# Security Policy

Presented and Created by Michael Hanlon

## Overview

Increase security across multiple vectors of attack

Prevent threats before they occur, as well as how to respond when they do

This is important to take into account for as it increases defense in depth across the entire system which in turn develops a robust and workable system

# Overview - Threat Matrix

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
|---|---|---|---|---|---|
| **Data Type** | Low | Somewhat Likely | Low | Medium | 1 |
| **Data Value** | High | Unlikely | Medium-High | High | 2 |
| **String Validation** | Medium-High | Somewhat Unlikely | Low-Medium | High | 2 |
| **SQL Injection** | High | Likely | High | High | 5 |
| **Memory** | High | Unlikely | Medium-High | High | 3 |
| **Assertions** | Medium | Likely | Low | Low | 2 |
| **Exceptions** | Medium | Likely | Low | High | 1 |
| **Encryption** | High | Unlikely | High | High | 5 |
| **No Except** | Low | Likely | Low | Medium | 1 |
| **Authentication** | High | Unlikely | Medium-High | High | 4 |

## Principles

1. **Keep it Simple**
    a. SQL Injection, NoExcept, Authentication
2. **Architect and Design for Security Policies**
    a. Data Value, Memory Protection, Exceptions, Encryption, NoExcept
3. **Adhere to Principle of Least Privilege**
    a. Data Types, SQL Injection
4. **Practice Defense in Depth**
    a. SQL Injection,  Assertions, Exceptions, Encryption, NoExcept, Authentication
5. **Default Deny**
    a. Encryption, Authentication

1. **Sanitize Data sent to Other Systems**
    a. String Validation, Assertions
2. **Validate Data Input**
    a. Data Types, Data Value, String Validation, Memory Protection, Assertions
3. **Use Effective Quality Assurance Techniques**
    a. Memory Protection
4. **Provide Only One Way to Do an Operation**
    a. SQL Injection, Encryption
5. **Heed Compiler Warnings**
    a. Exceptions

## Encryption Policy

In Use: This describes encryption on data or information that is currently in use

At Flight: This describes the encryption policy when the data is being transferred or moving from one part of the system to another

At Rest: Describes how data is protected while it is in storage

## Triple-A Framework

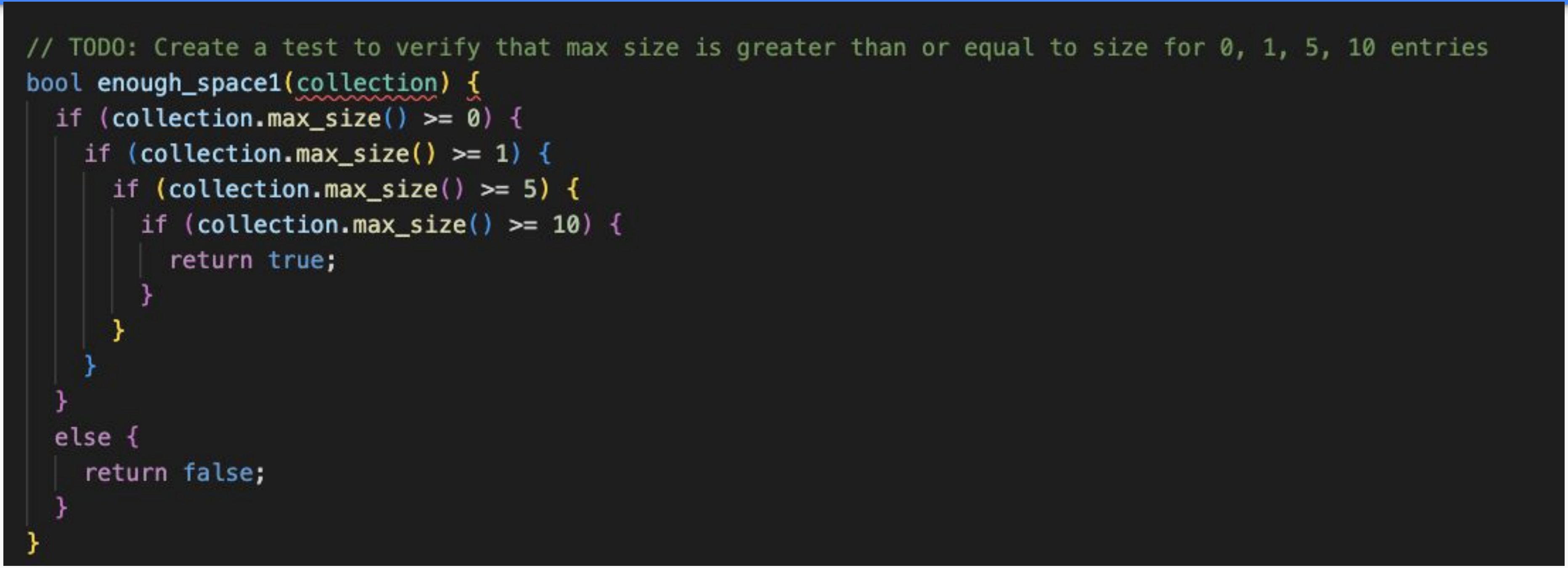
**Authentication -** How users are verified and what is tracked

