

**Security Manual**

**The Black Guide**

© 2006-2013 Codice Software

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Codice Software cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

**Table of contents**

[1 Motivation 2](#_Toc263520897)

[2 Secured objects 3](#_Toc263520898)

[2.1 ACL structure 5](#_Toc263520899)

[2.2 Available permissions 5](#_Toc263520900)

[2.3 Permission calculation 7](#_Toc263520901)

[3 Users and groups 8](#_Toc263520902)

[3.1 Special users 9](#_Toc263520903)

[3.1.1 ALL USERS 10](#_Toc263520904)

[3.1.2 OWNER 10](#_Toc263520905)

[4 Permissions on the GUI tool 11](#_Toc263520906)

[4.1 Setting basic permissions 11](#_Toc263520907)

[4.2 Advanced permission information 12](#_Toc263520908)

[4.3 Break inheritance 13](#_Toc263520909)

[4.4 Reapply inheritance 13](#_Toc263520910)

[4.5 Extend inheritance 14](#_Toc263520911)

[5 Setting security from the command line client 16](#_Toc263520912)

[5.1 Displaying permissions 16](#_Toc263520913)

[5.2 Setting permissions 18](#_Toc263520914)

[5.3 Manipulating inheritance 19](#_Toc263520915)

[6 Securing a Plastic SCM system 21](#_Toc263520916)

[7 Case studies 22](#_Toc263520917)

[7.1 Case 1: Restrict integration branches 22](#_Toc263520918)

[7.2 Case 2: Restricted access to source directories 22](#_Toc263520919)

[7.3 Case 3: Restricted access to branches 23](#_Toc263520920)

[7.4 Case 4: Guest consultants 23](#_Toc263520921)

[7.5 Case 5: Set a repository as readonly for a given set of users 23](#_Toc263520922)

[7.6 Case 6: Only integrators can check in to the main branch 24](#_Toc263520923)

[7.7 Case 7: Only integrators can modify the main and the maintenance branches 24](#_Toc263520924)

[7.8 Case 8: Only the owner of a check out can undo the check out 24](#_Toc263520925)

[7.9 Case 9: Open a repository to a given group 24](#_Toc263520926)

**Figures**

[Figure 1. Permission inheritance hierarchy on repositories 4](#_Toc263520927)

[Figure 2. Permission inheritance hierarchy on workspaces 5](#_Toc263520928)

[Figure 3. List of available permissions 5](#_Toc263520929)

[Figure 4. Permission calculation simple 7](#_Toc263520930)

[Figure 5. Security: user in two groups. Permissions on branch. 9](#_Toc263520931)

[Figure 6. Security sample: denied permissions on groups 9](#_Toc263520932)

[Figure 7. Permissions dialog 12](#_Toc263520933)

[Figure 8. Permissions dialog advanced options 13](#_Toc263520934)

[Figure 9. Sample directory structure before extending inheritance 14](#_Toc263520935)

[Figure 10. Sample directory structure after extending inheritance 14](#_Toc263520936)

[Figure 11. Full sample showing how an ACL with inheritance is removed 15](#_Toc263520937)

[Figure 12. Command to display the permissions of an item 16](#_Toc263520938)

[Figure 13. Extended ACL information 17](#_Toc263520939)

[Figure 14. Command line tool displaying permissions of a given revision 17](#_Toc263520940)

[Figure 15. Detailed explanation of a revision's permissions 17](#_Toc263520941)

[Figure 16. Explanation of the *acl* command syntax 18](#_Toc263520942)

[Figure 17. Denying check out permission on a branch 18](#_Toc263520943)

[Figure 18. Extended information 19](#_Toc263520944)

[Figure 19. Cut inheritance copying simple 19](#_Toc263520945)

[Figure 20. Add inheritance scenario 20](#_Toc263520946)

**About this guide**

**Audience**

This guide is targeted to the average developer, assuming familiarity with Plastic SCM and operating system security concepts. It is recommended to read first the User Guide, then this paper.

**Online documentation**

Besides this document and the rest of the guides, Plastic SCM provides online reference throughout its different client frontends.

On the command line interface, both Windows and Linux, this reference can be obtained with the command:

cm help

For extended information on a specific command, type:

cm help *command*

The graphical interface provides online reference through the Help menu.

**Documentation errors**

If you find any problem in this guide or any other part of the online reference, please report it using the following email address:

[support@codicesoftware.com](mailto:support@codicesoftware.com)

# Motivation

The main motivations behind the Plastic SCM security system are:

* Provide a mechanism to control access to repositories and restrict certain operations. A big number of companies require setting different permissions to different projects, users and groups, being able to efficiently restrict the access and prevent sensitive data to be read or modified by unauthorized personnel.
* Enforce policies both for development and deployment. Even on widely open organizations the access to certain parts of a repository can be restricted, not only for security related reasons but to prevent mistakes from happening. This way many development groups restrict the access to the main development line (the *br:/main* branch in Plastic parlance) so that only *integrators* can make modifications to it. The reason is not a lack of trust on the rest of the team, but preventing unwanted errors. A big amount of similar scenarios can be detected on different companies. Plastic SCM security system will be a big aid solving these configuration issues.

