Annotating public bugs with ACSL: Experience report[1]

David R. Cok
Ian Blissard
Joshua Robbins

GrammaTech, Inc.

1 Introduction

To verify that software behaves as desired, one must have a description of the intended behavior. If the verification of the software is to be automated, then that description must be machine readable, in an unambiguous logical representation. Such a representation is typically called the specification and expressed as software annotations.

One goal of software verification using behavioral specifications is that if the software is properly specified and the specifications and implementation are proved consistent, then the implementation is correct. In actual practice, behavioral specifications are not widely used and bugs in implementations are common. An interesting question is this: if specifications had been used during the original development would the bugs have been found? This is, of course, an after-the-fact determination. Just because, knowing the bug now and being able to write specifications that demonstrate the error does not mean that a regimen of specification at the time would have prevented it. However, if a bug cannot be identified even after-the-fact, then the software verification approach would need some rethinking.

This paper is an experience report in this regard. We identified several publicly reported bugs in commonly used Linux utilities for which we could obtain the code (from source code repositories) just before and just after the bug fix. We then included behavioral annotations in the code and demonstrated that the relevant tools pointed out that the code was faulty before the fix and the same annotations were consistent with the code after the fix.

The next section gives some background on the tools we used. The subsequent sections present the various bugs we analyzed, and the last section presents some observations.

[1] This material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2 ACSL and Frama-C

For C programs, there are a variety of annotation systems. However most are fairly rudimentary, in that they only describe limited aspects of the program, such as that some variables are not null pointers, or that some assertion holds at a given program point. Other annotations systems are proprietary.

One system that seeks to be a language that can describe full behavioral requirements and be used for verifying, by proof, the correctness of a piece of software, is the ANSI-C Specification Language (ACSL) [https://frama-c.com/acsl.html]. This language is in turn based on other specification languages for other source programming languages, such as JML for Java and Spec# for C#; these are all in a family known as Behavioral Interface Specification Languages.

ACSL is a specification language. The Frama-C framework is a tool suite that performs various analyses on (ANSI-C) software and corresponding ACSL specifications. For this project we used Frama-C with its wp plugin, which performs static, proof-based checking that the implementation and the specifications are consistent.

Most software makes heavy use of libraries. For verification systems to work, those libraries must also have behavioral specifications. Lack of library specifications is in fact one of the impediments to wider use of specification-based software verification. In a companion report that is part of the same NSF project that supported the work reported here, we describe our effort to expand the available specifications for C system libraries. In this work we simply use those specifications.

The demonstration experiments reported here were mostly accomplished with Frama-C versions Sodium and Magnesium. They were later revalidated with the Silicon version.

3 Demo bugs

Each of the following sections describes a particular publicly reported bug in Linux or busybox utilities.

4 Bug in pr

Pr is a Unix text reformatting tool.

4.1 Revisions Used

Repository: git://git.sv.gnu.org/coreutils.git

Version with bug

- Version 6.10
- Commit: 1440ca24fd216bb3a4d7c388b23a9512a4585508
- Mon, 21 Jan 2008 23:31:02 +0000 (00:31 +0100)

First presence

- Initial commit of file
- Commit: b25038ce9a234ea0906ddcbd8a0012e917e6c661
- Sun, 8 Nov 1992 02:50:43 +0000 (02:50 +0000)

Initial fix

- Commit: 6856089f7bfaca2709b303f01dae001a30930b61
- Sat, 19 Apr 2008 11:34:38 +0000 (13:34 +0200)

4.2 Description of Bug

A global variable *input_position* is used to track the horizontal position in a file. This position is used to convert special characters such as tabs, which represent multiple spaces, and backspaces, which have a negative space. Because *input_position* represents the position in a file, it does not make sense for the value to be less than zero, and the bug occurs when this variable is negative.

