Security

“Writing secure software requires deep knowledge of how /everything/ works. Naively following ‘best practices’ without understanding the reasons behind them is a recipe for accidentally making mistakes that introduce new security holes” - Simon Willison.

“Security and innovation is driven by different people with conflicting goals” – Lars Albertsson.

“Security complexity comes from engineering complexity that itself comes (mostly) from organization complexity” – Guillaume Montard.

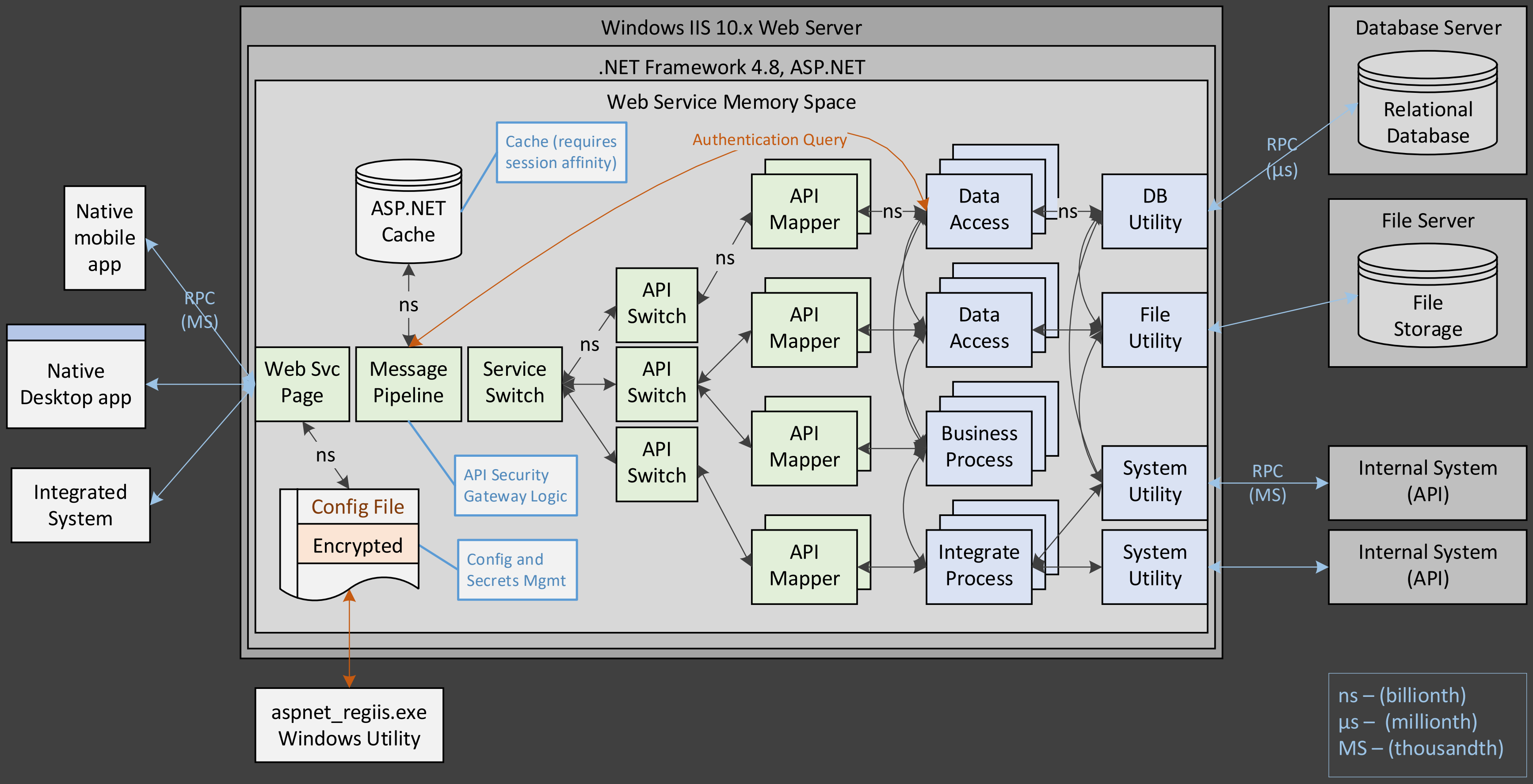
For these and other reasons, strong security is not something that can be bought or tacked onto a system as an afterthought (like a bandage) - it must be designed and built into each software system from the ground up by people that “know everything” about that system at a deep level. The more complex a system is, the more difficult this process becomes.

