Assignment 5: Attack Phase

Secure Systems Engineering

(CS 6510)

Submitted To:

Prof. Chester Rebeiro
Department of
Computer Science Engineering

Submitted By:

Arnav Karn (CS24M801)
Dipanshu Kumar (CS24M019)

Team 91 - Binary Analysis Report

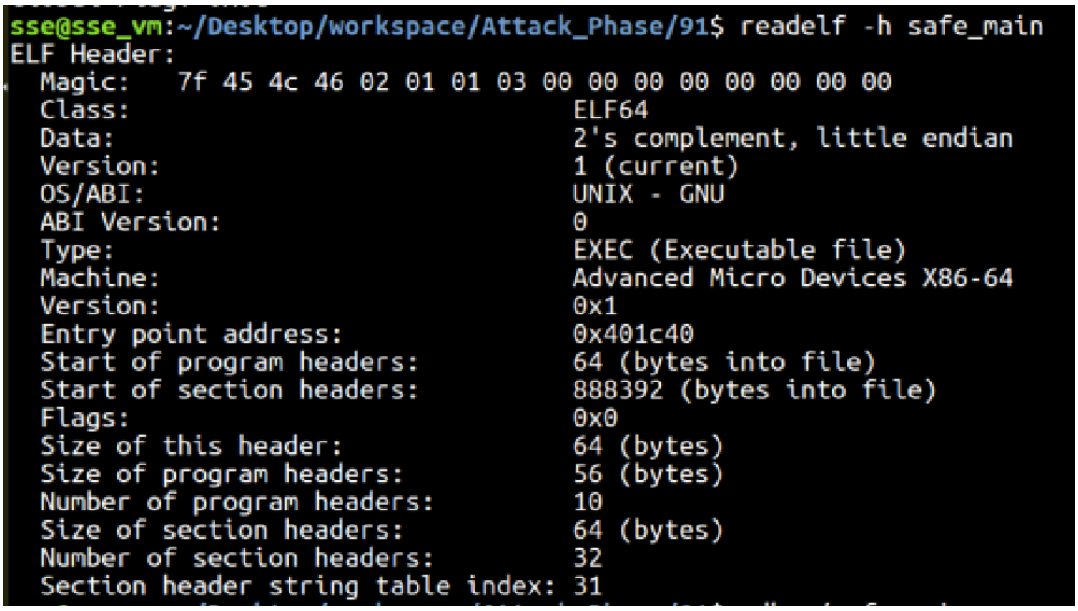
Tools Used: Ghidra, GDB

Approach:

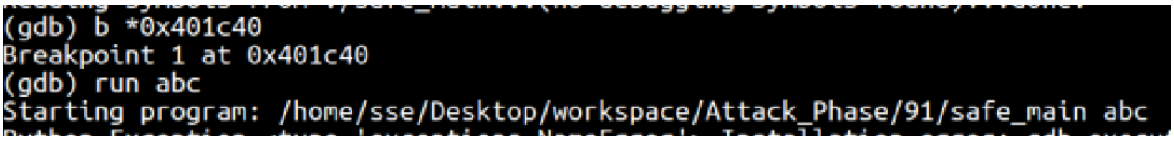
1. **Initial Analysis & Main Function Identification:**
   This was the first binary tackled, so the approach was broad initially. The binary was decompiled in Ghidra, and the `main` function was located.

2. **Egg Extraction:**

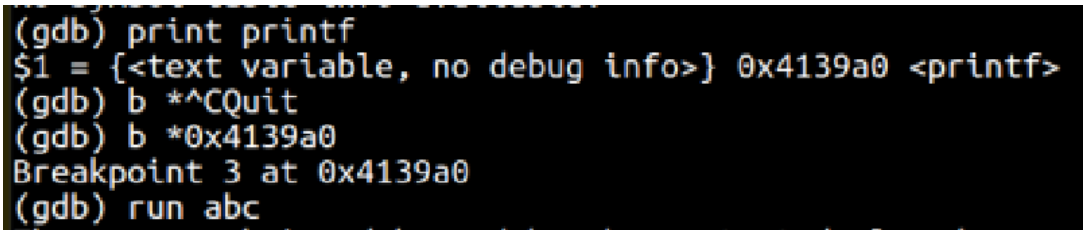
   - The strategy involved setting a breakpoint at the program's entry point using **readelf -h safe_main**.



   - Break at the entry point and run the safe_main.



   - Then, the address of `printf` was identified, and a breakpoint was placed there.



- As the program executed, each `printf` was intercepted until the first plaintext (egg 0) was observed.

```
Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
```

- And continued till I saw the Egg 0 printed.

```
Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.
Ciphertext:: a5 23 e2 4f 51 f8 13 81 2d f1 8c 37 3d 36 24 18

Breakpoint 3, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders funct
s missing:
0x00000000004139a0 in printf ()
(gdb) c
Continuing.
Egg 0 : 0x51
```

- At this point, the egg values were dumped from memory after identifying the address of the egg..

```
(gdb) x/6xb 0x4c7345
0x4c7345 <eggs>:        0x51    0xf0    0xe0    0x7b    0xb5    0x00
```

3. **Key Retrieval:**

   - A function named F4 was identified, which called another function F1, passing the AES key as the second parameter.

   - Inside F1, there was no obfuscation applied to the key handling.

```
                        f1                                              XRE
00401d65 f3 0f 1e fa    ENDBR64
00401d69 55             PUSH        RBP
00401d6a 48 89 e5       MOV         RBP,RSP
00401d6d 48 83 ec 30    SUB         RSP,0x30
00401d71 48 89 7d d8    MOV         qword ptr [RBP + local_30],RDI
00401d75 48 89 75 d0    MOV         qword ptr [RBP + local_38],RSI
00401d79 64 48 8b 04    MOV         RAX,qword ptr FS:[0x28]
         25 28 00 00 00
00401d82 48 89 45 f8    MOV         qword ptr [RBP + local_10],RAX
00401d86 31 c0          XOR         EAX,EAX
00401d88 c7 45 e8 00    MOV         dword ptr [RBP + local_20],0x0
         00 00 00
00401d8f e9 a2 00 00    JMP         LAB_00401e36
                        LAB_00401d94                                    XRE
```

```
void f1(long param_1,long param_2)

{
  long lVar1;
  uint uVar2;
  uint uVar3;
  long in_FS_OFFSET;
  uint local_20;
  byte local_14;
  byte local_13;
  byte local_12;
  byte local_11;

  lVar1 = *(long *)(in_FS_OFFSET + 0x28);
  for (local_20 = 0; local_20 < 4; local_20 = local_20 + 1) {
    *(undefined1 *)(param_1 + (ulong)(local_20 << 2)) =
         *(undefined1 *)(param_2 + (ulong)(local_20 << 2));
    *(undefined1 *)(param_1 + (ulong)(local_20 * 4 + 1)) =
         *(undefined1 *)(param_2 + (ulong)(local_20 * 4 + 1));
    *(undefined1 *)(param_1 + (ulong)(local_20 * 4 + 2)) =
         *(undefined1 *)(param_2 + (ulong)(local_20 * 4 + 2));
    *(undefined1 *)(param_1 + (ulong)(local_20 * 4 + 3)) =
         *(undefined1 *)(param_2 + (ulong)(local_20 * 4 + 3));
  }
```

   - A breakpoint was placed at the start of F4, and the value of ESI was dumped, which revealed the key because it was ESI holding the value of param_2.

```
(gdb) info reg
rax            0x7fffffffdcf0   140737488346352
rbx            0x400518 4195608
rcx            0x453ae7 4537063
rdx            0x7fffffffdda0   140737488346528
rsi            0x7fffffffdda0   140737488346528
rdi            0x7fffffffdcf0   140737488346352
rbp            0x7fffffffdc70   0x7fffffffdc70
rsp            0x7fffffffdc48   0x7fffffffdc48
r8             0xa        10
r9             0x3        3
r10            0x1        1
r11            0x246      582
r12            0x405dc0 4218304
r13            0x0        0
```

```
(gdb) x/16xb 0x7fffffffdda0
0x7fffffffdda0: 0xa0    0x51    0x38    0xb9    0x44    0xfc    0x3a    0xa0
0x7fffffffdda8: 0x08    0xfc    0xce    0xf7    0xa4    0x21    0x42    0xb0
(gdb)
```

4. **Egg Parameter (`egg_param`) Handling:**

    - The logic for computing egg parameters involved a sequence of nested functions and multiple `switch` cases, making the code flow complex and hard to reverse manually.

    - To simplify the process, the entire egg parameter generation logic was copied from the decompiled code and reused directly in our own `main.c`.

    - The logic replicated the egg parameter behavior accurately without needing to reverse the entire structure.

5. **Global Flag Computation:**
    - The function responsible for global flag computation was clearly named `F15`.

```
int f15(long param_1)

{
  return (uint)*(byte *)(param_1 + 4) * (uint)*(byte *)(param_1 + 3) * 7 -
         (uint)*(byte *)(param_1 + 4);
}
```

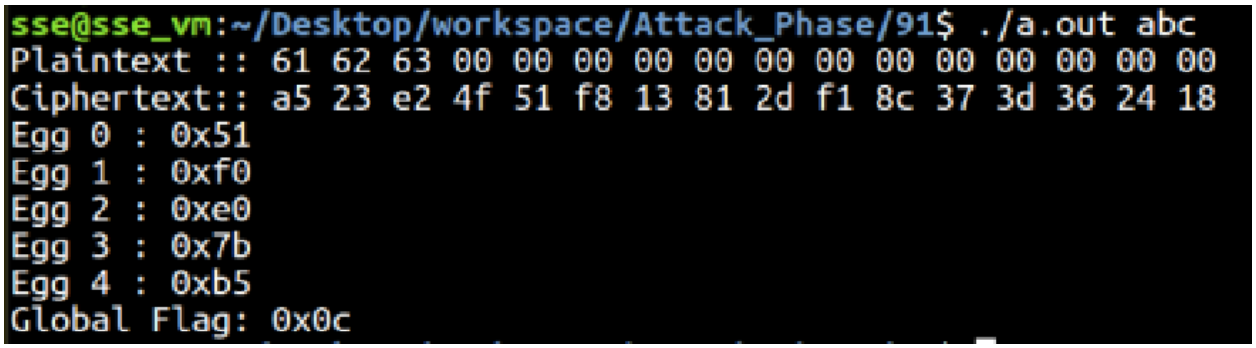
    - Its logic was straightforward and unobfuscated.

    - The computation logic was directly translated and re-implemented into the attacker's own `main.c`, allowing accurate reproduction of the global flag.

```
138 uint8_t compute_gf(uint8_t* eggs) {
139     return eggs[4] * eggs[3] * 7 - eggs[4];
140 }
```

Conclusion:

Despite some redundant efforts early on, the binary was successfully broken. Eggs were extracted using live `printf` tracing. Keys were obtained by monitoring function calls without obfuscation, and the egg parameter logic was directly reused due to its complexity. The overall lack of code protection enabled a complete bypass via code reuse and strategic breakpoints.



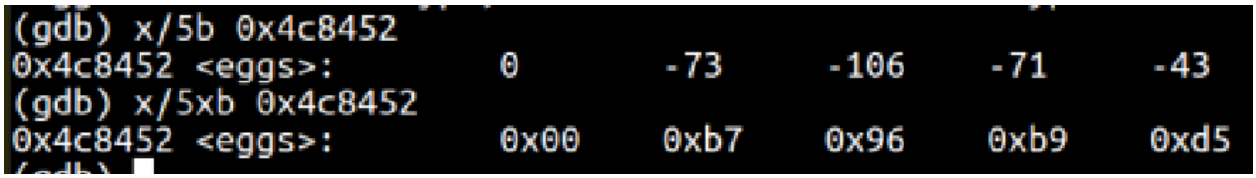
Team 24 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Decompilation and Main Function Identification:**
   The binary was decompiled using Ghidra. The `main` function was located using the function filter. The lack of obfuscation made analysis straightforward.

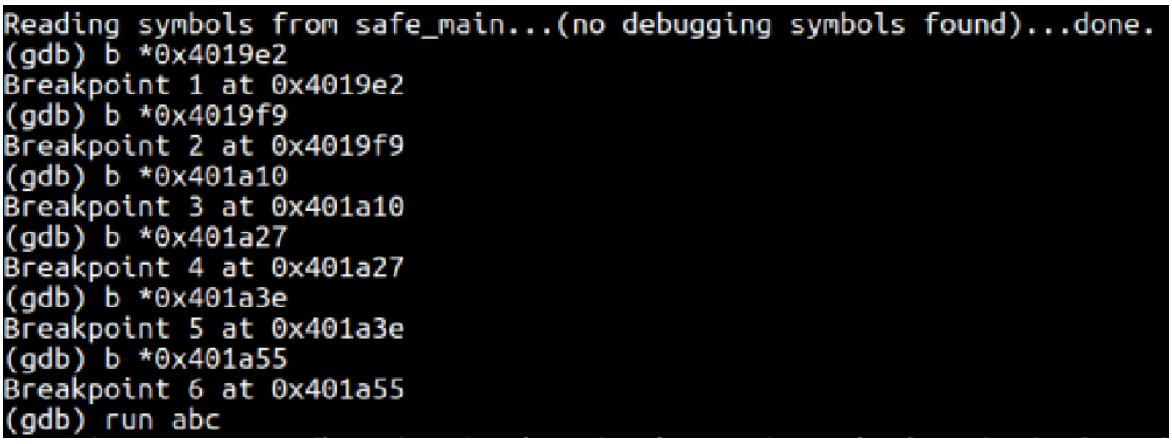
2. **Egg Extraction:**
   A breakpoint was set at the first `printf` of `egg 0`. From there, the memory addresses containing egg values were identified and dumped directly. The egg values were accessible via known offsets from the main function.