The code fails when, before a tab (\t), the number of backspaces (\b) exceeds the number of characters that can be removed. The failure happens when the macro TAB_WIDTH gets a negative value for the input_position variable (the position for the next character to be written). Thus, the macro returns a value that is larger than the size of array clump_buff and the variable *width* (which gets its value from the macro) exceeds the size of the buffer. This bug can be found at version 6.10.

```
// This is required to validate the precondition for isprint (uc) in the else
if below
//@ ghost int __gt_fc_int_arg = uc;
//@ assert (unsigned char)(__gt_fc_int_arg) == uc;
if (c == input_tab_char || c == '\t')
  {
    width = TAB_WIDTH (chars_per_c, input_position);

...

// 6.10 BUG
//  input_position += width;
// Initial fix:
/* Too many backspaces must put us in position 0 -- never negative.  */
if (width < 0 && input_position == 0)
  {
    input_position = 0;
  }
else if (width < 0 && input_position <= -width)
   input_position = 0;
else
   input_position += width;
return chars;
 }
```

4.3 Annotations

A predicate was defined to represent an invariant for *input_position*.

```
/*@ predicate valid_input_position = (input_position >= 0); */
```

This predicate was then added as a pre and post condition to the function *char_to_clump*, which is responsible for converting characters to the proper format, and incrementing or decrementing *input_position* by the width.

```
/*@
    requires valid_input_position;
    ensures valid_input_position;
*/
static int
 char_to_clump (char c)
```

Noting that the function *read_line* also modifies *input_position*, and makes calls to *char_to_clump*, the same contract was added to *read_line* to show it preserves the invariant.

4.4 Results

With the annotations applied on version 6.10, the only goal which is not validated is the post-condition of *char_to_clump*. Validation of the function *read_line* shows that it both preserves the invariant, and makes valid calls to *char_to_clump*. However, failing to validate *char_to_clump* indicates the function may not preserve the invariant.

After applying the fix in commit 6856089f7bfaca2709b303f01dae001a30930b61, all of the goals are validated.

4.5 Modifications & Mitigations

The c11 feature _Static_assert is used, but frama-c only supports c99. The statement using this feature was removed from the file lib/verify.h.

A global invariant would have been used to check valid_input_position, but the feature is not well supported. Instead, the invariant was added to the pre and post condition of functions being analyzed.

5 Bug in arp

Running *busybox arp -Ainet* in version 1.6.2 results in *Segmentation fault.*

5.1 Version used

Version with bug

- Version 1.6.2
- Available at http://www.busybox.net/downloads/

5.2 Description of Bug

Below is a section of code from arp_main. The error is in the second conditional, which should be checking ARP_OPT_H, and ARP_OPT_t. As is, the call to get_hwtype may have a null argument. When corrected, hw_type is known to be valid, because of the relationship defined by getopt32_arp. The error originates from a copy-paste error: the two if conditions are identical and should not be.

```
getopt32_arp(argc, argv, "A:p:H:t:i:adnDsv", &protocol, &protocol,
                                &hw_type, &hw_type, &device);  // LINE A
argv += optind;

if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
  ap = get_aftype(protocol);
  if (ap == NULL)
    bb_error_msg_and_die("%s: unknown %s", protocol, "address family");
}

if (option_mask32 & ARP_OPT_A || option_mask32 & ARP_OPT_p) {
  hw = get_hwtype(hw_type);  // LINE B
  if (hw == NULL)
    bb_error_msg_and_die("%s: unknown %s", hw_type, "hardware type");
  hw_set = 1;
}
```

The bug can be found on the line marked A in the int *arp_main* method. Since neither the H nor t option was given in the command line, *hw_type* was set to Null. Thus, at line B we have a Null passed to *get_hwtype*. This value is then used in a call to *strcmp*. The behaviour of *strcmp* is not defined for invalid strings, so this call should be avoided.

5.3 Annotations

No annotations are required for the function *arp_main*, which contains the error.

The function *get_hwtype* was annotated with a pre-condition to require a valid argument.

```
/*@ requires valid_name: \valid(name); */
```

Additionally, a loop inside *get_hwtype* was annotated with invariants and a variant to show calls to *strcmp* satisfy its pre-condition.

```
//@ ghost const struct hwtype *const *hwp_orrig = hwp;
```

```
//@ ghost int hwp_offset = 0;
/*@
    loop assigns hwp, hwp_offset;
    loop invariant \valid(name);
    loop invariant hwp == hwp_orrig + hwp_offset;
    loop invariant 0 <= hwp_offset <= hwtypes_size;
    loop variant hwp_orrig+hwtypes_size - hwp;
*/
while (*hwp != NULL) {
  if (!strcmp((*hwp)->name, name))
    return (*hwp);
  hwp++;
  //@ ghost hwp_offset++;
}
```

