TrustInSoft results

TrustInSoft results on the Secure Coding Validation Suite

accfree

accfree_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

accfree_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

accfree_e03

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

TrustInSoft correctly detects another Undefined Behavior before reaching the "diagnostic required" line in this example. The call to `realloc()` is invalid, as `c_str1` is not a reallocable address (the declaration is `char s[MAX_LEN];`, then `s` is passed through the `c_str1` argument). See detailed results either with the GUI (click on the *Inspect with TrustInSoft Analyzer* button in the *Summary* tab) or look directly in the Analyzer Log tab:

```
tests/accfree/accfree_e03.c:76:[kernel] warning: Unclassified alarm: assert
\warning("free expects a free-able address");
                  possibly invalid address {{ &s }} for free.
                  stack: realloc :: tests/accfree/accfree_e03.c:76 <-
                         f :: tests/accfree/accfree_e03.c:71 <-
                         main
```

Now, if we correct this Undefined Behavior, we stumble at another problem - the expected Undefined Behavior, as described in the test, actually does not exist. There is no possible execution of this program where a double free happens. This is the description in the example:

```
 * Rule: [accfree]
 * Description: diagnostic is required because realloc may free c_str1
 *              when it returns NULL, resulting in c_str1 being freed
 *              twice.
 * Diagnostic: required on line 78
```

But in the C17 section *The realloc function* in paragraph 3, we can read:

> If size is nonzero and memory for the new object is not allocated, the old object is not deallocated.

The example's description directly contradicts the C17 standard, which states that if this call to `realloc` returns `NULL` then it cannot free `c_str1` in the same time.

Possible confusion may have been caused by the following statement in the C17 standard:

> If size is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated.

Also, Defect Report #400 (from February 2012) could suggest that the initial idea behind this test was calling `realloc()` with `size` equal zero:

> There are at least three existing `realloc` behaviors when NULL is returned; the differences only occur for a size of 0

So, for comparison, an example modified to call `realloc` with `size` equal zero was added and analyzed. In this case however TrustInSoft warns about another Undefined Behavior, caused by calling `realloc` with `size` equal zero - this is explicitly considered Undefined Behavior in the upcoming C2X standard.

accsig

Signal handling is out of scope.

accsig_e01

Result = **OUT OF SCOPE**

TrustInSoft does not handle signals.

addrescape

addrescape_e01

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

What happens at the "diagnostic required" line - assigning an *escaping address* to a global variable - is not Undefined Behavior.

TrustInSoft correctly detects Undefined Behavior here when the *escaping address* is actually used - on line 72 in the statement `puts(p);`.

addrescape_e02

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

What happens at the "diagnostic required" line - returning an *escaping address* from a function - is not Undefined Behavior.

TrustInSoft correctly detects Undefined Behavior here when the *escaping address* is actually used - on line 65 in the expression `!init_array()`.

addrescape_e03

Result = **NO UB**

Holding an *escaping address* in a local variable is not Undefined Behavior. As this *escaping address* is never actually used, there is no Undefined Behavior in this example.

alignconv

alignconv_e01

Result = **NOT IMPLEMENTED YET**

This is Undefined Behavior according to the C Standard - indeed there is no guarantee neither that `(int*)&c != 0` nor that `(char*)(int*)&c == &c`.

Expert quote:

> Ils ont raison, le premier exemple est UB d'après le standard, il n'y a pas de garantie que `(int*)&c != 0` ou que `(char*)(int*)&c == &c`. Par contre dans une discussion avec un développeur de GCC j'ai appris que c'était “presque comme si c'était documenté” (çad c'est documenté mais la documentation est une réponse à un bug report ou une discussion dans la mailing list des développeurs GCC) que GCC, pour les cibles qu'il vise, a une représentation uniforme des pointeurs et garantit exactement les deux propriétés dont il est question.

alignconv_e02

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

argcomp

argcomp_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

argcomp_e02

Result = **OK : TRUE POSITIVE**

Incompatible Declaration detected as expected.

argcomp_e03

Result = **OK : TRUE POSITIVE**

Incompatible Declaration detected as expected.

argcomp_e04

Result = **OK : TRUE POSITIVE**

Incompatible Declaration detected as expected.

asyncsig

Signal handling is out of scope.

asyncsig_e01

Result = **OUT OF SCOPE**

TrustInSoft does not handle signals.

asyncsig_e02

Result = **OUT OF SCOPE**

TrustInSoft does not handle signals.

asyncsig_e03

Result = **OUT OF SCOPE**

TrustInSoft does not handle signals.

boolasgn

boolasgn_e01

Result = **NO UB (infinite loop)**

Using an assignment expression (i.e. `x = y`) as a loop controlling expression is usually a typo and may cause unexpected behavior, but it is not Undefined Behavior.

