

Final Project Report
-How to use ChatGPT to analyze codes efficiently
by Prompt Engineering

Background:

While I was initially tasked with creating a tool to assess code coverage, I encountered a roadblock. The current tools in the market seemed quite effective, leaving little room for improvement. Amidst my confusion over this dilemma, a friend sought my assistance in debugging his code. Aware of ChatGPT's impressive capabilities, I suggested he consult ChatGPT for solutions. However, he expressed his frustration, having already exhausted his efforts in that direction. Although slightly impatient, I set aside my code coverage project to aid him. This decision ultimately led to a pivotal shift in my perspective on choosing my project topic. As I observed my friend's interaction with GPT, I noticed that he was relying on it to identify potential bugs. This observation sparked a novel idea in my mind – why not explore ways to enhance the efficiency of utilizing ChatGPT in programming environments?

Motivation:

The project's motivation stems from recognizing the limitations of conventional code analysis methods, like Randoop and fuzz testing. These traditional approaches often fall short in efficiency due to three key drawbacks: their inherent randomness, the absence of logical explanations for their findings, and constraints imposed by specific programming languages. This project proposes a novel approach: leveraging ChatGPT as an alternative tool for code analysis. The central question driving this project is not just about integrating ChatGPT into the analysis process, but rather, how to optimize its use to ensure it outperforms traditional methods. We aim to explore strategies and frameworks that can effectively utilize ChatGPT's capabilities, ensuring that it provides a more efficient, logical, and versatile solution for code analysis than its predecessors.

Methodology:

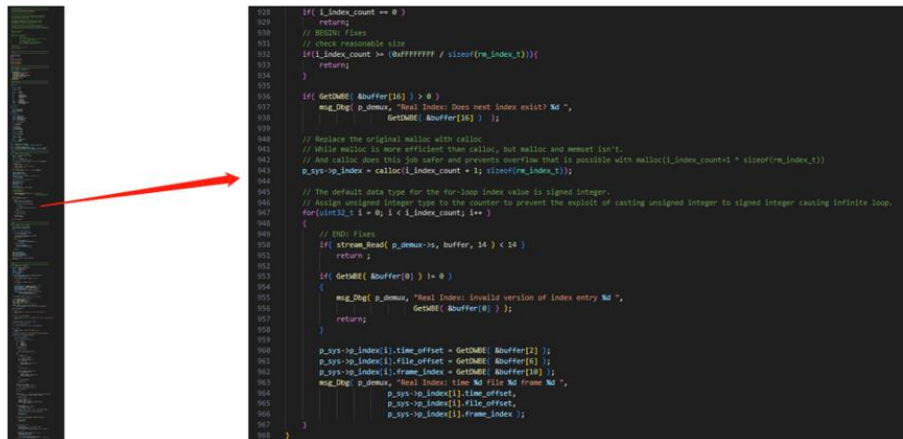
1. General steps:

- (a)File Analysis
- (b)Prompt Generation
- (c)Output generated by LLM.

2. Description: File Analysis is like linting, it leads to understand the class and figure out what is related to the class (e.g., other class or packages). If we are not using the strong-type programming language (such as C++), we only get to know what the type of the data is in runtime. Prompt generation aims to solve this problem.

Implementation:

1. File Analysis:



Like linting, it leads to understand the class and figure out what is related to the class(e.g. other class or packages).

```

668
669 #define GetWBE( p )    U16_AT( p )
670 #define GetDWBE( p )  U32_AT( p )
671 #define GetQWBE( p )  U64_AT( p )

```

From the screenshot, we can see that the program was analyzed manually, and the picture below is an example of “what is related to the class”.

2. Prompt Generation & Output generated by LLM

(a)

Anonymous

The following C program may have vulnerabilities/bugs inside.

```

static void ReadRealIndex( demux_t *p_demux )
{
    demux_sys_t *p_sys = p_demux->p_sys;
    uint8_t buffer[100];
    uint32_t i_id;
    uint32_t i_size;
    int i_version;
    unsigned int i;

    uint32_t i_index_count;

    if ( p_sys->i_index_offset == 0 )
        return;

    stream_Seek( p_demux->s, p_sys->i_index_offset );

    if ( stream_Read( p_demux->s, buffer, 20 ) < 20 )
        return;

    i_id = VLC_FOURCC( buffer[0], buffer[1], buffer[2], buffer[3] ); // 4 characters
    i_size = GetDWBE( &buffer[4] ); // U32_AT
    i_version = GetWBE( &buffer[8] ); // U16_AT

    msg_Dbg( p_demux, "Real index %4.4s size=%d version=%d",
              (char*)&i_id, i_size, i_version );

    if ( (i_size < 20) && (i_id != VLC_FOURCC('T','V','D','X')) )
        return;

