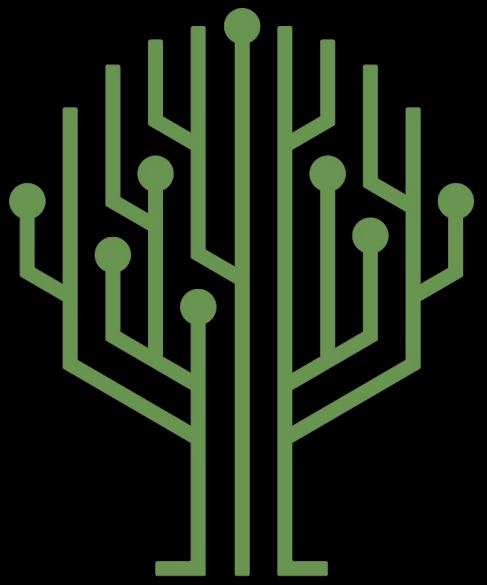
Green Pace

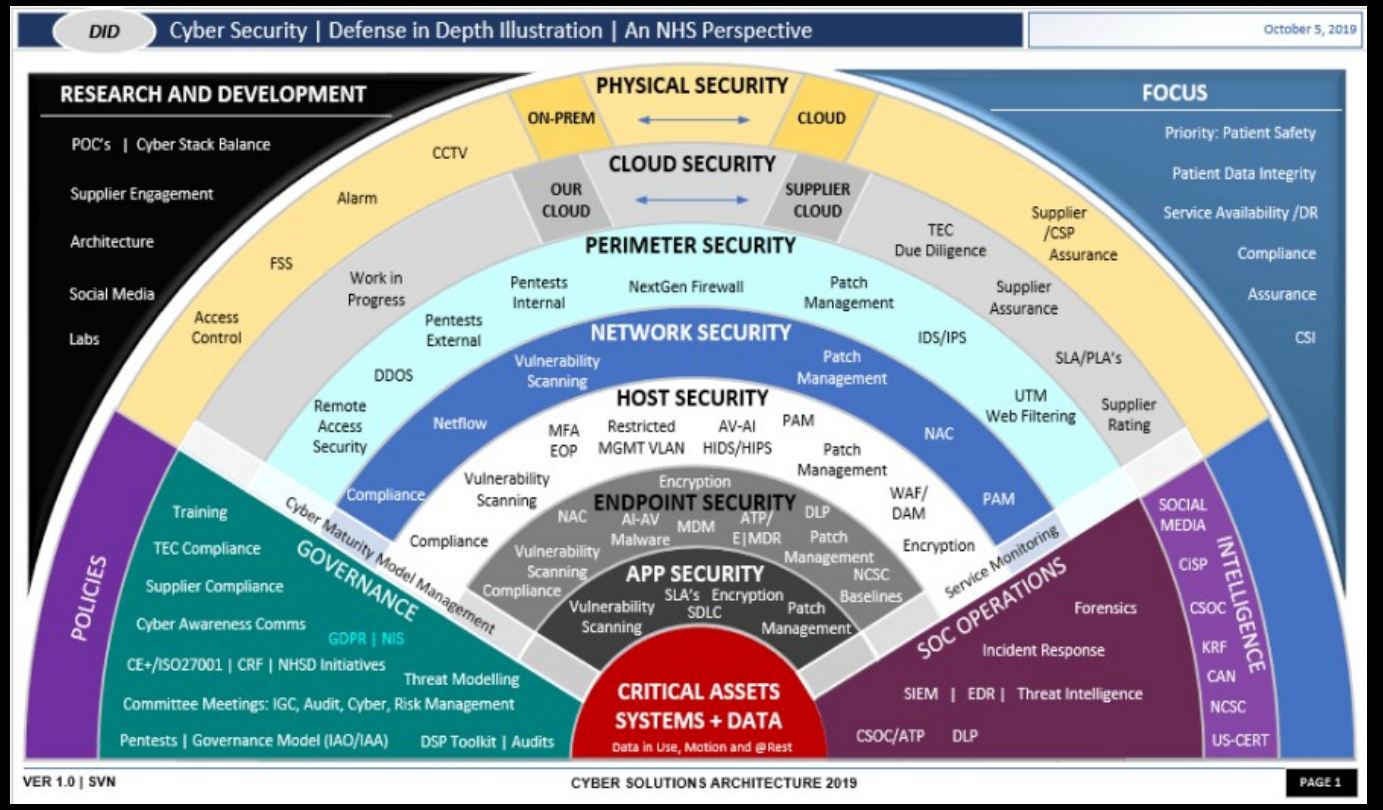
Security Policy Presentation

Developer: *Jason Verrill*



OVERVIEW: DEFENSE IN DEPTH

The proposed security policy will ensure defense in depth through standardized operations ranging from external attack surfaces to deeper, multi-layered defenses. This will limit the damage and cost should the outward defenses fail.



