Design Proposal

HBase Cell-level Security

Contents

Overview 3

Requirements 3

Design proposal 3

High-level Design 3

Lower-level Details 4

I) KeyValue/Cell Changes 4

II) Client API changes 5

III) Server side changes 6

IV) Semantics 9

V) Example 10

Other Design Choices 12

Row level/Meta column family 13

Defining new KeyValue Types for Tags 13

Hierarchy support for application defined groups 13

Performance 13

Future Work 14

Glossary 14

Open Questions 15

References 15

# Overview

HBase allows users to define multi-tenant schemas, where a single table can be shared across multiple users. This requires a way to define access policies for each user, so that user data is secured from unauthorized users. Authorization should be done as per the data model, and not per table/column family only.

Current security functionality in HBase provides authorization at table, column family, or column qualifier level granularities. This works well for use cases that only require global or column-based security, but does not address finer-grained controls at the level of a cell or group of cells. The current implementation also caches all the ACLs defined in the \_acl\_ table, which can become very large when a user starts defining rules at a column qualifier granularity.

Cell-level security means defining and implementing accessing policy for each cell. This document presents a design to achieve cell-level security by attaching visibility tags to cells. A visibility tag is an alphanumeric, user-defined character sequence.

# Requirements

Please refer to the PRD [1].

# Design proposal

This section discusses the design proposal for adding cell- and row-level security in HBase. We generalize row-level security to cell-level as row-level security can be implemented on top of a cell-level implementation. We will discuss more about row-level security specifically in the [Other Design Choices](#_Row_level/Meta_column) section.

The key aspect of the proposed design is to attach a visibility tag expression to each key/value pair (cell) while inserting data into HBase. While reading, a user will provide a visibility tag expression, depending up on his authorization, and the expression will be evaluated with the key/value pair’s visibility tag expression. If the user is not allowed to see a cell, it will be excluded from the result.

## High-level Design

This section provides a high-level overview of the proposed design.  The salient points are:

1. HBase is required to be running on a secure setup (i.e., with secure HDFS and secure Zookeeper configured as mentioned in [3]).
2. HBase security rules will be evaluated at table-level, column family-level, column qualifier-level and cell-level, in descending precedence order.  e.g. table-level rules will be evaluated first, then column family-level rules next, then column qualifier-level rules, next, and finally cell-level rules will be evaluated.
3. In order to define cell-level security, an expression of visibility tags is inserted in each key/value pair. Support of simple Boolean expressions of these tags will be provided.
4. The design provides extensibility mechanism:
   1. In the most-simple implementation, a tag may be mapped to a user group; but the Tag interface will be extensible so that it can be overloaded by users.
   2. The Client APIs pass in generic attributes for potential future items.
5. Versions will be kept for each unique visibility expression.
6. The visibility tags are defined by user applications running on top of HBase. From HBase perspective, they have no semantic meaning; it treats them as simple byte array. These tags represent a one to many mapping for a user. In the simplest design, a user group can also be used as visibility tags. These tags can have different Read/Write acls for different tables, as shown in the [Example](#_V)_Example:).
7. HBase does not support any hierarchy among tags. These tags have Read/Write actions for a table/column family/column qualifier level, as shown in Table 1.
8. Multi-user/group support: There are cases when an application that accesses HBase will have multiple end users, or users are given group level permissions. Visibility tags can be logically thought of as a set of groups assigned to the user. Hadoop provides a pluggable group locating support to fetch the user group, such as from the underlying operating system, or from an LDAP server (using proxy services for remote users [2]). Thus, we can validate the passed set of visibility tags in any Get/Put/Delete/Scan request with the user’s group. The proposed design re-uses the Hadoop user-group mechanism.

## Lower-level Details

This section fleshes out the lower-level details such as proposed changes/additions in the code base and additions in the client-facing APIs to enable cell-level security.