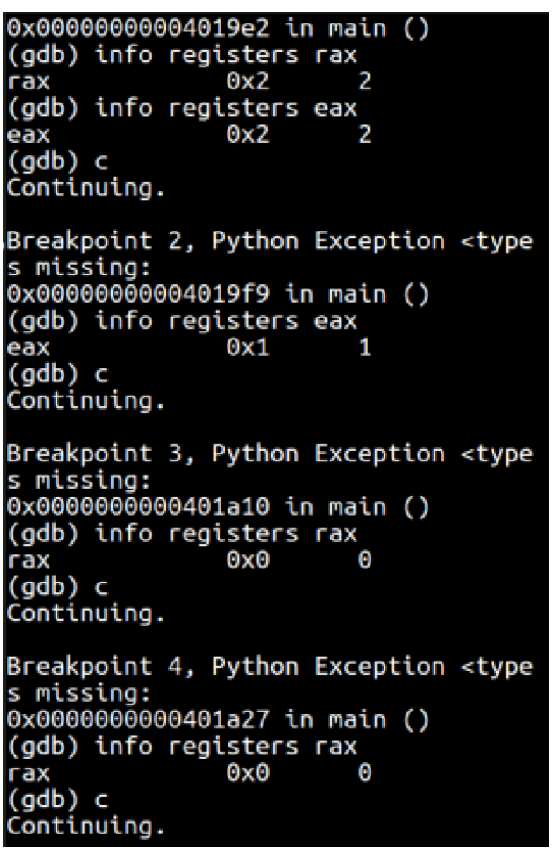
3. **Key Retrieval:**
   The AES keys were hardcoded and present clearly within the `main` function. No encryption or obfuscation techniques were used to hide them.

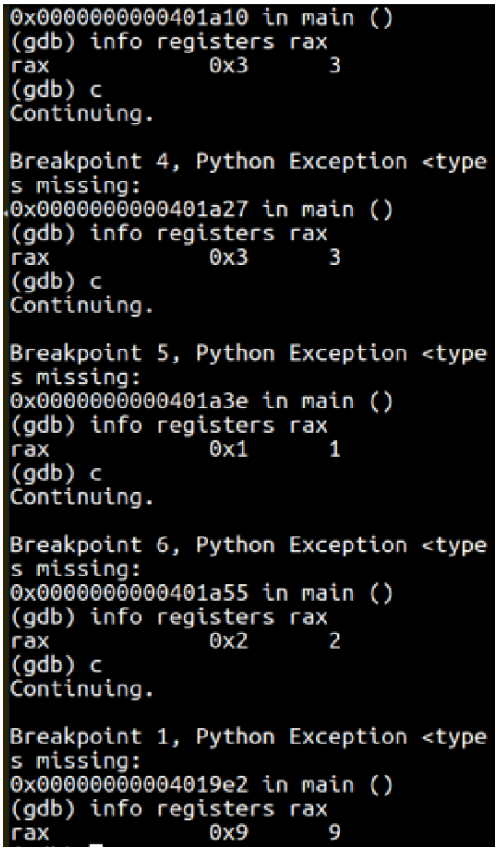
4. **Egg Parameter (`egg_param`) Extraction:**
   The egg parameters were dynamically calculated and stored in the `EAX` register immediately after a function call within a loop.

   - A breakpoint was placed just after the function call that generated the egg parameters.



- Values of `EAX` were dumped for each iteration.





- Duplicate parameter rows were observed and skipped, retaining only unique sets.

5. **Egg Parameter Erasure Check:**
   It was noticed that the `egg_param` values were cleared after execution, making live debugging essential for extraction.

6. **Computation Function:**
   The compute function was found directly within the `main` function and was not isolated.
   The logic was reverse-engineered by analyzing the relation between the egg's base address and output, which matched the expected computation.
7. **Correct Global flag computation:**

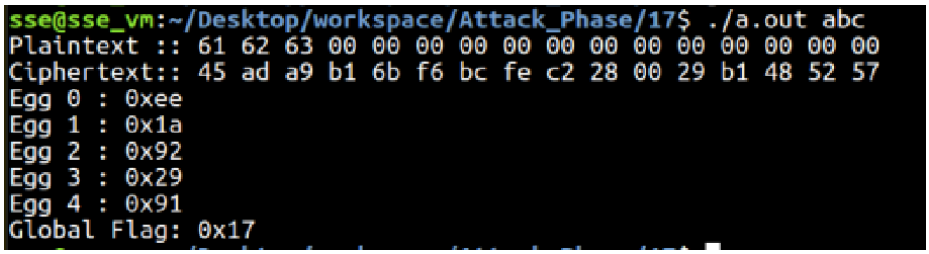
```
uint32_t compute_gf(uint8_t* eggs) {

 return (11 * eggs[1] - eggs[4]) + (-68 * eggs[3]);

}
```

Conclusion:

Due to the lack of obfuscation and direct access to critical values in the `main` function, extraction of eggs, keys, and parameters was relatively simple. Strategic breakpoints and live memory inspection allowed complete recovery of required data.

```
sse@sse_vm:~/Desktop/workspace/Attack_Phase/24$ ./a.out abc
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext:: f7 76 c7 e0 fb eb c6 cb aa b3 13 aa 0b e3 32 4a
Egg 0 : 0x00
Egg 1 : 0xb7
Egg 2 : 0x96
Egg 3 : 0xb9
Egg 4 : 0xd5
Global Flag: 0xe4
```

Team 17:
Made Main.c and it output:



Team 7 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Initial Decompilation and Main Function Location:**
   The binary was decompiled using Ghidra, and the `main` function was located using the entry point reference.

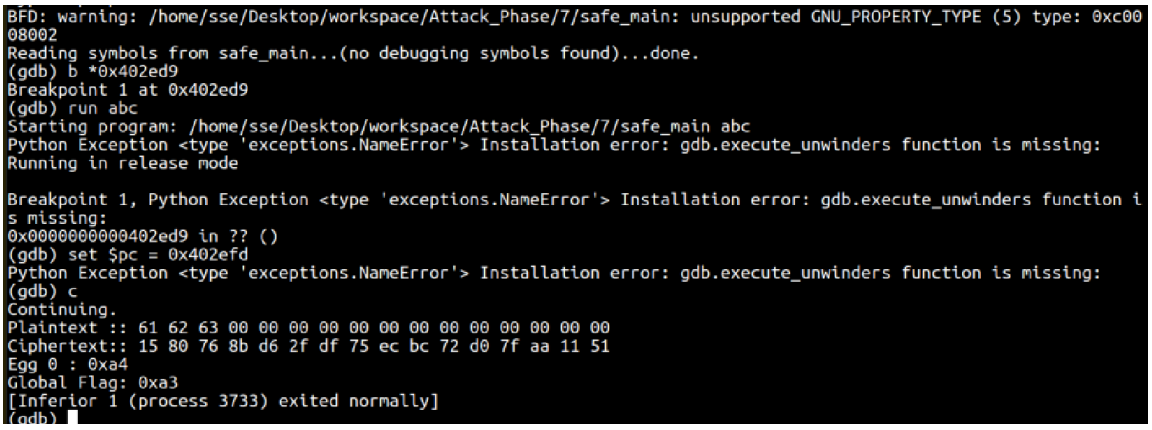
2. **Anti-Debugging Bypass:**

   - The binary included anti-debugging logic that prevented standard debugging with GDB.

```
FUN_00411c90("Running in release mode");
iVar1 = FUN_00401a11();
if (iVar1 != 0) {
  FUN_00411c90("Debugger detected! Halting program...");
  FUN_00401a65();
}
if (param_1 < 2) {
  FUN_00411c90("Invalid Usage!!");
  FUN_00411c90("Usage: ./encrypt <plain_text>");
  uVar2 = 1;
}
```

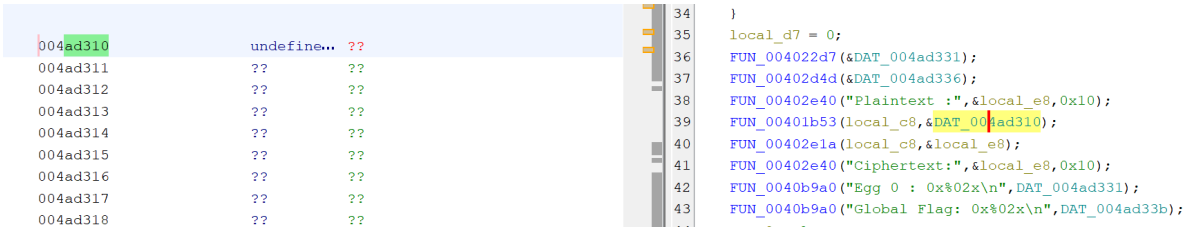
   - To bypass this, a breakpoint was set at the beginning of the anti-debugging routine.

   - The program counter (PC) was then manually set to the end of the anti-debugging block, effectively skipping the detection mechanism.



3. **Key and Egg Extraction:**

   - Upon analyzing the `main` function, the memory addresses where the AES keys and egg values were stored were identified.



   - Set the breakpoint at the print of the global flag.

```
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x000000000040b9a0 in ?? ()
(gdb) c
Continuing.
Ciphertext:: 15 80 76 8b d6 2f df 75 ec bc 72 d0 7f aa 11 51

Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x000000000040b9a0 in ?? ()
(gdb) c
Continuing.
Egg 0 : 0xa4

Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x000000000040b9a0 in ?? ()
```

   - These values were dumped directly using the identified addresses.

```
(gdb) x/5xb 0x4ad331
0x4ad331:       0xa4    0xa9    0x07    0xf5    0x2a
(gdb) x/16xb 0x4ad310
0x4ad310:       0x1e    0xba    0xe5    0x4a    0xe2    0xae    0x66    0xae
0x4ad318:       0x72    0x5c    0x17    0x7c    0xcc    0x51    0x84    0x43
```

4. **Egg Parameter (`egg_param`) Retrieval:**

   - The egg parameters were stored in a global variable.

   - By locating this global variable, the egg parameter values were dumped at the end of the `cipher` function.

   - No obfuscation was applied to the egg parameter logic, making extraction straightforward.

```
(gdb) x/30xb 0x4ad340
0x4ad340:       0x01    0x01    0x02    0x00    0x01    0x01    0x02    0x04
0x4ad348:       0x03    0x02    0x00    0x00    0x03    0x03    0x02    0x02
0x4ad350:       0x02    0x00    0x08    0x04    0x02    0x01    0x01    0x00
0x4ad358:       0x09    0x03    0x01    0x03    0x03    0x01
(gdb)
```

5. **Global Flag Computation:**

   - The computation of the global flag involved multiple obfuscated helper functions.

```
1
2 int FUN_00402580(long param_1)
3
4 {
5   int iVar1;
6   int iVar2;
7   int iVar3;
8
9   iVar1 = FUN_00402454(*(undefined1 *)(param_1 + 2));
10  iVar2 = FUN_00402509(*(undefined1 *)(param_1 + 1));
11  iVar3 = FUN_004024af(*(undefined1 *)(param_1 + 2));
12  return iVar3 + (iVar1 - iVar2);
13 }
14
```

   - Rather than reversing the entire structure, all relevant functions were copied from the decompiled code and converted into runnable C code.

   - These were then pasted into the our own `main.c` and executed successfully, reproducing the correct global flag.

```
sse@sse_vm:~/Desktop/workspace/Attack_Phase/7$ ./safe_main abc
Running in release mode
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext:: 15 80 76 8b d6 2f df 75 ec bc 72 d0 7f aa 11 51
Egg 0 : 0xa4
Global Flag: 0xa3
sse@sse_vm:~/Desktop/workspace/Attack_Phase/7$ ./a.out abc
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext:: 15 80 76 8b d6 2f df 75 ec bc 72 d0 7f aa 11 51
Egg 0 : 0xa4
Egg 1 : 0xa9
Egg 2 : 0x07
Egg 3 : 0xf5
Egg 4 : 0x2a
Global Flag: 0xa3
```

Conclusion:

Team 7 attempted to implement anti-debugging but left an easily skippable section. Key data like eggs, keys, and parameters were stored unprotected in memory. Although the global flag logic was heavily split across multiple functions, code reuse from decompilation provided a reliable bypass. Overall, the attack succeeded with moderate effort due to weak obfuscation and reliance on static data.

Team 31 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

Reverse Engineering compute_gf() Logic from Binary

To understand the behavior of the compute_gf() function used in the binary, we analyzed both its disassembled machine code and its corresponding decompiled C-like representation. The function operates on a 5-byte array eggs, and we aimed to express its logic in the form:

```
uint8_t compute_gf(uint8_t* eggs){
        return A*eggs[X] $ B*eggs[Y] @ C*eggs[Z];
        // where A, B, C are constants; X, Y, Z are indices; and $,@ ∈ {+, -, *}
}
```

Step 1: Locating `compute_gf` in Ghidra

Using Ghidra for disassembly, I identified the `compute_gf` function explicitly — it was clearly labeled and distinguishable in the decompiled view. Upon inspecting the function, I noticed it was composed of multiple nested function calls. These helper functions returned intermediate values that were ultimately combined to perform the main `compute_gf` computation.

Step 2: Replicating the Function Outside the Binary

To understand the inner workings, I recreated the `compute_gf` function in my own local development environment. This included:

- Copying the main function logic from Ghidra.
- Re-implementing each nested function that `compute_gf` depended on.
- Running the function with different input arrays to observe output variations.

