

PasswordManager

This challenge performs some calculations that generate the flag.

To obtain the flag, a breakpoint can be set in an instruction that uses the flag once it has been calculated. For instance, we can set a breakpoint in the LEA instruction on main+166.

The flag can be observed in the rax register, as shown in the following screenshot:

```
$rax : 0x00007fffffffdf60 -> "jctf{wh3r3s_m@y@?}"
$rbx : 0x0000000000400518 -> add BYTE PTR [rax], al
$rcx : 0x00007fffffff386 -> 0x3d4c4c4548530031 ("1"?)
$rdx : 0x7d
$rsp : 0x00007fffffffdf20 -> 0x00007ffffffe0b8 -> 0x00007ffffffe364 -> "/home/josedominguez/challenges/pw"
$rbp : 0x00007fffffffdf80 -> 0x0000000000402de0 -> <_libc_csu_init+0> endbr64
$rsi : 0x00007ffffffe0b8 -> 0x00007ffffffe364 -> "/home/josedominguez/challenges/pw"
$rdi : 0x2
$rip : 0x0000000000401daf -> <main+170> mov edx, 0x12
$r8 : 0x0
$r9 : 0x0
$r10 : 0x4
$r11 : 0x1
$r12 : 0x0000000000402e80 -> <_libc_csu_fini+0> endbr64
$r13 : 0x0
$r14 : 0x00000000004c0018 -> 0x0000000000446980 -> <_strcpy_avx2+0> endbr64
$r15 : 0x0
$eflags: [zero carry PARITY ADJUST sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x00007fffffffdf20 +0x0000: 0x00007ffffffe0b8 -> 0x00007ffffffe364 -> "/home/josedominguez/challenges/pw" - $rsp
0x00007fffffffdf28 +0x0008: 0x00000002ffffff0b8
0x00007fffffffdf30 +0x0010: 0x00007ffffffe0d0 -> 0x00007ffffffe388 -> "SHELL=/bin/bash"
0x00007fffffffdf38 +0x0018: 0x0000001200000002
0x00007fffffffdf40 +0x0020: 0x164d525e4351464f
0x00007fffffffdf48 +0x0028: 0x655c65487a561657
0x00007fffffffdf50 +0x0030: 0x00007ffffff581a -> 0x0000000000000000
0x00007fffffffdf58 +0x0038: 0x0000000000400518 -> add BYTE PTR [rax], al

0x401da4 <main+159> add rax, 0x8
0x401da8 <main+163> mov rcx, QWORD PTR [rax]
0x401dab <main+166> lea rax, [rbp-0x20]
- 0x401daf <main+170> mov edx, 0x12
0x401db4 <main+175> mov rsi, rcx
0x401db7 <main+178> mov rdi, rax
0x401dba <main+181> call 0x401d00
```

The flag for this challenge is **jctf{wh3r3s_m@y@?}**

searching-through-vines

This program emulates a limited shell. It only accepts commands containing five letters or fewer, excluding ls, cat, cd, pwd, or less.

One solution is executing the bash command, which comprises 4 letters and spawns a new shell where commands can be executed without constraints. Hence, the content of the flag.txt file can be read.

Another solution is to execute the vi command, which only comprises 2 letters, and executing shell commands from the vi console.

The flag for this challenge is **jctf{nav1gat10n_1s_k3y}**

MathTest

This program does the following:

1. Asks for a name
2. Asks for a number x such that:
 - a. $0x9000 * x < 0$
 - b. $x \geq 0$
3. Asks for a number y such that:
 - a. $0xdeadbeef * y \neq 0$
 - b. $y < 0$
4. Asks for a char z such that:
 - a. $'O' * z = 'A'$
5. Verifies $x + y + z == \text{name}$

A possible value for y is very easy to guess. For instance, -1 satisfies both conditions. However, the rest of the inputs are difficult to guess and the Z3 solver was used to calculate each of these inputs. In addition, a Python script was created since the name is treated as binary data with no encoding, which is difficult to handle using the terminal.

The flag for this challenge is `jctf{C4CLULAT0R_US3R}`

RunningOnPrayers

The binary has the following security features:

```
josedominguez@offsec:~/challenges$ pwn checksec RunningOnPrayers
[*] '/home/josedominguez/challenges/RunningOnPrayers'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX unknown - GNU_STACK missing
PIE:       No PIE (0x400000)
Stack:     Executable
RWX:       Has RWX segments
```

It is worth noting that that stack is executable, so if we find a way to inject code into the stack and hijack the execution to the inserted code, we can obtain a shell by executing the `execve` syscall with an address to `bin/sh`.

After inspecting the binary, we can see that the program has a **buffer overflow** vulnerability due to the use of the `gets` function. Additionally, the bytes `"\xFF\xE4"` are present in the program. These bytes correspond to the `jmp rsp` instruction, which can be used to execute code in the stack. Consequently, we can use the buffer overflow vulnerability to replace the return address with the address to the `jmp rsp` gadget (PIE is disabled) and insert the shell payload immediately after this instruction. See the Python script for this exploit.

The flag for this challenge is `jctf{Really_Obvious_Problem}`

StageLeft

This challenge is very similar to the previous one, except that instead of using a gets function, the fgets function is used with a size of 0x40, which does not allow to inject the entire payload after the return address in the stack. To solve this problem, the payload is inserted into the beginning of the buffer and the following instructions are inserted after the jmp rsp instruction:

- sub rsp, 48
 - Moves the stack pointer to the beginning of the buffer
- jmp rsp
 - Executes the payload contained in the buffer

The security features for the binary are the same as those of the previous one:

```
josedominguez@offsec:~/challenges$ pwn checksec StageLeft
[*] '/home/josedominguez/challenges/StageLeft'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX unknown - GNU_STACK missing
PIE: No PIE (0x400000)
Stack: Executable
RWX: Has RWX segments
```

See the Python script for this exploit.

The flag for this challenge is **jctf{Center_Of_Attention}**