# Mitigating use-after-free security vulnerabilities in C and C++ with language support for type-isolating allocators

Oliver Hunt

LLVM Dev Meeting 2024

# Agenda

Use-After-Free Errors

Type Isolating Allocators

Typed Memory Operations

Type Descriptors

Type Aware Allocators in C++

# Use-After-Free: A Highly Exploitable Bug

Class of memory safety error

Difficult to statically prevent in C and C++

Highly exploitable vector for security vulnerabilities

- Provides attacker with path to type confusion
- Attacker uses this to get control of “trusted” data
- RCE, privilege elevation, further corruption, ...

# Type Isolating Allocators

Try to mitigate the exploitability of UAF errors

- Don’t reuse same memory for different types
- Don’t mix pointers and data

Effective in many security sensitive environments

- Kernels (kalloc_type in xnu)
- Browsers (Libpas, PartitionAlloc, ...)

# Problem:
# Manual adoption is expensive.

# Need to Communicate Type to Allocator

Existing allocators require manually specifying type information or use heuristics like allocator return address to determine “type” identity

Limiting factor for adopting type isolating allocators

- Existing allocation APIs are not type aware
- C based languages do not provide type information to allocators
- Almost all existing code uses untyped APIs (such as malloc)

## Solution: Typed Memory Operations in Clang

Annotation to let library author provide a type aware allocation API that is *automatically* adopted by existing users of untyped APIs

Allows the library author to specify parameter to perform type inference over

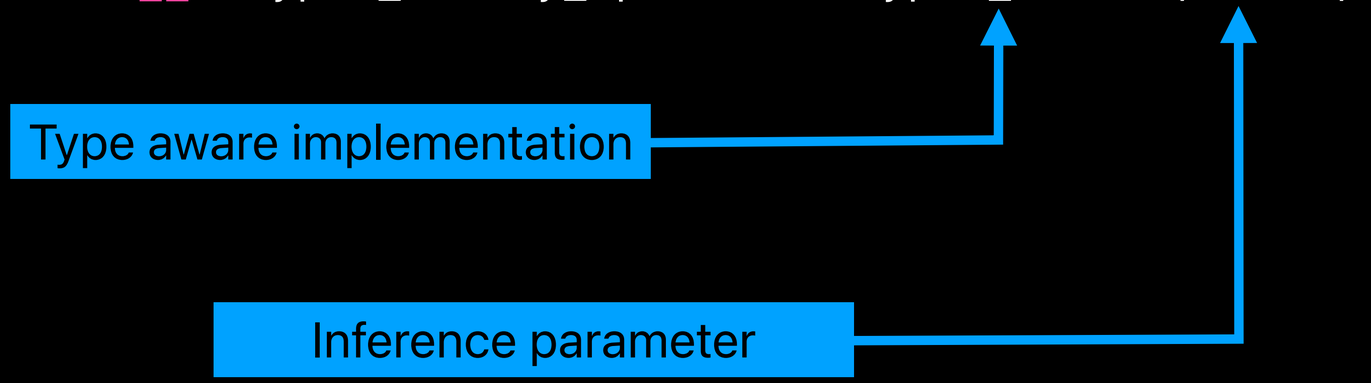
Compiler retargets calls to annotated APIs to typed variant

Clang RFC

# Typed Memory Operations

## Example

```
void *typed_malloc(size_t size, <type>);
void *malloc(size_t size)
  __attribute__((typed_memory_operation(typed_malloc, 1)));
```



```
malloc(sizeof(struct Foo));
```

Clang Codegen



```
typed_malloc(sizeof(struct Foo), <type>);
```

# Allocator Needs A Runtime Type Representation

Minimal impact to binary size

Types may not match declared types

Need to provide semantic information about the type being allocated

# A Compact Type Representation

Type descriptor

- Single 64-bit value
- Set of bits reserved for semantically important type information
- Type identity represented as a hash of structural type

No additional metadata

Only overhead is the constant parameter passed to allocator

# Providing Semantic Information

Structural and content information

- Is it polymorphic? Are there unions? Data or code pointers? ...

Call site information

- Array vs non-array, fixed vs variable size, ...

This semantic information allows an allocator to control how to apply isolation policies

# Identifying Distinct Types

Type descriptors provide type identity through a hash of their “structural type”

What is a structural type?

- Derived from work to support XNU’s kalloc_type
- Core idea is to linearize the C type into a representation of the content of each byte in that type
- Byte content is not fully typed

# Automatically Adoption

Infer a type descriptor from the original allocation call

Necessarily heuristic approach

- `sizeof(T)`
- `sizeof(T) * N`
- `sizeof(T) + sizeof(U)`
- …

Can fail

# Manual Adoption

Inference is great as a default option but does not handle all cases

Many common idioms can defeat inference

- Wrappers - can be resolved by adopting TMO on the wrapper
- Style guidelines
- …

Can construct a type descriptor manually when needed via `__builtin_tmo_get_type_descriptor(<type or expression>)`

# Deployment

Deployed as system allocator across multiple platforms

Code size and runtime performance in noise

No impact on build times

Majority of call sites successfully inferred type being allocated

# Typed Allocators in C++

Memory allocation is semantically visible in C++

Type is visible to the compiler, but not the allocator

Proposal P2719

- Type-aware allocation and deallocation functions

Passes type as a template parameter to operator

# Typed Allocators in C++

Allocated type passed as a template parameter

```
template <typename T> void *operator new(std::type_identity<T>, std::size_t);
...
template <typename T> void  operator delete(std::type_identity<T>, void *);
...
```

type_identity tag is used prevent ambiguity with existing code and to allow clean interaction with template deduction.

# Interop with Typed Memory Operations

C++ allocation operators are typically implemented in terms of C APIs

Want to be able to share the allocator implementation

Can just use the builtin:

```
template <typename T>
void *operator new(std::type_identity<T>, std::size_t size) {
  return typed_malloc(size, __builtin_tmo_get_type_descriptor(T));
}
```

# Summary

Attribute allows specification of a typed variant of standard allocator APIs

Automatically redirects untyped calls to the type variant including an inferred type

Provides information allowing allocators to efficiently enforce type and policy based isolation

- RFC has been extended to allow other schemas to be specified in future

C++ proposal provides a standards based path that works with existing custom allocators

# Questions?