Step 3: Analyzing Constant Behavior

By testing the function with different versions of the `egg` array, I observed that certain intermediate values remained unchanged across runs. This indicated that those parts of the computation were using fixed constants — most likely the hardcoded ABC values

Step 4: Index Correlation and Reverse Mapping

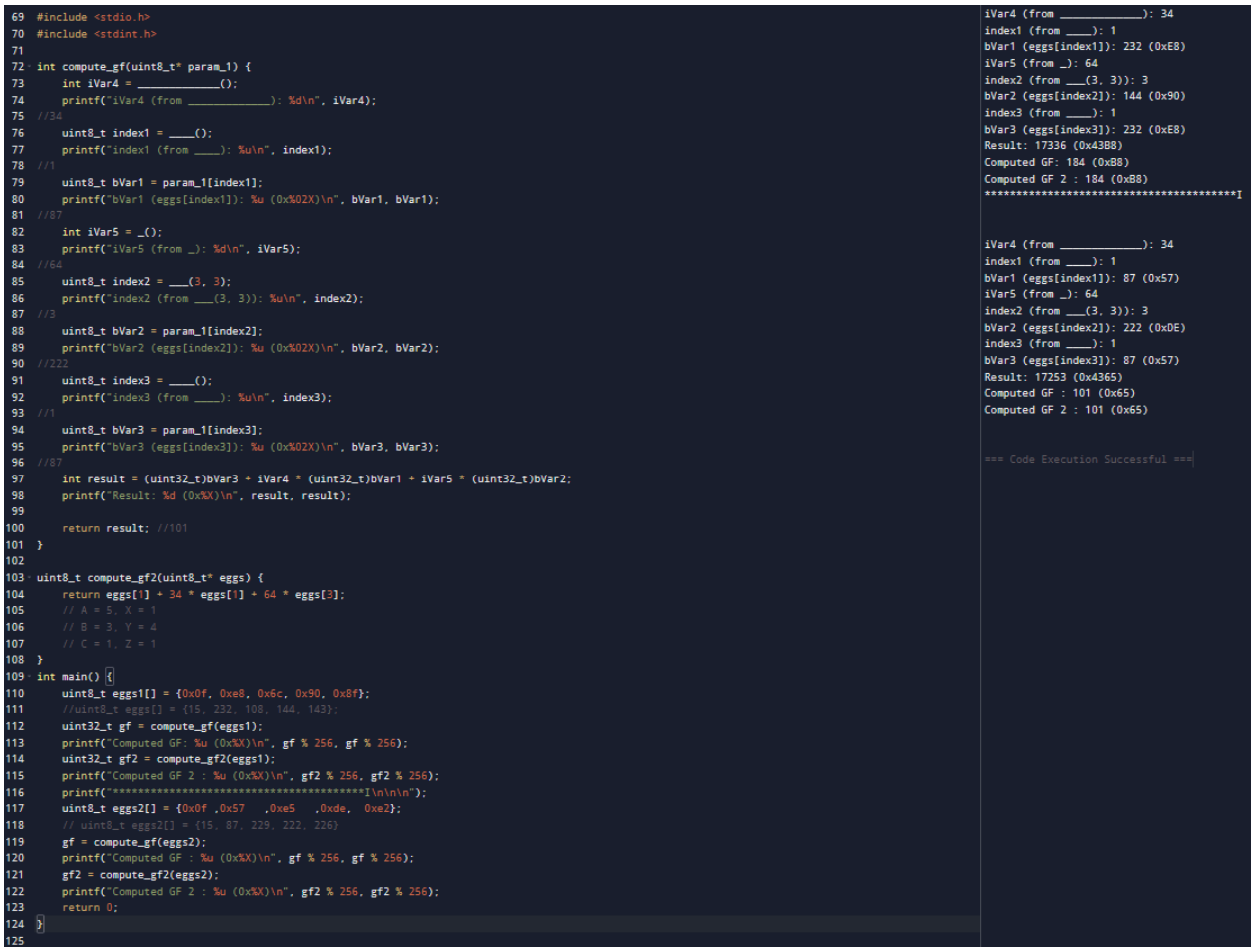
I traced how the input array indices were being used and noticed that the same indexes were accessed across both input arrays. This consistent access pattern allowed me to infer which indices were actually involved in the computation. By isolating these, I could match them back to the XYZ indexes known values of the `egg` arrays.

Step 5: Validation of Deobfuscation

To confirm my understanding, I created a simplified version of the `compute_gf` function that stripped away all obfuscation and nested indirections. I ran both the original and the simplified versions on the same inputs and verified that they produced identical outputs for both versions of the `egg` array.

The figure attached demonstrates this equivalence: it visually compares the original and simplified outputs side-by-side, proving that the core computation has been successfully recovered.

eggs[1] + 34 * eggs[1] + 64 * eggs[3]



```
#include <stdio.h>
#include <stdint.h>

int compute_gf(uint8_t* param_1) {
    int iVar4 = ____________();
    printf("iVar4 (from ____________): %d\n", iVar4);
//34
    uint8_t index1 = ____();
    printf("index1 (from ____): %u\n", index1);
//1
    uint8_t bVar1 = param_1[index1];
    printf("bVar1 (eggs[index1]): %u (0x%02X)\n", bVar1, bVar1);
//87
    int iVar5 = _();
    printf("iVar5 (from _): %d\n", iVar5);
//64
    uint8_t index2 = ___(3, 3);
    printf("index2 (from ___(3, 3)): %u\n", index2);
//3
    uint8_t bVar2 = param_1[index2];
    printf("bVar2 (eggs[index2]): %u (0x%02X)\n", bVar2, bVar2);
//222
    uint8_t index3 = ____();
    printf("index3 (from ____): %u\n", index3);
//1
    uint8_t bVar3 = param_1[index3];
    printf("bVar3 (eggs[index3]): %u (0x%02X)\n", bVar3, bVar3);
//87
    int result = (uint32_t)bVar3 + iVar4 * (uint32_t)bVar1 + iVar5 * (uint32_t)bVar2;
    printf("Result: %d (0x%X)\n", result, result);

    return result; //101
}

uint8_t compute_gf2(uint8_t* eggs) {
    return eggs[1] + 34 * eggs[1] + 64 * eggs[3];
    // A = 5, X = 1
    // B = 3, Y = 4
    // C = 1, Z = 1
}
int main() {
    uint8_t eggs1[] = {0x0f, 0xe8, 0x6c, 0x90, 0x8f};
    //uint8_t eggs[] = {15, 232, 108, 144, 143};
    uint32_t gf = compute_gf(eggs1);
    printf("Computed GF: %u (0x%X)\n", gf % 256, gf % 256);
    uint32_t gf2 = compute_gf2(eggs1);
    printf("Computed GF 2 : %u (0x%X)\n", gf2 % 256, gf2 % 256);
    printf("****************************************I\n\n\n");
    uint8_t eggs2[] = {0x0f ,0x57   ,0xe5   ,0xde,  0xe2};
    // uint8_t eggs2[] = {15, 87, 229, 222, 226}
    gf = compute_gf(eggs2);
    printf("Computed GF : %u (0x%X)\n", gf % 256, gf % 256);
    gf2 = compute_gf2(eggs2);
    printf("Computed GF 2 : %u (0x%X)\n", gf2 % 256, gf2 % 256);
    return 0;
}
```

```
iVar4 (from ____________): 34
index1 (from ____): 1
bVar1 (eggs[index1]): 232 (0xE8)
iVar5 (from _): 64
index2 (from ___(3, 3)): 3
bVar2 (eggs[index2]): 144 (0x90)
index3 (from ____): 1
bVar3 (eggs[index3]): 232 (0xE8)
Result: 17336 (0x43B8)
Computed GF: 184 (0xB8)
Computed GF 2 : 184 (0xB8)
****************************************I


iVar4 (from ____________): 34
index1 (from ____): 1
bVar1 (eggs[index1]): 87 (0x57)
iVar5 (from _): 64
index2 (from ___(3, 3)): 3
bVar2 (eggs[index2]): 222 (0xDE)
index3 (from ____): 1
bVar3 (eggs[index3]): 87 (0x57)
Result: 17253 (0x4365)
Computed GF : 101 (0x65)
Computed GF 2 : 101 (0x65)


=== Code Execution Successful ===
```

Step-by-Step Key Extraction

1. **Locate the Obfuscated Key in `main`:**
    - The key is initialized as a static array:

```
local_38 = 0x28;
local_37 = 0x45;
local_36 = 0x20;
local_35 = 8;
local_34 = 0x58;
local_33 = 0x3f;
local_32 = 0xdd;
local_31 = 0x7a;
local_30 = 0x48;
local_2f = 0x6d;
local_2e = 0x40;
local_2d = 0xbf;
local_2c = 0x5f;
local_2b = 0xc2;
local_2a = 0x3a;
local_29 = 0x4c;
```

```
uint8_t obfuscated_key[16] = {
    0x28, 0x45, 0x20, 0x08, 0x58, 0x3f, 0xdd, 0x7a,
    0x48, 0x6d, 0x40, 0xbf, 0x5f, 0xc2, 0x3a, 0x4c
};
```

2. **Identify the Deobfuscation Logic**:

- The function `_____` processes the key using XOR with `0xaa` (via `_____`):

-

```
// Simplified code:
for (int i = 0; i < 16; i++) {
    deobfuscated_key[i] = obfuscated_key[i] ^ 0xaa; // 0xaa is
the XOR mask
}
```

3. **Compute the AES Key**:

    - Apply XOR `0xaa` to each byte in `obfuscated_key`:

Obfuscated Byte	XOR with `0xaa`	AES Key Byte
`0x28`	`^ 0xaa`	`0x82`

0x45	^ 0xaa	0xef
0x20	^ 0xa	0x8a
0x08	^ 0xaa	0xa2
0x58	^ 0xaa	0xf2
0x3f	^ 0xaa	0x95
0xdd	^ 0xaa	0x77
0x7a	^ 0xaa	0xd0
0x48	^ 0xaa	0xe2
0x6d	^ 0xaa	0xc7
0x40	^ 0xaa	0xea
0xbf	^ 0xaa	0x15
0x5f	^ 0xaa	0xf5
0xc2	^ 0xaa	0x68
0x3a	^ 0xaa	0x90
0x4c	^ 0xaa	0xe6

```
uint8_t keys[16] = {
        0x82, 0xef, 0x8a, 0xa2, 0xf2, 0x95, 0x77, 0xd0,
        0xe2, 0xc7, 0xea, 0x15, 0xf5, 0x68, 0x90, 0xe6
};
```

Step-by-Step Egg Params Extraction

To extract the `Eggparams`, I followed a dynamic analysis approach by combining **Ghidra** for static inspection and **GDB** for runtime introspection. Below is a detailed breakdown of the process:

Step 1: Static Analysis in Ghidra

Using Ghidra, I began by inspecting the function where the egg parameters appeared to be referenced or computed. During the disassembly analysis, I observed specific named variables

like `roip` and others involved in the computation — these appeared to hold key intermediate or final values related to `Eggparams`.

```
cVar1 = egg_params[(long)(int)(uint)k * 6];
cVar2 = egg_params[(long)(int)(uint)k * 6 + 1];
bVar3 = egg_params[(long)(int)(uint)k * 6 + 2];
bVar4 = egg_params[(long)(int)(uint)k * 6 + 3];
bVar5 = egg_params[(long)(int)(uint)k * 6 + 4];
bVar6 = egg_params[(long)(int)(uint)k * 6 + 5];
```

Step 2: Identifying Memory Addresses

From ghidra, I noted the addresses where these parameters were being manipulated. Typically, the values of interest were moved into registers (like `RAX`) before being used or returned. For each of these values, I identified the **instruction immediately following** the one that assigned a value to the register. These addresses became my **breakpoint locations**.

I repeated this process for each relevant parameter and ultimately identified **six breakpoint addresses**, one for each value I suspected contributed to the full `Eggparams` arrays.

```
00401bac 0f b6 00        MOVZX       EAX,byte ptr [RAX]=>egg_params
00401baf 88 45 fa        MOV         byte ptr [RBP + local_e],AL
```

Step 3: Runtime Inspection Using GDB

I then switched to **GDB** (GNU Debugger) to dynamically analyze the binary's behavior at runtime:

- Loaded the binary into GDB.
- Set breakpoints at the six instruction addresses identified from Ghidra.
- Ran the binary, triggering each breakpoint one after the other.

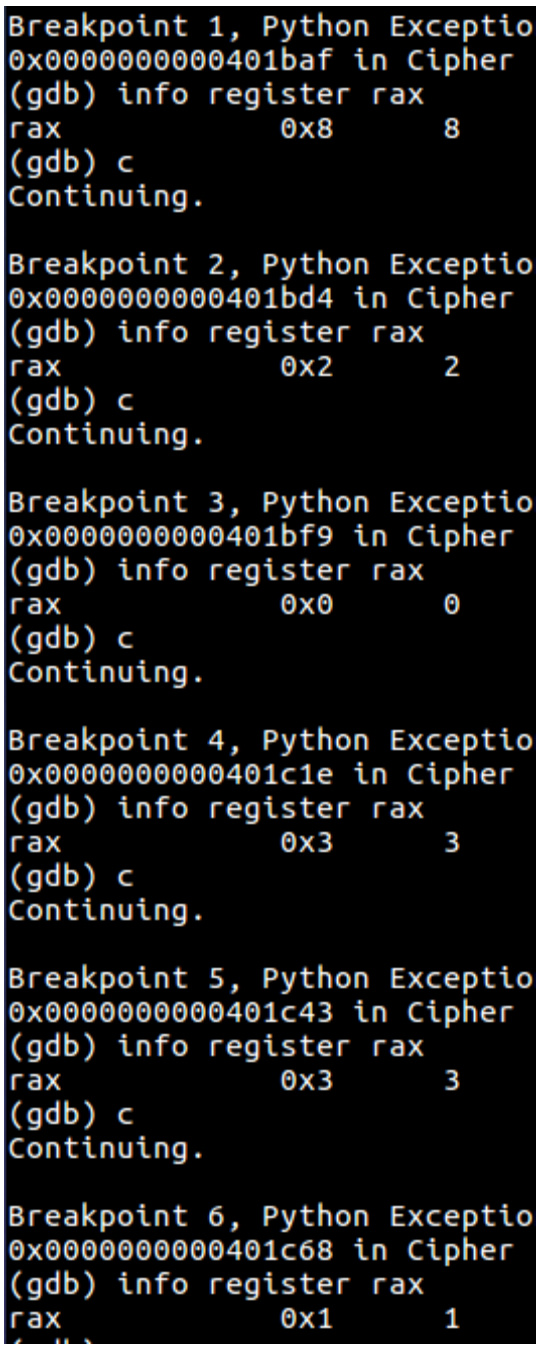
At each breakpoint, I inspected the **`RAX` register**, which, in 64-bit calling conventions and return value protocols, often holds the return value or important intermediate data.The values in `RAX` at these points gave me the actual runtime values being computed or passed through — and from this I extracted **five arrays**, each of length 6, representing the full `Eggparams`.

