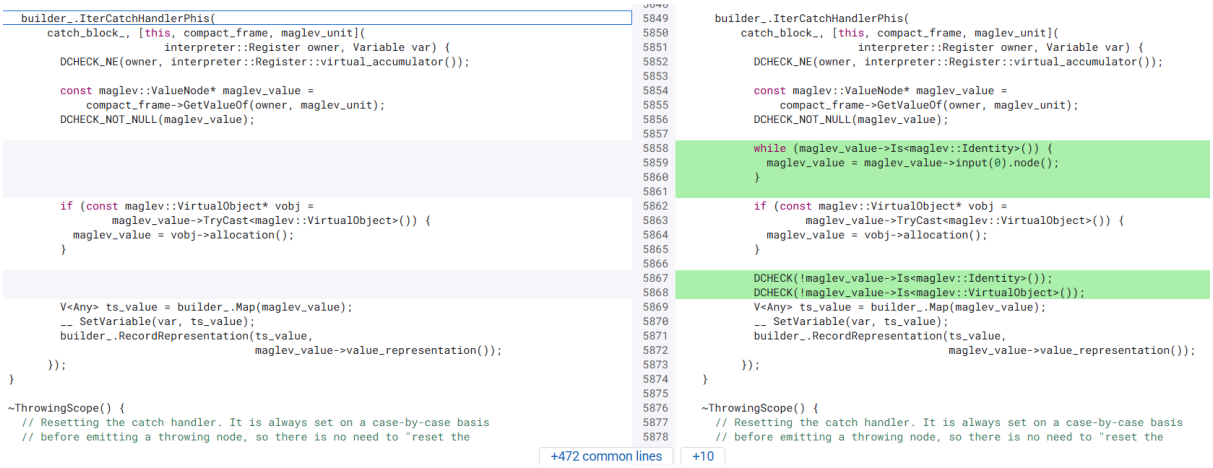
5주차 과제

결과



```
builder_.IterCatchHandlerPhis(
    catch_block_, [this, compact_frame, maglev_unit](
                      interpreter::Register owner, Variable var) {
      DCHECK_NE(owner, interpreter::Register::virtual_accumulator());

      const maglev::ValueNode* maglev_value =
          compact_frame->GetValueOf(owner, maglev_unit);
      DCHECK_NOT_NULL(maglev_value);

      if (const maglev::VirtualObject* vobj =
            maglev_value->TryCast<maglev::VirtualObject>()) {
        maglev_value = vobj->allocation();
      }

      V<Any> ts_value = builder_.Map(maglev_value);
      __ SetVariable(var, ts_value);
      builder_.RecordRepresentation(ts_value,
                                    maglev_value->value_representation());
    });
}

~ThrowingScope() {
  // Resetting the catch handler. It is always set on a case-by-case basis
  // before emitting a throwing node, so there is no need to "reset the
```

```
builder_.IterCatchHandlerPhis(
    catch_block_, [this, compact_frame, maglev_unit](
                      interpreter::Register owner, Variable var) {
      DCHECK_NE(owner, interpreter::Register::virtual_accumulator());

      const maglev::ValueNode* maglev_value =
          compact_frame->GetValueOf(owner, maglev_unit);
      DCHECK_NOT_NULL(maglev_value);

      while (maglev_value->Is<maglev::Identity>()) {
        maglev_value = maglev_value->input(0).node();
      }

      if (const maglev::VirtualObject* vobj =
            maglev_value->TryCast<maglev::VirtualObject>()) {
        maglev_value = vobj->allocation();
      }

      DCHECK(!maglev_value->Is<maglev::Identity>());
      DCHECK(!maglev_value->Is<maglev::VirtualObject>());
      V<Any> ts_value = builder_.Map(maglev_value);
      __ SetVariable(var, ts_value);
      builder_.RecordRepresentation(ts_value,
                                    maglev_value->value_representation());
    });
}

~ThrowingScope() {
  // Resetting the catch handler. It is always set on a case-by-case basis
  // before emitting a throwing node, so there is no need to "reset the
```

+472 common lines +10

본 취약점은 turbolev-graph-builder.cc 에서 발생된다. 해당 취약점을 이해하기 위해 **Root Cause** 코드 분석을 먼저 진행하고자 했으나 부족한 코드 이해 실력으로 `IterCatchHandlerPhis` 함수 원형이 **Maglev**의 **Phi** 노드를 가져오는 분기까지 이해하였다.