### I) KeyValue/Cell Changes

The basic idea to provide cell-level security is to add an access policy for each cell. An HBase cell has its type, a Put or a Delete, which is in accordance whether it was inserted with the Put or Delete API. To add visibility tags, we define a placeholder for visibility tags (or their encoding, as defined in Table XX). Table 1 presents the existing and proposed KeyValue structure.

|  |  |
| --- | --- |
| Existing KeyValue | \*Row:CF:CQ:Timestamp:KeyValue.Type:Value |
| New KeyValue with visibility tags | \*Row:CF:CQ:Timestamp:KeyValue.Type:**VisibilityTags**:Value |

Table 1. Old and new KeyValue (location of Visibility tag attribute in the backing byte array)

A HFile will have only one type of KeyValue, and the information whether its keyvalues has visibility tags or not is kept in the metadata block of the HFile. Using this information, HFileReader will instantiate the KeyValue objects while reading the data.

Given a KeyValue, there should be a way to know whether it has visibility tags or not (it will be used by Read path/ScanQueryMatcher for handling deletes, and also by the CellVisibilityFilter). This can be done by having an attribute in the KeyValue for visibility tags. But it requires change in core HBase ([Open Question](#_Open_Questions))

These visibility tags are part of the key [[Glossary](#_Glossary)].

### II) Client API changes

1. hbase-site.xml: Set the properties used for current hbase-security (for authentication, authorization, secure rpc);
2. HTable-level property: a property “ENABLE\_CELL\_LEVEL\_SECURITY” in the .tableinfo to indicate whether cell level security is enabled or not. Enabling it will result in writing KeyValues with visibility expression.
   1. The above property can be made at column family level too.
3. CellACLExpression: A Wrapper class on the visibility Expression; provides methods to get the boolean expression value, and conversion from byte to char[], etc.
4. HTable: Following are the tentative changes in the client side API to incorporate the cell level security.
   1. The visibility tags are made to be part of the KeyValue. So, the Mutation classes, Put, Delete, Increment, Append will add methods to set the visibility tags in the Mutation object.
   2. Put->add(byte[] family, byte[] qualifier, long ts, byte[] value, CellACLExpression expression) // will add a KeyValue with the given expression;
   3. Put-setCellACLExpression(CellACLExpression expression) // will update acl expressions for all the KeyValues;
   4. Delete->deleteFamily(byte [] family, CellACLExpression expression)
   5. Delete->deleteFamily(byte [] family, long ts, CellACLExpression expression)
   6. And, similar methods to set the visibility expression along with defining deletes.

-> In case the “ENABLE\_CELL\_LEVEL\_SECURITY” properties is false or security is not enabled, using methods to set the CellACLExpression will give OperationNotSupported Exception.

-> In case one uses the Put/Delete/Append mutation object without adding a CellACLExpression and “ENABLE\_CELL\_LEVEL\_SECURITY” is set to true, a default expression is added (an empty value). It will be done transparent to the client applications.

1. HBase Shell:
   1. It will be used for entering/updating visibility tags for users:A super user can insert visibility tags for user in the \_acl\_ table using existing commands such as Put, which will eventually invoke the HTable->Put() API.
   2. It will used to enable/disable cell level security on a table level. This means setting ENABLE\_CELL\_LEVEL\_SECURITY to true/false from the shell.

The property “ENABLE\_CELL\_LEVEL\_SECURITY” can be change only after enabling/disabling the table.

### III) Server side changes

This section discusses server side changes in order to enable cell-level security.

These includes adding a global switch to enable cell-level security on the cluster level, a system level Coprocessor, and a Filter (apart from code changes to handle reads such as Delete, DeleteFamily, etc):

1. Add the CellVisibilityAccessController (described below) to the System level coprocessors. In the preXXX hooks, it will check for the table/cf/cq level perms. If the Cell level support is provided at the table level (ENABLE\_CELL\_LEVEL\_SECURITY), it will authorize at cell level.
2. CellVisibilityAccessController (CVAC): a system level coprocessor used for checking the visibility permissions for a user.
   1. Write-path: For a given mutation action, it will check permissions for table/column-family/column qualifier level in the preXXX method. If the table also supports cell-level security, it will authorize the passed visibility-tags (in the same order) by referring to the \_acl\_ table. If the mutation doesn’t have visibility tags, it will insert a default value. This is to ensure that an HFile has uniform keyValues.
   2. MemStore-Flush:  In the preFlush hooks, it will append the information in the metadata block that this HFile contains KeyValue-with-visibility-tags.
   3. Compaction: In the preCompaction hook, it will check the table attribute ENABLE\_CELL\_LEVEL\_SECURITY, write KeyValues with visibility tags in case this is true, or without visibility tags otherwise.
   4. Read-path: The read path behavior is fail-fast. First, it will do the authorization of the table, column family, and column qualifier level read permissions. If the user is not allowed to read the table, or column family, and any of the passed column qualifier, he will get an AccessDeniedException. In case he has table/cf/cq perms, the control will move forward to look for cell level visibility tags and the passed visibility tag expression is validated against the allowed ones (by referring to the readableVisibilityTags for the table). If it fails, an empty result set is returned, else CellVisibilityFilter with the validated visibility tags is attached to the Scan/Get object.