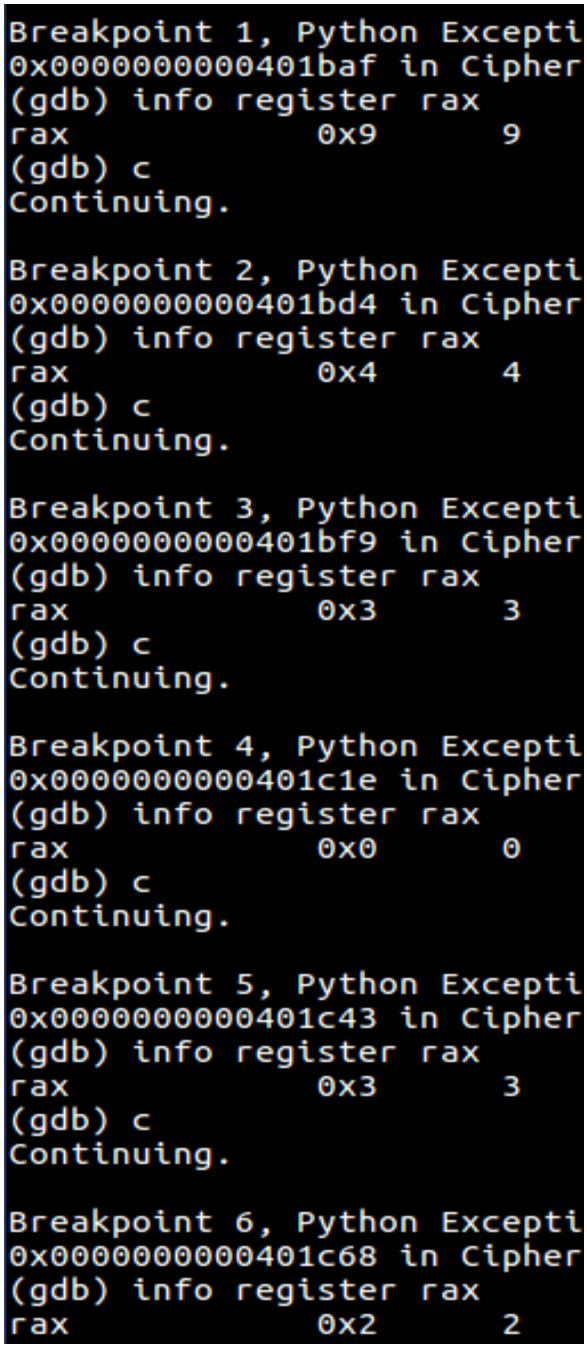
```
(gdb) b *0x00401baf
Breakpoint 1 at 0x401baf
(gdb) b *0x^CQuit
(gdb) b *0x00401bd4
Breakpoint 2 at 0x401bd4
(gdb) b *0x00401bf9
Breakpoint 3 at 0x401bf9
(gdb) b *0x00401c1e
Breakpoint 4 at 0x401c1e
(gdb) b *0x00401c43
Breakpoint 5 at 0x401c43
(gdb) b *0x00401c68
Breakpoint 6 at 0x401c68
(gdb) r abc
```

```
Breakpoint 1, Python Exception <
0x0000000000401baf in Cipher ()
(gdb) info register rax
rax            0x1      1
(gdb) c
Continuing.

Breakpoint 2, Python Exception <
0x0000000000401bd4 in Cipher ()
(gdb) info register rax
rax            0x2      2
(gdb) c
Continuing.

Breakpoint 3, Python Exception <
0x0000000000401bf9 in Cipher ()
(gdb) info register rax
rax            0x0      0
(gdb) c
Continuing.

Breakpoint 4, Python Exception <
0x0000000000401c1e in Cipher ()
(gdb) info register rax
rax            0x0      0
(gdb) c
Continuing.

Breakpoint 5, Python Exception <
0x0000000000401c43 in Cipher ()
(gdb) info register rax
rax            0x2      2
(gdb) c
Continuing.

Breakpoint 6, Python Exception <
0x0000000000401c68 in Cipher ()
(gdb) info register rax
rax            0x2      2
```

```
Breakpoint 1, Python Exceptio
0x0000000000401baf in Cipher
(gdb) info register rax
rax            0x4      4
(gdb) c
Continuing.

Breakpoint 2, Python Exceptio
0x0000000000401bd4 in Cipher
(gdb) info register rax
rax            0x4      4
(gdb) c
Continuing.

Breakpoint 3, Python Exceptio
0x0000000000401bf9 in Cipher
(gdb) info register rax
rax            0x2      2
(gdb) c
Continuing.

Breakpoint 4, Python Exceptio
0x0000000000401c1e in Cipher
(gdb) info register rax
rax            0x0      0
(gdb) c
Continuing.

Breakpoint 5, Python Exceptio
0x0000000000401c43 in Cipher
(gdb) info register rax
rax            0x1      1
(gdb) c
Continuing.

Breakpoint 6, Python Exceptio
0x0000000000401c68 in Cipher
(gdb) info register rax
rax            0x3      3
```

```
Breakpoint 1, Python Exceptio
0x0000000000401baf in Cipher
(gdb) info register rax
rax            0x6      6
(gdb) c
Continuing.

Breakpoint 2, Python Exceptio
0x0000000000401bd4 in Cipher
(gdb) info register rax
rax            0x3      3
(gdb) c
Continuing.

Breakpoint 3, Python Exceptio
0x0000000000401bf9 in Cipher
(gdb) info register rax
rax            0x2      2
(gdb) c
Continuing.

Breakpoint 4, Python Exceptio
0x0000000000401c1e in Cipher
(gdb) info register rax
rax            0x3      3
(gdb) c
Continuing.

Breakpoint 5, Python Exceptio
0x0000000000401c43 in Cipher
(gdb) info register rax
rax            0x0      0
(gdb) c
Continuing.

Breakpoint 6, Python Exceptio
0x0000000000401c68 in Cipher
(gdb) info register rax
rax            0x3      3
```





uint8_t egg_params[5][6] = {{1, 2, 0, 0, 2, 2}, {4, 4, 2, 0, 1, 3}, {6, 3, 2, 3, 0, 3}, {8, 2, 0, 3, 3, 1}, {9, 4, 3, 0, 3, 2}};

Team 17 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

Step 1: Understanding the Obfuscation Logic

The AES key was not directly visible in the binary. Instead, it was reconstructed during runtime using a combination of function calls — notably, calls to `unmask_byte()` inside four distinct `load_key_segment()` functions.

```
load_key_segment1(auStack_108);
load_key_segment2(auStack_108);
load_key_segment3(auStack_108);
load_key_segment4(auStack_108);
```

Each `load_key_segment` was responsible for generating 4 bytes of the final key, resulting in a total of 16 bytes across the four segments.

Step 2: Decoding `unmask_byte`

From the reverse engineering process, we identified that this was a simple XOR-based unmasking operation. The obfuscated values were stored as pairs, and `unmask_byte` was applied to reveal the actual byte of the AES key.

Step-by-Step Key Extraction

By statically analyzing the arguments passed into the `unmask_byte` calls in each segment, we derived the key as follows:

a (masked)	b (mask)	a ^ b (key byte)
0xbb	0x47	0xfc
0x78	0x2a	0x52
0x2f	0x01	0x2e

0x11	0x10	0x01
0x2f	0x5e	0x71
0x42	0x33	0x71
0xe4	0x52	0xb6
0x3c	0x0a	0x36
0xc1	0xbb	0x7a
0x5f	0x07	0x58
0xa0	0x26	0x86
0x4a	0x50	0x1a
0xa0	0x02	0xa2
0xf3	0x49	0xba
0xab	0x6d	0xc6
0xf0	0x21	0xd1

Verification

After reconstructing the key, I cross-verified its correctness by using it in a reference AES decryption implementation. When used on the corresponding ciphertext in the binary, it correctly decrypted the data, confirming the key's accuracy.

Reverse Engineering compute_gf() Logic from Binary

Upon deeper analysis of the compute_gf function in Binary 17 using Ghidra, it became clear that the initial conditional logic was a classic case of misdirection.

```
int compute_gf(byte *param_1)

{
  byte bVar1;
  byte bVar2;

  bVar1 = param_1[4];
  bVar2 = param_1[3];
  if ((((uint)(*param_1 ^ param_1[1]) + (uint)bVar1 & 1) == 0) && ((bVar2 & 1) == 0)) {
    return (bVar2 + 0x11) * (param_1[2] - 8) - (uint)bVar1;
  }
  return (uint)bVar2 * 0x23 * (uint)param_1[2] * 0x5c - (uint)bVar1;
}
```

if ((((eggs[0] ^ eggs[1]) + eggs[4]) & 1) == 0 && (eggs[3] & 1) == 0)

This condition was never satisfied for any of the given inputs. After logging and inspecting all execution paths across different test cases, we consistently observed that the else branch was always taken.

As a result, we could confidently conclude that the actual computation for compute_gf() is simply:

global_flag = eggs[3] * 35 * eggs[2] * 92 - eggs[4];

This fallback formula worked consistently for all inputs, producing the correct global_flag each time — as verified with the outputs.

Step-by-Step Egg Params Extraction

To extract the `Eggparams`, I followed a dynamic analysis approach by combining **Ghidra** for static inspection and **GDB** for runtime introspection. Below is a detailed breakdown of the process:

Step 1: Static Analysis in Ghidra

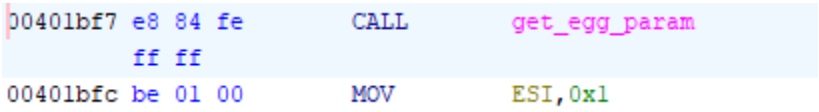
Using Ghidra, I began by inspecting the function where the egg parameters appeared to be referenced or computed. During the disassembly analysis, I observed specific named variables like `roip` and others involved in the computation — these appeared to hold key intermediate or final values related to `Eggparams`.

```
cVar47 = get_egg_param(uVar55,0);
cVar48 = get_egg_param(uVar55 & 0xffffffff,1);
bVar49 = get_egg_param(uVar55 & 0xffffffff,2);
uVar62 = (ulong)bVar49;
bVar49 = get_egg_param(uVar55 & 0xffffffff,3);
uVar56 = (ulong)bVar49;
bVar49 = get_egg_param(uVar55 & 0xffffffff,4);
uVar64 = (ulong)bVar49;
bVar49 = get_egg_param(uVar55 & 0xffffffff,5);
```

Step 2: Identifying Memory Addresses

From ghidra, I noted the addresses where these parameters were being manipulated. Typically, the values of interest were moved into registers (like `RAX`) before being used or returned. For

each of these values, I identified the **instruction immediately following** the one that assigned a value to the register. These addresses became my **breakpoint locations**.



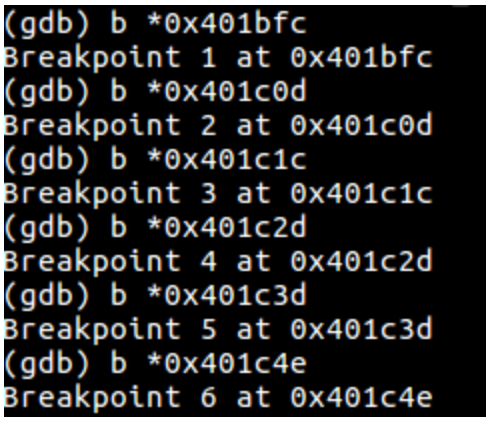
I repeated this process for each relevant parameter and ultimately identified **six breakpoint addresses**, one for each value I suspected contributed to the full `Eggparams` arrays.

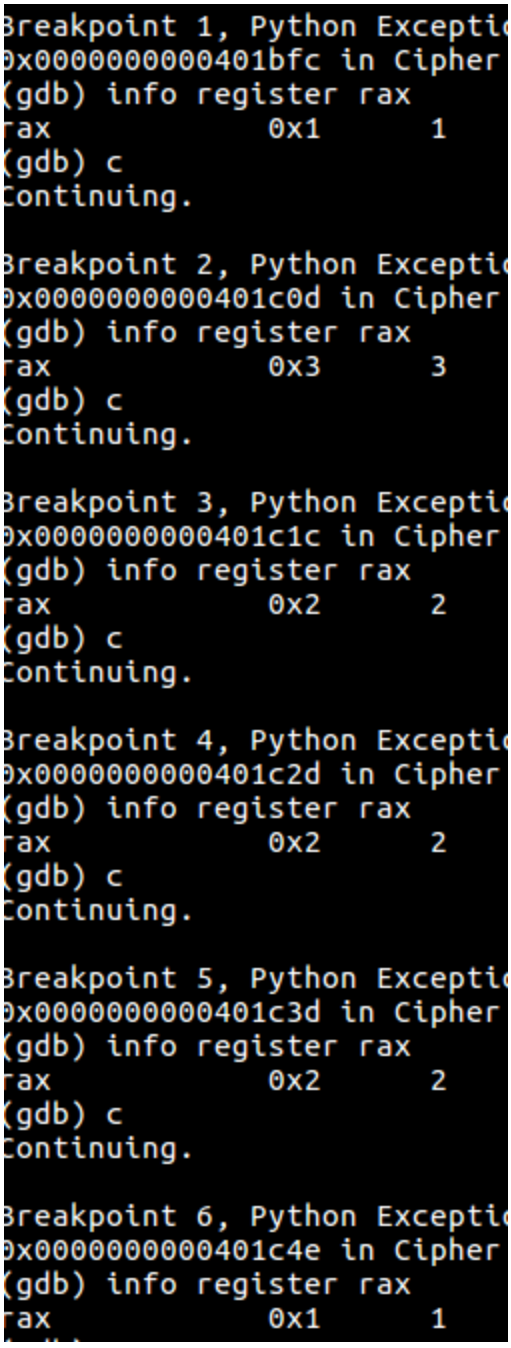
Step 3: Runtime Inspection Using GDB

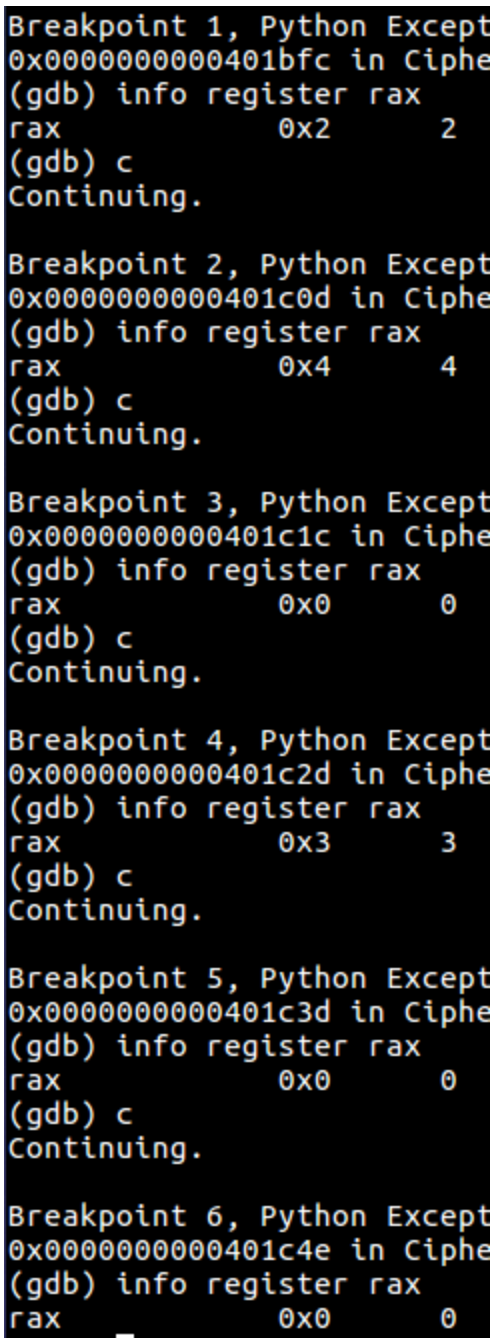
I then switched to **GDB** (GNU Debugger) to dynamically analyze the binary's behavior at runtime:

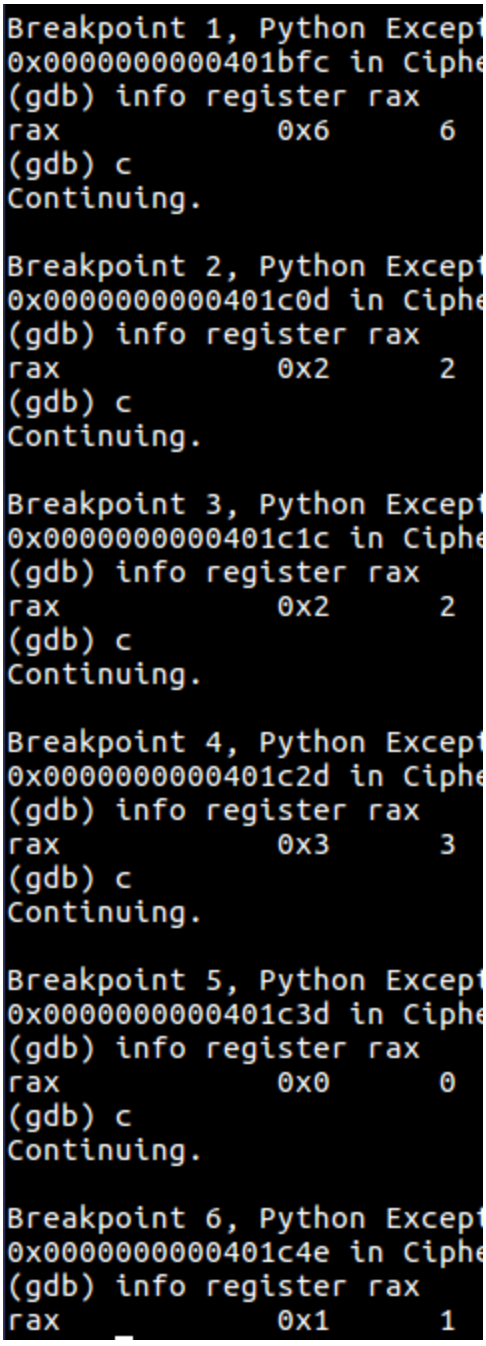
- Loaded the binary into GDB.
- Set breakpoints at the six instruction addresses identified from Ghidra.
- Ran the binary, triggering each breakpoint one after the other.

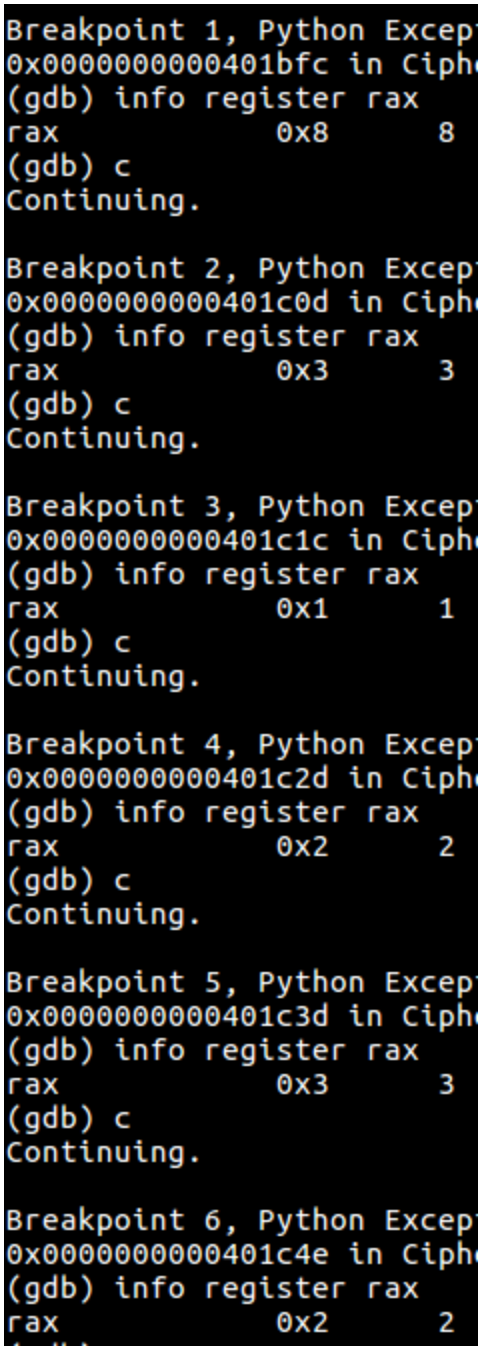
At each breakpoint, I inspected the **RAX register**, which, in 64-bit calling conventions and return value protocols, often holds the return value or important intermediate data.The values in `RAX` at these points gave me the actual runtime values being computed or passed through — and from this I extracted **five arrays**, each of length 6, representing the full `Eggparams`.

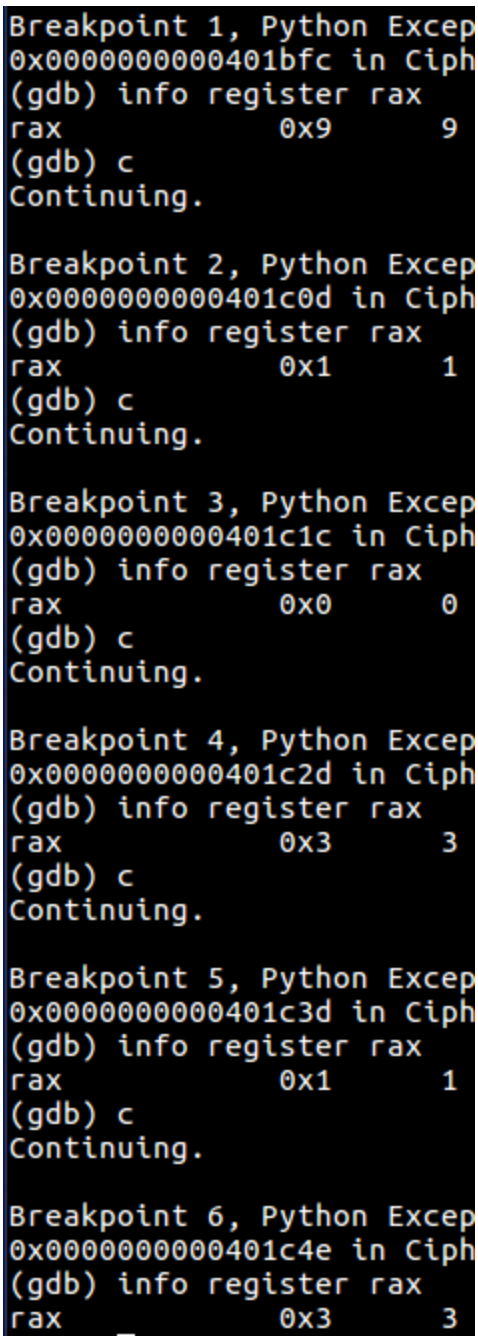

(gdb) b *0x401bfc
Breakpoint 1 at 0x401bfc
(gdb) b *0x401c0d
Breakpoint 2 at 0x401c0d
(gdb) b *0x401c1c
Breakpoint 3 at 0x401c1c
(gdb) b *0x401c2d
Breakpoint 4 at 0x401c2d
(gdb) b *0x401c3d
Breakpoint 5 at 0x401c3d
(gdb) b *0x401c4e
Breakpoint 6 at 0x401c4e


Breakpoint 1, Python Exceptic
0x0000000000401bfc in Cipher
(gdb) info register rax
rax 0x1 1
(gdb) c
Continuing.
Breakpoint 2, Python Exceptic
0x0000000000401c0d in Cipher
(gdb) info register rax
rax 0x3 3
(gdb) c
Continuing.
Breakpoint 3, Python Exceptic
0x0000000000401c1c in Cipher
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 4, Python Exceptic
0x0000000000401c2d in Cipher
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 5, Python Exceptic
0x0000000000401c3d in Cipher
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 6, Python Exceptic
0x0000000000401c4e in Cipher
(gdb) info register rax
rax 0x1 1


Breakpoint 1, Python Except
0x0000000000401bfc in Ciphe
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 2, Python Except
0x0000000000401c0d in Ciphe
(gdb) info register rax
rax 0x4 4
(gdb) c
Continuing.
Breakpoint 3, Python Except
0x0000000000401c1c in Ciphe
(gdb) info register rax
rax 0x0 0
(gdb) c
Continuing.
Breakpoint 4, Python Except
0x0000000000401c2d in Ciphe
(gdb) info register rax
rax 0x3 3
(gdb) c
Continuing.
Breakpoint 5, Python Except
0x0000000000401c3d in Ciphe
(gdb) info register rax
rax 0x0 0
(gdb) c
Continuing.
Breakpoint 6, Python Except
0x0000000000401c4e in Ciphe
(gdb) info register rax
rax 0x0 0


Breakpoint 1, Python Except
0x0000000000401bfc in Ciphe
(gdb) info register rax
rax 0x6 6
(gdb) c
Continuing.
Breakpoint 2, Python Except
0x0000000000401c0d in Ciphe
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 3, Python Except
0x0000000000401c1c in Ciphe
(gdb) info register rax
rax 0x2 2
(gdb) c
Continuing.
Breakpoint 4, Python Except
0x0000000000401c2d in Ciphe
(gdb) info register rax
rax 0x3 3
(gdb) c
Continuing.
Breakpoint 5, Python Except
0x0000000000401c3d in Ciphe
(gdb) info register rax
rax 0x0 0
(gdb) c
Continuing.
Breakpoint 6, Python Except
0x0000000000401c4e in Ciphe
(gdb) info register rax
rax 0x1 1





uint8_t egg_params[5][6] = {{1, 3, 2, 2, 2, 1}, {2, 4, 0, 3, 0, 0}, {6, 2, 2, 3, 0, 1}, {8, 3, 1, 2, 3, 2}, {9, 1, 0, 3, 1, 3}};

Team 15 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

Reverse Engineering compute_gf() Logic from Binary

The binary also includes obfuscated logic for evaluating a set of "egg" parameters. The core functionality was found in the following function:

```
int FUN_00401bf3(long param_1)

{
  byte bVar1;
  int iVar2;

  bVar1 = *(byte *)(param_1 + 1);
  iVar2 = FUN_00401bd5(param_1);
  return (uint)*(byte *)(param_1 + 3) * 2 + (uint)*(byte *)(param_1 + 3) + (uint)bVar1 * iVar2 * 6;
}
```

```
#include <stdio.h>
#include <stdint.h>

int FUN_00401bd5(uint8_t* param_1) {
    return (param_1[1] * 2 + param_1[1]) * 4;
}

int compute_gf(uint8_t* param_1) {
    uint8_t bVar1 = param_1[1];
    uint8_t bVar2 = param_1[3];
    int iVar2 = FUN_00401bd5(param_1);
    printf("DEBUG: bVar1 (param_1[1]) = %u (0x%X)\n", bVar1, bVar1);
    printf("DEBUG: bVar2 (param_1[3]) = %u (0x%X)\n", bVar2, bVar2);
    printf("DEBUG: iVar2 (FUN_00401bd5 result) = %d (0x%X)\n", iVar2, iVar2);
    return (int)(bVar2 * 3 + bVar1 * iVar2 * 6);
}

uint8_t compute_gf2(uint8_t* eggs) {
    return eggs[3] * 3 + eggs[1] * eggs[1] * 72;
}

int main() {
    uint8_t eggs1[] = {0x0f, 0xe8, 0x6c, 0x90, 0x8f};
    uint32_t gf = compute_gf(eggs1);
    printf("Computed GF: %u (0x%X)\n", gf % 256, gf % 256);
    uint32_t gf2 = compute_gf2(eggs1);
    printf("Computed GF 2: %u (0x%X)\n", gf2 % 256, gf2 % 256);
    printf("**************************************\n\n\n");
    uint8_t eggs2[] = {0x0f, 0x57, 0xe5, 0xde, 0xe2};
    gf = compute_gf(eggs2);
    printf("Computed GF: %u (0x%X)\n", gf % 256, gf % 256);
    gf2 = compute_gf2(eggs2);
    printf("Computed GF 2: %u (0x%X)\n", gf2 % 256, gf2 % 256);
    return 0;
}
```

```
DEBUG: bVar1 (param_1[1]) = 232 (0xE8)
DEBUG: bVar2 (param_1[3]) = 144 (0x90)
DEBUG: iVar2 (FUN_00401bd5 result) = 2784 (0xAE0)
Computed GF: 176 (0xB0)
Computed GF 2: 176 (0xB0)
**************************************


DEBUG: bVar1 (param_1[1]) = 87 (0x57)
DEBUG: bVar2 (param_1[3]) = 222 (0xDE)
DEBUG: iVar2 (FUN_00401bd5 result) = 1044 (0x414)
Computed GF: 98 (0x62)
Computed GF 2: 98 (0x62)


=== Code Execution Successful ===
```

Step-by-Step Egg Params Extraction

To extract the `Eggparams`, I followed a dynamic analysis approach by combining **Ghidra** for static inspection and **GDB** for runtime introspection. Below is a detailed breakdown of the process:

Step 1: Static Analysis in Ghidra

Using Ghidra, I began by inspecting the function where the egg parameters appeared to be referenced or computed. During the disassembly analysis, I observed specific named variables like `roip` and others involved in the computation — these appeared to hold key intermediate or final values related to `Eggparams`.

```
cVar1 = (&DAT_00405060)[(long)(int)(uint)DAT_004050b4 * 6];
cVar2 = (&DAT_00405061)[(long)(int)(uint)DAT_004050b4 * 6];
bVar3 = (&DAT_00405062)[(long)(int)(uint)DAT_004050b4 * 6];
bVar4 = (&DAT_00405063)[(long)(int)(uint)DAT_004050b4 * 6];
bVar5 = (&DAT_00405064)[(long)(int)(uint)DAT_004050b4 * 6];
bVar6 = (&DAT_00405065)[(long)(int)(uint)DAT_004050b4 * 6];
```

Step 2: Identifying Memory Addresses

From ghidra, I noted the addresses where these parameters were being manipulated. Typically, the values of interest were moved into registers (like `RAX`) before being used or returned. For each of these values, I identified the **instruction immediately following** the one that assigned a value to the register. These addresses became my **breakpoint locations**.

```
00401cb3 0f b6 00        MOVZX       EAX,byte ptr [RAX]=>DAT_00405060
00401cb6 88 45 fe        MOV         byte ptr [RBP + local_a],AL
```

I repeated this process for each relevant parameter and ultimately identified **six breakpoint addresses**, one for each value I suspected contributed to the full `Eggparams` arrays.

