Threat Modeling Report

Created on 06/02/2026 12:40:47

Threat Model Name: NXG Supply MTM

Owner: Gideon Daniel

Reviewer: Mr Ali

Contributors: Mr Murphy, Mr Muyiwa

Description:

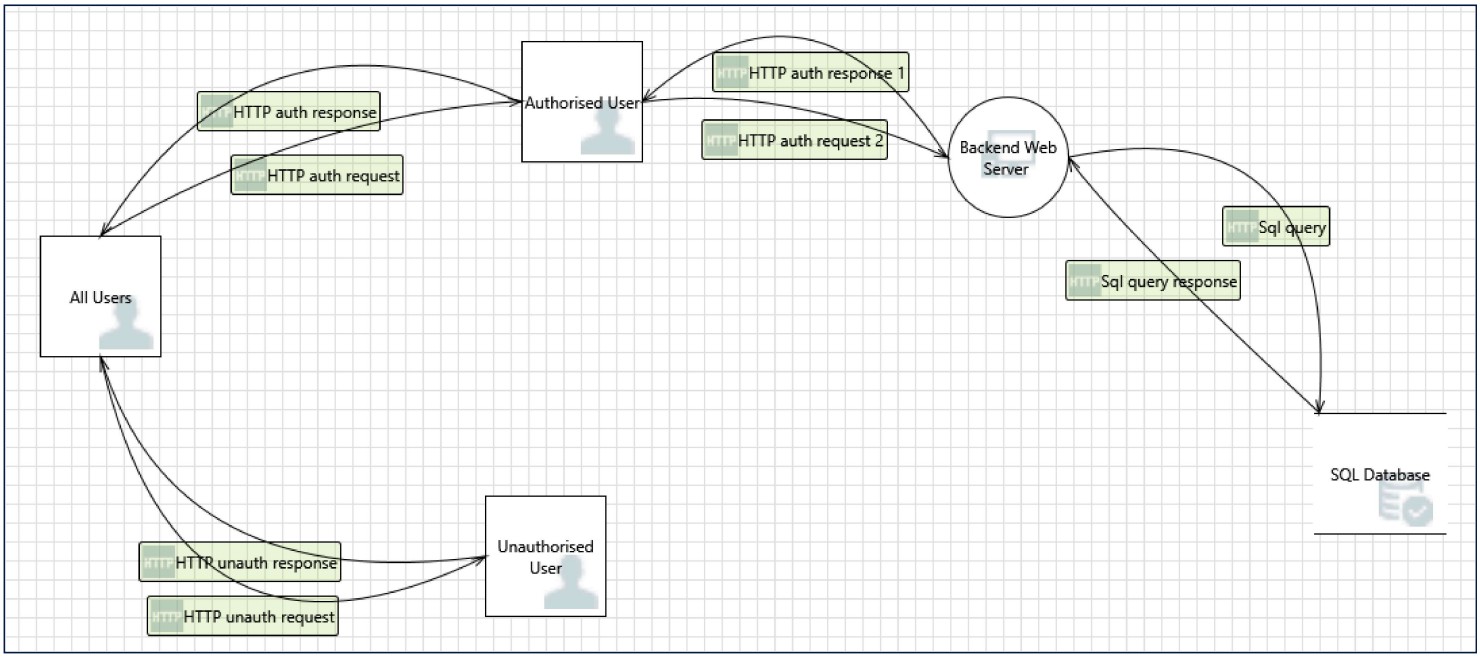
Assumptions:

External Dependencies:

Threat Model Summary:

Not Started	0
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	15
Total	15
Total Migrated	0

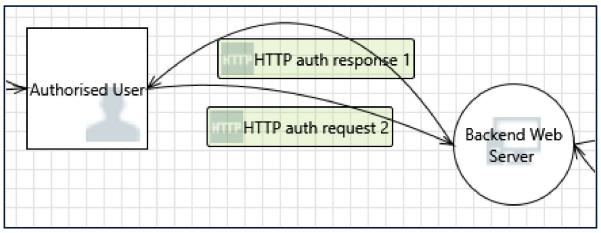
Diagram: HTTP response



HTTP response Diagram Summary:

Not Started	0
Not Applicable	0
Needs Investigation	0
Mitigation Implemented	15
Total	15
Total Migrated	0

Interaction: HTTP auth request 2



1. Spoofing the Authorised User External Entity [State: Mitigation Implemented] [Priority: High]

Category: Spoofing

Description: Authorised User may be spoofed by an attacker and this may lead to unauthorized access to Backend Web Server . Consider using a standard authentication mechanism to identify the external entity.

