Trust Me

An exploration of `@trusted` code in D

Steven Schveighoffer - DConf Online 2020
Code: https://github.com/schveiguy/dconf2020
Memory Safety!
“Developers using C and C++ have full control over how they manage an app’s memory pointers, but these programming languages do not have the capabilities to alert developers when they’re making memory management errors.”

“[Because] Windows has been written mostly in C and C++, two "memory-unsafe" programming languages that allow developers fine-grained control of the memory addresses where their code can be executed. One slip-up in the developers' memory management code can lead to a slew of memory safety errors that attackers can exploit with dangerous and intrusive consequences --such as remote code execution or elevation of privilege flaws.” 

Memory Safety
What does “Memory Safe” mean?

Walter Bright (DConf 2017): “I believe memory safety will kill C” (Scott Meyers: “Wow.”)

• Memory safety violations consist of:
  • Accessing memory you should NOT have access to (e.g. buffer overflow)
  • Treating memory that is a scalar type as a pointer type
  • Dangling pointers
Memory Safety
D’s @safe implementation

- A @safe function:
  - Cannot do pointer math or indexing
  - Cannot access array elements out of bounds
  - Cannot use scalar data re-interpreted as a pointer (i.e. casting or unions)
  - Cannot change mutability type constructors (e.g. immutable -> mutable)
  - Cannot access __gshared data
  - Cannot take the address of a local variable
  - Cannot declare uninitialized pointers
  - Cannot call @system functions
Unsafety is sexy
Most interesting things are not @safe

• Most interesting part of computer programming: i/o!

• A hello world program cannot be fully @safe — it must use @system calls to print to the screen.

• We all want to be dangerous rebels! It’s in our code!
Trust Me
Trust Me
@trusted functions bridge the gap

• Posix write function is not @safe:

```c
extern (C) @system ssize_t write(int fd, const scope void* buf, size_t numBytes);
```

• But we can wrap it in a @safe D function:

```d
@trusted ssize_t safeWrite(int fd, const void[] buf) {
    return write(fd, buf.ptr, buf.length);
}
```

• We can use our knowledge of the POSIX API to prove that this call is safe to use from a @safe function.
The Benefit of @trusted

Limiting the review

• @trusted allows calling @system functions.

• But the true benefit is being able to limit what code needs to be checked by hand.

• If @trusted functions and APIs are written correctly, then there should be no reason to check @safe code.

• This is accomplished by manually verifying the code in @trusted functions does not violate memory safety rules.
Tagged Union
Writing a \texttt{@safe union of any two types}

- Tagged unions are a pairing of a union with a tag to identify the valid member.
- If written properly, tagged unions can be considered memory-safe
- But we want the compiler to help us!
Tagged Union
Issue 20655 (https://issues.dlang.org/show_bug.cgi?id=20655)

- Templates *should* infer `@safe` or `@system`
- [REG: 2.072] attribute inference accepts unsafe union access as `@safe`
- Explicit `@safe/@system` tags in implementation (shouldn’t be necessary)
Implementation part 1

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion1.d

```d
module taggedunion;
import std.exception;

struct Tagged(T1, T2) {
    private union Values {
        T1 t1;
        T2 t2;
    }
    private {
        Values values;
        bool tag;
        enum useT1 = false;
        enum useT2 = true;
    }
    this(T1 t1) {
        values.t1 = t1;
        tag = useT1;
    }
    this(T2 t2) {
        values.t2 = t2;
        tag = useT2;
    }
    ...
}
```
module taggedunion;
import std.exception;

struct Tagged(T1, T2) {
...

    void opAssign(T1 t1) {
        if(tag == useT2)
            destroy(values.t2);
        values.t1 = t1;
        tag = useT1;
    }

    void opAssign(T2 t2) {
        if(tag == useT1)
            destroy(values.t1);
        values.t2 = t2;
        tag = useT2;
    }

    ...
}
module taggedunion;

struct Tagged(T1, T2) {
  ...