Step 3: Runtime Inspection Using GDB

I then switched to **GDB** (GNU Debugger) to dynamically analyze the binary's behavior at runtime:

- Loaded the binary into GDB.
- Set breakpoints at the six instruction addresses identified from Ghidra.
- Ran the binary, triggering each breakpoint one after the other.

At each breakpoint, I inspected the **RAX register**, which, in 64-bit calling conventions and return value protocols, often holds the return value or important intermediate data.The values in `RAX` at these points gave me the actual runtime values being computed or passed through — and from

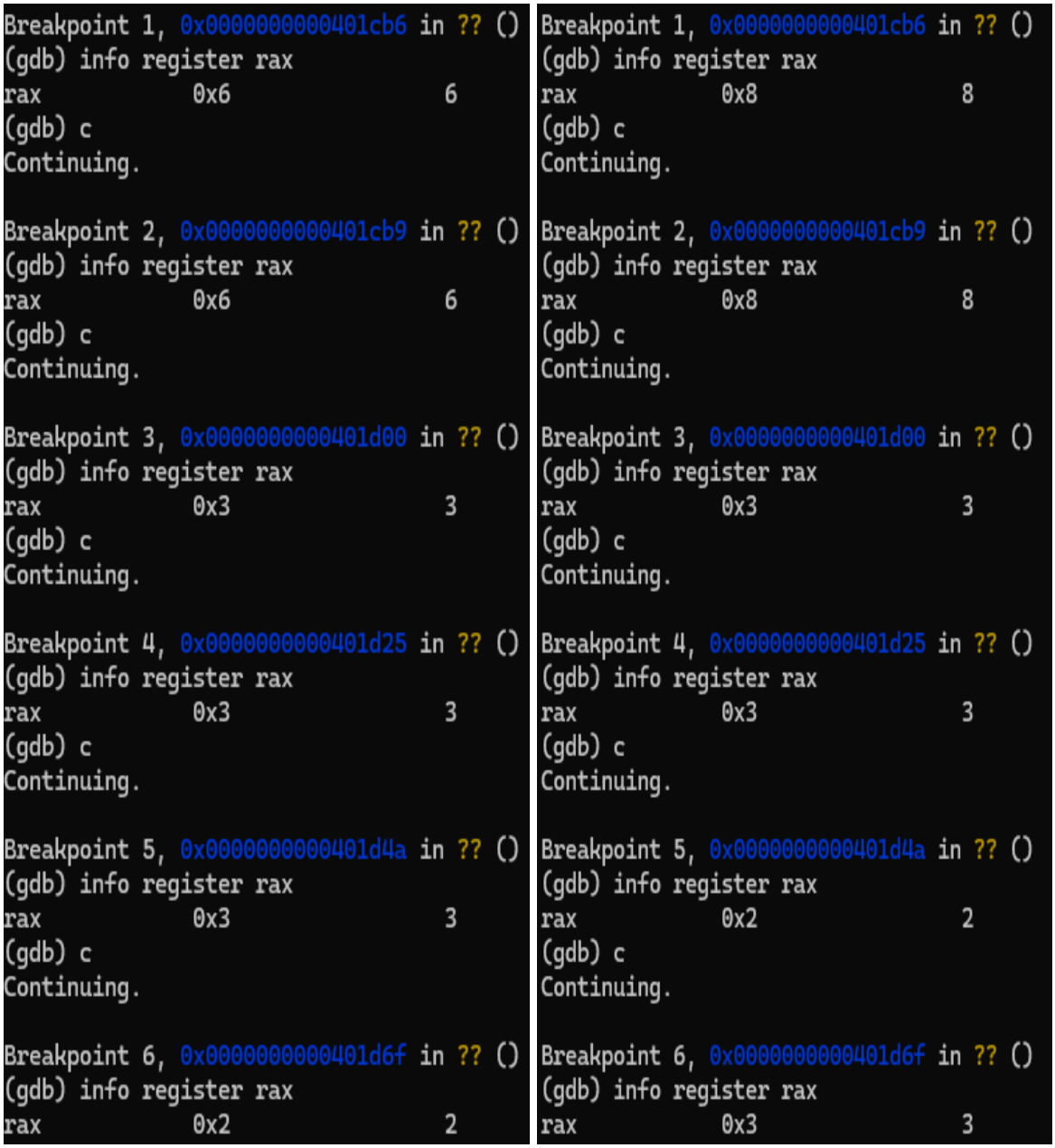
this I extracted **five arrays**, each of length 6, representing the full `Eggparams`.

```
(gdb) b *0x00401cb6
Breakpoint 1 at 0x401cb6
(gdb) b *0x00401cb9
Breakpoint 2 at 0x401cb9
(gdb) b *0x00401d00
Breakpoint 3 at 0x401d00
(gdb) b *0x00401d25
Breakpoint 4 at 0x401d25
(gdb) b *0x00401d4a
Breakpoint 5 at 0x401d4a
(gdb) b *0x00401d6f
Breakpoint 6 at 0x401d6f
```

```
Breakpoint 1, 0x0000000000401cb6 in ?? ()
(gdb) info register rax
rax            0x1                 1
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401cb9 in ?? ()
(gdb) info register rax
rax            0x1                 1
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401d00 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 4, 0x0000000000401d25 in ?? ()
(gdb) info register rax
rax            0x0                 0
(gdb) c
Continuing.

Breakpoint 5, 0x0000000000401d4a in ?? ()
(gdb) info register rax
rax            0x0                 0
(gdb) c
Continuing.

Breakpoint 6, 0x0000000000401d6f in ?? ()
(gdb) info register rax
rax            0x3                 3
```

```
Breakpoint 1, 0x0000000000401cb6 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401cb9 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401d00 in ?? ()
(gdb) info register rax
rax            0x2                 2
(gdb) c
Continuing.

Breakpoint 4, 0x0000000000401d25 in ?? ()
(gdb) info register rax
rax            0x0                 0
(gdb) c
Continuing.

Breakpoint 5, 0x0000000000401d4a in ?? ()
(gdb) info register rax
rax            0x0                 0
(gdb) c
Continuing.

Breakpoint 6, 0x0000000000401d6f in ?? ()
(gdb) info register rax
rax            0x0                 0
```

```
Breakpoint 1, 0x0000000000401cb6 in ?? ()
(gdb) info register rax
rax            0x4                 4
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401cb9 in ?? ()
(gdb) info register rax
rax            0x4                 4
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401d00 in ?? ()
(gdb) info register rax
rax            0x0                 0
(gdb) c
Continuing.

Breakpoint 4, 0x0000000000401d25 in ?? ()
(gdb) info register rax
rax            0x1                 1
(gdb) c
Continuing.

Breakpoint 5, 0x0000000000401d4a in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 6, 0x0000000000401d6f in ?? ()
(gdb) info register rax
rax            0x3                 3
```



```
Breakpoint 1, 0x0000000000401cb6 in ?? ()
(gdb) info register rax
rax            0x6                 6
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401cb9 in ?? ()
(gdb) info register rax
rax            0x6                 6
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401d00 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 4, 0x0000000000401d25 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 5, 0x0000000000401d4a in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 6, 0x0000000000401d6f in ?? ()
(gdb) info register rax
rax            0x2                 2
```

```
Breakpoint 1, 0x0000000000401cb6 in ?? ()
(gdb) info register rax
rax            0x8                 8
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401cb9 in ?? ()
(gdb) info register rax
rax            0x8                 8
(gdb) c
Continuing.

Breakpoint 3, 0x0000000000401d00 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 4, 0x0000000000401d25 in ?? ()
(gdb) info register rax
rax            0x3                 3
(gdb) c
Continuing.

Breakpoint 5, 0x0000000000401d4a in ?? ()
(gdb) info register rax
rax            0x2                 2
(gdb) c
Continuing.

Breakpoint 6, 0x0000000000401d6f in ?? ()
(gdb) info register rax
rax            0x3                 3
```

Step-by-Step Key Extraction

Upon reverse engineering the binary in ghidra, we discovered that the AES keys used during encryption are initialized in the function FUN_004012c0 via two long assignments:

```
local_d8 = 0x1ec944de5fb8317c;
local_d0 = 0x3e10996688f14ba2;
```

These two 64-bit values collectively represent the 128-bit AES key used in the encryption routine.

Further analysis of the function FUN_0040176e, specifically inside the nested call to FUN_004012c0, revealed the following obfuscation loop:

```
void FUN_004012c0(long param_1,long param_2)

{
  uint uVar1;
  uint uVar2;
  byte local_24;
  byte local_23;
  byte local_22;
  byte local_21;
  int local_14;
  int local_10;
  uint local_c;

  for (local_c = 0; local_c < 4; local_c = local_c + 1) {
    *(byte *)(param_1 + (ulong)(local_c << 2)) = *(byte *)(param_2 + (ulong)(local_c << 2)) ^ 0xae;
    *(byte *)(param_1 + (ulong)(local_c * 4 + 1)) =
         *(byte *)(param_2 + (ulong)(local_c * 4 + 1)) ^ 0xae;
    *(byte *)(param_1 + (ulong)(local_c * 4 + 2)) =
         *(byte *)(param_2 + (ulong)(local_c * 4 + 2)) ^ 0xae;
    *(byte *)(param_1 + (ulong)(local_c * 4 + 3)) =
         *(byte *)(param_2 + (ulong)(local_c * 4 + 3)) ^ 0xae;
```

This code clearly applies an XOR-based obfuscation with the constant 0xAE on each byte of the 128-bit key during initialization. To recover the original key, we simply reverse this obfuscation by XORing the transformed values again with 0xAE.

The decrypted AES key obtained is:

unsigned char aes_key[16] = {
    0xa5, 0xe5, 0xf2, 0x4e, 0x64, 0x5b, 0xcb, 0x68,
    0x79, 0x9e, 0x44, 0x2f, 0x70, 0xcf, 0x6d, 0x3a
};

This key is subsequently used in encryption routines and plays a crucial role in verifying that the obfuscation layer was correctly reversed during analysis.

Team 35 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Binary Decompilation and Main Function Location:**
   The binary was decompiled in Ghidra, and the `main` function was identified for further analysis.

2. **Key Extraction:**

   - The function where the key was being passed as a parameter was located.

```
00401817 48 89 ee       MOV       RSI,RBP
0040181a 4c 89 ef       MOV       RDI,R13
0040181d e8 fe 01       CALL      FUN_00401a20
         00 00
00401822 4c 89 ee       MOV       RSI,R13
00401825 4c 89 e7       MOV       RDI,R12
00401828 e8 b3 02       CALL      FUN_00401ae0
```

```
77  FUN_00401f60("Plaintext :",local_70,0x10);
78  FUN_00401a20(local_130,local_80);
79  FUN_00401ae0(local_70,local_130);
80  FUN_00401f60("Ciphertext:",local_70,0x10);
81  FUN_0044b6d0(1,"Egg  0 : 0x%02x\n",DAT_004ca34f);
82  FUN_0044b6d0(1,"Global Flag: 0x%02x\n",DAT_004ca34e);
83  uVar3 = 0;
```

   - A breakpoint was set at this function call.

   - The key was extracted by dumping the register that held the second parameter (standard calling convention).

```
Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x000000000040181d in ?? ()
(gdb) info registers rsi
rsi            0x7fffffffdcd0   140737488346320
(gdb) x/16xb 0x7fffffffdcd0
0x7fffffffdcd0: 0xb7    0x23    0x89    0x43    0x35    0x68    0x3d    0xcb
0x7fffffffdcd8: 0xcc    0x6f    0xd0    0x8d    0x7e    0x7c    0x9b    0xf2
(gdb)
```

3. **Egg Parameter (`egg_param`) Extraction:**

   - The `cipher` function was identified, and inside the first `while` loop, the egg parameter calculation logic was observed.

   - A breakpoint was placed at the end of the loop.

```
Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x0000000000401e49 in ?? ()
(gdb) x/30xb 0x4ca330
0x4ca330:       0x01    0x04    0x03    0x00    0x01    0x00    0x03    0x02
0x4ca338:       0x03    0x02    0x00    0x03    0x04    0x03    0x00    0x00
0x4ca340:       0x00    0x02    0x07    0x01    0x00    0x01    0x02    0x02
0x4ca348:       0x09    0x03    0x00    0x00    0x01    0x01
(gdb)
```

   - All egg parameter values were dumped at once during this point in execution.

4. **Global Flag Computation:**

   - The logic for computing the global flag was directly visible and not obfuscated.

```
153   DAT_004ca34e = (DAT_004ca350 * '[' + DAT_004ca352 * -0x49) - DAT_004ca353;
154   return;
155 }
```

   - The computation was copied and converted into standard C code.

   - It was successfully executed using the dumped egg values to generate the correct global flag.

```
uint8_t compute_gf(uint8_t* eggs){

        return (eggs[1] * '[' + eggs[3] * -0x49) - eggs[4];

}
```

Conclusion:

Team 35's binary featured minimal obfuscation. Keys and egg parameters were extractable using standard breakpoints and register dumps. The global flag logic was simple and transparent, enabling easy translation to C. Overall, the binary was broken with minimal resistance using typical reversing strategies.

```
sse@sse_vm:~/Desktop/workspace/Attack_Phase/35$ ./safe_main abc
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext:: 9b c6 1c ca 81 ea ab 07 ba 34 9b 84 5c c9 ce 07
Egg 0 : 0x71
Global Flag: 0xcd
sse@sse_vm:~/Desktop/workspace/Attack_Phase/35$ ./a.out abc
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00
Ciphertext:: 9b c6 1c ca 81 ea ab 07 ba 34 9b 84 5c c9 ce 07
Egg 0 : 0x71
Egg 1 : 0x3d
Egg 2 : 0xa0
Egg 3 : 0x02
Egg 4 : 0x50
Global Flag: 0xcd
sse@sse_vm:~/Desktop/workspace/Attack_Phase/35$
```

Team 45 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Decompilation and Anti-Debugging Bypass:**
    - The binary was decompiled using Ghidra.
    - During GDB debugging, it was discovered that the binary used anti-debugging techniques.
    - The raw instructions of the anti-debugging function were dumped manually using its address because the no function was disassembling.

