CS6570: Secure Systems Engineering
Assignment-5: Attack Phase

Team: Trojan
Members: Pradeep Peter Murmu (CS24M033), Vishnu K. (CS24M022)

Submission:

1. A modified main.c
2. safe_main binary - the final binary generated from docker
3. patch_sections.py - A python script to obfuscate the binary
4. Dockerfile- Runs the patch_sections.py and some other command on virtual container and transfers compiled binary to the host.
5. The report (You are reading)

Obfuscations Methods:

1. XOR encryption in main.c

The original AES key is not stored in plaintext. Instead, an obfuscated version (key_obf) is stored. At runtime, our function (key_5611()) XORs each byte of the obfuscated key with a constant mask (jskeynew, set to 0xAA) to recreate the real AES key.

Similarly, the egg_params are stored in an encrypted form in the array egg_params_enc. Before encryption, the original egg parameters (e.g., a 5×6 matrix) were XORed with a constant mask (jskegg, set to 90 in decimal) to produce egg_params_enc. The function (egg_8989()) is then used at runtime to decrypt these values by XORing them with the same mask.

We have used obscure names for variables and function names to make it difficult while reading the code.

2. Obfuscating compute_gf()

we first decompose the constant 91 into a product of 13 and 7. We compute 13 times eggs[1] by using two left shifts and one addition: shifting eggs[1] left by 3 bits gives 8 times its value, shifting it left by 2 bits gives 4 times the value, and adding the original value yields 13 times eggs[1]. Multiplying the result by 7 reconstructs the intended multiplication of 91.

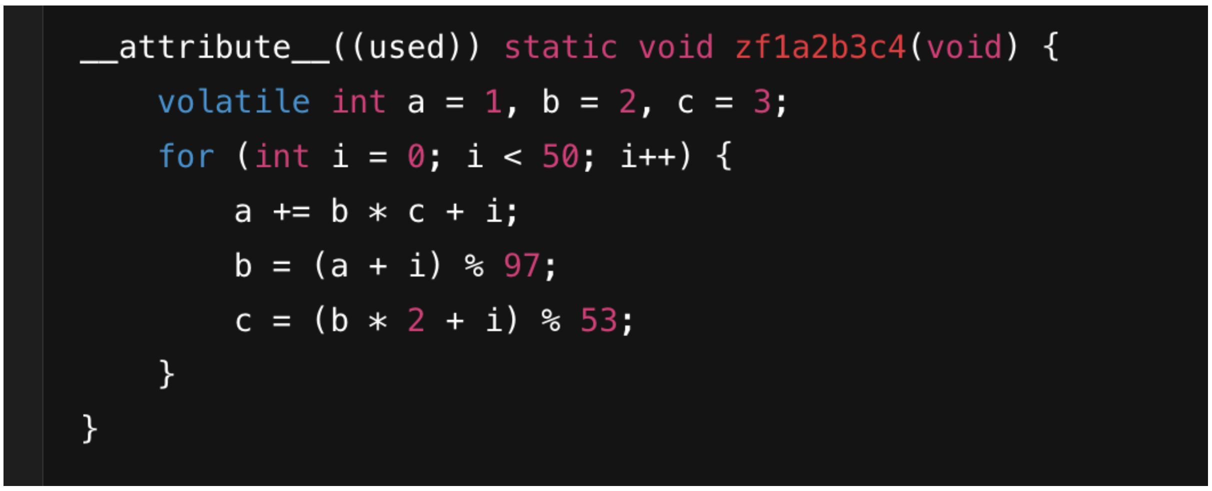
Next, we handle the constant 73 associated with eggs[3] by computing 72 times eggs[3] and then adding eggs[3] to that product. The shift of eggs[3] left by 3 bits multiplies it by 8, and then multiplying by 9 produces 72 times eggs[3]. Adding eggs[3] yields the desired 73 times eggs[3].

Additionally, we introduce a dummy operation that calculates zero. By multiplying 35 by 3 and subtracting 105, we generate an expression that always evaluates to 0. This value is added to eggs[4] purely to add noise to the computation and confuse static analysis tools without changing the result.