Annotations relating to the global variable *hwtypes[]* were added. First, a logical integer was declared setting the known size.

```
/*@
    axiomatic HWTypes {
    logic integer hwtypes_size{L};
    axiom hwtypes_const_len{L}: hwtypes_size == 5;
}
*/
```

Next, a global invariant was defined showing some properties of the structure.

```
/*@
  global invariant hwtype_structure:
    \valid(hwtypes+(0..hwtypes_size))
    && \forall integer i; (0 <= i < hwtypes_size) ==>; (\valid(*(hwtypes+i)))
    && \forall integer i; (0 <= i < hwtypes_size) ==>
      (\valid((*(hwtypes+i))->name))&& *(hwtypes+hwtypes_size) 0;
*/
```

Minimal contracts were added for functions called in arp_main.

A contract was added to *get_aftype* showing it does not have any side effects.

```
/*@ assigns \nothing; */
```

A post-condition of false was added to *bb_error_msg_and_die*, which exits, and does not return from the function in any case.

```
/*@ ensures \false; */
```

5.4 Results

A precondition required by get_hwtype in arp_main fails, indicating the possible error. The bug exists because incorrect flags are being checked. By fixing this error, arp_main is validated completely.

The function get_hwtype is validated completely, when assuming the correct properties of hwtypes[].

5.5 Modifications & Mitigations

The definition of *strlen* provided by frama-c's libc headers while correct, was too difficult to be proven in several instances. To have a useable definition, the specification was weakened by changing the pre-condition from the *valid_string* predicate, to the built in *\valid*.

String literals were refactored as needed to instead use a temporary variable, because *\valid* was incorrectly failing.

Function *getopt32_arp* which proxies to *getopt32* was added specifically for the *arp_main* function, because *getopt32* uses a variable argument list. Because of the setup, the annotations are not proven, but provides a contract which can be assumed.

```
/*@
   ensures \result == option_mask32;
   ensures (option_mask32 & (0x1)) <==> \valid(*protocol); // ARP_OPT_A
   ensures (option_mask32 & (0x2)) <==> \valid(*protocol2); // ARP_OPT_p
   ensures (option_mask32 & (0x4)) <==> \valid(*hw_type); // ARP_OPT_H
   ensures (option_mask32 & (0x8)) <==> \valid(*hw_type2); // ARP_OPT_t
*/
```

Because global invariants are not well supported, the property was simply asserted at the beginning of get_hwtype.

```
//@ assert \valid(hwtypes+(0..hwtypes_size));
//@ assert \forall integer i; (0 <= i < hwtypes_size) ==>
(\valid(*(hwtypes+i)));
//@ assert \forall integer i; (0 <= i < hwtypes_size) ==>
(\valid((*(hwtypes+i))->name));
//@ assert *(hwtypes+hwtypes_size) == 0;
```

5.6 References

Partha Pratim Ray and Ansuman Banerjee. Debugging memory issues in embedded linux: a case study. In Students' Technology Symposium (TechSym), pages 23–28. IEEE, 2011.

6 Bug in cut

The `cut` utility performs editing on lines of text, removing specified portions of the line according to instructions given in the command-line options.

6.1 Description of Bug Behavior

The bug is described at this web page:
http://debbugs.gnu.org/cgi/bugreport.cgi?msg=5;att=0;bug=13627

It is interesting to note that the bug was fixed, introducing a second bug. When the second bug was fixed, the first was reintroduced. The bug can be reproduced with the command:

```
echo '' | ./src/cut --output-d=: -b1,1234567890-
```

6.2 Revisions Used

Repository: git://git.sv.gnu.org/coreutils.git

Version with bug

- Commit: d57ebc45ba4c59cc6f8bb0e9a435ecbddc84b982
- Fri, 1 Feb 2013 21:33:21 +0000

Introduced

- Commit: ec48beadfa0ae1216788eaf6bf558ee2013eac18
- Thu, 6 Dec 2012 18:29:23 +0000

Initial fix

- Commit: be7932e863de07c4c7e4fc3c1db3eb6d04ba9af5
- Mon, 4 Feb 2013 13:55:01 +0000

6.3 Description of Bug

The function set_fields allocates the printable_field array. This function calls is_printable_field which assumes the argument refers to a valid location in printable_field. However this precondition cannot be guaranteed, and the test case will cause a segmentation fault due to attempting to access a value past the array bounds.