Justification: Ensure the use of standard Authentication Api

2. Elevation Using Impersonation [State: Mitigation Implemented] [Priority: High]

Category: Elevation Of Privilege

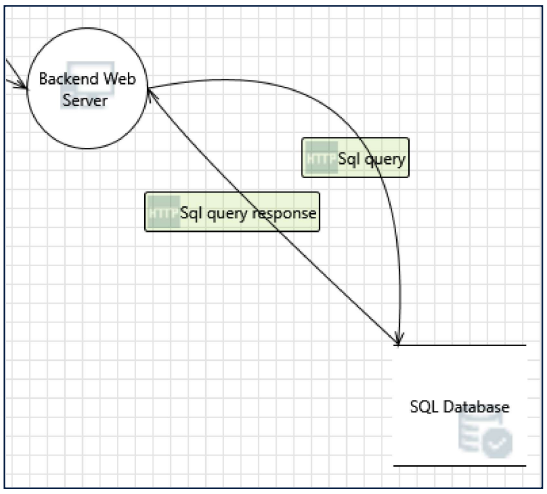
Description: Backend Web Server may be able to impersonate the context of Authorised User in order to gain additional privilege.

Justification: Isolation: Placing the server within a Secure Trust Boundary protected by firewalls and HIDS.

Restriction: Limiting the account to Read-Only access to prevent data tampering.

Logic: Utilizing a Trusted Subsystem Model, where the backend acts as a strictly controlled gatekeeper for all database interactions.

Interaction: Sql query



3. Spoofing of Destination Data Store SQL Database [State: Mitigation Implemented] [Priority: High]

Category: Spoofing

Description: SQL Database may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of SQL Database. Consider using a standard authentication mechanism to identify the destination data store.

Justification: Mutual Authentication (mTLS): Implement Mutual Transport Layer Security, which requires the SQL Database to present a valid digital certificate that the Backend Web Server must verify before any data is sent. If the certificate is missing or self-signed by an attacker, the connection is dropped.

Encrypted Connection String Validation: Configure the connection string with TrustServerCertificate=False and Encrypt=True. This forces the server to validate the database's identity against a trusted Root Certificate Authority (CA).

Service Principal Names (SPN) & Kerberos: In a Windows/Active Directory environment, using Kerberos authentication ensures the Backend Web Server verifies the SPN of the database. An attacker cannot spoof this without the secret keys from the Domain Controller.

Hardened Connection Endpoints: Hard-code the database identity using a Private DNS or a static IP address within a Secure Trust Boundary, and using firewall rules to block the Backend Server from making any outbound database connections to the public internet.

4. Potential SQL Injection Vulnerability for SQL Database [State: Mitigation Implemented] [Priority: High]

Category: Tampering

Description: SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker.

Justification: To address the potential for advanced SQL injection, we have implemented a multi-layered defense. 1) Frontend input validation restricts character sets (whitelisting). 2) Backend parameterized queries prevent command injection. 3) A Web Application Firewall (WAF) is configured with SQLi detection patterns to block malicious payloads before they reach the application. This redundant approach ensures that if one layer is bypassed, the database remains protected.

5. Potential Excessive Resource Consumption for Backend Web Server or SQL Database [State: Mitigation Implemented] [Priority: High]

Category: Denial Of Service

Description: Does Backend Web Server or SQL Database take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.

Justification: This threat is addressed through Defense-in-Depth. At the network edge, we implement Rate Limiting to prevent individual users from monopolizing server resources. On the backend, we handle resource requests asynchronously to prevent thread-blocking. We have also delegated low-level memory management to the Operating System's resource governor while maintaining application-level oversight through heartbeat monitoring. These steps ensure that resource requests do not result in a system hang or a permanent 'Locked' state for authorized users.

6. Lower Trusted Subject Updates Logs [State: Mitigation Implemented] [Priority: High]

Category: Repudiation

Description: If you have trust levels, is anyone other outside of the highest trust level allowed to log? Letting everyone write to your logs can lead to repudiation problems. Only allow trusted code to log.

Justification: Privileged Logging Access: Restrict write access to log files and the logging service exclusively to High-Trust Processes (e.g., the Backend Web Server or Kernel-level services).

Asynchronous Relays: Use a secure, one-way logging relay where the application sends data to a "buffer," and a separate, higher-trusted process handles the actual writing to the disk.

Identity Tagging: Every log entry should be automatically tagged with the verified identity and trust level of the source process. This ensures that even if a lower-trust subject logs data, it is clearly marked as "Untrusted" during an investigation.

WORM Storage: Store logs on Write-Once-Read-Many (WORM) media or in a centralized SIEM (like Splunk or Azure Monitor) where even the original logging process cannot delete or modify an entry once it is sent.