**Authorization -** Designates where users are allowed to go and which functions are performed

**Accounting -** Gives information on what resources users are using

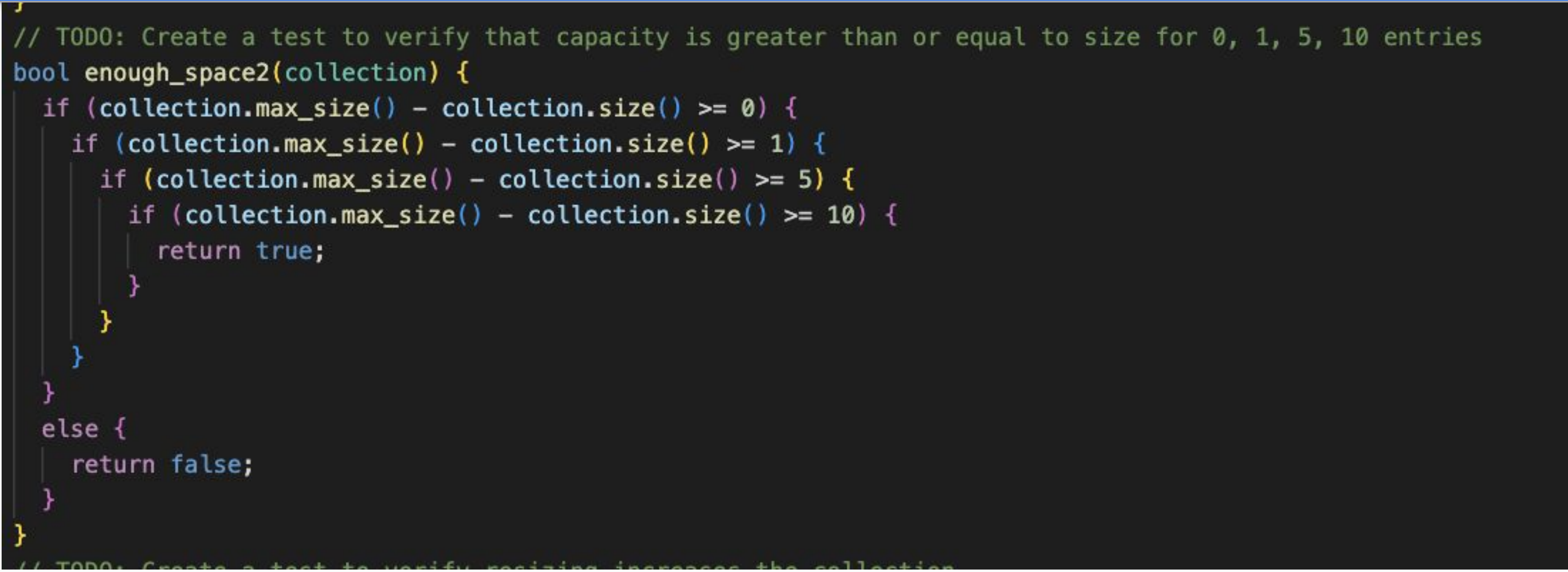
## Unit Testing

# Enough Space?



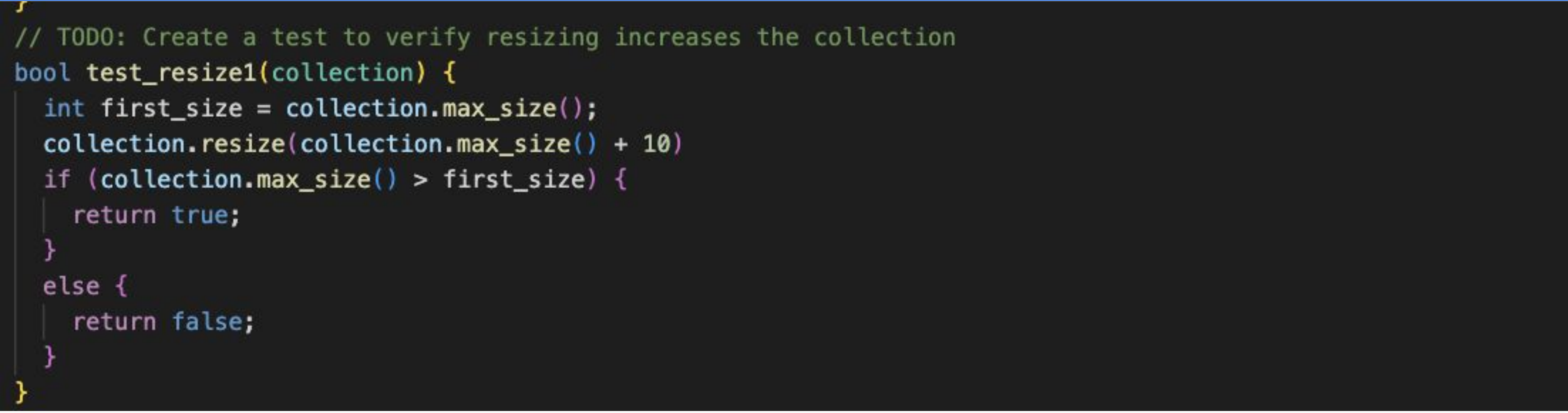
```
// TODO: Create a test to verify that max size is greater than or equal to size for 0, 1, 5, 10 entries
bool enough_space1(collection) {
  if (collection.max_size() >= 0) {
    if (collection.max_size() >= 1) {
      if (collection.max_size() >= 5) {
        if (collection.max_size() >= 10) {
          return true;