Deletes are to be handled differently, as semantically, Delete will effectively deleting all accessible cells, based on the passed visibility expression.

We have different levels of Delete:

1. DeleteRow: Deletes all the row
2. DeleteColumnFamily: Deletes a specific column family
3. DeleteColumn: Deletes a specific column

All these above Delete APIs also take an optional timestamp argument, which does the delete before this timestamp. The semantic idea of Delete with visibility expressions is perform the above deletes for all the cells which are accessible with the given visibility expressions. Going by brute force approach, it means doing a Get for the provided arguments and then deleting all of them with the given visibility expression.

The accessibility of cells is checked during the read path before applying filters, i.e. at ScanQueryMatcher class level.

1. CellVisibilityFilter: This is a filter used by CVAC in the read path. It is attached to the passed Scan/Get object in the preXXX method. It will use column visibility tags as defined in the Scan/Get object by the user, and filter out the cells which does not have the provided visibilities. The selected keyvalues are bubbled up the priority heap through the current mechanism. But it will involve some effort in the Delete processing as mentioned above, most importantly in *ScanQueryMatcher* and *ScanDeleteTracker* class.
2. \_acl\_ table: The existing security functionality uses this table to store ACLs for users for tables/column-family and column-qualifiers. In the proposed design, it will also be used to store visibility tags for a user. Only a superuser can enter entries for a user.

