ASSEMBLY BOMB LAB

Ethan Lim

TABLE OF CONTENTS

Introduction

This report on Tian Lab's Starter Task Assembly Bomb Lab contains the following:

- The reverse-engineered logic
- Assembly snippets with annotations.
- Control flow diagrams (e.g., loops, conditionals) describe each phase's flow.
- Key observations (e.g., how you identified the input validation logic).
- Methodology of how I approached each problem.

This report was written by Ethan Lim of UCLA.

System Information

- Virtual Machine: Oracle VirtualBox
- Operating System: Ubuntu (Linux-based, user-friendly)
- Configuration:
    - Base memory: 6949 MB
    - Processors: 4
    - Boot Order: Hard Disk, Optical, Floppy
    - EFI: Enabled
    - Acceleration: Nested Paging, KVM Paravirtualization
- Reverse Engineering Tool: Ghidra 11.4.2

General Methodology

The primary approach to defusing each phase involved:

- Identifying the code blocks responsible for returning true (indicating successful defusal)
- Examining constraints within these blocks
- Deriving input values that satisfy these constraints

Phase 1: String Comparison Against Hidden String

Decompiled Code

```
bool FUN_00101378(char *param_1)
{
  int iVar1;

  iVar1 = strcmp(param_1,"Were located at 63-129 in Engineering IV");
  return iVar1 == 0;
}
```

Explanation

For the function to return True, iVar1 must equal 0, which occurs only when the input string exactly matches the hardcoded string. Thus, the input must be:

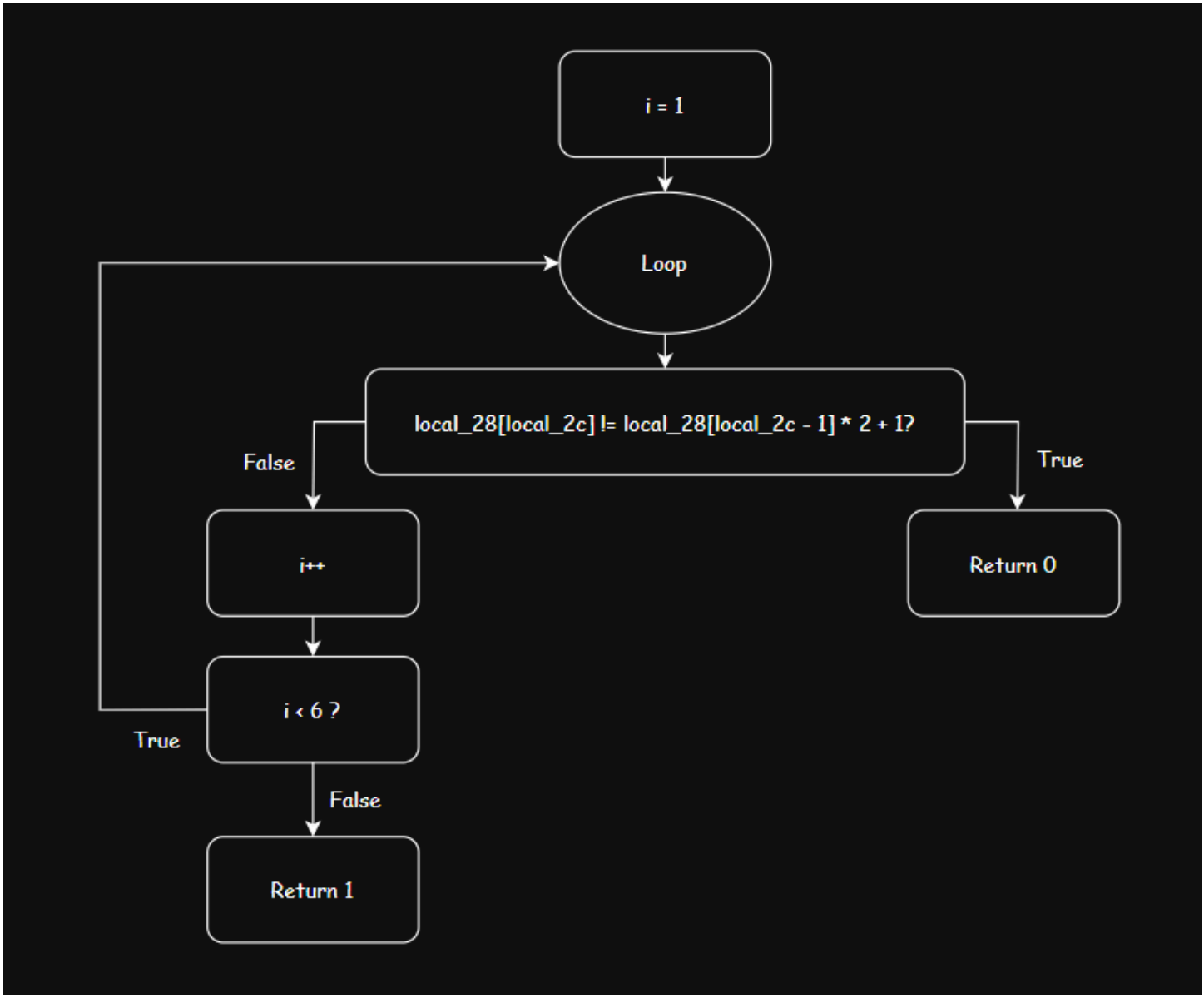
```
Were located at 63-129 in Engineering IV
```

Phase 2: Arithmetic Sequence Validation

Decompiled Code

```
undefined8 FUN_001013b0(undefined8 param_1)
{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  int local_2c;
  int local_28[2];
  undefined1 local_20[4];
  undefined1 local_1c[4];
  undefined1 auStack_18[4];
  undefined1 auStack_14[4];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = __isoc99_sscanf(param_1, "%d %d %d %d %d %d", local_28, local_28 + 1,
local_20, local_1c, auStack_18, auStack_14);
  if (iVar1 == 6) {
    if (local_28[0] < 1) {
      uVar2 = 0;
    }
    else {
      for (local_2c = 1; local_2c < 6; local_2c = local_2c + 1) {
        if (local_28[local_2c] != local_28[local_2c - 1] * 2 + 1) {
          uVar2 = 0;
          goto LAB_00101476;
        }
      }
      uVar2 = 1;
    }
  }
  else {
    uVar2 = 0;
  }
LAB_00101476:
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return uVar2;
}
```

Flow Diagram



Explanation

The function reads six integers from the input, separated by spaces. These integers are assigned to the variables local_28, local_28 + 1, local_20, local_1c, auStack_18, and auStack_14. __isoc_sscanf returns the number of input items that were successfully matched and assigned, and hence the first condition will be met.

```
if (iVar1 == 6) {...}
```

The first integer (local_28[0]) must be at least 1, as verified by the check:

```
if (local_28[0] < 1) {
  uVar2 = 0;
}
```

Although local_28 has been defined to only have two elements, the for-loop loops through the array 5 times, starting from the second element. This, along with the consecutive declarations of local_28, local_28 + 1, local_20, local_1c, auStack_18, and auStack_14, suggest that these variables collectively form a six-element array due to how the compiler laid out the stack.