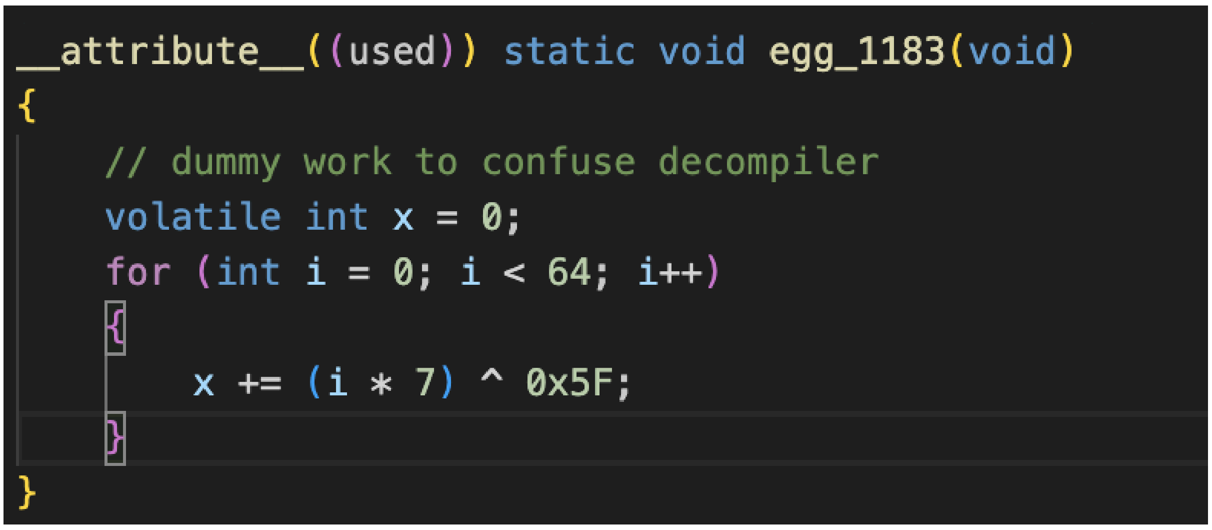
Finally, the obfuscated function subtracts the computed value for eggs[3] and eggs[4] from that of eggs[1] to generate the final result.

3. Garbage code

Added some garbage functions like below to add noise to the binary.



```c
__attribute__((used)) static void zf1a2b3c4(void) {
    volatile int a = 1, b = 2, c = 3;
    for (int i = 0; i < 50; i++) {
        a += b * c + i;
        b = (a + i) % 97;
        c = (b * 2 + i) % 53;
    }
}
```



```c
__attribute__((used)) static void egg_1183(void)
{
    // dummy work to confuse decompiler
    volatile int x = 0;
    for (int i = 0; i < 64; i++)
    {
        x += (i * 7) ^ 0x5F;
    }
}
```

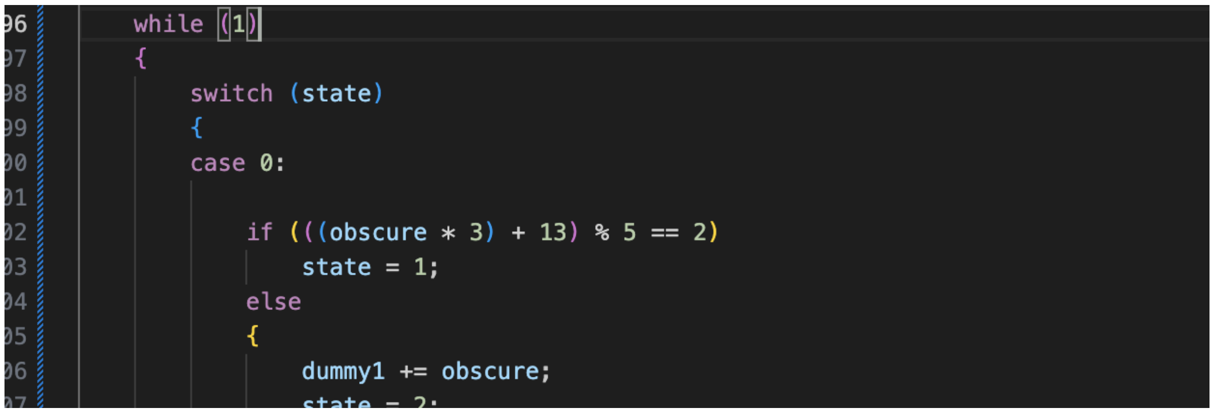
4. Control flow flattening

It transforms the natural, structured control flow of a program into a uniform state machine or dispatcher loop. By using opaque predicates and dummy branches along with a central state variable, the true logic is hidden among many unnecessary and confusing control-flow paths.