  ~this() {
    if(tag == useT2)
      destroy(values.t2);
    else
      destroy(values.t1);
  }

  ref get(T)() if (is(T == T1) || is(T == T2)) {
    import std.exception : enforce;
    enforce((tag == useT2) == is(T == T2),
      "attempt to get wrong type from tagged union of "
      ~ T1.stringof ~ ", " ~ T2.stringof);
    static if(is(T == T2))
      return values.t2;
    else
      return values.t1;
  }
}
Implementation part 1

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion1.d

```d
module taggedunion;

struct Tagged(T1, T2) {
    ...
}

// not @safe yet
unittest {
    import std.exception : assertThrown;
    alias TU = Tagged!(int, int *);
    auto tu = TU(1);
    assert(tu.get!int == 1);
    assertThrown(tu.get!(int *));
    int **x = new int(1);
    tu = x;
    assert(tu.get!(int *) == x);
    assertThrown(tu.get!int);
}

Let's run it!

% rdmd -main -unittest taggedunion1.d
1 modules passed unittests
Is it @safe?
Safety of Tagged

is it @safe?

• Using Tagged cannot result in a memory violation.
  • No access to memory we don’t own
  • No treating scalars as pointers
  • No dangling pointers
• Very similar to memory allocation.
Compiler: “not safe!”


@safe unittest {
...
}

% rdmd -main -unittest taggedunion2.d
taggedunion2.d(59): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system destructor taggedunion.Tagged!(int, int*).Tagged.@this

taggedunion2.d(38): taggedunion.Tagged!(int, int*).Tagged.@this is declared here
taggedunion2.d(59): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system destructor taggedunion.Tagged!(int, int*).Tagged.@this

taggedunion2.d(38): taggedunion.Tagged!(int, int*).Tagged.@this is declared here
taggedunion2.d(60): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system function taggedunion.Tagged!(int, int*).Tagged.get!int.get

taggedunion2.d(45): taggedunion.Tagged!(int, int*).Tagged.get!int.get is declared here
taggedunion2.d(61): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system function taggedunion.Tagged!(int, int*).Tagged.get!int.get

taggedunion2.d(45): taggedunion.Tagged!(int, int*).Tagged.get!(int*).get is declared here
taggedunion2.d(63): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system function taggedunion.Tagged!(int, int*).Tagged.get!int.get

taggedunion2.d(45): taggedunion.Tagged!(int, int*).Tagged.get!(int*).get is declared here
taggedunion2.d(65): Error: @safe function taggedunion.__unittest_L56_C7 cannot call @system function taggedunion.Tagged!(int, int*).Tagged.get!int.get

taggedunion2.d(45): taggedunion.Tagged!(int, int*).Tagged.get!int.get is declared here

Compiler: “not safe!”


• Because every function’s safety is inferred, instead of seeing the actual part that makes the code unsafe, you see just the “cannot call @system function” error messages

• Use explicit @safe tags to further diagnose those errors.
Compiler: “not safe!”


```
struct Tagged(T1, T2) {
    @safe:
    ...
}
taggedunion3.d(20): Error: field Values.t2 cannot access pointers in @safe code that overlap other fields
taggedunion3.d(26): Error: field Values.t2 cannot access pointers in @safe code that overlap other fields
taggedunion3.d(34): Error: field Values.t2 cannot access pointers in @safe code that overlap other fields
taggedunion3.d(40): Error: field Values.t2 cannot access pointers in @safe code that overlap other fields
taggedunion3.d(58): Error: template instance taggedunion.Tagged!(int, int*) error instantiating
```
Compiler: “not safe!”


• As expected, all the problems stem from accessing the union pointer member that overlaps the non-pointer member.

• However, our tag tells us which one is valid. So we can mark @trusted the portions of the code that determine which value is valid, and provide a reference to that value.
Extract trusted portions