        }
      }
    }
  }
  else {
    return false;
  }
}
```

## Enough Capacity?



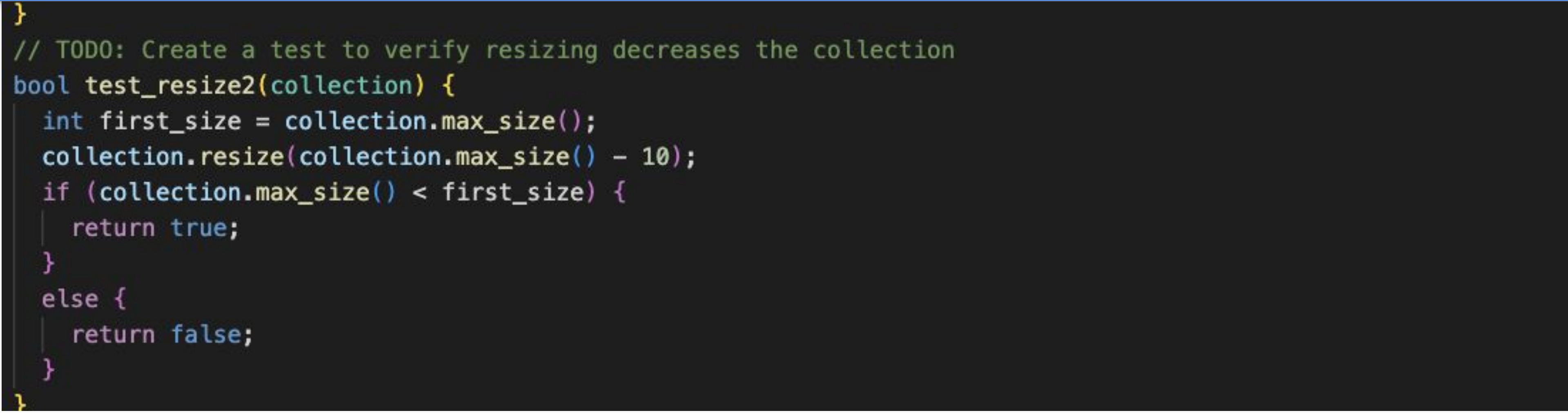
```
// TODO: Create a test to verify that capacity is greater than or equal to size for 0, 1, 5, 10 entries
bool enough_space2(collection) {
  if (collection.max_size() - collection.size() >= 0) {
    if (collection.max_size() - collection.size() >= 1) {
      if (collection.max_size() - collection.size() >= 5) {
        if (collection.max_size() - collection.size() >= 10) {
          return true;
        }
      }
    }
  }
  else {
    return false;
  }
}
```

