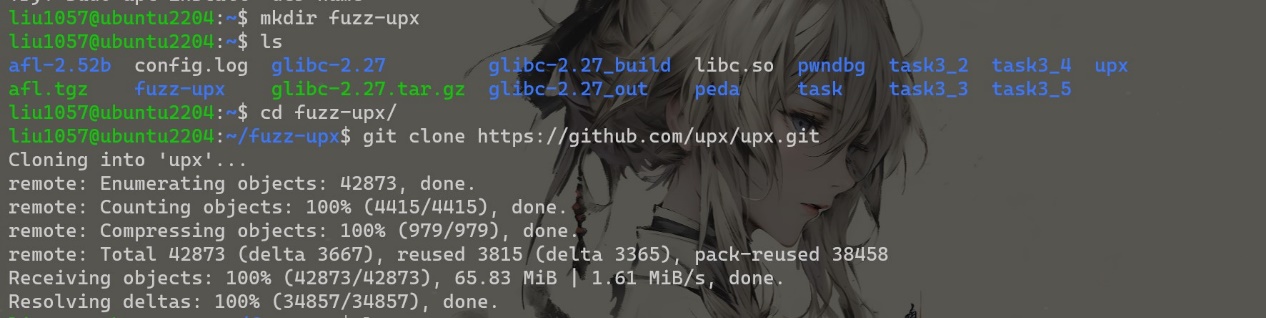
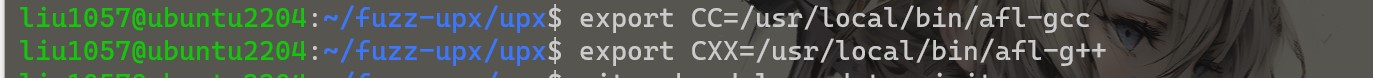
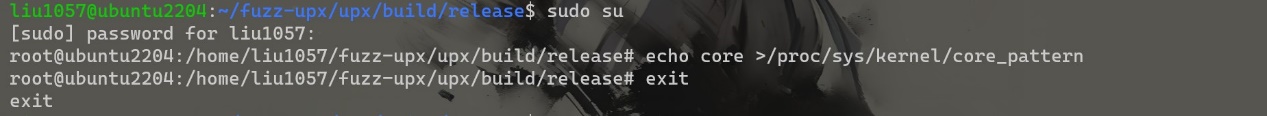
创建fuzz-upx目录用来漏洞发现，从github上拉取upx最新版源码