Moreover, both `gcc` and `clang` find this kind of possible typos and suggest adding parentheses around the assignment expression if it was really intended to be an assignment and not a comparison.

boolasgn_e02

Result = **NO UB (infinite loop)**

Exactly the same explanation as in the previous case - `boolasgn_e01`.

boolasgn_e03

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

boolasgn_e04

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

boolasgn_e05

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

boolasgn_e06

Result = **NO UB**

I believe there is no Undefined Behavior in this example. Although I am not sure what the description means and what problem was expected to appear here:

```
* Rule: [boolasgn]
* Description: diagnostic is required because && is not a comparison
*              operator and the entire expression is not primary.
* Diagnostic: required on line 68
* Additional Test Files: None
* Command-line Options: None
```

Either way `gcc` and `clang` don't emit any warnings.

boolasgn_e07

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

chreof

chreof_e01

Result = **NO UB**

The problem here is that the function `getchar()` returns an integer value which can be either:

- an `unsigned int` value cast to `int` (values between 0 and 255)
- or the value of the macro `EOF` (which is a macro which expands to an integer constant expression, with type `int` and a negative value).

So if this is an `EOF` we cast this integer value to `char` or `unsigned char` the value for `EOF` (in many implementations equal `-1`) will become indistinguishable from one of the values corresponding to a normal correctly read character (if `EOF` is `-1`, then after casting to `unsigned int` it will become `255`).

Although this clumping together of `EOF` and one character is most probably not what the programmer intends, this is not causing directly Undefined Behavior - casting an `int` value to `unsigned char` or `char` is completely legal. TrustInSoft will detect a problem only if this causes Undefined Behavior later on.

See:

- About the `getchar()` function:

    - C17 Standard
    - Linux man page for getchar
- About the `EOF` macro - C17#7.21.1.p3

chreof_e02

Result = **NO UB**

Exactly like the precedent case `chreof_e01`, but for wide characters.

chrsgnext

chrsgnext_e01

Result = **NO UB**

TrustInSoft detects no Undefined Behavior here, because in this example all the values passed to `isspace()` are valid - they are representable as an `unsigned char`. And, according to the C17 Standard, `isspace()` accepts both all the values representable as `unsigned int` and the value of the `EOF` macro:

> In all cases the argument is an int, the value of which shall be representable as an unsigned char or shall equal the value of the macro EOF. If the argument has any other value, the behavior is undefined.

For comparison, I have added a second test where we pass the invalid value `-2` to `isspace()` and we can see that TrustInSoft correctly detects an Undefined Behavior.

NOTE: in `glibc`, the lookup table which is used inside the implementation of the `isspace()` function is defined in such a way that both `char` and `unsigned char` values are accepted - the valid range is between `-128` and `255`.

dblfree

dblfree_e01

Result = **NO UB**

Calling `free()` with null pointer as argument does nothing. We can repeat it as many times as we want - this is not Undefined Behavior. There is no possible execution of this program where a double free happens.

dblfree_e02

Result = **NO UB**

(Note that this is similar to `accfree_e03`.)

There is no possible execution of this program where a double free happens.

This is the decription in the example:

```
  * Rule: [dblfree]
  * Description: diagnostic is required because realloc may free c_str1
  *              when it returns NULL, resulting in c_str1 being freed twice
  * Diagnostic: required on line 88
```

But in the C17 section *The realloc function* in paragraph 3, we can read:

> If size is nonzero and memory for the new object is not allocated, the old object is not deallocated.