The loop then iterates from index 1 to 5, ensuring the arithmetic relation:

```
local_28[i] = local_28[i−1]×2+1local\_28[i]
            = local\_28[i - 1] \times 2 + 1local_28[i]
            = local_28[i−1]×2+1
```

Holds true. Or, more commonly:

$$x = x_{n-1} * 2 + 1$$

If any element violates this, the function returns false. Therefore, the valid input consists of six integers that follow this recursive formula. A valid example input is:

```
1 3 7 15 31 63
```

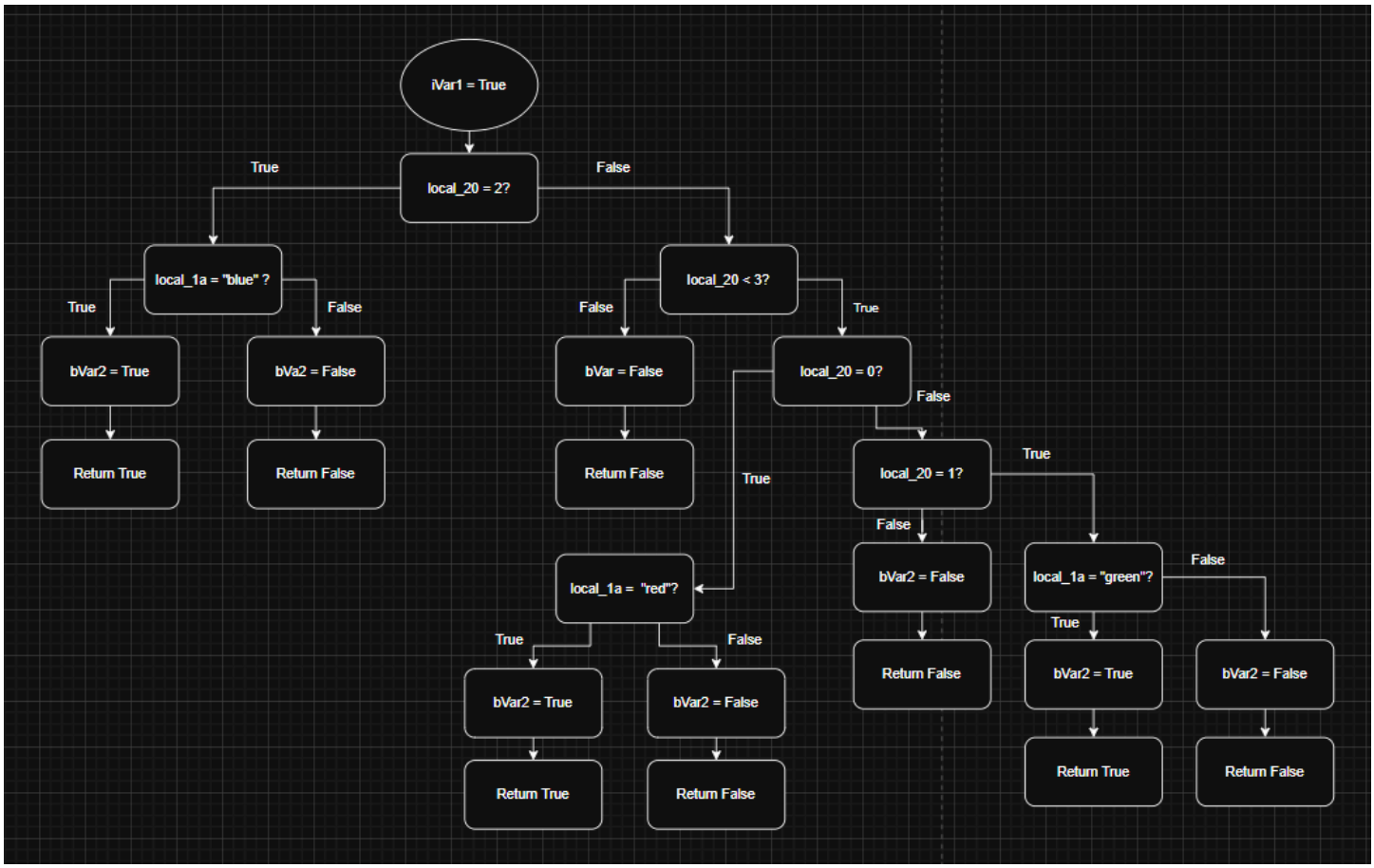
Phase 3: Switch-Case Logic with String Matching

Decompiled Code

```
bool FUN_0010148c(undefined8 param_1)
{
  int iVar1;
  long in_FS_OFFSET;
  bool bVar2;
  int local_20;
  char local_1a[10];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = __isoc99_sscanf(param_1, "%d %9s", &local_20, local_1a);
  if (iVar1 == 2) {
    if (local_20 == 2) {
      iVar1 = strcmp(local_1a, "blue");
      bVar2 = iVar1 == 0;
    }
    else {
      if (local_20 < 3) {
        if (local_20 == 0) {
          iVar1 = strcmp(local_1a, "red");
          bVar2 = iVar1 == 0;
          goto LAB_00101554;
        }
        if (local_20 == 1) {
          iVar1 = strcmp(local_1a, "green");
          bVar2 = iVar1 == 0;
          goto LAB_00101554;
        }
      }
      bVar2 = false;
    }
  }
  else {
    bVar2 = false;
  }
LAB_00101554:
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
```

```
    return bVar2;
  }
  __stack_chk_fail();
}
```

Flow Diagram



Explanation

This function expects the input to contain a digit followed by a string. It uses __isoc99_sscanf to parse these two components into local_20 (integer) and local_1a (string).
The function returns true by way of this only if the input matches one of the following conditions:

- If the digit is 2, the string must be "blue".
- If the digit is 0, the string must be "red".
- If the digit is 1, the string must be "green".

Any other input causes the function to return false, resulting in the bomb exploding. Therefore, valid inputs for this phase are:

```
2 blue
0 red
1 green
```

Phase 4: Recursive Function Analysis

Decompiler Code:

```
bool FUN_001015b6(undefined8 param_1)
{
  int iVar1;
  long in_FS_OFFSET;
  bool bVar2;
  int local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = __isoc99_sscanf(param_1, &DAT_001020e1, &local_14);
  if (iVar1 == 1) {
    if ((local_14 < 0) || (0x14 < local_14)) {
      bVar2 = false;
    }
    else {
      iVar1 = FUN_0010156a(local_14);
      bVar2 = iVar1 == 0x37;
    }
  }
  else {
    bVar2 = false;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return bVar2;
}
```

Explanation:

Phase 4 accepts a single decimal integer input, indicated by the __isoc99_sscanf call with format specifier, DAT_001020e1 which specifies "%d".

This integer, stored in local_14, must be within the range 0 to 20 (0x14) inclusive; otherwise, the function returns false.

If the input passes this check, it is passed to the recursive function FUN_0010156a:

```
int FUN_0010156a(int param_1)
{
  int iVar1;

  if (param_1 < 0) {
    param_1 = -1;
  }
  else if (1 < param_1) {
    iVar1 = FUN_0010156a(param_1 - 1);
    param_1 = FUN_0010156a(param_1 - 2);
    param_1 = param_1 + iVar1;
  }
  return param_1;
}
```

This function computes the nth Fibonacci number recursively. The phase expects the Fibonacci number corresponding to the input index to equal 0x37 (decimal 55). This occurs when the input is 10. In reverse-engineering logic,

local_14 must be 10. local_14 is assigned by param_1. param_1 must be equal to 10.

