2024 암호분석경진대회 - 비밀분쇄기팀 3번 문제 풀이

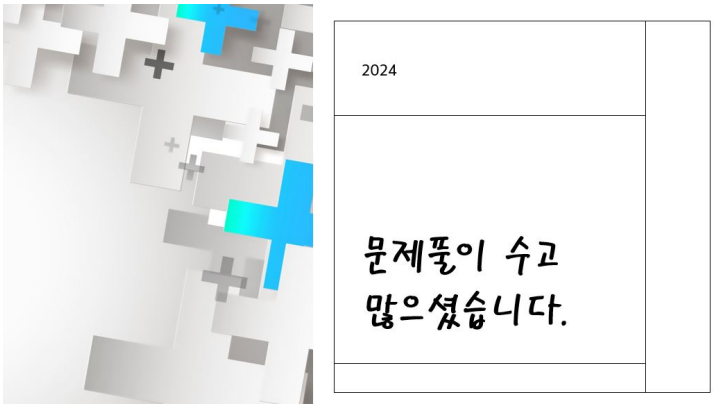

Figure 1: 복호화된 이미지.
복호화를 위해 시도해야 하는 쿼리 회수는 최대 $2^{12} = 4096$개이며 본 풀이에서는 0x197=407번째 쿼리에서 복호화에 성공하였다.

1 암호화 프로그램 구조 파악

정적 분석에 Ghidra, 동적 분석에 x64dbg를 사용하였고, ret-sync 플러그인을 활용하여 두 프로그램을 연동하여 사용하였다. Ghidra는 디컴파일 된 함수명을 `FUN_(코드의 주소)` 형태로 표기하므로 풀이에서도 해당 표기법을 사용하고자 한다.

`cryptocontest.exe`를 Ghidra로 디컴파일 후 전체적인 구조를 살펴본 결과 `FUN_1400012f0` 함수에서 암호화 된 이미지를 출력하는 것을 확인할 수 있었다. 따라서 해당 함수부터 분석을 시작하였다.

1.1 `FUN_1400012f0` 함수 분석

`FUN_1400012f0` 함수의 후반부 코드를 human-readable C로 옮겨보면 다음과 같다.

```
FILE *fin = fopen("c_contest_2024.jpg", "rb");
FILE *fout = fopen("c_contest_2024_out.jpg", "wb");
if (fin && fout) {
  fseek(fin, 0, SEEK_END);
  long nbytes = ftell(fin);
  fseek(fin, 0, SEEK_SET);
  for (long i = 0; i < nbytes; ++i) {
    unsigned char x;
    fread(&x, 1, 1, fin);
    x = x ^ key[rand() % 64];
    fwrite(&x, 1, 1, fout);
  }
  fclose(fin);
  fclose(fout);
}
```

64바이트 배열 `key`가 존재하고, 입력 이미지의 각 바이트를 읽어 `key` 배열의 랜덤한 원소와 xor을 적용하여 출력하는 것을 볼 수 있다. 따라서 `key` 배열이 어떻게 생성되는지 파악하는 것이 문제의 핵심이다. `key` 배열은 `FUN_1400012f0` 함수의 전반부에서 생성되고 있으며 이를 간추리면 다음과 같다.

```
iVar3 = FUN_140001070(local_148, &local_188);
if (iVar3 != 0) {
  do {
    iVar3 =
        FUN_140001070(puVar6 + (longlong)(int)uVar10 * 0x2f,
                      (int *)((longlong)pvVar7 + (longlong)(int)uVar10 * 4));
    uVar10 = uVar10 + 1;
  } while (uVar10 < 0x10);
```

```
  if ((puVar6 != (uint *)0x0) && (*(short *)(puVar6 + 0x2e) != 0)) {
    FUN_140001b90(puVar6, (uint *)local_68, 0x40, param_4);
  }
  lVar12 = 0x10;
  do {
    if ((puVar6 != (uint *)0x0) && (*(short *)(puVar6 + 0x2e) != 0)) {
      FUN_140001b90(puVar6, (uint *)local_88, 0x20, param_4);
    }
    // ...
    // xor 0x20 bytes of local_68 and local_88
    // ...
    if ((puVar6 != (uint *)0x0) && (*(short *)(puVar6 + 0x2e) != 0)) {
      FUN_140001b90(puVar6, (uint *)local_88, 0x20, param_4);
    }
    // ...
    // xor 0x20 bytes of local_48 and local_88
    // ...
    lVar12 = lVar12 + -1;
  } while (lVar12 != 0);
  ...
```

