OVS Project

Finding Security Vulnerabilities with CBMC

Stefan Diels Philippe Tanghe Li Quan

December 28, 2011

Contents

Task 1

First, we fixed the problem of `cbmc` with respect to the `strdup` function (unexpected array expression: 'byte_update_little_endian') as follows[1]. The implementation used in this case is not very memory efficient (especially when `strlen(tmp)` $\ll$ `INPUT_BUF_SIZE` -1), but does deal with the `cbmc` problem and is safe when `input` guarantees[2] `strlen(tmp)` $\leq$ `INPUT_BUF_SIZE` -1.

```
#define get_string(tag, dst){ printf (tag); \
                              tmp = input (INPUT_BUF_SIZE); \
                              if (strlen(tmp) == 0) { check_abort } \
                              dst = my_malloc(INPUT_BUF_SIZE * (sizeof (char))); \
                              if (!dst) { goto ABORT_INPUT; } \
                              strcpy(dst, tmp);}
```

To check the address book application for absence of `NULL` pointer dereferences—in particular, checking that the result of a memory allocation is never dereferenced without a runtime-test ensuring that a non-`NULL` pointer was returned, we instrumented the `malloc` function using a non-deterministic choice (boolean) function to model the possibility of a memory allocation failure:

```
#ifdef __VERIFY
_Bool nondet_bool ();
void *my_malloc (size_t s)
{
  if (nondet_bool()) { return (NULL); }
  return (malloc (s));
}
(...)
#endif
```

Then we added `assert` statements, not only everywhere a pointer was dereferenced after a `malloc`, but also to check that the functions are used with correct preconditions (instead of extra runtime `if`-checks which result in extra overhead, compared to `assert` statements which can be turned off in the final, optimized release version). In general, we also added explicit initialization and used intermingled declarations of variables and code if possible for increased readability (which are C99 language features which were already used in the original program).

```
struct list_element *list_new_elem (struct address *addr)
{
  assert(addr);

  struct list_element *elem = my_malloc (sizeof (struct list_element));
  if (!elem) { return (NULL); }

  assert(elem);
  elem->prev = elem;
  (...)
}
```

[1] See `http://stackoverflow.com/questions/252782`

[2] See Task 4 for the buffer overflow fix in `input`.

```
struct list *list_new (void)
{
  struct list *list = my_malloc (sizeof (struct list));
  if (!list) { return (NULL); }

  assert(list);
  list->first = NULL;
  (...)
}

void list_append (struct list *list, struct address *addr)
{
  assert(list);
  assert(addr);

  struct list_element *elem = list_new_elem(addr);
  if(!elem) { return; }
  (...)
}

void list_remove (struct list *list, struct list_element *elem)
{
  assert(list);
  if (!elem) { return; }
  (...)
}

void list_release (struct list *list)
{
  assert(list);
  (...)
}

struct address *input_addr ()
{
  (...)
  struct address *address = my_malloc (sizeof(struct address));
  if(!address) { goto ABORT_INPUT; }

  assert(address);
  assert(name);
  assert(addr);
  assert(phone);

  address->name    = name;
  address->address = addr;
  address->phone   = phone;
  return (address);
  (...)
}

void print_addr (struct address *addr)
{
  assert(addr);
  (...)
}

void list_addrs (struct list *list)
{
  assert(list);
  struct list_element *elem = list->first;
  (...)
```

```
}

void del_addr (struct list *list)
{
  assert(list);
  struct list_element *elem = list->first;
  (...)
}

int cmp_addr (struct address *addr, char *str)
{
  assert(addr);
  assert(str);
  (..)
}

void search_addr (struct list *list)
{
  assert(list);
  (...)
}
```

We bounded all loops to three iterations—and turned off the unwinding assertions (and thus used `cbmc` 'only' for bug hunting, not full verification). We did this because the while loop in the main function has no (useful) run-time bound. We also limited the buffer size to 8 (instead of the original 1024) to limit the number of branches (this should not have an influence on the result of the `cbmc` verification for the objectives of this task). These parameters were chosen to limit the memory usage for the SAT solver.

```
cbmc addrbook.c -D __VERIFY --unwind 3 --no-unwinding-assertions
```

