“FuN51llyM4ch1n3” Official Writeup

27th January 2026

Prepared By: {CYNX}`Team

Title: FuN51llyM4ch1n3

Description: Beep boop! The age of robots is here! But this one's kinda funky. Maybe it stores some kind of password? Try running it :D

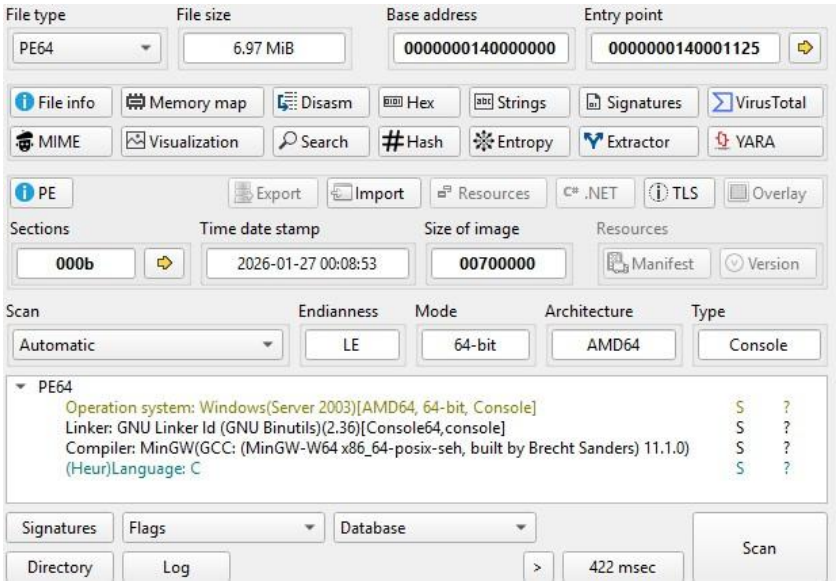
Flag: flag{FSM_1s_S0_c0000l}

Difficulty: Medium/Hard

Writeup classification: Official

Solve

Given a single executable named ` FuN51llyM4ch1n3.exe`. Running the binary executes *something* but not significant to the solve itself. We can check the binary details with DiE to see if we can find anything to note down:

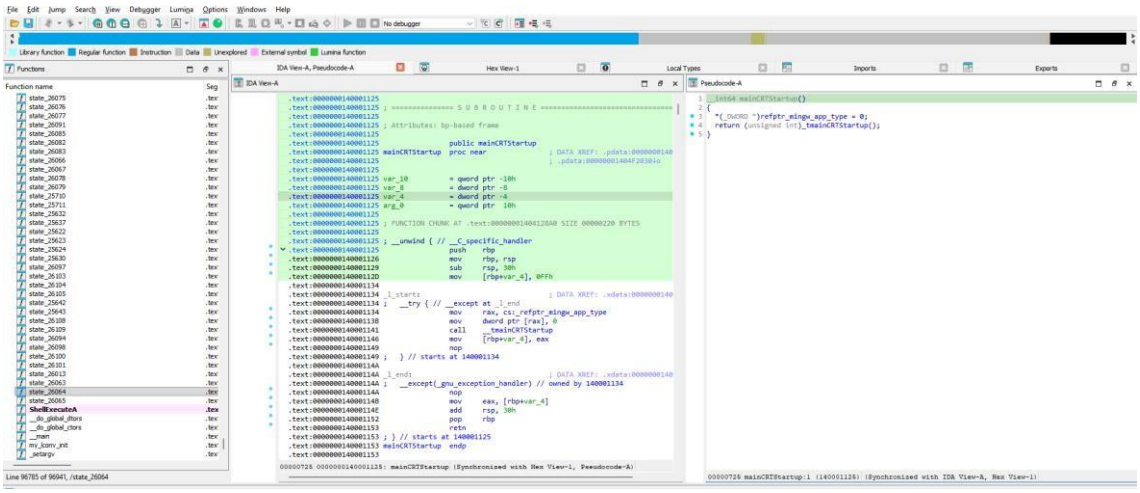


Seems like a simple binary but for some reason, it's 10MB in size. Quite large for a C program.