먼저, FUN_140001070 함수를 최상단에서 한번, do-while loop에서 16번, 총 17번 호출하는 것을 볼 수 있다. 해당 함수는 첫번째 인자로 포인터를 받는데 함수 호출 직후 해당 메모리 영역을 출력해보면 아래와 같은 값을 볼 수 있다.

```
65 78 70 61 6E 64 20 33 32 2D 62 79 74 65 20 6B
95 26 30 FD 2F 0F 0E 68 50 75 3F C9 ED 94 E7 B8
EB A4 56 B8 E7 31 B0 4A AA 4C B4 F0 CE 3F 39 E7
01 00 00 00 00 00 00 00 6D 93 56 E8 D0 5B 39 9D
```

여기서 첫 16-byte는 ASCII 문자열로 해석했을때 "expand 32-byte k"로 chacha20 알고리즘의 첫 16-byte state로 사용되는 문자열이다. 또한 chacha20 state의 counter에 해당하는 위치에 1이라는 작은 값이 들어있다. 따라서 FUN_140001070 함수는 chacha20 알고리즘의 상태 등을 초기화하는 함수라고 추측할 수 있다.

이후에는 FUN_140001b90 함수를 17번 호출하는데, 2번째 인자(puVar6)가 가리키는 메모리 위치에 3번째 인자(0x20 또는 0x40) 만큼의 바이트가 수정되는 것으로 보아 chacha20 알고리즘으로 암호 스트림을 생성하는 함수로 추측할 수 있다. 동작을 자세히 살펴보면, 첫번째 호출은 초기화한 16개의 chacha20 state 중 첫번째를 이용하여 0x40 바이트의 스트림을 생성한다. 그리고 do-while loop로 16개의 chacha20 state를 순회하면서 0x40 바이트의 스트림을 생성한 후 직전에 생성한 스트림과 xor하는 것을 알 수 있다.

그러므로 FUN_140001070와 FUN_140001b90의 동작을 정확히 알아내면 전체 프로그램의 동작을 파악할 수 있다.

1.2 FUN_140001070 함수 분석

FUN_140001070 함수의 핵심 루틴을 간추리면 다음과 같다.

```
do {
  iVar7 = rand();
  cVar2 = (char)iVar8;
  iVar8 = iVar8 + 1;
  *pbVar9 = (char)iVar7 * (cVar2 + '\x01' + *(char *)param_2);
  pbVar9 = pbVar9 + 1;
} while (iVar8 < 0x28);
FUN_1400020f0(local_58,0x28,param_1);
iVar7 = rand();
_Memory = (undefined8 *)malloc((longlong)(iVar7 % 0x100 + 9));
if ((_Memory != (undefined8 *)0x0) && (*(short *)(param_1 + 0x2e) != 0)) {
  *_Memory = 0;
  FUN_140001b90(param_1,(uint *)_Memory,8,(uint *)_Memory);
}
free(_Memory);
```

첫 부분은 pbVar9가 가리키는 메모리 영역의 28-byte를 랜덤한 값으로 채우는 것을 볼 수 있다. Human-readable C로 옮겨보면 다음과 같다.

```
for (int i = 0; i < 0x28; ++i) {
  int iVar7 = rand();
  char x = (char)iVar7 * (i + 1);
  pbVar9[i] = x;
}
```

그 다음 FUN_1400020f0 함수가 호출되는데, 이 함수의 핵심 루틴을 간추리면 다음과 같다.

```
iVar2 = FUN_140001b90(param_3,(uint *)local_38,0x28,(uint *)local_38);
if (iVar2 == 0) {
```

```
  uVar8 = 0;
  // ...
  // xor 0x28 bytes of param_1 and local_38
  // ...
  uVar3 = FUN_140001ea0(param_3,(undefined4 *)local_38);
  if ((int)uVar3 == 0) {
    *(ulonglong *)(param_3 + 0xc) = uVar8;
    param_3[0xe] = (uint)local_18;
    param_3[0xf] = (uint)((ulonglong)local_18 >> 0x20);
  }
}
```

먼저, FUN_140001b90 함수 호출을 통해 현재 chacha20 state(param_3)로부터 0x28 바이트의 스트림을 local_38에 생성하는 것을 볼 수 있다. 이 값은 param_1, 즉 FUN_140001070에서 생성한 0x28 바이트의 랜덤 값과 xor 된다. 이 값은 최종적으로 FUN_140001ea0 함수에 넘겨지는데, 해당 함수는 첫번째 인자가 가리키는 메모리 영역을 "expand 32-byte k"라는 스트링과 두번째 인자가 가리키는 메모리 영역의 첫 0x20 바이트로 채우는 함수이다. 즉, chacha20 state의 key에 해당하는 영역이 앞서 xor로 연산한 값으로 채워지게 된다. 마지막으로, chacha20 state의 counter에 해당하는 영역은 0(uVar8)로 초기화되고, nounce에 해당하는 영역은 0x28 바이트 스트림 중 사용하지 않은 마지막 0x8바이트로 채워지게 된다.

