Assignment 3: Uncover Hidden Artifacts from Malware

Context:

You are a cyber forensic investigator. You obtain a criminal's computer and locate a suspicious program. When you scan the program with an anti-virus program, it reports nothing. Then, you look at the binary. You find that the code is obfuscated, and all the constant strings are somewhat unnatural (doesn't look like normal strings). You look at closer, and you find out that whenever the program is using the string, there is a common code snippet used to modify the string before it is used. For example, "fopen(decode("some_weird_text"), ..." You realize that by applying the same logic of the "decode()" function, you can successfully decode "some_weird_text." However, you find that there are so many those strings (e.g., more than a thousand strings are like that), making it difficult to do automatically.

This assignment asks you to (1) locate those weird strings, (2) identify the function like "decode" that would give you instructions how to decode the weird strings, (3) construct a logic from the logic of the identified "decode()" function, and (4) apply the logic and decode the strings.

Description:

You are given two sample programs' bitcode files: sample1.bc and sample2.bc. Note that source code files for them are also provided. However, ***your goal is to create an LLVM tool that operates on the bitcode files***. Source code files are only provided to help you to understand the sample.

```
extern "C" char* decode(char* s)
{
    strcpy( g_szTemp, s );

    for( int i = 0; i < 200; i++ ) {
        if( g_szTemp[i] != 0 ) {
            g_szTemp[i] -= 5;
        }
    }
    return g_szTemp;
}

int main()
{
    puts(decode("r{%~tzwdknqj%4ij{4szqq"));

    return 0;
}
```

As shown in the above code, the string used in the decode() function is encoded. The decoding logic is shown in the above. Imagine this is a malware that you want to know the string value after decoding, and you have a number of those samples with different encoding schemes. To handle this, you need to build an automated tool.

Real-world context
This is a typical concern in the software industry. For instance, when you develop an Apple App, you are allowed to use any APIs provided by OS. Unfortunately, some of those are security sensitive, and Apple banned some of those. Any apps using those banned APIs cannot be registered on the Apple Store. How do they do? There are so many apps registered every day. The answer is simple. They have an automated scanner to find out apps that are using those banned APIs. Well, it sounds great. However, the problem did not end there. Apple later finds that many apps try to use banned APIs via dynamic loading functionalities. Specifically, iOS apps allow developers to use a string to execute a function. Apple then scans the strings in the apps to ban them again. Finally, developers encode strings to avoid detection. *How do they encode? Similar to the decode function above.*

A human can easily identify the decoding scheme from source code. Since our goal is to build an automated tool, we are following how a human would reason the decoding scheme and write code that follows the reasoning process.

Step 1: Understanding the Return and Its Source. Since we know the decode()'s return would be the real string, we see how the decode()'s return is constructed. First, we find the return is g_szTemp. In addition, we analyze that the return value is originated from the argument of the decode function.

```
extern "C" char* decode(char* s)
{
    strcpy( g_szTemp, s );

    for( int i = 0; i < 200; i++ ) {
        if( g_szTemp[i] != 0 ) {
            g_szTemp[i] -= 5;
        }
    }
    return g_szTemp;
}
```

Step 2: Understanding the Computation. Then we check how it is computed. We focus on "write" operations on the return value.

```
extern "C" char* decode(char* s)
{
    strcpy( g_szTemp, s );

    for( int i = 0; i < 200; i++ ) {
        if( g_szTemp[i] != 0 ) {
            g_szTemp[i] -= 5;
        }
    }
    return g_szTemp;
}
```

Step 3: Condition and Index. The operation may not always be executed. We find that it only works when the value is not 0. In addition, we check the range of i. From the for loop, the range is 0 to 200, and i is incremented by one in each iteration.

```
extern "C" char* decode(char* s)
{
    strcpy( g_szTemp, s );

    for( int i = 0; i < 200; i++ ) {
        if( g_szTemp[i] != 0 ) {
            g_szTemp[i] -= 5;
        }
    }
    return g_szTemp;
}
```

With those findings, one can automatically construct the following logic: The decode function takes a string, and for each character, if it is not 0, it will subtract the value by 5. It does for elements range from 0 to 199.

To write an automated tool to this, you may need to write a tool as follows.
First, you identify the return value. Then, you check all the code that is touching the return value. By analyzing them, you will know the decoding logic. In some cases, if there are additional computations used in the decoding logic (e.g., computation of [i] in this case), you need to also track them and consider them in the computation.
By keep tracking them until you reach the source of all those variables. In this case, the source is the input argument (char* s).

What to do:

1. Download the LLVM and install it – check the lecture slides for this.
2. Download the provided sample1 and sample2. (There are two more samples. If you are taking 6501 and you did take Software Security via Program analysis in 2019 Fall, you need to do sample3 and sample4. Otherwise you do not need to do sample 3 and 4. Of course if you do, those will be extra credits).
3. Make your own LLVM tool (module) to analyze the bitcode to understand the logic of decode() function in the sample1 and sample2. Then, your tool should identify all the global strings, and apply the extracted logic to decode the strings. The tool should print out all the decoded strings.
4. Write the report as described below.

Do not do:

1. Decoding logic is shown in the source code that is provided. The goal is not to know the decoding logic (which is already known) but to build an automated tool. Hence, you can't hardcode the decoding logic in the LLVM tool. You have to extract it from the bitcode file. I will test another sample to make sure your code works.
2. Answers that do not leverage your LLVM module with automated analysis as you asked to conduct will not be considered.

What to submit?

1. Your LLVM module. (**Submit a single .cpp file please**)
2. A report that includes

(1) high-level descriptions of how your LLVM module works (around, **1 page**) (20%),

(2) how did you identify data dependency between the argument and return values of the decode function (around **0.25 page**) (15%),

(3) how did you identify range of the index (around **0.25 page**) (15%)

(4) how did you identify the decoding computation (around **0.25 page**) (15%)

(5) how did you handle variable aliasing (around **1 page**) (25%)

(6) how did you stitch all the above logics together? (10%)

- Please submit only two files: **(1) .cpp file**, and **(2) .pdf file**.