Green Pace

THREATS MATRIX

Likely SQL injection Array bounds error	Priority Numeric overflow Updated methods SQL sanitization Array bounding
Low priority Throw exceptions Verify error codes	Unlikely Deprecated code System errors Dependency errors Old active memory

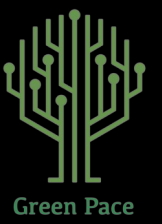
THREATS MATRIX

Security focus is on SQL injection attacks, array out-of-bounds programming errors, and numeric overflow/underflow errors. Good practice of try-catch exceptions is included, but not a top risk-based priority. Use static analysis tools and unit tests to verify code autonomously. Don’t use old deprecated code libraries and dependencies.

Green Pace

10 PRINCIPLES

- **Validate Input** –> STD-002-CPP, STD-003-CPP, STD-004-CPP, STD-005-CPP, STD-006-CPP
- **Heed Compiler Warnings** –> STD-002-CPP, STD-003-CPP, STD-006-CPP
- **Architect and Design for Security Policies** –> STD-002-CPP, STD-003-CPP, STD-004-CPP, STD-007-CPP, STD-008-CPP, STD-009-CPP, STD-010-CPP
- **Keep it Simple** -> STD-001-CPP, STD-009-CPP
- **Default Deny**
- **Adhere to the Principle of Least Privilege**
- **Sanitized Data Sent to Other Systems** - STD-004-CPP
- **Practice Defense in Depth** – STD-002-CPP, STD-006-CPP, STD-007-CPP, STD-008-CPP, STD-009-CPP, STD-010-CPP
- **Use Effective Quality Assurance Techniques** – STD-006-CPP, STD-007-CPP, STD-008-CPP, STD-009-CPP, STD-010-CPP
- **Adopt a Secure Coding Standard** – STD-001-CPP, STD-003-CPP, STD-006-CPP, STD-007-CPP, STD-008-CPP, STD-010-CPP



CODING STANDARDS

Rule	Priority
STD-002-CPP	High
STD-003-CPP	High
STD-004-CPP	High
STD-005-CPP	High
STD-007-CPP	Low
STD-008-CPP	Low
STD-001-CPP	Medium
STD-006-CPP	Medium
STD-009-CPP	Medium
STD-010-CPP	Medium

Green Pace

ENCRYPTION POLICIES

- All sensitive data needs to be encrypted
- Encryption needs to be implemented while data is at rest, in flight, and in use*
- Encryption needs to use up-to-date, strong, and well known encryption methods
- Sensitive data includes user account information (usernames, passwords, credit cards) and intellectual property. Data at rest includes data stored on all hard drives for servers, desktops, laptops, mobile devices and external storage devices (for example, USB drives). Data in flight includes information being sent over a network, whether internal or external (internet). Data in use includes information stored in active memory (open programs and files).
- *Not all data needs to be encrypted, just sensitive data. Attempting to encrypt all data in active memory would cause systems to be extremely slow and/or unusable. Memory in use is difficult to encrypt and should be used when reasonable.

Green Pace

TRIPLE-A POLICIES

- Authentication – Policies that support authentication include “Validate Input Data”. This helps to ensure that users are who they say they are.
- Authorization – Policies that support authorization include “Adhere to Principle of Least Privilege” and “Default Deny”. Users should be authorized to only have the privileges they need and for only as long as they need them. By default, interfaces and filters should blacklist by using a default access denial.
- Accounting – Policies that support accounting include “Architect and Design for Security Policies” and “Validate Input Data”. For example, logs should be kept and reviewed of user login attempts and higher level permission access.