6.4 Annotations

Contracts were added to required functions from gnulib by adding headers.

```
/*@ requires s>0;
    ensures \valid((unsigned char *)\result+(0..(s-1)));
*/
```

```
extern void *xzalloc(size_t s);

/*@ assigns *table; */
extern void *hash_insert (Hash_table *table, void const *entry);
```

Minimal contracts were added to some functions used by *set_fields*.

```
/*@ assigns *range_start_ht; */
static inline void mark_range_start (size_t i)

/*@ assigns printable_field[i/8]; */
static inline void mark_printable_field (size_t i)

/*@ requires \valid(printable_field+(i/8));
    assigns \nothing;
*/
static inline bool is_printable_field (size_t i)
```

In the function *set_fields*, properties of the variable *max_range_endpoint* are discovered through a loop invariant.

```
max_range_endpoint = 0;
//@ ghost size_t max_range_endpoint_old = max_range_endpoint;
/*@
    loop assigns i, max_range_endpoint, max_range_endpoint_old;
    // max_range_endpoint only increases
    loop invariant max_range_endpoint >= max_range_endpoint_old;
    // max_range_endpoint is max of rp[..].hi
    loop invariant \forall integer idx; (0 <= idx < i) ==>
                                    (max_range_endpoint > rp[idx].hi);
    loop variant n_rp - i;
*/
for (i = 0; i &lt; n_rp; i++)
{
  //@ ghost max_range_endpoint_old = max_range_endpoint;
  if (rp[i].hi > max_range_endpoint)
    max_range_endpoint = rp[i].hi;
}
```

It is now known that at the end of the loop, *max_range_endpoint* holds the maximum *hi* value from the *rp* array.

Additional annotations are added to later loops to show variable *max_range_endpoint* is unchanged, and maintains the relationship with *printable_field*.

```
/*@ loop assigns i, *range_start_ht, printable_field[0 ..
(max_range_endpoint/8)]; */

/*@ loop assigns j, printable_field[0 .. (max_range_endpoint/8)]; */
```

The attempt to validate the pre-condition to *is_printable_field*, wp was having difficulty, so a lemma was added. The lemma validates, indicating it is always true.

```
/*@
    lemma Inner_Printable_Field_Validity:
      (max_range_endpoint ==>
          \valid(printable_field+(0..(max_range_endpoint / 8)))) ==>
      (max_range_endpoint ==>
          (((0 <= eol_range_start <= max_range_endpoint)) ==>
             \valid(printable_field+(eol_range_start / 8))));
*/
```

6.5 Results

Earlier in the function, a relationship is formed between *max_range_endpoint* and the allocated size of *printable_field*.

```
if (max_range_endpoint)
  printable_field = xzalloc (max_range_endpoint / CHAR_BIT + 1);
```

The later loops are annotated with `assigns` clauses to preserve this property. The code below shows the error.

```
if (output_delimiter_specified
    && !complement
    && eol_range_start
    && max_range_endpoint
    && !is_printable_field (eol_range_start))
        mark_range_start (eol_range_start);
```

Even though *max_range_endpoint* is checked, the restrictions are not strong enough. The call to *is_printable_field* requires the parameter to reference a valid index of *printable_field*, and we have insufficient information on *eol_range_start* to make that claim. The fix adds a condition relating *eol_range_start* to *max_range_endpoint*, which we have already established is related to the size of *printable_field*.

```
if (output_delimiter_specified
    && !complement
    && eol_range_start
    && max_range_endpoint
    && (max_range_endpoint < eol_range_start
            || !is_printable_field (eol_range_start)))
        mark_range_start (eol_range_start);
```

As expected, the precondition on *is_printable_field* in the original code does not validate, but does validate once the fix is applied. Furthermore, when assuming the contracts provided for the functions in gnulib, the contracts for *mark_range_start*, *mark_printable_field*, and *is_printable_field* validate. Additionally, all but one of the loops validates.

```
/*@ loop assigns j, printable_field[0 .. (max_range_endpoint/8)]; */
for (size_t j = rp[i].lo; j <= rp[i].hi; j++)
                    mark_printable_field (j);
```

