get source libc6-dev
```

但是因为系统自带的 glibc 是 发行版 的，所以在编译的是时候选用了优化参数 -O2,所以在调试的过程中会出现 变量被优化无法读取 以及 代码运行的时候与源码的行数对不上 的情况。

所以需要自己编译一个可调试且没有过度优化的 glibc 进行调试。

(2)手动编译安装

前文介绍，glibc 版本大于 2.9 都存在漏洞，选择你喜欢的一个版本，我选择的是 glibc 2.20。
编译安装 glibc 教程，相关教程可自行 google 。

需要注意的是在进行 configure 时需要设置一些特殊的参数。如果需要调试宏可以添加 -gdwarf-2,glibc 无法使用 -O0 编译，不过 -O1 也够用了。

```
tar zxvf glibc-2.20.tar.gz
sudo mv glibc-2.20 /opt/
cd /opt/glibc-2.20
sudo mkdir build && cd build
/opt/glibc-2.20/configure --prefix=/usr/local/glibc220/ --enable-debug CFLAGS="-g -O1" CPPF
"-g -O1"
sudo make
sudo make install
```

Tips: CFLAGS 和 CPPFLAGS 中的“-O1”，前一个大写字母 O，紧跟着是数字 1。

③ 配置本地 DNS 服务器

运行 poc 的 python 服务器之前，修改/etc/resolv.conf 配置，将域名服务器改为 127.0.0.1，本机器会无法正常访问网页。

```
nameserver 127.0.0.1
```

III、运行 POC

① POC 分析

在 google 给出的 poc（该 poc 只能造成溢出，并不能利用）的基础上进行调试。
该 POC 文件包括两部分：

- .c 文件：调用 getaddrinfo 解析 “foo.bar.google.com”。
- .py 文件：绑定 53 端口，模拟 DNS 服务器对 getaddrinfo 的请求进行响应。

② 运行 python 脚本

```
sudo python CVE-2015-7547-poc.py
```

```
star@ubuntu:~/glibc/CVE-2015-7547-master$ sudo python CVE-2015-7547-poc.py
```

服务器端等待客户端发送请求

③ 运行客户端程序

(1) 使用调试版本 glibc 编译 POC

```
gcc -o client CVE-2015-7547-client.c -Wl,-rpath /usr/local/glibc220
```

通过 ldd 指令可以看到，确实使用了刚编译的 glibc。

```
star@ubuntu:~/glibc/CVE-2015-7547-master$ ldd client
        linux-gate.so.1 =>  (0xb7732000)
        libc.so.6 => /usr/local/glibc220/lib/libc.so.6 (0xb758c000)
        /lib/ld-linux.so.2 (0xb7733000)
star@ubuntu:~/glibc/CVE-2015-7547-master$
```

(2) 运行

```
1    ./client
```

```
star@ubuntu:~/glibc/CVE-2015-7547-master$ ./client
Segmentation fault (core dumped)
```

出现段错，说明成功溢出.

(3) GDB 调试

```
star@ubuntu:~/glibc/CVE-2015-7547-master$ gdb client -q
Reading symbols from client...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/star/glibc/CVE-2015-7547-master/client