Green Pace

Unit Testing

```
// Test that a collection is empty when created.
TEST_F(CollectionTest, IsEmptyOnCreate)
{
    // is the collection empty?
    ASSERT_TRUE(collection->empty());

    // if empty, the size must be 0
    ASSERT_EQ(collection->size(), 0);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, CanAddToEmptyVector)
{
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add one element to the collection
    add_entries(1);

    // is the collection still empty?
    EXPECT_FALSE(collection->empty());

    // if not empty, what must the size be?
    EXPECT_EQ(collection->size(), 1);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, CanAddFiveValuesToVector)
{
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add one element to the collection
    add_entries(5);

    // if not empty, what must the size be?
    EXPECT_EQ(collection->size(), 5);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, IsMaxSizeGESize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add one element to the collection
    add_entries(1);

    // verify that max_size is greater than or equal to the size
    EXPECT_GE(collection->max_size(), collection->size());


    // add four additional elements to make size = 5, then verify
    add_entries(4);
    EXPECT_GE(collection->max_size(), collection->size());

    // add five additional elements to make size = 10, then verify
    add_entries(5);
    EXPECT_GE(collection->max_size(), collection->size());

    // add five additional elements to make size = 20, then verify
    add_entries(10);
    EXPECT_GE(collection->max_size(), collection->size());
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, IsCapacityGESize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add one element to the collection
    add_entries(1);

    // verify that capacity is greater than or equal to the size
    EXPECT_GE(collection->capacity(), collection->size());

    // add four additional elements to make size = 5, then verify
    add_entries(4);
    EXPECT_GE(collection->capacity(), collection->size());

    // add five additional elements to make size = 10, then verify
    add_entries(5);
    EXPECT_GE(collection->capacity(), collection->size());

    // add five additional elements to make size = 20, then verify
    add_entries(10);
    EXPECT_GE(collection->capacity(), collection->size());
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesResizeIncreaseSize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // resize collection to 1, then verify
    collection->resize(1);
    EXPECT_EQ(collection->size(), 1);

    // resize collection to 5, then verify
    collection->resize(5);
    EXPECT_EQ(collection->size(), 5);

    // resize collection to 10, then verify
    collection->resize(10);
    EXPECT_EQ(collection->size(), 10);

    // resize collection to 20, then verify
    collection->resize(20);
    EXPECT_EQ(collection->size(), 20);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesResizeDecreaseSize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // resize collection to 20, then verify
    collection->resize(20);
    EXPECT_EQ(collection->size(), 20);

    // resize collection to 10, then verify
    collection->resize(10);
    EXPECT_EQ(collection->size(), 10);

    // resize collection to 5, then verify
    collection->resize(5);
    EXPECT_EQ(collection->size(), 5);

    // resize collection to 1, then verify
    collection->resize(1);
    EXPECT_EQ(collection->size(), 1);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesResizeDecreaseSizeToZero) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add 20 entries to the collection, then verify elements and size
    add_entries(20);
    EXPECT_FALSE(collection->empty());
    EXPECT_EQ(collection->size(), 20);

    // resize collection to 0, then verify
    collection->resize(0);
    EXPECT_TRUE(collection->empty());
    EXPECT_EQ(collection->size(), 0);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesClearEraseCollection) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add 20 entries to the collection, then verify its not empty and size equals 20
    add_entries(20);
    EXPECT_FALSE(collection->empty());
    EXPECT_EQ(collection->size(), 20);

    // clear collection, then verify its empty and has size equals 0
    collection->clear();
    EXPECT_TRUE(collection->empty());
    EXPECT_EQ(collection->size(), 0);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesEraseEraseCollection) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // add 20 entries to the collection, then verify its not empty and size equals 20
    add_entries(20);
    EXPECT_FALSE(collection->empty());
    EXPECT_EQ(collection->size(), 20);

    // clear collection, then verify its empty and has size equals 0
    collection->erase(collection->begin(), collection->end());
    EXPECT_TRUE(collection->empty());
    EXPECT_EQ(collection->size(), 0);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesReserveIncreaseCapacityButNotSize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // Verify capacity is 0 and size is equal to 0
    EXPECT_EQ(collection->capacity(), 0);
    EXPECT_EQ(collection->size(), 0);

    // Reserve 1 element space, and verify capacity has increased to 1 and size is still equal to 0
    collection->reserve(1);
    EXPECT_EQ(collection->capacity(), 1);
    EXPECT_EQ(collection->size(), 0);

    // Reserve 5 element spaces, and verify capacity has increased to 5 and size is still equal to 0
    collection->reserve(5);
    EXPECT_EQ(collection->capacity(), 5);
    EXPECT_EQ(collection->size(), 0);

    // Reserve 10 element spaces, and verify capacity has increased to 10 and size is still equal to 0
    collection->reserve(10);
    EXPECT_EQ(collection->capacity(), 10);
    EXPECT_EQ(collection->size(), 0);

    // Reserve 20 element spaces, and verify capacity has increased to 20 and size is still equal to 0
    collection->reserve(20);
    EXPECT_EQ(collection->capacity(), 20);
    EXPECT_EQ(collection->size(), 0);
}
```