## Test Resize



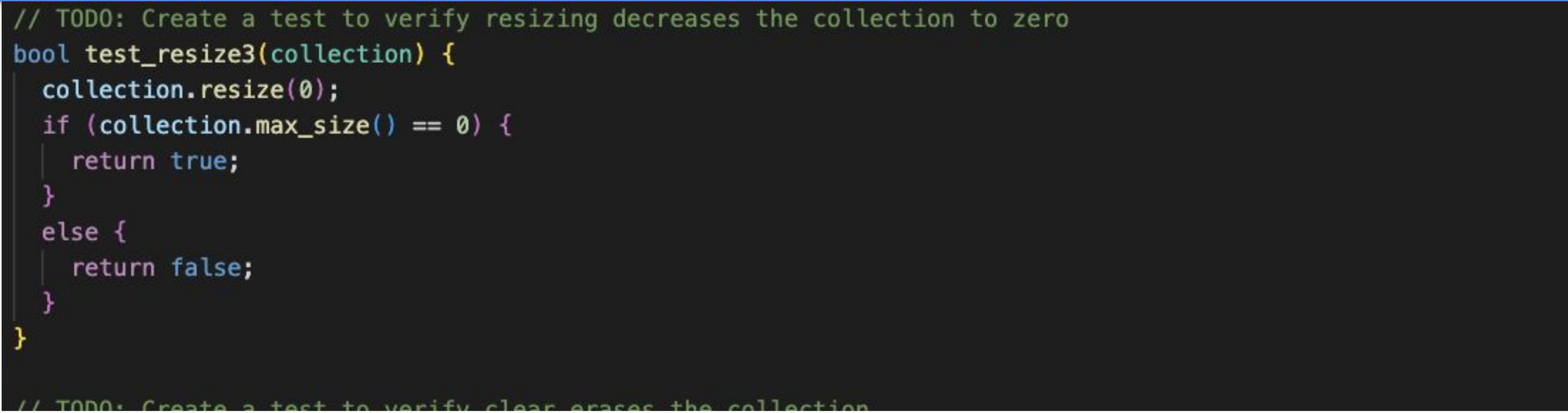
```
// TODO: Create a test to verify resizing increases the collection
bool test_resize1(collection) {
  int first_size = collection.max_size();
  collection.resize(collection.max_size() + 10)
  if (collection.max_size() > first_size) {
    return true;
  }
  else {
    return false;
  }
}
```

## Test Downsize



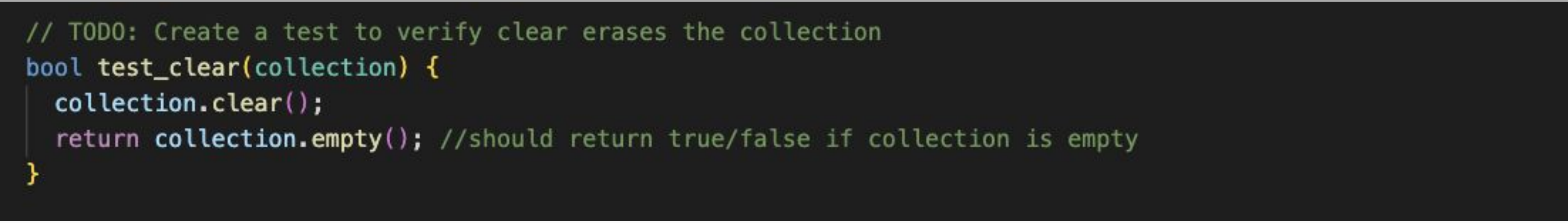
```
}
// TODO: Create a test to verify resizing decreases the collection
bool test_resize2(collection) {
  int first_size = collection.max_size();
  collection.resize(collection.max_size() - 10);
  if (collection.max_size() < first_size) {
    return true;
  }
  else {
    return false;
  }
}
```