修改环境变量的gcc和g++为afl-fuzz工具的编译器，使得编译时对upx程序进行插桩



关闭核心转储



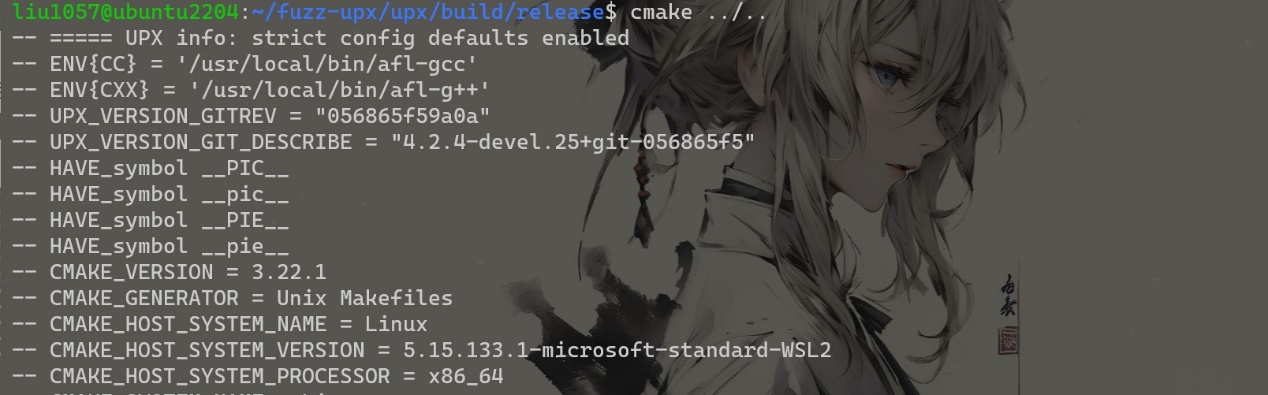
下载项目需要的依赖包

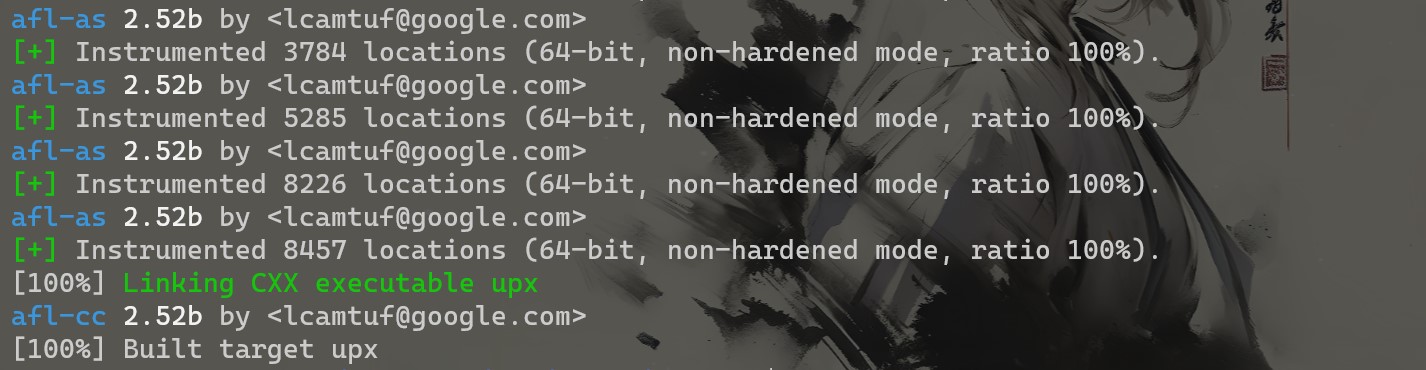


创建build/release目录用于编译项目

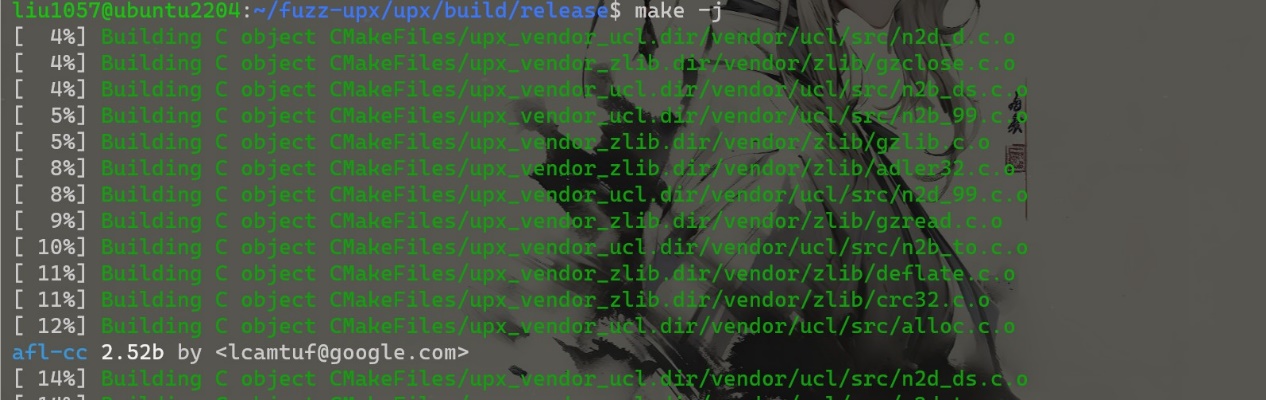


cmake编译项目

