Code Obfuscation Assignment

Date : 27 – 09 - 2025

Raj Shekhar - 240545002004

An analysis report on a C program that implements string obfuscation techniques to protect sensitive data. The analysis represents insights into basic software protection mechanisms and their limitations in real-world security scenarios.

```
static void secure_zero(void *p, size_t n)
{
   volatile unsigned char *vp = (volatile unsigned char *)p;
   while (n--)
      *vp++ = 0;
}
```

Fig1: Secure Zero function implementation

```
static const uint8_t enc0[] = {'M' ^ 0x5A, 'e' ^ 0x5A, 'e' ^ 0x5A, 'n' ^ 0x5A, 'a' ^ 0x5A};

static const uint8_t enc1[] = {'H' ^ 0x1B, 'e' ^ 0x1B, 'l' ^ 0x1B, 'l' ^ 0x1B, 'o' ^ 0x1B, ',' ^ 0x1B, ' ' ^ 0x1B};

static const uint8_t enc2[] = {'A' ^ 0x79, 'P' ^ 0x79, 'I' ^ 0x79, '_' ^ 0x79, 'K' ^ 0x79, 'E' ^ 0x79, 'Y' ^ 0x79,
'=' ^ 0x79, 'a' ^ 0x79, 'b' ^ 0x79, 'c' ^ 0x79, 'd' ^ 0x79, '-' ^ 0x79, '1' ^ 0x79, '2' ^ 0x79, '3' ^ 0x79,
'4' ^ 0x79, '-' ^ 0x79, 'X' ^ 0x79, 'Y' ^ 0x79, 'Z' ^ 0x79};
```

Fig2: enc0, en1, enc2 static const variables of 8 bit unsigned integer each

```
typedef struct
{
   const uint8_t *data;
   size_t len;
   uint8_t key;
   char *cached;
} obf_str_t;
static obf_str_t OBF_TABLE[] = {
   {enc0, sizeof(enc0), 0x5A, NULL}, /* index 0 = "Meena" */
   {enc1, sizeof(enc1), 0x1B, NULL}, /* index 1 = "Hello, " */
   {enc2, sizeof(enc2), 0x79, NULL}  /* index 2 = API key */
};
```

Fig3: Defined struct and initialised values in OBF_TABLE static variable

The provided C program demonstrates a sophisticated approach to string obfuscation that combines compile-time encoding with runtime decoding. The core functionalities are as follows :

1. XOR-based string obfuscation
2. Lazy decoding with caching
3. Secure memory management
4. Protection of sensitive information (API keys, passwords, etc)

```
static char *obf_get(size_t idx)
{
  if (idx >= OBF_COUNT)
    return NULL;
  obf_str_t *rec = &OBF_TABLE[idx];
  if (rec->cached)
    return rec->cached; /* return cached decoded string */
  char *out = (char *)malloc(rec->len + 1);
  if (!out)
    return NULL;
  for (size_t i = 0; i < rec->len; ++i)
  {
    out[i] = (char)(rec->data[i] ^ rec->key);
  }
  out[rec->len] = '\0';
  rec->cached = out;
  return out;
}
```

Fig4: A static function which returns string

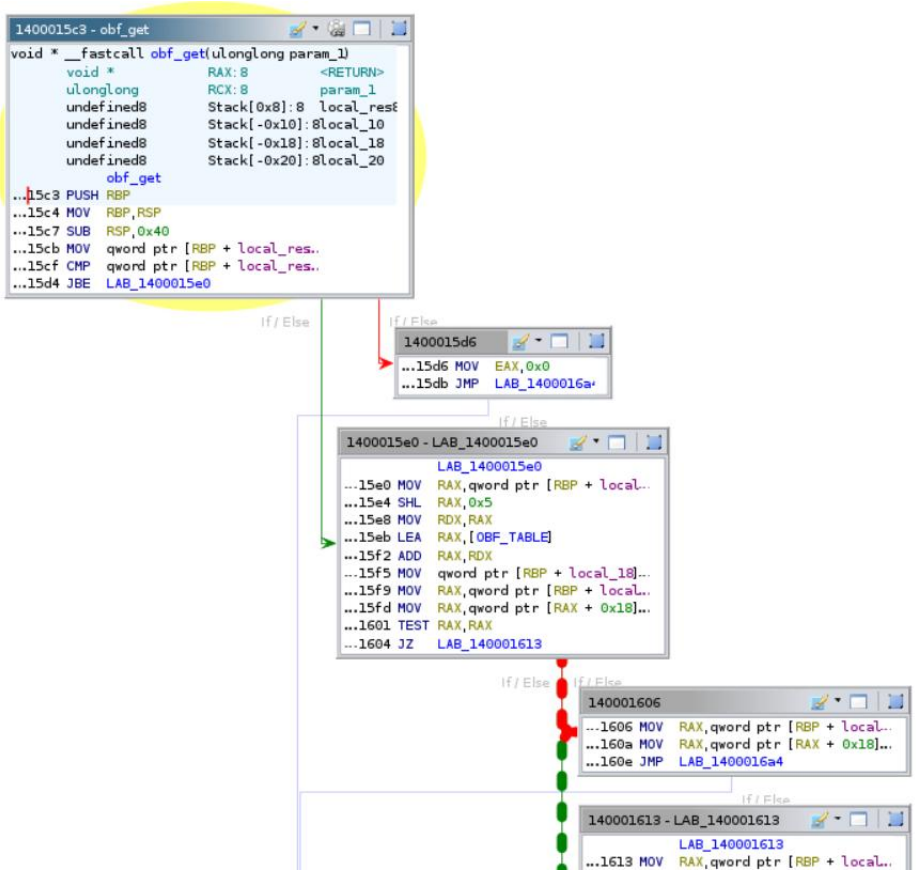

Fig5: Code execution flow of the program 1st half in Ghidra

Code Spotted #1: XOR-based string obfuscation

out[i] = (char)(rec->data[i] ^ rec->key);