## Test Set Size to Zero



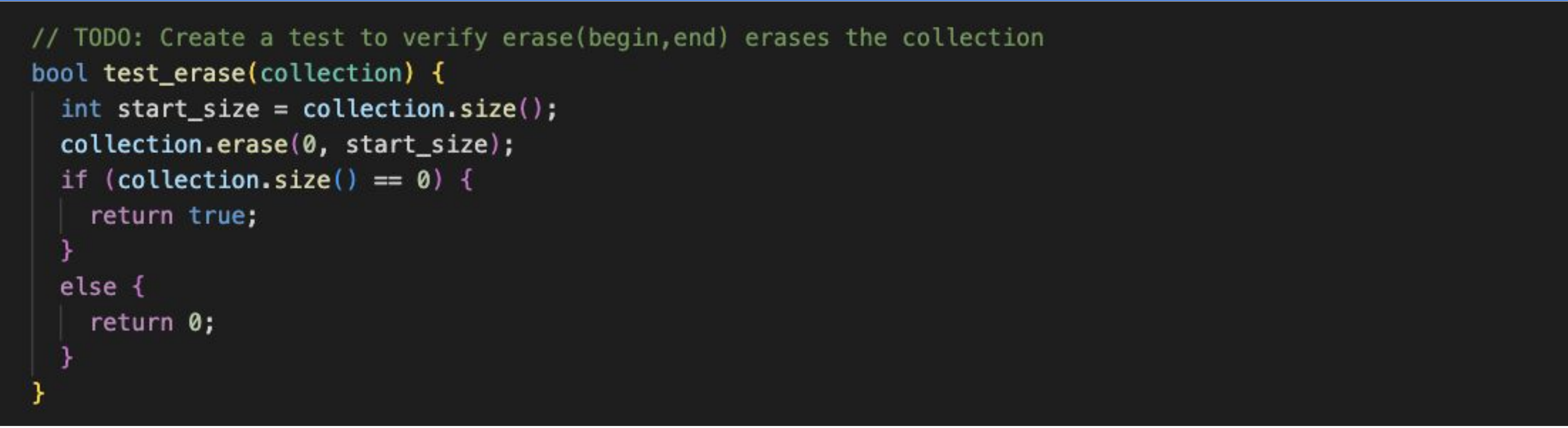
```
// TODO: Create a test to verify resizing decreases the collection to zero
bool test_resize3(collection) {
  collection.resize(0);
  if (collection.max_size() == 0) {
    return true;
  }
  else {
    return false;
  }
}

// TODO: Create a test to verify clear erases the collection
```

## Test Clear



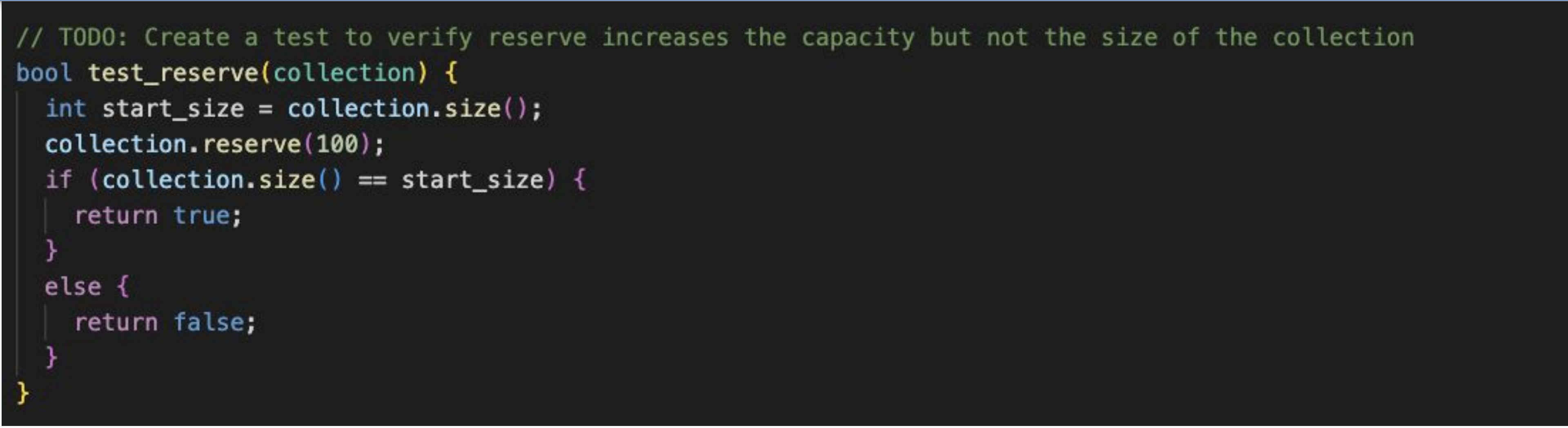
```
// TODO: Create a test to verify clear erases the collection
bool test_clear(collection) {
  collection.clear();
  return collection.empty(); //should return true/false if collection is empty
}
```