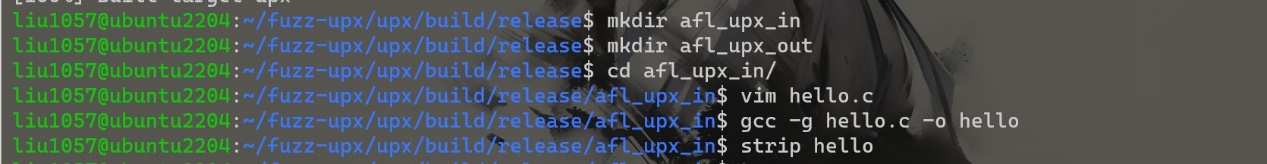




make构建项目

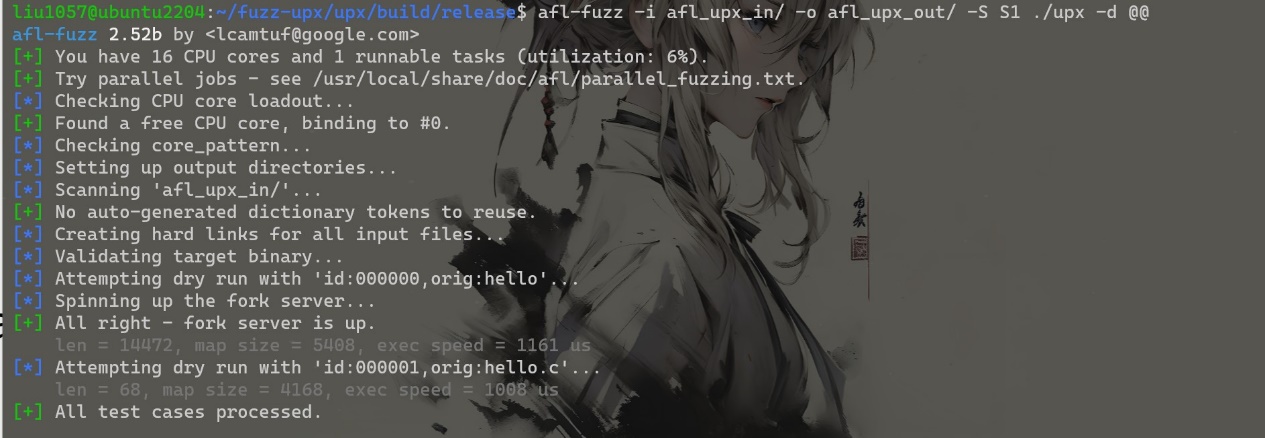


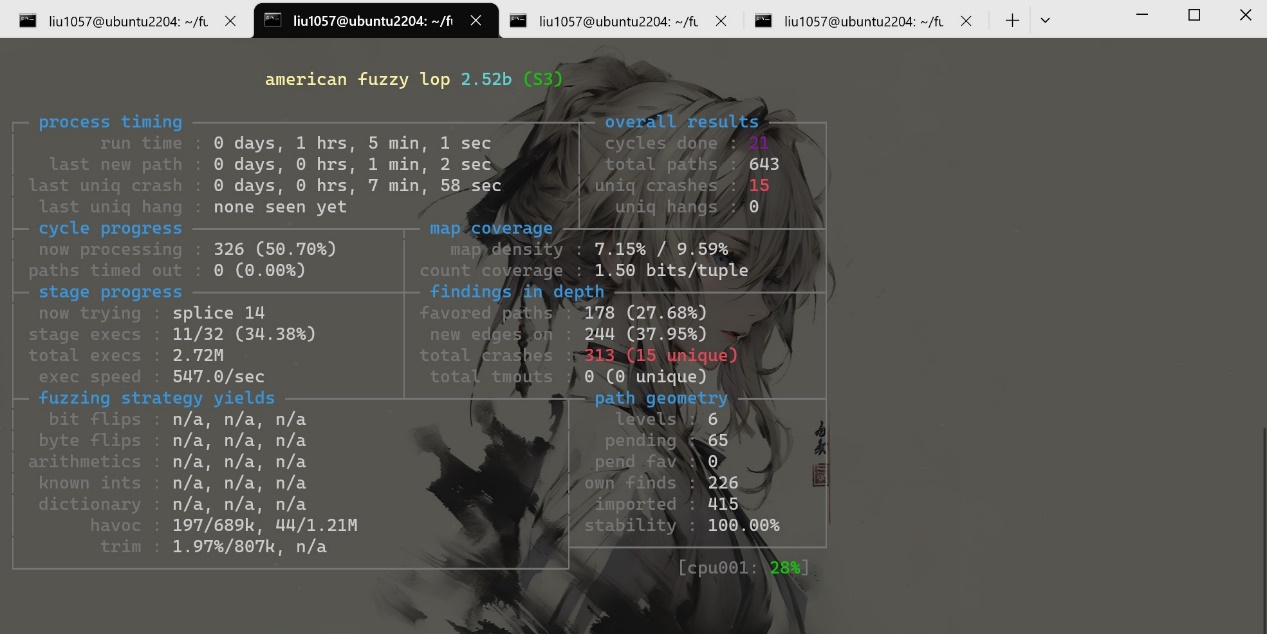
创建in、out目录，分别存储fuzz的样本及fuzz出的crash还有覆盖率等信息，编写一个简单程序，gcc编译并strip掉符号作为初始样本，放入afl\_upx\_in目录