The description contradicts the C17 standard, which states that if this call to `realloc` returns `NULL` then it cannot free `c_str1` in the same time.

diverr

diverr_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

diverr_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

diverr_e03

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

diverr_e04

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

fileclose

fileclose_e01

Result = **NO UB**

Leaving a file open when program exits is technically not Undefined Behavior. However, as certain execution environments do not guarantee the open files to be in a coherent state after the program exits without closing them properly, this feature is in TrustInSoft's roadmap.

fileclose_e02

Result = **NO UB**

Memory leak is not Undefined Behavior. Still, TrustInSoft is capable of detecting such issues - the appropriate warning can be found in the *Analyzer Log* tab:

```
tests/fileclose/fileclose_e02.c:78:[value] warning: memory leak detected
for {__malloc_fun_l84}
```

filecpy

filecpy_e01

Result = **NO UB - NOT SURE**

This one is pretty unclear! Most probably it is not Undefined Behavior, but Unspecified Behavior.

The C17 Standard says:

> The address of the FILE object used to control a stream may be significant; a copy of a FILE object need not serve in place of the original.

The example copies the FILE object used to control the `stdout` stream and then uses that copy. The copy operation itself is not a problem, however when this copy is used then something bad might happen - apparently compiling and running this example in a particular environment (e.g. compiling in Microsoft Visual Studio 2013 and running on Windows) can cause "access violation" runtime error.

After some consideration we decided to categorize this as Unspecified Behavior, as:

- the operation of copying a FILE object is completely valid according to the C17 Standard,
- using the copied FILE object is not officially Undefined Behavior neither and does not seem to cause any trouble in most situations.

funcdecl

funcdecl_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

funcdecl_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

funcdecl_e03

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

funcdecl_e04

Result = **OUT OF SCOPE**

TrustInSoft expects that the compiler will not truncate variable names at 8 characters.

funcdecl_ex1

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

intoflow

intoflow_e01

Result = **NO UB**

Is this a typo? The test's description talks about integer overflow:

```
 * Rule: [intoflow]
 * Description: diagnostic is required on implementations that trap on
 *              signed integer overflow because the expression x + 1 may
 *              result in signed integer overflow
 * Diagnostic: required on line 79
```

However, the only variables that get incremented in this program ar `i` and `ui`:

- The variable `i` only goes from 1 to 10. So there cannot be a be a overflow here.
- And the variable `ui` is not a signed integer but an `unsigned int`. There cannot be a signed overflow on an `unsigned int` value.

Added a corrected version - changed `add(unsigned int ui)` to `add(int ui)` and now TrustInSoft detects this Undefined Behavior as expected.

intoflow_e02

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

intptrconv

intptrconv_005

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

What happens at the "diagnostic required" line - converting an integer number to a pointer - is not Undefined Behavior.

TrustInSoft correctly detects Undefined Behavior here when this value is is actually used in a comparison - on line 68 in the expression `c > 0`.

intptrconv_e01

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

What happens at the "diagnostic required" line - converting a pointer to an unsigned integer - is not Undefined Behavior.

TrustInSoft correctly detects Undefined Behavior here when this value is is actually used as an operand to a binary and operator - on line 80 in the expression `number & 0x7fffff`.

intptrconv_e02

Result = **NO UB**

What happens at the "diagnostic required" line - converting a constant number to a pointer - is not Undefined Behavior. Also what happens at the next line - returning such a value from a function - is not Undefined Behavior. This program is correct.

intptrconv_ex1

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

intptrconv_ex2

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

inverrno

inverrno_e01

Result = **NO UB**

Not setting `errno` to zero before calling a library function is not Undefined Behavior.

Although the usual coding pattern is to set `errno` before calling a library function that modifies `errno` upon encountering an error, doing things differently is not always an error. Another common pattern is also chaining calls to multiple library functions one after another without checking the `errno` value in between. As no library function is allowed to set `errno` to zero upon successful completion, this way we can check its value just one at the end of such a chain and know if an error occurred somewhere on the way.

Moreover, as `errno` is always initialized to zero, in this particular example it is actually still equal zero when the library function `strtoul` is called, so no problem can occur - therefore no misunderstanding can occur here.

inverrno_e02

Result = **NO UB**

Not checking the return value of `signal()` before checking the value of `errno` is not Undefined Behavior.