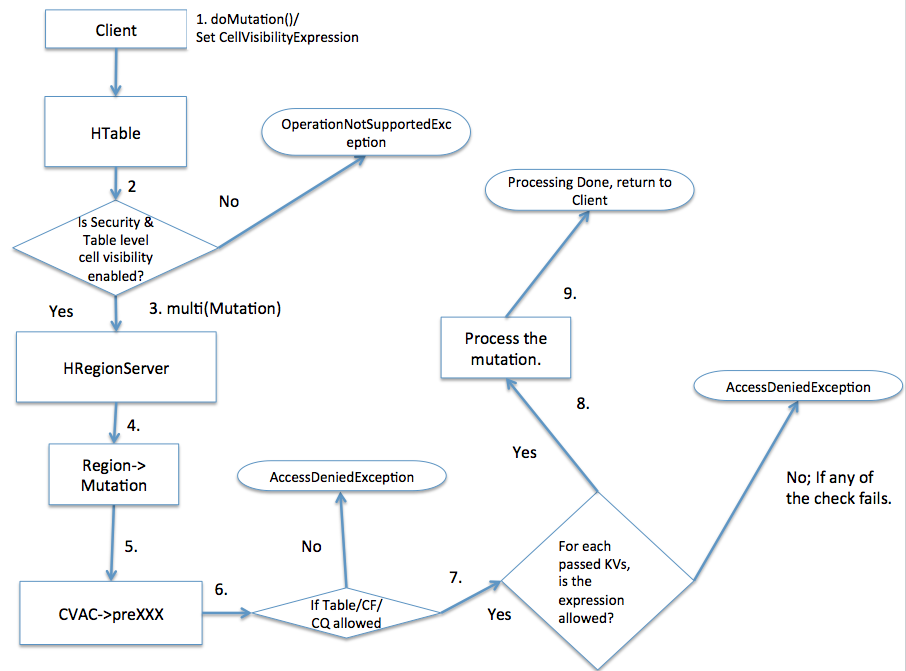


Fig.2 Flowchart for mutation flow

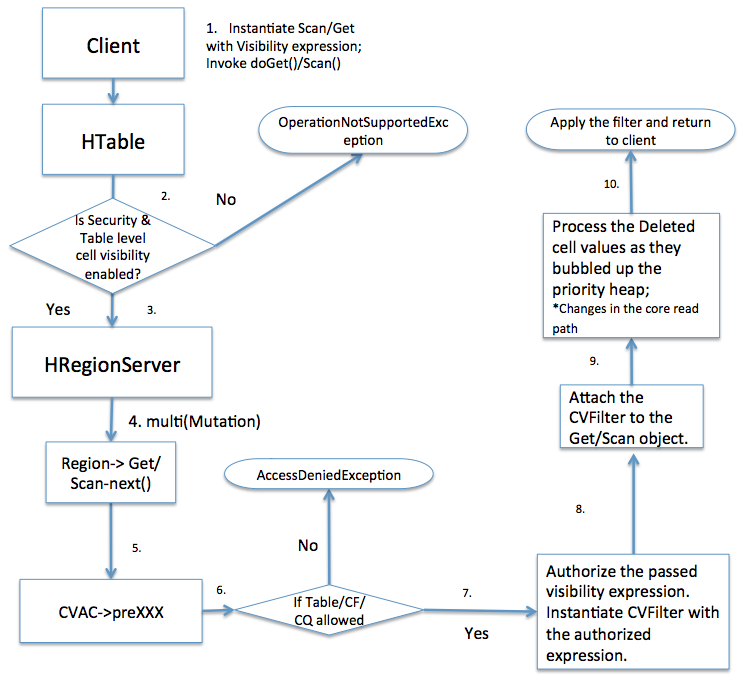


Fig.3 Flowchart for tentative Read (Get/Scan) Path;  

### IV) Semantics

1. If a user issues a delete command, the cells which are accessible by the given expression will be deleted.
2. CellVisibilityFilter will pass through (takes it in the result) if the passed keyType does not has visibility expression as it represents old keyvalue or default types.
3. If cell level security is disabled after some Mutations have been done, those mutations will be visible to all. The visibility expressions will be present but will be ignored.
4. There will be scenarios when a region has different HFiles with different types of KeyValues. The metadata in the metablock of the HFile will enable us to read the KeyValues properly from these files. If cell-level security is turned off, then the visibility expression will be ignored while reading. Similarly, current value of ENABLE\_CELL\_LEVEL\_SECURITY is checked during compaction. If it is true, then the new HFile will have KeyValues with visibility tags. In case the cell type not tagged (existing type), it will treat it as a cell with default visibility tag (i.e., readable by all). This is necessary to read existing data in the cluster.

### V) Example

**Overview**:

We have a table “*Ingest*”, which has a column family *CF\_Metrics*, in which a user *Flume* ingests some metrics of an application. There is a user *Hive*, which performs computation by reading the data from the CF\_Metrics column family. He can access data marked with Secret Tag, and not with Top\_Secret tag. *Flume* can write with any tag/expression having Secret or Top\_Secret tags.  If *Flume* writes some data with Top\_Secret visibility tag, it is not accessible to *Hive*.

Group/Tags Info (the group information is fetched from Hadoop group-locating service, as mentioned [in High-Level-Design section](#_High-level_Design)):

*Flume: Secret, Top\_Secret*

*Hive:Secret*

**Explanations:**

We use the \_acl\_ table to describe permissions at table, column family and column qualifier level for  users; we also use it for defining visibility tags Read/Write ACLs. Here is the sample ACLs for the above use case\*:

* The primary key of the \_acl\_ table is the table name: in this case, *Ingest*.
* The first two rows are column family level permission of users *Flume* and *Hive*. Flume can read/write the column family, whereas Hive has only read permission.
* The third row defines visibility Tags (vTag) permission for TopSecret tag: this Tag can be used for Read/Write on the *Ingest* table. Similarly, the fourth row defines vTag permission for Secret tag. Note, both the Secret and Top\_Secret are groups.

So, if *Flume* inserts data with (*Top\_Secret* ***|*** *Secret)* visibility expression, *Hive* can read it with its *Secret* tag. If *Flume* inserts data with *Top\_Secret* tag, *Hive* can’t access it because he doesn’t have *Top\_Secret* tag*.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Primary-key** | **CF** | **CQ** | **Value** |
| Ingest | I | user:Flume,CF\_Metrics | R|W |
| Ingest | l | user:Hive,CF\_Metrics | R |
| Ingest | l | vTag:TopSecret | R|W |
| Ingest | l | vTag:Secret | R|W |

Table 2. Sample snippet of the \_acl\_ table\*

**\***The above table is a layout of \_acl\_ table; basically it represents a keyvalue where the Primary-key is a table name, CF is column family (default column family “l”), CQ is Column qualifier, and Value is the value of the cell.  
  