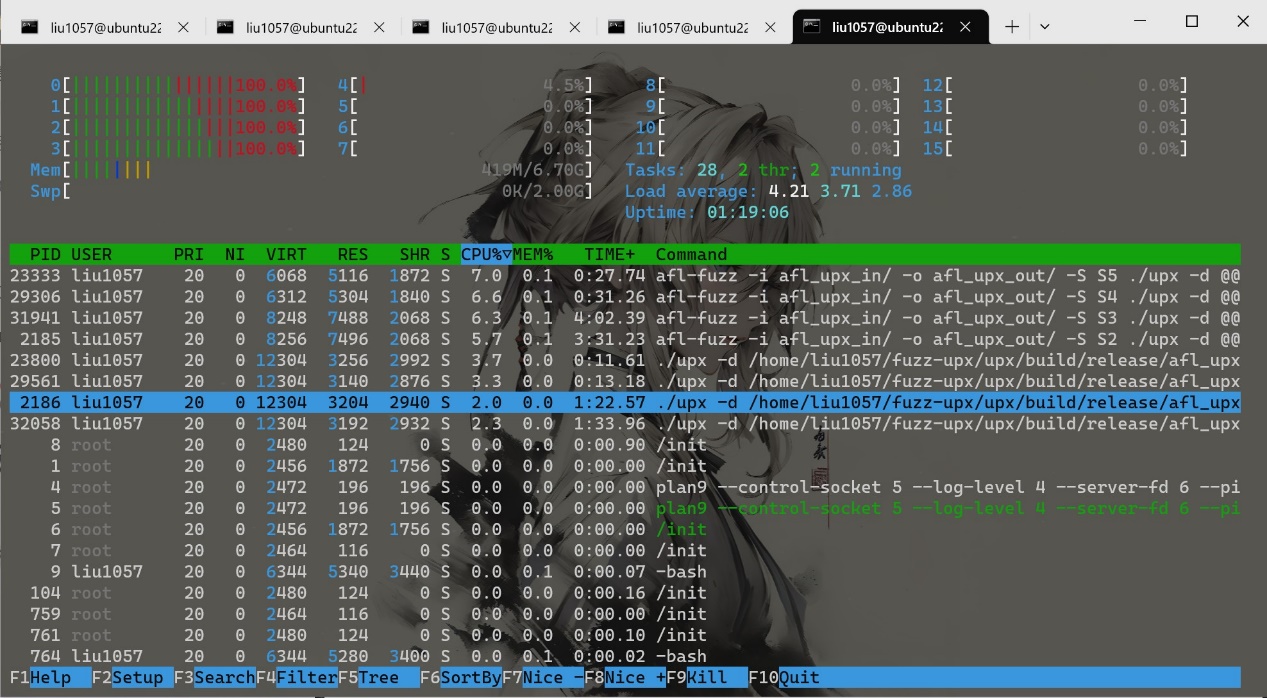


启动fuzz，等待出crash样本

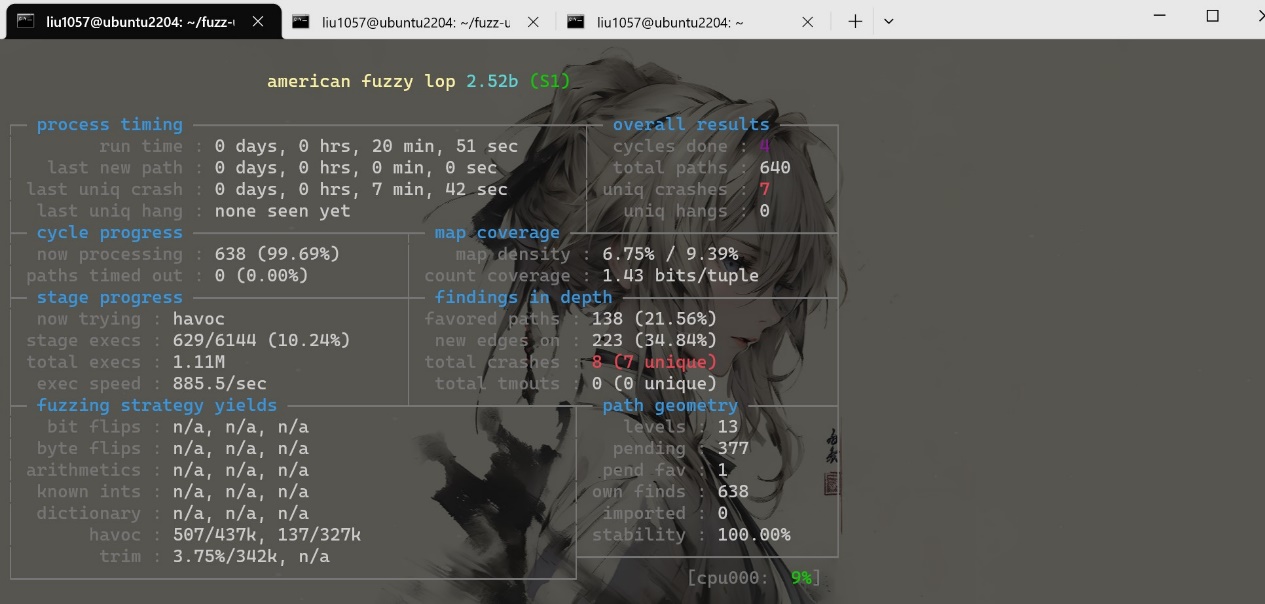




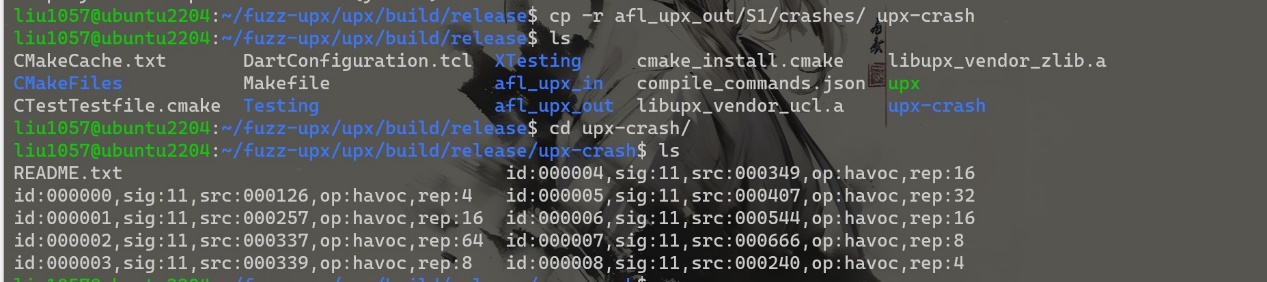
使用htop查看查看系统资源的使用情况。



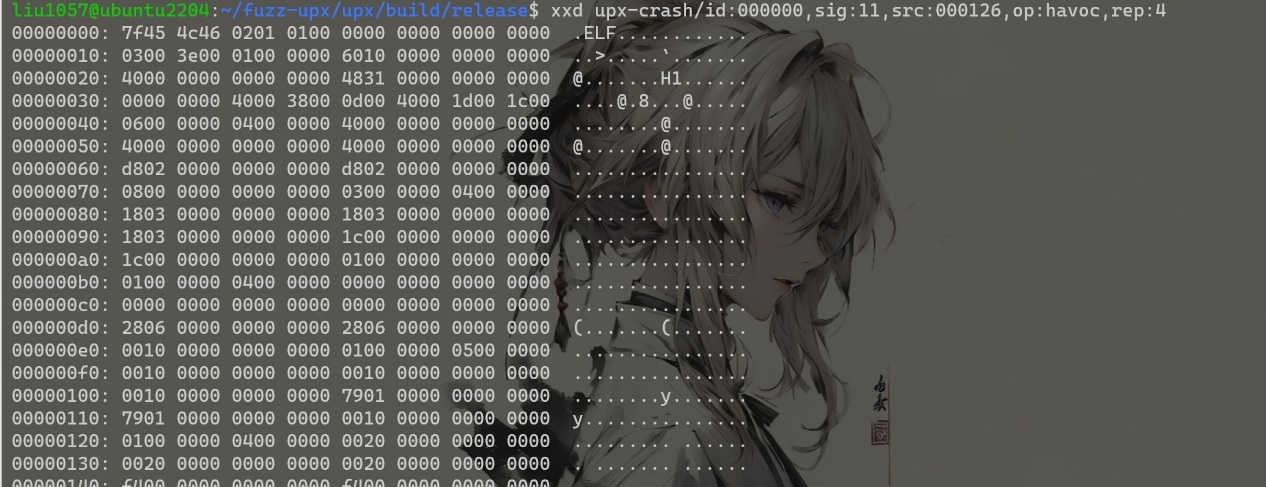
运行20分钟后，可以得到8个crash



先将得到的crash备份

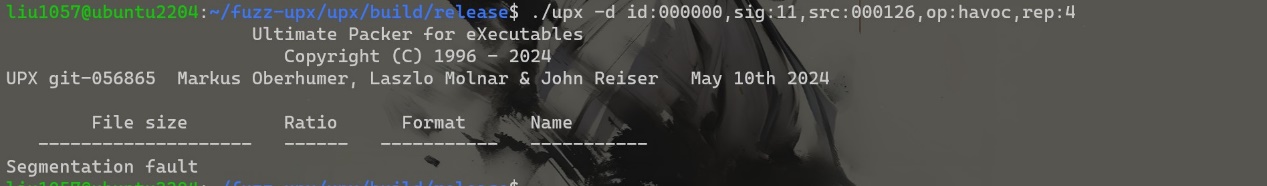


查看id0的程序内容

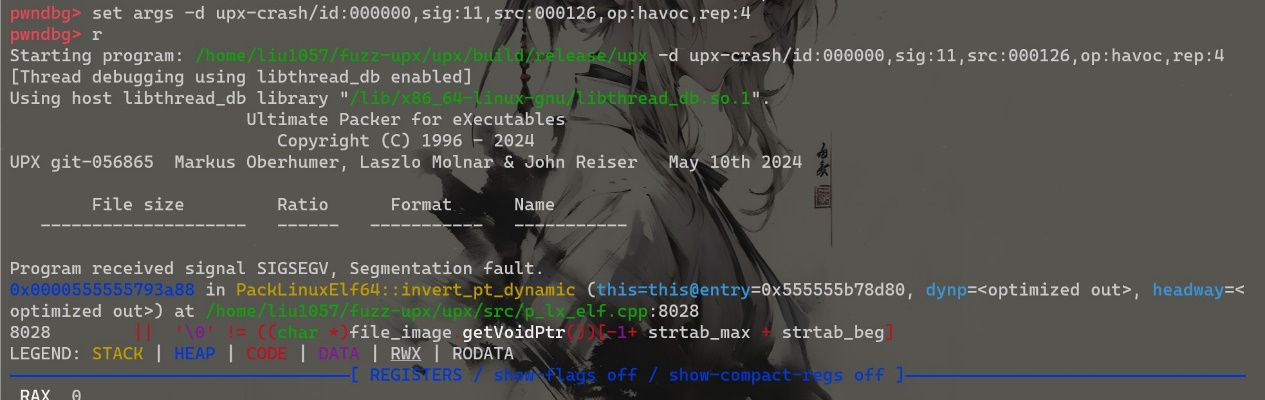


使用upx对其中一个程序加壳，出现Segmentation fault错误