```d
struct Tagged(T1, T2) {
  ...
  @trusted private ref accessValue(bool expectedTag)()
  {
    import std.exception;
    enforce(tag == expectedTag, "attempt to get wrong type from tagged union of "
      ~ T1.stringof ~ ", " ~ T2.stringof);
    static if(expectedTag == useT2)
      return values.t2;
    else
      return values.t1;
  }
  @trusted private void setTag(bool newTag)
  {
    if(tag != newTag)
    {
      if(tag == useT2)
        destroy(values.t2);
      else
        destroy(values.t1);
    }
    tag = newTag;
  }
  ...
}
```
struct Tagged(T1, T2) {

    this(T1 t1) {
        setTag(useT1);
        accessValue!useT1 = t1;
    }

    this(T2 t2) {
        setTag(useT2);
        accessValue!useT2 = t2;
    }

    void opAssign(T1 t1) {
        setTag(useT1);
        accessValue!useT1 = t1;
    }

    void opAssign(T2 t2) {
        setTag(useT2);
        accessValue!useT2 = t2;
    }

    ~this() {
        setTag(!tag);
    }

    ...
}
struct Tagged(T1, T2) {

...  
	nref get(T)() if (is(T == T1) || is(T == T2)) {

		sstatic if(is(T == T2))
		return accessValue!useT2;
	else
		return accessValue!useT1;

}  
}
Is it @safe?
It’s safe! But...

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion5.d

- Running destructors might not be safe, but we have it trusted.
- Easy to fix!

```d
/* @safe inferred */
private void setTag(bool newTag)
{
    if(tag != newTag)
    {
        if(tag == useT2)
            destroy(accessValue!useT2);
        else
            destroy(accessValue!useT1);
    }
    tag = newTag;
}
```
Are we done?

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion5.d

- How do we know the code is @safe?
- Is reviewing @trusted function enough?

```d
/* @safe */
private void setTag(bool newTag)
{
    if(tag != newTag)
    {
        if(tag == useT2)
            destroy(accessValue!useT2);
        else
            destroy(accessValue!useT1);
    }
    tag = newTag;
}
```

- Must review entire module, including @safe functions!
Mitigation
Attempt 1: Tag in the union


```d
module taggedunion;

struct Tagged(T1, T2) {
    private struct T1Val {
        bool tag;
        T1 val;
    }

    private struct T2Val {
        bool tag;
        T2 val;
    }

    private union Values {
        T1Val t1;
        T2Val t2;
        bool tag;
        int *poison;
    }

    ...
}
```
Attempt 1: Tag in the union


```
struct Tagged(T1, T2) {
    ...
    @trusted private ref accessValue(bool expectedTag)() {
        import std.exception;
        enforce(values.tag == expectedTag, "attempt to get wrong type from tagged union of 
            ~ T1.stringof ~ ", " ~ T2.stringof);
        static if(expectedTag == useT2)
            return values.t2.val;
        else
            return values.t1.val;
    }

    /* @safe */
    private void setTag(bool newTag) {
        if(values.tag != newTag) {
            if(values.tag == useT2)
                destroy(accessValue!useT2);
            else
                destroy(accessValue!useT1);
        }
        values.tag = newTag; // not @safe!
    }
    ...
}
```
Attempt 1: Tag in the union


% rdmd -main -unittest taggedunion6.d
1 modules passed unittests

• Oops! should not have compiled
Attempt 1: Tag in the union **FAILED**


% rdmd -main -unittest taggedunion6.d
 1 modules passed unitests

- Oops! should not have compiled
- Compiler allows access to non-pointer union data, *even if it overlaps a pointer.*
module systemtag;

struct SystemTag {
    private bool _tag;
    @system opAssign(bool newValue) {
        _tag = newValue;
    }
    @system opAssign(SystemTag st) {
        this._tag = st._tag;
    }
    @safe tag() {
        return _tag;
    }
    alias tag this;
}
Attempt 2: Use a specialized tag

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion7

module taggedunion;
import systemtag;

struct Tagged(T1, T2) {
    private union Values {
        T1 t1;
        T2 t2;
    }
    private {
        Values values;
        SystemTag tag;
        enum useT1 = false;
        enum useT2 = true;
    }
    ...
}
Attempt 2: Use a specialized tag

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion7