**Sequence of events:**

1. Flume does some inserts on table Ingest:

* r1:CF\_Metrics:cq1:Put:Top\_Secret:v1
* r1:CF\_Metrics:cq1:Put:Top\_Secret|Secret:v2
* r1:CF\_Metrics:cq3:Put:Secret:v3
* r1:CF\_Metrics:cq1:Put:Secret|Top\_Secret:v4

1. Hive does a get on row “r1” with the following Get object:

* Get(r1:CF\_Metrics:cq1)

1. Hive has only one group, Secret. The corresponding tag for this group has read/write permission on Ingest:CF\_Metrics column family. He will get cells with value v2 and v4 (but not v1, as it is written with visibility tags Top\_Secret).

With the above scheme of storing the acls of visibility tags, one can define Read or Write permissions on a table/Column family basis.

**Write path:**

*Flume*:

HTable ingest = new HTable(“ingest”, conf);

Put p = new Put(“r1”);

p.add(“CF\_Metrics”,”cq1”,”Top\_Secret|Secret”,”v2”);

// inserts r1:CF\_Metrics:cq1:Put:Top\_Secret|Secret:v2

// inserts r1:CF\_Metrics:cq3:Put:Secret:v3

p.add(“CF\_Metrics”,”cq1”,”Secret”,”v3”);

ingest.put();

ingest.flush();

….

HBase:

    // Verification steps:  
a) AccessController coprocessor checks whether Flume has write access on the “ingest” table and “CF\_Metrics” column family or not.   
b) CVAC fetches the allowed tags for the user: referring to Hadoop’s group service: UserGroupInformation.getGroupNames(); there are entries for these groups in the \_acl\_ table as vTags:<group\_name>.   
It sees the writableTags for the table as: Top\_Secret,Secret. Flume has groups Top\_Secret and Secret; so his expression evaluates to true.  
//inserts the records;  
  
**Read path:**

I) Wrong visibility tags:  
Hive:  
    HTable ingest = new HTable(“ingest”, conf);  
    Get g1 = new Get(“r1”);  
    g.add(“CF\_Metrics”,”cq1”,”Top\_Secret);  
      
    ingest.get(g1); //  
HBase:   
    // Verification steps:  
a) AccessController processor checks whether Hive has read access on the “Ingest” table and “CF\_Metrics” column family or not.  
b) CVAC fetches the allowed tag for Hive: it locates Secret group, which is in the readableTags for the table.  
It verifies the expression and sees that he can’t use Top\_Secret, and returns empty result.  
  
  
II) Correct visibility tags:  
Hive:  
    HTable ingest = new HTable(“ingest”, conf);  
    Get g1 = new Get(“r1”);  
    g.add(“CF\_Metrics”,”cq1”,”Secret);  
      
    ingest.get(g1); //  
HBase:   
    // Verification steps:  
a) AccessController processor checks whether Hive has read access on the “Ingest” table and “CF\_Metrics” column family or not.  
b) CVAC fetches the allowed tag for Hive: it locates Secret group, which is in the readableTags for the table.  
It verifies the expression and sees that he can use Secret.  
It attaches the CVFilter to the Get object, and when HBase (core read path) returns a set of KeyValues, it passes only the ones which pass the “Secret” visibility expression.He will get v1 and v4.

# Other Design Choices

## Row level/Meta column family

This alternative is along the idea of adding a meta-column for a table. This meta column will have a security policy for each row (who can access it with what perms, etc). Adding a meta column for a row will not suffice for cell level security where we would like to provide *different* level of access to cells (which have same row, column family and column qualifier). Since HBase doesn’t have the support of locality groups across multiple column families, this approach might also suffer from higher read latency, as every read will entail read from two families.

## Defining new KeyValue Types for Tags

This entails defining client side classes for Puts and Deletes, such as TaggedPut, TaggedDelete, TaggedDeleteColumn, and TaggedDeleteFamily.  In this design, we need to create classes for every taggedType, and adding more Types in the KeyValue would have been more difficult because of the limited bitspace for KeyValue types.

