**Slide 1:**
Hello my name is Michael Hanlon and today I will be talking about our prospective security policy.

**Slide 2:** Overview Slide

**Slide 3:**
Data Type: Severity is marked as low as a data type error will likely not be very cost heavy to fix and the damages likely won't be much either. Can be automated by sanitizing data going in and out of any different portion of the system whether that be database to server or functions.

Data Value: Should be a high priority especially in the design phase of a system as it can not only threaten data breaches, through buffer overflow, but also shut down a system in its entirety. These types of attacks can be prevented heavily in the design phase through string and value checking.

String Verification: Severity medium-high as buffer overflow is still a threat, as well as memory issues as well. These things can be prevented by first checking string length as well as validity. Through unit testing we can reveal many different types of errors as well as vectors for attack, and static code analysis can also be utilized to identify potential threats.

SQL Injection: One of the highest priority threats in our security policy, SQL Injection can leak sensitive data, as well as perform functions usually not permissible in a database. Through authentication, not only at login, but also for any potentially dangerous functions will ensure that not only someone who isn't supposed to be within certain scopes of a database is not there, but also against anyone even making a mistake in functionality.

Memory: Memory attacks and even protection against vulnerabilities are one of the most common security practices. Buffer and Stack overflow are one way memory can be exploitable, and improper handling pointers or memory storage can also lead to issues.

Assertions: Assertions ensure that certain parts of the system are always true, such as type, exception handling, as well as constant values. If certain checks are not made ahead of time assertions or lack thereof can range anywhere from system crashes to even data leaks, so prevention and response is a key factor here.

Exceptions: How an exception is handled can determine whether or not a program crashes or performs unexpectedly. Ensuring each possible exception is well handled and dealt with is a key priority in security. Automation such as cppcheck (static code analysis) can prevent potential exceptions from occurring, as well as unit testing which can find any potential invalid inputs or outputs to or from functions.

Encryption: Without encryption any potential attacks, whether at flight, use, or rest, are severe threats to data security. Encryption at all levels ensures that we won't have to worry about data

being lost or stolen as it will be protected at every level possible. Multifactor authentication is one way to protect against data being stolen in the first place, and encryption verification ensures that data will not be compromised or corrupted by improper encryption.

No Except: Potential violations of noexcept functions are dangerous for system health as well as maintenance. Although its a low priority issue, as it may not be vulnerable for attack, it is important for overall system maintenance that everything that should not throw an exception does not, this will increase maintainability as it easier to train new programers to work on well defined and typed systems like this. This will increase overall robustness and allow for easier fixes when attacks do occur. Cppcheck is an easy way to tell if any noexcept function has any potential vectors for exceptions to occur.

Authentication: Ensuring that each and every user is authenticated with the system at both login and also when any important or critical functionality is ongoing increases security during all times. It is also important to track every user's activity to see if there is anything suspicious as well there. Entra ID is one way to increase authentication security as it is a multi factor identification system.

**Slide 4:**

Keep It Simple

- Keeping the code simple in its function allows for easier onboarding of new team members and upkeep of code in general, this also
- In Keeping it Simple by declaring different functions which are protected from exception increases readability, useability, and maintainability by the active team and newcomers alike
- Keeping it simple applies as adding unnecessary checks, not only protrudes efficiency but also robustness

Architect and Design for Security

- Architecting for data value vulnerability increases security, and should be a principle from the design stage as it is a well documented security vulnerability and easy to protect against
- Design always be cognizant of the vulnerabilities associated with different methods that will be accessing, storing, and executing, different blocks of memory
- Designing around exceptions is paramount in security as they define how a language is used and checked around
- Encryption should be a primary focus of your security design as it protects data even in the face of attacks, and XOR based is near impossible to brute force
- This exemplifies Designing for Security as it focuses on streamlining while also further protecting and understanding the system as a whole

Adhere to principle of least privilege

- Even if you type cast back the data becomes more vulnerable to attack, and breaks the principle of Least Privilege, adding unnecessary dangers and vectors of attack
- Least Privilege applies for injection as the dynamic sql leaves another vector for attack by leaving the functionality open ended to be misused