Analysis :

- XOR operation properties (reversible nature)

- Static storage duration implications
- Constant declaration

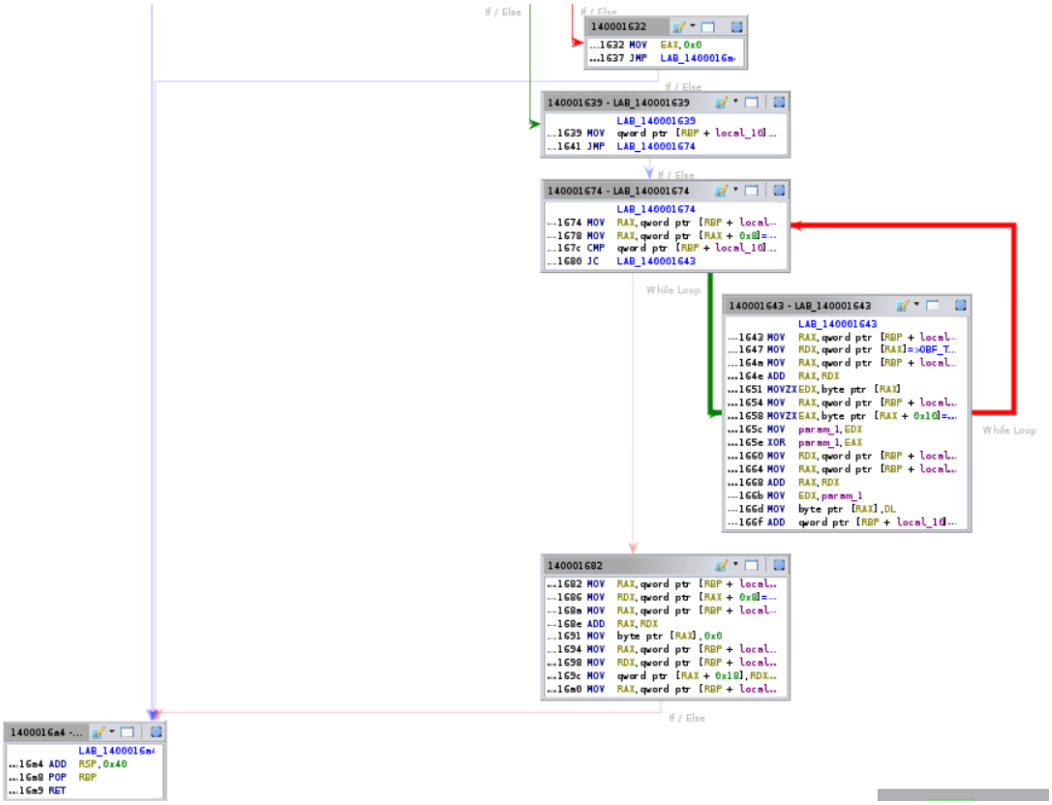

Fig6: Program execution Flow Chart 2nd half

Code Spotted #2: Secure memory management

```
static void secure_zero(void *p, size_t n)
{
    volatile unsigned char *vp = (volatile unsigned char *)p;
    while (n--)
        *vp++ = 0;
}
```

Analysis :

- Volatile keyword used to prevent compiler optimization
- Provides defenses against cold boot attacks
- Timing of memory cleanup is important for memory-mapped hardware, shared memory, or clearing sensitive data (like passwords or keys).

Code Spotted #3: Lazy decoding with caching

```
if (rec->cached)
    return rec->cached;
```

Analysis :

- If it's NULL → false.
- If it points to something → true.

- Performance optimization used to avoids decoding the same string multiple times unnecessarily.

Code Spotted #4: Protection of sensitive information
static const uint8_t enc0[ ] = {'M' ^ 0x5A, 'e' ^ 0x5A, 'e' ^ 0x5A, 'n' ^ 0x5A, 'a' ^ 0x5A};

Analysis :

- Data is embedded in the binary at compile time.
- Dumping the binary or searching with strings won't reveal the real string or key.
- Unsigned Integer type which can only hold 0-255 and accessible globally but cannot be modified.

Encoded with keys :

enc0 with key 0x5A

enc1 with key 0x1B

enc2 with key 0x79

It has a multi-layered approach that demonstrates fundamental principles of software security while highlighting the trade-offs between protection strength and implementation complexity. The XOR encoding mechanism operates by applying a bitwise exclusive OR operation between each character of the plaintext string and a predetermined single-byte key.

(plaintext ^ key) ^ key = plaintext

The obf_str_t structure serves as a centralized management system for obfuscated strings, containing four essential components: a pointer to the encoded data, the length of the encoded data, the XOR key for decoding, and a cached pointer to the decoded string. And it enables efficient memory management by allowing decoded strings to persist in memory until explicitly cleared, reducing redundant decoding operations.

The obf_get() function implements lazy decoding through a conditional caching mechanism that defers actual decoding until the moment a string is requested. If cached data exists, it immediately returns the pointer, avoiding unnecessary processing.

Also, the static const uint8_t declaration serves multiple purposes in enhancing both security and efficiency.

Using uint8_t guarantees precise 8-bit unsigned integer representation across different platforms, ensuring consistent XOR operations regardless of the underlying architecture.

Therefore, a true security requires a layered approach combining multiple techniques rather than using just a single method.