The consequences of poor security can result in extensive downtime, a ruined reputation or brand, lost sales, fines and lawsuits. Strong security is frequently a prerequisite for many certifications of compliance such as PCI-DSS, HIPPA, GDPR, etc. However, security is a cost center, so the challenge when designing software systems is to provide strong, effective security at the lowest possible cost, and without negatively impacting the productivity of developers and users of the system.

For DGP, this is accomplished by

1. Using message-based APIs for all functionality, which are backed by integrated zero-trust, data-driven RBAC identity stores.
2. Simplification of the system architecture and code as much as possible to reduce the complexity of the software system itself. This is accomplished by using a hub-and-spoke overall architecture, with a modular multi-tiered hub.
3. Designing multiple reinforcing layers of security into the system architecture from the ground up, which themselves are also as simple as possible. Simplicity of the architecture and code helps to improve the strength and effectiveness of security.

This “DevSecOps” process that merges the requirements of development, testing, security and operations with all the business requirements of a software system generally occurs at the design stage of each iteration of a software systems’ implementation.



One way of looking at a DGP system is that all functionality within the hub is implemented as reusable libraries of code (shown in blue above). Whenever possible, these libraries call each other within the shared memory space of each web server. This practice greatly improves performance and also improves the security of the system as well. The message-based web service APIs (shown in green above) are used to provide a very secure, high-performance mechanism to remotely call the functionality of some of those internal libraries. In addition, it is assumed that 1) locations have simple, strong and effective network perimeter security, and 2) TLS or DGP encryption is being used to encrypt API messages during transport over networks. DGP uses message-based RPC web service API's with a single-argument front controller pattern. This provides many benefits, including the ability to expose the minimum possible external attack surface (a single controller method that accepts a single input parameter).

The single method and single input parameter of each web service, along with size restrictions and the lack of strong data types in API messages all combine to effectively prevent buffer overflow attacks. In addition, DGP has been designed with the equivalent of an API security gateway integrated into each web service (the message pipeline class). This class is reused by all web services, and checks for the correct structure of each message, the message TTL, user account authentication, user account rate limit, failed authentication count plus account authorization (via an ACL) for each API method called by the request message batch.

Also strengthening security are the 2 types of encryptions used inside of DGP systems. Application-level encryption is used in the web services to encrypt record values before they are stored in the database, and decrypt values retrieved from the database before they are returned to a client application. The second type of encryption is used to protect the secret values in the .config files using Microsoft's PKI, which is transparent to the web services and applications and is used to provide the same functionality as an HSM.

The single-argument front controller of each web service only accepts an XML fragment as the input to its controller method. The message is limited to a maximum of 80K in size, while it takes 2 GB of data to overflow a text input. The formatted text has no header or metadata and no strong data types, which effectively eliminates all XML-related vulnerabilities.

The message is read as a forward scrolling cursor through a memory stream, so there are also no serialization/deserialization vulnerabilities to be exploited within the XML reader itself. All inputs and returned data are encapsulated in CDATA blocks, so any special characters contained inside them are ignored and have no effect on the structure of the message.

Once the API request message has been read to populate instances of several custom types, the API request data is passed into the message-processing pipeline, which is a single class that is reused in all web services. It enforces all of the API security checks, account authentication, authorization of each internal method call, and so on.

User account authentication and API method authorization depend on the security data maintained by the data-driven Role Based Access Control (RBAC) security system. By default, the web services use a SHA-256 hash mechanism to authenticate each API request message sent from the client to the server, and can optionally use an HMAC hash mechanism for even better security as desired.

The RBAC security system is "zero-trust", and all accounts (including admins) initially have no authorization to access to any API methods. Access to methods must be assigned to each account, and is managed via role memberships. The role memberships of each account are used to create a superset of all the authorization of all the roles they belong to (a whitelist/ACL). A lazy just-in-time process caches the ACL data in each user's account record to improve security performance.

