Assignment 5 - Report

Secure Systems Engineering

Shreyas Bargale CS22B016
Raadhes Chandaluru CS22B069

Explanation of Obfuscation Techniques

8 - bit integer Representation

The file repr.h contains representation of the 8 bit integer we have used. The representation used is struct boolrepr2.

```
struct boolrepr{
    bool eight;
    bool three;
    bool four;
    bool six;
    bool one;
    bool two;
    bool five;
    bool seven;
};
struct boolrepr1{
    struct boolrepr x;
    struct boolrepr y;
};
struct boolrepr2
{
    struct boolrepr1 x;
    struct boolrepr1 y;
};
```

- boolrepr: Represents a 8 bit integer with individual bits
- boolrepr1: Represents an 8 bit integer "Z" as the sum of x and y.
- boolrepr1: Represents an 8 bit integer "Z" as the sum of member variables x and y.

The structs have several functions such as addition, subtraction, xor, and multiplication that we have implemented so that operations can be performed without full conversion to int. These functions are manually implemented atleast for the lowest struct boolrepr in order to make understanding it much harder.

Several #define have been created the top of the main.c file. These simply allow use to quickly change representation being used.

```
#define STRUCT_REPR struct boolrepr2
#define CONSTR_ boolconstr2_
```

```
#define REV_CONSTR_ rev_boolconstr2_
#define ADD_ booladd2_
#define SUB_ boolsub2_
#define MULT_ boolmult2_
#define XOR_ boolxor2_
#define AND_ booland2_
#define EQUALS_ boolequals2_
```

The following arrays have been modified to use the above struct representation:

- egg_params => Now egg_params1. Although in this several modifications have been beyond representation to ensure safety.
- sbox => Now sbox1. sbox[2048] is also created of int8_t as data bloat.
- Eggs, Roundkey, key, Rcon, and state arrays have also been modified to use above struct representation.

Ignore repr and repr1 structs in the file. They were used previously in or code but not anymore. They have been left there as bloating code.

Shuffling of arrays & Index mapping

Egg_params1 in our code uses a simple shuffling mechanism. The values are swapped between each other in a 30 size 2-d array every time one access is made. The x,y indices are stored to keep of track of where each element is present. [egg_param_x and egg_param_y arrays]

state uses a more complicated shuffling and index mapping.
Three arrays are used:

- int8_t statemap[16] : Maps to one element in hidden layer
- int hiddenstatemap[100] : Maps to final state array
- STRUCT_REPR finalstate[100] : Has state values

shuffle_state() function has been created to shuffle around the state values in final_state

Force Inlining & Code Bloat

In repr.h:

```
#define INLINER1_ __attribute__((always_inline)) static inline
```

This forces inlining of high level boolrepr2 functions. We have choose not to force inline boolrepr1 and boolrepr functions.

Code bloat has been done by adding some useless functions and lines of code. Code has been taken from previously written code on my machine.

Added few junk threads. Added a few junk functions. Added a few functions not required for the program.

Function trees

Interesting scheme where we have a large binary tree of functions. The computation is spread out over these functions or is done at only few nodes such that anyone trying to trace the functionality will be lost. The large number of functions were generated using python scripts.

These function trees were created for:

- Computing global flag. [compute_gf_bomb.h]
- Transferring the final data for printing to egg[0] and global_flag variables. [transfer_bomb.h]
- Decrypting part of the .text section. [crypt_bomb.h]
- Egg calculations. [egg_calculator_bomb.h]
- Key Expansion [key_expansion_bomb.h]
- Shift rows [shift_rows_bomb.h]

Asm code in C & codecave

Assembly code can be added in C functions using asm(). We have done this to mess with static analysis of ghidra and objdump.
Control flow has been messed with and variable length instructions concept have been used to confuse such tools.

- Simple: This simply skips a few bytes to skip label. The instruction start with 0xb8 0x78 and 0x56 expects two more bytes. Hence this can confuse tools. Objdump is easily fooled after symbols are stripped by this. The executed instructions will be different from the objdumped ones just after skip label.

```
    asm volatile (
        "jmp skip\n"
        ".byte 0xB8, 0x78, 0x56\n"
        "skip:\n"
    );
```

- More: Similar but uses a different control flow mechanism. (ret)

```
    asm volatile(
        "lea skip6(%%rip), %%rax\n"
        "push %%rax\n"
        "ret\n"
        ".byte 0xB8, 0x78, 0x58\n"
        "skip6:\n"
```

```
         :
         :
         : "rax");
```

