**GDB的使用**

姓名：徐昕妍 学号：2023K8009970008 提交日期：2024年10月27日

一、摘要

本实验旨在通过gdb 调试练习 chall，使其能通过两次密码检测，从而理解掌握gdb调试的使用方法。

二、实验步骤

练习：

完成 gdb 调试练习 chall，要求能够通过两次密码检测（截图程序通过的返回信息），体现你的运行调试步骤（报告中体现 gdb 调试指令、调试思路与运行结果）。

1. 查找程序代码中的所有函数。

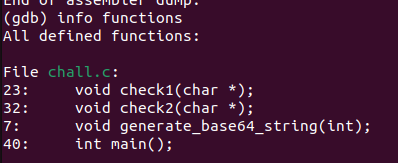
在终端启动gdb后输入info functions列出所有函数。函数如下：

23: void check1(char \*);

32: void check2(char \*);

7: void generate\_base64\_string(int);

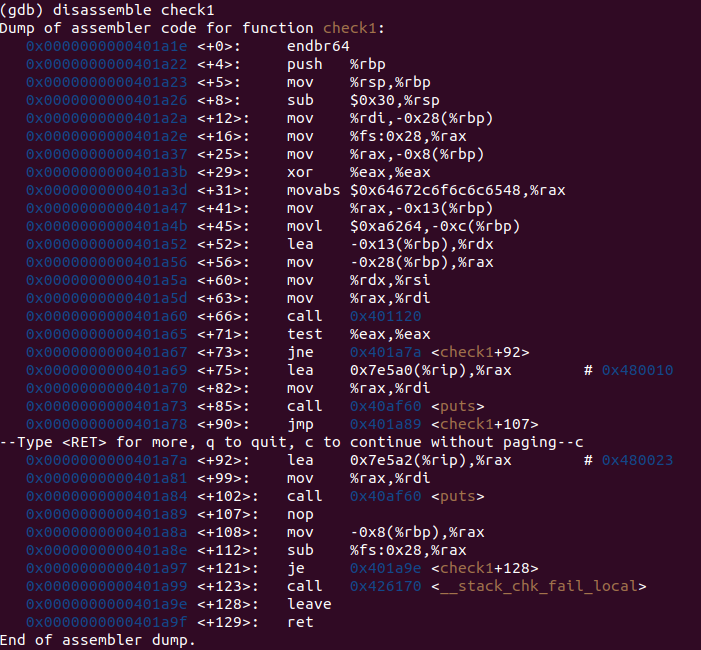
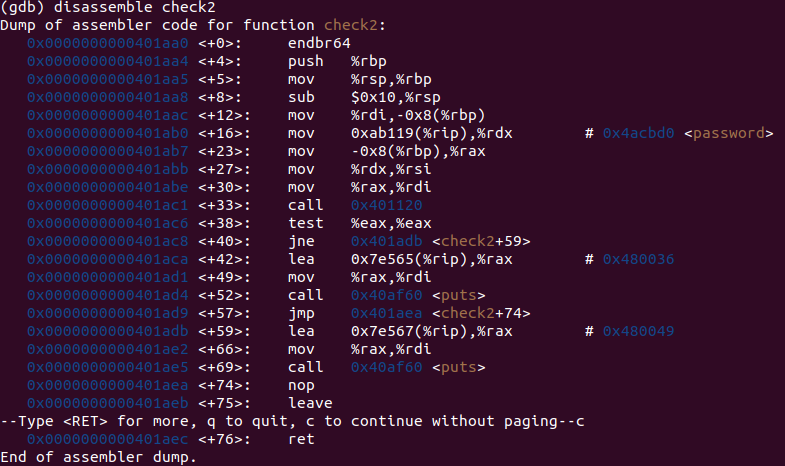
40: int main();