Thus, the valid input to defuse Phase 4 is:

```
10
```

Activation of the Secret Phase

The secret phase triggers when the substring "SMARTT" appears at &DAT_001040a0, detected by:

```
pcVar1 = strstr(&DAT_001040a0, "SMARTT");
if (pcVar1 != (char *)0x0) {
  puts("\n!!! Secret phase triggered !!!");
  FUN_00101b87(FUN_0010191f, "Secret Phase");
}
```

And is otherwise skipped if pcVar1 is null ((char *)0x0).

Recall that the format specifier in Phase 4 was also an address of a pointer (&DAT_001020e1O), suggesting a pointer's significance in activating the secret phase. Exploring the binary, let's look into FUN_00101b87, the function in main that takes in each phase as an argument.

```
void FUN_00101b87(code *param_1,undefined8 param_2)

{
  int iVar1;
  undefined8 uVar2;
  uint local_14;

  local_14 = 5;
  printf("\n--- %s ---\n",param_2);
  while( true ) {
    if (local_14 == 0) {
      FUN_00101289();
      return;
    }
    printf("%s (%d tries left): ",param_2,(ulong)local_14);
    uVar2 = FUN_00101301();
    iVar1 = (*param_1)(uVar2);
    if (iVar1 != 0) break;
    puts("Incorrect input. Try again.");
    local_14 = local_14 - 1;
  }
  printf(">>> %s defused! <<<\n",param_2);
  return;
}
```

Specifically, let's explore FUN_00101301, whose result is being assigned to uVar2.

```
undefined * FUN_00101301(void)

{
  char *pcVar1;

  pcVar1 = fgets(&DAT_001040a0,100,stdin);
  if (pcVar1 == (char *)0x0) {
    puts("Error reading input or EOF reached.");
    FUN_00101289();
  }
  pcVar1 = strchr(&DAT_001040a0,10);
  if (pcVar1 != (char *)0x0) {
    *pcVar1 = '\0';
  }
  return &DAT_001040a0;
}
```

Note how this function saves each phase's input in &DAT_001040a0. Since __isoc99_sscanf in Phase 4 only scans the initial decimal integer and ignores trailing characters, appending "SMARTT" directly after 10 both satisfies Phase 4 and stores "SMARTT" in &DAT_001040a0, allowing us to activate the secret phase.

Therefore, entering:

```
10SMARTT
```

As Phase 4's input will activate the secret phase.

Secret Phase

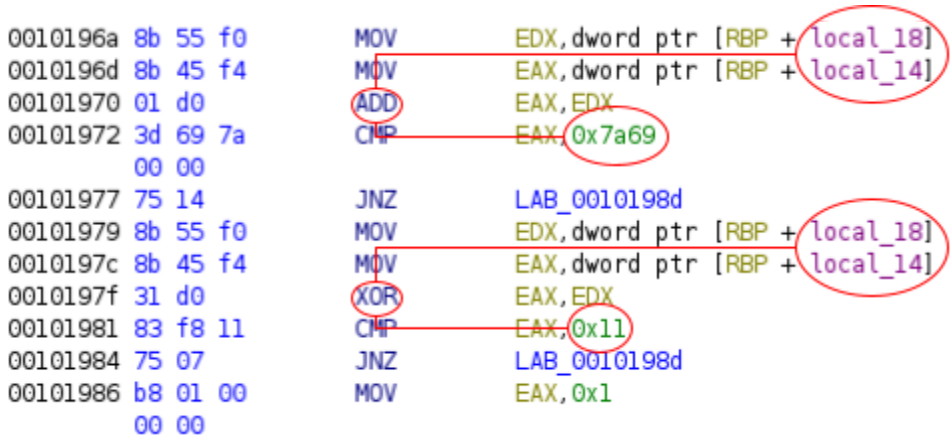
Decompiler Code:

```
undefined8 FUN_0010191f(undefined8 param_1)
{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  uint local_18;
  uint local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = __isoc99_sscanf(param_1, "%d %d", &local_18, &local_14);
  if (iVar1 == 2) {
    if ((local_14 + local_18 == 0x7a69) && ((local_14 ^ local_18) == 0x11)) {
      uVar2 = 1;
    }
    else {
      uVar2 = 0;
    }
  }
  else {
    uVar2 = 0;
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return uVar2;
}
```

Explanation:

The secret phase expects two decimal integers separated by a space, confirmed by the format specifier “%d %d” in the __isoc99_sscanf call.
The two integers, stored in local_18 and local_14, respectively, must satisfy these constraints simultaneously:

- Sum equals 0x7a69 (hexadecimal), or 31337 decimal.
- Bitwise XOR equals 0x11 (hexadecimal), or 17 decimal.
- Assembly Annotation:



Formally:

```
local_18 + local_14 == 31337
(local_18 ^ local_14) == 17
```

Solving this analytically is challenging and gruesome, however a simple brute-force Python script with time complexity $O(n^2)$ can be used for efficiency.

```python
# 7A69 (hex) = 31337 (decimal)

should_break = False

for local_14 in range(0, 31337):
    for local_18 in range(0, 31337):
        if (local_14 + local_18 == 0x7a69) and ((local_14 ^ local_18) == 0x11):
          print(f"local_14: {local_14}, local_18: {local_18}")
          should_break = True
          break
    if should_break:
        break
```

Output:

```
local_14 = 15660, local_18 = 15677
```

Thus, the first valid solution is:

```
15660 15677
```

Inputting these two integers separated by a space resolves the secret phase.

Phase 5: Bitwise/Array Transformations [Impossible]

Decompiled Code:

```
bool FUN_0010163f(char *param_1)
{
  int iVar1;
  size_t sVar2;
  long in_FS_OFFSET;
  bool bVar3;
  int local_24;
  char local_20[15];
  undefined1 local_11;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  builtin_strncpy(local_20, "arishem", 8);
  sVar2 = strlen(param_1);
  if (sVar2 == 7) {
    for (local_24 = 0; local_24 < 7; local_24 = local_24 + 1) {
      if ((param_1[local_24] < 'a') || ('z' < param_1[local_24])) {
        bVar3 = false;
        goto LAB_00101735;
      }
      local_20[(long)local_24 + 8] = (char)(param_1[local_24] ^ 0x42U) % '\x1a' + 'a';
    }
    local_11 = 0;
    iVar1 = strcmp(local_20 + 8, local_20);
    bVar3 = iVar1 == 0;
  } else {
    bVar3 = false;
  }
LAB_00101735:
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return bVar3;
}
```

Analysis:

The goal of this function is to compare a transformed version of the input string with the hardcoded string "arishem". The transformation is done using a bitwise XOR followed by modulo and offset operations. If the transformed result matches "arishem", the function returns true.