Unit Testing

```
TEST_F(CollectionTest, IsExceptionThrownWhenAtIsOutOfBounds) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // Add entries to the collection
    add_entries(5);

    // Verify that .at() with collection size - 1 does NOT throw an exception (indexing starts at 0)
    EXPECT_NO_THROW(collection->at(collection->size() - 1), std::out_of_range);

    // Verify that .at() with collection size DOES throw an exception (indexing starts at 0)
    EXPECT_THROW(collection->at(collection->size()), std::out_of_range);
}
```

Green Pace

Unit Testing

```
TEST_F(CollectionTest, DoesAccessingOldLocationOfShiftedElementThrowException) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // Add entries to the collection
    add_entries(10);

    // Statically capture last element location
    const unsigned int lastElementIndex = collection->size() - 1;

    // Access element at max size
    // Should not throw an exception
    EXPECT_NO_THROW(collection->at(lastElementIndex));

    // Erase 1 element in the middle of the vector
    collection->erase(collection->begin() + 4);

    // Reaccess element at previous max size
    // Should throw an out of range exception because last element was shifted and the vector resized by erase()
    EXPECT_THROW(collection->at(lastElementIndex), std::out_of_range);
}
```

Green Pace

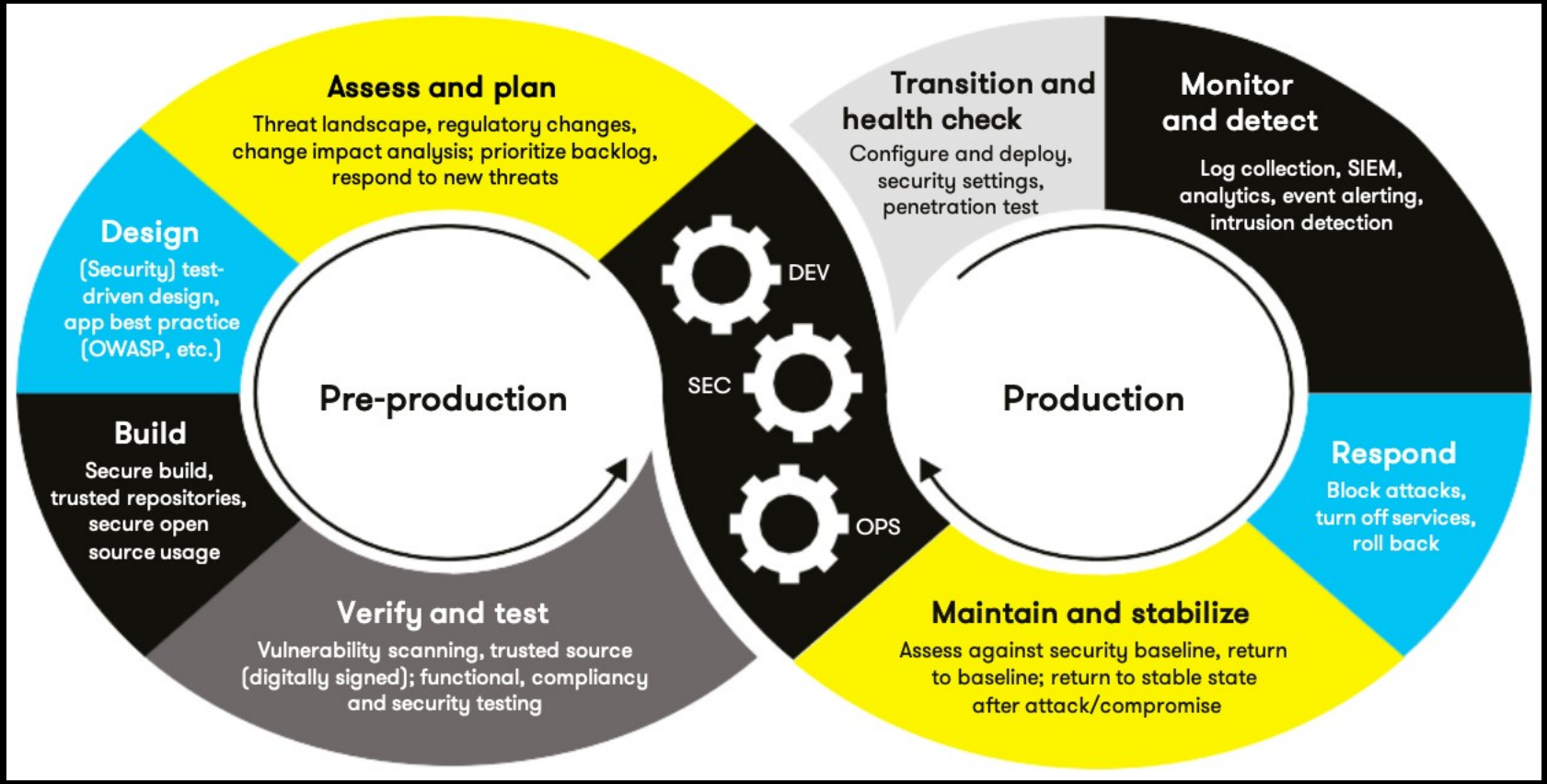
Unit Testing

```
TEST_F(CollectionTest, DoesPopBackDecreaseSize) {
    // Verify collection is empty and size is equal to 0
    ASSERT_TRUE(collection->empty());
    ASSERT_EQ(collection->size(), 0);

    // Add entries to the collection and verify size
    add_entries(10);
    EXPECT_EQ(collection->size(), 10);

    // Pop element and verify new size
    collection->pop_back();
    EXPECT_EQ(collection->size(), 9);
}
```