Defense In Depth

- For Defense in Depth, the SQL injection obviously invites many different vectors of attack and vulnerability so defending and planning for this across the board is vital to system health and robustness
- The above also shows defense in depth, as the system should ensure that all parts are covered even if the protection may seem redundant
- Defense in Depth is represented here as all exceptions should be prepped and prepared for across systems, and will also be handled by the system when they do eventually occur
- Defense in Depth applies for encryption as it is not enough to protect your data against attacks you should also protect your data in the case that an attack eventually does occur
- This enables Defense in Depth as it broadens the robustness of each function and entire system to run more smoothly throughout
- Defense in Depth applies here, as although in theory passwords would be protected by database, an additional level of “redundant” defense can not hurt, only help

Default Deny

- Encryption is a shining example of default deny as even if data is breached it is still protected, and MFA also trends toward this principle
- Default Deny applies here as passwords are not only protected at storing level, but little information will be given on what field was wrong

Sanitize Data Sent Between Systems

- The assurance that both the strings are correctly concatenated and strictly the same increases robustness throughout the system as it can be consistently worked with as a string, as it is processed throughout the system
- Data Sanitation, all incoming values should be checked prior to the function even being called ensuring that the data is already correct and the check within is just an additional security blanket

Validate Data Input

- By using stricter data types, the system becomes more robust and sturdy against attack
- Again validating the data, in this case for length, ensures that the system continues to run normally when given unexpected input
- In this case we don’t restrict the bounds explicitly with string, so that the data type can self manage its bounds as it is updated
- Data Validation comes back into play as it is wise to ensure that data stored in memory doesn’t get overwritten unexpectedly or used when it is incorrect
- Data validation ensures that we will not have unexpected exceptions occur when an unexpected value is given

Use Effective Quality Assurance Techniques

- Memory should be strained and tested in a controlled environment prior to deployment to ensure the different functions are properly storing, and deleting

Provide Only One Way to Do an Operation

- Only allowing one way to perform an operation applies here, as dynamic SQL leaves vulnerability for additional unexpected functions and behavior

- One Function relates to the strategy of cryptography as it shouldn't be done in multiple ways as that could lead to problems internally when accessing data

Heed Compiler Warnings

- Heeding Potential dangers in the warnings and accounting for there possible malfunctions/exceptions will further protect against unexpected behavior and errors at runtime

**Slide 5**

In Use:
Encrypting data in use describes the encryption of data while it is being actively processed and worked through. This seems impractical as it will make many functions seemingly unnecessarily complicated, but in fact it is vital to protect data at this stage as unencrypted data actively in use is much more likely to suffer from data breaches.

In Flight:
Encryption at flight describes the type of encryptions needed to protect data moving from device to device or within the cloud. It is used when passwords are being authenticated through third party systems or between an owned server. This is important to protect for authentication purposes obviously and data can be particularly vulnerable at flight so strong encryption is key.

At Rest:
Encryption at rest, is the encryption of data that is not actively being used and is often stored in a hard drive or database. This data at rest is often some of the most sensitive, such as bank information, address, and contact details. This makes it a target for attackers even though it may be harder to breach than data in transit. It is important to have strong encryption at rest for this reason so breaches are less likely to be successful.

**Slide 6**

Authentication is usually the verification of an end-user, to make sure they're actually who they are. This is usually through username and passwords, but now two factor authentication using your cell phone or email is also fairly common, especially in enterprise accounts. This policy is important to differentiate admins from basic users, as well as verify a valid user and operations. New users being added are usually authenticated, or confirmed through an email system or text system as well.

After authentication, authorization takes place to designate where different users are allowed to go, or what commands they can enter or perform. Authorization ensures bad actors will at least not have an easy time corrupting the system or even an ignorant user who accidentally sets off a command or function they shouldn't.

Gives the system information on the resources the end-user is taking up as well as authorization control. This is useful for learning where resources need to be devoted more, or if there may even be a bad actor abusing resources to try to take advantage of the system. This will also account for the different features the account was using such as file access and will enable the system to track down anyone misusing.