If we attempt to execute the binary in a terminal, we get the following:

Seems like it requires a 16 char password. Let's try entering an input with the length of 16 chars:

Let's open it up in IDA and see what we're dealing with:



In the decompilation, we see a huge function table. We also see a bunch of `state_x` functions and in the main functions, we can see other functions such as `g_position`, `g_state`, `g_transition`, and `g_input`. Based on this, and the challenge name itself, it seems like we're dealing with some sort of Finite State Machine.

Pseudocode-A

```
22    v8 = (unsigned __int8)g_input ^ 0x1337LL;
23    g_position = 0;
24    g_transitions = 0;
25    g_state = v8;
26    do
27    {
28      v9 = g_position + 1;
29      if ( v8 <= 0x179FE )
30      {
31        refptr_STATES[v8]();
32        v8 = g_state;
33        v10 = g_position;
34        if ( (unsigned __int64)g_state <= 0x179FE )
35        {
36          v11 = *((_BYTE *)refptr_g_flag_data + g_state);
37          if ( v11 )
38            flag_buf[g_position] = g_state ^ v11 ^ 0xAA;
39        }
40        v9 = v10 + 1;
41      }
42      g_position = v9;
43      --v7;
44    }
45    while ( v7 );
46    if ( g_transitions == 16 )
47    {
48      byte_1406CC0D0 = 0;
49      printf("\nBeep Boop. \nN1c3 J0b! Here: flag{%s}\n", flag_buf);
50    }
51    else
52    {
53      printf("\nBeep Boop. \nNo.\n");
54    }
55    return 0;
56  }
57  else
58  {
59    printf("Usage: .\\FuN51llyM4ch1n3 <16 characters password>\n");
60    return 1;
61  }
62 }
```

We can see in the pseudocode that the win condition is just to provide a 16-character string where each character causes a valid transition in the state machine. Each of those 16 transitions must successfully increment `g_transitions`.

We can also see that the starting state is calculated by XORing the first character of our input with 0x1337 and taking the modulo of the total state space (96,767).

$$g\_state = (0x1337 \oplus input[0]) (mod\ 96767)$$

In each iteration of the program, it executes a function pointer from a massive array called `STATES` based on the current `g_state`.

```
.data:000000014041A010 deregister_frame_fn dq 0                ; DATA XREF: __gcc_register_frame+5B↑w
.data:000000014041A010                                         ; __gcc_register_frame+8E↑w ...
.data:000000014041A018                 align 20h
.data:000000014041A020                 public STATES
.data:000000014041A020 STATES          dq offset state_0       ; DATA XREF: .rdata:_refptr_STATES↓o
.data:000000014041A028                 dq offset state_1
.data:000000014041A030                 dq offset state_2
.data:000000014041A038                 dq offset state_3
.data:000000014041A040                 dq offset state_4
.data:000000014041A048                 dq offset state_5
.data:000000014041A050                 dq offset state_6
.data:000000014041A058                 dq offset state_7
.data:000000014041A060                 dq offset state_8
.data:000000014041A068                 dq offset state_9
.data:000000014041A070                 dq offset state_10
.data:000000014041A078                 dq offset state_11
.data:000000014041A080                 dq offset state_12
.data:000000014041A088                 dq offset state_13
.data:000000014041A090                 dq offset state_14
.data:000000014041A098                 dq offset state_15
.data:000000014041A0A0                 dq offset state_16
.data:000000014041A0A8                 dq offset state_17
.data:000000014041A0B0                 dq offset state_18
.data:000000014041A0B8                 dq offset state_19
.data:000000014041A0C0                 dq offset state_20
.data:000000014041A0C8                 dq offset state_21
.data:000000014041A0D0                 dq offset state_22
.data:000000014041A0D8                 dq offset state_23
.data:000000014041A0E0                 dq offset state_24
.data:000000014041A0E8                 dq offset state_25
.data:000000014041A0F0                 dq offset state_26
.data:000000014041A0F8                 dq offset state_27
.data:000000014041A100                 dq offset state_28
.data:000000014041A108                 dq offset state_29
.data:000000014041A110                 dq offset state_30
.data:000000014041A118                 dq offset state_31
.data:000000014041A120                 dq offset state_32
.data:000000014041A128                 dq offset state_33
.data:000000014041A130                 dq offset state_34
.data:000000014041A138                 dq offset state_35
.data:000000014041A140                 dq offset state_36
```

State Machine Mechanics

The core of this challenge lies in the `STATES` array. Each entry is a pointer to a unique function representing a specific state.