Program received signal SIGSEGV, Segmentation fault.
__GI___libc_res_nquery (statp=0xb7fd6340 <_res@GLIBC_2.0>,
    name=0x8048653 "foo.bar.google.com", class=1, type=62321,
    answer=0xbfffe160 "\301\376", 'B' <repeats 198 times>..., anslen=2048,
    answerp=0xbfffe9ac, answerp2=0xbfffe9a8, nanswerp2=0xbfffe9a4,
    resplen2=0xbfffe9a0, answerp2_malloced=0xbfffe99c) at res_query.c:264
264             if ((hp->rcode != NOERROR || ntohs(hp->ancount) == 0)
(gdb) bt
#0  __GI___libc_res_nquery (statp=0xb7fd6340 <_res@GLIBC_2.0>,
    name=0x8048653 "foo.bar.google.com", class=1, type=62321,
    answer=0xbfffe160 "\301\376", 'B' <repeats 198 times>..., anslen=2048,
    answerp=0xbfffe9ac, answerp2=0xbfffe9a8, nanswerp2=0xbfffe9a4,
    resplen2=0xbfffe9a0, answerp2_malloced=0xbfffe99c) at res_query.c:264
#1  0xb7dfc9a2 in __libc_res_nquerydomain (
    statp=statp@entry=0xb7fd6340 <_res@GLIBC_2.0>, name=<optimized out>,
    name@entry=0x8048653 "foo.bar.google.com", domain=<optimized out>,
    class=1, type=62321,
    answer=0xbfffe160 "\301\376", 'B' <repeats 198 times>..., anslen=2048,
    answerp=0xbfffe9ac, answerp2=0xbfffe9a8, nanswerp2=0xbfffe9a4,
    resplen2=0xbfffe9a0, answerp2_malloced=0xbfffe99c) at res_query.c:592
#2  0xb7dfcd85 in __GI___libc_res_nsearch (statp=0xb7fd6340 <_res@GLIBC_2.0>,
    name=0x8048653 "foo.bar.google.com", class=1, type=62321,
    answer=0xbfffe160 "\301\376", 'B' <repeats 198 times>..., anslen=2048,
    answerp=0xbfffe9ac, answerp2=0xbfffe9a8, nanswerp2=0xbfffe9a4,
    resplen2=0xbfffe9a0, answerp2_malloced=0xbfffe99c) at res_query.c:381
#3  0xb7e243b4 in _nss_dns_gethostbyname4_r (
    name=0x42424242 <error: Cannot access memory at address 0x42424242>,
    pat=0x42424242,
    buffer=0x42424242 <error: Cannot access memory at address 0x42424242>,
    buflen=1111638594, errnop=0x42424242, herrnop=0x42424242, ttlp=0x42424242)
    at nss_dns/dns-host.c:315
---Type <return> to continue, or q <return> to quit---
#4  0x42424242 in ?? ()
#5  0x42424242 in ?? ()
```

IV、触发流程

① 库函数调用流程

getaddrinfo 解析 URL 时，库函数的调用流程如下：

- 首先 getaddrinf 函数开始，getaddrinfo 函数在 resolv/nss_dns/dns-host.c 中 _nss_dns_gethostbyname4_r 会调用 alloca 在栈上分配了 2048 个字节。

```
306   querybuf *orig_host_buffer;
307   host_buffer.buf = orig_host_buffer = (querybuf *) alloca (2048);
308   u_char *ans2p = NULL;
309   int nans2p = 0;
310   int resplen2 = 0;
311   int ans2p_malloced = 0;
312
313   int olderr = errno;
314   enum nss_status status;
315   int n =  __libc_res_nsearch (&_res, name, C_IN, T_UNSPEC,
316            host_buffer.buf->buf, 2048, &host_buffer.ptr,
317            &ans2p, &nans2p, &resplen2, &ans2p_malloced);
```

alloca 函数在栈分配 2048 个字节，用于存放 DNS 服务器的响应数据。__libc_res_nsearch 函数还传递了 ans2p，nans2p，resplen2 三个参数，存放 DNS 服务器的响应数据。ans2p 用于返回存放第二份响应数据包缓冲区地址，nans2p 返回存放第二份响应数据包的扩充缓冲区的大小，resplen2 返回第二份响应数据包数据包的大小。函数的返回值是第一份响应数据包的大小。

getaddrinfo 函数若传递 AF_UNSPEC 参数，会同时进行 IPv4,IPv6 查询，分配组建 IPv4 和 IPv6 的数据包发送和接受。

- libc_res_nsearch 继续调用 libc_res_nquery 函数，libc_res_nquery 在调用 libc_res_nsend 函数。__libc_res_nsend 用于发送和接受 DNS 相关的数据包。

