Security

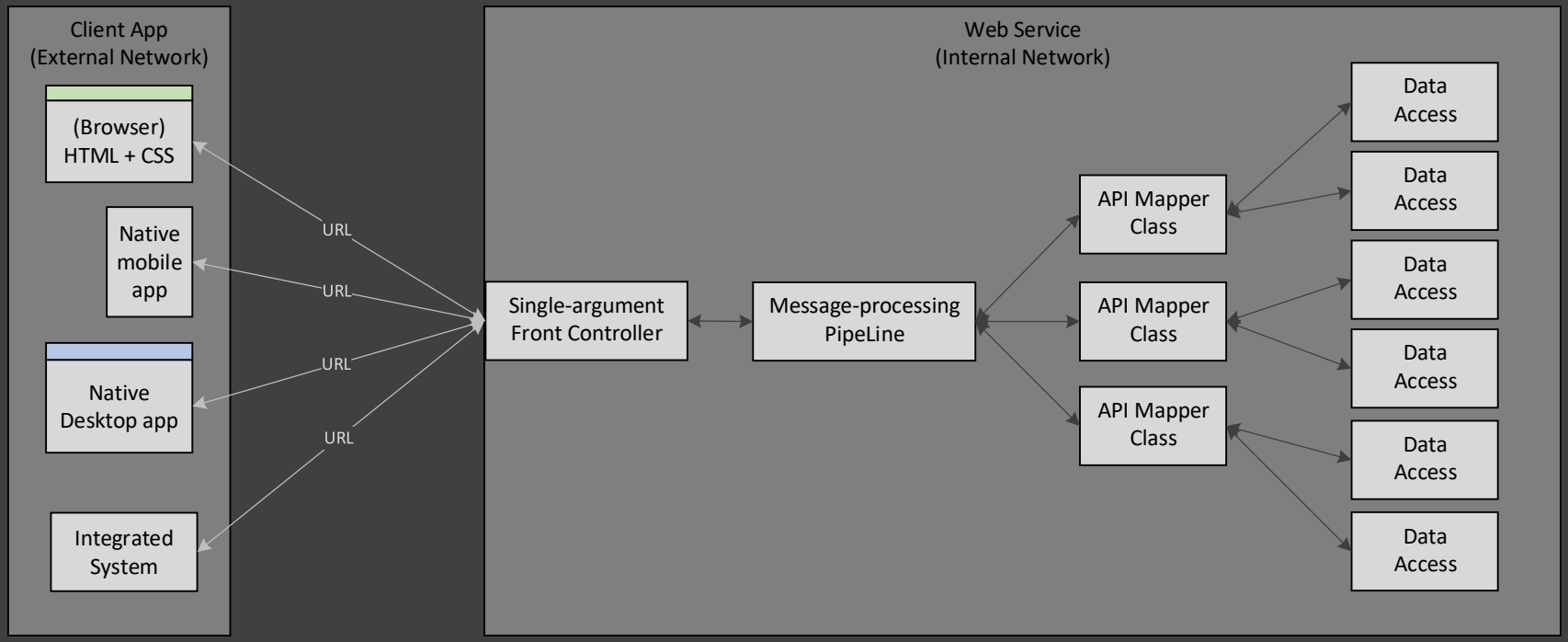
Strong security is necessary to prevent a software system's data from being stolen or damaged, and also to prevent damage to the system itself. The consequences of poor security can result in extensive downtime, a ruined reputation or brand, lost sales, fines and lawsuits. Strong security is also a prerequisite 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 security at a relatively low cost, and without negatively impacting the productivity of the users. For DGP, this is accomplished by designing many reinforcing layers of security into the system's architecture from the ground up.

The overall context for DGP security is that 1) production networks are separate from the business network, 2) production locations have strong network perimeter security, and 3) TLS is used to encrypt traffic to and from each production location. To further strengthen security, DGP uses message-based web service API's and a single-argument front controller pattern 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 lack of strong data types in API messages 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, and account authorization for each individual API method called in the request message batch.

Also strengthening security are the 2 types of encryption used in DGP systems. Zero-knowledge encryption is used in the web services to encrypt data before it is stored in the database, and decrypt data retrieved from the database before it is 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.

These multiple layers of security and encryption have successfully passed three penetration tests run by two different auditing firms conducting PCI-DSS audits, with no vulnerabilities found in those tests. The security capabilities are just as effective at extremely small scales (such as a single computer) as they are in large distributed systems.



Finding zero vulnerabilities in a penetration test is the expected outcome due to DGP's security architecture. 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 64K in size, and it takes 2 GB of data to overflow a text input. The formatted text has no header or metadata and no strong data types, so there are no vulnerabilities in the API request message itself to be exploited. The message is read as a forward scrolling cursor through the HTTP memory stream, so there are no serialization/deserialization vulnerabilities to be exploited. All inputs and returned data are encapsulated in CDATA blocks, so any special characters contained in them are ignored and have no effect on the XML reader.