The following assertion violations were found by `cbmc` and fixed:

```
Violated property:
  file addrbook.c line 251 function del_addr
  assertion list
  FALSE
```

```
int main (int argc, char **argv)
{
  /* list initialisation */
  struct list *list = list_new();
  if (!list) { abort(); } // abort the program if empty address book list cannot
      be created
  (...)
}
```

```
Violated property:
  file addrbook.c line 205 function input_addr
  assertion address
  FALSE
```

```
struct address *input_addr ()
{
  (...)
```

```
  address = my_malloc (sizeof(struct address));
  if (!address) { goto ABORT_INPUT; } // NULL pointer dereference fixed
  (...)
}
```

```
Violated property:
  file addrbook.c line 105 function list_append
  assertion addr
  FALSE
```

```
int main (int argc, char **argv)
{
  (...)
  char c = 0;
  /* main loop */
  while (c != 'q')
    {
    (...)
    switch (c) {
      case 'a':
        struct address *addr = input_addr();
        if (addr) { list_append (list, addr); } //only append address if it was
            created succesfully
        break;
      (...)
    }
}
```

Task 2

The stack of integers uses a singly linked list based on the following data structure:

```
// forward declaration of struct and typedef
typedef struct node Node;

// Linked list data structure
struct node {
    int data;
    Node* next;
};
```

Popping from an empty stack has in general undefined behavior.[3] Thus we chose to include a function to check if a stack is empty so that the programmer can check the stack before popping. If a pop happens from an empty stack, the program displays an error message and aborts.

```
void push(Node** headRef, int newData) {
    // allocate node
    Node* newNode = (Node*) my_malloc(sizeof(Node));
    if (!newNode) return;

    newNode->data = newData;
    newNode->next = (*headRef);
```

[3]See http://stackoverflow.com/questions/7390126

```
    (*headRef) = newNode;
}

int pop(Node** headRef) {
    Node* head = *headRef;
    if(!head) { fputs("Error: pop from empty stack\n", stderr);  abort();  }
    int result = head->data; // pull out the data before the node is deleted
    *headRef = head->next;
    my_free(head); // free the head node
    return result;
}

int is_empty(Node* head) {
    return head == NULL;
}
```

Our test harness consists of a `main` which simply initializes a empty stack, pushes a large number of integers (value i in the ith iteration) and then popping until the stack is empty and checking the bounds of the popped elements. We first tested the program using some `printf` statements for 1024 elements by running it manually.

```
int main() {
    Node* stack = NULL; //initialize stack
    int i,popped;
    for (i=0; i < NB_ELEMENTS; ++i) { push(&stack, i);  }
    while( stack ) {  popped = pop(&stack); assert( popped >= 0 && popped <
        NB_ELEMENTS); }
    assert(counter==0);
    return 0;
}
```

To check the program for the absence for `NULL` pointer dereferences, buffer overflows and pointer-safety errors using `cbmc`, we then again instrumented the `malloc` function (as in Task 1) and ran the program with the `--pointer-check` and `--bounds-check` options (first separately, then together). We also added a counter for keeping track of the number of `malloc`s minus `free`s to check for memory leaks by putting a `assert` statement at the end that this number is zero.

```
#ifdef __VERIFY
int counter = 0;
_Bool nondet_bool ();
void *my_malloc (size_t s)
{
  if (nondet_bool()) { return (NULL); }

  counter++;
  return (malloc (s));
}

void my_free(void* arg)
{
  free(arg);
  counter--;
}
(...)
#endif
```

The while loop in the main function has no (useful) run-time bound. Thus, the `--unwind` parameter has to be used—using (trivial) loop-bounds, that is, the number of elements—in order to prevent infinite unwinding.

When we tried to verify the program using the test harness for many elements ($n > 1000$), the verification step took too long to complete. This is a manifestation of the state explosion problem, caused by the fact that when creating an element, it can have either a successful or failed `malloc` operation, which results in 2^n possible paths. A couple of computation times are shown: they confirm that `cbmc` does not scale well for a large number of elements in this program.

n	4	8	12	16	20	24
time (s)	0.1	0.9	15	199	1828	11728

Combinations with `--bounds-check` and/or `--pointer-check` options also do not pose extra problems, e.g., for $n = 8$, following verification succeeds in 1.6 s.

```
cbmc stack_int.c -D __VERIFY --unwind 9 --pointer-check --bounds-check
```

When using smaller loop bounds, `cbmc` obviously reports unwinding assertion errors.

Task 3

We extracted the doubly-linked list of the address book application. To test this implementation, we changed the data structure to use integers as data and used the same test harness as in Task 2. This avoids testing the whole address book application, as the address book application in essence is just this data structure with some higher-level address book helper functions.

```
struct list_element {
 struct list_element *prev;
 struct list_element *next;
 int data; // changed struct address pointer to integer data
};
```

We have also applied the workaround to avoid multiple dereferences in one assignment by introducing temporary variables, e.g.,

```
//elem->prev->next = elem->next;
struct list_element *elem_prev_element = elem->prev;
elem_prev_element->next = elem->next;
```

This finally results in a successfully verified program using all options:

```
cbmc -D __VERIFY --unwind 9 dllist.c --pointer-check --bounds-check
```

The same problem with scalability as in Task 2 arises, so therefore only a small number of elements were tested (e.g., $n = 8$ takes about 2 minutes).

