1 Wuffs Parsing

1.1 Wuffs Overview

A final extension implemented for this project involves the Wuffs language. Wuffs is a language created by Google for memory-safe handling of untrusted file formats. Originally intended for use in security-specific portions of a program, Wuffs allows for compile-time checking of various security vulnerabilities, as well as some semi-rigorous proofs of program safety.

A simple parsing program in C is shown below. The same parsing program is also shown in Wuffs as a comparison of the two languages. Both of these programs take as input a string of numbers, for example, `"123"`, and return the associated numeric value, in this case, 123.

Leaving aside the syntactic peculiarities of Wuffs, note that the Wuffs code is signficantly more verbose than the C code in the above examples. However, we consider the edge cases of parsing some string, particularly with regards to integer overflow. In C, an integer overflow error can occur, but it does so silently. Obviously, in a real C parser implementation, we would implement security checks to cover these overflow cases. The point, however, is less about the fact that this is possible in C and more about the fact that these checks are mandatory in Wuffs. The Wuffs compiler will not allow an addition expression that could possibly overflow the maximum integer value.

This compile-time security checking for basic security vulnerabilites is incredibly useful, particularly in contexts where we are parsing unknown code or strings that could be malicious. We don't need to worry about runtime security checks when we can use Wuffs to guarantee no buffer overflows, out-of-bounds array accessing, integer overflow, or integer underflow at compile time.

```
uint32_t parse(char * p, size_t n) {
    uint32_t ret = 0;
    for (size_t i = 0; (i < n) && p[i]; i++) {
        ret = 10 * ret + (p[i] - '0';
    }
    return ret;
}
```

Figure 1: Parsing in C

```
pub func parser.parse?(src: base.io_reader) {
    var c : base.u8
    while true {
        c = args.src.read_u8?()
        if c == 0 {
            return ok
        }
        if (c < 0x30) or (0x39 < c) { // '0' and '9' in ASCII
            return "#not a digit"
        }
        //Rebase from ASCII to numeric value
        c -= 0x30

        if this.val < 429_496729 {
            this.val = (10 * this.val) + (c as base.u32)
            continue
        } else if (this.val > 429_496729) or (c > 5) {
            return "#too large"
        }

        this.val = (10 * this.val) + (c as base.u32)
    } endwhile
}
```

Figure 2: Parsing in Wuffs

1.2 Wuffs Extension

In this section, we discuss the mechanics of the Wuffs extension. For more details, examine the concrete syntax file for the `ABLEC`-wuffs extension. The extension is surprisingly simple, given the benefits it adds to a project.

The extension simply looks for a block of code delineated by the markers `WUFFS` and `WUFFS_END`. The code within these two markers is assumed to be valid Wuffs code. This code is loaded into its own `.wuffs` file, which is then compiled using the Wuffs compiler into a C file. In the original `ABLEC` file, we replace the original block of Wuffs code with boilerplate code that includes linking the new C file with the Wuffs functions, as well as calling those functions to parse the input, store the input, and validate the results. Note in the code snippet of the parse function using the extension that all of the Wuffs code is within the larger `parse()` function. This means that we need to do minimal editing of the rest of the file. We can retain the already written functions provided out-of-the-box by the server. In fact, in this case, the only function from the `parse.c` file that needs to be modified is the primary `parse()` function. The end programmer does not need to concern themselves with writing the C code to call any Wuffs functions – the extension is able to take care of inserting those function calls for us, allowing the programmer to simply write the parsing details in Wuffs.

1.3 Wuffs Implementation

Below, we have two code snippets of a parsing function. The first snippet is from the original server code. It contains a C implementation of the parser. The second code snippet is an edited Wuffs implementation of the parser, with some code removed for brevity.

We consider the differences between these two implementations. Obviously, there is a difference in syntax between the two languages. Beyond that, however, we consider the differences between the two implementations. First, we must primarily note that the C implementation has no guarantees of safety. If the program compiles, all we can be sure of is

```
GLOBAL bool Parse_Request( CONN_ID Idx, char *Request ){
        REQUEST req;
        char *start, *ptr;
        bool closed;

//Code omitted for brevity

        if (Request[0] == ':') {
                req.prefix = Request + 1;
                ptr = strchr( Request, ' ' );
                if( ! ptr ) {
                        return Conn_WriteStr(Idx, "ERROR :Prefix without command");
                }
                *ptr = '\0';
                start = ptr + 1;
        } else start = Request;
        ptr = strchr( start, ' ' );
        if( ptr ) {
                *ptr = '\0';
        }
        req.command = start;
        if( ptr ) {
                start = ptr + 1;
                while( start ) {
                        if( start[0] == ':' ) {
                                req.argv[req.argc] = start + 1;
                                ptr = NULL;
                        } else {
                                req.argv[req.argc] = start;
                                ptr = strchr( start, ' ' );
                                if( ptr ) {
                                        *ptr = '\0';
                                }
                        }
                        req.argc++;
                        if( start[0] == ':' ) break;
                        if( req.argc > 14 ) break;
                        if( ptr ) start = ptr + 1;
                        else start = NULL;
                }
        }
        if(!Validate_Prefix(Idx, &req, &closed))
                return !closed;
        if(!Validate_Command(Idx, &req, &closed))
                return !closed;
        if(!Validate_Args(Idx, &req, &closed))
                return !closed;

        return Handle_Request(Idx, &req);
} /* Parse_Request */
```

Figure 3: The parse function before adding the Wuffs extension

```
GLOBAL bool Parse_Request( CONN_ID Idx, char *Request ) {
  WUFFS
    pub struct parser?(
      prefix: slice base.u8, cmd : slice base.u8,
      args : array[15] slice base.u8, argc : base.u32[..=15]
    )
    pub func parser.parse?(src: base.io_reader) {
      var c : base.u8 = args.src.read_u8?() //Code removing whitespace omitted
      var i : base.u32[..15] = 0
      var extra : base.u32
      if c == ':' {
        extra = 0
        while true {
          c = args.src.read_u8?()
          if c == ' ' {
            args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.prefix)
            break
          }
          else { extra += 1 }
        }endwhile
        c = args.src.read_u8?() //Prep for reading command
      } else { //Handle command
        extra = 1
        while true {
          c = args.src.read_u8?()
          if c == ' ' {
            args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.cmd)
            break
          } else { extra += 1 }
        }endwhile
      }
      c = args.src.read_u8?()
      if c == 0{ return } //No more args, we are finished
      if c == ' '{ /* Code moving past whitespace omitted */ }
      else { //At least one arg
        extra = 1
        while.loop1 true {
          while.loop2 true {
            c = args.src.read_u8?()
            if c == 0 { break.loop1 } // c is null, we are finished parsing
            if c == ' ' {
              args.src.limited_copy_u32_to_slice!(up_to: extra, s: this.args[i])
              extra = 0
              i += 1
              if i >= 14{ break.loop1 }
              else{ break.loop2 }
            } else { extra += 1 }
          }endwhile.loop2
        }endwhile.loop1
        this.argc = i+1
      }
    } WUFFS_END
} // Parse_Request
```

Figure 4: The parse function after adding the Wuffs extension

that there are no syntax errors in the file. We have no guarantees that, given some malicious input string, we won't be vulnerable to any attacks. In our Wuffs implementation, this code is sent off to the Wuffs compiler during compilation. The Wuffs compiler is able to tell us whether we are vulnerable to buffer overflows or other attacks. This is the largest benefit of using this extension. We get the speed of C code, but the safety and guarantees provided by the Wuffs compiler.