The code above does not validate with frama-c, but is true. Adding the invariant `j >= 0` validates, so the loop variable range is `0 <= j <= rp[i].hi`, where `0 <= i < n_rp` from an outer loop. Additionally, the previous loop held the invariant `\forall integer idx; (0 <= idx < n_rp) ==> (max_range_endpoint >= rp[idx].hi)`, or `max_range_endpoint` is the largest *hi* value in *rp* . Therefore, the largest possible range for *j* is `0 <= j <= max_range_endpoint`. The single function called in this loop with argument *arg* assigns `printable_field[arg/8]`. Therefore, the worst case assignments from this loop is *printable_field* with indexes `j/8`, which is bounded by `0 <= j/8 <= max_range_endpoint/8`. This is written as `printable_field[0 .. (max_range_endpoint/8)]`.

6.6 Modifications & Mitigations

- Removed _Static_assert calls from gnulib/lib/verify.h to comply with C99.
- Used Typed+cast model because of the pointer type conversions required.

6.7 Expected Failures

Some goals do not prove:

- typed_cast_set_fields_loop_assign_3_part3: This loop assignment is valid, but cannot be proven.

Other goals cannot prove due to the use of undefined behavior when assigning to some variables:

- typed_cast_set_fields_call_strspn_pre_valid_string_src
- typed_cast_set_fields_call_free_deallocation_pre_freeable
- typed_cast_set_fields_call_qsort_pre
- typed_cast_set_fields_call_free_deallocation_pre_freeable_2

7 Bug in fetchsms

Fetchsms is a function in OpenSER which gets an SMS message from SIM memory.

7.1 Source

http://www.kamailio.org/pub/openser/1.1.0/src/openser-1.1.0-tls_src.tar.gz

7.2 Description of bug

“A buffer, pdu[], is passed to fetchsms(). fetchsms() writes into pdu[] from another buffer, answer[], which it gets from the modem. fetchsms() does some heavyweight string parsing of answer[], and copies part of answer[] into pdu[]. Unfortunately, pdu[] is too small to hold this substring of answer[].”

A response string is written into *answer*, which has a size limit of 512. The function that writes the response guarantees a valid string with a null terminator will be written, and truncates the response to fit. A substring from the response is found, and written into the function argument *pdu*. Because it is a substring of a bounded size string, we can assert if the write into *pdu* is legal, if the size of *pdu* is known. Until this point, *pdu* is unmodified, so if an inadequately sized array is provided, the bug could manifest.

7.3 Annotations

The libc header files from frama-c for string.h were used. Additionally, the provided contract for *sprintf* was added.

The function *put_command* in file *modules/sms/libsms_modem.c* writes a command to the modem *mdm*, and copies as much of the response as can fit into *answer*. The value in *answer* is always null terminated.

```
/*@
    requires \valid(answer+(0..max-1));
    assigns *(answer+(0..(max-1)));
    ensures valid_string(answer);
*/
 int put_command( struct modem *mdm, char* cmd, int cmd_len, char* answer,
                      int max, int timeout,char* exp_end)</verbatim>
```

The function *fetchsms* in file *modules/sms/libsms_getsms.c* is where the bug occurs. The final two requires statements prevent the bug from occurring by constraining the size of *pdu*.

```
/*@
    requires \valid(mdm);
    requires \valid(pdu);
    requires \separated(mdm, pdu);

    requires \valid(pdu+(0..511));
    requires \separated(mdm, pdu+(0..511));
 */
 int fetchsms(struct modem *mdm, int sim, char* pdu)
```

Three simple loops in *fetchsms* were annotated with assigns statements.

```
/*@ loop assigns end; */
```

Several assertions were added into the function *fetchsms* to show the sizes of substrings are bounded, and that *pdu* remained valid.

The lemma was added to assist the prover. The lemma verifies, indicating it is a tautology.

```
/*@
    lemma ValidAtLeastToZero:
        \forall char *s; valid_string(s) ==>
        ( \exists integer i;
```

```
            *(s+i) == 0
            && \valid(s+(0..i))
        );
*/
```

7.4 Results

No fix was found to resolve this potential bug. However, there is an implicit pre-condition on *fetchsms* which prevents the scenario.

