Flare-On 3: Challenge 5 Solution - smokestack.exe

Challenge Author: Tyler Dean

Let's take a look at solving the Flare-On challenge binary #5 named `smokestack.exe`. Both running the file and looking at the strings of `smokestack.exe` produce very few clues for this challenge. After opening the file in a disassembler (we like IDA Pro), and looking at the `main` function, we see that the executable accepts at least one command line argument. Looking at Figure 1, the code makes sure there is more than one command line argument. The first command line argument after the program name is used as a parameter to a `strlen` call. Next, the result of the `strlen` is compared with 0xA or decimal 10. Now we know that the executable accepts at least one command line argument that is expected to be at least 10 characters in length.

```
.text:00402F76     cmp      [ebp+argc], 1     ; check if one command line argument
.text:00402F7A     jle      loc_403047        ; jump if argc <= 1
.text:00402F80     mov      eax, [ebp+argv]   ; get here if argc > 1
.text:00402F83     mov      ecx, [eax+4]      ; get the first command line argument
.text:00402F86     push     ecx
.text:00402F87     call     _strlen           ; strlen(argv[1])
.text:00402F8C     add      esp, 4
.text:00402F8F     mov      [ebp+var_4], eax ; set var_4 to result of strlen call
.text:00402F92     cmp      [ebp+var_4], 0Ah ; compare the length to 0xA (10)
.text:00402F96     jl       loc_403047
```
Figure 1: Argument count and string length comparisons

Each byte of the command line argument is then placed in a global array of two-byte shorts (WORDs) at virtual address (VA) `0x40DF20`.

After filling this global array, the executable calls function `sub_401610`. That function returns some value that is copied to the end of the command line argument string. Afterwards, a function that performs an MD5 hash in a loop is called. Next, an RC4 function is called and the result is printed to the console. The result is likely the challenge key.

The next step is to jump into `sub_401610` and see what this validation function does. The first thing we notice in `sub_401610` is more global variables being initialized. We also see that our return value is a global named by IDA Pro as `word_40DF18`. The challenge binary loops and continues to call function `sub_401540`. This loop breaks when the global at `word_40DF1E` reaches 0x182 (386).

Next, let's explore `sub_401540`. We see that the global `word_40DF1E` is used as an index into an array at global `word_40A140`. After fetching a value in the global array `word_40A140` (why so many globals?!), the executable uses that as an index into a function table. Looking at the cross references, it looks like the function table was initialized in the `__cfltcvt_init` function we skipped over before.

Exploring the functions in the function table, we see a lot of calls to `sub_401000` and `sub_401080`.

Looking into `sub_401000` a bit deeper, shown in Figure 2, we see the function retrieves the value stored at global `word_40DF1C` and increments the value by one. The function puts the value of `arg_0` at the new incremented offset of the global array `word_40DF20`. To summarize, the function takes the value of `arg_0` and inserts it at the end of a global array.

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 mov     ax, word_40DF1C         ; get value stored at global word_40DF1C
.text:00401009 add     ax, 1                   ; increment the value by 1
.text:0040100D mov     word_40DF1C, ax         ; set incremented value back to word_40DF1C
.text:00401013 movzx   ecx, word_40DF1C        ; get word_40DF1C again and store it in ecx
.text:0040101A mov     dx, [ebp+arg_0]         ; get the argument and store it in dx
.text:0040101E mov     word_40DF20[ecx*2], dx  ; put arg_0 at the incremented offset
.text:00401026 pop     ebp
.text:00401027 retn
```
Figure 2: Push function

Next, we take a quick look at `sub_401080`, another function that seems to be called a lot. This function seems to perform the opposite operation of `sub_401000`. It subtracts one, retrieves the last value from the same array and returns the retrieved value. This seems like a pop operation, and `sub_401000` sounds like a push operation.

After renaming these functions in IDA Pro to `push_op` and `pop_op`, we go back into each of the functions in the function table to see if they start to make any more sense.

`sub_4010E0`, shown in Figure 3, contains a call to `pop_op`, another call to `pop_op`, and a call to `push_op`. The results of those two `pop_op` calls are added together and the result of the addition is passed to the push_op function.