id:000000,sig:11,src:000126,op:havoc,rep:4



使用gdb调试upx程序，设置输入，同样是Segmentation fault





可以看到错误发生在 PackLinuxElf64::invert\_pt\_dynamic 函数中，具体位置在源代码的第8028行。问题出现在尝试访问字符串表 (strtab) 的终结位置，确切地说是这一行代码:

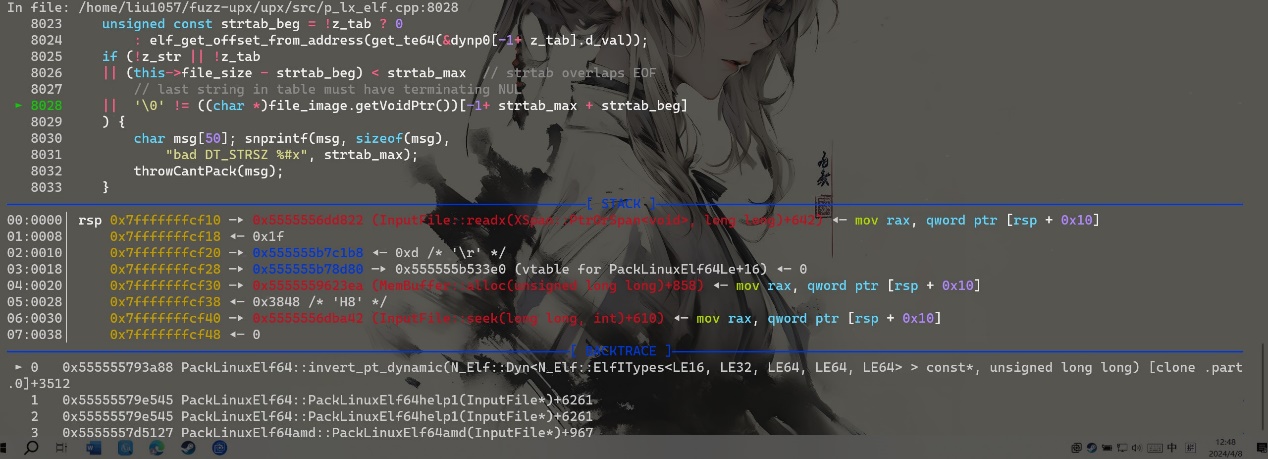
|| '\0' != ((char \*)file\_image.getVoidPtr())[-1 + strtab\_max + strtab\_beg]

这行代码在检查字符串表的最后一个字符是否为 null 字符 (\0)，这是为了验证字符串表的格式正确性。如果字符串表的最后一个字符不是 null 字符，这通常表明有问题。

