C 기반 라이선스 검증 코드 난독화

Assignment 3

김가람 단국대학교 산업보안학과

Dankook univ.

과목명 : 시큐어코딩
담당교수 : 이승광
학과 : 산업보안학과
학번 : 32230324
이름: 김가람
제출일 : 2025/12/20

Ⅰ. Control-flow 난독화

1. Opaque predicate + Dead code 삽입

먼저 rand() 함수의 반환값에 % 50 연산을 적용한 뒤 +10을 더함으로써 변수 a와 b가 항상 10 이상 59 이하의 양수 값을 갖도록 한다.

```
// secret_check
int secret_check(const char *user, const char *key) {
    srand(time(0));
    int a = (rand() % 50) + 10;
    int b = (rand() % 50) + 10;
    int x = 77;
```

아래 코드는 switch 문 내의 case 256으로 (a * b) > 0 조건문은 Opaque predicate이다. 사람의 관점에서는 앞서 초기화하였던 a와 b가 항상 양수의 값을 가진다는 점을 통해 해당 조건이 항상 참임을 쉽게 판단할 수 있다. 그러나 컴파일러나 정적 분석 도구의 관점에서는 rand() 호출과 산술 연산이 포함되어 있어 참, 거짓을 명확히 판단하기 어렵다. 이로 인해 해당 조건문은 의미 있는 분기처럼 인식되어 실제론 실행되지 않는 Dead Code를 포함시키게 된다.

```
        case 256: { // xor로 복호화
            xor_decode(e_u , 5, K);
            xor_decode(e_k, 23, K);
            e_u[5] = '\0';
            e_k[23] = '\0';

            // Opaque Predicate
            if ((a * b) > 0) {
                // VM-based 난독화
                uintptr_t memory[256] = {0};
                memory[0xA0 - 0xA0] = (uintptr_t)user;
```

... 생략 ...

```
            else {
                x = 999; // Dead code
            }
            goto sw;
        }
```

아래 코드는 switch 문 내부의 case 999로 실제론 실행되지 않는 Dead Code이다. “root”와 “SECURE-CODING-2025”라는 또 다른 사용자 정보와 라이선스 키를 비교하지만 정상 실행 흐름에서는 이 분기로 진입하지 않는다. 공격자가 strings와 같은 명령어로 정적 분석을 수행할 경우 해당 부분을 실제 검증 로직으로 오인할 가능성이 있으며, 이를 통해 분석을 혼란스럽게 만든다. 만약 비정상적인 실행 흐름으로 인해 해당 분기가 실행되더라도 실제 유효한 사용자 정보(admin, SECURE-OBFUSCATION-2025)와 일치하지 않으므로 x가 0으로 설정되어 인증 실패 분기인 case 0으로 이동하게 된다.

```
    case 999: {// Dead Code
        const char *expected_user = "root";
        const char *expected_key = "SECURE-CODING-2025";
        if (strcmp(user, expected_user) == 0 && strcmp(key, expected_key) == 0) {
            x = 0;
        }
        goto sw;
    }
```

2. Control-flow flattening

함수 시작 시 변수 x는 77로 초기화되며 이는 초기 진입점을 의미한다. 즉 switch문 내부의 case 77이 가장 먼저 실행된다.

```
// secret_check
int secret_check(const char *user, const char *key) {
    srand(time(0));
    int a = (rand() % 50) + 10;
    int b = (rand() % 50) + 10;
    int x = 77;
```

아래 코드는 switch 문을 이용한 control-flow Flattening 구조이다. 변수 x의 값에 따라 실행 경로가 결정되며 각 분기 이후에는 goto을 이용해 다시 swtich 문으로 복귀하도록 설계하였다.

case 77에서 난독화되어 저장된 데이터를 재조립하고, 재조립 완료 후 x를 256으로 설정하고 goto를 통해 sw로 복귀하고, case 256에서 xor로 재조립한 데이터들을 복호화 후 x를 82로 설정하고 다시 goto를 통해 sw로 복귀한다. 이후 case 82에서 라이선스 검증 로직을 수행한다. 검증 결과에 따라 x를 0(실패)이나 100(성공)으로 설정하고 goto를 통해 sw로 복귀 후, 인증 성공(case 100) 또는 실패(case 0) 분기로 이동하여 각각 1과 0을 반환하게 된다.