Input Constraints:

- The input must be exactly 7 lowercase characters. This is enforced by the following condition inside the loop:

```
if ((param_1[i] < 'a') || ('z' < param_1[i])) {
    bVar3 = false;
    goto LAB_00101735;
}
```

- Each character is transformed using:

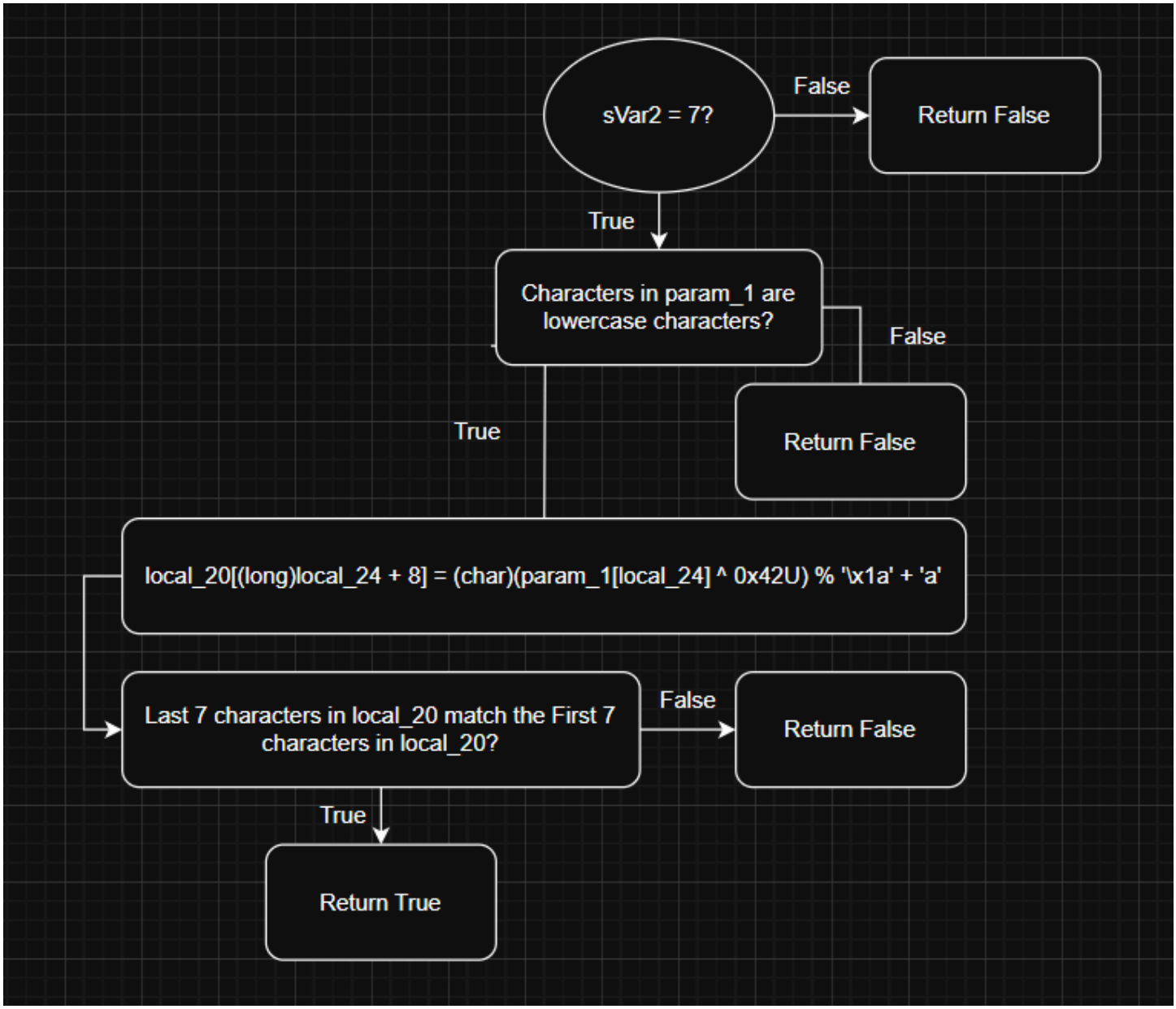
```
local_20[i + 8] = (char)(param_1[i] ^ 0x42U) % 0x1a + 'a';
```

This transforms the character using:

```
output_char = ((ASCII(input_char) XOR 66) % 26) + 97
```

The comparison is then made between the transformed 7 characters and the static string "arishem" stored in the first half of the local_20 array.

Flow Diagram



Reverse Engineering Attempt:

To reverse this logic, the goal was to find an input string such that each transformed character matched the corresponding character in "arishem". We first rearrange the transformation equation to

```
output_char - 97 = ((ASCII(input_char) XOR 66) % 26)
```

The following code in C++ was used for calculations:

```
#include <iostream>

using namespace std;

int main()
{
    int result;

    for (int i = 97; i < 123; i++) {
        result = (i ^ 66) % 26;
        cout << "When param_1[] is " << i << " (character '" << (char)i << "\"), " <<
"(i ^ 66) % 26 = " << result << "\n";
    }
}
```

Output:

```
When param_1 is 97 (character 'a'), (i ^ 66) % 26 = 9
When param_1 is 98 (character 'b'), (i ^ 66) % 26 = 6
When param_1 is 99 (character 'c'), (i ^ 66) % 26 = 7
When param_1 is 100 (character 'd'), (i ^ 66) % 26 = 12
When param_1 is 101 (character 'e'), (i ^ 66) % 26 = 13
When param_1 is 102 (character 'f'), (i ^ 66) % 26 = 10
When param_1 is 103 (character 'g'), (i ^ 66) % 26 = 11
When param_1 is 104 (character 'h'), (i ^ 66) % 26 = 16
When param_1 is 105 (character 'i'), (i ^ 66) % 26 = 17
When param_1 is 106 (character 'j'), (i ^ 66) % 26 = 14
When param_1 is 107 (character 'k'), (i ^ 66) % 26 = 15
When param_1 is 108 (character 'l'), (i ^ 66) % 26 = 20
When param_1 is 109 (character 'm'), (i ^ 66) % 26 = 21
When param_1 is 110 (character 'n'), (i ^ 66) % 26 = 18
When param_1 is 111 (character 'o'), (i ^ 66) % 26 = 19
When param_1 is 112 (character 'p'), (i ^ 66) % 26 = 24
When param_1 is 113 (character 'q'), (i ^ 66) % 26 = 25
When param_1 is 114 (character 'r'), (i ^ 66) % 26 = 22
When param_1 is 115 (character 's'), (i ^ 66) % 26 = 23
When param_1 is 116 (character 't'), (i ^ 66) % 26 = 2
When param_1 is 117 (character 'u'), (i ^ 66) % 26 = 3
When param_1 is 118 (character 'v'), (i ^ 66) % 26 = 0
When param_1 is 119 (character 'w'), (i ^ 66) % 26 = 1
When param_1 is 120 (character 'x'), (i ^ 66) % 26 = 6
```

```
When param_1 is 121 (character 'y'), (i ^ 66) % 26 = 7
When param_1 is 122 (character 'z'), (i ^ 66) % 26 = 4
```

The table below summarizes the analysis:

Target Char	ASCII	ASCII - 97	Desired Result from (x ^ 66) % 26	Valid Input ASCII	Valid Input Char
'a'	97	0	0	118	'v'
'r'	114	17	17	105	'i'
'i'	105	8	8	—	— (no solution)
's'	115	18	18	110	'n'
'h'	104	7	7	99	'c'
'e'	101	4	4	122	'z'
'm'	109	12	12	100	'd'

There is no lowercase character that produces the transformation result needed for 'i'. Therefore, there is no valid input that satisfies the full transformation.

The transformation logic used in this phase makes it unsolvable under the given constraints. Specifically, there is no lowercase input character that produces the transformed value corresponding to 'i' in "arishem". Therefore, Phase 5 is mathematically impossible.

Bypassing Phase 5

Binary Patching:

Since it is impossible to satisfy the transformation constraint in Phase 5, the decision was made to bypass this phase using binary patching.