分析错误原因

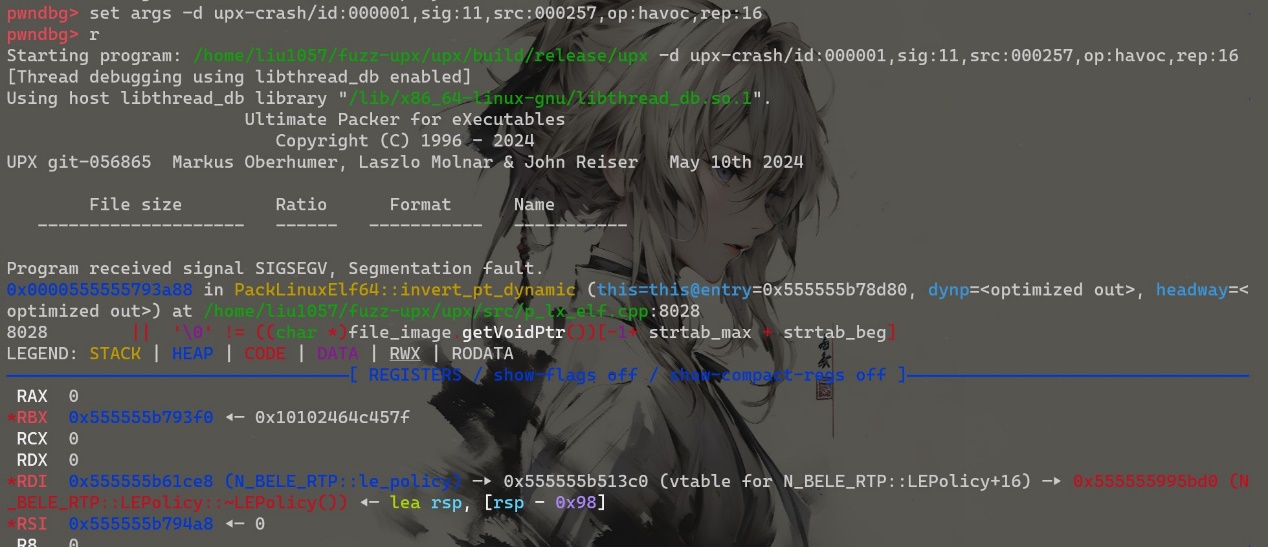
（1）内存访问越界: 这行代码中的 [-1 + strtab\_max + strtab\_beg] 可能导致越界访问。如果 strtab\_max + strtab\_beg 超出了实际映射到内存中的文件大小，那么访问这个地址就会导致段错误。strtab\_beg 是字符串表的起始位置，strtab\_max 是字符串表的大小。

（2）指针计算错误: 这行代码使用 -1 索引来访问字符串，这有可能是因为数组索引错误或计算不正确导致的。需要检查 strtab\_max 和 strtab\_beg 的计算是否正确。

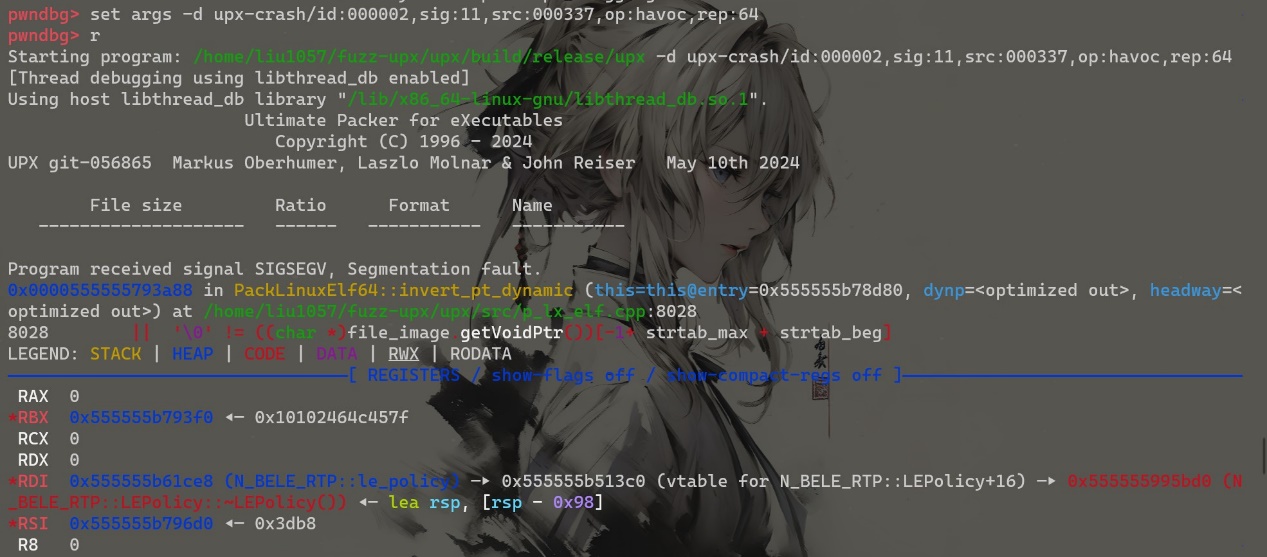


接着将其他七个crash作为upx的输入程序，可以看到出现错误的地方都相同，这几个crash都是相同类型的

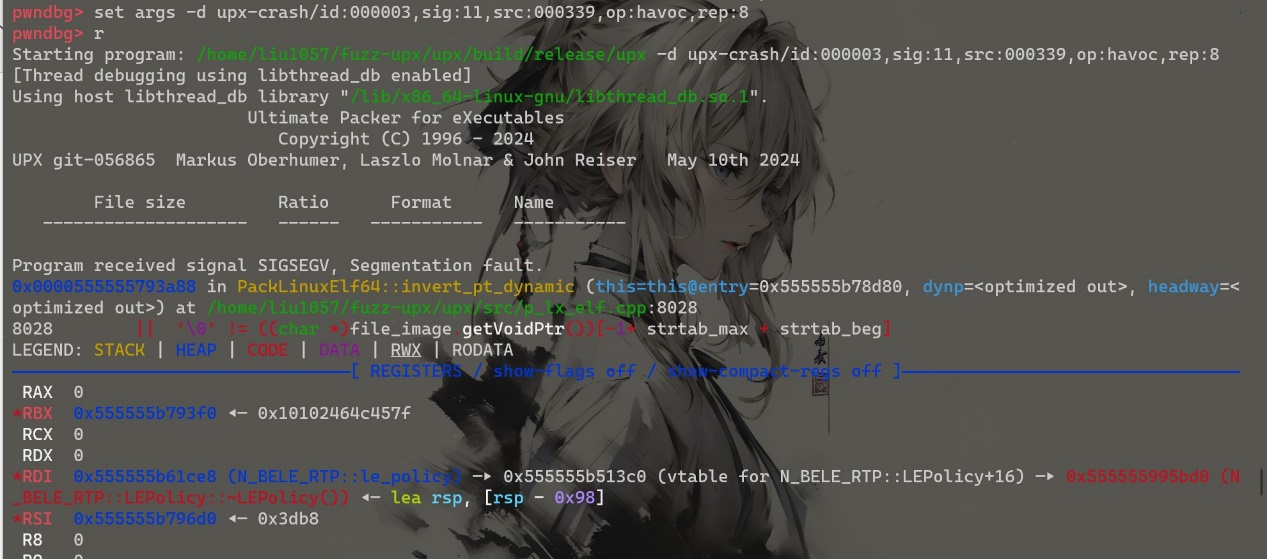
id:000001,sig:11,src:000257,op:havoc,rep:16