```
1    __libc_res_nsend(statp, query1, nquery1, query2, nquery2, answer, anslen,
     answerp, answerp2, nanswerp2, resplen2)
```

- __libc_res_nsend 先后调用 send_dg 和 send_vc 函数和 DNS 服务器交互，TCP 场景下调用 send_vc，UDP 调用 send_dg。

Tips: 函数位置在 库函数调用流程图 中都有标注。

② 漏洞成因

(1) 深入分析 send_dg 函数。

```
1    send_dg(statp, buf, buflen, buf2, buflen2, &ans, &anssiz, &terrno, ns, &v_circuit,
      &gotsomewhere, ansp, ansp2, nansp2, resplen2)
```

send_dg 首先调用 __sendmmsg 发送 buf, buflen, buf2, buflen2 中的数据包到 DNS 服务器。buf, buflen, buf2, buflen2 是对应的查询消息。再调用 recvfrom 接受从 DNS 的响应包，而问题就出现在这段代码。

(2) 接着分析 send_dg 函数中 recvfrom 的使用

thisansp 变量标识接受数据缓冲区的地址
thisanssizp 变量标识接受数据缓冲区的大小。

```
1    *thisresplenp = recvfrom(pfd[0].fd, (char*)*thisansp, *thisanssizp, 0,
     (struct sockaddr *)&from, &fromlen);
```

(3) 继续分析 send_dg 对 thisansp，thisanssizp 变量的处理逻辑。

第一次收到数据包，使用之前在栈上分配的 2048 个字节，代码处理如下：