이를 통해 프로그램의 실제 실행 순서는 코드 상의 배치 순서와 무관하게 동작하게 되며 제어 흐름을 직관적으로 파악하기 어렵게 만드는 효과를 가진다.

```
    // Control-flow Flattening
    sw: switch (x) {
        case 0: { // 검증 실패
            return 0;
        }

        case 999: {// Dead Code
            const char *expected_user = "root";
            const char *expected_key = "SECURE-CODING-2025";
            if (strcmp(user, expected_user) == 0 && strcmp(key, expected_key) == 0) {
                x = 0;
            }
            goto sw;
        }
```

```
case 100: { // 검증 성공
    return 1;
}

case 77: { // 루프로 재조립
    int ui = 0, ki = 0;
    for (int i = 0; i < 31; i++) {
        int take_u = ((i % 4) == 0 && ui < 8);
        if (take_u) {
            e_u[ui++] = (char)aggr[i];
        } else {
            e_k[ki++]  = (char)aggr[i];
        }
    }
    x = 256;
    goto sw;
}

case 1: { // Dead Code 종료
    return -1;
}

case 256: { // xor 복호화 및 검증
    xor_decode(e_u , 5, K);
    xor_decode(e_k, 23, K);
    e_u[5] = '\0';
    e_k[23] = '\0';
    x = 82;;
    goto sw;
}
case 82: {
    // Opaque Predicate
    if ((a * b) > 0) {
        // VM-based 난독화
        uintptr_t memory[256] = {0};
        memory[0xA0 - 0xA0] = (uintptr_t)user;
        memory[0xA1 - 0xA0] = (uintptr_t)key;
        memory[0xA2 - 0xA0] = (uintptr_t)e_u;
        memory[0xA3 - 0xA0] = (uintptr_t)e_k;
        memory[0xA4 - 0xA0] = 1;
```

... 생략 ...

```
        int ok = vm_execute(bytecode, sizeof(bytecode), memory);
        if (ok) {
            x = 100;
        }
        else {
            x = 0;
        }
    }
    else {
        x = 999; // Dead code
    }
    goto sw;
}
```

```
        default: {
            return 0;
        }

    }
}
```

II . Data 난독화

1. 문자열 암호화 : XOR 인코딩

문자열 데이터를 보호하기 위해 xor 연산을 이용한 데이터 난독화 기법을 적용하였다. xor 연산은 동일한 키로 두 번 연산할 경우 원래 값으로 복원되는 특성을 가진다. 이를 이용해 xor_decode 함수에서는 암호화된 문자열의 각 바이트에 동일한 키 값을 다시 XOR 연산함으로써 원래의 문자열을 복원한다. 이러한 복호화 과정은 런타임에섬나 발생하므로 정적 분석 단계에서는 실제 문자열 값을 확인하기 어렵다.

```
// xor 복호화 함수
// 문자열의 각 바이트를 xor 키(k)로 다시 xor 연산하여 복호화
static void xor_decode(char *s, size_t n, uint8_t k) {
    for (size_t i = 0; i < n; i++) {
        s[i] ^= k;
    }
}
```

문자열을 평문 형태로 저장하지 않고 각 바이트를 XOR 키 0x5A와 연산한 값들을 aggr 배열에 저장되도록 하였다. 이로 인해 strings와 같은 정적 분석 도구를 사용하더라도 사용자 ID나 라이선스 키를 직접적으로 확인할 수 없게 된다.

```
    const uint8_t K = 0x5A; // xor 키

    // data aggregation + xor 인코딩
    static const uint8_t aggr[31] = {
        ('a' ^ 0x5A), ('S' ^ 0x5A), ('E' ^ 0x5A), ('C' ^ 0x5A),
        ('d' ^ 0x5A), ('U' ^ 0x5A), ('R' ^ 0x5A), ('E' ^ 0x5A),
        ('m' ^ 0x5A), ('-' ^ 0x5A), ('O' ^ 0x5A), ('B' ^ 0x5A),
        ('i' ^ 0x5A), ('F' ^ 0x5A), ('U' ^ 0x5A), ('S' ^ 0x5A),
        ('n' ^ 0x5A), ('C' ^ 0x5A), ('A' ^ 0x5A), ('T' ^ 0x5A),
        (0 ^ 0x5A), ('I' ^ 0x5A), ('O' ^ 0x5A), ('N' ^ 0x5A),
        (0 ^ 0x5A), ('-' ^ 0x5A), ('2' ^ 0x5A), ('0' ^ 0x5A),
        (0 ^ 0x5A), ('2' ^ 0x5A), ('5' ^ 0x5A)
    };

    char e_u[9];
    char e_k[24];
```

