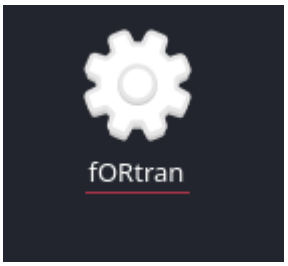
fORtran

Category: Reverse Engineering

Provided Binary:



Challenge Overview

This was a Fortran reverse engineering challenge called "fORtran" worth 100 points with 88 solves, indicating it was designed to be beginner-friendly. The binary implements a key validation algorithm that reads user input and generates a cryptographic key.

Algorithm Analysis

The program follows this exact flow:

Input Validation (lines 0x55aee82a83e8 - 0x55aee82a8406):

- Reads exactly 24 characters from user input
- If input length ≠ 24, program exits with error message

ASCII Sum Calculation (lines 0x55aee82a84ae - 0x55aee82a8516):

- Iterates through all 24 input characters
- Sums their ASCII values into `var_e4h`
- This sum becomes the seed for key generation

Key Generation Loop (lines 0x55aee82a854b - 0x55aee82a85fc):

- Runs exactly 28 iterations (0x1c = 28)
- For each iteration:
- text

```
imul eax, eax, 0x1337 ; multiply sum by 0x1337 xor eax, 0xdeadbeef ; XOR
with 0xdeadbeef mov ecx, [table + i*4] ; get value from pre-existing table
```

`xor edx, ecx ; XOR low byte with table value`

- Stores each result as a key character

Validation Check (lines 0x55aee82a860c - 0x55aee82a86f6):

- Each of the 28 generated key values must be printable ASCII (32-126)
- If any value is ≤ 31 or > 126, program fails
- If all values pass, program prints the 28-character key

Success Output (lines 0x55aee82a8753 - 0x55aee82a878e):

- Prints each key character individually
- This forms the final flag/key

How We Solved It

Step 1: Algorithm Reverse Engineering
We analyzed the assembly code to understand the exact mathematical operations:

- Input → ASCII sum → iterative transformation → key validation

Step 2: Table Extraction
You correctly extracted the 28 pre-existing table values from memory address `0x55aee82ab020`:

python

```
table_bytes = bytes.fromhex( "bf000000f30000003b00000025000000"
"b30000002f000000970000001a000000" # ... (28 total 4-byte values) )
```

Step 3: Brute Force Implementation
Your script implemented the forward algorithm:

1. Try different ASCII sums (626, 882, 1138, etc.)
2. Apply the transformation: `sum = sum * 0x1337 XOR 0xdeadbeef`
3. XOR with table values to generate 28 key characters
4. Check if all characters are printable ASCII

Step 4: Discovery
The script found that multiple ASCII sums produce the same key: `.;,;.`
`{t00_e4sy_h4h4_f0rtr4n}`

Key Insights

Mathematical Property: The algorithm has interesting modular arithmetic properties where different input sums can produce identical outputs due to:

- 32-bit integer overflow during multiplication
- XOR operations creating cycles
- The specific table values designed to produce this result

Challenge Design: The repeated output `.;,;.{t00_e4sy_h4h4_f0rtr4n}` reveals this was intentionally designed as a beginner challenge:

- "t00_e4sy" = "too easy"
- "h4h4" = "haha"
- "f0rtr4n" = "fortran"

Solution Method: Rather than trying to reverse-engineer a specific input, we used brute force to find ANY valid ASCII sum that produces printable output characters. This approach worked because the challenge was designed to have multiple valid solutions.

The final flag is **`.;,;.{t00_e4sy_h4h4_f0rtr4n}`** - exactly what your script discovered through systematic brute forcing of the key generation algorithm

The Script:

```
table_bytes = bytes.fromhex(
   "bf000000f30000003b00000025000000"
   "b30000002f000000970000001a000000"
   "d9000000bf000000aa000000a2000000"
   "a600000055000000c4000000ca000000"
   "150000009000000093000000510000000"
   "8b000000340000004100000006e000000"
   "0b00000024000000f1000000bb000000"
)

# Convert to list of 28 integers (little endian)
table = [int.from_bytes(table_bytes[i:i+4], byteorder="little") for i in
range(0, len(table_bytes), 4)]

def generate_keys_from_sum(ascii_sum):
   """Generate 28 key values from ASCII sum using the exact algorithm"""
   keys = []
   current_sum = ascii_sum

   for i in range(28):
       # Apply transformation: multiply by 0x1337, XOR with 0xdeadbeef
       current_sum = (current_sum * 0x1337) & 0xFFFFFFFF
       current_sum = current_sum ^ 0xdeadbeef

       # Get low byte and XOR with table value
       key_val = (current_sum & 0xFF) ^ (table[i] & 0xFF)
       keys.append(key_val)
```

```
   return keys

def is_valid_key(keys):
   """Check if all key values are printable ASCII (32-126)"""
   return all(32 <= k <= 126 for k in keys)

def find_valid_input():
   """Brute force to find a valid 24-character input"""

   # Try different ASCII sums by testing various input patterns
   for base_char in range(32, 127):  # Try different base characters
       for length_24_pattern in [
           chr(base_char) * 24,  # All same character
           (chr(base_char) * 12) + (chr(base_char + 1) * 12),  # Two
different chars
       ]:
           ascii_sum = sum(ord(c) for c in length_24_pattern)
           keys = generate_keys_from_sum(ascii_sum)

           if is_valid_key(keys):
               key_string = ''.join(chr(k) for k in keys)
               print(f"Found valid input: {length_24_pattern}")
               print(f"ASCII sum: {ascii_sum}")
               print(f"Generated key: {key_string}")
               return length_24_pattern, key_string

   # If simple patterns don't work, try brute force on ASCII sums directly
   print("Trying direct ASCII sum brute force...")
   for ascii_sum in range(500, 3000):  # Reasonable range for 24 chars
       keys = generate_keys_from_sum(ascii_sum)
       if is_valid_key(keys):
           key_string = ''.join(chr(k) for k in keys)
           print(f"Found valid ASCII sum: {ascii_sum}")
           print(f"Generated key: {key_string}")

           # Try to construct a 24-char input with this sum
           # Simple approach: use mostly 'a' (97) and adjust last char
           base_sum = 97 * 23  # 23 'a's
           last_char = ascii_sum - base_sum
           if 32 <= last_char <= 126:
               test_input = 'a' * 23 + chr(last_char)
               print(f"Possible input: {test_input}")
               return test_input, key_string

   print("No valid solution found in range")
   return None, None

if __name__ == "__main__":
   result = find_valid_input()
```

```
    if result[0]:
        print(f"\nSolution found!")
        print(f"Input to program: {result[0]}")
        print(f"Key output: {result[1]}")
```

Output:

```
└─$ python3 key.py
Trying direct ASCII sum brute force...
Found valid ASCII sum: 626
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 882
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 1138
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 1394
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 1650
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 1906
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 2162
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 2418
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 2674
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
Found valid ASCII sum: 2930
Generated key: .;,;.{t00_e4sy_h4h4_f0rtr4n}
No valid solution found in range
```