As in the previous case: although the usual coding pattern in case of functions like `signal()`, which can indicate an error using their return value, is to check the return value before checking the `errno` value (because if `signal()` succeeds it does not modify the `errno` value, so it could possibly be set by some previous library function call), doing things differently is not always an error. Again, the chaining multiple library calls pattern is a good example when this is OK.

Moreover, as `errno` is always initialized to zero and `signal()` is the only function that can modify `errno` in this particular example, so no previous function could have set it already - therefore, again, no misunderstanding can occur here.

inverrno_e03

Result = **NO UB**

Checking the value of `errno` after a call to `setlocale()` is not Undefined Behavior.

Although `setlocale()` does not modify `errno` upon encountering an error, checking the value of `errno` after such a call is not forbidden and might have sense in certain situations.

Note, that in this particular example TrustInSoft can find that the `if` branch where `errno` is not equal zero is dead code and will show it in red in the GUI (the *Inspect with TrustInSoft Analyzer* button in the *Summary* tab).

invfmtstr_002

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invfmtstr

invfmtstr_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invptr

invptr_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invptr_e02

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e03

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e04

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

What happens at the "diagnostic required" line - converting a pointer to an unsigned integer - is not Undefined Behavior.

The first Undefined Behavior that happens in this program is not on the "diagnostic required" line. Before defereferencing past the end of the `name` buffer the program will perform memory access in the while loop controlling expression `*path != '\\'` which is the Undefined Behavior that TrustInSoft detects here.

A modified version of this test with the string `str` made abstract was added. For this version TrustInSoft finds the desired Undefined Behavior.

invptr_e05

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e06

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invptr_e07

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e08

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invptr_e09

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e10

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

invptr_e11

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

invptr_e12

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

ioileave

ioileave_e01

Result = **NOT IMPLEMENTED YET**

Interleaving input and output operations on a file without an intervening flush or positioning call is Undefined Behavior according to the C Standard. TrustInSoft currently does not detect it.

liberr

liberr_e01

Result = **NO UB**

Not checking the return value of `fseek()` for error conditions is not Undefined Behavior. Also even in case of an error happening in `fseek()`, passing concerned file subsequently to `fread()` is not Undefined Behavior neither.

liberr_e02

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

liberr_ex1

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

liberr_ex2

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

libmod

libmod_e01

Result = **NOT IMPLEMENTED YET**

Modifying the string returned from `setlocale` is indeed Undefined Behavior. TrustInSoft currently does not detect it.

libmod_e02

Result = **NOT IMPLEMENTED YET**

Modifying the string returned from `localeconv` is indeed Undefined Behavior. TrustInSoft currently does not detect it.

libmod_e03

Result = **NOT IMPLEMENTED YET**

Modifying the string returned from `getenv` is indeed Undefined Behavior undetected currently by TrustInSoft.

libmod_e04

Result = **NOT IMPLEMENTED YET**

Modifying the string returned from `strerror` is indeed Undefined Behavior undetected currently by TrustInSoft.

libptr

libptr_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

libptr_e02

Result = **NO UB**

In this example Undefined Behavior only happens if the size of `int` is larger than the size of `float`. Unfortunately all the architectures that TrustInSoft currently handles the size of `int` is smaller or equal than the size of `float`. Therefore the call to `memset` is always valid - it will never go out of the array's bounds.

Architecures where this would be an Undefined Behavior exist (e.g. ilp64) and being able to model them is in TrustInSoft's roadmap.

libptr_e03

Result = **NO UB**

The variable `n` will be equal to size of type `int`, which is smaller that the size of type `double`. Therefore the call `memcpy(p, q, n)` may have unexpected results, but all no invalid read nor write is possible, so no Undefined Behavior will happen here.

In order to have Undefined Behavior here it would be necessary for size of type `int` to be larger that the size of `double`.

libptr_e04

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

libptr_e05

Result = **NO UB**

As `q` is a `wchar_t` pointer and when it is allocated the result of `malloc()` is casted to `wchar_t *`, most probably the intention of programmer was to compute `n` as size of the type `wchar_t` times length of wide string `L"Hello, World!"`. Instead `sizeof(p)` returns the size of a pointer, not size of the type `wchar_t *`. The desired expression was probably `sizof(*p)`.