可以看出check1、check2就是我们所查找的检验函数。

1. 获取检验函数汇编代码。

输入disassemble check1、disassemble check2获得check1、check2汇编代码。具体结果如下：

1. 分析汇编代码。

借助AI分析可知，对于check1函数的汇编代码：

(1)函数开始

0x0000000000401a1e <+0>: endbr64

0x0000000000401a22 <+4>: push %rbp

0x0000000000401a23 <+5>: mov %rsp,%rbp

0x0000000000401a26 <+8>: sub $0x30,%rsp

这部分是函数的标准序言，它设置了新的堆栈帧。

(2)保存参数和寄存器

0x0000000000401a2e <+16>: mov %fs:0x28,%rax

0x0000000000401a37 <+25>: mov %rax,-0x8(%rbp)

0x0000000000401a2a <+12>: mov %rdi,-0x28(%rbp)

保存了指向参数的寄存器 %rdi 和一个在 %fs:0x28 位置的值（可能是用于安全检查的值）。

(3)初始化局部变量

0x0000000000401a3b <+29>: xor %eax,%eax

0x0000000000401a3d <+31>: movabs $0x64672c6f6c6c6548,%rax

0x0000000000401a47 <+41>: mov %rax,-0x13(%rbp)

0x0000000000401a4b <+45>: movl $0xa6264,-0xc(%rbp)

将 %eax 清零，并设置了两个局部变量。第一个局部变量是一个字符串，第二个局部变量是一个数字。

(4)调用另一个函数进行字符串比较

0x0000000000401a52 <+52>: lea -0x13(%rbp),%rdx

0x0000000000401a56 <+56>: mov -0x28(%rbp),%rax

0x0000000000401a5a <+60>: mov %rdx,%rsi

0x0000000000401a5d <+63>: mov %rax,%rdi

0x0000000000401a60 <+66>: call 0x401120

设置了两个参数 %rsi 和 %rdi，然后调用了一个函数（可能是字符串比较函数）。%rsi 指向局部变量中的字符串，而 %rdi 指向函数参数。

(5)检查返回值

0x0000000000401a65 <+71>: test %eax,%eax

0x0000000000401a67 <+73>: jne 0x401a7a <check1+92>

检查调用返回的 %eax 寄存器的值。如果 %eax 不为零，则跳转到地址 0x401a7a。

(6)输出消息

0x0000000000401a69 <+75>: lea 0x7e5a0(%rip),%rax # 0x480010

0x0000000000401a70 <+82>: mov %rax,%rdi

0x0000000000401a73 <+85>: call 0x40af60 <puts>

如果 %eax 为零（即字符串匹配），则输出一个消息（可能是成功消息）。

(7)错误处理

0x0000000000401a7a <+92>: lea 0x7e5a2(%rip),%rax # 0x480023

0x0000000000401a81 <+99>: mov %rax,%rdi

0x0000000000401a84 <+102>: call 0x40af60 <puts>

如果跳转发生（即字符串不匹配），则输出另一个消息（可能是错误消息）。

(8)函数结束

(9)安全检查

0x0000000000401a97 <+121>: je 0x401a9e <check1+128>

0x0000000000401a99 <+123>: call 0x426170 <\_\_stack\_chk\_fail\_local>

函数执行了一个安全检查。它比较了之前在函数开始时保存的 %fs:0x28 的值和当前值。如果它们相同，则跳转到 0x401a9e，否则调用 \_\_stack\_chk\_fail\_local，这通常表示栈缓冲区溢出检测失败。

(10)函数尾声

0x0000000000401a9e <+128>: leave

0x0000000000401a9f <+129>: ret

leave 指令是函数尾声的标准操作，它相当于 mov %rbp, %rsp 后跟 pop %rbp，用于恢复调用者的堆栈帧。ret 指令则从函数返回到调用者。

**总结 check1 函数的行为**

·函数开始时设置了堆栈帧。

·保存了函数参数和 %fs:0x28 的值。

·初始化了两个局部变量，一个看起来是硬编码的字符串，另一个是数字。

·调用了一个函数（可能是字符串比较函数）来比较局部变量中的字符串和函数参数。

·根据 strcmp 的结果，函数要么打印一条成功消息，要么打印一条错误消息。

·执行了一个安全检查，以确保没有栈溢出。

·恢复堆栈帧并返回。

对于check2的汇编代码：

(1)在地址 0x401ab0 处，汇编代码执行了 mov 0xab119(%rip), %rdx 指令，这表明它正在从地址 0x4acbd0 加载一个值到寄存器 %rdx。这个地址标记为 <password>，这强烈暗示了它可能就是存储密码的内存位置。