```
.text:004010E0  push    ebp
.text:004010E1  mov     ebp, esp
.text:004010E3  sub     esp, 0Ch
.text:004010E6  call    pop_op              ; call to pop_op
.text:004010EB  mov     [ebp+var_4], ax     ; store result of pop_op in var_4
.text:004010EF  call    pop_op              ; call to pop_op
.text:004010F4  mov     [ebp+var_8], ax     ; store result of pop_op in var_8
.text:004010F8  movzx   eax, [ebp+var_4]
.text:004010FC  movzx   ecx, [ebp+var_8]
.text:00401100  add     eax, ecx            ; add var_4 and var_8
.text:00401102  mov     [ebp+var_C], ax     ; store result in var_C
.text:00401106  movzx   edx, [ebp+var_C]
.text:0040110A  push    edx
.text:0040110B  call    push_op             ; call to push_op with the add result (var_C)
.text:00401110  add     esp, 4
.text:00401113  mov     ax, word_40DF1E
.text:00401119  add     ax, 1               ; increment word_40DF1E
.text:0040111D  mov     word_40DF1E, ax
.text:00401123  mov     esp, ebp
.text:00401125  pop     ebp
.text:00401126  retn
```
Figure 3: Add operation

sub_401130 contains two calls to pop_op, an x86 sub instruction, and a call to push_op. sub_401260 contains two calls to pop_op, an x86 xor instruction, and a call to push_op. Other functions are similar as well.

These functions are dispatched in a loop and the specific function called is dependent upon a global array of values. At this point, we should be suspicious of a stack-based virtual machine.

With this assumption, we suspect that word_40A140 is an array of opcodes, and word_40DF1E is the index into the array of opcodes. We rename word_40DF1E to g_program_counter and word_40A140 to g_code. Our stack is global word_40DF20 and is renamed to g_stack, and our stack pointer is global word_40DF1C, which is renamed to g_stack_pointer.

At this point, we know that the input is 10 characters long. We also see that the 10 input characters are being placed onto the virtual stack. We assume that the stack-based virtual machine validates those input characters, and produces some output. Armed with this information, we have a few options to solve this challenge. One option is to write our own disassembler for the virtual opcodes. After all, there are only 14 opcode types, so that option shouldn't be too hard. The second option is to brute force the input value and see if the virtual machine leaks any information that helps us find the key. This write-up focuses on reverse engineering the virtual machine.

Custom disassembler solution

The first step to understanding the virtual machine is to reverse engineer the opcode handler routines to understand each virtual instructions. Fortunately, there are only 14. Table 1 lays out our understanding of each opcode:

Opcode	Mnemonic	Arg used	Description
0	`push`	yes	Pushes the next value in the code onto the top of the stack
1	`pop`	no	Pops a value off the top of the stack
2	`add`	no	Pops the top two values off the stack, adds them, pushes the result onto the stack
3	`sub`	no	Pops the top two values off the stack, subtracts them, pushes the result onto the stack
4	`ror`	no	Pops the top two values off the stack, rotates the second value right by the first value, pushes the result onto the stack
5	`rol`	no	Pops the top two values off the stack, rotates the second value left by the first value, pushes the result onto the stack
6	`xor`	no	Pops the top two values off the stack, xors them, pushes the result onto the stack
7	`not`	no	Pops the top value off the stack, performs a bitwise not and pushes the result back onto the stack
8	`eq`	no	Pops the top two values off the stack, checks if they are

			equal, pushes 1 if true or 0 if false onto the stack
9	`if`	no	Pops the top three values off the stack, if the third value is 1 (true), the second value is pushed onto the stack, if the third is 0, the first value is pushed onto the stack
10	`br`	no	Pops the first value off the stack, sets the program counter to the new value
11	`store`	yes	Stores a register value to the stack
12	`load`	yes	Loads a value from the stack to a register
13	`nop`	no	No operation

Table 1: Opcode summary

Further, there are four registers in this virtual machine: two general-purpose registers, one register used for the program counter and one register used as the stack pointer. They are shown in Table 2.

register	name	description
0	`r1`	General purpose register number one
1	`r2`	General purpose register number two
2	`sp`	Stack pointer
3	`pc`	Program counter

Table 2: Register summary

Once each of these components is understood, we write a simple disassembler. An example disassembler written in Python is shown in Figure 4.

```
code = [0, 33, 2, 0, ...

mnem = {
    0: "push",
    1: "pop",
    2: "add",
    3: "sub",
    4: "ror",
    5: "rol",
    6: "xor",
    7: "not",
    8: "eq",
    9: "if",
    10: "br",
    11: "store",
    12: "load",
    13: "nop",
}
regs = {
    0: "r1",
    1: "r2",
    2: "sp",
    3: "pc",
}

i = 0

while i < len(code):
    opcode = code[i]
    line = "%04d: %s" % (i, mnem[opcode])
    if opcode == 0: # push
        i += 1
        val = code[i]
        line = "%s %d" % (line, val)
    elif opcode in (11, 12): # store / load
        i += 1
        reg_type = code[i]
        opnd = regs[reg_type]
        line = "%s %s" % (line, opnd)
    print line

    i += 1
```
Figure 4: Sample Python based disassembler

After running the disassembler on the virtual opcodes, the beginning of the output is shown in Figure 5.

