Backporting Safety

Taylor Foxhall
Backporting Safety

C++Now 2024
May 1, 2024

Taylor Foxhall
Bloomberg Managed Services (BMS) - Queuing Core

TechAtBloomberg.com

© 2024 Bloomberg Finance L.P. All rights reserved.
● Distributed message queue

● Resilient to errors (“highly available”)

● Deployed in multi-hop topologies

● Published as open source in July 2023
• Built on long existing C++ libraries
• Needed to meet performance goals
• Required support for platforms without C++11 compilers
• Resilient to unexpected states
What is Safety?

- Memory safety
- Type safety
- Thread safety
- ... safety

https://www.youtube.com/watch?v=Gh79wcGJdTg
What is Safety?

- “An operation is safe if it cannot lead to undefined behavior.”

- “An unsafe operation may lead to undefined behavior if its preconditions are violated.”

Sean Parent

Keynote: The Tragedy of C++, Acts One & Two - Sean Parent - CppNorth 2022
https://www.youtube.com/watch?v=kZCUPRMH744

TechAtBloomberg.com
Why Safety?

- Unsafe code can lead to undefined behavior
- Undefined behavior can cause *incorrect behavior*
- Some of that incorrect behavior is dangerous
- Many operations in C++ can cause undefined behavior by default
- C++ is trying to evolve to make it more difficult to do by default

https://herbsutter.com/2024/03/11/safety-in-context/
Goals

● Explore causes for different types of safety bugs

● Highlight defensive design patterns techniques used to make BlazingMQ

● Show what possible defaults we can “backport” from future standards
Memory Safety
What is Memory Safety?

- Reading/writing out of bounds
- Use-after-free
- Using uninitialized data
What is Memory Safety?

- Reading/writing out of bounds
- Use-after-free
- Using uninitialized data

Spatial

Temporal

Accepting Undefined Behaviors

- Undefined behavior is fundamental to C++ Standards
- We don’t have complete solutions
- Can we embrace them?
## Patching Memory Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No out of bounds indexing</td>
<td></td>
</tr>
<tr>
<td>No use-after-free</td>
<td></td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No use-after-free</td>
<td></td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>

© 2018 Bloomberg Finance L.P. All rights reserved.

TechAtBloomberg.com

© 2024 Bloomberg Finance L.P. All rights reserved.

Bloomberg

Engineering
/// Returns the message corresponding to id in the event queue.
int EventQueue::getMessage(size_t id) {
    return d_messages[id];
}
/// Returns the message corresponding to id in the event queue.
int EventQueue::getMessage(size_t id) {
    return d_messages[id];
}
/// Returns the message corresponding to id in the event queue.
/// @pre This function is undefined unless 0 <= id < this->size()
int EventQueue::getMessage(size_t id) {
  return d_messages[id];
}
/// Returns the message corresponding to id in the event queue.
/// @pre This function is undefined unless 0 <= id < this->size()
int EventQueue::getMessage(size_t id) {
    CONTRACT_ASSERT(0 <= id
                    && id < size());
    return d_messages[id];
}
Contract Programming in Brief

- **Preconditions** ⇒ checks before a function executes
- **Postconditions** ⇒ checks after a function returns
- **Assertions** ⇒ checks for everything else
# Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>No use-after-free</td>
<td></td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
## Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No use-after-free</strong></td>
<td></td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
/// Get the latest event.
const Event& EventQueue::getEvent() {
    return d_events.front();
}
Use-After-Free

● How long does the return value of `getEvent()` live?

● Can programmers validate whether an `Event` is alive?

● What responsibility does `getEvent()` have to clients who still may store references to its return value?
/// Get the latest event.
shared_ptr<const Event>
EventQueue::getEvent() {
  // ...
}
Garbage Collection

- Obviously the wrong default
- But it does solve the problem
- For some, it is the right default
- Not for BlazingMQ
# Patching Memory Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>No out of bounds indexing</td>
<td>Contract checking</td>
</tr>
<tr>
<td><strong>No use-after-free</strong></td>
<td>Garbage collection</td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
## Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No use-after-free</strong></td>
<td>Garbage collection + ??</td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
Custom Allocators

- Obviously the wrong default
- But it does solve the problem
- For some, it is the right default
- Right for BlazingMQ
Custom Allocators

- BlazingMQ uses them for memory leak detection
- Lets us rig the allocator with contracts
- We can control behavior as a definite memory leak is detected
- Like a mini valgrind!
<Speculation>
Google Security Blog
The latest news and insights from Google on security and safety on the Internet

Use-after-freedom: MiraclePtr
September 13, 2022

Posted by Adrian Taylor, Bartek Nowierski and Kentaro Hara on behalf of the MiraclePtr team