FUN_1400020f0 호출 이후에는 랜덤한 크기(rand() % 0x100 + 9)의 메모리를 할당한 뒤 FUN_140001b90 함수를 호출하여 해당 영역에 0x8 바이트의 chacha20 스트림을 생성한 뒤 사용하지 않고 할당해제 하는 것을 볼 수 있다. 이 부분의 정확한 구현 의도는 모르겠으나 rand 함수의 state와 chacha20의 state를 모두 변경하므로 프로그램을 재현할때 고려되어야만 한다.

1.3 FUN_140001b90 함수 분석

FUN_140001b90 함수는 앞선 추측대로 chacha20 스트림을 생성하는 함수임을 확인할 수 있었다. 3개의 인자를 받는데, 첫번째 인자는 chacha20 알고리즘의 state를 가리키는 포인터고, 두번째 인자는 스트림을 생성할 메모리 영역이며, 세번째 인자는 생성할 스트림의 길이이다.

2 암호화 프로그램 재현

앞서 분석한 내용을 바탕으로 암호화 프로그램의 동작을 재현할 수 있다. chacha20의 구현은 https://github.com/Ginurx/chacha20-c를 사용하였다. 아래는 chacha state를 초기화하는 FUN_140001070 함수의 재현 코드이다.

```
void init_chacha_ccstyle(chacha20_context *ctx) {
  chacha20_context init_ctx;
  uint8_t empty_key[32] = {0};
  uint8_t empty_nonce[12] = {0};
  chacha20_init_context(&init_ctx, empty_key, empty_nonce, 0);
  uint8_t iv[0x28];
  for (int i = 0; i < 0x28; ++i) {
    int iVar7 = myrand();
    char x = (char)iVar7 * (i + 1);
    iv[i] = x;
  }
  chacha20_xor(&init_ctx, iv, 0x28);

  uint8_t iv_nonce[12] = {0};
  memcpy(&iv_nonce[4], &iv[0x20], 8);
  chacha20_init_context(ctx, &iv[0], iv_nonce, 0);
  myrand();
  uint8_t dummy_buf[8] = {0};
  chacha20_xor(ctx, dummy_buf, 8);
}
```

먼저, key와 nonce를 0으로 설정한 chacha20 context로 0x28 바이트 스트림을 생성하고, FUN_140001070과 같은 방식으로 0x28 바이트의 랜덤값을 생성한 후에 xor을 취하여 배열 iv에 저장한다. 그리고 새 chacha20 context를 생성한 후에 key와 nonce를 iv를 활용하여 초기화한다. 마지막으로 암호화 프로그램과 동작을 일치시키기 위해 rand 함수를 1회 추가 호출하고 더미 버퍼를 생성하여 0x8 바이트만큼 스트림을 생성한뒤 값을 버려준다.

아래는 전체적인 이미지 암호화를 관장하는 FUN_1400012f0 함수의 재현 코드이다.

```
void encrypt(unsigned int seed, uint8_t key[64]) {
  mysrand(seed & 0xf0f0f0f0);
  chacha20_context dummy_ctx;
  chacha20_context main_ctx[16];
  init_chacha_ccstyle(&dummy_ctx);
  for (int i = 0; i < 16; ++i) {
    init_chacha_ccstyle(&main_ctx[i]);
  }
  chacha20_xor(&main_ctx[0], key, 64);
  for (int i = 0; i < 16; ++i) {
```

```
    chacha20_xor(&main_ctx[i], key, 64);
  }

  FILE *fin = fopen("c_contest_2024.jpg", "rb");
  FILE *fout = fopen("c_contest_2024_out.jpg", "wb");
  if (fin && fout) {
    fseek(fin, 0, SEEK_END);
    long nbytes = ftell(fin);
    fseek(fin, 0, SEEK_SET);
    for (long i = 0; i < nbytes; ++i) {
      unsigned char x;
      fread(&x, 1, 1, fin);
      x = x ^ key[myrand() % 64];
      fwrite(&x, 1, 1, fout);
    }
    fclose(fin);
    fclose(fout);
  }
}
```

`FUN_1400012f0` 함수와 마찬가지로 17개의 chacha20 context를 생성해준다. 그 중 1개의 context는 사용하지 않고, 나머지 16개 context에서 64-byte 키를 생성하여 해당 키로 암호화를 해준다. 주어진 암호화 프로그램은 암호화된 이미지와 함께 seed와 사용한 key를 출력하는데, 같은 seed를 사용했을때 같은 key를 생성함을 확인하여 재현 여부를 검증할 수 있었다.