```
1197  /*第一次收到数据包时，recvreps1 和 recvresp2 都为 0 ，buf2 == NULL，进入此分支*/
1198  if ((recvresp1 | recvresp2) == 0 || buf2 == NULL) {
1199      thisanssizp = anssizp;
1200      thisansp = anscp ?: ansp;
1201      assert (anscp != NULL || ansp2 == NULL);
1202      thisresplenp = &resplen;
1203  } else
```

当第二次收到数据包时，处理如下(/opt/glibc220/resolv/nss_dns/res_send.c:line 1198 ~ 1247)：

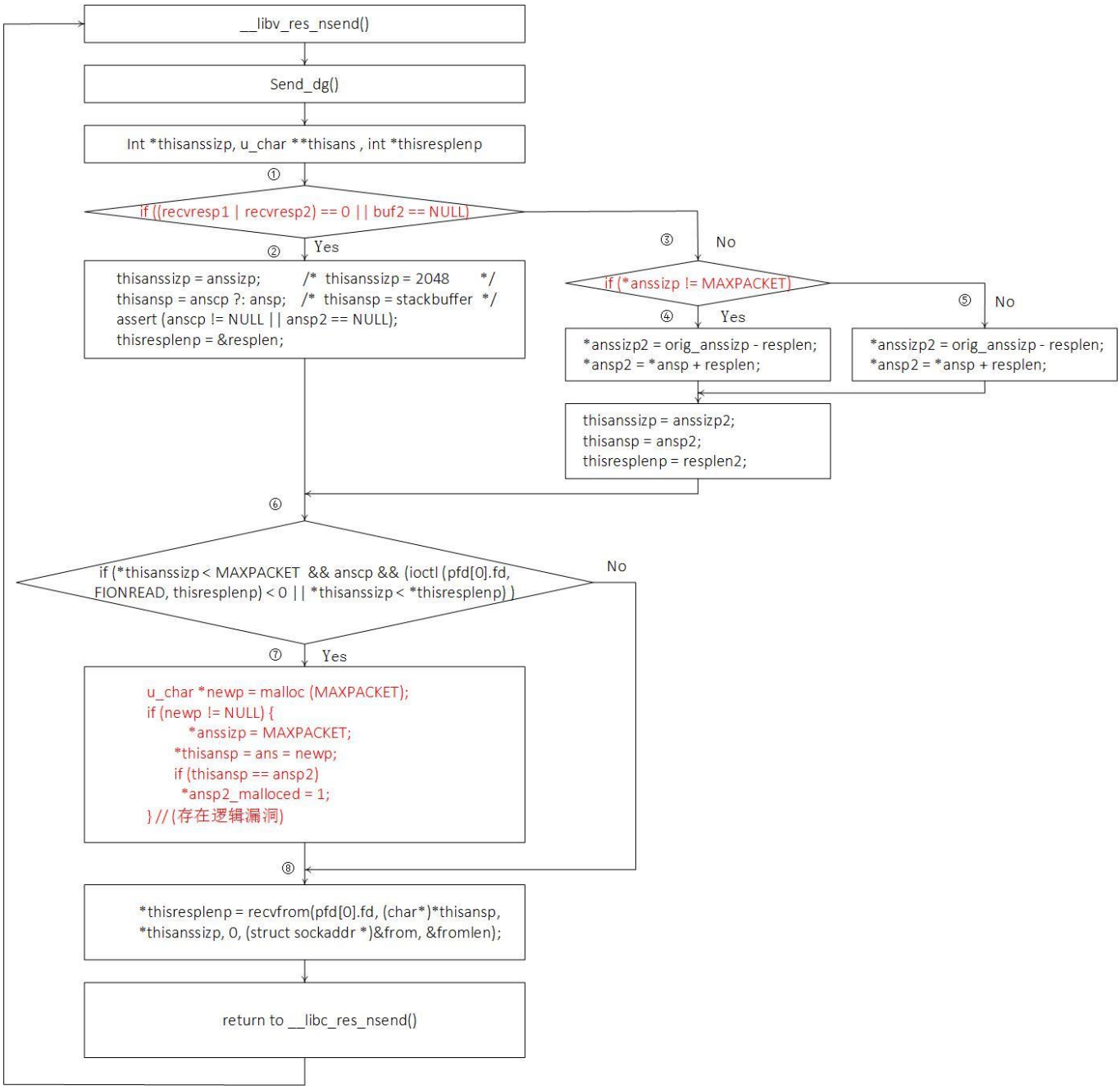
```
if ((recvresp1 | recvresp2) == 0 || buf2 == NULL) {
thisanssizp = anssizp;
thisansp = anscp ?: ansp;
assert (anscp != NULL || ansp2 == NULL);
thisresplenp = &resplen;
}else{
/* 第二次接收到数据包时，进入此分支 */
if (*anssizp != MAXPACKET) {
/* 判断第一个缓冲区长度 anssizp 是否是 65536，
    如果不是，表示 2048 个缓冲区在接受第一个数据包的时候足够，
    因此第二个缓冲区继续使用 2048 个缓冲区的剩余部分 */

        *anssizp2 = orig_anssizp - resplen;
        *ansp2 = *ansp + resplen;
    } else {
/* anssizp 等于 65536，说明之前的接受数据包大于 2048 个字节，
    栈中分配的空间不足以存放，但是第二次查询的数据有可能小于 2048，
    因此尝试使用栈中的内存保存响应包 */

        *anssizp2 = orig_anssizp;
        *ansp2 = *ansp;
    }
/* 修改 thisanssizp thisansp thisresplenp，
    分别表示调用 recvfrom 函数接受缓冲区的大小、接受缓冲区地址、接受到的数据包长度 */
thisanssizp = anssizp2;
thisansp = ansp2;
thisresplenp = resplen2;