When we analyze one of the states, we see the following:

```
1  __int64 state_16()
2  {
3    char current_input; // al
4
5    current_input = get_current_input();
6    if ( current_input == 107 )
7    {
8      increment_transitions();
9      return set_next_state(9878);
10   }
11   else if ( current_input == 74 )
12   {
13     increment_transitions();
14     return set_next_state(28202);
15   }
16   else
17   {
18     return show_error_message();
19   }
20 }
```

We see that in this state, it fetches the character at the current input position using `get_current_input()`, then compares that character against one or more hardcoded values. If it's a match, then it calls `increment_transitions()` and updates `g_state` to a new ID via `set_next_state()`. If it fails: It calls `show_error_message()`, which acts as a dead end.

To break it down in lament terms:

In `state_16`, if the current character is 'k', we move to state 9878. But if it is 'J', we move to 28202.

Obviously, we're not going to explore the 90,000+ states manually, instead we'll utilize the Breath-First-Search to walk the graph for us. We can do this by crafting a script to achieve this.

Locating the State Machine

The core of the binary is a massive array of function pointers named STATES located at 0x14041a020. Each pointer in this array leads to a unique function representing a specific "state" in the machine.

- Extraction: Read 96,767 quadwords from this address to map every state ID to its corresponding memory address.
- Initialization: The program determines the starting state by XORing the first character of the input with 0x1337 and taking the modulo of the total number of states.

```
import idc
import idaapi
import idautils
from collections import deque

STATES_ARRAY_ADDR = 0x14041a020
NUM_STATES = 96767

def calculate_entry_state(first_char):
    return (0x1337 ^ ord(first_char)) % NUM_STATES

def extract_addresses():
    print(f"[*] Reading {NUM_STATES} pointers from 0x{STATES_ARRAY_ADDR:x}")
        addrs
= []
    ea = STATES_ARRAY_ADDR
        for i in
range(NUM_STATES):        if i
% 10000 == 0:
            print(f"[*] {i}/{NUM_STATES}...")

        ptr =
idc.get_qword(ea)
addrs.append(ptr)        ea +=
8

    return addrs
```

Dissecting the state transitions

Every state function follows a rigid pattern to decide which state comes next based on out input. Instead of reading these manually, we use the script to parse the assembly instructions:

- Input Comparison: The script scans for cmp instructions where the input character (stored in al or cl) is checked against a hardcoded value.
- Success Branching: It follows the jz (jump if zero) or je (jump if equal) instructions to find the logic that executes when a character match is found.
- Destination Extraction: It looks for a mov instruction (typically into registers like ecx or rdi) that sets the next state ID, allowing us to draw an "edge" between states.

```
def parse_small_state(func_ea, state_id):     if not
idc.get_func_attr(func_ea, idc.FUNCATTR_START):
        idc.create_function(func_ea)
        idc.set_name(func_ea, f"state_{state_id}", idc.SN_NOWARN)

    transitions  =  {}         func_start  =  idc.get_func_attr(func_ea,
idc.FUNCATTR_START)            func_end   =   idc.get_func_attr(func_ea,
idc.FUNCATTR_END)
        if not func_end or func_end == idc.BADADDR:
        func_end = func_start + 0x200

    ea = func_start
max_ins = 100     count = 0
        current_char = None
        while ea < func_end and count < max_ins:
        mnem = idc.print_insn_mnem(ea)
                if mnem ==
"cmp":
            op1 = idc.print_operand(ea, 0)
                        if op1 in ["al",
"cl"]:
                char_val = idc.get_operand_value(ea, 1)
if 0x20 <= char_val <= 0x7E:
                    current_char = chr(char_val)
```

```
        elif mnem in ["jz", "je"] and current_char:
target = idc.get_operand_value(ea, 0)


            scan_ea = target
for _ in range(10):                 if
scan_ea >= func_end:
                    break
                                scan_mnem =
idc.print_insn_mnem(scan_ea)
                                if
scan_mnem == "mov":
                    scan_op1 = idc.print_operand(scan_ea, 0)
                                        if scan_op1 in ["ecx",
"rcx", "edi", "rdi"]:
                        next_state = idc.get_operand_value(scan_ea, 1)
                                                if 0 <=
next_state < NUM_STATES:
                            transitions[current_char] =
next_state                              current_char = None
break
                                scan_ea =
idc.next_head(scan_ea)
                elif mnem
== "call":
            target_name = idc.get_func_name(idc.get_operand_value(ea, 0))
if target_name == "get_current_input":
                current_char = None


        ea    =    idc.next_head(ea)
count += 1


    return transitions
```