분석

```
builder_.IterCatchHandlerPhis(
    catch_block_, [this, compact_frame, maglev_unit](
                      interpreter::Register owner, Variable var) {
      DCHECK_NE(owner, interpreter::Register::virtual_accumulator());

      const maglev::ValueNode* maglev_value =
          compact_frame->GetValueOf(owner, maglev_unit);
      DCHECK_NOT_NULL(maglev_value);

      while (maglev_value->Is<maglev::Identity>()) {
        maglev_value = maglev_value->input(0).node();
```

```
    }

    if (const maglev::VirtualObject* vobj =
          maglev_value->TryCast<maglev::VirtualObject>()) {
      maglev_value = vobj->allocation();
    }

    DCHECK(!maglev_value->Is<maglev::Identity>());
    DCHECK(!maglev_value->Is<maglev::VirtualObject>());
    V<Any> ts_value = builder_.Map(maglev_value);
    __ SetVariable(var, ts_value);
    builder_.RecordRepresentation(ts_value,
                                  maglev_value->value_representation());
  });
```

위 코드는 `builder_` 클래스의 `IterCatchHandlerPhis()` 메소드며, **turboshaft**의 `turbolev-graph-builder.cc`에 구현되어 있다. 각 인자로 `catch_block_`과 람다 함수가 전달된다.

```
void IterCatchHandlerPhis(const maglev::BasicBlock* catch_block,
                          Function&& callback) {
  DCHECK_NOT_NULL(catch_block);
  DCHECK(catch_block->has_phi());
  for (auto phi : *catch_block->phis()) {
    DCHECK(phi->is_exception_phi());
    interpreter::Register owner = phi->owner();
    if (owner == interpreter::Register::virtual_accumulator()) {
      // The accumulator exception phi corresponds to the exception object
      // rather than whatever value the accumulator contained before the
      // throwing operation. We don't need to iterate here, since there is
      // special handling when processing Phis to use `catch_block_begin_`
      // for it instead of a Variable.
      continue;
    }

    auto it = regs_to_vars_.find(owner.index());
    Variable var;
```

```
    if (it == regs_to_vars_.end()) {
      // We use a LoopInvariantVariable: if loop phis were needed, then the
      // Maglev value would already be a loop Phi, and we wouldn't need
      // Turboshaft to automatically insert a loop phi.
      var = __ NewLoopInvariantVariable(RegisterRepresentation::Tagged());
      regs_to_vars_.insert({owner.index(), var});
    } else {
      var = it->second;
    }

    callback(owner, var);
  }
}
```

위는 `IterCatchHanderPhis`의 원형이며, 한 줄 한 줄 이해해보기로 한다. 우선 상수 포인터로 정의된 `catch_block`의 의미를 따라가 본다.

catch_block은 `maglev::BasicBlock` 객체를 가리키는 상수 포인터다. 앞서 호출한 코드를 보면 `this->catch_block_`이 되므로 `IterCatchHandlerPhis`의 `catch_block_`이 된다.

```
const maglev::ExceptionHandlerInfo* handler_info =
throwing_node->exception_handler_info();

...

catch_block_ = handler_info->catch_block();
```

`catch_block_`은 무엇인가? `handler_info`는 `ExceptionhandlerInfo` 클래스 포인터다. 결과적으로 `handler_info->catch_block()` 로 `handler_info`는 `ExceptionhandlerInfo`에 담긴 `catch_block()` 메소드의 내용을 `catch_block_`으로 넘기게 된다.

```
class ExceptionHandlerInfo {
...

BasicBlock* catch_block() const { return catch_block_.block_ptr(); }
}
```

결국 `ExceptionhandlerInfo` 의 `catch_block()` 포인터는 `BasicBlock*` 타입으로 리턴 되므로 `ExceptionhandlerInfo` 의 BasicBlock 객체가 `IterCatchHandlerPhis` 의 catch_block로 넘어가게 된다.

- (BasicBlock* catch_block() const { return catch_block_.block_ptr(); } } 이 코드 부분 설명 추가 요망
- ExceptionhandlerInfo의 BasicBlock 객체 분석 추가 요망

```
for (auto phi : *catch_block->phis()) {
    DCHECK(phi->is_exception_phi());
    interpreter::Register owner = phi->owner();
    if (owner == interpreter::Register::virtual_accumulator()) {
      // The accumulator exception phi corresponds to the exception object
      // rather than whatever value the accumulator contained before the
      // throwing operation. We don't need to iterate here, since there is
      // special handling when processing Phis to use `catch_block_begin_`
      // for it instead of a Variable.
      continue;
    }
```

`catch_block` 포인터의 `phis()` 메소드의 값을 가져와 `phi`에 담는다. 이 때 phi는 auto로 정의되어 있어, 컴파일러가 타입을 추론하게 된다.