2. Data aggregation

아래 코드는 사용자 ID와 라이선스 키를 그대로 저장하지 않고 배열 aggr 안에 분해된 형태로 혼합하여 저장하는 Data aggregation 기법을 적용한 부분이다. 문자열의 각 문자는 인덱스 기준으로 i % 4 == 0과 같은 규칙을 적용하여 특정 위치의 값만 사용자 ID, 그 외는 라이선스 키에 해당하도록 설계하였다. 또한 사용자 ID(admin)와 라이선스 키(SECURE-OBFUSCATION-2025)는 문자열 길이가 서로 다르기에 두 문자열을 하나의 배열로 통합하는 과정에서 일부 의미 없는 더미 값을 함께 삽입하였다. 이러한 더미 값의 존재로 인해 사용자 ID를 저장하는 버퍼 e_u 역시 실제 문자열 길이보다 크게 9바이트로 할당하였다. 이를 통해 공격자는 단순히 배열만 보고 원문을 추론하기 어렵게 된다.

```
// secret_check
int secret_check(const char *user, const char *key) {
    srand(time(0));
    int a = (rand() % 50) + 10;
    int b = (rand() % 50) + 10;
    int x = 77;

    const uint8_t K = 0x5A; // xor 키

    // data aggregation + xor 인코딩
    static const uint8_t aggr[31] = {
        ('a' ^ 0x5A), ('S' ^ 0x5A), ('E' ^ 0x5A), ('C' ^ 0x5A),
        ('d' ^ 0x5A), ('U' ^ 0x5A), ('R' ^ 0x5A), ('E' ^ 0x5A),
        ('m' ^ 0x5A), ('-' ^ 0x5A), ('O' ^ 0x5A), ('B' ^ 0x5A),
        ('i' ^ 0x5A), ('F' ^ 0x5A), ('U' ^ 0x5A), ('S' ^ 0x5A),
        ('n' ^ 0x5A), ('C' ^ 0x5A), ('A' ^ 0x5A), ('T' ^ 0x5A),
        (0 ^ 0x5A), ('I' ^ 0x5A), ('O' ^ 0x5A), ('N' ^ 0x5A),
        (0 ^ 0x5A), ('-' ^ 0x5A), ('2' ^ 0x5A), ('0' ^ 0x5A),
        (0 ^ 0x5A), ('2' ^ 0x5A), ('5' ^ 0x5A)
    };

    char e_u[9];
    char e_k[24];
```

다음은 switch문의 case 77로 앞서 aggr 배열에 혼합하여 저장한 데이터를 루프를 통해 다시 e_u(사용자)와 e_k(키)로 재조립하는 단계이다. 반복문 내부에서 take_u 조건을 이용해 특정 인덱스(i % 4 == 0)의 값만 e_u에 저장하고 나머지는 e_k에 저장하도록 하였다.

```
        case 77: { // 루프로 재조립
            int ui = 0, ki = 0;
            for (int i = 0; i < 31; i++) {
                int take_u = ((i % 4) == 0 && ui < 8);
                if (take_u) {
                    e_u[ui++] = (char)aggr[i];
                } else {
                    e_k[ki++]  = (char)aggr[i];
                }
            }
            x = 256;
            goto sw;
        }
```

III . VM-based 난독화

라이선스 검증 로직의 핵심 부분을 일반적인 C 코드가 아닌 사용자 정의 가상 머신 위에서 실행되도록 구성한 부분이다. 바이트코드 형태로 정의된 명령어를 vm_execute 함수가 해석하고 실행하는 방식으로 동작한다.

vm_execute 함수는 가상 레지스터 배열(reg), 가상 메모리(memory), 프로그램 카운터(pc), zflag로 구성된 구조를 가진다. 또한 각 명령어는 1바이트 opcode와 그에 따른 피연산자로 구성하였는데, 아래와 같다.