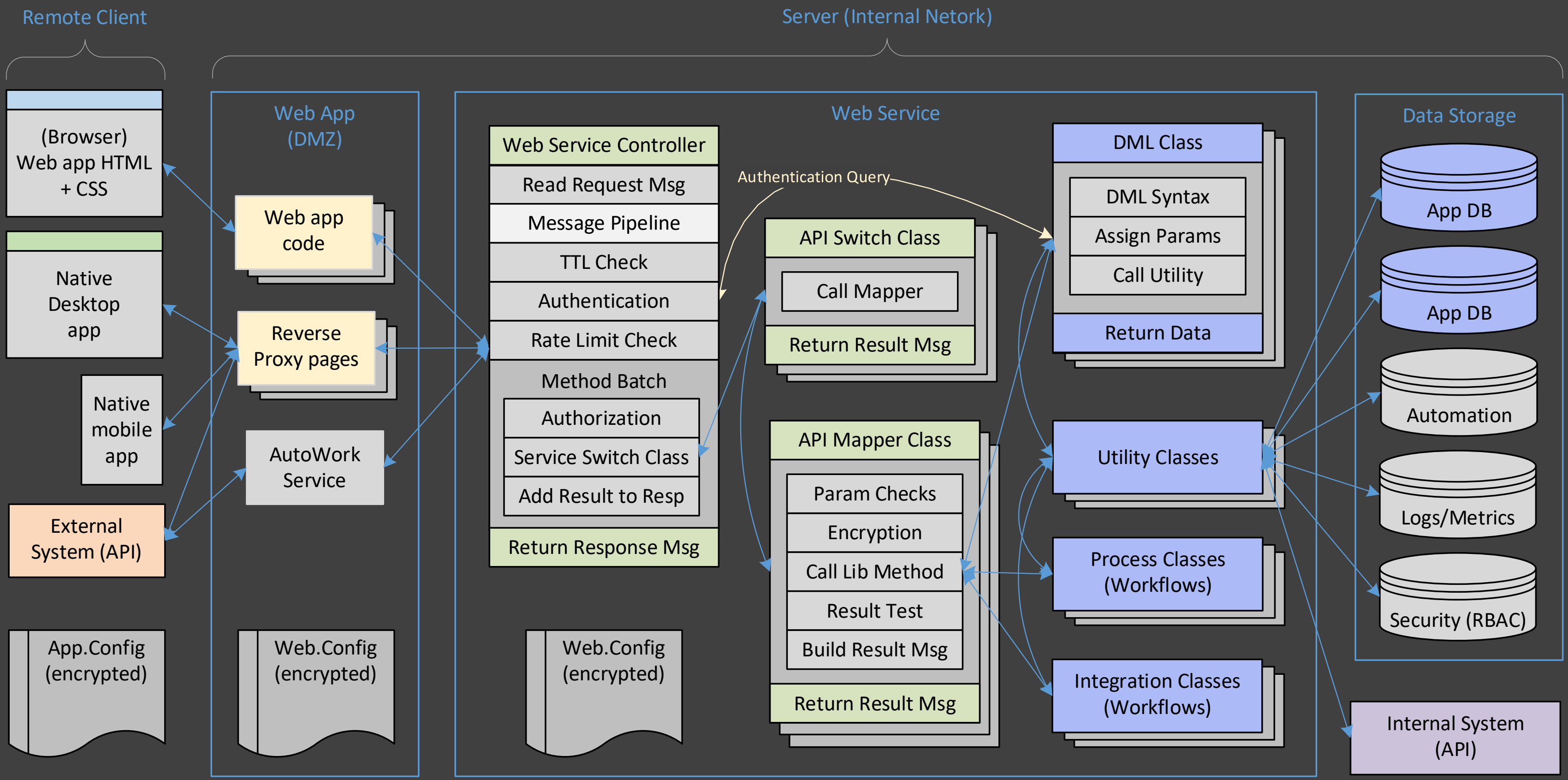
DGP has no default admin account, admin password, or naming conventions for the web services or databases used in a DGP system. All of those values must be defined by administrators when a system is initially deployed and configured. This eliminates on of the most common security vulnerabilities – default admin account passwords that are not reset by admins as intended, along with well-known names and endpoints of system resources due to static values embedded in systems.