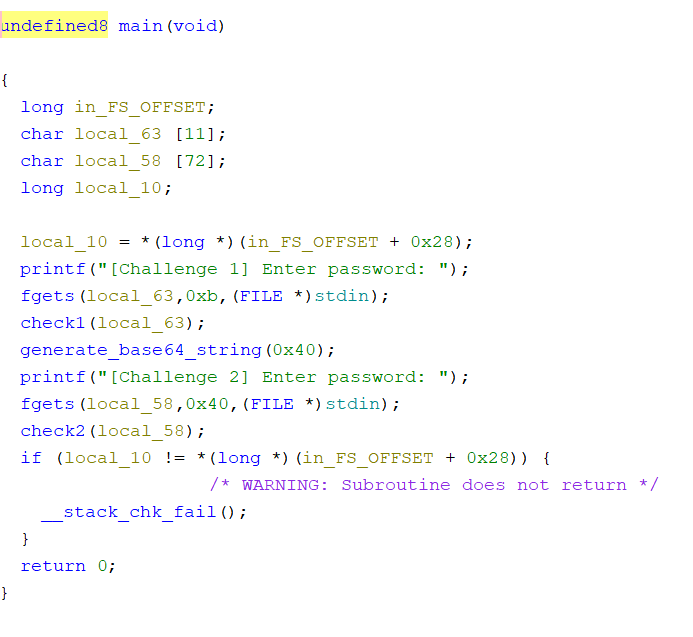
(2)接下来的指令 mov -0x8(%rbp), %rax 和 mov %rdx, %rsi，以及 mov %rax, %rdi，似乎是在准备参数以供调用另一个函数 0x401120。这可能是用于比较输入的密码与存储在 %rdx 寄存器中的密码。

(3)在 call 0x401120 之后，test %eax, %eax 指令用于检查调用返回的结果。如果 %eax 寄存器不为零，表示比较成功，则跳转到地址 0x401adb，打印出成功消息；如果为零，表示比较失败，则跳转到地址 0x401ad1，打印出失败消息。

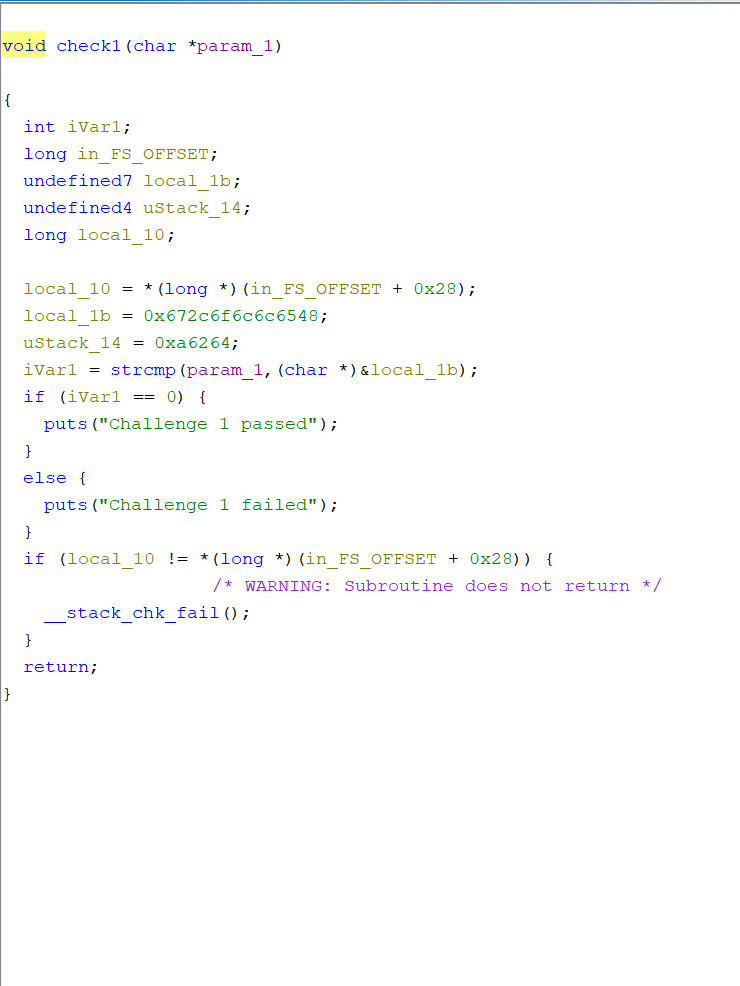
基于以上分析，密码很可能是存储在地址 0x4acbd0 处，这个地址被标记为 <password>。