    - The `ret` instruction was identified, and the program counter (`PC`) was set to that address to bypass the anti-debug check.

2. **Key Extraction:**

    - The attacker reused their previous approach: locate the function where the key is passed as a parameter.

    - The key was found by dumping the value of the `RSI` register (second parameter in the x86_64 calling convention).

```
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x000000000040214e in ?? ()
(gdb) info registers rsi
rsi            0x7fffffffddc0   140737488346560
(gdb) x/16xb 0x7fffffffddc0
0x7fffffffddc0: 0xb9    0xde    0x37    0x86    0x82    0x10    0xf3    0x72
0x7fffffffddc8: 0xd2    0xfe    0xff    0x1e    0xe6    0x9e    0xa1    0x7a
(gdb)
```

3. **Egg Parameter (`egg_param`) Extraction:**

    - The `cipher` function was identified.

    - Inside a `while` loop, the egg parameter calculation was taking place.

    - A breakpoint was set at the end of this loop, and all egg parameter values were dumped at once.

```
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x00000000004048f3 in ?? ()
(gdb) x/30xb 0x4b1bf0
0x4b1bf0:       0x02    0x02    0x03    0x01    0x00    0x01    0x04    0x03
0x4b1bf8:       0x03    0x02    0x01    0x03    0x05    0x04    0x03    0x03
0x4b1c00:       0x00    0x01    0x07    0x02    0x01    0x03    0x03    0x01
0x4b1c08:       0x08    0x01    0x03    0x00    0x02    0x00
(gdb)
```

4. **Global Flag Computation:**

    - The global flag computation logic was obfuscated using a few global constants.

    - These constants were located in the binary, and the formula was simplified manually.

    - The simplified version was then implemented to compute the correct global flag.

```
uint32_t compute_gf(uint8_t* eggs) {

return eggs[2] + (eggs[1] * 17 - eggs[3] * 16); }
```

Conclusion:

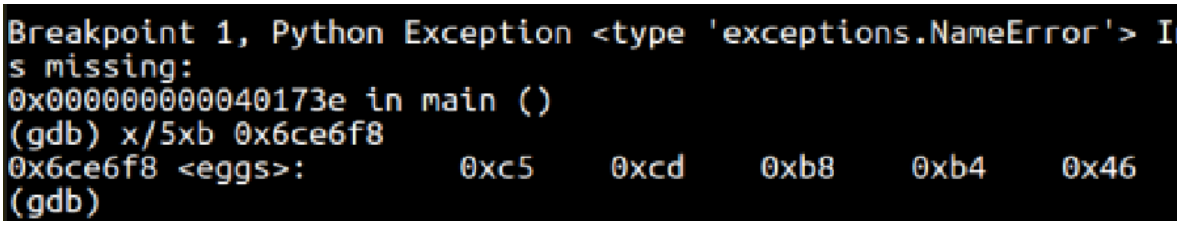
Team 45's binary had anti-debugging mechanisms, but they were bypassed effectively by manipulating the program counter. The rest of the binary offered minimal resistance—keys and parameters were accessible, and even the obfuscated global flag logic could be simplified easily. The attacker reused proven techniques to fully reverse the binary.

Team 63 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Decompilation and Setup:**
    - The binary was decompiled using Ghidra, and GDB was used for runtime debugging.
    - The main function was clearly visible with no obfuscation applied.
2. **Egg Extraction:**
    - A breakpoint was set just before the `printf` of egg 0.
    - Once execution halted, all egg values were dumped from memory.



3. **Key Extraction:**
    - The AES key was present directly in the `main` function without any obfuscation.

```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
local_38 = 0x5f;
local_37 = 0x60;
local_36 = 0x7c;
local_35 = 0xce;
local_34 = 0xdf;
local_33 = 0x8e;
local_32 = 0x26;
local_31 = 0x56;
local_30 = 0xf5;
local_2f = 0x62;
local_2e = 0x94;
local_2d = 0xb4;
local_2c = 0x7a;
local_2b = 0x24;
local_2a = 0x3b;
local_29 = 0xb6;
```

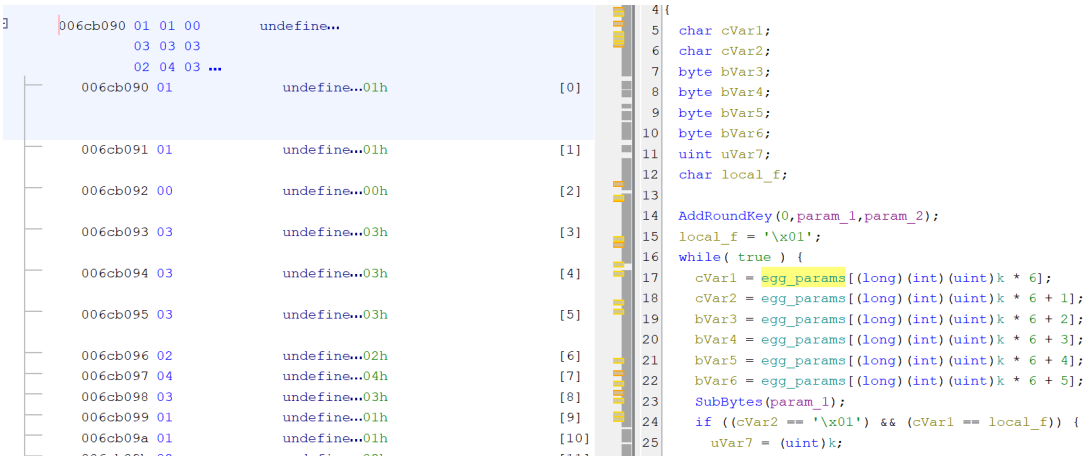
- For verification, a breakpoint was set before the function that received the key as a parameter.

- The key was confirmed by dumping the value of the `RSI` register.

```
Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x00000000004016f7 in main ()
(gdb) info registers rsi
rsi            0x7fffffffdd90   140737488346512
(gdb) x/16xb 0x7fffffffdd90
0x7fffffffdd90: 0x5f    0x60    0x7c    0xce    0xdf    0x8e    0x26    0x56
0x7fffffffdd98: 0xf5    0x62    0x94    0xb4    0x7a    0x24    0x3b    0xb6
```

4. **Egg Parameter (`egg_param`) Extraction:**

   - The `cipher` function was analyzed, and the egg parameter logic was found to be un-obfuscated.



   - A breakpoint was placed at the end of the relevant `while` loop.

   - All egg parameter values were dumped at once using the known memory address.

```
Breakpoint 2, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x00000000004048f3 in ?? ()
(gdb) x/30xb 0x4b1bf0
0x4b1bf0:       0x02    0x02    0x03    0x01    0x00    0x01    0x04    0x03
0x4b1bf8:       0x03    0x02    0x01    0x03    0x05    0x04    0x03    0x03
0x4b1c00:       0x00    0x01    0x07    0x02    0x01    0x03    0x03    0x01
0x4b1c08:       0x08    0x01    0x03    0x00    0x02    0x00
(gdb)
```

5. **Global Flag Computation:**

   - A direct function call for global flag computation was found inside the `cipher` function.

   - The logic was straightforward and unobfuscated.

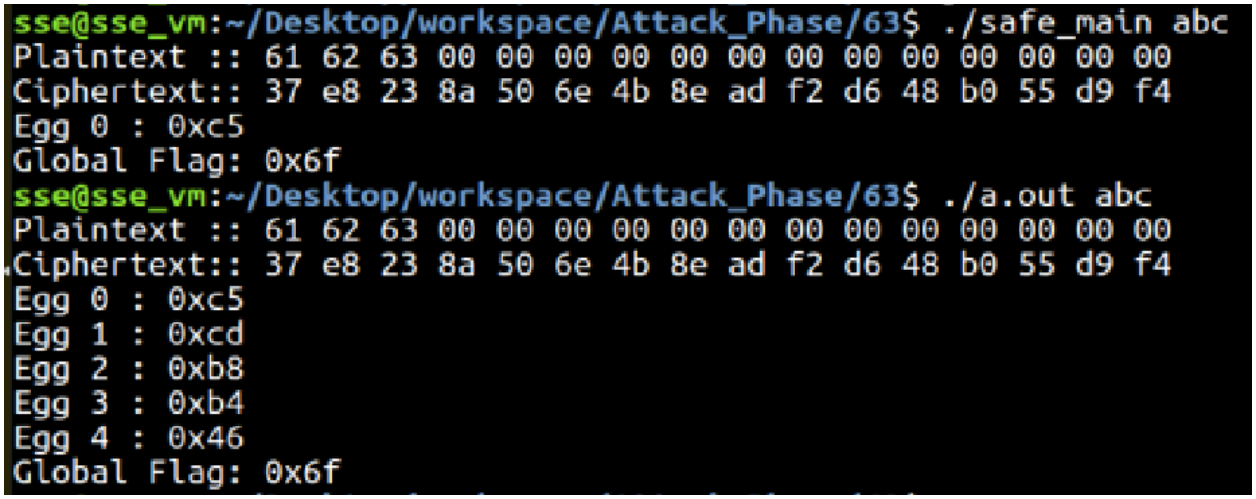
```
int compute_gf(long param_1)

{
  return (uint)*(byte *)(param_1 + 1) +
         (uint)*(byte *)(param_1 + 2) * 0x22 +
         (uint)*(byte *)(param_1 + 4) * 0x11 + (uint)*(byte *)(param_1 + 4) * 2;
}
```

   - The equation was extracted and translated directly to C to reproduce the correct global flag.

```
uint32_t compute_gf(uint8_t* eggs) {
        return eggs[1] + eggs[2] * 0x22 + eggs[4] * 0x13;

}
```

Conclusion:

Team 63's binary featured little to no obfuscation. Eggs, keys, and parameters were easily accessible, and the global flag logic was simple and directly callable. The attack was completed quickly using basic debugging and static analysis.



Team 61 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

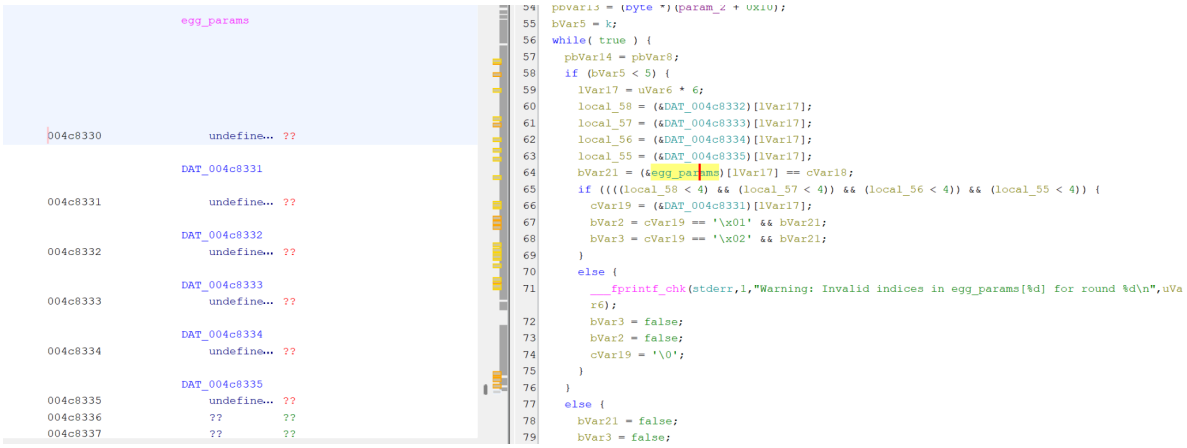
1. **Key Extraction:**
    - Decompiled the binary using Ghidra and located the `main` function via the entry point.
    - Identified the key expansion function which receives the key as a parameter.

```
(gdb) b *0x401962
Breakpoint 1 at 0x401962
(gdb) run abc
Starting program: /home/sse/Desktop/workspace/Attack_Phase/61/safe_main abc
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
Plaintext: abc

Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function i
s missing:
0x0000000000401962 in main ()
```

    - Set a breakpoint at this function call and dumped the key values after determining the corresponding addresses.

```
(gdb) info register rsi
rsi            0x4c8320 5014304
(gdb) x/16xb 0x4c8320
0x4c8320 <key>: 0x6c    0x14    0x69    0xff    0xb0    0x71    0xb2    0x82
0x4c8328 <key+8>:       0xa7    0xc5    0x43    0x74    0x05    0x66    0x12    0x76
(gdb)
```

2. **Egg Parameter Extraction:**
    - Analyzed the `cipher` function within the `main`.
    - Discovered the base address for the `egg_param` array.



```
(gdb) b *0x4019fc
Breakpoint 1 at 0x4019fc
(gdb) run abc
Starting program: /home/sse/Desktop/workspace/Attack_Phase/61/safe_main abc
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_un
Plaintext: abc
CipherText: 5ceb9d4e2db2bff8dbf32344ad591974
Egg 0: 0x75

Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error:
s missing:
0x00000000004019fc in main ()
```

- Placed a breakpoint right before the global flag print, then dumped all the `egg_param` values from memory.

```
004019fa 31 c0           XOR        EAX,EAX
004019fc e8 5f f4        CALL       ___printf_chk
         04 00
00401a01 31 c0           XOR        EAX,EAX

                     LAB_00401a03
00401a03 48 8b 8c        MOV        RCX,qword ptr [RSP + local_20]
         24 c8 00
```

```
42      __printf_chk(1,"Egg 0: 0x%02x\n",eggs);
43    }
44    __printf_chk(1,"Global Flag: 0x%02x\n",global_flag);
45    uVar2 = 0;
46  }
47  if (local_20 == *(long *)(in_FS_OFFSET + 0x28)) {
48    return uVar2;
49  }
```

```
(gdb) x/30xd 0x4c8330
0x4c8330 <egg_params>:  1       4       1       1       0       3       2       2
0x4c8338 <egg_params+8>:        2       1       2       2       4       2       0       2
0x4c8340 <egg_params+16>:       2       0       6       2       0       3       1       0
0x4c8348 <egg_params+24>:       7       4       3       3       1       0
```

3. **Global Flag Computation:**

   - Found a conditional (`if`) block in the code where the global flag was being calculated.