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 encapsulates 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. The web services use an HMAC hash mechanism to authenticate each API request message sent from the client to the server, and also uses the same mechanism to authenticate the server to the client in the response from the Login method. This works to prevent man-in-the-middle attacks, but is only used for the first method called by client applications (not for all responses from all API methods called).

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 concrete method parameters. Once that step is completed, the internal library method is executed.

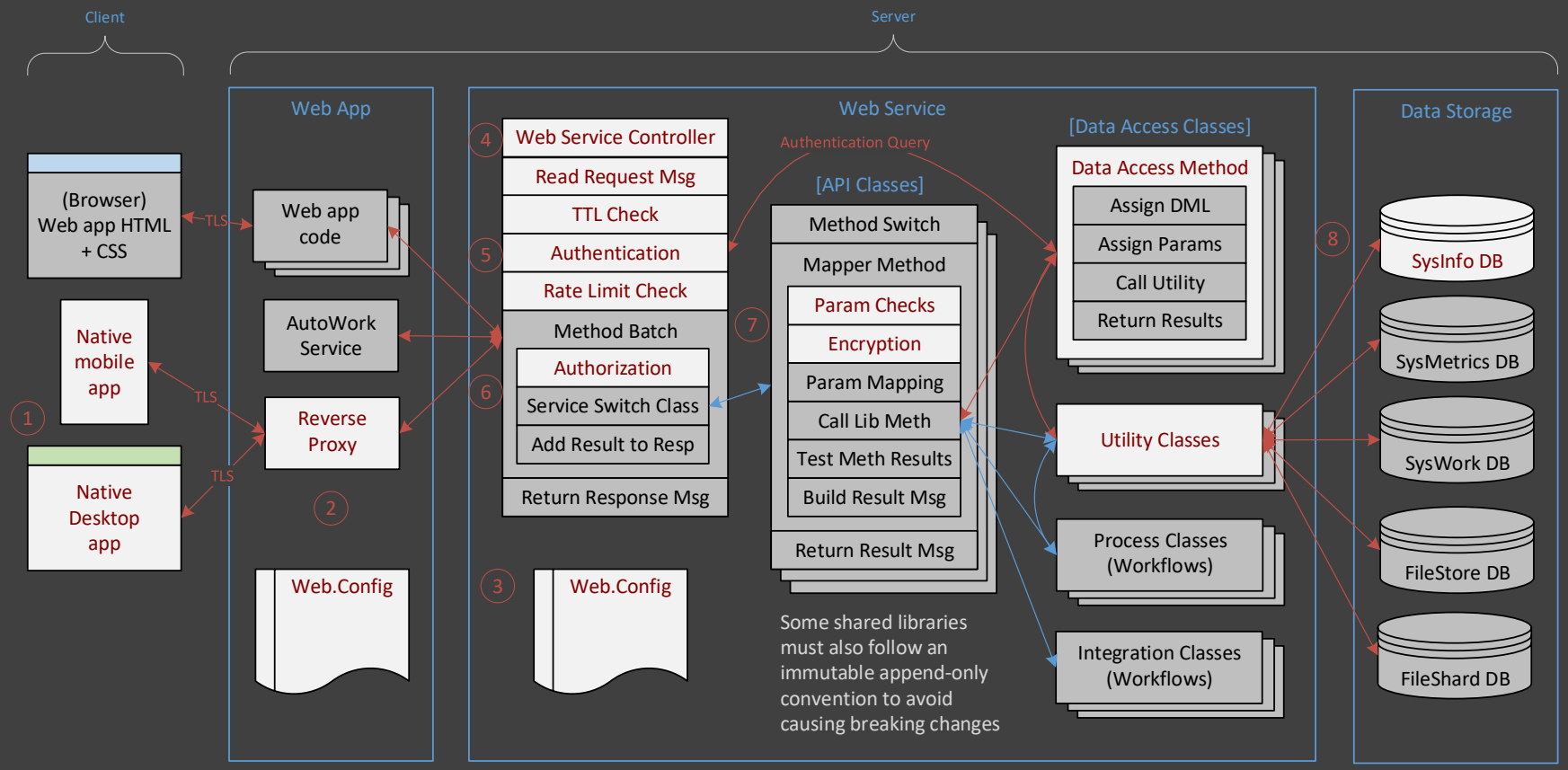
The internal library method will perform any zero-knowledge encryption of data to be stored in the database, as well as decryption of encrypted data retrieved from the database. This is the process used to encrypt and decrypt user account passwords stored in the SysInfo database, for example. These methods also enforce DataGroup authorization and access level in the SQL syntax of the data access (DML) methods, which are currently only used by the FileStore application.

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.

The final DGP security feature is that it 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.

Client applications read the API response messages in the same way that the web services read the API request messages, with no serialization/deserialization vulnerabilities to exploit. They also use the result codes to determine if the API methods worked correctly or encountered an error, rather than using their own logic to evaluate the data returned by the internal method.