- Zero-day exploits. The best practical mitigation for zero-day exploits is immediate patching of the OS and all application servers, etc. Windows arguably has the best patching and update mechanisms of the top 3 computer operating systems (Windows, MacOS, Linux). Windows Update patches all of the parts of DGP and DGPDrive automatically.
- "Supply Chain" exploits (open source). One of the most popular attack vectors for hackers is to inject their malicious code into open-source libraries. These libraries are then included in open-source tools, and the tools are used to build software systems, effectively deploying the malicious code into those software systems. DGP only has 2 dependencies in its software bill of materials (SBOM): 1. Windows (includes the .NET Framework 4.8.1 and IIS 10.x web server) and 2. SQL Server, both of which are closed-source. No open-source tools are used to build DGP or DGPDrive prototype.

Messages that make it through all of the security checks of the pipeline are passed to the respective API mapper classes for execution. The mapper methods verify message inputs and map them to their respective internal library method parameters. Once that step is completed, the internal library method is executed.

The internal library data access methods are completely parameterized (immune to SQL injection attacks) and perform any application-level encryption of data to be stored in the database, as well as decryption of encrypted data retrieved from the database. For example, this is the process used to encrypt and decrypt user account passwords stored in the SysInfo database. These methods also enforce security group membership and access restrictions in the SQL syntax of the data access (DML) methods, which is used to enforce logical multi-tenancy partitioning as just one example.

The results of the internal method execution are evaluated in the mapper class as a form of simplified unit test. The results of the unit test are returned as a result code along with the data returned by the internal method call and server performance measurements in every API response message.