```d
struct Tagged(T1, T2) {
    ...
    /* @safe */
    private void setTag(bool newTag)
    {
        if(tag != newTag)
        {
            if(tag == useT2)
                destroy(accessValue!useT2);
            else
                destroy(accessValue!useT1);
        }
        tag = newTag;
    }
    ...
}
```

% rdmd -main -unittest taggedunion7/taggedunion.d
taggedunion7/taggedunion.d(48): Error: @safe function taggedunion.Tagged!(int, int*).Tagged.setTag cannot call @system function systemtag.SystemTag.opAssign
Use a specialized tag **PASS**


```d
struct Tagged(T1, T2) {
    ...  
    /* @safe */
    private void setTag(bool newTag) {
        if(tag != newTag) {
            if(tag == useT2)
                destroy(accessValue!useT2);
            else
                destroy(accessValue!useT1);
        } () @trusted {tag = newTag;} ();

    }
    ...
}
```

% rdmd -main -unittest taggedunion7/taggedunion.d
1 modules passed unit tests
Use a specialized tag PASS-ish

code: https://github.com/schveiguy/dconf2020/blob/master/taggedunion7

```d
struct Tagged(T1, T2) {
    ...
    /* @safe */
    private void setTag(bool newTag)
    {
        if(tag != newTag)
        {
            if(tag == useT2)
                destroy(accessValue!useT2);
            else
                destroy(accessValue!useT1);
        }
        tag.tupleof[0] = newTag;
    }
    ...
}
```

% rdmd -main -unittest taggedunion7/taggedunion.d
1 modules passed unitests
Attempt 3: Travel into the future (DIP1035)

DIP: https://github.com/dlang/DIPs/blob/master/DIPs/DIP1035.md

- Be able to tag data as only accessible to @system or @trusted functions
- Use the compiler to enforce our semantics
- Eliminates back doors

```d
@system int x;

void foo() @safe {
    x = 5; // Error
}
```
```d
module taggedunion;

struct Tagged(T1, T2) {
    private union Values {
        T1 t1;
        T2 t2;
    }
    private {
        @system Values values;
        @system bool _tag;
        @trusted bool tag() { return _tag; }
        enum useT1 = false;
        enum useT2 = true;
    }

    ...  
}
```

- Probably works...
Make `@safe` no-review

- Attempt 1: tag inside union.
  - FAIL — Compiler doesn’t stop us from accessing
- Attempt 2: Specialized “system only” tag
  - PASS-ish — Add `.tupleof` as another problem to look for.
- Attempt 3: DIP1035
  - PASS — Compiler now helps us by restricting access to the tag without extra effort or wrappers.

*But are we done? Really done?…*
import taggedunion;

@safe:
void foo(ref int x, ref int* ptr) {
    import std.stdio;
    x += 4; // malicious pointer increment
    writeln(*ptr);
}

int publicVal = 1;
private int secretVal = 42;

void main() {
    auto item = Tagged!(int, int*)(5);
    void helper(ref int x) {
        item = &publicVal;
        foo(x, item.get!(int*));
    }
    helper(item.get!int);
}

% dmd lifetime.d taggedunion.d
% ./lifetime
42
Lifetime Problems

• Need to enforce when a reference becomes invalid
• Or limit utility of the union (disallow changing types mid-program)
• Or disallow reassignment to a different type while a reference is held
Ownership solution?

- Walter’s @live solution: https://dlang.org/blog/2019/07/15/ownership-and-borrowing-in-d/

- Solution isn’t viable for a tagged union, because it’s not enough to require const, we also must require the type doesn’t change.

- Possible to enhance to allow more user semantics? e.g. opBorrow
module taggedunion;

struct BorrowedRef(T) {
    this(T* val, int *cnt) {
        this.val = val;
        this.count = cnt;
        ++(*this.count);
    }

    private int *count;
    private T *val;

    @disable this(this); // disable copying
    ~this() { --(*count); }

    @property ref T _get() { return *val; }
    alias _get this;

    void opAssign(V)(auto ref V v) { *val = v; } // bug 16426
}
Solve with reference counting