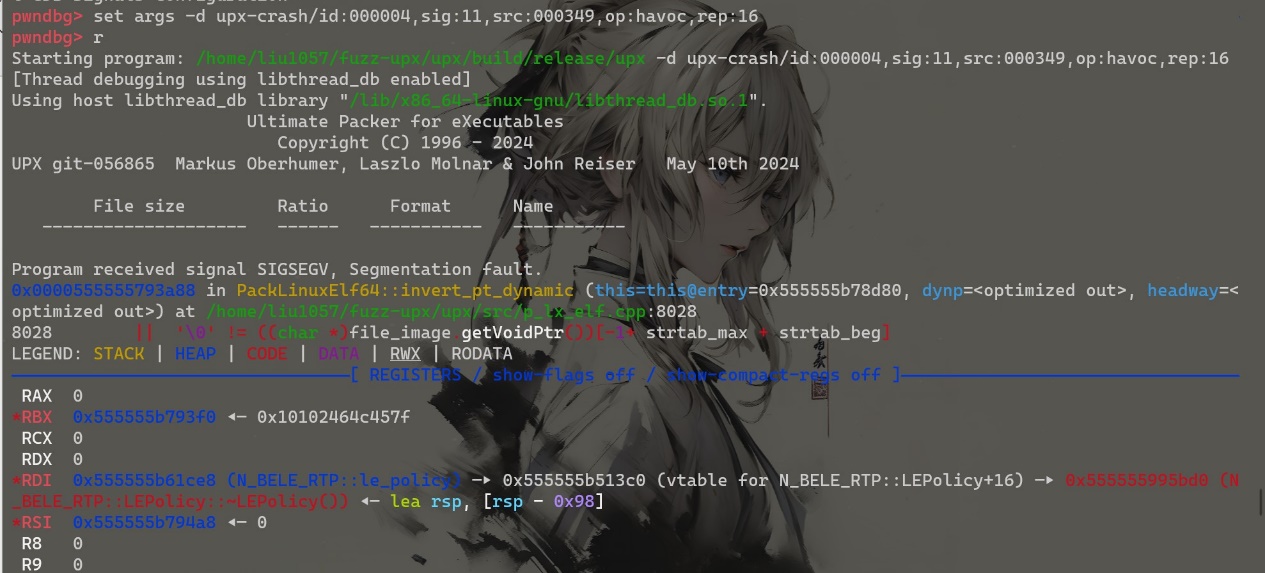
id:000002,sig:11,src:000337,op:havoc,rep:64



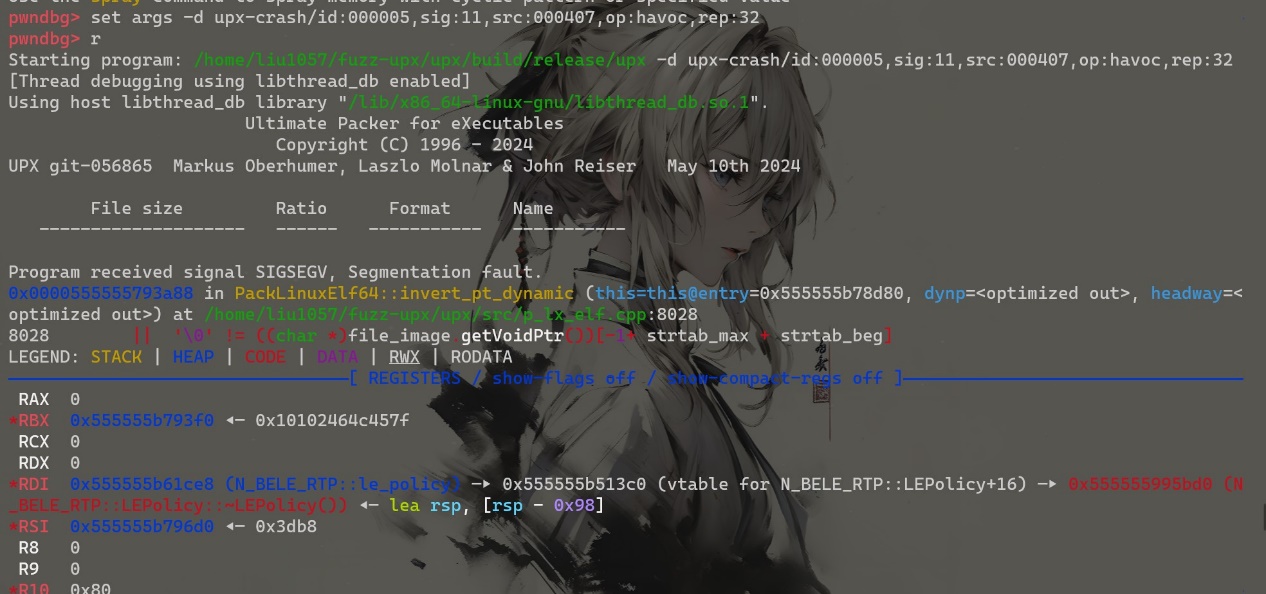
id:000003,sig:11,src:000339,op:havoc,rep:8