```

- Buffer Overflow:**
 - The function uses a fixed-size buffer `buffer[100]` for reading data. Ensure that the data read from the stream doesn't exceed the buffer size to prevent buffer overflows.
- Memory Leak:**
 - The function allocates memory for `p_sys->p_index` using `malloc` but doesn't free it anywhere. Make sure to free this memory when it's no longer needed to avoid memory leaks.
- Error Handling:**
 - The function lacks proper error handling. For example, when memory allocation fails (`malloc` returns `NULL`), it logs an error but continues execution. It would be better to handle such errors more gracefully, perhaps by returning an error code or cleaning up resources before returning.
- Type Mismatch:**
 - The loop counter `i` is declared as an unsigned integer, but it's compared with `i_index_count`, which is of type `uint32_t`. Consider making `i` of type `uint32_t` as well to avoid any potential comparison issues.
- Magic Numbers:**
 - The function contains magic numbers (e.g., `20`, `14`, etc.) that may be unclear to someone reading the code. Consider using named constants or `#define` statements to give them meaningful names.
- Incomplete Data Validation:**
 - The function checks if `i_size` is less than 20 and `i_id` is not `INDEX`, but it doesn't handle other potential issues in the data (e.g., invalid version). You may want to add more comprehensive validation checks.
- Code Duplication:**
 - The code that reads data from the stream and extracts values into variables is repeated. Consider refactoring this code into a separate function to improve maintainability.
- Logging:**
 - Ensure that log messages provide sufficient information for debugging. For example, log the stream position or any other relevant details when an error occurs.

This is a super simple prompt, and the screenshot indicates that there is no constraints on the prompt.

Analysis Report:

1. It is Stack Buffer Overflow
2. It is Heap Buffer Overflow
3. We can ignore Error handling for malloc to return null
4. And forth one is also related to the heap buffer overflow attack.
5. The fifth one occurs because ChatGPT does not have the context of our code so we

can also ignore this. And so does 6 and 7.

6. The logging issue is a generic output from ChatGPT. It is there for liability reasons.

(b)

You

The following C program may have vulnerabilities/bugs inside. Requirements: Nothing will make the size of buffer[] more than 100, so it is unnecessary to throw buffer overflow as a bug.

```
static void ReadRealIndex( demux_t *p_demux )
{
    demux_sys_t *p_sys = p_demux->p_sys;
    uint8_t  buffer[100];
    uint32_t i_id;
    uint32_t i_size;
    int      i_version;
    unsigned int i;

    uint32_t i_index_count;

    if ( p_sys->i_index_offset == 0 )
        return;

    stream_Seek( p_demux->s, p_sys->i_index_offset );

    if ( stream_Read( p_demux->s, buffer, 20 ) < 20 )
        return;

    i_id = VLC_FOURCC( buffer[0], buffer[1], buffer[2], buffer[3] ); // 4 characters
    i_size = GetDWBE( &buffer[4] ); // U32_AT
    i_version = GetWBE( &buffer[8] ); // U16_AT

    msg_Dbg( p_demux, "Real index %4s size=%d version=%d",
              (char*)&i_id, i_size, i_version );

    if ( (i_size < 20) && (i_id != VLC_FOURCC('N','D','X')) )
        return;

    i_index_count = GetDWBE( &buffer[10] ); // U32_AT. Supplied by user
    data, entry point. The 10,11,12,13 bytes are the ...index count data.
```