Allocators & Contracts

- MiraclePtr combines a custom allocator with a `raw_ptr<T>` type
- Quarantines & poisons memory based on ref counts
- Types like `raw_ptr<T>` give us an opportunity to add a contract through `operator->`/`operator*`
template <typename T>
T* raw_ptr<T>::operator->() {
    CONTRACT_ASSERT(isAlive(d_ptr));
    return d_ptr;
}
template <typename T>
T* live(T* ptr) {
    CONTRACT_ASSERT(isAlive(ptr));
    return ptr;
}

int* a;
std::cout << *live(a);
Allocators & Contracts

- Dereferencing invalid pointers is undefined
- C++ implementations could choose to make invalid pointer derefs fail a contract check
- Is this a better default?
</Speculation>
<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No use-after-free</strong></td>
<td>Garbage collection</td>
</tr>
<tr>
<td></td>
<td>Allocators &amp; Contracts &amp; raw_ptr?</td>
</tr>
<tr>
<td>No accessing uninitialized memory</td>
<td></td>
</tr>
</tbody>
</table>
## Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>No use-after-free</td>
<td>Garbage collection</td>
</tr>
<tr>
<td></td>
<td>Allocators &amp; Contracts &amp; raw_ptr?</td>
</tr>
<tr>
<td><strong>No accessing uninitialized memory</strong></td>
<td></td>
</tr>
</tbody>
</table>
# Patching Memory Safety

<table>
<thead>
<tr>
<th>No out of bounds indexing</th>
<th>Contract checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>No use-after-free</td>
<td>Garbage collection</td>
</tr>
<tr>
<td></td>
<td>Allocators &amp; Contracts &amp; raw_ptr?</td>
</tr>
<tr>
<td><strong>No accessing uninitialized memory</strong></td>
<td>Constructors?</td>
</tr>
</tbody>
</table>
Erroneous behaviour for uninitialized reads

(speculative) contract violation could be erroneous

Current work on contracts comes up against the question of what should happen in case of a contract violation. The notion of erroneous behaviour might provide a useful answer.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2795r5.html
```cpp
void StorageManager::registerQueue(const bmqt::Uri& uri,
                                   int partitionId,
                                   mqbi::Domain* domain)
{
    // executed by the *CLUSTER DISPATCHER* thread

    // PRECONDITIONS
    BSLS_ASSERT_SAFE(d_dispatcher_p->inDispatcherThread(d_cluster_p));
    BSLS_ASSERT_SAFE(uri.isValid());
    BSLS_ASSERT_SAFE(0 <= partitionId &&
                     partitionId < static_cast<int>(d_fileStores.size()));
    BSLS_ASSERT_SAFE(domain);

    ...
}
```
int ClusterUtil::getNextPartitionId(const ClusterState& clusterState,
                                   const bmqt::Uri& uri)
{
    // Try to assign to the partition which has a primary and the least number
    // of queues assigned. If no partitions have a primary, then assign to the
    // partition with the least number of queues.

    int res = -1;

    ...

    // POSTCONDITIONS
    BSLS_ASSERT_SAFE(res >= 0 &&
                      res < static_cast<int>(clusterState.partitions().size()));

    return res;
}
const QueueRecordHeader* qrh = queueRecordHeader();
const char* begin = d_blockIter.block()->base() + d_blockIter.position() +
  (qrh->headerWords() * bmq::Protocol::k_WORD_SIZE);

unsigned int paddedLen = qrh->queueUriLengthWords() *
  bmq::Protocol::k_WORD_SIZE;

BSLS_ASSERT_SAFE(0 < paddedLen);

*data = begin;
*length = paddedLen - begin[paddedLen - 1];
Type Safety
Type Safety

- Prevent invalid (perhaps undefined) operations on data
- “Make it impossible/hard to do the wrong thing”

Examples
- `std::optional/std::variant`
- URIs
- Message protocols
Type Safety

- Part of the advantage of types was that some of those “wrong things” were other safety bugs!

- Many types from future standards are literally backportable:
  - e.g. optional, shared_ptr, unordered_map, array
template <class VALUE>
inline ArraySpan<VALUE>::ArraySpan(VALUE* b, VALUE* e)
: d_begin_p(b),
  d_end_p(e)
{
    BSLS_ASSERT_SAFE(d_begin_p <= d_end_p);
}
template <class VALUE>
inline VALUE& ArraySpan<VALUE>::operator[](size_t index)
{
    BSLS_ASSERT_SAFE(d_begin_p < d_end_p);
    BSLS_ASSERT_SAFE(index < size());
    return d_begin_p[index];
}
Thread Safety
Data Races

- Data races are undefined behavior by the standard

- A data race is:
  - Two threads use data at the same time
  - At least one of them is modifying the data

- Can cause objects to be invalidated

- That might lead to other downstream safety bugs!
Ways to Fix Data Races

● Make everything const

● Synchronize access to data with atomics & mutexes

● Prevent sharing data
Actor Model

- No sharing memory
- Computations are isolated into individual threads of execution
- Actors can pass messages to each other
Actor Model in BlazingMQ