`for (auto x : RANGE) { ... }` 의 형태에서 RANGE는 iterable한 타입이 되어야 하므로 `phis()`는 컨테이너 타입으로 return되는 요소들을 `phi`에 담게 된다.

```
Phi::List* phis() const {
  DCHECK(has_phi());
  return state_->phis();
}
```

maglev-basic-block.h

위에서 분석했듯 이 때 `catch_block` 포인터는 `BasicBlock` 객체이므로, `BasicBlock`의 `phis()`를 가져오는데 위 코드에서 보듯 이는 `Phi` 네임스페이스의 `List` 포인터다.

```
//v8/src/maglev/maglev-ir.h

using Base = ValueNodeT<Phi>;

  public:
   using List = base::ThreadedList<Phi>;

   // TODO(jgruber): More intuitive constructors, if possible.
   Phi(uint64_t bitfield, MergePointInterpreterFrameState* merge_state,
       interpreter::Register owner)
       : Base(bitfield),
         owner_(owner),
         merge_state_(merge_state),
         type_(NodeType::kUnknown),
         post_loop_type_(NodeType::kUnknown) {
     DCHECK_NOT_NULL(merge_state);
   }
```

`phis()` 는 `Phi::List*` 타입 반환한다. List는 `base::ThreadedList<phi>`의 별칭이다. (참고로 이 때의 phis()는 `maglev-basic-block.h`에 정의되어 있다.)

```
//maglev-basic-block.h

MergePointInterpreterFrameState* state_;
```

```
//v8/src/maglev/maglev-interpreter-frame-state.h

public:
   Phi::List* phis() { return &phis_; }

private:
  // ...여러 private 멤버 선언...
  Phi::List phis_;
```

phis() 메소드가 반환하는 **state_ → phis()** 는 `MergePointInterpreterFrameState*` 타입이다. `MergePointInterpreterFrameState`에는 phis()가 위와 같이 정의되어 있고, 이는 다시 `MergePointInterpreterFrameState` 객체의 phis_의 주소를 List 타입으로 가져온다.

다시 돌아와서 결과적으로
`for (auto phi : *catch_block→phis())` 은 **&phis_** 반환 된 주소(컨테이너 타입?)의 데이터를 **phi** 변수에 담는다.

이쯤에서 phi란 무엇일지 궁금해진다. 간단하게 이해해보자면 **SSA 컴파일러**의 최적화 과정에서 사용되는 함수이다. Maglev는 Ignition(Bytecode 생성기)-Sparkplug(Bytecode 실행기)와 **Turbofan** 사이에서 IR 언어로 **Bytecode**를 변환하는 **JIT** 컴파일러다.

이러한 JIT 컴파일러는 **SSA 컴파일** 방식을 사용하는데, **SSA 컴파일**은 변수 재사용(정확히는 IValue)을 하지않는다. 이를테면 두 개의 다른 조건문 `if(a < 5){return a;}` 와 `if(a > 5){return a;}` 가 있다면, 이 a는 조건문 안에서 각 a1, a2로 재정의 되고 파이 함수를 이용하여 phi(a1, a2)로 두 개의 경로를 나눈다.

지금 보고있는 **Phi** 함수 코드는 전부 **SSA**의 **Phi**와 관련이 있음을 유추할 수 있다.

```
interpreter::Register owner = phi->owner();
    if (owner == interpreter::Register::virtual_accumulator()) {
      // The accumulator exception phi corresponds to the exception object
      // rather than whatever value the accumulator contained before the
      // throwing operation. We don't need to iterate here, since there is
      // special handling when processing Phis to use `catch_block_begin_`
      // for it instead of a Variable.
      continue;
    }
```

위 코드는 interpreter를 사용하여 phi의 owner() 메소드를 누산기 값과 비교하는 조건문이다.

- `owner()` 메소드의 정확한 역할을 아직 이해하지 못하였음.

```
auto it = regs_to_vars_.find(owner.index());
 Variable var;
 if (it == regs_to_vars_.end()) {
   // We use a LoopInvariantVariable: if loop phis were needed, then the
   // Maglev value would already be a loop Phi, and we wouldn't need
   // Turboshaft to automatically insert a loop phi.
   var = __ NewLoopInvariantVariable(RegisterRepresentation::Tagged());
   regs_to_vars_.insert({owner.index(), var});
 } else {
   var = it->second;
 }
```

이 코드에서 `regs_to_vars_` 는 `ZoneUnorderedMap<int, Variable> regs_to_vars_;` 으로 선언되어 있다.

`owner.inder()` 에서 `ZoneUnorderedMap` 의 Variable 객체가 존재하는지 찾고 만약 존재한다면 it 가 이를 가리키게 된다.

`fine(owner.index())` 에서 `ZoneUnorderedMap` 의 Variable 객체가 없다면 it는 `reg_tovars_.end()` 즉, `ZoneUnorderedMap`의 끝을 가리키게 된다.

- `ZoneUnorderedMap` 의 정확한 이해가 필요