Breath-First-Search Solve

Because the state space is too large for a simple brute-force approach, we use a BreadthFirst Search (BFS) to "walk" the graph efficiently.

Since the first character of the password determines the starting state and acts as the first transition, we iterate through all printable characters (A-Z, 0-9):

- Calculate the `entry_state` for a character.
- Parse that `entry_state` function to see if it accepts that same character.
- If it does, we have a valid starting point for our search.

```
def build_graph(addrs):
    print("[*] Building graph...")
        graph = {}
visited = set()

    import string
queue = deque()
        for first_char in string.ascii_letters +
string.digits:
        entry     =     calculate_entry_state(first_char)
if entry not in visited and entry < len(addrs):
            queue.append(entry)
        print(f"[*] Starting BFS from {len(queue)} potential entry
states...")
        while
queue:
        sid = queue.popleft()
                if sid in visited or sid >=
len(addrs):
            continue

visited.add(sid)
                if len(visited) %
100 == 0:
            print(f"[*] States: {len(visited)}", end='\r')

        trans = parse_small_state(addrs[sid], sid)
graph[sid] = trans
```

```
        if len(visited) <= 10:
            print(f"\n[DEBUG] State {sid} at 0x{addrs[sid]:x}:
{len(trans)} transitions")            for c, n in trans.items():
print(f"  '{c}' -> {n}")
                for nxt in
trans.values():            if nxt
not in visited:
queue.append(nxt)

    print(f"\n[+] Graph: {len(graph)} states, {sum(len(t) for t in
graph.values())} edges")    return graph
```

Graph Traversal and Pruning

From a valid starting point, the BFS explores every possible valid transition layer-by-layer.

- Path Length: We only care about paths that reach exactly 16 characters.
- Visited States: To prevent the script from getting stuck in infinite loops, we track (state_id, path_length). If we reach the same state at the same character position again, we discard that branch.

```
def solve(graph):
    print("[*] Solving for 16-character password...")

    import           string
solutions = []
         for fc in string.ascii_letters +
string.digits:
        entry = calculate_entry_state(fc)
                if entry not
in graph:
            continue
                if fc not in
graph[entry]:
            continue
                print(f"[*] Trying '{fc}' (entry={entry})...",
end='\r')

        first_next = graph[entry][fc]
q = deque([(first_next, fc)])
vis = set()
                while q and
len(solutions) < 10:
            s, p = q.popleft()
                        if (s,
len(p)) in vis:
continue            vis.add((s,
len(p)))
                        if
len(p) == 16:
                print(f"\n[+] SOLUTION: {p}")
solutions.append(p)                 break
```

```
                        if s in graph:
for c, n in graph[s].items():
if len(p) < 16:
                        q.append((n, p + c))
return solutions
```

Running this script in IDA > File > Script file, we're able to find the exact password that the program uses:

```
[DEBUG] State 4957 at 0x14037d4b0: 0 transitions
[*] States: 100
[+] Graph: 146 states, 84 edges
[*] Solving for 16-character password...
[*] Trying 'n' (entry=4953)...
[+] SOLUTION: n0e35cMn8sljP2cS

============================================================
[+] Found 1 solution(s):
    n0e35cMn8sljP2cS
```

Inputting this password into the program, we get the flag:

Flag = flag{FSM_1s_S0_c0000l}