# Une analyse pour détecter les violations de strict aliasing

Pascal Cuoq & Loïc Runarvot

TrustInSoft

Saturday 16th July, 2016

# Summary

# *strict aliasing* rule

- Introduced in C89 standard
- Updated and clarified in C99/C11 standards
- The name “strict aliasing” from GCC option *circa* 1999
- Involve two concepts of types: “effective type” and “declared type”

## C11 Standard 6.5p7

*An object shall have its stored value accessed only by an lvalue expression that has one of the following types:*

*[...]*

- *qualified/signed/unsigned corresponding type of the object's effective type.*
- *a character type.*

*[...]*

```c
int f(int *p, float *q)
{
    *p = 42;
    *q = 0.01;
    return *p;
}

int main(void)
{
    int x = 0;
    return f(&x, (float *) &x);
}
```

## gcc -O2 -std=c11

gcc 5.3

```
f:
        movl    $42, (%rdi)
        movl    $42, %eax
        movl    $0x3c23d70a, (%rsi)
        ret
main:
        movl    $42, %eax
        ret
```

## clang -O2 -std=c11

clang 3.8

```
f:
        movl    $42, (%rdi)
        movl    $1008981770, (%rsi)
        movl    $42, %eax
        retq
```

```c
int f(int *p, float *q)
{
    *p = 42;
    *q = 0.01;
    return *p;
}

int main(void)
{
    int x = 0;
    return f(&x, (float *) &x);
}
```

## gcc -O2 -std=c11

gcc 5.3

```
f:
        movl    $42, (%rdi)
        movl    $42, %eax
        movl    $0x3c23d70a, (%rsi)
        ret
main:
        movl    $42, %eax
        ret
```

## clang -O2 -std=c11

clang 3.8

```
f:
        movl    $42, (%rdi)
        movl    $1008981770, (%rsi)
        movl    $42, %eax
        retq
```

Neither GCC nor Clang warn here.

# Strict aliasing: situation as of 2016

- The rule is not well known by C developers
- Compilers (`GCC` and `Clang`) optimization aggressively assuming the *strict aliasing* rule is respected
- BUT they cannot tell when it's violated: weak warnings

## Firefox's crashreporter toolkit

```
# Use -fno-strict-aliasing by default since gcc 4.4 has periodic
# issues that slip through the cracks. [...]
```

## ghc runtime, *makefile*

```
# -fno-strict-aliasing is required for the runtime, because we often
# use a variety of types to represent closure pointers [...]
```

## python2.12, *configure*

```
# Python violates C99 rules, by casting between incompatible
# pointer types. GCC may generate bad code as a result of that,
# so use -fno-strict-aliasing if supported.
```

## Linux Kernel

Instead of solving the problem into the source code itself, the Linux Kernel set the option `-fno-strict-aliasing` to deactivate the optimization about *strict aliasing*.

# Allocated Memory

A specific rule for allocated memory

# Allocated Memory

A specific rule for allocated memory

## C11 Standard Foonote 87

*Allocated objects have no declared type.*

# Allocated Memory

A specific rule for allocated memory

## C11 Standard Foonote 87

*Allocated objects have no declared type.*

## C11 Standard 6.5p6

*[...] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. [...]*

```c
float fl;

int f(int *p, float *q)
{
    *p = 42;
    *q = 0.01;
    fl = *(float *)p;
    return *p;
}

int main(void)
{
    int *p = malloc(sizeof(int));
    return f(p, (float *) p);
}
```

## gcc -O2 -std=c11

gcc 5.3

```
f:
        movl    $42, (%rdi)
        movl    $0x3c23d70a, (%rsi)
        movl    (%rdi), %eax
        movl    %eax, fl(%rip)
        ret
```

## clang -O2 -std=c11

clang 2.8

```
f:
        movl    $42, (%rdi)
        movl    $1008981770, (%rsi)
        movl    (%rdi), %eax
        movl    %eax, fl(%rip)
        retq
```

# Beyond the standard: structs

```c
#include <stdlib.h>

struct s1 { int a; } s1;
struct s2 { int a; } s2;

int f(struct s1 *p, struct s2 *q)
{
    p->a = 42;
    q->a = 7;
    return p->a;
}

int main(void)
{
    int *p = malloc(sizeof(int));
    return f((struct s1 *)p,
             (struct s2 *) p);
}
```