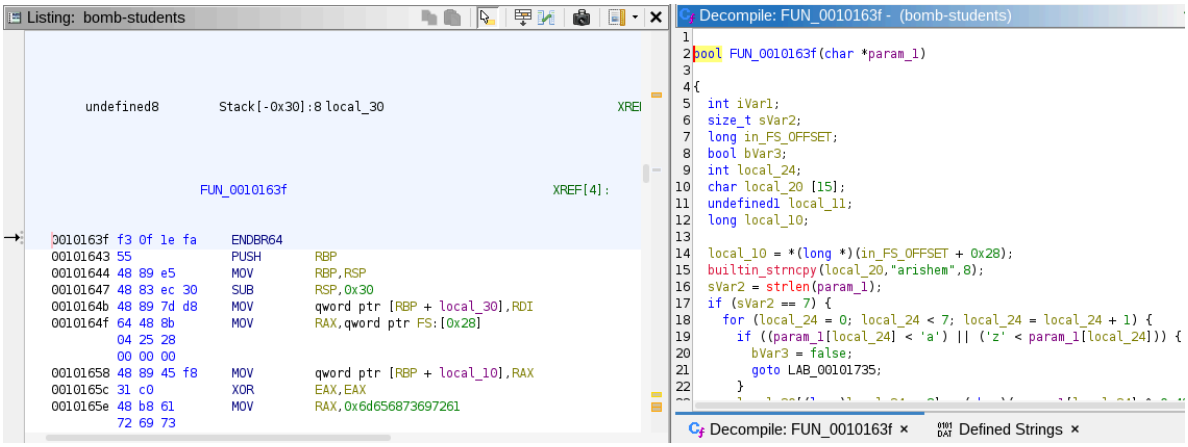
Steps Taken:

1. Patch FUN_0010163f to Always Return Success:
    - Apply the following patched instructions, starting four bytes after:

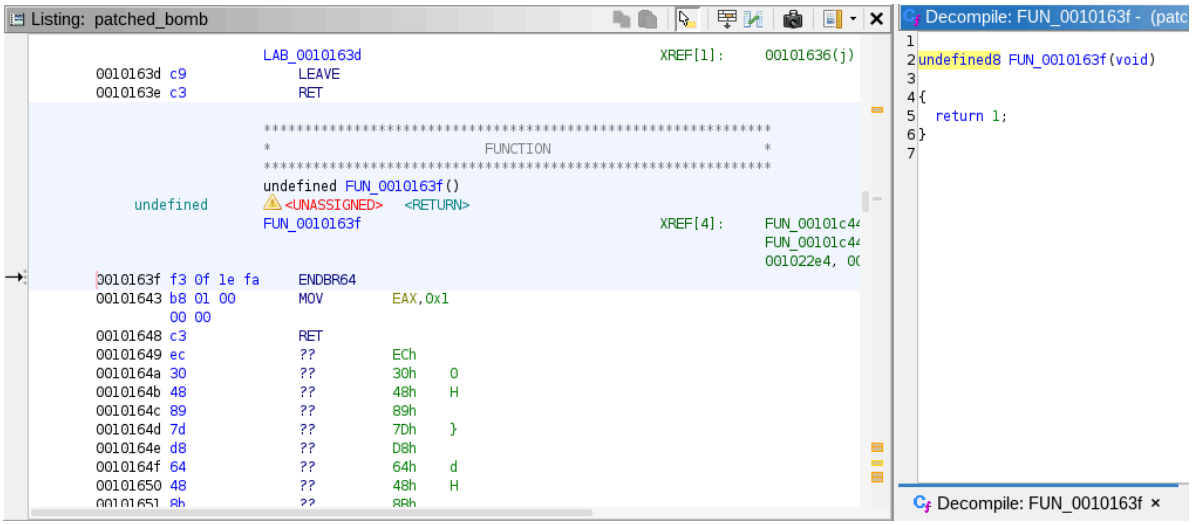
```
MOV EAX, 1
RET
```

    - This forces the function to always return true.

Before:

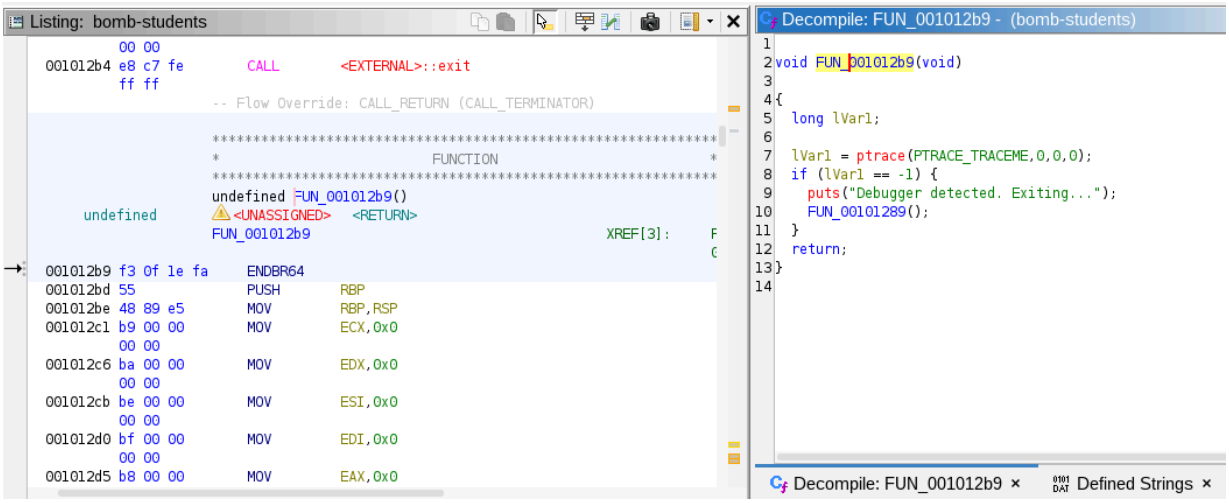


After:

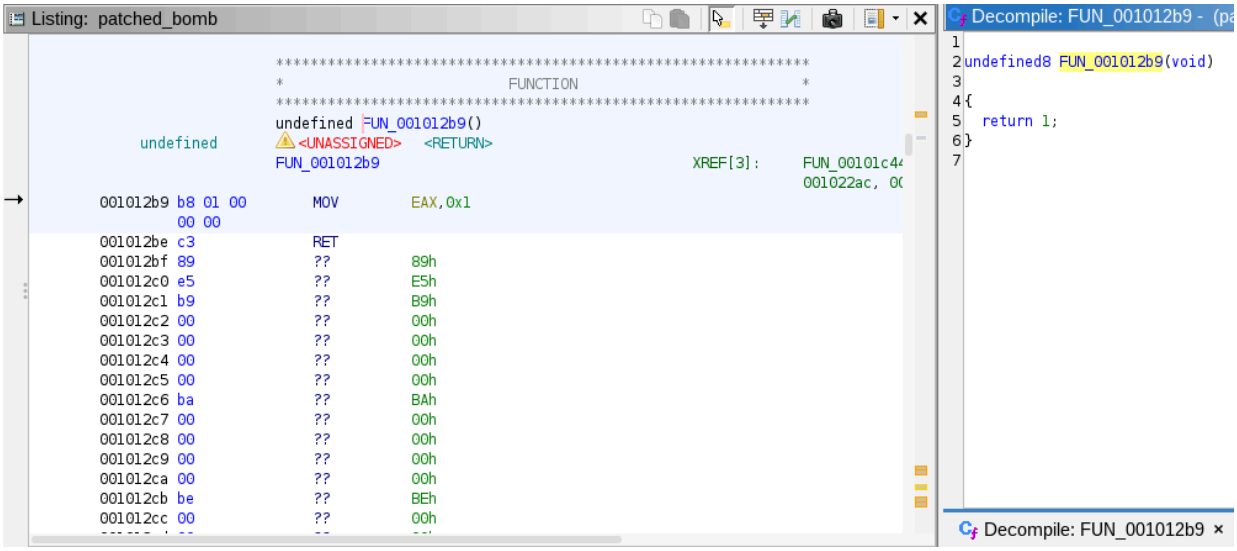


2. Patch FUN_001012b9:
    - Same approach as above, but starting with the function.

Before:



After:

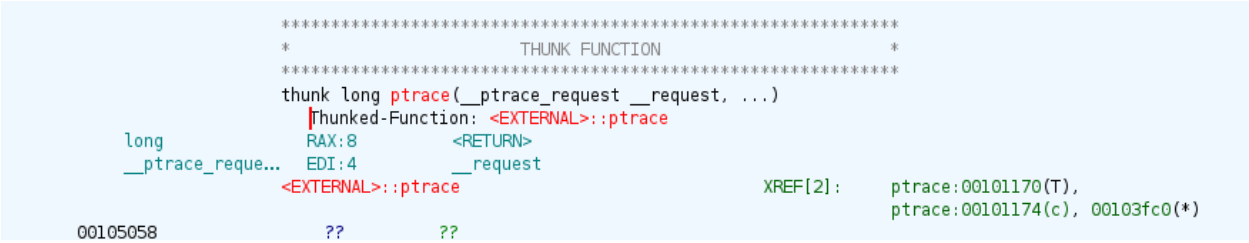


3. Disable ptrace Anti-Debugging Check:
    - Locate the thunk function for ptrace, typically at ptrace:00101170(T).
    - Replace its contents with:

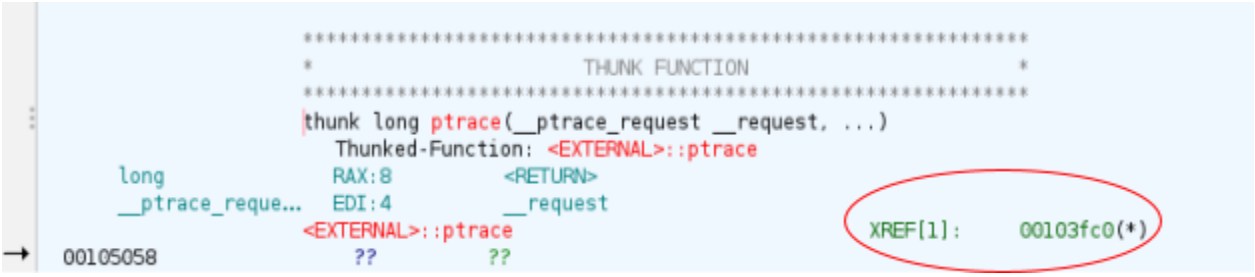
```
XOR EAX, EAX
RET
```

- This disables debugger detection by always returning 0.

Before:



After:



4. Recompile Patched Binary:
   - Export the patched binary from the disassembler and save it.
   - Execute this version for Phase 6 and beyond.

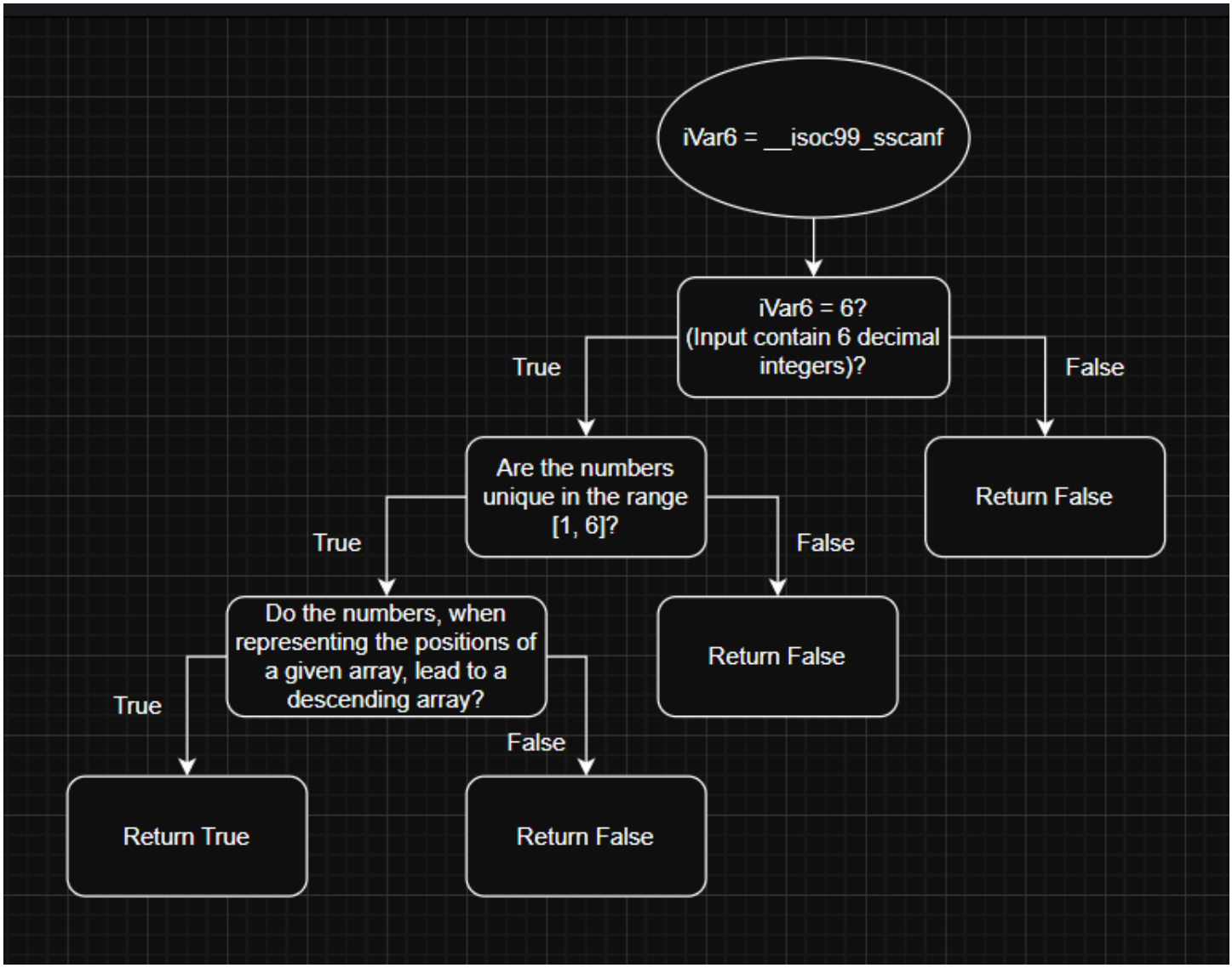
Phase 6: Linked List Traversal and Reordering