However, this is not Undefine Behavior by itself. And there is no way to make it an Undefined Behavior, because the allocated buffer is never used.

libuse

libuse_e01

Result = **NOT IMPLEMENTED YET**

There is effectively an Undefined Behavior caused to accessing the results of first call to `getenv()` after calling `getenv()` again in this example. TrustInSoft currently does not detect it.

libuse_e02

Result = **NOT IMPLEMENTED YET**

There is effectively an Undefined Behavior caused to accessing the results of first call to `setlocale()` after calling `setlocale()` again in this example. TrustInSoft currently does not detect it.

libuse_e03

Result = **NOT IMPLEMENTED YET**

There is effectively an Undefined Behavior caused to accessing the results of first call to `strerror()` after calling `strerror()` again in this example. TrustInSoft currently does not detect it.

nonnullstr

nonnullstr_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

nonnullstr_e02

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

The string `cur_msg`, passed to `wcslen()`, is not only null-terminated, it is completely uninitialized. This is the Undefined Behavior that TrustInSoft detects here.

Corrected example was created, where the string is initialized. We can see that TrustInSoft still detects the Undefined Behavior in the same place, this time because the string is invalid - not null-terminated.

nonnullstr_e03

Result = **UB**

Again same thing happens as in the previous example - the string passed to `wcslen()` is completely uninitialized - so TrustInSoft detects Undefined Behavior.

And again a corrected example was created, where the string is initialized. We can see that in this case TrustInSoft detects the Undefined Behavior caused by not null-terminated string, as desired.

nullref

nullref_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

padcomp

padcomp_e01

Result = **NO UB**

There are two reasons why there is no Undefined Behavior detected here:

- As the variables `a` and `b` are static (their declaration is `static buffer a, b;`), both structures are zero-initialized, including their padding. Therefore the padding is initialized and can be read.
- As the structures `a` and `b` differ already on the value of their first field (for the structure `a` the value of `buff_type` is `'a'`, for the structure `b` it's `'b'`), the `memcmp()` function does not go any further in order to return the result. Therefore in such situation it will not read the padding data at all.

A corrected example was added that removes both reasons:

- The definition of `a` and `b` was changed to a non-static one, i.e. `buffer a, b;`.
- The values of the field `buff_type` of `a` and `b` were made the same.

Now padding data will be actually accessed by `memcpy()` and so TrustInSoft correctly detects this as Undefined Behavior.

ptrcomp

ptrcomp_e01

Result = **OK**

The Undefined Behavior concerns strict aliasing properties - the appropriate warning can be found in the *Analyzer Log* tab:

```
tests/ptrcomp/ptrcomp_e01.c:81:[sa] warning: The pointer ip has type int *.
It violates strict aliasing rules by accessing
              a cell with effective type float.
              Callstack: test_function :: tests/ptrcomp/ptrcomp_e01.c:66 <-
main
```

ptrobj

ptrobj_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

ptrobj_ex1

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

ptrobj_ex2

Result = **TYPO**

There is a typo in the example - `subrtact` instead of `subtract`.

A corrected example, with `subrtact` changed to `subtract`, was added. As expected, TrustInSoft does not detect any Undefined Behavior ins such case.

resident

OUT OF SCOPE

Checking for reserved identifiers is out of scope of TrustInSoft.

resident_e01

Result = **OUT OF SCOPE**

Defining reserved identifier `errno`.

resident_e02

Result = **OUT OF SCOPE**

Defining reserved symbol `_RESIDENT_HEADER_H`.

resident_e03

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

resident_e04

Result = **OUT OF SCOPE**

Defining reserved file scope identifier `_limit`.

resident_e05

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

resident_e06

Result = **PARTLY OK, PARTLY OUT OF SCOPE**

TrustInSoft detected correctly reusing the identifier `SIZE_MAX`.

Still, defining reserved identifier `INTFAST16_LIMIT_MAX` is out of scope.

resident_e07

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

resident_e08

Result = **OUT OF SCOPE**

Defining reserved identifiers `malloc()` and `free()`.

resident_e09

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

restrict

restrict_e01

Result = **NO UB**

The memory zones passed to `memcpy` do not overlap here, so this example does not contain Undefined Behavior.