Task 4

We limit the buffer size to 4 so that we can easily set a small global loop bound of 5:

```
cbmc addrbook.c -D __VERIFY
     --unwind 5 --no-unwinding-assertions
     --pointer-check --bounds-check
```

```
Violated property:
  file addrbook.c line 81 function input
  array 'buf' upper bound
  (long int)i < 4
```

This problem arises when the null character is written at index i equal to `INPUT_BUF_SIZE` exceeding the length of the array, and was fixed by adjusting the while loop condition. We also rewrote the function somewhat to improve readability and remove the need to check the `c!='\n'` condition twice.

```
char *input (int n)
{
  static char buf[INPUT_BUF_SIZE];
  char c;
  int i = 0;
  while ( i <= n && i < (INPUT_BUF_SIZE - 1) ) //fixed buffer overflow
      vulnerability
   {
     c = getchar();
     printf ("%c", c);
     if (c != '\n' ) { buf[i] = c; i++; }
     else { break; } // we can immediately break out of the loop
   }

  buf[i] = '\0';

  return (buf);
}
```

The second problem is a memory leak which occurs when an address entry is created successfully (by `input_addr`) but cannot be appended to the list. This could not be known to the caller of `list_append` and consequently, the allocated memory for the address entry was not freed.

```
Violated property:
  file addrbook.c line 396 function main
  assertion counter == 0
  FALSE
```

We chose to fix this problem by letting the function `list_append` return a success code so that the caller can free the previously created element if needed.

```
int list_append (struct list *list, struct address *addr)
{
  (...)
  struct list_element *elem = list_new_elem(addr);
```

```
  if (!elem) { return 0; } //failed to create list element from address element

  (...)
  return 1; //successfully added address element
}

int main (int argc, char **argv)
{
  (...)
  while (c != 'q')
  {
    (...)
    switch (c) {
     case 'a':   struct address *addr = input_addr();
                 if (addr) {
                    if (!list_append (list, addr)) {
                        my_free(addr->name);
                        my_free(addr->address);
                        my_free(addr->phone);
                        my_free(addr);
                    }
                 }
                 break;
    (...)
    }
  }

  /* cleanup */
  list_release (list);
  (...)
  assert(counter == 0);
  return (0);
}
```

Task 5

As a first guideline we would suggest to write enough assert statements. When using `cbmc` for writing safe C code, it is a good practice to use a custom `malloc` instrumentation to also test the program in presence of memory allocation failures. The use of a counter variable to count the number of `malloc`s and `free`s also detects memory leaks. It is recommended to test the code regularly during the development phase, since memory leaks are very common in C programs and very hard to find.

The counterexamples given by `cbmc` are somewhat difficult to interpret but when analyzed carefully they provide good insight in somewhat subtle bugs (as we experienced personally for memory leaks in the address book application). Memory leaks can be spotted by examining the trace of the counterexample by searching memory (de)allocations that aren't followed by a changing counter.

As a final note, we should note that the approach to manually run `cbmc` with the different options is very tedious. Using scripts of course can alleviate this problem, but ultimately, a tight integration with an IDE is invaluable for the development progress[4] of real-world complex applications. `cbmc` still has some obscure bugs and quirks such as the

[4]We also did not manage to get the `cbmc` plugin for Eclipse working.

fixes needed for `strdup` or the multiple dereference problem. Nevertheless, using `cbmc` (as any other verification tool) forces the programmer to more carefully think about the code and its potential problems, and helps finding vulnerabilities and bugs.

Task 6

It is not clear how `cbmc` should be used to detect format string vulnerabilities [3, 4]. However, *possible* format string vulnerabilities are usually very easy to detect:

- manually, which can be done only on short programs;
- using a `grep`-like approach (`PScan`) or more advanced static analysis tools such as `flawfinder`, `RATS` and `ITS4`;
- using compiler flags (`gcc` using e.g. `-Wformat` which was even a default parameter on our system);
- using existing compiler extensions such as FormatGuard [1] which a.o. perform argument counting.