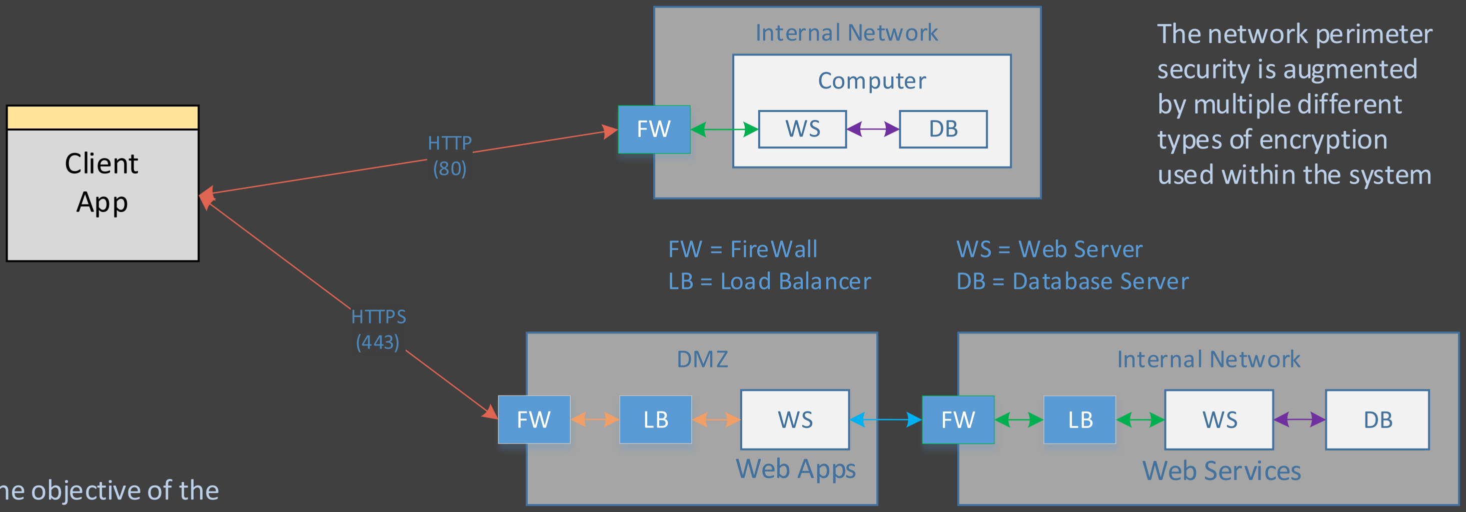
Security Verification

1. *Penetration tests are used to prove the effectiveness of the security built into the single-argument front controllers and the API security gateway integrated into each web service. There are no admin backdoors or ways to bypass the web service security. This includes all automated processes such as replication, file uploads and downloads, etc.*
2. *The final overall test of a system's security would be to pass a PCI-DSS audit. Systems built with this same architecture have passed 3 penetration tests (with no vulnerabilities found) and 2 PCI audits in the recent past. The PCI-DSS audit is not necessary for most systems, but it does serve as a good, objective standard for security even if no audit is ever performed.*
3. *Lattice and DGP do not use any 3rd party frameworks, tools, utilities or code libraries. Their only dependency is on Microsoft's full (closed-source) .NET Framework 4.8.1. This eliminates the need for expensive software to scan the tree all of the open-source projects recursively for the inclusion of any malicious code (all the way down to the lowest level open-source utilities). Also, it is not clear how effective those scans for malicious code embedded in open-source projects really are in practice, so the most effective solution is to eliminate the need for those types of scans in the first place.*
4. *The reverse proxy pages are used when the location has grown large enough to have a DMZ. They act as a simple pass-through that is tested by posting messages to their URL, just like a web service. They also control which internal web services can be accessed from external networks.*
5. *HTTP POST MIME type restriction to only accept "text/xml" requires a tool such as Postman to post non-XML requests to the web service to test the enforcement of the MIME type rule. Also, attempting to upload a file to the web service (which requires* multipart/form-data *MIME type) also violates the MIME type rule.*
6. *API request XML fragment restriction requires a tool such as Postman to post an XML document to the web service and test the enforcement of the XML fragment rule.*
7. *API request size is shown in the test results of each method call in the API Tester. There is also a test file that sends a large request that exceeds the size limit to test the enforcement of the limit.*
8. *API response size is shown in the test results of each method call in the API Tester. There is also a test method that returns a large response that exceeds the size limit to test the enforcement of the limit.*
9. *API request TTL can be tested using the API Tester in debug mode, setting a breakpoint in the API Tester code, pausing the API request for a number of seconds greater than the limit, then allowing execution to continue and observing the TTL error response.*

10. *API method authorization can be tested using the API Tester by running a test file for an API that the user is not authorized to call. Authorization data can be viewed in Lattice with the User Methods form, or by using SSMS to view the value of the MethodList field of the APIUser table in the SysInfo database. The steps of the authentication and authorization process can be verified by stepping through the code of the message pipeline class. NOTE: DGP depends on TLS to encrypt the API request and response messages during transport.*
11. *API request rate limit is tested by a test account with a low rate limit using the API Tester to run a test file that calls many API methods sequentially (which will exceed the low rate limit of the test account).*
12. *Just-in-time updates to the cached authorization data by the Login method for standard accounts are verified in Lattice with the User Methods form, or by using SSMS to view the value of the MethodList field of the APIUser table in the SysInfo database.*
13. *The immediate updates to the cached authorization data for system accounts are also verified in Lattice in the same way as for standard accounts.*
14. *Data group authorization can be viewed in the ReadList and WriteList fields of an account record in SSMS and the Group Access form in the Lattice UI. A test account with a mix of ReadOnly and ReadWrite data group membership will also display the effects of those results in the folder tree and file edit forms of the FileStore application.*
15. *Config file encryption is verified by viewing the encrypted AppSettings section of the config file after encryption and decryption.*
16. *The HMAC hash value of the request time value is used to authenticate each API request message to the web service, using the account password as the secret key. This functionality can be verified by stepping through the code of the message pipeline class.*
17. *The HMAC hash value of the response time value is used to authenticate the results of the Login method to the client application, using the password as the secret key. Stepping through the code of the Connect form of the Lattice application can verify this functionality.*
18. *Zero knowledge encryption is used in the web services to encrypt data before it is stored in a database, and is verified using SSMS to view the encrypted contents of the Password field in the APIUser table of the SysInfo database. The decryption can be verified by stepping through the code of the authentication process in the message pipeline class.*
19. *The correct use of the encryption key history stored in each web.config file is tested by creating some test data, then adding a new encryption key, after which some more test data is created (so there is a mix of different keys), and then using a Lattice UI to edit the old and new test data. Stepping through the code of the encryption/decryption logic in the APIUser data access methods is another way to verify the correct functionality*

Network Perimeter Security

The network connectivity of a distributed system is generally its biggest weakness. In highly redundant, fault tolerant software systems (such as a distributed grid), firewall rule and configuration mistakes are the root cause of many system downtime events. All of the efforts to improve the quality of DGP software systems are useless if network errors, misconfigurations, etc. take the system offline or create vulnerabilities. The only practical solution to this problem is to simplify the network perimeter security itself.



