CWE Challenge - Input2

Michael Mendoza

2023-01-21

Contents

Information Gathering

Ghidra

Opening the binary in Ghidra, we see that program reads in 0x13 bytes from the user and does 2 checks before running the system call to cat the flag.txt file.

```
puts("Please enter the password: ");
read(0,local_28,0x13);
iVar1 = check1(local_28);
iVar2 = check2(local_28);
if ((iVar1 != 0) && (iVar2 != 0)) {
  puts("Congratulations!");
  system("cat flag.txt");
```

Figure 1: Main Function

Check 1

Opening up the check1 function we see the following:

```
bool check1(long param_1)

{
  int i;

  for (i = 0; *(char *)(param_1 + i) != '\0'; i = i + 1) {
  }
  return i == 0xf;
}
```

Figure 2: Check 1 Function

We can see that the first check returns true if the user enters 0xf, or 16 bytes of data. It does check for a null byte, so I made sure to add this to the end of payload.

Check 2

Opening up check2 shows us the following:

```
undefined8 check2(long param_1)

{
  int i;

  i = 0;
  while( true ) {
    if (0xe < i) {
      return 1;
    }
    if ((buffers[(long)(i % 3) * 0xf + (long)i] ^ *(byte *)(param_1 + i)) != final[0xe - i]
    i = i + 1;
  }
  return 0;
}
```

Figure 3: Check 2 Function

Here we can see that the while loop takes the users input data and xor's it with one of the bytes in "buffers". If the byte doesnt correspond to the bytes in "final", then check2 will return false.

Buffers

The following is a constant set of numbers that will obtain a byte depending on what the value of i is in "buffers[(i % 3) * 0xf + i]."

Clicking into buffers shows all the bytes that are going to be xor'd with the users data.

```
                    buffers                                      XREF[3]:     Entry Point(*),
                                                                              check2:00101273(*),
                                                                              check2:0010127d(R)
 00104020 a8 66 e5        undefine...
          a2 af 8d
          7e b2 c7 ...
    00104020 a8            undefined1A8h                [0]                        XREF[3]:     Entry Point(*),
                                                                                                check2:00101273(*)
                                                                                                check2:0010127d(R)
    00104021 66            undefined166h                [1]
    00104022 e5            undefined1E5h                [2]
    00104023 a2            undefined1A2h                [3]
    00104024 af            undefined1AFh                [4]
    00104025 8d            undefined18Dh                [5]
    00104026 7e            undefined17Eh                [6]
    00104027 b2            undefined1B2h                [7]
    00104028 c7            undefined1C7h                [8]
    00104029 c6            undefined1C6h                [9]
    0010402a 98            undefined198h                [10]
    0010402b 95            undefined195h                [11]
    0010402c 65            undefined165h                [12]
    0010402d 12            undefined112h                [13]
    0010402e ee            undefined1EEh                [14]
    0010402f 45            undefined145h                [15]
    00104030 e8            undefined1E8h                [16]
    00104031 e4            undefined1E4h                [17]
    00104032 22            undefined122h                [18]
    00104033 84            undefined184h                [19]
    00104034 c6            undefined1C6h                [20]
    00104035 e9            undefined1E9h                [21]
    00104036 b9            undefined1B9h                [22]
    00104037 aa            undefined1AAh                [23]
    00104038 61            undefined161h                [24]
    00104039 73            undefined173h                [25]
    0010403a d0            undefined1D0h                [26]
    0010403b 0f            undefined10Fh                [27]
    0010403c b6            undefined1B6h                [28]
    0010403d a6            undefined1A6h                [29]
    0010403e de            undefined1DEh                [30]
    0010403f 06            undefined106h                [31]
    00104040 d4            undefined1D4h                [32]
    00104041 bb            undefined1BBh                [33]
    00104042 e0            undefined1E0h                [34]
```

Figure 4: buffers

Here we can see the bytes and there corresponding offsets, for example 0xa8 at offset 0 and 0x66 at offset 1. We can manually calculate what each byte obtained from buffers will be and xor it with the corresponding “final” byte so get what the user should enter.

Final

After clicking into final, we can see what’s expected after xoring the users input to the correct buffer bytes.

```
                    final

 00104050 84 e7 cc          undefine...
          30 10 7f
          fe 40 e2 ...
    00104050 84                undefined184h                     [0]
    00104051 e7                undefined1E7h                     [1]
    00104052 cc                undefined1CCh                     [2]
    00104053 30                undefined130h                     [3]
    00104054 10                undefined110h                     [4]
    00104055 7f                undefined17Fh                     [5]
    00104056 fe                undefined1FEh                     [6]
    00104057 40                undefined140h                     [7]
    00104058 e2                undefined1E2h                     [8]
    00104059 4d                undefined14Dh                     [9]
    0010405a b9                undefined1B9h                     [10]
    0010405b 99                undefined199h                     [11]
    0010405c 9c                undefined19Ch                     [12]
    0010405d 18                undefined118h                     [13]
    0010405e 03                undefined103h                     [14]
```

Figure 5: final

Creating the Exploit

This exploit can be created by reversing the algorithm used to get to the "final" byte. By xoring the final byte with the corresponding buffers byte, you will get the byte that the user needed to input.

Esentially, we are using this methodology:

x ^ y = z

If x = buffers[(i % 3) * 0xf + i], y = user input, and z = final, then

y = x ^ z

So the user input is equal to buffers ^ final.

Python Script

After hand-jamming the math for each byte needed, I came up with this payload.

```python
#! /usr/bin/env python3

from pwn import *

target = process('./input2')


#went through ghidra and found the 2 checks.
#check2 contained the formula "buffers ^ userInput = final"
#which means, "final ^ buffers = userInput" which is the payload below
payload = b'\xab\xf0\x48\x3b\x3d\xdb\x9c\xf9\x38\xb9\x63\x3a\xa9\x51\xd0\x00'

#for check1, I ensured there was a null byte at the end of the payload

log.info(f'payload sent => {payload}')
target.sendline(payload)

target.interactive()
```

Flag

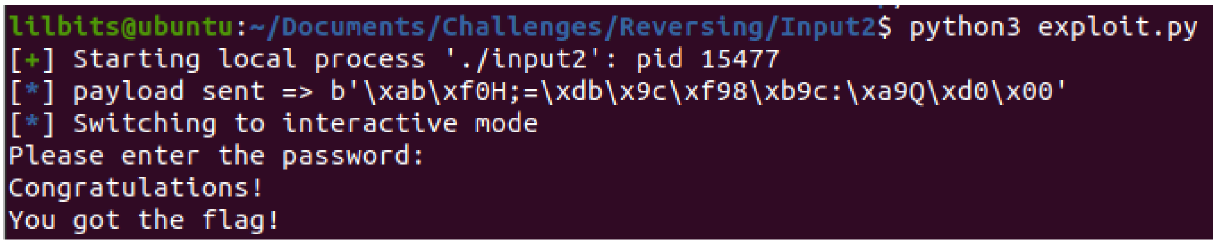

Figure 6: Flag

Conclusion

This challenge was only possible to reverse since the final bytes and the buffers bytes were constants. Knowing how to reverse the xor was key to solving this problem!

References

1. https://stackoverflow.com/questions/14279866/what-is-inverse-function-to-xor