Decompiled Code:

```
undefined8 FUN_0010174b(undefined8 param_1)
{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  int local_90;
  int local_8c;
  int local_88[2];
  undefined1 local_80[4];
  undefined1 local_7c[4];
  undefined1 auStack_78[4];
  undefined1 auStack_74[12];
  int local_68[8];
  long local_48;
  undefined *local_40;
  undefined *local_38;
  undefined *local_30;
  undefined *local_28;
  undefined *local_20;
  undefined *local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_48 = 0;
  local_40 = &DAT_00104060;
  local_38 = &DAT_00104050;
  local_30 = &DAT_00104040;
  local_28 = &DAT_00104030;
  local_20 = &DAT_00104020;
  local_18 = &DAT_00104010;
  iVar1 = __isoc99_sscanf(param_1,"%d %d %d %d %d %d",local_88,local_88 + 1,local_80,
                          local_7c, auStack_78, auStack_74);
  if (iVar1 == 6) {
    local_68[0] = 0;
    local_68[1] = 0;
    local_68[2] = 0;
    local_68[3] = 0;
```

```
    local_68[4] = 0;
    local_68[5] = 0;
    local_68[6] = 0;
    for (local_90 = 0; local_90 < 6; local_90 = local_90 + 1) {
      if (((local_88[local_90] < 1) || (6 < local_88[local_90])) ||
         (local_68[local_88[local_90]] != 0)) {
        uVar2 = 0;
        goto LAB_00101909;
      }
      local_68[local_88[local_90]] = 1;
    }
    for (local_8c = 0; local_8c < 5; local_8c = local_8c + 1) {
      if (*(int *)((&local_48)[local_88[local_8c]] + 4) <
          *(int *)((&local_48)[local_88[local_8c + 1]] + 4)) {
        uVar2 = 0;
        goto LAB_00101909;
      }
    }
    uVar2 = 1;
  }
  else {
    uVar2 = 0;
  }
LAB_00101909:
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return uVar2;
}
```

Flow Diagram:



Analysis

Phase 6 evaluates a sequence of 6 integers. These integers are used as indices to traverse an array of structures, with the constraint that the resulting traversal must follow a strictly descending order based on the second field (i.e., offset +4) of each structure.

Input Parsing

The function expects 6 space-separated integers passed as a string:

```
__isoc99_sscanf(param_1, "%d %d %d %d %d %d", local_88,local_88 + 1,local_80,
                         local_7c, auStack_78, auStack_74)
```

Each of the six integers is written into contiguous memory starting at local_88. Despite only local_88[0] and local_88[1] being directly declared, the next four values local_80, local_7c, auStack_78, and auStack_74 are stored in variables immediately following local_88, making local_88 functionally a 6-element decimal-integer array.

Stack:

```
Higher memory addresses
┌──────────────────────────────────────┐
│ local_88[0]  (int)        │  <- first element
├──────────────────────────────────────┤
│ local_88[1]  (int)        │
├──────────────────────────────────────┤
│ local_80     (int)        │
├──────────────────────────────────────┤
│ local_7c     (int)        │
├──────────────────────────────────────┤
│ auStack_78   (int)        │
├──────────────────────────────────────┤
│ auStack_74   (int)        │  <- sixth element
└──────────────────────────────────────┘
Lower memory addresses
```

Constraints

Constraint 1: Valid and Unique Values Between 1 and 6

```
if (((local_88[local_90] < 1) || (6 < local_88[local_90])) ||
         (local_68[local_88[local_90]] != 0)) {
        uVar2 = 0;
        goto LAB_00101909;
}
```

- Each value must be between 1 and 6 (inclusive).
- No duplicates are allowed. A flag array local_68 ensures each value is used only once.

This implies the input must be a permutation of [1, 2, 3, 4, 5, 6].

Constraint 2: Descending Comparison of Structure Values

```
if (*(int *)((&local_48)[local_88[i]] + 4) < *(int *)((&local_48)[local_88[i+1]] + 4))
{
    uVar2 = 0;
    goto LAB_00101909;
}
```

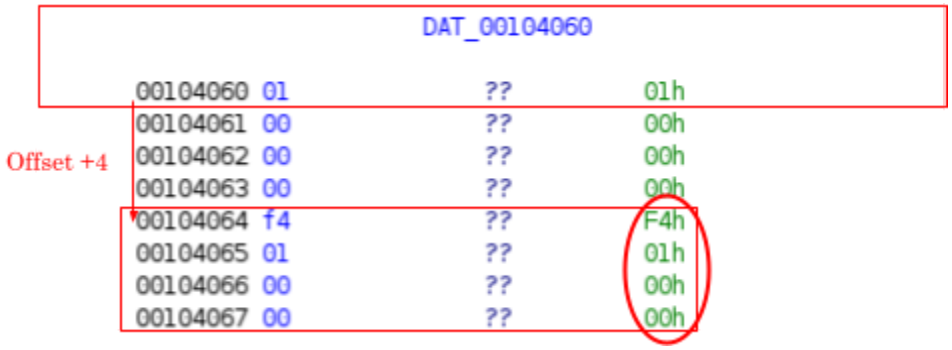
This accesses the second field (offset +4) of each structure using the index provided in local_88. These structure pointers are pre-initialized:

```
local_48 = 0;
local_40 = &DAT_00104060;
local_38 = &DAT_00104050;
local_30 = &DAT_00104040;
local_28 = &DAT_00104030;
local_20 = &DAT_00104020;
local_18 = &DAT_00104010;
```

These form a lookup array from indices 1 to 6:

Index	Pointer	+4 Offset Value (hex)	+4 Offset Value (decimal)
1	DAT_00104060	0x01F4	500
2	DAT_00104050	0x00C8	200
3	DAT_00104040	0x0064	100
4	DAT_00104030	0x0190	400
5	DAT_00104020	0x012C	300
6	DAT_00104010	0x0258	600

How to calculate, using example DAT_00104060:



00 00 01 F4 in little-endian format; 0x01F4 in hexadecimal format. To convert to decimal:

$$1 * 16^2 + 15 * 16^1 + 4 * 16^0 = 256 + 240 + 4 = 500$$

To pass the second for-loop, the values at offset +4 of each pointer (dereferenced by the permutation indices) must be in descending order.
So we want:

```
value_at_index[local_88[0]] >= value_at_index[local_88[1]] >= ... >=
value_at_index[local_88[5]]
```

Using the values:

Index	Value at +4
6	600
1	500
4	400
5	300
2	200
3	100

This results in the correct input:

```
6 1 4 5 2 3
```

This is a valid permutation of [1–6] and produces descending values at the +4 offset of each structure.