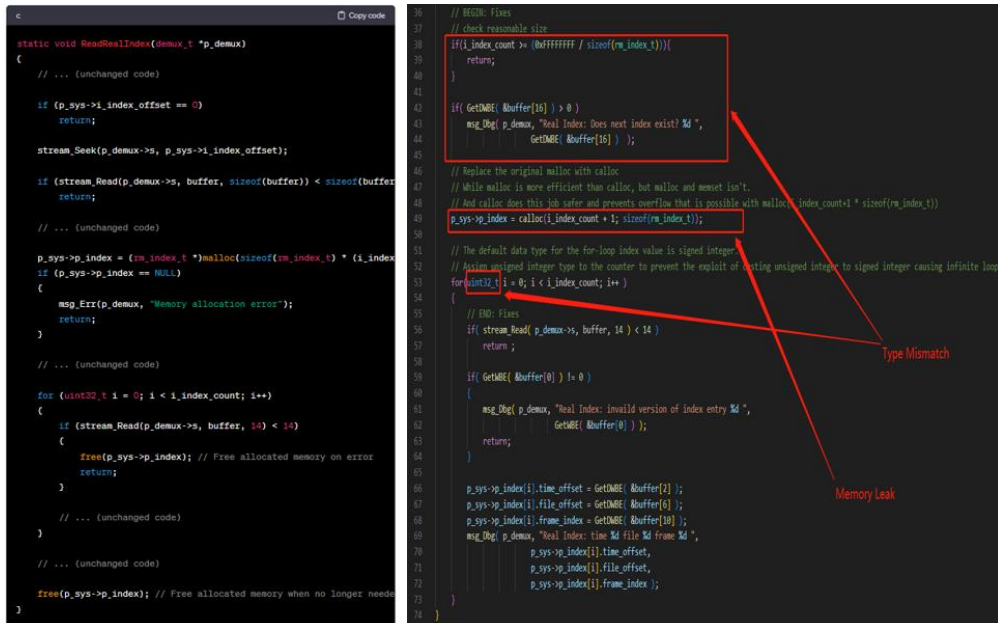
1. **Unchecked Return Value:** The `stream_Seek()` function is called without checking its return value. If `stream_Seek()` fails, it may leave the stream in an inconsistent state, leading to undefined behavior in subsequent operations.
2. **Integer Overflow:** The expression `sizeof(m_index_t) * (i_index_count+1)` is vulnerable to integer overflow. If `i_index_count` is large enough, this expression could overflow, resulting in a smaller-than-expected allocation and potentially a buffer overflow when writing to `p_sys->p_index`.
3. **Unchecked User Data:** The `i_index_count` variable is directly derived from user-supplied data (`GetDWBE(&buffer[10])`). This value is not checked for sanity before being used to calculate memory allocation and control the loop. If it's extremely large, it can lead to the aforementioned integer overflow or cause other resource exhaustion issues.
4. **Potential Heap Buffer Overflow:** In the loop, `i` is of type `unsigned int`, and `i_index_count` is also an unsigned value. However, if `i_index_count` is very large (close to `UINT32_MAX`), the loop condition `i < i_index_count` will almost always be true, potentially leading to a heap buffer overflow when accessing `p_sys->p_index[i]`.
5. **Error Handling:** The function does not provide a means to convey to the caller that an error occurred (other than the initial stream read checks). This can be problematic in scenarios where it's essential to know whether the operation was successful.
6. **Memory Allocation Failure:** While there is a check for `NULL` after `malloc`, the function simply returns without freeing any other resources that may have been allocated or opened earlier. This could potentially lead to resource leaks in other parts of the program.
7. **Magic Numbers:** The use of magic numbers (like `20`, `14` in `stream_Read` calls) makes the code less readable and maintainable. It would be beneficial to replace these with named constants that describe what they represent.
8. **Lack of Comments:** The code lacks comments explaining its logic, especially in complex or non-obvious parts. This can make maintenance and debugging more challenging.

After setting up this constrain, the stack buffer overflow issue will disappear.

Analysis Report:

1. Unchecked Return Value is a new issue, but it can be ignored because ChatGPT does not know what `stream_Seek()` is. And this is also what we can add to the prompt if we want to improve it more.
2. Integer Overflow issue and Unchecked User Data issue are related to the previous issue named as "Type mismatch".
3. Potential Heap Buffer Overflow is same as the previous issue named as Memory Leak.
4. Memory Allocation Failure is like the issue Error Handling.
5. Magic Numbers and Error handling have existed before.
6. Lack of Comment is a friendly remind.

Evaluation:

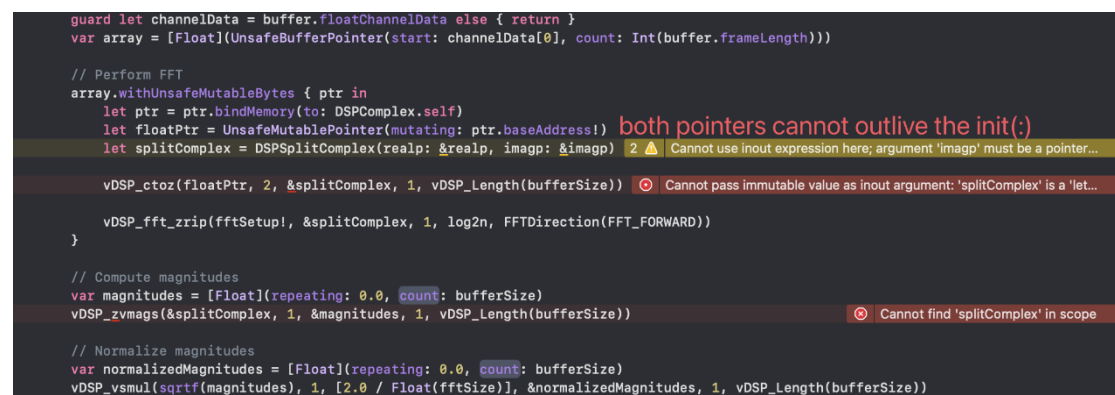


We finally figured out debugging on this part of program by following ChatGPT's instructions and improving the prompt generation.