## Hierarchy support for application defined groups

A line of thought was to let user application define their grouping scheme (similar to unix like group scheme, or Oracle level support), where it gives uid/gid to various users/groups, and defines a hierarchy among groups/levels: CEO group > HR > Engineering (permissions order). If a cell is accessible to HR, it is implicitly accessible to CEO. But letting user application to define the group relation is a possible security loophole (security shouldn’t be governed by HBase client application; PRD doc Goals#2 in P1 section), and also a maintenance nightmare as we add more groups/rules.

# Performance

This section lists out the areas where we might have some performance penalties in the areas of  memory and CPU utilization:

1. Visibility tags are stored in the keyvalue; this bloats its size and so the memory footprint of the application increases. We can reduce the cost by introducing one level of indirection by encoding the visibility expressions and using the id rather than the entire expression. But then, we will want to avoid any extra DB trip to look for the encodings, so use some cache in the Regionserver. On any changes in the encoding table we need to evacuate/or refresh the cache. This scheme can ease out the effort in case we need to update the visibility tags for cell in future; just change the mapping and no need to rewrite all the cells.
2. Regionserver Cache:
3. We plan to use cache for storing the acl (both the table/cf/cq level and visibility tag level) to reduce the table lookup for read/write ops. This will be a LRU cache.
4. The user name and group information is already cached in the Hadoop mechanism.
5. The write path involves additional checks on all the passed KeyValue in a single mutation. One can cache the local validation result for the keyvalues and can reuse them for similar expression. Handling of various Delete APIs will incur some additional cost as it involves treating the visibility tags as part of the row-key.
6. The read path involves change in the current handling of Delete cells. It needs to be explored.
7. Simple vs complex Boolean expressions for visibility tags: Complex boolean expression will be costlier, from validation to execution.

# Future Work

1. Consider the option for introducing column family level cell level security rather than table level. Food for thought: A Put can span across multi column families. We may define it at CF level after we figure out the structure of Client side APIs?

# Glossary

1. Cell: A cell is the atomic unit of storage in HBase. It is aware of its dimension: row, Column family, column qualifier, timestamp, Type, and value. The row mentioned above is the user-defined “rowkey” what he uses to insert/access the data.
2. VisibilityTag: is a character sequence, consisting of alphanumeric characters, with simple Boolean expressions as & and |, and parentheses to form these expressions. For example: sample valid visibilityTag are A&B, (A&B)|C. An invalid visibilityTag is A&B|C.
3. key: is the “non-value” part of the Cell. So, as from Table 1, it spans across “primary-key:cf:cq:timestamp:Type:VisibilityTags”.
4. User: the process owner of the application that is accessing HBase. HBase can access the remote user using the Hadoop user-group mechanism.
5. Group: A user can belongs to multi group, be it unix group like groups, or based on functionality. For example, an employee can belong to multi groups such as employee, engineer, geek groups. HBase can access the remote user’s group using the Hadoop user-group mechanism. In the primary phase, we overloaded the group concept with the visibility tags, i.e., group can also be used as a visibility tag.

# Open Questions

1. Visibility tags are stored in the keyvalue.
2. Mutation classes such as Put, Delete, Increment, Append should add methods to set the visibility tags in the Mutation object; this is a core HBase change, even though cell level security may not be a core HBase feature. Shall KeyValue be made an interface?
3. It bloats the KeyValue size and increases the memory footprint of the application. What are the ramifications on performance?
4. Delete Semantics: As mentioned in CVAC section of [Server-Side-changes,](#_III)_Server_side) handling of delete section presents an interesting use case. Shall we delete exact match expression, or push a delete to all the cells that are accessible. For example, in the below sequence:

    row1:cf1:cq1:Put:Top\_Secret|Secret:v1

    row1:cf1:cq1:Put:Secret:v2

    row1:cf1:cq1:Delete:Secret

    Will “v1” should be deleted or not?

1. The current behavior is that a user is not allowed a tag to read/write on which he does not have read/write permissions. Should there be an option for admin so that he can allow a user to Write to a table cell with visibility expression which he does not have write visibility?

# References

[1] PRD: Attached to the jira

[2] http://hadoop.apache.org/docs/r1.0.3/Secure\_Impersonation.html

[3] https://ccp.cloudera.com/display/CDH4DOC/HBase+Security+Configuration