In summary, penetration tests finding no vulnerabilities in DGP's web service API's is the expected outcome due to the fact that DGP's security architecture gives attackers nothing to work with.



1. HMAC hash mechanism provides bi-directional authentication by hashing a timestamp that expires with the TTL check (prevents replays)

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 Lattice and DGP databases

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. This eliminates the need for expensive software to scan 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.*

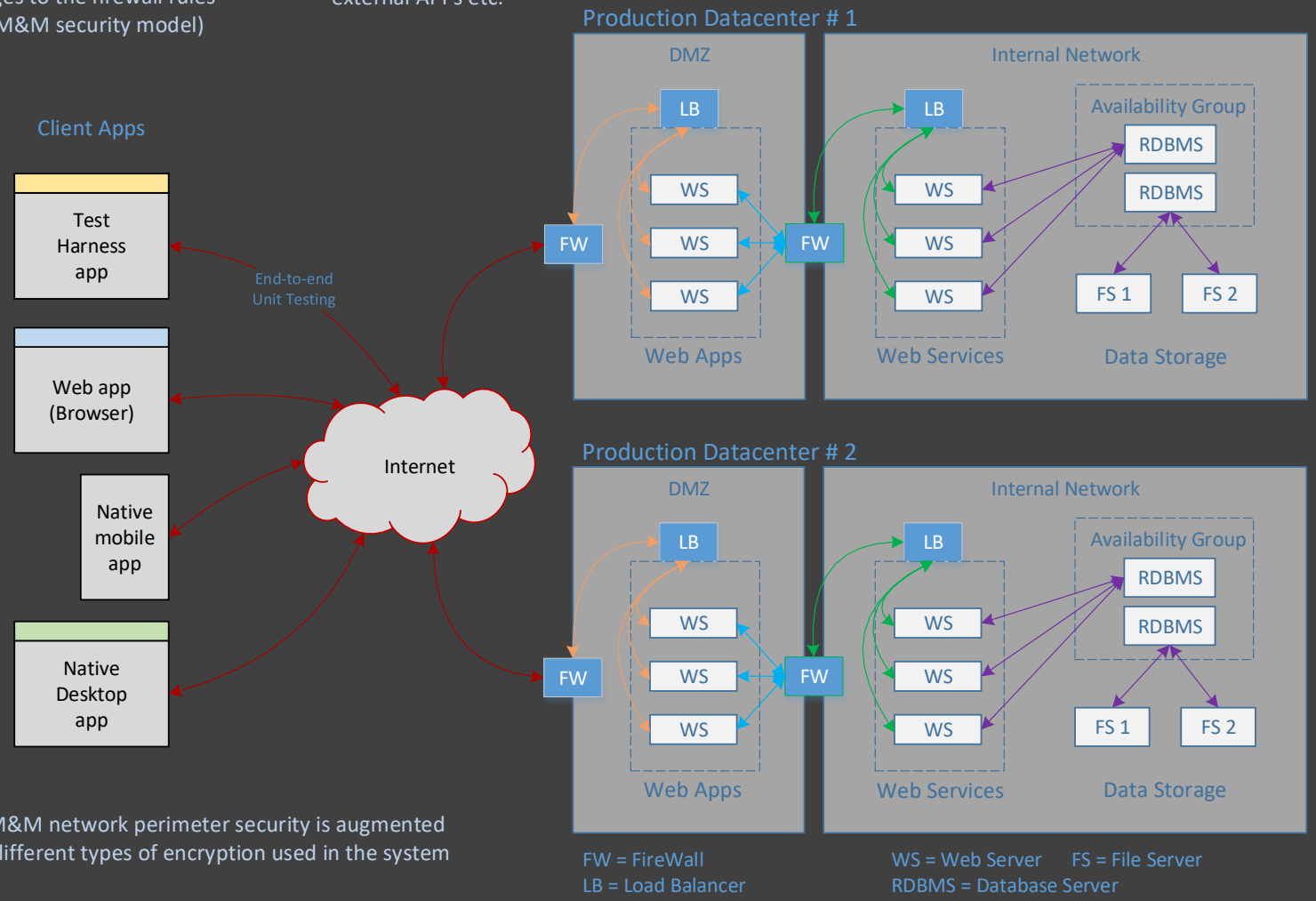
10. *API method authorization can be tested using the API Tester by running a test file for an API that the user is not authorized to call. Authorization data can be viewed in Lattice with the User Methods form, or by using SSMS to view the value of the MethodList field of the APIUser table in the SysInfo 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*

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

The DMZ firewall rules only allows 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 allows traffic from the DMZ server farm to the VIP of the Internal load balancer

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



The M&M network perimeter security is augmented by 3 different types of encryption used in the system

FW = FireWall
LB = Load Balancer

WS = Web Server FS = File Server
RDBMS = Database Server