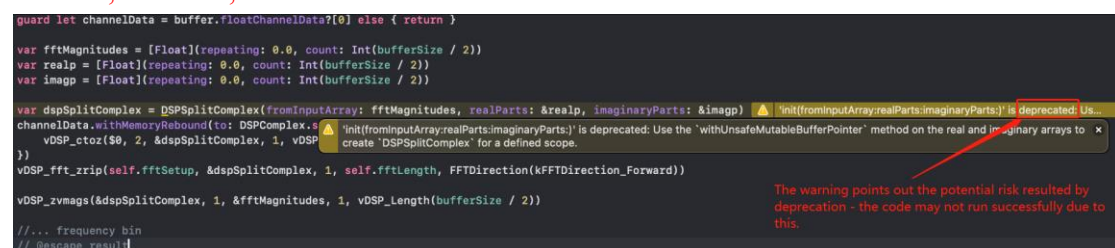
Futural Enhancement:

While the generated codes are correct, it doesn't check deprecation status from the vendor.

(a) Let's move on to some high-level programming language like swift.



(b) This screenshot indicates that the current method is unstable. The warning shows that early access may occur. This code generated by GPT does not have a correct context of the variable, therefore, the error occurs.



(c) This is a method with the latest guidance from Apple. In this way, we don't have to use the deprecated method.

```

guard let channelData = buffer.floatChannelData?[0] else { return }

var fftMagnitudes = [Float](repeating: 0.0, count: Int(bufferSize / 2))
var realp = [Float](repeating: 0.0, count: Int(bufferSize / 2))
var imagp = [Float](repeating: 0.0, count: Int(bufferSize / 2))

realp.withUnsafeMutableBufferPointer{
    realBP in
    imagp.withUnsafeMutableBufferPointer{
        imagBP in
        var dspSplitComplex = DSPSplitComplex(realp: realBP.baseAddress!, imagp: imagBP.baseAddress!)
        channelData.withMemoryRebound(to: DSPComplex.self, capacity: Int(bufferSize)) {
            vDSP_ctoz($0, 2, &dspSplitComplex, 1, vDSP_Length(bufferSize / 2))
        }

        vDSP_fft_zrip(self.fftSetup, &dspSplitComplex, 1, self.fftLength, FFTDirection(kFFTDirection_Forward))

        vDSP_zvmags(&dspSplitComplex, 1, &fftMagnitudes, 1, vDSP_Length(bufferSize / 2))

        //... frequency bin
        // @escape result
    }
}

```

Explanation:

Apple's shift towards Swift 6, with its emphasis on strict concurrency-safe coding, comes alongside major updates to iOS that introduce new SDKs. This transition has led Apple to phase out older methodologies in favor of more contemporary approaches. A key feature in Xcode, Apple's Integrated Development Environment (IDE), is the ability to easily identify concurrency and thread-safety issues. This is achieved by enabling strict concurrency checking within the project settings.

A straightforward method to ensure data processing occurs on the main thread involves the use of the `@MainActor` flag. This flag can be applied either to specific methods or to an entire class. Its role is pivotal in leveraging the compiler's understanding of the code, aiming to operate within defined actor boundaries. However, it's important to note that simply using the `@MainActor` flag does not automatically make existing code compliant with Swift 6, especially when data within an execution block crosses these actor boundaries.

The concept of a Swift actor, introduced in Swift 5.5, is central to these updates. Swift actors are a concurrency feature designed to manage shared mutable state safely and efficiently. Functioning as a type of object, they can be accessed concurrently across multiple threads. What sets them apart is their ability to ensure that access to their mutable state is serialized and thread-safe, marking a significant advancement in Swift's approach to modern, safe coding practices.

Conclusion:

Our exploration into Prompt Engineering reveals that while it may appear straightforward, as evidenced by our example, it is in fact a multifaceted and intricate process. The effectiveness of Prompt Engineering hinges on the precision and adequacy of the information (or hints) provided to the model. This is crucial for producing outputs that are not just relevant, but also of practical utility. A prime illustration of this is found in the realm of AI-generated imagery. It is evident that the quality of prompts directly influences the resultant image quality. Professional artists, with their detailed and accurate descriptions, can elicit far superior images from AI models compared to novices who might input vague or simplistic sentences

and expect detailed outcomes. This disparity underscores the significance of skillful Prompt Engineering in optimizing the capabilities of AI models, a key takeaway from our project findings.