Text for comment rhs-10 of document N1618

Annex A
(informative)
Intra- to Interprocedural Transformations

Most of the examples given in the individual rules are very simple and most often intraprocedural, only requiring analysis of a single function to either diagnose the violation or determine that there is no violation. Real-world code will often be much more complicated. Often some form of interprocedural analysis is required. Collecting evidence for or against a violation may require examining the flow of data from one function to another, either explicitly through argument passing and return values or through global variables and the heap. There are many different approaches to interprocedural analysis and this standard does not advocate one over another. This annex describes a transformational framework that can be used to generate more complex interprocedural examples from the simple intraprocedural examples given in the rules. These generated examples can then be used as an extended set of requirements for interprocedural analysis.

A.1 Function arguments and return values

The simplest case is a rule involving only one value, such as **Failing to detect and handle standard library errors** [liberr]. The following is an intraprocedural example (the fact that liberr violations are intrinsically interprocedural is irrelevant here, since the interprocedural aspect is built into the rule itself):

```
FILE *fp = fopen(name, mode);
if (fp != NULL) /* checking for success */
    ...
```

The basic interprocedural transformations are to pass the value into a function or return it from a function:

```
void check_it(FILE *fp)
{
    if (fp != NULL) /* checking for success */
        ...
}

...
check_it(fopen(name, mode));

/* a "wrapper" around fopen */
FILE *xfopen(const char *name, const char *mode)
{
    return fopen(name, mode); /* return for checking elsewhere */
}

...
FILE *fp = xfopen(name, mode);
if (fp != NULL) /* checking for success */
    ...
```

An important special case combines argument passing and returning to form an "identity" operation:

```
/* trivial example */
FILE *identity(FILE *fp)
{
    return fp;
}
```

```
FILE *fp = identity(fopen(name, mode));
if (fp != NULL) /* checking for success *
    …
```

A.2 Indirection

An additional transformation is indirection:

```
void check_indirect(FILE **pfp)
{
    if (*pfp != NULL) /* checking for success */
        ...
}

...
FILE *fp = fopen(name, mode);
check_indirect(&fp);

void return_result_thru_param(const char *name, const char *mode,
                              FILE **result)
{
    *result = fopen(name, mode);
}

...
FILE *fp;
return_result_thru_param(name, mode, &fp);
if (fp != NULL) /* checking for success */
    ...
```

Indirection can also involve fields of structs or unions. Indirection can be applied recursively, although precise handling of indirection often becomes increasingly expensive as the number of levels of indirection increases.

When a rule involves multiple values, such as **Forming or using out-of-bounds pointers or array subscripts** [invptr] (where a violation is an interaction between an array and an index), these transformations apply separately or in combination to each of the values. The following is a simple intraprocedural example:

```
int array[2];
int index = 2; /* part of violation */
array[index] = 0; /* violation */
```

Applying some of the interprocedural transformations yields

```
void indexer(int *array, int index)
{
    array[index] = 0; /* part of violation */
}

...
int array[2];
int index = 2; /* part of violation */
indexer(array, index); /* violation */
```

or

```
static int array[2];
int *get_array()
{
    return array; /* part of violation */
}
```

```
...
get_array()[2] = 0; /* violation */
```

or

```
struct array_params {
    int *array;
    int index;
};

void indexer(struct array_params *ap)
{
    ap->array[ap->index] = 0; /* part of violation */
}

...
int array[2];
struct array_params params;
params.array = array; /* part of violation */
params.index = 2; /* part of violation */
indexer(&params); /* violation */
```

These violations involve three steps: the array, the index, and the address arithmetic, where each could occur in a different function:

```
int *add(int *base, int offset)
{
   return base + offset; /* part of violation */
}

...
int array[2];
int index = 2; /* part of violation */
*add(array, index) = 0; /* violation */
```

Indirection may also be applied to a pointer being returned from a function:

```
const int *ptr_to_index()
{
    static int rv;
    rv = 2; /* part of violation */
    return &rv; /* part of violation */
}
…
int array[2];
array[*ptr_to_index()] = 0; /* violation */
```

A.3 Transformation involving standard library functions

The following transformation involves tracing the flow of data through the C Standard Library function `strchr()` that returns a pointer to an element in the array specified by its first argument if the element's value equals that of the second argument, and a null pointer otherwise. Because the effects and the return value of the function are precisely specified, an analyzer can determine that the assignment to the `*slash` object, in fact, modifies an element of the `const` string `pathname`, potentially causing undefined behavior.

```
const char* basename(const char *pathname) {
  char *slash;

  slash = strchr(pathname, '/');
  if (slash) {
    *slash++ = '\0';   /* violates EXP40-C. Do not modify constant values */
```

```
    return slash;
  }

  return pathname;
}
```

Note that interprocedural analysis could identify such data flow through an arbitrary function, but in the case of standard library functions it is not necessary to analyze the function's implementation in order to derive the flow.

A.4 Data flow through globals

Interprocedural data flow may not be explicit through argument passing and value returning; it may involve extern or static variables:

```
int global_index;

void set_it()
{
    global_index = 2; /* part of violation */
}

…
int array[2];
set_it(); /* part of violation */
array[global_index] = 0; /* violation */
```

A.5 Data flow through the heap

Data may also flow from one function to another through heap locations:

```
struct some_record {
    int index;
    …
};

/* returns the heap-allocated record (created elsewhere)
 * matching 'key'
 */
struct some_record *find_record(const char *key);

void set_it()
{
    struct some_record *record = find_record("xyz");
    record->index = 2;
}

…
int array[2];
set_it(); /* part of violation */
struct some_record *record = find_record("xyz"); /* part of violation */
array[record->index] = 0; /* violation */

/* note that find_record() could even be called before set_it() */
```

A.6 Combined example

This example combines some of these transformations in a way that a reasonably sophisticated interprocedural analysis might diagnose:

```
struct trouble {
    int *array;
    int index;
    int *effective_address;
};

void set_array(struct trouble *t, int *array)
{
    t->array = array; /* part of violation *.
}

void set_index(struct trouble *t, int *index)
{
    t->index = *index; /* part of violation */
}

void compute_effective_address(struct trouble *t)
{
    t->effective_address = t->array + t->index; /* part of violation */
}

void store(struct trouble *t, int value)
{
    *t->effective_address = value; /* part of violation */
}

...
int array[2];
int index = 2; /* part of violation */
struct trouble t;
set_array(t, array); /* part of violation */
set_index(t, &index); /* part of violation */
compute_effective_address(&t); /* violation */
store(&t, 0); /* violation */
```