## Test Erase



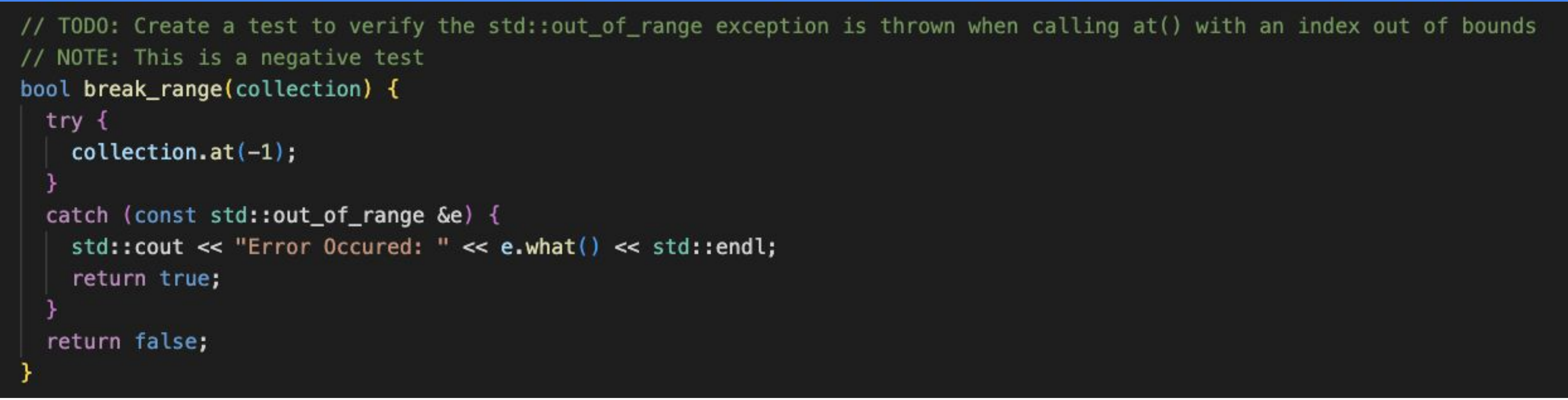
```
// TODO: Create a test to verify erase(begin,end) erases the collection
bool test_erase(collection) {
  int start_size = collection.size();
  collection.erase(0, start_size);
  if (collection.size() == 0) {
    return true;
  }
  else {
    return 0;
  }
}
```