**Slide 7**
UNIT TESTING

**SLIDE 8**
Enough Space?
Test to verify space, instead of testing 4 specific values, potentially we could add an input variable that would allow for greater usability and variability among size

**SLIDE 9**
Enough Capacity?
This function will check to see if the collection given has enough capacity for multiple sizes, again to improve we could add a specific size input, potentially a length of a vector that is to be added to a collection

**SLIDE 10**
Test Resize
This function tests to see if the the resize function is actually working on a given collection by checking size before and after

**SLIDE 11**
Test Downsize
Similar to the last test we will check size before and after to determine if the resize can downsize a given collection

**SLIDE 12**
Test Resize to Zero
This tests if the resize function works when resizing to zero, All of these resize tests could be consolidated to one by changing the resize to a specific size across multiple collections

**SLIDE 13**
Test Clear
Tests to see if the clear function works by using clear() on the collection, and then returns with empty()? To return a true or false depending on if the collection has been cleared or not

**SLIDE 14**
Test Erase
Tests the erase function by checking if the size is zero after the erase occurred, including the start_size variable is vestigial code and should be removed

**SLIDE 15**
Test Reserve
This time the start size is utilized as we test to see if the size remained the same through the attempt of the reserve which increases capacity but not space

**SLIDE 16**
Break Range
This is a negative test, or it confirms that a negative will happen such as an exception out of bounds. If an exception out of bounds does not occur, the test will return false
**SLIDE 17**
Check Front Empty

After a clear the front of the collection should be empty, confirm this and return true if it is in fact empty

**SLIDE 18**

Check Push Back

Push Back should increase the size of the collection as it adds a new element, this tests to make sure that the size actually increases when push back is used on a collection

**SLIDE 19**

Unit Testing:

Testing upper and lower bounds of every type of input and output allows Developers to know in the preproduction phase 'Verify and test', what their vulnerabilities are, and how to properly manage them. This can also be applied in the 'Maintain and stabilize' phase of production

MFA:

Ensuring Multi-Factor Authentication across all relevant structures of systems ensures that all users are who they say they are, and allows us to respond accordingly, as we monitor and detect during the production, and deployment process

Rate Limiting:

Whether it is someone trying to brute force a password, or a buffer overflow, users should be rate limited across the board when submitting too many requests, as well as monitored closely from then on out. This should be planned for in pre-production, tested, and implemented in production, and post-production.

Static Code Analysis:

During production all code should be subject to static code analysis (cppcheck for c++), as it can identify more risks and dangers not immediately apparent at runtime to the compiler. This fits into monitor and detect as well as maintain and stabilize.

**SLIDE 20**

Risk and Benefits of Implementing Now vs Later

Should we implement the security policy now, while we risk loss of funding, time, and system efficiency, we will also increase security, protect company image in case of attack, protection from many different attacks, increase robustness and maintainability throughout the entire system. Should we wait the risks and benefits flip, as funding, time, and system efficiency will increase, and the benefits then decrease or evaporate completely in the protection against attack case

**SLIDE 21**

After implementation of the security policy continuous updating and researching into new and evolving security vulnerabilities and attack vectors to stay up to date with changing requirements is necessary. One type of attack we can look into protecting in the future is a time based encryption breaker. Different levels of encryption require different amounts of time, using a brute forcer to check many different encrypted strings, and how long it takes, one can

then reverse engineer the key and break encryption. While our system is not using DES it is easy to see through this article how crypto-breaking is becoming an ever growing attack. With the rise of crypto currencies and cryptography in general, it should be a concern going forward. How strong is our encryption?

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

**SLIDE 22**
Conclusions

Should Breaches occur, notify stakeholders, what this means is when an attack occurs everyone who holds any sort of stake in what the attack entailed should be notified shortly. Hiding the fact may hold the reputation of the company/system for a little while, but for the most part full transparency is beneficial for not only the company image, but for finding a solution for the vulnerable code as well.