AUTOMATION SUMMARY



Green Pace

TOOLS

Visual Studio IDE includes an on-the-fly basic code analyzer to catch many basic errors as the developer writes code. Developers should be frequently compiling and testing their code every few lines which is where the built in analyzer will flag some issues. Once a particular section of code is complete (such as an entire function, method, or class), Cppcheck will need to be used for a deeper scan of the code. Issues flagged must be resolved or else be a false positive. Once flagged issues are resolved, the scan must be run again to verify that no new vulnerabilities have been introduced. This process should be repeated until no issues are visible or are all false positives.

Once a full block of code has been written (again, a function, method, or class), it must be tested using GoogleTest to ensure it functions as expected and handles unexpected use appropriately. Use of the written code and mock tests need to be conducted to ensure the code integrates with the rest of the system as well.

Green Pace

RISKS AND BENEFITS

Defense in depth is a strategy that attempts to defend not only the front line of assault, but to have additional fallback defenses should the front defense fail (*Cyber Security Minute: How does defense in depth work?,* 2017). With each layer of additional defense, not only does security rise, but so does the cost. Assessments will need to be made to weigh the cost versus benefit. Questions such as "How much of a target is your data worth to attackers?" (bank versus personal computer) and "What is your budget?" need to be answered.

Projects are on a limited time-frame, have a limited budget, and have limited human resources. It is important to focus on where attacks are most likely to occur and where an application is most susceptible to attack. It is usually easiest to break in through the front door (so to speak), where users can unknowingly let attackers in. Ensuring users only have access to what they need, for only as long as they need it, and understand what to watch out for will increase security. Also, once security has been breached, reputation is damaged which can be impossible to repair completely.

Green Pace

RECOMMENDATIONS

- Many more standards could be applied for deeper understanding and practices of the policies listed. For example, standards relating to how to implement default denial or least privileges on certain systems. Details could be included as to who gets what permission and why. Standards such as these ought to be added in a future addendum.

Green Pace

CONCLUSIONS

- When creating additional standards, being reasonable is important. While security is critical, understanding time and budget constraints must also be analyzed and appraised. Good judgment needs to be applied when creating further standards and must be realistic. Automation is important to keep developers productive, on schedule, and compliant (Vijayan, 2022).

Green Pace

REFERENCES

- YouTube. (2017). Cyber Security Minute: How does defense in depth work? YouTube. Retrieved November 5, 2022, from https://www.youtube.com/watch?v=InIRqw0jutA.

- Vijayan, J. (2022, November 16). 6 devsecops best practices: Automate early and often. TechBeacon. Retrieved December 6, 2022, from https://techbeacon.com/security/6-devsecops-best-practices-automate-early-often

Green Pace

Summary

- 10 PRINCIPLES
- CODING STANDARDS
- ENCRYPTION POLICIES
- TRIPLE-A POLICIES
- Unit Testing
- AUTOMATION SUMMARY
- TOOLS
- RISKS AND BENEFITS
- RECOMMENDATIONS
- CONCLUSIONS
- REFERENCES