A corrected version of this example, where the memory zones passed to `memcpy` were made to overlap. In this case TrustInSoft detects the desired Undefined Behavior.

restrict_e02

Result = **NOT IMPLEMENTED YET**

TrustInSoft does not handle the qualifier `restrict` in general.

sigcall

sigcall_e01

Result = **OUT OF SCOPE**

TrustInSoft does not handle signals.

signconv

signconv_e01

Result = **NO UB**

The `EOF` macro expands to an integer constant expression, with type `int` and a negative value (see C17#7.21.1.p3). Most often it's implemented as `-1`. And, as the values of type `char` can be both positive and negative, one of them can have the same value as `EOF`. So, if we want to convert `char` to `int` and keep the distinction between `EOF` and the actual character with value `-1`, we usually first cast the `char` value to `unsigned char` (which can have only non-negative values) and then cast the result to `int`. This way no overlap is possible.

However, this is just a good coding practice - not following such a pattern is not Undefined Behavior, as casting a `char` value to an `int` value is perfectly legal.

sizeofptr

sizeofptr_e01

Result = **NO UB**

Most probably the intention of the programmer when writing `sizeof(array)` was to compute the whole size of the array `i`, i.e. `int[10]`. Instead this the expression will have the value of the size of a pointer. So instead of iterating over the whole array, the `for` loop will most probably iterate over just two first cells (the exact number might depend on the architecture, as it depends on the size of `int` type and size of a pointer). This may be unexpected behavior, but the program will not cause Undefined Behavior.

Using the TrustInSoft GUI (click on the *Inspect with TrustInSoft Analyzer* button in the *Summary* tab) we can inspect the values of `i` in this case.

NOTE: By the way, it is forbidden to give incomplete types, like `int array[]`, as arguments to `sizeof`. However, the type of `int array[]` is actually complete in this particular case, as it is used for a function argument, so can be safely used as an argument to 'sizeof'. See C17#6.7.6.2.p4.

strmod

strmod_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

strmod_e02

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

strmod_e03

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

strmod_e04

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

strmod_e05

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

strmod_e06

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

swtchdflt

swtchdflt_e01

Result = **NO UB**

Having a non-exhaustive switch statement is not Undefined Behavior.

syscall

Handling calls to `system()` is out of scope.

Using TrustInSoft to analyze where does the data passed to arguments of the `system()` function comes from is possible if needed, but is not done automatically.

syscall_e01

Result = **OUT OF SCOPE**

TrustInSoft does not handle calls to `system()`.

syscall_e02

Result = **OUT OF SCOPE**

TrustInSoft does not handle calls to `system()`.

taintformatio

taintformatio_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

taintformatio_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

taintnoproto

taintnoproto_e01

Result = **NO UB**

This case is almost exactly the same as `sizeofptr_e01`.

taintsink

taintsink_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

taintsink_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

taintstrcpy

taintstrcpy_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

uninitref

uninitref_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

uninitref_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

uninitref_e03

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

uninitref_e04

Result = **MISPLACED "REQUIRED DIAGNOSTIC"**

Undefined Behavior detected as expected, but the actual first access to unitialized memory happens just before the "diagnostic required" line - in the expression `a[i] < 0`.

usrfmt

usrfmt_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

NOTE: There is more than one problem detected here.

usrfmt_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

NOTE: There is more than one problem detected here.

usrfmt_e03

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

NOTE: There is more than one problem detected here.

usrfmt_e04

Result = **OK : TRUE NEGATIVE**

No Undefined Behavior detected, as expected.

xfilepos

xfilepos_002

Result = **NOT IMPLEMENTED YET**

Checking if `fsetpos()` argument always comes from a previous successful call to `fgetpos()` is currently not implemented in TrustInSoft.

xfilepos_e01

Result = **NOT IMPLEMENTED YET**

Checking if `fsetpos()` argument always comes from a previous successful call to `fgetpos()` is currently not implemented in TrustInSoft.

xfree

xfree_e01

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.

xfree_e02

Result = **OK : TRUE POSITIVE**

Undefined Behavior detected as expected.