## Test Reserve



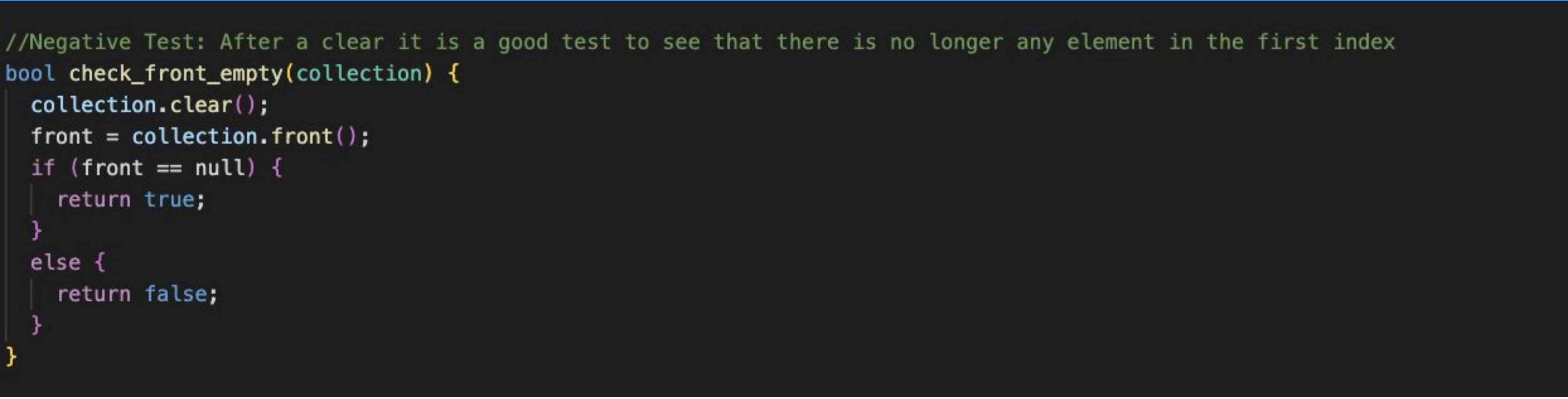
```
// TODO: Create a test to verify reserve increases the capacity but not the size of the collection
bool test_reserve(collection) {
  int start_size = collection.size();
  collection.reserve(100);
  if (collection.size() == start_size) {
    return true;
  }
  else {
    return false;
  }
}
```

## Break Range



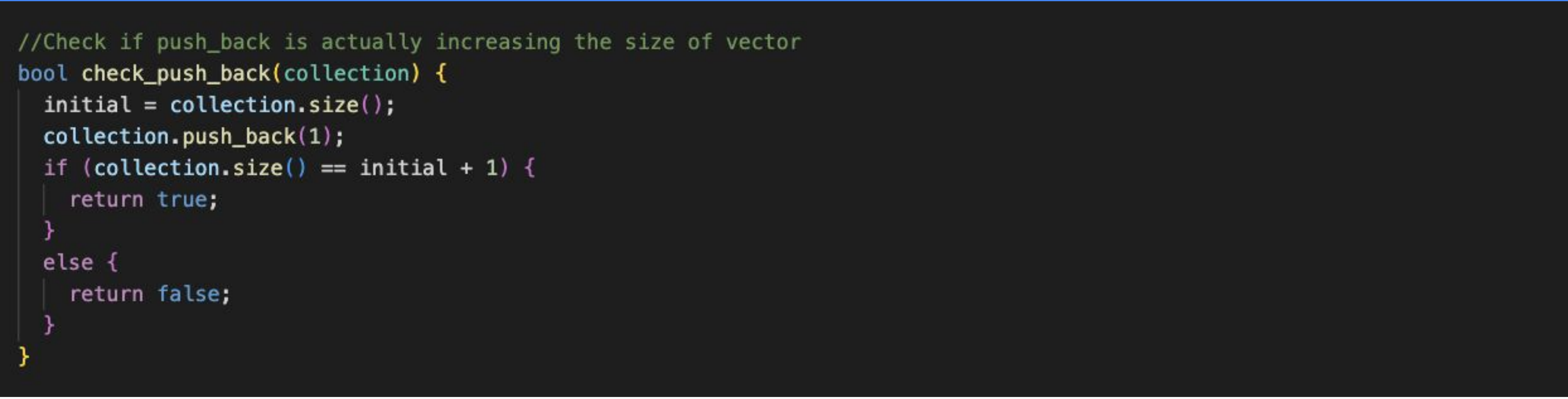
```cpp
// TODO: Create a test to verify the std::out_of_range exception is thrown when calling at() with an index out of bounds
// NOTE: This is a negative test
bool break_range(collection) {
  try {
    collection.at(-1);
  }
  catch (const std::out_of_range &e) {
    std::cout << "Error Occured: " << e.what() << std::endl;
    return true;
  }
  return false;
}
```