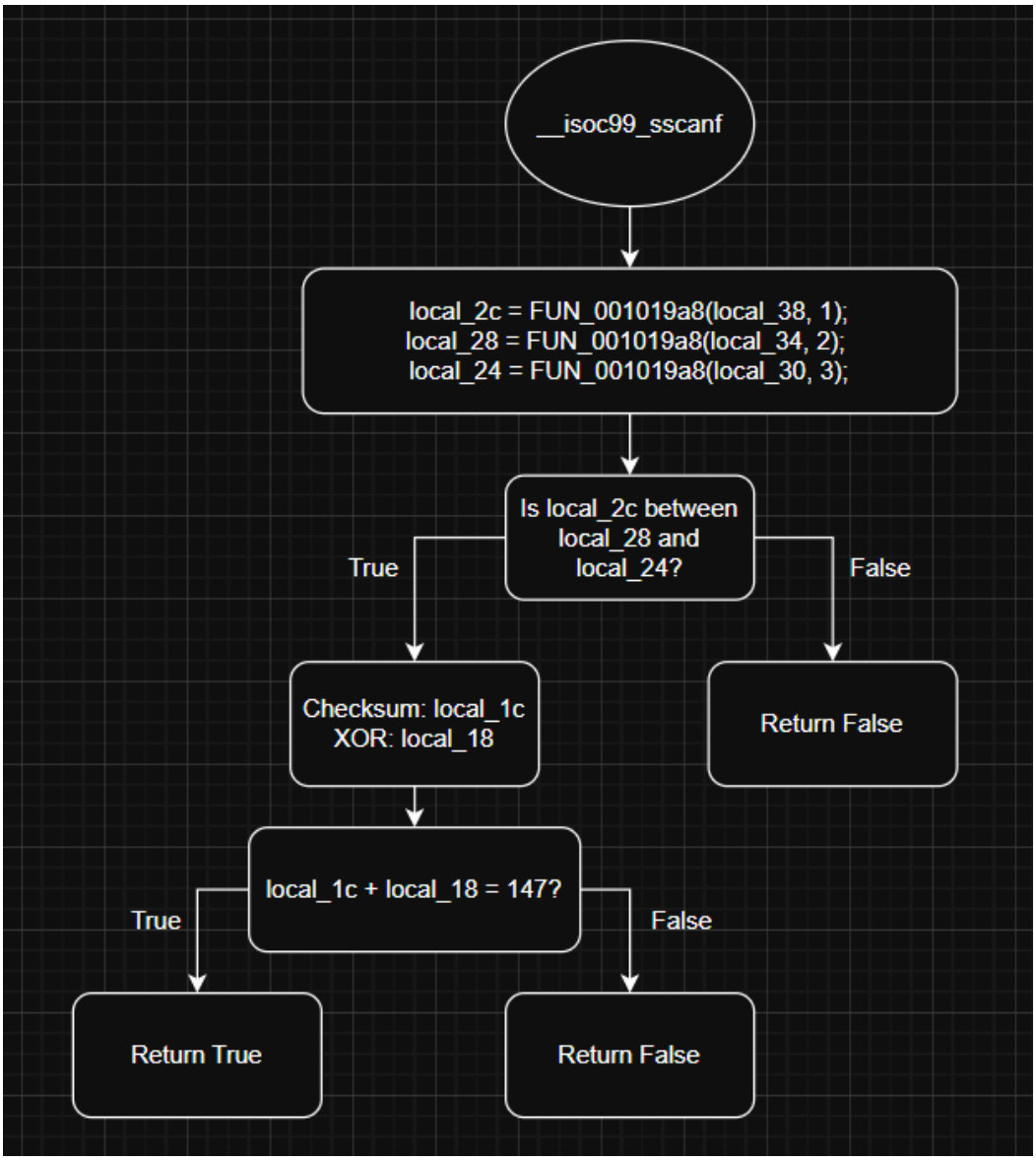
Phase 7 — Surprise !!

Decompiled Code:

```
bool FUN_00101a3b(undefined8 param_1)
{
  int iVar1;
  long in_FS_OFFSET;
  bool bVar2;
  uint local_38, local_34, local_30;
  uint local_2c, local_28, local_24;
  int local_20;
  uint local_1c, local_18;
  int local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = __isoc99_sscanf(param_1,"%d %d %d",&local_38,&local_34,&local_30);
  if (iVar1 == 3) {
    local_2c = FUN_001019a8(local_38, 1);
    local_28 = FUN_001019a8(local_34, 2);
    local_24 = FUN_001019a8(local_30, 3);

    if ((local_2c == local_28) || (local_2c == local_24) || (local_28 == local_24)) {
      bVar2 = false;
    } else {
      if (((int)local_28 < (int)local_2c && (int)local_2c < (int)local_24) ||
          ((int)local_24 < (int)local_2c && (int)local_2c < (int)local_28))) {
        local_20 = 1;
      } else {
        local_20 = 0;
      }

      if (local_20 == 0) {
        bVar2 = false;
      } else {
        local_1c = local_30 & local_24 | local_38 & local_2c | local_34 & local_28;
        local_18 = local_30 ^ local_38 ^ local_34 ^ local_2c ^ local_28 ^ local_24;
        local_14 = local_18 + local_1c;
        bVar2 = local_14 == 0x93;
      }
    }
  } else {
```

```
    bVar2 = false;
  }

  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
    __stack_chk_fail();
  }
  return bVar2;
}
```

Flow Diagram:



Analysis:

This phase takes three space-separated decimal integers as input. These values undergo a series of custom transformations and logical operations, followed by multiple conditional checks involving:

- Value uniqueness
- Strict ordering
- Bitwise AND and XOR arithmetic

Ultimately, the program verifies whether the computed checksum equals a target constant (0x93 or decimal 147).

Input Parsing

```
__isoc99_sscanf(param_1, "%d %d %d", &local_38, &local_34, &local_30);
```

- Inputs: local_38, local_34, local_30
- All values are integers, parsed from the string param_1.

Constraints and Conditions

Constraint 1: All transformed values must be unique

```
if (local_2c == local_28 || local_2c == local_24 || local_28 == local_24) {
    bVar2 = false;
}
```

Constraint 2: Ordering

```
if ((local_28 < local_2c < local_24) || (local_24 < local_2c < local_28)) {
}
```

This implies local_2c must be between local_28 and local_24.

Constraint 3: Bitwise Operations

```
local_1c = local_30 & local_24 | local_38 & local_2c | local_34 & local_28;
local_18 = local_30 ^ local_38 ^ local_34 ^ local_2c ^ local_28 ^ local_24;
local_14 = local_18 + local_1c;
bVar2 = (local_14 == 0x93);
```

The total sum of the bitwise XOR of all six values, plus a bitwise AND/OR hybrid, must equal 147.

Brute-force Search Strategy

Python Script

```
def FUN_001019a8(param_1, param_2):


  if param_2 == 3:
    return param_1 / 4 + param_1 * 5

  if param_2 < 4:
    if (param_2 == 1):
      return param_1 ^ 0x2a

    if (param_2 == 2):
      return (param_1 + 0x11) % 0x65 ^ 0xf


  print("Error: Invalid round_id in oracle_transform.")

  return 0;



target_value = 0x93
break_time = False

for local_38 in range(0, 100):
    local_2c = FUN_001019a8(local_38, 1)
    for local_34 in range(0, 100):
        local_28 = FUN_001019a8(local_34, 2)
        for local_30 in range(0, 100):
            local_24 = FUN_001019a8(local_30, 3)

            local_30 = int(local_30)
            local_24 = int(local_24)
            local_38 = int(local_38)
            local_2c = int(local_2c)
            local_34 = int(local_34)
            local_28 = int(local_28)

            if (local_24 < local_2c and local_2c < local_28) or (local_28 < local_2c
```

```
and local_2c < local_24):
                if (local_30 & local_24 | local_38 & local_2c | local_34 & local_28) +
(local_30 ^ local_38 ^ local_34 ^ local_2c ^ local_28 ^ local_24) == target_value:
                    print(f"{local_38} {local_34} {local_30}")
                    break_time = True
                    break
        if break_time:
            break
    if break_time:
        break
```

Brute-force Key Elements:

- Iterates all combinations of local_38, local_34, and local_30 from 0 to 99
- Applies the three custom transformations
- Checks all constraints in order
- Stops at first valid match
- Time complexity: $O(n^3)$ in worst case (up to 1 million iterations)

Output:

```
0 84 42
```

The above numbers correspond as such:

```
local_38 = 0
local_34 = 84
local_30 = 42
```

Final Answer:

```
0 84 42
```

Solutions

phase_inputs.txt

```
Were located at 63-129 in Engineering IV
1 3 7 15 31 63
2 blue
10SMARTT
15660 15677

6 1 4 5 2 3
0 84 42
```