The call to *strcpy* on line 192 will fail the *room_string* precondition if the precondition *requires \valid(pdu+(0..511))* on *fetchsms* is not present.

Additionally, it was found that there is potential for a run time error if a malformed response is received. On line 171, *position* is set to the location of "+CMGR:", which guarantees valid memory from *position* to *position+6*, for each letter and the null terminator. Later, on line 178, *beginning* is set to *position+7*, which is may not be a valid address. Then on line 184, *end* is set to *beginning*, and dereferenced.

In actuality, an error here is not observed because a well formed response in this case follows "+CMGR:" with a space, then additional information (http://www.developershome.com/sms/cmgrCommand.asp). The potential error could be avoided by changing the search string to "+CMGR: ", with a space.

For this experiment, an annotation was added after the initial call on line 171 to assume a well formed response. The assertion *(position == 0) || (6<= strlen(position) < 512)* validates, and the stronger assertion *(position == 0) || (7<= strlen(position) < 512)* does not validate, as expected.

7.5 Modifications & Mitigations

String literals were refactored as needed to instead use a temporary array, because *\valid* was incorrectly failing.

There was an issue with including the libc headers provided by frama-c, causing an error in gcc.

The option -pp-annot is used to allow the use of macros in annotations. However, this causes gcc warnings.

Some properties which were valid were timing out. If given enough time, it is possible the goals would be validated. Instead, because the property was known at one point in the program, assertions were added at points between the known location, and the location where the property was utilized, to assist in showing the property held. These extra annotations allowed validation in reasonable time.

Validation of string properties is extremely slow. The option -wp-split was sufficient for most, goals, but some required an extended timeout. A full validation took over an hour.

7.6 References

http://blog.frama-c.com/public/scam09.pdf

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6876

https://github.com/REMath/implementations/tree/master/tanalysis/tanalysis/verisec%20v1.0/OpenSER/CVE-2006-6876

8 Bug in seq

Seq is a Unix utility that generates a sequence of numbers. The command-line options allow different ranges of numbers and allow the numbers to be formatted in different ways.

8.1 Description of Bug Behavior

The bug is that specify a floating point format with a length of zero causes an out of read:

$ ./seq -f %0 1

./seq: memory exhausted

8.2 Revisions Used

Version with bug: 6.10

8.3 Annotations

Preconditions, and loop annotations were added to long_double_format to prove the generated RTE assertions.

```
//@ predicate is_format_content_valid(char *fmt, integer i ) = (0 <= i &&
\forall integer j;  0 <= j < i ==> fmt[j] != '\0') ;

/*@
   requires valid_read_string(fmt);
   requires strlen(fmt) < SIZE_MAX - 1;
   requires \valid(layout);
   requires \valid_read(fmt+(0..strlen(fmt)+1)); //correction for slight
innacuracy in valid_read_string
*/
static char const *
long_double_format (char const *fmt, struct layout *layout)

/*@
      loop invariant is_format_content_valid(fmt, i) && 0 <= i <= strlen(fmt)
&& 0 <= prefix_len <= i;
      loop assigns i, prefix_len ;
 */
```

```
   for (i = 0; ! (fmt[i] == '%' && fmt[i + 1] != '%'); i += (fmt[i] == '%') +
1)
```

xmalloc in xalloc.c was also annotated because it is used by long_double_format.

8.4 Results

The invariant `strlen(fmt) >= i` in the second loop allows the rte assertion to validate, which checks access to fmt[i]. Without the bug fix, this invariant is not proven. However after the fix, all loop annotations, and rte assertions are proven.

8.5 Modifications & Mitigations

There are four cases where valid_read_string fails as a precondition. In all cases the parameter is a string literal, thus valid. This was left as is because the normal mitigation was too slow in proving, and these are obviously valid.

8.6 References

Bug report: http://lists.gnu.org/archive/html/bug-coreutils/2008-03/msg00181.html

Fix:
http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=b8108fd2ddf77ae79cd014f4f37798a52be13fd1

9 Bug in jq

The jq library is a parser for JSON input, written in C.

9.1 Description

The bug reported is #896 in the jq github project (https://github.com/stedolan/jq/issues). It is a use-after-free bug from short but unusual input, originally found by the AFL fuzzer (American Fuzzy Lop). From the command line, the input ‘jq –n ‘.[{}] = 0’ will cause a segmentation fault.