· 0x01 (LOAD) : 가상 메모리에 저장된 값을 지정한 가상 레지스터로 로드한다.
· 0x02 (STORE) : 가상 레지스터에 저장된 값을 가상 메모리의 저장된 위치에 저장한다.
· 0x03 (ADD) : 두 개의 가상 레지스터 값을 더하여 결과를 첫 번째 레지스터에 저장한다.
· 0x04 (SUB) : 두 개의 가상 레지스터 값을 빼 첫 번째 레지스터에서 두 번째 레지스터 값을 차감한다.
· 0x05 (CMPSTR) : 두 가상 레지스터가 가리키는 문자열을 strcmp로 비교한다. 문자열이 동일할 경우 플래그(zflag)를 설정한다.
· 0x06 (JZ) : 플래그(zflag)가 설정된 경우 프로그램 카운터를 지정한 위치로 이동시킨다.
· 0xFF (RETURN) : 지정한 가상 레지스터의 값을 반환하고 VM 실행을 종료한다.

```c
// VM-based 난독화
static int vm_execute(const uint8_t *code, size_t code_len, uintptr_t *memory) {
    uintptr_t reg[16] = {0};
    uint8_t pc = 0;
    uint8_t zflag = 0;

    while (pc < code_len) {
        switch (code[pc]) {
            case 0x01: { // LOAD
                uint8_t reg_index = code[++pc];
                uint8_t mem_address = code[++pc] - 0xA0;
                reg[reg_index] = memory[mem_address];
            }
            break;

            case 0x02: { // STORE
                uint8_t mem_address = code[++pc] - 0xA0;
                uint8_t reg_index = code[++pc];
                memory[mem_address] = (uintptr_t)reg[reg_index];
            }
            break;

            case 0x03: { //ADD
                uint8_t reg_index1 = code[++pc];
                uint8_t reg_index2 = code[++pc];
                reg[reg_index1] += reg[reg_index2];
            }
            break;
```

```
            case 0x04: { // SUB
                uint8_t reg_index1 = code[++pc];
                uint8_t reg_index2 = code[++pc];
                reg[reg_index1] -= reg[reg_index2];
            }
            break;

            case 0x05: { // CMPSTR
                uint8_t reg_index1 = code[++pc];
                uint8_t reg_index2 = code[++pc];
                const char *str1 = (const char *)reg[reg_index1];
                const char *str2 = (const char *)reg[reg_index2];
                if (str1 && str2 && strcmp(str1, str2) == 0) {
                    zflag = 1;
                } else {
                    zflag = 0;
                }
            }
            break;

            case 0x06: { // target
                uint8_t target_address = code[++pc];
                if (zflag) {
                    pc = target_address - 1;
                }
            }
            break;

            case 0xFF: { // RETURN
                uint8_t reg_index = code[++pc];
                return (int)reg[reg_index];
            }

            default: {
                return 0;
            }
        }
        pc++;
    }
    return 0;
}
```

바이트 코드 실행에 앞서 가상 머신이 참조할 데이터들을 먼저 준비해야 한다. 가상 머신에서 사용할 가상 메모리 배열(memory)을 초기화 후, 입력된 사용자명(user), 입력된 라이선스 키, 앞서 복호화된 문자열(e_u, e_k)과 성공 여부를 나타내기 위한 상수 값 1을 적재한다. 이후 정의된 바이트코드(bytecode)가 프로그램 카운터를 기준으로 순차적으로 해석, 실행된다. 먼저 사용자 입력과 기준 문자열의 주소가 LOAD 명령어를 통해 가상 레지스터에 적재하고 CMPSTR 명령어를 통해 사용자명에 대한 비교를 수행한다. 비교 결과가 일치할 경우 조건 분기 명령(JZ)을 통해 라이선스 키 검증 단계로 이동하며 일치하지 않을 경우 즉시 실패 값을 반환하도록 설계하였다. 사용자명 검증 통과 후 라이선스 키 검증도 통과하면 성공 값을 반환하고 가상 머신의 실행이 종료된다. 이러한 VM 기반 난독화를 통한 검증 흐름은 정적 분석 시 전체 로직을 직관적으로 파악하기 어렵게 만든다는 장점이 있다.

```
        case 82: {
            // Opaque Predicate
            if ((a * b) > 0) {
                // VM-based 난독화
                uintptr_t memory[256] = {0};
                memory[0xA0 - 0xA0] = (uintptr_t)user;
                memory[0xA1 - 0xA0] = (uintptr_t)key;
                memory[0xA2 - 0xA0] = (uintptr_t)e_u;
                memory[0xA3 - 0xA0] = (uintptr_t)e_k;
                memory[0xA4 - 0xA0] = 1;

                uint8_t bytecode[] = {
                    // user_check:
                    0x01, 0x00, 0xA0, // LOAD user -> r0
                    0x01, 0x01, 0xA2, // LOAD e_u -> r1
                    0x05, 0x00, 0x01, // CMPSTR r0, r1
                    0x06, 0x0D,       // JZ 0x0D (user_ok로 점프)
                    0xFF, 0x06, // RETURN r6 (0)
                    // user_ok:
                    0x01, 0x03, 0xA1, // LOAD key -> r3
                    0x01, 0x04, 0xA3, // LOAD e_k -> r4
                    0x05, 0x03, 0x04, // CMPSTR r3, r4
                    0x06, 0x1A,       // JZ 0x1A (key_ok로 점프)
                    0xFF, 0x06, // RETURN r6 (0)

                    // key_ok:
                    0x01, 0x06, 0xA4, // LOAD 1 -> r6
                    0xFF, 0x06  // RETURN r6 (1)
                };

                int ok = vm_execute(bytecode, sizeof(bytecode), memory);
                if (ok) {
                    x = 100;
                }
                else {
                    x = 0;
                }
            }
            else {
                x = 999; // Dead code
            }
            goto sw;
        }

        default: {
            return 0;
        }

    }
}
```