- Complex: This is more complicated. Codecave is a dummy function. We fill required some instructions in the function at runtime , then use asm code to jump to it.
- The instructions written add 20 to rax (0X488C014), push %rax to stack (0x50), ret (0xc3). This moves the control to target label.
- Notes: The instructions written at makes static analysis harder. Additionally I have subtracted 20 initially as Ghidra seems to be very smart and places labels at few places if the program stores pointers to those locations. We do not want ghidra to place a predict a label at location target and decompile from there as it will reveal the code after target. Note that the target symbol will not be present in the sripped binary.

```
     char* cave = codecave;
     cave[4] = 0x48;
     cave[5] = 0x83;
     cave[6] = 0xc0;
     cave[7] = 0x14;
     cave[8] = 0x50;
     cave[9] = 0xc3;
     asm volatile(
         "call 1f\n"
         "1:\n"
         "pop %%rax\n"
         "sub $20, %%rax\n"
         "add $target-1b, %%rax\n"
         "lea codecave(%%rip), %%rbx\n"
         "add $4, %%rbx\n" // move to byte 5
         "jmp *%%rbx\n"    // jump to the ret
         ".byte 0xB8, 0x78, 0x58\n"
         "target:"
         :
         :
         : "rax", "rbx");
```

Simple Encryption of binary

Implemented encryption of the a large portion of the .text section of the binary containing the AES functions and self - made functions. This prevents static analysis attacks. Ghidra fails to decompile from binary due to this.

Encryption: Binary is generated with gcc compiler. The python script crypt.py then encrypts the .text section in 3 loops with 3 simple encryption schemes on individual bytes.

- Swap MSB and LSB bytes of each byte
- XOR each byte with a key that increases by 67 each iteration (mod)
- Predefined mapping of byte to byte. Uses a 256 size array

During the runtime the binary first sets the required pages as writable using mprotect. Then the decryption takes place with 3 decryption functions names crypt1, crypt2, crypt3. The order of application will be reverse to the encryption.

Pthreads

Threads were launched in order to disturb control flow especially if the program is being debugged and then joined at appropriate points to ensure correct control flow and execution. Random junk threads were also launched alongside main thread to disturb debugging and add noise. This makes it difficult to follow program execution and differentiate between useful and junk code.

Trampoline

This used a function pointer to dynamically set it and call a function. This can help in combat static analysis from finding out which function is being called.

Strip Symbols

After the encryption we strip all variable and function symbols from the binary to make it harder for anyone to understand.
Strip command is used for this.
strip –strip-all ./<binary>

DockerFile explanation

The Dockerfile automates the generation of statically linked obfuscated binary.
The DockerFile requires all the source code files required for obfuscating the main.c. It first loads all the regular softwares such as gcc, python and the required dependencies. The image is based on a lightweight python base. The output binary is copied to /output which is mounted from host using -v flag.

```
# syntax=docker/dockerfile:1
FROM python:3.9-slim-buster
```

```
RUN apt-get update && apt-get install -y gcc build-essential
```

The python dependencies are downloaded from the requirements.txt (pyelftools) using pip.

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

The working directory is set.Then all the source codes required for creating the binary are copied into the docker image.

```
WORKDIR /workspace


COPY . .
```

This is followed by the instructions to actually compile the binary in its totality similar to a Makefile. We use a python script to generate headers such as crypt_bomb.h which has a lot of functions(function bomb) as well as compute_gf_bomb.h. These are included in the obfuscated main.c statically(can be done without doing it everytime as well). Then we compile the code using flags like -pthread and -static for statically linked binary(fully self-contained binary) and support for multi-threading. This compiled binary is then fed to a python encryption file which finally gives an encrypted binary. This binary still has labels in it so we strip then using the strip command.

```
# Build steps
RUN gcc main.c -static -o safe_main -lpthread  && \
    python3 crypt.py 0xFF && \
    strip --strip-all safe_main
```

We then run the command cp /workspace/a.out /output/safe_main to copy the a.out binary to the folder we pass as argument to the docker run command.

```
# Copy the final binary to /output (must be mounted by host)
CMD cp /workspace/safe_main /output/safe_main
```

Commands to run the Dockerfile

1. Put all the source codes and the Dockerfile in the same directory
2. **Build the docker image using-**
    - docker build -t build_main .
3. **Create the output folder using-**
    - mkdir -p ./output
4. **Run the docker image and get the final binary in the output folder with-**
    - docker run --rm   -v "$PWD/output":/output build_main