```
0000: push 33
0002: add
0003: push 145
0005: eq
0006: push 22
0008: push 12
0010: if
0011: br
0012: store r1
0014: push 12
0016: add
0017: load r1
0019: push 29
0021: br
0022: store r1
0024: push 99
0026: add
0027: load r1
0029: push 24
0031: xor
0032: push 84
0034: eq
[removed]
```
Figure 5: Disassembly listing

Let’s start going through this disassembly. Remember, the command line argument is what initially makes up the stack. So, as an example, let’s use `ABCDEFGHIJ` as the command line argument. This makes the top of the stack is ‘`J`’ or 74 in decimal. Looking at the disassembly, we see the value 33 is pushed on to the stack. The top two values of the stack are then added: 74 + 33 = 107. Next, the value 145 is pushed onto the stack. The top two values are then compared. In our example case, 145 is compared with 107. These values are not equal. Figure 6 shows these first few operations from the perspective of the stack.

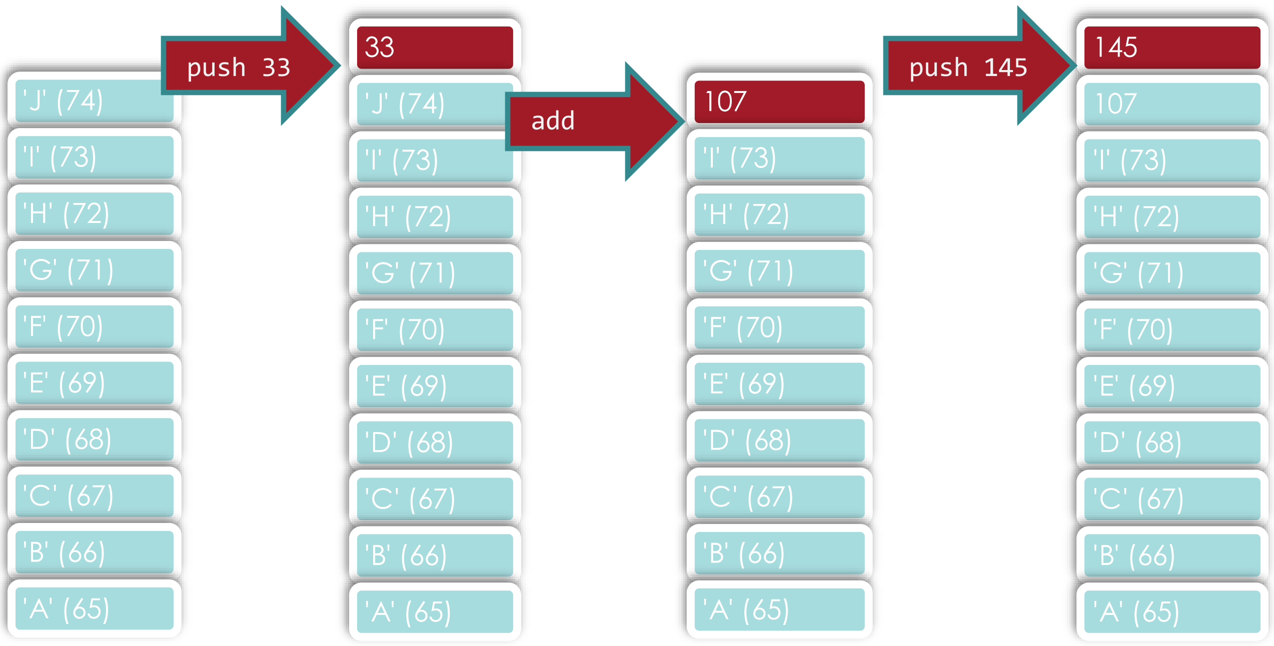

Figure 6: Solving first step

If the result of the addition with 33 was equal to 145, the virtual machine branches to the instruction at location 12. In our case, they are not equal, so the virtual machine branches to the instruction at location 22. We assume that the equal condition is the desired state. We derive the correct initial input by solving for `x` in `33 + x = 145`. In this case the input value is `112` in decimal or the character `p` in ASCII.

The next interesting operation occurs at line 29 in the disassembly. The value 24 is pushed onto the stack and XORed with the current top value (in our example case, this would be '`I`'). The result is then compared with 84. To cause this comparison to be equal, we need to find the value for `x` where `24 ^ x = 84`. In this case, `x` is equal to `76`, or the character '`L`' in ASCII. Now we have figured out the last two input characters, `L` and `p`. We continue to reverse engineer the results until we arrive at the correct input string: `kYwxCbJoLp`.

Using the correct input text, we see the key to unlock the next challenge in Figure 7.

```
C:\> smokestack.exe kYwxCbJoLp

A_p0p_pu$H_&_a_Jmp@flare-on.com
```

Figure 7: Final key

Further down the disassembly listing, some of the calculations become more complex. To aid in our understanding, we can recreate the logic behind each opcode operation in a scripting language such as Python. This allows us to add print statements in each opcode emulation code to better understand the results of different actions. Creating an emulator for this virtual machine is left as an exercise for the reader.