7. Risks from Logging [State: Mitigation Implemented] [Priority: High]

Category: Tampering

Description: Log readers can come under attack via log files. Consider ways to canonicalize data in all logs. Implement a single reader for the logs, if possible, in order to reduce attack surface area. Be sure to understand and document log file elements which come from untrusted sources.

Justification: Data Canonicalization: Sanitize and clean all data before it is written to the log. This ensures that special characters (like <script> tags or command-line symbols) are neutralized so they cannot execute when the log is opened.

Attack Surface Reduction: Utilize single, hardened log reader or a centralized SIEM (like Splunk) rather than allowing multiple different tools to open raw log files.

Untrusted Source Documentation: Explicitly labeling log data that originates from external users (e.g., IP addresses, User-Agent strings) to alert investigators that the data may be untrusted.

Digital Signing: Implementing cryptographic hashing or digital signatures on log files to ensure that any tampering with existing logs is immediately detectable.

8. Weak Credential Storage [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: Credentials held at the server are often disclosed or tampered with and credentials stored on the client are often stolen. For server side, consider storing a salted hash of the credentials instead of storing the credentials themselves. If this is not possible due to business requirements, be sure to encrypt the credentials before storage, using an SDL-approved mechanism. For client side, if storing credentials is required, encrypt them and protect the data store in which they're stored

Justification: Server-Side Hashing: Utilize Salted One-Way Hashes (e.g., Argon2 or bcrypt) for passwords. This ensures that even if the database is breached, the actual plain-text passwords cannot be recovered.

Encryption as Fallback: If raw credentials must be stored for legacy integration, they are protected using AES-256 encryption with keys managed in a secure Hardware Security Module (HSM) or Key Vault.

Client-Side Hardening: If local storage is necessary, credentials are never kept in plain text; they are encrypted using Platform-Specific Secure Storage (such as Windows Credential Manager or macOS Keychain) to prevent theft from the local file system.

9. Authorization Bypass [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: Can you access SQL Database and bypass the permissions for the object? For example by editing the files directly with a hex editor, or reaching it via filesharing? Ensure that your program is the only one that can access the data, and that all other subjects have to use your interface.

Justification: OS-Level Locking (ACLs): Use NTFS permissions to restrict raw database file access (.mdf/.ldf) exclusively to the Database Service Account, blocking all other users.

Data Cryptography (TDE): Implement Transparent Data Encryption so that any data stolen via hex editors or file copying remains unreadable without the master keys.

Service Hardening: Disable SMB/File Sharing services to close "backdoor" network paths to the storage folder.

Full Disk Encryption: Use BitLocker to protect the physical or virtual drive, preventing data theft via disk cloning or physical drive removal.

10. Data Logs from an Unknown Source [State: Mitigation Implemented] [Priority: High]

Category: Repudiation

Description: Do you accept logs from unknown or weakly authenticated users or systems? Identify and authenticate the source of the logs before accepting them.

Justification: Source Authentication: Make sure all systems and services authenticate (via API Keys, Mutual TLS, or Service Accounts) before they are permitted to transmit logs to the central collector.

Endpoint Validation: Implement a Whitelisting Policy that only accepts log traffic from known, trusted IP addresses or internal subnets within the Generic Trust Border Boundary.

Cryptographic Binding: Using Digital Signatures or HMACs on log batches at the source. This ensures that the log collector can verify the data originated from a legitimate, identified system and has not been altered in transit.

Strict Metadata Tagging: Automatically append immutable metadata to every log entry, including the verified Source ID and Timestamp, to prevent an attacker from "injecting" logs that appear to come from another system.

11. Insufficient Auditing [State: Mitigation Implemented] [Priority: High]

Category: Repudiation

Description: Does the log capture enough data to understand what happened in the past? Do your logs capture enough data to understand an incident after the fact? Is such capture lightweight enough to be left on all the time? Do you have enough data to deal with repudiation claims? Make sure you log sufficient and appropriate data to handle a repudiation claims. You might want to talk to an audit expert as well as a privacy expert about your choice of data.

Justification: Event Granularity: Configure logs to capture the "Full Context" of every transaction, including the User Identity, Source IP, Timestamp, Action Performed, and the Success/Failure status.

State-Change Tracking: Specifically audit all "Create, Update, and Delete" (CUD) operations within the SQL Database to ensure a permanent record of data modifications.

Balanced Logging Level: Implement an optimized logging schema that is detailed enough for forensics but lightweight enough to remain active 24/7 without degrading the performance of the OS Processes.