2.1 rand 함수 구현에 대한 고려

rand 함수의 구현은 컴파일 환경에 따라 다를 수 있다. 리눅스에서 gcc로 테스트를 진행했기 때문에, 암호화 프로그램이 사용한 MSVC의 rand 구현과 일치시키기 위하여 아래와 같은 함수를 구현하여 사용하였다. rand의 구현은 x64dbg에서 rand 함수 호출을 따라가서 확인할 수 있었다.

```
unsigned int myseed = 0;
void mysrand(unsigned int seed) {
  myseed = seed;
}
unsigned int myrand() {
  myseed = myseed * 214013L + 2531011L;
  return myseed>>16 & 0x7FFF;
}
```

3 이미지 복호화

이미지를 복호화하기 위해서는 크게 두가지가 필요하다. 먼저, seed에 따라 key가 달라지기 때문에 seed를 추측할 필요가 있다. 두번째로 평문의 일부를 알고 있어야 해당 key가 맞는 key인지 판단할 수가 있다.

첫번째 문제는 암호화 프로그램의 특성으로부터 쉽게 해결할 수 있다. 암호화 프로그램은 현재 시간과 `0xf0f0f0f0`을 and 연산한 값을 seed로 사용하고 있다. 또한 프로그램에서 `rand()` 반환값의 하위 8비트만 사용하고 있기 때문에 seed의 상위 4비트는 결과에 영향을 미치지 않는다. **그래서 실질적으로 가능한 seed의 수가 최대 2^{12}개밖에 되지 않아 쉽게 전수조사 할 수 있다.**

두번째로 암호화된 이미지의 이름으로부터 원본이미지가 jpg 포맷임을 추측할 수 있고, 따라서 jpg 파일의 첫 4바이트가 `FF D8 FF E0`인 점을 활용할 수 있다. 아래는 이를 활용한 복호화 코드이다.

```
void test_with_seed(unsigned int seed, uint8_t key[64]) {
  mysrand(seed);
  chacha20_context dummy_ctx;
  chacha20_context main_ctx[16];
  init_chacha_ccstyle(&dummy_ctx);
  for (int i = 0; i < 16; ++i) {
    init_chacha_ccstyle(&main_ctx[i]);
  }
  chacha20_xor(&main_ctx[0], key, 64);
  for (int i = 0; i < 16; ++i) {
    chacha20_xor(&main_ctx[i], key, 64);
  }
}

bool test_jpg(uint8_t key[64]) {
  if (
    (0xff ^ key[myrand() % 64]) == 0x74
    && (0xd8 ^ key[myrand() % 64]) == 0x5c
    && (0xff ^ key[myrand() % 64]) == 0xd6
    && (0xe0 ^ key[myrand() % 64]) == 0x69
    ) {
    return true;
  }
```

```
  return false;
}

void decrypt_with_seed(unsigned int seed) {
  uint8_t key[64] = {0};
  test_with_seed(seed, key);

  FILE *fin = fopen("c_contest_2024_out.jpg", "rb");
  FILE *fout = fopen("c_contest_2024.jpg", "wb");
  if (!fin || !fout) {
    printf("file open failed\n");
    return;
  }
  fseek(fin, 0, SEEK_END);
  long nbytes = ftell(fin);
  fseek(fin, 0, SEEK_SET);
  uint8_t enc[nbytes];
  fread(enc, nbytes, 1, fin);
  uint8_t rand_seq[nbytes];
  for (long i = 0; i < nbytes; ++i) {
    rand_seq[i] = myrand() % 64;
  }
  uint8_t dec[nbytes];
  for (int i = 0; i < nbytes; ++i) {
    dec[i] = enc[i] ^ key[rand_seq[i]];
  }
  fwrite(dec, nbytes, 1, fout);
  fclose(fin);
  fclose(fout);
}

int main() {
  for (unsigned int i = 0; i < 1 << 12; ++i) {
    unsigned int seed = ((i & 0xf) << 4) | ((i & 0xf0) << 8) | ((i & 0xf00) << 12);
    int success = 0;
    uint8_t key[64] = {0};
    test_with_seed(seed, key);
    if (test_jpg(key)) {
      printf("seed: %08X\n", seed);
      decrypt_with_seed(seed);
      exit(0);
    }
  }
  return 0;
}
```

`main` 함수에서는 가능한 seed를 순회하면서 암호 프로그램과 같은 방식으로 key를 생성 후에 해당 key로 첫 4바이트(`FF D8 FF E0`)를 암호화한 값이 암호화된 이미지의 첫 4바이트(`74 5c d6 69`)와 일치하는지 확인한다. 일치한다면 암호화된 이미지의 각 바이트를 key 배열의 값과 xor 연산하여 복호화 할 수 있다.

위 복호화 프로그램은 1ms 내외로 seed를 찾을 수 있었으며 0x00109070로 확인이 되었다. 복호화된 이미지는 풀이의 첫 부분에 첨부하였다.