## gcc -O2 -std=c11

gcc 5.3

```
f:
        movl    $42, (%rdi)
        movl    $42, %eax
        movl    $7, (%rsi)
        ret
```

## clang -O2 -std=c11

clang 2.8

```
f:
        movl    $42, (%rdi)
        movl    $7, (%rsi)
        movl    $42, %eax
        retq
```

# Summary

# TIS Analyzer

- Static analysis tool for C
- One major plugin: Value Analysis
- Made to scale
- Made to require as little user intervention as possible

# TIS Analyzer

- Static analysis tool for C
- One major plugin: Value Analysis
- Made to scale
- Made to require as little user intervention as possible

The *strict aliasing* analysis is made as a TIS Analyzer plugin
Relies on Value Analysis.

# Target of *strict aliasing* analysis

- Analyze large `C` projects to find *strict aliasing* violation.
- Implementation of the "__may_alias__" `GCC`'s extension (also supported by `Clang`)
- Accurate warnings.

# Quick example of "__may_alias__"

```
typedef
  float __attribute__((__may_alias__))
  float_a;

int f(int *p, float_a *q)
{
    *p = 42;
    *q = 0.01;
    return *p;
}

int main(void)
{
    int x = 0;
    return f(&x, (float_a *) &x);
}
```

## gcc -O2 -std=c11

gcc 5.3

```
f:
        movl    $42, (%rdi)
        movl    $0x3c23d70a, (%rsi)
        movl    (%rdi), %eax
        ret
```

## clang -O2 -std=c11

clang 2.8

```
f:
        movl    $42, (%rdi)
        movl    $1008981770, (%rsi)
        movl    (%rdi), %eax
        retq
```

# The Typing System

⟨*integer_type*⟩ ::= _Bool | short | int | long | long long

⟨*float_type*⟩ ::= float | double | long double

⟨*struct_info*⟩ ::= ...

⟨*union_info*⟩ ::= ...

⟨*field_info*⟩ ::= ...

⟨*simple_type*⟩ ::=
- void
- char
- MayAlias
- pointer of <simple_type>
- <integer_type>
- <float_type>
- `union of` ( <union_info>, <set_type> )
- `struct of` <struct_info>
- `field of` ( <field_info>, <set_type> )

⟨*set_type*⟩ ::= <simple_type>+

⟨*effective_type*⟩ ::= top | bottom | <set_type>

Translation of C type into effective type is direct.

# The Typing System

⟨*integer_type*⟩ ::= _Bool | short | int | long | long long

⟨*float_type*⟩ ::= float | double | long double

⟨*struct_info*⟩ ::= ...

⟨*union_info*⟩ ::= ...

⟨*field_info*⟩ ::= ...

⟨*simple_type*⟩ ::=
- void
- char
- MayAlias
- pointer of <simple_type>
- <integer_type>
- <float_type>
- `union of` ( <union_info>, <set_type> )
- `struct of` <struct_info>
- `field of` ( <field_info>, <set_type> )

⟨*set_type*⟩ ::= <simple_type>+

⟨*effective_type*⟩ ::= top | bottom | <set_type>

Translation of C type into effective type is direct.

## Exceptions

- If a type has the "___may_alias___" attribute, it's automatically translated into `MayAlias`.
- A field access (ie. with . or ->) is translate using the `'field of'` construction.

# Examples of *strict aliasing* analysis.

```
1  int f(int *p, float *q)
2  {
3    *p = 1;
4    *q = 2.0;
5    return *p;
6  }
7
8  int main(void)
9  {
10     int x;
11     return f(&x, (float *) &x);
12 }
```

## Execution output

```
$ gcc-5 strict.c ; ./a.out ; echo $?
0
$ gcc-5 -O2 strict.c ; ./a.out ; echo $?
1
$ clang-3.6 strict.c ; ./a.out ; echo $?
0
$ clang-3.6 -O2 strict.c ; ./a.out ; echo $?
1
```

## *strict aliasing* analysis output.

```
[...]
[value] computing for function f <- main.
        Called from strict.c:11.
[value] Recording results for f
[from] Computing for function f
[from] Done for function f
strict.c:4:[sa] warning: The lvalue `q' has type float *.
 It enters in conflict with its effective type int *.
[value] Done for function f
[...]
```