E.g., taking the most basic format string vulnerability example in [3]

```
void vulnfunc (char *user)
{
    printf(user);
}
```

and compiling with `gcc` gives following warning:

```
string_vul.c: In function ‘vulnfunc’:
string_vul.c:7: warning: format not a string literal and no format arguments
```

Task 7

The following code fragment contained a format string vulnerability and was easily fixed:

```
void print_addr (struct address *addr)
 {
  assert(addr);
  /*printf (addr->name); printf("; ");  //format string vulnerability
  printf ("%s; %s\n", addr->address, addr->phone);*/

  printf ("%s; %s; %s\n", addr->name, addr->address, addr->phone);
 }
```

The following macro contained a possible format string bug if used incorrectly by the programmer (`printf(tag)`). However all uses in the program are internally and pass a literal string, so this fix is not really needed as opposed to the previous one (also see Task 8).

```
#define get_string(tag, dst) { printf("%s", tag); (...) }
```

Task 8

As mentioned in Task 7, the vulnerability is in the name element of an address when printed using `print_addr`. Thus an attacker can chose to add an address book entry ('a') with a carefully chosen string with control arguments for the name (e.g., to crash the program or to view the stack contents) and then execute the list entries command ('l').

For example, the following commands result in a crash of the address book program due to a *Segmentation fault*.

```
int main (int argc, char **argv)
{
  /* list initialisation */
  struct list *list = list_new ();
  if (!list) { abort(); }
  (...)
  struct address *address1 = malloc (sizeof(struct address));
  if(!address1) { printf("failed to allocate memory for an address element\n");
      abort(); }
  address1->name    = "%n %n %n %n";
  address1->address = "test address";
  address1->phone   = "0123456789";

  list_append(list, address1);
  list_addrs (list);

  (...)
}
```

Additionally there is a problem in the following function:

```
void del_addr (struct list *list)
{
  (...)
  unsigned int i, n;
  printf ("delete #: ");
  tmp = input (INPUT_BUF_SIZE);
  n = atoi (tmp); //undefined behavior possible or unwanted behavior
  (...)
}
```

The user can provide a non-valid number for the entry he wishes to remove, and n will be set to 0 (`atoi` does not support error reporting, so a valid zero cannot be distinguished), which will result in the deletion of the first element. Also, the user can give a number string such as '99999999999999999999' causing an integer overflow which will result in undefined behavior (e.g., crash, random number, ...). This security vulnerability should probably be fixed by using `strtol` which provides error checking.[5] This was not done because this was not detected by `cbmc`.

Task 9

We already knew some software testing techniques such as unit testing, and more in general software development as a process, including test cycles, requirements/security

[5] See `http://stackoverflow.com/questions/3420629/`

analysis, ...

Previously, for formal verification we only had real experience with (symbolic) model checkers such as NuSMV `http://nusmv.fbk.eu/`. The advantage of those tools that verification of very complex properties can be done, and counterexamples can be given which provide valuable insight in subtle bugs; however they are quite difficult to work with and are not immediately usable for general purpose languages. Tools such as `splint` need annotations to work properly, and therefore require considerable more effort of the programmer. We also tried to use `valgrind`, but failed to detect the memory leak as found in Task 4, probably due to faulty usage.

It is important to note that all verification techniques are currently an important topic of research. While some of these tools are very mature and really usable, others are very user-unfriendly and have bad support. Thus these tools should not only be able to detect many kinds of potential vulnerabilities, but they should do it with a low false positive rate, be scalable for real-world applications and give a good explanation or even a solution.

A very nice and comprehensive overview of the different verification techniques and their advantages and disadvantages is given in [2]:

> Static analysis techniques based on abstract interpretation scale well at the cost of limited precision, which manifests itself in a possibly large number of false warnings. The tool support is mature.
>
> Model Checking tools that use abstraction can check complex safety properties, and are able to generate counterexamples. Bounded Model Checkers are very strong at detecting shallow bugs, but are unable to prove even simple properties if the program contains deep loops.
>
> The model checking-based tools for software are less robust than static analysis tools and the market for these tools is in its infancy.
>
> The challenges for future research in software analysis are dynamically allocated data structures, shared-variable concurrency, and the environment problem.
>
> None of the tools we survey is able to assert even trivial properties of dynamically allocated data structures, despite the large body of (theoretical) research in this area. Similarly, concurrency has been the subject of research for decades, but the few tools that analyze programs with shared-variable concurrency still scale poorly.
>
> Any formal analysis requires a formal model of the design under test—in case of software, this model often comprises a non-trivial environment that the code runs in (libraries, other programs, and hardware components). As programs often rely on properties of this environment, substantial manual effort is required to model these parts of the environment.

In summary, all these tools make different trade-offs between coverage, soundness, usability, performance, ... Programmers should not blindly trust these formal verification tools: a successfully verified program can still have severe problems, and sometimes false positives are given. *A fool with a tool is still a fool.*

References

[1] C. Cowan, M. Barringer, S. Beattie, G. Kroah-hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*, 2001.

[2] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[3] K.-S. Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33:423–460, April 2003.

[4] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.