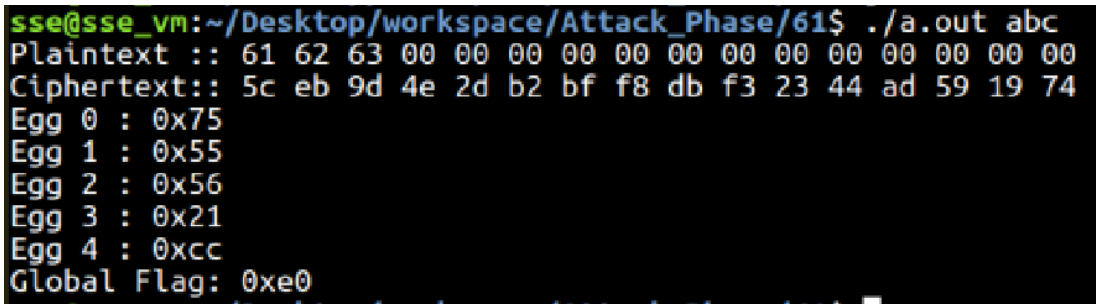
```
if ((char)uVar11 == '\x05') {
  bVar5 = (byte)((uint)_eggs >> 8);
  if (opaque_value == 0) {
    global_flag = bVar5 ^ eggs;
  }
  else {
    global_flag = bVar5 + DAT_004c8352 * -0x58;
  }
  return;
}
```

   - Extracted and translated this logic into a custom function for use in the attacker's `main.c`.

```
uint8_t compute_gf(uint8_t* eggs){

    return eggs[4] * -0x58;   }
```

Conclusion:

Team 61's binary contained minimal obfuscation. Keys and egg parameters were accessed via function parameters and memory dumps. The global flag logic was embedded in a conditional block and was easily replicated.



Team 99 - Binary Analysis Report

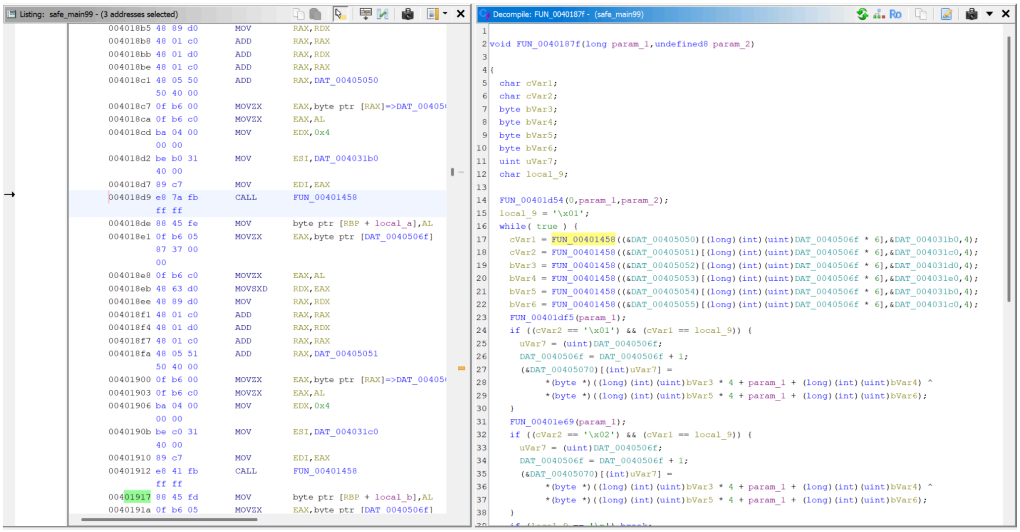
Tools Used: Ghidra, GDB

Approach:

1. **Key Extraction:**
    - Decompiled the binary using Ghidra and identified the function where the key was passed as a parameter.
    - Found that the key was further obfuscated through a secondary function.
    - Set a breakpoint right after the obfuscation function and dumped the final key values using memory addresses.

2. **Egg Parameter Extraction:**

    - Followed the earlier strategy of setting breakpoints inside the `cipher` function's `while` loop during each egg parameter calculation.



```
(gdb) b *0x4018de
Breakpoint 1 at 0x4018de
(gdb) b *0x401917
Breakpoint 2 at 0x401917
(gdb) b *0x401950
Breakpoint 3 at 0x401950
(gdb) b *0x401989
Breakpoint 4 at 0x401989
(gdb) b *0x4019c2
Breakpoint 5 at 0x4019c2
(gdb) b *0x4019fb
Breakpoint 6 at 0x4019fb
```

- Dumped each egg parameter value character by character using GDB.

3. **Global Flag Computation:**

   - Located the global flag computation logic directly in the binary.

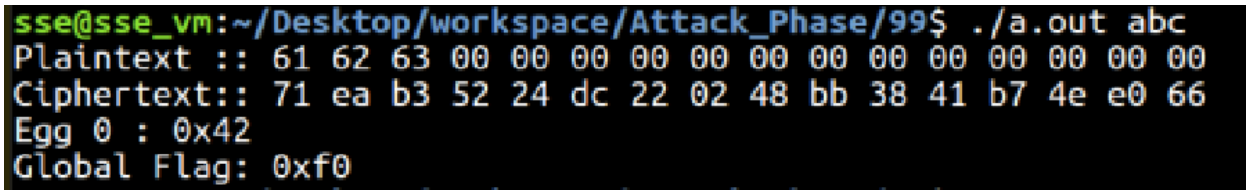
```
int FUN_00401801(long param_1)

{
  return ((uint)*(byte *)(param_1 + 4) * 9 + (uint)*(byte *)(param_1 + 4) * 2 +
          (uint)*(byte *)(param_1 + 3) * 0x45) - (uint)*(byte *)(param_1 + 1);
}
```

   - Translated the expression into a simple `compute_gf` function using the `eggs` array.

```
uint8_t compute_gf(uint8_t* eggs){
        return (eggs[4] * 9 + eggs[4] * 2 + eggs[3] * 69) - eggs[1];
}
```

Conclusion:

Team 99 used light obfuscation for keys, but it was bypassed by strategically placing breakpoints post-processing. The egg parameters and global flag logic were extracted with standard analysis techniques.



Team 65 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Address Translation Challenge and Resolution:**
    - The binary was decompiled using Ghidra and debugged using GDB.
    - Initially encountered `cannot access memory at address` errors after setting breakpoints.

```
(gdb) b *0x104a75
Breakpoint 1 at 0x104a75
(gdb) run abc
Starting program: /home/sse/Desktop/workspace/Attack_Phase/65/safe_main abc
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x104a75
```

    - The issue stemmed from dynamic address calculations using offsets shown in Ghidra.

Used `info proc mappings` to get the program's base address, and computed real memory addresses using:

```
process 7971
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555555000     0x1000        0x0 /home/sse/Desktop/workspace/Attack_Phase/65/safe_ma
in
      0x555555555000     0x555555559000     0x4000     0x1000 /home/sse/Desktop/workspace/Attack_Phase/65/safe_ma
in
      0x555555559000     0x55555555a000     0x1000     0x5000 /home/sse/Desktop/workspace/Attack_Phase/65/safe_ma
in
      0x55555555a000     0x55555555c000     0x2000     0x5000 /home/sse/Desktop/workspace/Attack_Phase/65/safe_ma
in
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ff5000     0x7ffff7ff7000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffe000     0x2000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

```
real_address = base_address + offset_from_ghidra
```

2. **Key Extraction:**

   - Located the function where the key was passed as a parameter and dumped the value from the register (e.g., `RSI`).

```
(gdb) x/16xb 0x7fffffffdd40
0x7fffffffdd40: 0xb1    0x6b    0x00    0xb5    0xba    0xad    0xf0    0x0d
0x7fffffffdd48: 0x0b    0xad    0xc0    0xde    0xfe    0xe1    0xde    0xad
(gdb)
```

   - Upon deeper inspection, found another internal function obfuscating the key logic further.

```
void FUN_00102339(long param_1,long param_2)

{
  uint uVar1;
  uint uVar2;
  byte local_1c;
  byte local_1b;
  byte local_1a;
  byte local_19;
  uint local_c;

  FUN_0010365e(param_2);
  for (local_c = 0; local_c < 4; local_c = local_c + 1) {
    *(undefined1 *)(param_1 + (ulong)(local_c << 2)) =
         *(undefined1 *)(param_2 + (ulong)(local_c << 2));
```

   - Manually simplified this function to reveal the final key used for AES encryption.

```
uint8_t final_key[16] = {
    0x15, 0x72, 0x3e, 0x29, 0x95, 0x1f, 0x59, 0x00,
    0xde, 0x00, 0x98, 0xbe, 0x2a, 0x00, 0x61, 0x00 };
```

3. **Egg Parameter (`egg_param`) Extraction:**

   - Calculated actual addresses of the egg parameters using the offset formula.

- Dumped all values at once from memory.

```
(gdb) x/30xd 0x555555559030
0x555555559030: 17      51      68      18      89      32      1       2
0x555555559038: 3       4       5       6       3       6       9       12
0x555555559040: 15      18      1       1       1       1       1       1
0x555555559048: -1      -18     -35     -52     -69     -86
```

- Noticed some values appeared negative (invalid for `uint8_t`), so converted them to positive equivalents to reconstruct the correct egg parameters.

```
uint8_t egg_param[5][6] = {
        {17,  51,  68,  18,  89,  32},
        {1,   2,   3,   4,   5,   6},
        {3,   6,   9,  12,  15,  18},
        {1,   1,   1,   1,   1,   1},
        {255, 238, 221, 204, 187, 170}  };
```

4. **Global Flag Computation:**

- Found a direct computation formula for the global flag.

```
DAT_0010722b = (local_14d[1] * '\x02' + local_14d[0]) - local_14d[2];
```

- Translated the logic to C and built a custom `compute_gf()` function that correctly reproduced the flag.

```
uint32_t compute_gf(uint8_t* eggs) {

   return (eggs[1] * 2 + eggs[0]) - eggs[2];

}
```

Conclusion:

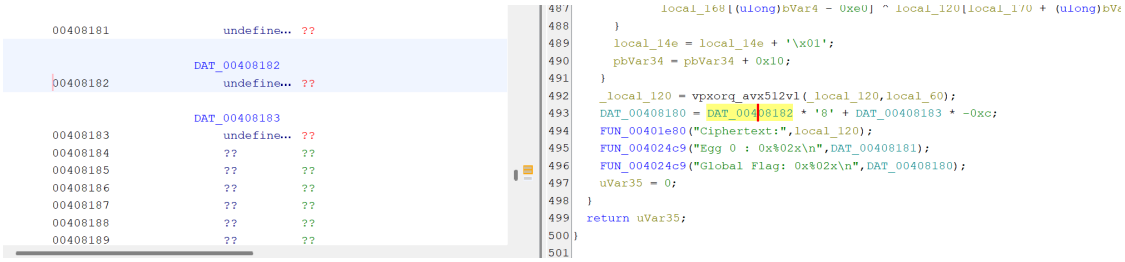
Team 65's binary involved dynamically calculated addresses and a partially obfuscated key logic. With address correction using base mapping and offset math, the attacker could apply standard reverse-engineering strategies. Egg parameters and global flag logic were handled with minor transformations. The outputs are wrong because of keys as per my suspicion.

Team 77 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Decompilation and Debugging Setup:**
    - The binary was decompiled using Ghidra.
    - GDB was used for debugging and validation.
2. **Global Flag Computation:**
    - After analyzing the `main` function, the global flag (`compute_gf`) calculation was directly visible and straightforward.



    - In the print statement of egg we can see that egg[0] address is DAT_00408181.
    - The logic was extracted and directly ported into the attacker's own `main.c`.

```
uint8_t compute_gf(uint8_t* eggs){

    return eggs[1] * 56 + eggs[2] * (-12);

}
```

Conclusion:
Team 77's binary offered minimal resistance. The global flag computation logic was left in plain sight within the main function, making it easy to extract and replicate without needing deep analysis or memory dumps.

Team 69 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

1. **Key Extraction:**
    - Decompiled the binary using Ghidra and identified the relevant function calls.
    - Set a breakpoint just before the function where the key was passed.

```
Breakpoint 1 at 0x40fb8f
(gdb) run abc
Starting program: /home/sse/Desktop/workspace/Attack_Phase/69/safe_main abc
Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function is missing:
Plaintext :: 61 62 63 00 00 00 00 00 00 00 00 00 00 00 00 00

Breakpoint 1, Python Exception <type 'exceptions.NameError'> Installation error: gdb.execute_unwinders function
s missing:
0x000000000040fb8f in ?? ()
```

    - Dumped the key value directly from the appropriate register (e.g., `RSI`), using GDB.

```
(gdb) info registers rsi
rsi            0x7fffffffdd90   140737488346512
(gdb) x/16xb 0x7fffffffdd90
0x7fffffffdd90: 0x7f    0x59    0x02    0xac    0x9d    0x5e    0xfc    0xa8
0x7fffffffdd98: 0xd0    0x55    0x81    0x65    0x45    0x11    0x0c    0xdf
(gdb)
```

Team 89 - Binary Analysis Report

Tools Used: Ghidra

Approach:

1. **Global Flag Computation:**
    - Decompiled the binary using Ghidra.
    - Directly located the `compute_gf` function by analyzing the function names and structure.
    - The logic inside was straightforward, and the attacker translated it into their own `main.c` for correct global flag computation.

```
int compute_gf(long param_1)

{
  return ((uint)*(byte *)(param_1 + 4) * 0x46 + (uint)*(byte *)(param_1 + 3) * -0x31) -
         (uint)*(byte *)(param_1 + 1);
}
```

Team 68 - Binary Analysis Report

Tools Used: Ghidra, GDB

Approach:

2. **Key Extraction:**
    - Decompiled the binary using Ghidra and identified the relevant function calls.
    - Set a breakpoint just before the function where the key was passed.
    - Dumped the key value directly from the appropriate register (e.g., `RSI`), using GDB.