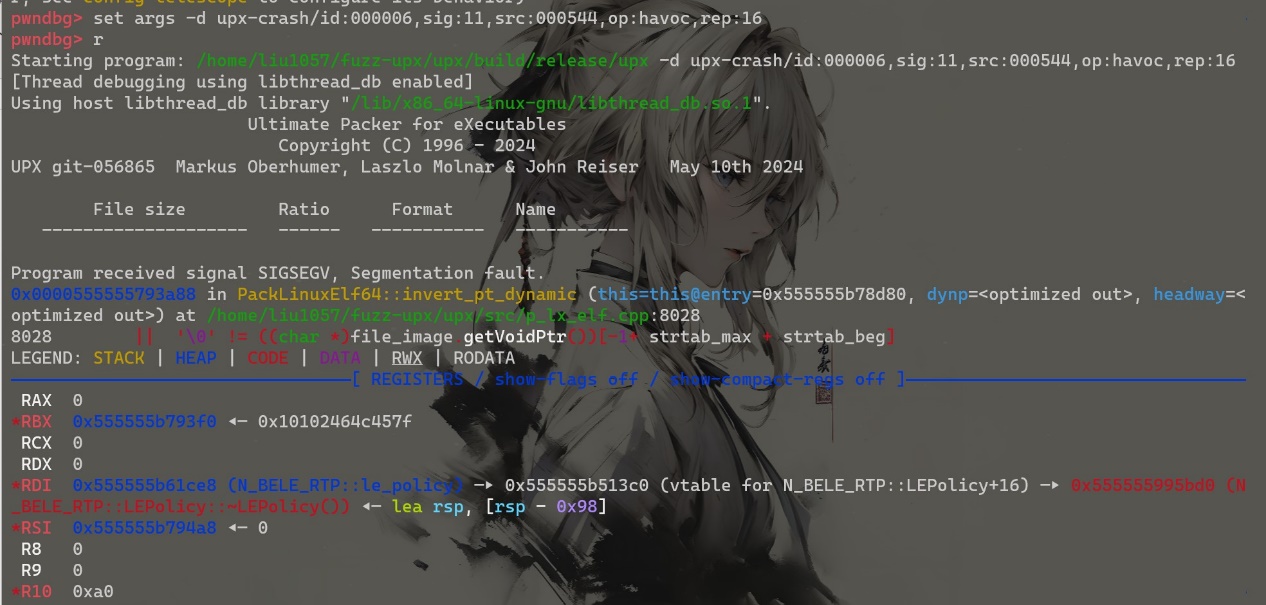
id:000004,sig:11,src:000349,op:havoc,rep:16



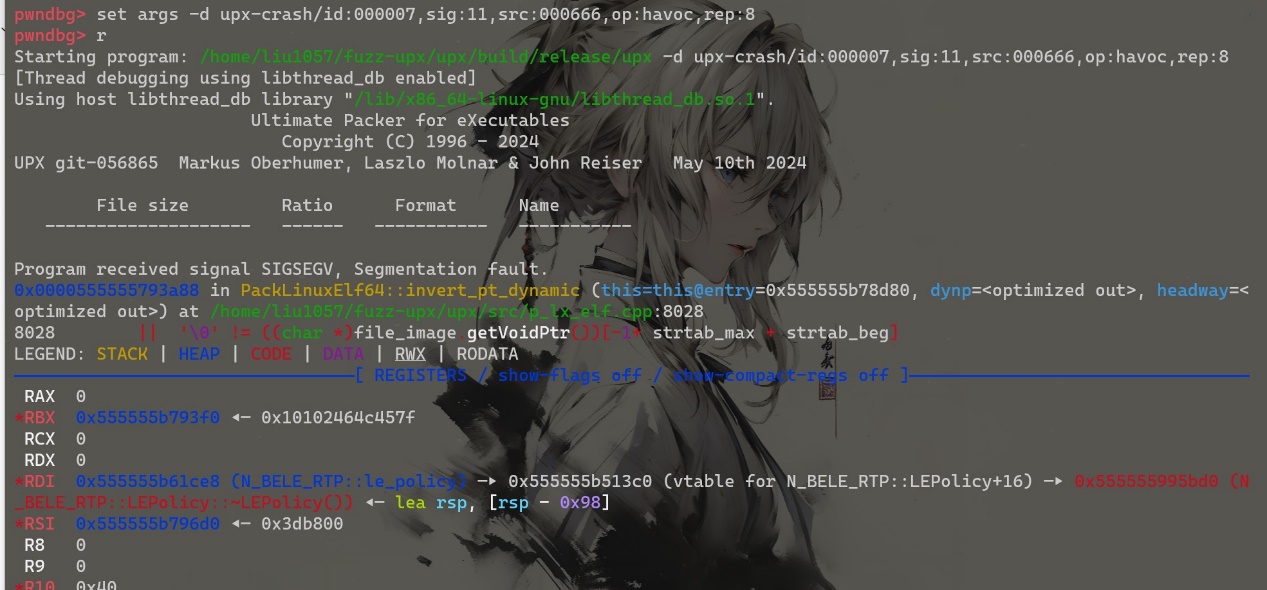
id:000005,sig:11,src:000407,op:havoc,rep:32



id:000006,sig:11,src:000544,op:havoc,rep:16



id:000007,sig:11,src:000666,op:havoc,rep:8



id:000008,sig:11,src:000240,op:havoc,rep:4

