**Key References:**

* *Next Generation Access Control - Implementation Requirements,*

*Protocols and API Definitions (****NGAC-IRPAD****)*, stored locally as

C:\XpressRules LLC\Business Development\NIST\NGAC\NGAC-IRPAD - rev 0\_70-final-COPY.docx

* *Next Generation Access Control – Functional Architecture (****NGAC-FA****)*, stored locally as

C:\XpressRules LLC\Business Development\NIST\NGAC\INCITS+499-2013\_2\_UNLOCKED.pdf

* Peter Mell, James Shook, Richard Harang, and Serban Gavrila*, Linear Time Algorithms to Restrict Insider Access using Multi-Policy Access Control Systems,* stored locally as

C:\XpressRules\Policy-Machine-master\Documentation\Access Control Algorithms jowua-v8n1-1.pdf

Most important design factor common to both NGAC and XpressRules: autonomic run-time generation, summarized in *Linear Time Algorithms* thus:

“Our visualization appears to be a simple file directory hierarchy, but in reality is *an automatically generated structure* abstracted from the underlying access control graph . . [italics added].”

We concentrate here on the user’s *run-time request for resource access,* specified in §4.4.2 of *NGAC-FA*. I.E. What happens when the User requests access to a Resource? Specifying the data structures at this juncture is essentially defining the API between Request and PDP (aka RMS in the PM).

**Start Point.** Essential definition of code-level run-time workflow by Serban Gavrila, with the module names based on a previous release of PM (as Harmonia):

The RMS is our PDP, also known as PM engine, or PM server. The main classes are in PmEngine.java, SQLPacketHandler.java, CommonSQLDAO.java, etc. **The server receives requests from the clients as commands containing a command code followed by arguments.** Each command is **dispatched to a corresponding method** (the dispatcher is in SQLPacketHandler.java). **The most important command and method, in fact the purpose of the server, is getPermittedOps, which given a session, process, user, and object, finds the operations that the process running on behalf of that user can perform on that object, and returns them to the requester client, which then can decide whether the operation it wants to execute is allowed or not.**

For example, we suggest you to run a most simple application, the RTF editor, and follow the chain of calls to open and read a rtf file. The application calls openObject3() for an object. The call is transformed into a command openObject3 which is sent to the kernel simulator (which has the function of a PEP). In the kernel simulator, the command is dispatched to a function doOpenObject3() which returns a handle to the object. The handle is used then in the application to read the content of the object using the function readObject3(). Siimilarly, this function is transformed into a command readObject3 which is sent to the kernel simulator. In the kernel, the command is dispatched to a function doReadObject3(), which first calls getPermittedOps() of the server. Next, doReadObject3() checks whether the returned set of permitted operations contains the read operation, etc.  More details can be learnd by tracing the function calls and the commands.

Note that the client calls to openObject3 and readObject3 return from the kernel simulator, they don't get to the server. The exception is the call to getPermittedOps issued by doReadObject3 of the kernel that gets to the server in order to extract the operations permitted by the policy. There are a lot of  other function calls that can be issued by an application, reach the kernel and pass-thru to the server. All these functions deal with the policy management, so they need to reach the server, which is the only entity that can read/write/delete/create policy elements and relations in the access database.

Updated comments on the above:

1. First sentence: “The RMS is our **PDP**, also known as PM engine, or PM server.” Takeaway: NIST has given us a PDP for free.
2. Serban is the developer for all of the sources he cites above. The code is available here: <https://github.com/PM-Master/Harmonia-1.6>
3. Not a SINGLE .java file he cited is in the *current* GitHub repo that you and I downloaded: <https://github.com/PM-Master/Policy-Machine> Confirms only that Serban is not cognizant of the current status of the Policy-Machine product. But his explanations are helpful.
4. The logic that corresponds most closely to **getPermittedOps (Harmonia)** isthe **PmAccess** **(Policy-Machine)** class defined in C:\XpressRules\Policy-Machine-master\src\main\java\gov\nist\policyserver\access\**PmAccess.java.** Methods in this class appear to define precisely what the PM expects a query to look like. If that’s true then we can accurately infer the API for a run-time query.
5. The source for **getPermittedOps** is in Harmonia’s PmEngine.java (37K lines in length!)
6. We already have coded and demonstrated this workflow:

**NL🡪XML🡪OWL Ontology🡪Cypher**. We will next “override” that workflow to implement a series of transformations that will get us to the API for run-time queries:

**NL (Query)🡪XML🡪OWL🡪PM Command**.