# Examples of *strict aliasing* analysis.

```
1  struct object { int id; };
2  struct thing { int id; int stuff; };
3
4  int g(struct object *p, struct thing *q)
5  {
6      p->id = 1;
7      q->id = 2;
8      return p->id;
9  }
10
11 int main(void)
12 {
13     struct thing saucer;
14     return g((struct object *) &saucer,
15              &saucer);
16 }
```

## Execution output

```
$ gcc-5 thing.c ; ./a.out ; echo $?
2
$ gcc-5 -O2 thing.c ; ./a.out ; echo $?
1
$ clang-3.6 thing.c ; ./a.out ; echo $?
2
$ clang-3.6 -O2 thing.c ; ./a.out ; echo $?
1
```

## *strict aliasing* analysis output.

```
[...]
[value] computing for function g <- main.
        Called from thing.c:14.
[value] Recording results for g
thing.c:8:[sa] warning: The lvalue `p->id' has type
 (struct object).id[int]. It enters in conflict with its
 effective type {(struct object).id[int],
                 (struct thing).id[int]}.
[value] Done for function g
[...]
```

# Examples of *strict aliasing* analysis.

```c
#include <stdlib.h>

struct X { int i; int j; };

int foo(struct X *p, struct X *q)
{
    q->j = 1;
    p->i = 0;
    return q->j;
}

int main(void)
{
    char *p = malloc(3 * sizeof(int));
    struct X *q = (struct X *)p;
    return foo((q + sizeof(int)), q);
}
```

## Execution output

```
$ gcc-5 krebbers.c ; ./a.out ; echo $?
0
$ gcc-5 -O2 krebbers.c ; ./a.out ; echo $?
1
$ clang-3.6 krebbers.c ; ./a.out ; echo $?
0
$ clang-3.6 -O2 krebbers.c ; ./a.out ; echo $?
1
```

## *strict aliasing* analysis output.

```
[...]
[value] computing for function foo <- main.
        Called from krebbers.c:15.
[value] Recording results for foo
krebbers.c:9:[sa] warning: The lvalue `q->j' has type
 (struct X).j[int]. It enters in conflict with its
 effective type (struct X).i[int].
[value] Done for function foo
[...]
```

# Summary

# What is “Expat” ?

- C library
- Parse XML files
- The first XML parser available inthe 1990s
- Widely used

# Example of structures used by Expat's parser

```
struct NAMED {
    char *name;
};

struct TAG {
    char *name;
    char *rawName;
    /* [...] */
};

struct ENTITY {
    char *name;
    char *textPtr;
    /* [...] */
};

struct HASH_TABLE {
    struct NAMED **v;
    size_t size;
    /* [...] */
};
```

- Structures share the same prefix.
- An attempt to have genericity in C.

# Simplified example of lookup function

```
typedef struct NAMED NAMED;

NAMED *lookup(struct HASH_TABLE *tbl, char *name, int size)
{
    while(/* ... */) { /* ... */ }
    /* The element was not found into the table: create it. */
    NAMED *n = malloc(size);
    memset(n, 0, size);
    n->name = name;
    store(tbl, n);
    return n;
}

void example(struct HASH_TABLE *tbl)
{
    char *name = ...;
    struct TAG *t = (struct TAG *) lookup(tbl, name, sizeof(struct TAG));
    if (!t->name) /* <- invalid use by strict aliasing analysis */
        /* [...] */*
    /* [...] */
}
```

# Proposed Fix

```
// typedef struct NAMED NAMED;
typedef void NAMED;

NAMED *lookup(struct HASH_TABLE *tbl, char *name, int size)
{
    while(/* ... */) { /* ... */ }
    /* The element was not found into the table: create it. */
    NAMED *n = malloc(size);
    memset(n, 0, size);
    // n->name = name;
    *(char **) n = name;
    store(tbl, n);
    return n;
}

void example(struct HASH_TABLE *tbl)
{
    char *name = ...;
    struct TAG *t = (struct TAG *) lookup(tbl, name, sizeof(struct TAG));
    char **tname = &t->name;
    if (!*tname) /* <- OK ! */
        /* [...] */
    /* [...] */
}
```

# Conclusion

## A problem: strict aliasing

Compilers breaking previsouly-working programs of their own volition
Sanitizers have ignored the problem until now
Continuously breaking programs since introduction
(because of interactions with other optimizations)

## A solution: a static analysis for detecting violations

Tuned to what compilers actually implement
Supports GCC workarounds