1. 反汇编代码

借助反汇编工具Ghidra，我们可以得到chall程序的伪C代码。



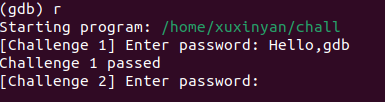
查看check1函数，我们可以得到check1函数的伪C代码，代码如下：



由伪C代码可知，local\_1b和uStack\_14应该共同组成challenge1的密码。这是由于密码的长度超出了local\_1b储存范围，所以被分成了两段。对照ASCII表（十六进制），我们可以得到：

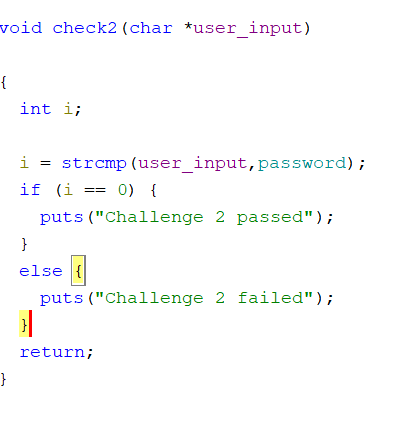
0x48=H;0x65=e;0x6c=l;0x6f=o;0x2c= , ;0x67=g;0x64=d;0x62=b;0x0a=换行