- **Client**
  - Reading/writing to client
  - Stats and message validation

- **Queue**
  - Storage and replication
  - Data routing

- **Cluster**
  - Reading/writing to cluster nodes
  - Cluster health
  - Primary node
Actor Model in BlazingMQ
Event Dispatcher
```cpp
void StorageManager::registerQueue(const bmqt::Uri& uri,
        int partitionId,
        mqbi::Domain* domain)
{
    // executed by the *CLUSTER DISPATCHER* thread

    // PRECONDITIONS
    BSLS_ASSERT_SAFE(d_dispatcher_p->inDispatcherThread(d_cluster_p));
    BSLS_ASSERT_SAFE(uri.isValid());
    BSLS_ASSERT_SAFE(0 <= partitionId &&
                        partitionId < static_cast<int>(d_fileStores.size()));
    BSLS_ASSERT_SAFE(domain);

    ...
}
```
Conclusion
What Did We Learn?

- When talking about safety in C++, qualify definitions
- Types are more easily backported than language features
- Maybe we need some design patterns for safety
- Undefined behavior has utility
Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers
Postscript: Correctness
Testing

- Unit Tests
- Integration Tests
- Sanitizers
  - ASAN, TSAN, MSAN, UBSAN
- System Correctness Verification
## System Testing with Jepsen

- **Nemesis**: network partitions, start/stop node, clock skews, etc.

<table>
<thead>
<tr>
<th></th>
<th>Eventual Consistency</th>
<th>Strong Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BlazingMQ</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partition-random-node</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>partition-random-halves</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>partition-majorities-ring</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
TLA+

- Formal specification language
- BlazingMQ's leader election and state machine replication is strongly inspired by the Raft consensus algorithm
- TLA+ verifies the correctness of the implementation
EXTENDS Naturals, FiniteSets, Sequences, Reals, TLC

/* Input parameters
CONSTANTS   Server, /** The servers involved. E.g. {S1, S2, S3}
            MaxRestarts, /** Maximum number of times a server should restart
            MaxScouting, /** Maximum number of times a server should send
            MaxUnavailable /** Maximum number of times a server should be

/* Model values
CONSTANTS Follower, Candidate, Leader
CONSTANTS Nil

CONSTANTS ElectionProposal, ElectionResponse,
        LeaderHeartbeat, HeartbeatResponse,
        ScoutingRequest, ScoutingResponse,
        LeadershipCession, NodeUnavailable
InitServerVars == \ currentTerm = [i \in Server |\rightarrow 0] \\
  state = [i \in Server |\rightarrow Follower] \\
  leaderId = [i \in Server |\rightarrow Nil] \\
  tentativeLeaderId = [i \in Server |\rightarrow Nil] \\
  supporters = [i \in Server |\rightarrow {}] \\
  scoutingInfo = [i \in Server |\rightarrow [\text{term |\rightarrow Nil, responses |\rightarrow {}}]]

InitAuxVars == \ restartCounter = [i \in Server |\rightarrow 0] \\
  scoutingCounter = [i \in Server |\rightarrow 0] \\
  unavailableCounter = [i \in Server |\rightarrow 0]

Init == \ messages = [m \in {} |\rightarrow 0] \\
  \ InitServerVars \\
  \ InitAuxVars

\* Server i restarts from stable storage. \\
\* It resets every server variable but its currentTerm

Restart(i) == \\
  \ restartCounter[i] < MaxRestarts \\
  \ state' = [\text{state EXCEPT ![i] = Follower}] \\
  \ leaderId' = [\text{leaderId EXCEPT ![i] = Nil}] \\
  \ tentativeLeaderId' = [\text{tentativeLeaderId EXCEPT ![i] = Nil}] \\
  \ supporters' = [\text{supporters EXCEPT ![i] = {}}] \\
  \ scoutingInfo' = [\text{scoutingInfo EXCEPT ![i] = ResetScoutingInfo}] \\
  \ restartCounter' = [\text{restartCounter EXCEPT ![i] = @ + 1}] \\
  \ UNCHANGED \langle \langle \text{messages, currentTerm, scoutingCounter, unavailableCounter} \rangle \rangle
References

- BlazingMQ: [https://github.com/bloomberg/blazingmq](https://github.com/bloomberg/blazingmq)
- BlazingMQ landing page: [https://bloomberg.github.io/blazingmq/](https://bloomberg.github.io/blazingmq/)

- BDE: [https://github.com/bloomberg/bde](https://github.com/bloomberg/bde)
- Jepsen: [https://jepsen.io/](https://jepsen.io/)
- TLA+: [https://lamport.azurewebsites.net/tla/tla.html](https://lamport.azurewebsites.net/tla/tla.html)

References

- The Case for Memory Safe Roadmaps: https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf
References

  https://www.youtube.com/watch?v=Gh79wcGJdTg

● All the Safeties: Safety in C++ - Sean Parent - CppNow 2023
  https://www.youtube.com/watch?v=MO-qehjc04s

● Delivering Safe C++ - Bjarne Stroustrup - CppCon 2023
  https://www.youtube.com/watch?v=I8UvQKvOSSw