The network perimeter security is augmented by multiple different types of encryption used within the system

The objective of the simplified network security is to provide good protection while also allowing the server farms to grow without any changes to the firewall rules (M&M security model)

The DMZ firewall rules only allow external traffic to the VIP of the DMZ load balancer, while the servers are allowed to call out to external API's etc.

The Internal firewall rules only allow traffic from the DMZ servers to the VIP of the Internal load balancer

The servers of the Internal network can communicate with each other freely, but are not allowed to call out to any servers outside the Internal network

Consolidation (Hub-and-Spoke)

Consolidation, which tries to collect all the parts of a system to run on a single computer or within a LAN in a single location, helps to significantly improve performance and security. Additional steps can be taken to deliberately simplify the network perimeter security to an even greater extent. The "M&M" model (hard outer shell, soft on the inside) augmented by various types of data encryption allows a system to grow as needed without requiring any changes to the firewall rules. This alone can significantly reduce firewall rule errors that result in system vulnerabilities and/or downtime.

In addition, the rules explained briefly in the diagram above are also a means of detecting any breaches of the network security. In general, the first thing that any malware must do is to make a call out to its command-and-control server. The firewall rules that block outbound calls from the internal network provide an easy way for attempted outbound calls to be detected and traced back to their source. This helps to solve the problem of the long duration of security breaches mentioned in the OWASP documentation.

Host Computer Security

The single most effective step to improve the security of software systems is prompt patching. For DGP systems, all tiers and parts of the system are patched automatically by Windows Update. The IIS web server and .NET Framework 4.8 are considered as part of Windows itself, and are patched accordingly. SQL Server Express and Standard are also patched by Windows Update when it is configured to also update applications.

In addition, there is documentation to "harden" each of the host computers (processing node web servers, storage node database servers) with a variety of configuration steps for the OS and application servers. Refer to the Windows, IIS and SQL Server documentation for more details.

Finally, endpoint security for each host computer is also important. For most systems this will consist of some type of virus scanning software. The Microsoft Defender anti-virus built into Windows 10 and Windows Server is a good alternative for the majority of systems, especially at a small scale. Another optional security measure is application whitelists. These can be configured as security policies on Windows server, but will require a 3rd party application for Windows 10. Since all logic runs on the server in a DGP system, the whitelist application for Windows 10 is optional.

Software Bill of Materials

A Software Bill of Materials (SBOM) is a list of all of the dependencies a software system has in terms of external code libraries, frameworks, tools, etc. It is useful when performing updates for all parts of a system, scanning open source software for malicious code, determining if versions of the dependencies contain vulnerabilities, etc.

DGP SBOM:

- [closed source] Windows 10/11 operating system for software development, very small systems and client applications.
- [closed source] Windows Server 2016 (or later) Standard Edition operating system or better for production systems.
- [closed source] IIS 10.x web server, which is part of Windows and updated by Windows Update.
- [closed source] .NET Framework 4.8.1, which is part of Windows and updated by Windows Update.
- [closed source] SQL Server 2016 (or later) Express and/or Standard Edition.

All of the items in the SBOM list are updated using Windows Update. In some cases, this also includes "cluster aware" updates of SQL Server Availability Groups and/or other clusters when applicable.

Neither DGP nor DGPDrive use any open-source libraries, NuGet packages, tools or frameworks other than the dependencies in the SBOM list above. This means that no scans for malicious software for open-source projects and their respective trees of dependencies are required.

API Security

The context for DGP security is that it focuses on the DGP software systems themselves while being an integral part of the overall security of an organization. Security tools that cover the NIST five phases (identify, protect, detect, respond and recover) provide the comprehensive security for an organization. Microsoft Defender for Business is one example that is tailored for organizations in the SMB market.

DGP security protects the web service APIs and data of software systems within that type of comprehensive security solution. As much as possible, all functionality in a DGP system is built as web service APIs, and the design of the entire DGP architecture is focused on providing good security easy maintenance for those APIs. DGP APIs are a message-based RPCs which use a single-argument front controller pattern that limits the external attack surface of an entire DGP system to a single method with a single input parameter. The functionality of an API security gateway, backed by a data-driven RBAC identity store is built into a message processing pipeline. All API request messages must pass through the security of the message processing pipeline, with no exceptions (no admin shortcuts or backdoors are possible).