所以challenge1的密码应该是：Hello，gdb



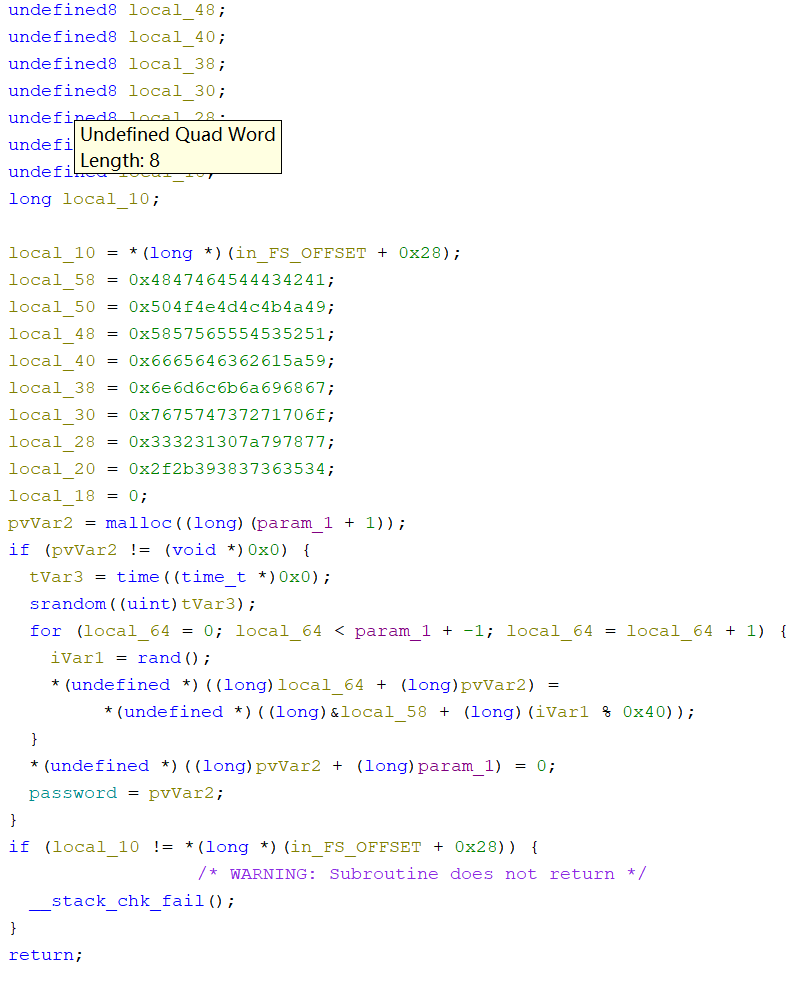
经测试，密码通过。

同理，我们可以查看check2的伪C代码：



经分析，check2是将用户输入与password进行比较，若一致则密码通过，不一致则失败。

查看chall程序中另一函数generate\_base64\_string(int)的伪C代码可知，password在generate\_base64\_string(int)中生成。故challenge2的密码应该是generate\_base64\_string(int)每次运行生成的password。



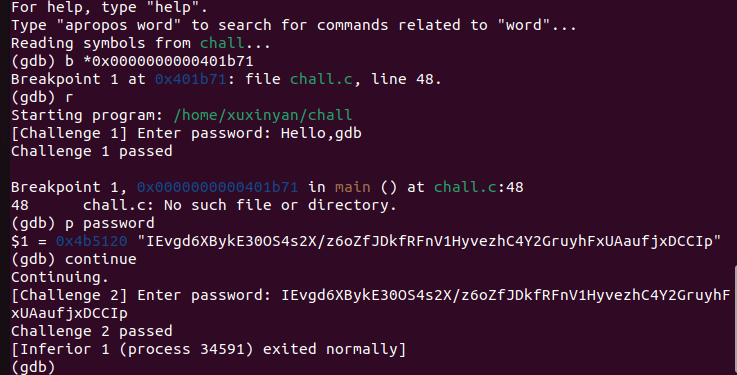
首先尝试用b \*0x0000000000401b7d在check2处打断点,运行后输入

p password，可以见到虽然打印出了密码，但继续运行程序后，我们并没有机会输入密码，程序就已经终止，且chall程序每次运行所得password均不相同，因此我们虽然得到了本次运行的密码，但challenge2失败。



查询后发现，失败原因在于我们虽然是在check2入口处打了断点，但当程序运行到断点处时，已经到达了该函数的起始地址，可以视为已经“进入该函数”，因此我们没有机会输入密码验证。

再次尝试，将断点设在check2函数之前的0x0000000000401b71处，gdb调试过程如下：



经测试，密码通过。

1. 讨论与总结

首先要感谢毕冠华同学和姜俊彦学长对我的帮助，在关键问题上为我指明思路。

在本次实验中，我遇到了诸多困难。在借助大语言模型分析汇编代码时，得到的结果始终难以直观理解。在参考了毕冠华同学的解题方法后，我选择了使用反汇编工具Ghidra来进行反汇编。初次尝试Ghidra，我选择了单独提取check1的汇编代码，并将其转化为可执行文件进行反汇编，结果并不理想。再次询问姜俊彦学长和毕冠华同学后，发现可以直接反汇编chall文件。在理解反汇编得到的伪C代码方面，再次感谢毕冠华同学给予的帮助。

顺利完成反汇编后，在设置断点的选择上也出现了一些问题。我最初选择将断点设置在check2的起始地址，在测试过程中虽然能够得到本次运行的密码，但却没有机会验证。查询后发现失败原因，最终选择将断点设置在check2函数前，成功获得并验证密码。