Note the following files are present:

compute_gf_bomb.h
egg_calculator_bomb.h
transfer_bomb.h
crypt_bomb.h
key_expansion_bomb.h
repr.h
crypt.py
main.c
requirements.txt
Dockerfile
Makefile
shift_rows_bomb.h
<REPORT FILE>

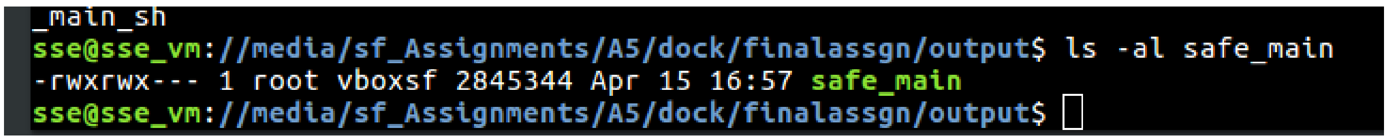
Output & Timings:

Below are output and timings on the provided tests:

```
sse@sse_vm://media/sf_Assignments/A5/32/32$ time ./safe_main abcdefghijklmnopq
Plaintext :: 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70
Ciphertext:: 25 47 73 f9 e0 d8 b9 b7 e6 85 dd 80 85 ae 0e c0
Egg 0 : 0xc6
Global Flag: 0xf6

real	0m0.068s
user	0m0.012s
sys	0m0.012s

sse@sse_vm://media/sf_Assignments/A5/32/32$ time ./safe_main qrstuvwxyzabcdef
Plaintext :: 71 72 73 74 75 76 77 78 79 7a 61 62 63 64 65 66
Ciphertext:: c1 25 a3 1c 46 68 71 9b 1a 10 99 32 5d c2 3b ce
```

```
Egg 0 : 0xe6
Global Flag: 0x62

real	0m0.073s
user	0m0.007s
sys	0m0.015s

sse@sse_vm://media/sf_Assignments/A5/32/32$ time ./safe_main 1234567890abcdef
Plaintext :: 31 32 33 34 35 36 37 38 39 30 61 62 63 64 65 66
Ciphertext:: 94 e3 be 16 e4 ba 65 45 66 93 7b 49 a9 70 c4 be
Egg 0 : 0x01
Global Flag: 0x3b

real	0m0.037s
user	0m0.015s
sys	0m0.008s

sse@sse_vm://media/sf_Assignments/A5/32/32$ time ./safe_main abcdef1234567890
Plaintext :: 61 62 63 64 65 66 31 32 33 34 35 36 37 38 39 30
Ciphertext:: cd d0 7a ca b1 d3 e1 8d 81 ae 2b d4 77 2d a0 31
Egg 0 : 0xb8
Global Flag: 0x40

real	0m0.044s
user	0m0.008s
sys	0m0.013s

sse@sse_vm://media/sf_Assignments/A5/32/32$ time ./safe_main ghijklmnopqrstuv
Plaintext :: 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76
Ciphertext:: dd db 9d 5f ce f4 de 10 a7 fb 2f bb b4 72 8b 00
Egg 0 : 0x4e
Global Flag: 0x34

real	0m0.075s
user	0m0.017s
sys	0m0.009s
```

Size:



```
_main_sh
sse@sse_vm://media/sf_Assignments/A5/dock/finalassgn/output$ ls -al safe_main
-rwxrwx--- 1 root vboxsf 2845344 Apr 15 16:57 safe_main
sse@sse_vm://media/sf_Assignments/A5/dock/finalassgn/output$
```

Libraries

Apart from the #includes of the self-written ".h" files we have included the following in code.

```
#include <stdint.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>    // std lib
#include <stdbool.h>   // Bools

#define _GNU_SOURCE
#include <unistd.h>
#include <stdint.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/ptrace.h>  // gdb debugging but unused now
#include <sys/prctl.h>   // gdb debugging stop but unused now
#include <errno.h>
#include <pthread.h>     // multithreading
```

Resources Used & Acknowledgements:

- Objdump, GDB
- Objdump -d and Objdump -s
- Pyelftools in python
- Docker
- Ghidra

Contributions

Raadhes focused on encryption, 8 bit representation obfuscations, func trees and asm code.
Shreyas focussed on threading obfuscations, trampoline obfuscation for mprotect function, testing the code, getting the dockerfile ready.