Privacy-Aware Logging: Ensure that while logs are detailed, they are stripped of sensitive Personally Identifiable Information (PII) or clear-text credentials to comply with privacy regulations (like GDPR) while maintaining security value.

12. Potential Weak Protections for Audit Data [State: Mitigation Implemented] [Priority: High]

Category: Repudiation

Description: Consider what happens when the audit mechanism comes under attack, including attempts to destroy the logs, or attack log analysis programs. Ensure access to the log is through a reference monitor, which controls read and write separately. Document what filters, if any, readers can rely on,

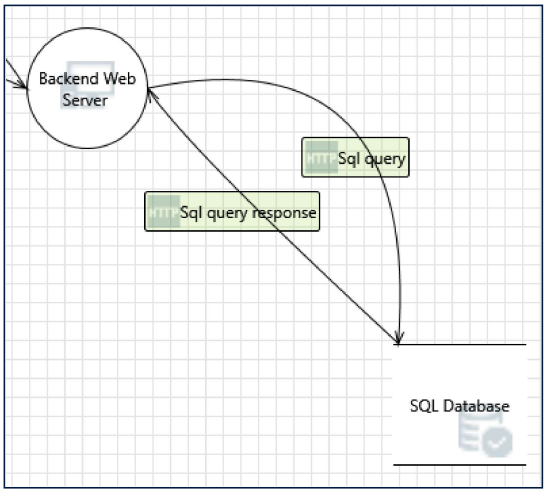
or writers should expect

Justification: Access Separation (Reference Monitor): Implement a strict "Reference Monitor" model where read and write permissions are handled by separate processes. The application can only write (append) to the log, while only a dedicated security auditor account can read or delete it.

Centralized Off-Site Logging: Automatically stream logs in real-time to a remote, hardened logging server (such as Splunk or Azure Log Analytics). Once the log is sent off-site, a local attacker cannot delete it to hide their footprint.

Immutable Storage: Utilize WORM (Write-Once-Read-Many) storage or "Object Locking" in the cloud, which physically prevents any data from being edited or deleted for a set retention period.

Interaction: Sql query response



13. Spoofing of Source Data Store SQL Database [State: Mitigation Implemented] [Priority: High]

Category: Spoofing

Description: SQL Database may be spoofed by an attacker and this may lead to incorrect data delivered to Backend Web Server . Consider using a standard authentication mechanism to identify the source data store.

Justification: Mutual Authentication: Use mTLS or Kerberos (SPNs) to ensure both the web server and the database must prove their identity before a connection is established.

Certificate Validation: Enforce encrypted connection strings that reject unverified certificates, ensuring the backend only communicates with a legitimate, trusted data store.

Network Pinning: Restrict traffic to static IPs or private DNS entries via firewall rules, preventing the application from being redirected to a rogue external source.

14. Weak Access Control for a Resource [State: Mitigation Implemented] [Priority: High]

Category: Information Disclosure

Description: Improper data protection of SQL Database can allow an attacker to read information not intended for disclosure. Review authorization settings.

Justification: Least Privilege (RBAC): Restricting the backend service account to specific roles (like db_datareader) to prevent broad administrative access.

Identity-Based Filtering (RLS): Implementing Row-Level Security so users can only view their own data, preventing "Horizontal Privilege Escalation."

Obfuscation (Data Masking): Using Dynamic Data Masking to hide sensitive PII/financial data from unauthorized eyes during standard viewing.

Encryption at Rest (TDE): Utilizing Transparent Data Encryption to protect the physical database files from theft or unauthorized disk access.

15. Risks from Logging [State: Mitigation Implemented] [Priority: High]

Category: Tampering

Description: Log readers can come under attack via log files. Consider ways to canonicalize data in all logs. Implement a single reader for the logs, if possible, in order to reduce attack surface area. Be sure to understand and document log file elements which come from untrusted sources.

Justification: Input Canonicalization: All data is sanitized and converted into a "safe" standard format before being written to the log. This neutralizes executable characters (like < > ; &) so they are treated as harmless text rather than commands.

Centralized, Hardened Log Reader: Instead of allowing various tools to open raw log files, a single, secure SIEM (like Splunk or ELK Stack) is used. These platforms are designed to display log data safely without executing embedded scripts.

Untrusted Source Labeling: All data originating from outside the Trust Boundary (such as User-Agent strings, Referrer URLs, or input fields) is explicitly tagged as "Untrusted" in the log schema to alert readers to handle the data with caution.

Output Encoding: The log viewing tool is configured to use strict output encoding, ensuring that even if a malicious string was logged, it cannot be rendered as active code by the viewer's browser or console.