A Case Study on Automated Fuzz Target Generation for Large Codebases – Appendix

To illustrate the methodology of our case study, we present the following example of a Fuzz Target created with each of the three methods for the same target function.

```
// @fuzztest Array(pBuf, Len)
// Output(ppMsg)
// Cleanup(ppMsg != NULL,
//         J2735_Free, ppMsg)
void J2735_MsgPERDecode(
            struct J2735Msg **ppMsg,
            char *pBuf,
            int Len) {
    ...
}
```

Listing 1: Example stack function

The function shown in Listing 1 takes a buffer containing a PER encoded J2735 message and decodes that buffer into a `J2735Msg` structure. The buffer `pBuf` is an array of type `char` with length `Len`; this is handled with the addition of an `Array` parameter to the `@fuzztest` directive. The decoded message is stored in the `ppMsg` parameter, which is marked as output using the `Output` parameter. Memory is also allocated within this function for the structure, which must be freed by the Fuzz Target in order to prevent a memory leak. There also exists a function specifically for freeing a `struct J2735Msg`, thus the allocated memory can be freed by calling this function. This call to free is achieved with the addition of a `Cleanup` parameter to the `@fuzztest` directive using the condition `ppMsg != NULL` to ensure that the `J2735_Free` function does not attempt to free a null pointer.

A. *Fully Automated*

```
int LLVMFuzzerTestOneInput(
                uint8_t *fuzz_input_data,
                size_t fuzz_data_size) {

    size_t fuzzer_input_min_size =
         sizeof(struct J2735Msg *) +
         sizeof(char) +
         sizeof(int);

    if(fuzz_data_size <
       fuzzer_input_min_size) {
        return 0;
    }

    uint8_t *fuzz_ptr = fuzz_input_data;

    struct J2735Msg *ppMsg;
    char pBuf;
    int Len;

    memcpy(&ppMsg, fuzz_ptr,
           sizeof(struct J2735Msg *));
    fuzz_ptr += sizeof(struct J2735Msg *);

    memcpy(&pBuf, fuzz_ptr, sizeof(char));
    fuzz_ptr += sizeof(char);

    memcpy(&Len, fuzz_ptr, sizeof(int));
    fuzz_ptr += sizeof(int);

    (void)J2735_MsgPERDecode(&ppMsg,
                             pBuf,
                             Len);

    return 0;
}
```

Listing 2: Automated Fuzz Target for J2735_MsgPERDecode

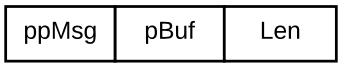

Fig. 1: Automated serialisation of J2735_MsgPERDecode

The fully automated method produces the Fuzz Target shown in Listing 2 and is unable to determine that there is an array in the function parameters, that one of the parameters is used as output, resulting in it serialising the function as shown in Figure 1. It is also unable to determine that a cleanup function needs to be called to prevent memory leaks. Because of this not only is it unable to properly Fuzz Test the target, as the length parameter passed for the buffer is determined by the `libFuzzer` input, and the actual length of the buffer will always have a size of 1, but there is also a high probability of a false positive access violation crash being found. However, although the appropriate cleanup function is never called, due to sanity checking within the function, the call to `J2735_PERDecode` will never progress far enough to actually allocate memory for `J2735Msg` structure and thus a false positive leak will not be found in addition to the false positive crash. In this case, the fully automated method has not been able to produce a functional Fuzz Target.

B. *Anotated*

```
int LLVMFuzzerTestOneInput(
                uint8_t *fuzz_input_data,
                size_t fuzz_data_size) {

    size_t fuzzer_input_min_size = 0;

    struct J2735Msg *ppMsg;
    char *pBuf = (char *)fuzz_ptr;
```

```
    int Len = (fuzz_data_size -
               fuzzer_input_min_size)
               / sizeof(char);

    (void)J2735_MsgPERDecode(&ppMsg,
                             pBuf,
                             Len);

    if(ppMsg != NULL) {
        J2735_Free(ppMsg);
    }

    return 0;
}
```

Listing 3: Annotated Fuzz Target for J2735_MsgPERDecode

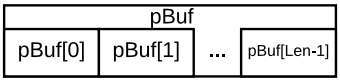

Fig. 2: Annotated serialisation of J2735_MsgPERDecode

As Listing 3 shows, with the addition of annotations, the problems with the fully automated method can be solved, resulting in the function being serialised as shown in Figure 2. The addition of the `Output` parameter allows `ppMsg` to be removed from the input as it is used as output by the target function. The `Array` parameter then allows `pBuf` to utilise all of the libFuzzer input data and assigns `Len` to the actual length of the buffer. Thus, the annotated method is able to create a Fuzz Target for this target function.

C. Handmade

```
int LLVMFuzzerTestOneInput(
                uint8_t *fuzz_input_data,
                size_t fuzz_data_size) {

    struct J2735Msg *pMsg;

    (void)J2735_MsgPERDecode(
            &pMsg,
            (char *)fuzz_input_data,
            fuzz_data_size);

    if(pMsg != NULL) {
        J2735_Free(pMsg);
    }

    return 0;
}
```

Listing 4: Handmade Fuzz Target for J2735_MsgPERDecode

This handmade Fuzz Target, shown in Listing 4 and created by a developer working at our industry partner, is functionally very similar to that generated through the annotated method, considering input in the same format described in Figure 2. However, it is written in a less generic way making it neater and slightly more efficient by removing some of the setup that occurs before the call to the target function in the Fuzz Target generated with the annotated method. The efficiency improvement between the handmade Fuzz Target in comparison to that generated through the annotated method is minor, with most of the effectiveness gained being from the accompanying seed corpus.