}

if (*thisanssizp < maxpacket  span>
/* 判断接受的数据包，thisanssizp 是否足够存放，
    如果不够的话，调用 malloc 从堆上分配 65536 个字节，用来存放接受到的数据包 */
    && anscp && (ioctl (pfd[0].fd, FIONREAD, thisresplenp) < 0 || *thisanssizp <  thisr
esplenp span>

    u_char *newp = malloc (MAXPACKET);
if (newp != NULL) {
        *anssizp = MAXPACKET;    /* 修改 anssizp 变量表示已经从堆上分配了内存 */
        *thisansp =ans = newp;     /* 修改 thisansp 变量，本地 recvfrom 使用新分配的内存进行
存放 */
if (thisansp == ansp2)
            *ansp2_malloced =1;
    }
}
```

处理第二次接收到的数据包时，存在逻辑漏洞。

u_char *newp = malloc (MAXPACKET)；在堆上分配了 65536 字节区域，但

- 在使用新分配的内存是，修改了 thisansp 变量，但是没有修改 thisanssizp 变量为新分配的 malloc 内存的大小。
- 更新了 anssizp 标识第一个缓存区的大小，但是没有更新 ansp 变量，ansp 还是指向之前在栈上分配的 2048 个字节。

③ 漏洞触发场景



- 程序调用 getaddrinfo 函数，传递 AF_UNSPEC 参数进行 DNS 查询，并发查询 IPv4,IPv6。
- 第一次的 DNS 响应数据包是 2048，正好使用完了在栈上分配的 2048 个字节。

1. 执行路径：① ② ⑥ ⑧ ；
2. 输入：服务器发送 2048 字节的响应包； ansp = stackbuffer; anssizp = 2048

3. 影响变量：thisanssizp = 2048; thisansp = ansp = stackbuffer; thisresplenp = 2048;
4. 结果：thisresplenp = recvfrom(thisansp, *thisanssizp) ; // recvfrom(stackbuffer, 2048);

- 接受第二次 DNS 响应数据包，由于之前第一个数据包已经使用完 2048 个字节，所以代码会走到 malloc 流程从堆上分配内存，但是由于前面提到的 bug，thisanssizp 没有被更新，而 thisanssizp 在这种场景下为 0，会导致 recvfrom 返回失败，导致 send_dg 直接退出。这个时候，ansp 指向栈上的 2048 个内存区，但是 anssizp 被修改为 65536。
  1. 执行路径：① ③ ④ ⑥ ⑦ ⑧;
  2. 输入：服务器发送 10000 字节的数据，ansp = stackbuffer; anssizp = 2048;
  3. 影响变量：thisanssizp = 0; thisansp = stackbuffer; thisresplenp = 10000
  4. 结果： heapbuffer = malloc(MAXPACKET); anssizp = MAXPACKET; thisansp = heapbuffer; thisresplenp = recvfrom(thisansp, thisanssizp) ; // recvfrom(heapbuffer, MAXPACKET);

- send_dg 再次被调用，这个时候接受第三个 DNS 响应数据包，ansp 指向栈上的 2048 个内存区，anssizp 被修改为 65536. 这个时候如果接受超过 2048 个数据包，会导致栈溢出。因此攻击者可以构造一个 65536 的数据包，前面 2048 个字节是正规的 DNS 数据，后面 63487 个字节利用栈溢出执行自己的代码。
  1. 执行路径：① ② ⑥ ⑧ ;
  2. 输入：服务器发送>2048 字节数据，ansp = stackbuffer, anssizp = MAXPACKET;
  3. 影响变量：thisanssizp = MAXPACKET, *thisansp = ansp = stackuffer;
  4. 结果： thisresplenp = recvfrom(thisansp, thisanssizp) ; // recvfrom(stackbuffer, MAXPACKET);
     MAXPACKET = 65535 > 2048 造成栈溢出。

V、参考文献

glibc - getaddrinfo Stack-Based Buffer Overflow

Proof of concept for CVE-2015-7547

glibc getaddrinfo() stack-based buffer overflow

CVE-2015-7547 简单分析与调试

Linux Glibc 函数库漏洞分析(CVE-2015-7547)

CVE-2015-7547 的漏洞分析

解读 | 一个 Linux 漏洞火了，什么情况？严重么？

linux 下编译安装 glibc