IV . Makefile

· gcc 사용, -Wall / -Wextra 옵션으로 경고 활성화, -O2로 최적화
· 생성될 실행 파일 이름 : license_obf, 소스 파일 : license_obf.c, 오브젝트 파일 : license_obf.o

· make : 빌드
· make run : 빌드된 실행 파일을 바로 실행
· make clean : 생성된 실행 파일과 오브젝트 파일 삭제하여 빌드 환경 초기화

```
CC      := gcc
CFLAGS  := -Wall -Wextra -O2 -std=c11
LDFLAGS :=

TARGET  := license_obf
SRC     := license_obf.c
OBJ     := $(SRC:.c=.o)

.PHONY: all clean run debug

all: $(TARGET)

$(TARGET): $(OBJ)
        $(CC) $(LDFLAGS) -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<

run: $(TARGET)
        ./$(TARGET)

clean:
        rm -f $(TARGET) $(OBJ)
```

V . 결과

- Dead Code

objdump를 통해 바이너리를 분석한 결과, 아래와 같은 부분을 볼 수 있는데 본 프로그램에서 변수 a 와 b는 각각 10 이상인 난수로 초기화되므로 곱셈 결과 a * b는 항상 양수가 된다. 따라서 조건 분기인 jg 14d7은 항상 참이 되어 해당 주소로 점프하게 되며 그 결과 아래에 위치한 strcmp 기반 사용자 및 라이선스 키 비교 로직은 실제로 실행되지 않지만 Opaque predicate로 인해 코드 상에는 Dead Code로 남아 있음을 확인할 수 있다.

```
1475:       41 0f af d9             imul   ebx,r9d
1479:       85 db                   test   ebx,ebx
147b:       7f 5a                   jg     14d7 <secret_check+0x1a7>
147d:       48 8d 35 80 0b 00 00    lea    rsi,[rip+0xb80]        # 2004 <_IO_stdin_used+0x4>
1484:       48 89 ef                mov    rdi,rbp
1487:       e8 74 fc ff ff          call   1100 <strcmp@plt>
148c:       85 c0                   test   eax,eax
148e:       75 40                   jne    14d0 <secret_check+0x1a0>
1490:       48 8d 35 72 0b 00 00    lea    rsi,[rip+0xb72]        # 2009 <_IO_stdin_used+0x9>
1497:       4c 89 e7                mov    rdi,r12
149a:       e8 61 fc ff ff          call   1100 <strcmp@plt>
149f:       85 c0                   test   eax,eax
14a1:       75 2d                   jne    14d0 <secret_check+0x1a0>
```

- strings

strings 명령어를 통해 바이너리 내 문자열을 확인한 결과 실제 검증에 사용되는 문자열은 보이지 않고, Dead Code에 넣어둔 미리 문자열(root, SECURE-CODING-2025)만 보이는 것을 확인할 수 있다.

```
root
SECURE-CODING-2025
User name:
License key:
Access granted!
Access denied.
```

- 실행 결과

프로그램을 실행하여 사용자 이름 admin, 라이선스 키 SECURE-OBFUSCATION-2025를 입력한 결과, 정상적으로 인증에 성공하였다. 반면, 사용자 이름이나 라이선스 키를 잘못 입력할 경우 모두 인증에 실패함을 확인할 수 있다.

```
user32230324@user32230324:~/바탕화면/SC/hw3$ make run
./license_obf
User name: admin
License key: SECURE-OBFUSCATION-2025
Access granted!
user32230324@user32230324:~/바탕화면/SC/hw3$ make run
./license_obf
User name: admin
License key: hi
Access denied.
user32230324@user32230324:~/바탕화면/SC/hw3$ make run
./license_obf
User name: hi
License key: SECURE-OBFUSCATION-2025
Access denied.
```