Consolidation (Hub-and-Spoke):

The consolidation of all of the logic in a system into the servers of the middle tier, and then into the web services of a single server, and finally into the same memory space of a single worker thread was all done primarily to improve performance. However, it also helps improve security compared to systems that are built from widely scattered subsystems connected together over some type of public network such as the Internet.

Message-based APIs:

The message-based APIs use a single-argument front controller pattern to limit the web service external attack surface to a single method, which accepts a single input parameter (the API request message). This results in the smallest external attack surface possible. DGP API request messages are not treated as serialized objects, but are instead flexible collections of various types of message elements. As such, they have no static structure and no strong data types. This allows API messages to be read in one pass through a memory stream (firehose cursor) with no serialization/deserialization. This eliminates the possibility of using strong data types or object serialization/deserialization as a basis for attacks.

Message security (message processing pipeline):

The message processing pipeline reads the API message (optionally decrypts the message) and performs the following checks:

- Message encryption (TLS or integrated) protects the content of the message during transport.
- Message TTL to help prevent replay attacks.

- Consecutive failed authentication limit per account to help prevent brute force password attacks.
- Message rate limit per account also helps prevent brute force attacks.
- Messages are not treated as serialized objects which eliminates serialization/deserialization vulnerabilities.

Data-driven RBAC Identity Store:

The zero-trust data-driven RBAC security handles authentication and creates the ACL's which provide fine-grained control of authorization to every version of every API method in a system. This authorization functionality is also the basis for the "feature toggles" used in DGP systems for both service evolution and champion/challenger experimentation.

- Admin UI manages the user accounts, roles and data security groups used to create dynamic access control lists
- User account credentials are used to authenticate each API request message individually
  - Role ACL is used to authorize each internal method call within each API request message individually
  - Data ACL is used to authorize access to data partitions for both read and write API methods
- Service accounts
  - Used for programmatic access to APIs, with credentials stored in an encrypted section of the web.config file
  - Has API method which allows expired password to reset account and return new password (to be stored)
  - Are limited to a single location (are not replicated)
- Standard accounts
  - Password expiration, expired password can only be used for reset until password is reset successfully
  - Replicated between all the locations of a system

The DBSetup utility is used to both build and update the standard database schemas, customized to follow an organization's naming convention, preferred storage locations, etc. This tool is also used to create and update the standard security data in every environment of every location in a system (each environment has its own individual security database in each location).

The data is added to a system as if it had been replicated from a master database. This virtual master database only exists within the code used by the DBSetup utility. This process synchronizes the security data using an idempotent process that prevents the creation of duplicate records, in a way that also causes the synchronized core data to be ignored by DGP's other replication processes.

Application-level Encryption:

The APIs contain logic to encrypt/decrypt sensitive data outside of the database, so it is not visible as clear text to database admins. This logic is implemented within the data access classes/methods called by the API mapper classes. The data encryption also requires key management to be implemented within the configuration data management subsystem.

Integrated Configuration Data Management:

The development tools include a simple mechanism to handle normal configuration data and encrypted configuration data (secret data such as connection strings, service account credentials, etc.) with very good performance. This substitutes for an HSM, and works very well for small systems.

Multi-tenancy:

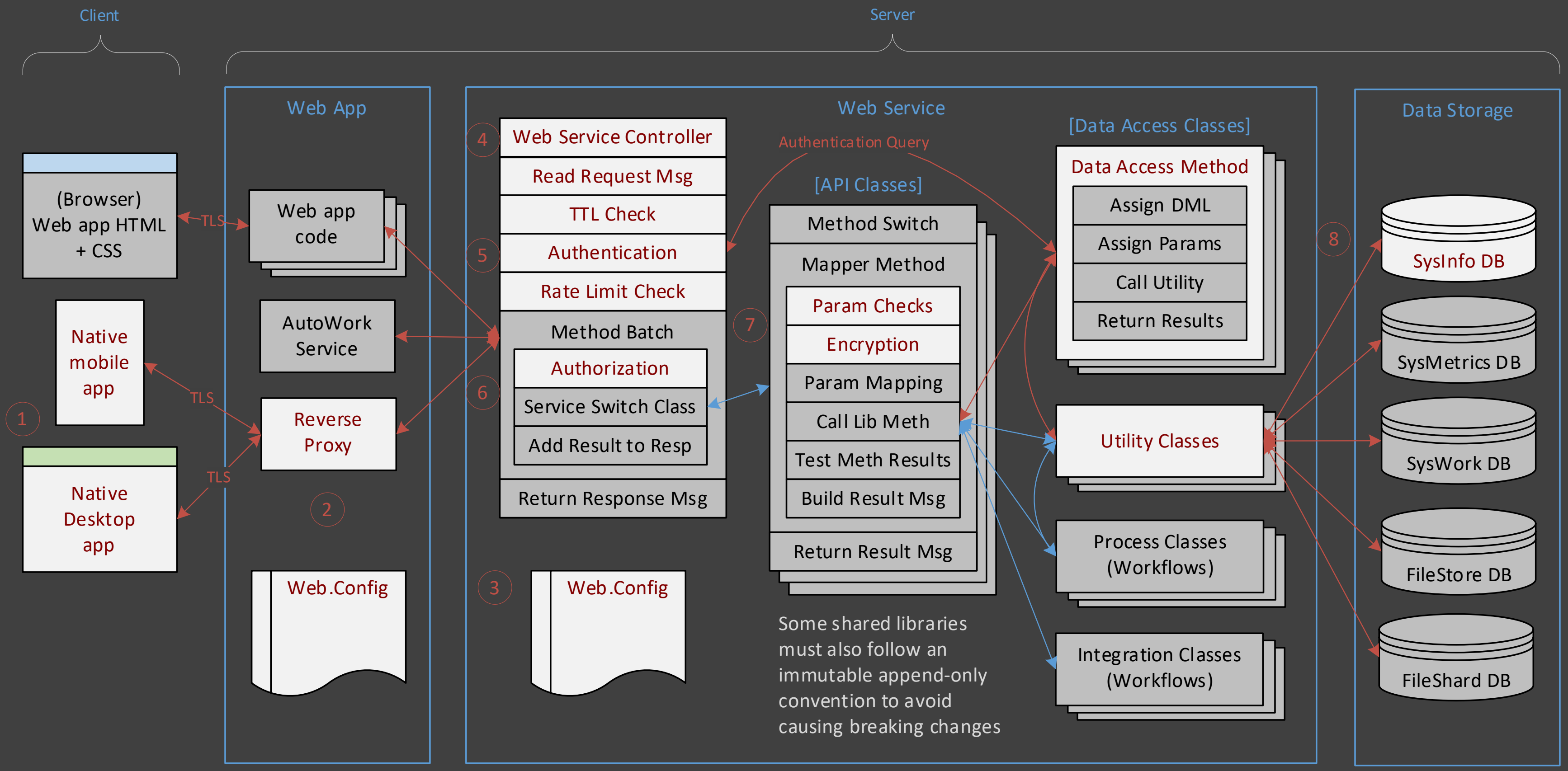
The RBAC manages security group and group membership data. Logical multi-tenancy uses SQL syntax in the data access methods to enforce access restrictions to data (for both reads and writes) based on security group membership. This applies to shared data within a system, such as the files stored by DGPDrive. It does not apply to the data-driven RBAC custom identity store (security system) itself.

Physical multi-tenancy gives each tenant their own separate web services and databases, relying on database engine security to control access to each database. This gives each tenant their own separate copy of the identity store database, etc. Physical multi-tenancy is the preferred choice for its much stronger security.

Data Access:

All DDL and DML SQL syntax is managed in the code of the respective data access classes. All SQL syntax is parameterized. The command syntax itself cannot be modified or affected by user inputs, and as such are immune to SQL-injection attacks. The SQL syntax can be run as dynamic DML or as stored procedures with little modification (dynamic DML by default for SQL Server).

Distributed Grid Platform (DGP): Security Design



1. Optional message encryption uses a PGP-style hybrid cryptosystem as an alternative to TLS for small systems.

2. Reverse Proxy only provides access to web services that are intended to be exposed externally

3. Web.config files are encrypted to protect endpoint URL's, connection strings, system account credentials, etc.

4. Each system will typically have a single web service, which exposes an external attack surface of a single controller method, which accepts a single input parameter (the API request message)

5. The message processing pipeline checks the message TTL, authenticates the user, checks the failed login count, checks the account rate limit, and then processes the batch of method calls

6. Each method call is individually authorized

7. Method input parameters are checked prior to mapping, and some values are also encrypted (zero-knowledge encryption)

8. The ADO.NET connection strings from the encrypted web.config are used by the data access methods to securely connect to the various DGPDrive and DGP databases