JQ internally uses a reference counting mechanism to free allocated memory not in use. The function parse_slice in jv_aux.c takes allocated blocks as input, and automatically decrements their reference counts when it is done with them. The function jv_get, also in jv_aux.c, passes memory into parse_slice, using the memory later in the code. However, it failed to increment the reference counter before passing it in to parse_slice, resulting in the memory potentially being freed if the reference counter hit zero. jv_get would later attempt to decrement the reference counter again, causing the memory to be freed again, which is a double-free, causing the crash.

9.2 Annotations

The functions parse_slice and jv_get were annotated in jv_aux.c, as well as all the called functions in the file jv.h. The core annotation to the proof is the logic for keeping track of reference counts.

```
/*@
axiomatic JvRefcount {
    logic integer jv_refcount{L}(struct jv_refcnt* p) reads \at(*p, L);
}
@*/
```

Many functions were given the requirement to expect arguments with positive reference counts, i.e. non-freed items.

```
requires jv_refcount{Pre}(j.u.ptr) > 0;
```

And the reference-counting increment and decrement functions were annotated.

```
ensures jv_refcount{Post}(j.u.ptr)  == ensures jv_refcount{Pre}(j.u.ptr) + 1
;
ensures jv_refcount{Post}(j.u.ptr)  == ensures jv_refcount{Pre}(j.u.ptr) - 1
;
```

In addition, asserts were added in jv_aux.c to emulate a weak type invariant:

```
//@ assert jv_refcount{Here}(j.u.ptr) > 0;
```

9.3 Running

Not all goals will pass in the current version. These are the ones that may fail:

- `*_assert_separation`: Separation logic of allocation cannot be currently expressed in Frama-C, so asserts are used instead.
- `*_assert_vac`: The vacuous test. If it proves, then there is a contradiction in logic allowing \false to be provable.
- `*_assert_bug_correction`: This is an assert to offset the bug desrcibed here.
- `jv_get_call_jv_free_pre_6`: Despite successfully asserting the preconditons for jv_free directly above it, the preconditions for this function will not prove. Possible Frama-C bug.

All other goals are expected to pass.

9.4 Details

See `jv_aux.c` in both versions for more details about the proof.

10 Bug in ag

This is a proof that a bug existed in the codebase of The Silver Searcher, a filesystem searching tool written in C. The issue we are checking for is fixed in this commit. When used on arrays more than 2 billion elements in length, the utility function binary_search would crash due to an overflow.

10.1 Description of bug

The binary_search function takes two integer arguments, start and end. When the difference between these two were greater than two billion, integer overflow would occur at this line, rendering the offset into the list negative (and thus a possibly invalid access).

```
mid = (start + end) / 2;
```

Since the difference between the two items is a value that will not underflow, this expression fixes the bug.

```
mid = start + ((end - start) / 2);
```

10.2 Annotations

A function contract was added to binary_search to require the normal semantics of a start and end parameter. The end must be after the start, they must be positive offsets, and are in valid ranges.

```
/*@
requires \valid(haystack+(start..end));
requires \valid(needle);
requires start >= 0;
requires end >= 0;
requires end - start >= 0;
@*/
int binary_search(const char *needle, char **haystack, int start, int end) {
```

10.3 Running

If Frama-C complains about header files missing, you probably don't have libpcre-dev or liblzma-dev installed on your system. Use apt-get to install them on Linux, or apt-cyg if on Cygwin.

All the goals are expected to pass in the fixed version of the code.

10.4 Details

See `src/util.c` in both versions for more details about the proof.

11 Observations

The overall mini-project we attempted was successful:

- We identified several publicly reported bugs in real code for which we could obtain the code before and after the fix
- We annotated the code without modification with specifications in ACSL.
- The Frama-C –wp tool failed to validate the code+annotations before the fix.
- The Frama-C –wp tool successfully validated the code+annotations after the fix.

We did encounter some limitations of Frama-C's current implementation which precluded validating all generated verification conditions or required small adjustments to the original C code. This experience added to our store or idioms and techniques that help with validating specifications using Frama-C. The issues and mitigations we encountered are noted in each section above and are also summarized in a separate larger experience report about Frama-C.