## Check Front Empty



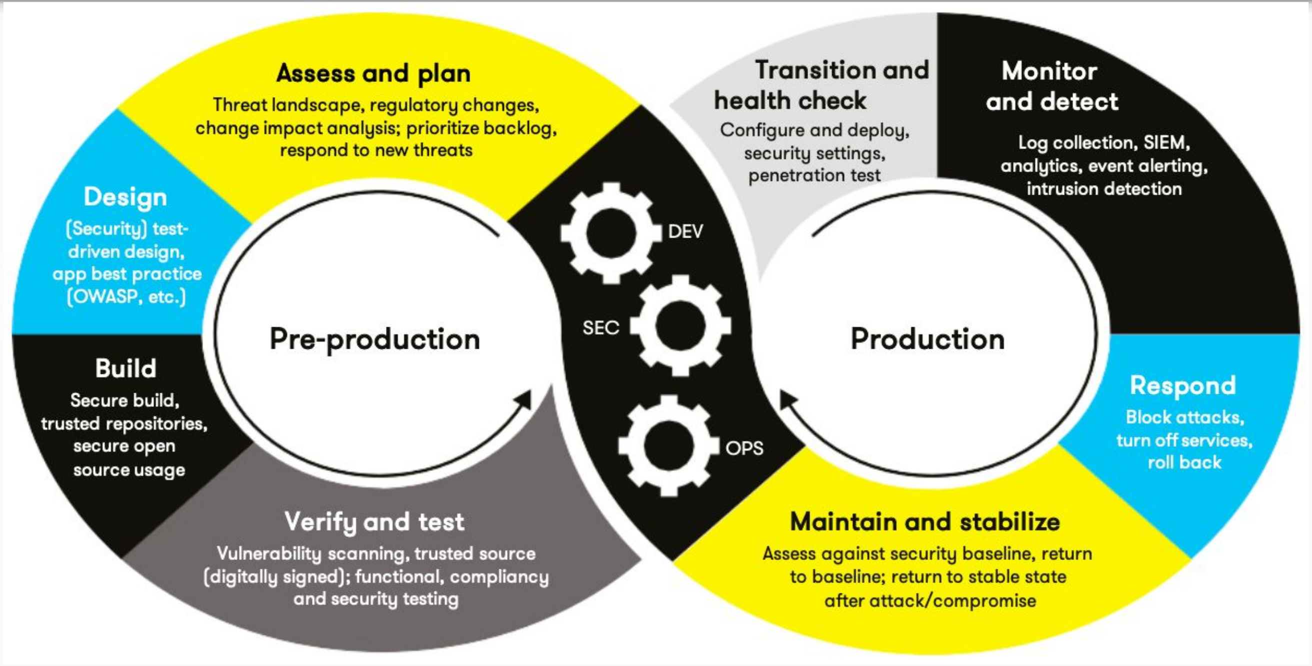
```
//Negative Test: After a clear it is a good test to see that there is no longer any element in the first index
bool check_front_empty(collection) {
  collection.clear();
  front = collection.front();
  if (front == null) {
    return true;
  }
  else {
    return false;
  }
}
```

## Check Push Back



```
//Check if push_back is actually increasing the size of vector
bool check_push_back(collection) {
  initial = collection.size();
  collection.push_back(1);
  if (collection.size() == initial + 1) {
    return true;
  }
  else {
    return false;
  }
}
```

## Automation Plan


Assess and plan
Threat landscape, regulatory changes, change impact analysis; prioritize backlog, respond to new threats
Design
(Security) test-driven design, app best practice (OWASP, etc.)
Build
Secure build, trusted repositories, secure open source usage
Verify and test
Vulnerability scanning, trusted source (digitally signed); functional, compliancy and security testing
Pre-production
DEV
SEC
OPS
Transition and health check
Configure and deploy, security settings, penetration test
Monitor and detect
Log collection, SIEM, analytics, event alerting, intrusion detection
Production
Respond
Block attacks, turn off services, roll back
Maintain and stabilize
Assess against security baseline, return to baseline; return to stable state after attack/compromise

## Risks and Benefits

Risks and Benefits:

Funding, Time, Efficiency, Increased Security, Better Image,

Attacks,  Robustness, Maintainability

## Recommendations

Continued Security Updates

Constant Research into New Vectors of Attack

Time Based Encryption Breaking

https://www.cnet.com/personal-finance/crypto/record-set-in-cracking-56-bit-crypto/

## Conclusions

Security Policy Should be Stringently Enforced and Implemented From the Beginning

Continuous Updating and Research Should Always be in Effect

Should Breaches Occur Notify Stakeholders

## Citations

CNet.com Staff. (n.d.). *Record set in cracking 56-bit crypto*. CNET. https://www.cnet.com/personal-finance/crypto/record-set-in-cracking-56-bit-crypto/