In the main function, I’ve added a switch case to do it.
State Machine & Opaque Predicate (Cases 0–2):
Inserts an opaque arithmetic condition based on obscure (set to argc) and routes the flow into either a dummy branch or the true flow, confusing static analysis.

Key & Egg Function Integration with Dummy Arithmetic (Cases 3–7):
In state 3, the genuine key de-obfuscation function (key_5611) is called; thereafter, dummy computations (using dummy1–dummy5) are performed before proceeding to state 7 where the genuine egg_8989() function is invoked.

Termination via goto (Case 8):
After a final dummy operation, the state machine is exited cleanly using a goto label so that normal processing of the input and AES encryption can continue as originally intended.



```
    while (1)
    {
        switch (state)
        {
        case 0:

            if (((obscure * 3) + 13) % 5 == 2)
                state = 1;
            else
            {
                dummy1 += obscure;
```

5. Obfuscation script (patch_sections.py)

We are using a python script to obfuscate the compiled binary.
The obfuscation in the script are as follows:

1. Encrypt .text Section Using AES:
• Generates a random 16-byte AES key to secure code.

• Encrypts the original .text section using AES in CBC mode with proper padding.
• Replaces the plaintext code with the ciphertext (IV prepended) to thwart static analysis.

2. Rename Sensitive Sections & Remove Extra Data:
• Renames standard sections like .rodata and .text to random names.
• Removes sections (e.g., .comment, .symtab, .debug) that reveal build details.
• Obscures useful metadata, confusing decompilers and reverse engineering tools.

3. Rename All Non-Critical Sections:
• Iterates over every section not essential for execution (e.g., .interp, .dynamic).
• Reassigns a random name to each, scrambling the expected ELF structure.
• This widespread renaming makes the binary layout unpredictable for analysis tools.

4. Randomize Section Alignments:
• For each section with alignment, a new power-of-2 alignment is randomly chosen.
• Altering alignments disrupts the typical and predictable memory layout.
• This extra layer further complicates the analysis by breaking standard layout patterns.

Dockerfile

Base Image:
Uses ubuntu:22.04 as the base for compatibility and stability.

Tool Installation:
Installs system tools (gcc, binutils, python3, etc.) and Python libraries:
lief – for ELF section manipulation.
keystone-engine – (assembly injection, if needed).
pycryptodome – for AES encryption of binary sections.
pwn – pwntools, for binary introspection and patching.

Build Setup:

Sets working directory to /app.
Copies main.c and patch_sections.py into the container.

Compile & Patch:

Compiles main.c statically into safe_main.
Runs patch_sections.py to encrypt, rename, and remove ELF sections. (Main logic of obfuscation is run inside this python script, so refer to the obfuscation section for more information)

Binary Cleanup:
Uses strip to remove any remaining debug symbols from the final binary.

Output Management:

Copies the patched binary to /output, which is the final working directory.
At runtime, it re-copies safe_main from /app to /output (in case host mapping is in use), and executes the binary with a test argument.

Steps to run the Dockerfile:

In the root directory with Dockerfile, main.c and patch_sections.py file run the below commands: (commands used here are for mac/linux system)

Step-1: docker build -t fortify-obf .

Step-2: docker run --rm -v "$PWD:/output" fortify-obf

Contribution

Pradeep P.M. - XOR encryption, Obfuscating compute_gf function, Obfuscating python script, Report

Vishnu K.- Garbage code, Control flow flattening, Obfuscating python script, Report

Acknowledgement:

1. https://digital.ai/catalyst-blog/how-to-obfuscate-c-code/
2. https://zimperium.com/blog/top-7-source-code-obfuscation-techniques