```d
struct Tagged(T1, T2) {
    private union Values {
        T1 t1;
        T2 t2;
    }
    private {
        Values values;
        bool tag;
        int borrowers;
        enum useT1 = false;
        enum useT2 = true;
    }
    this(this) { borrowers = 0;}
    ...
}
```
Solve with reference counting


```d
struct Tagged(T1, T2) { ... 
   @trusted private @property accessValue(bool expectedTag)() { import std.exception; enforce(tag == expectedTag, "attempt to get wrong type from tagged union of ", ~ T1.stringof ~ "", "", ~ T2.stringof); static if(expectedTag == useT2) return BorrowedRef!T2(&values.t2, &borrowers); else return BorrowedRef!T1(&values.t1, &borrowers); } 

private void setTag(bool newTag) { if(tag != newTag) { import std.exception; enforce(borrowers == 0, "Cannot change type when someone has a reference"); if(tag == useT2) destroy(accessValue!useT2._get); else destroy(accessValue!useT1._get); } @trusted { tag = newTag; } (); } ...
```
Solve with reference counting


```d
struct Tagged(T1, T2) {
 ...
    this(T1 t1) {
        setTag(useT1);
        accessValue!useT1[] = t1;
    }
    this(T2 t2) {
        setTag(useT2);
        accessValue!useT2[] = t2;
    }
    void opAssign(T1 t1) {
        setTag(useT1);
        accessValue!useT1[] = t1;
    }
    void opAssign(T2 t2) {
        setTag(useT2);
        accessValue!useT2[] = t2;
    }
    ...
}
```
import taggedunion;

@safe:
void foo(ref int x, ref int* ptr) {
    import std.stdio;
    x += 4; // next integer
    writeln(*ptr);
}

int publicVal = 1;
private int secretVal = 42;

void main() {
    auto item = Tagged!(int, int *)(5);
    void helper(ref int x) {
        item = &publicVal;
        foo(x, item.get!(int *)._get); // bug 21369
    }

    helper(item.get!int._get); // bug 21369
}
% dmd -g lifetime2.d taggedunion9.d
% ./lifetime2
object.Exception@taggedunion9.d(64): Cannot change type when someone has a reference

-----------------
lifetime2.d:18 pure @safe void std.exception.bailOut!(Exception).bailOut(immutable(char)[], ulong, scope
const(char)[]) [0x1007ca08a]
lifetime2.d:18 pure @safe bool std.exception.enforce!().enforce!(bool).enforce(bool, lazy const(char)[]),
immutable(char)[]], ulong) [0x1007ca006]
lifetime2.d:18 pure @safe void taggedunion.Tagged!(int, int*).Tagged.setTag(bool) [0x1007ca2a5]
lifetime2.d:18 pure @safe void taggedunion.Tagged!(int, int*).Tagged.opAssign(int*) [0x1007ca49f]
lifetime2.d:16 @safe void lifetime2.main().helper(ref int) [0x1007c974f]
lifetime2.d:20 _Dmain [0x1007c963c]

Line 16:        item = &publicVal;
Conclusion

• Writing @trusted code is not as easy as it seems

• Still a long way to go to allow “no need for review” @safe code

• DIP1035 would help!

• Lifetime issues not very easy to solve, not helped by compiler bugs

• BUT there is a path to where the compiler helps enforce the semantic guarantees we want!