Following the whole system’s design goals, the security system introduces a flexible, powerful and yet simple mechanism capable of supporting a wide set of different security scenarios. Plastic SCM eases setting up complex security policies, and does not require writing special scripts or triggers.

# Closing the door

The first step to secure your Plastic SCM server is to close the open doors.

By default Plastic SCM gets installed with all the open doors.

EXPLAIN that only recognized users can be logged in by default (UNKNOWN\_USER otherwise). But they can do almost everything.

Remove the “ALL USERS” from the top.

Set up an administrator.

# Security scenarios

## How to set up an administrator user

The admin is the owner of the repserver.

## Avoid accidentally deleting repositories

Deny or simply “not grant” the “rmrep” permission.

## Avoid some items to be changed

Introduce the new path based permissions. Deny “ci”.

(Deny CO would be great, but otherwise we’d need “shelving/unshelving”).

Same scenario but on branches, requested by powerPlan:

1. *I wanted to make sure that people will not accidentally check out objects on the main (production) branch and the Dev (integration) branch. So I just went to each of those branches, picked ALL USERS and checked Denied for the “co” operation. I tried this with and without breaking inheritance. However, if I try to check out an object on those branches, Plastic will still let me. Am I doing something wrong.*

## Child branch permissions

Is it worth to make permissions on child branches inherit from their parents?

*Another thing I was curious about was this. It looks like we will end up with production patch branches that will hang off the main production branch. In other words, if our code on the production branch is labeled v1.1.1, we will have a hanging branch off that label where each changeset will correspond to something like 1.1.1.1, then 1.1.1.2, then 1.1.1.3 and so on. My question is when those Patch branches are created, is there a way for them to automatically inherit the permissions from our production branch, so that admin does not get involved for changing permissions every time such a branch is created.*

## Lock certain branches except for some users

Lock main branches except for some users.

## Isolate a group of permissions for some items.

## ALL revs in branch permission. (This will be implemented by the path permissions).

## Specific pattern of permissions for some branches that are child of an specific branch.

## Explain that inheritance doesn’t need to be broken so often

***Note****: we detect that they're breaking inheritance more often than really needed because they're not aware of the following: you've a set of permissions set (not all) for a user or group a the repserver level. Then you go to a branch and you can't click on the permission because you need to create you own acl entry...*

## Avoid some items to be changed on certain branches only

*That's not quite how I understand the case. He doesn't want to prevent anyone from working on the files, nor does he want to prevent them from checking into their task branch. He wants to make sure that "newer" versions of these files are not checked into the main branch.*

*/main/pre-integration /main/pre-integration/james-task1001 /main/pre-integration/pablo-task1002 We both may be working on files in our respective task branch which has a common parent branch. I checkin and merge from james-task1001 to pre-integration. There may be 100 (and these may change) of 1000+ file that I don't want touched. Ideally, I want to lock these files. I can do that in CC, I may put an attribute on them of label and either lock the attribute or label (or use with permissions). But the key is that these files can be changed (checkout-checkin) on either james-task1001 or pablo-task1002, but no one can merge or checkin to pre-integration for these specific 100 files. So I haven't stopped parallel development, I've only prevented changes to the pre-integration branch, but on a subset of files. As long as those files are not affected, people can merge and checkin. Make sense?*

# Secured objects

The following objects are secured under Plastic SCM control and have an associated ACL. Check Figure 1 and Figure 2 for detailed information.

* **Server**: the server has an associated ACL. By default, all objects inside a given server will inherit their permissions from the server.
* **Repository**: a system can have multiple repositories. The repository inherits its permissions from the repository server. Each repository has its own associated ACL. By default all objects inside a given repository will inherit its permissions from it.
* **Item**: items inherit their permissions in a file system like way. The root item on a repository will inherit its permissions from the repository, but its files and directories will inherit from it. When a file is created it inherits its permissions from its parent directory. Permission inheritance is not recalculated after a move operation is performed to prevent possible security issues and parallel inconsistencies (a given item can be in different directories at the same time, on different branches). At any moment users can *extend* the permissions of a given directory to all its contained files and directories recursively, reestablishing a file system like permission inheritance model.
* **Branch**: all the branches inside a given repository will inherit their permissions from it. By default branch permissions will affect all its contained revisions.
* **Label**: labels inherit their permissions from the repository.
* **Attribute**: attributes inherit their permissions from the repository.
* **Action**: actions inherit their permissions from the repository.
* **Link**: links inherit their permissions from the repository.
* **All revisions in branch**: all the revisions of a given item on a branch will share, by default, its permissions. It is possible to apply permissions to all the revisions of an item at once by modifying this object’s settings. The *all revs in branch* object will inherit both from the item and the branch. By default it is the only object type inheriting from more than one source. Permissions are calculated in the following way: to be denied it will be enough if the branch or the item has it denied, to be allowed it must be allowed on both.
* **Revision**: a revision will inherit from the *all revisions in branch* object, and hence from both the item and the branch. By default a revision doesn’t have its own ACL, but takes the one created by the *all revisions in branch* object. When the specific permissions of a given revision are modified a new ACL will be created and assigned to the object.

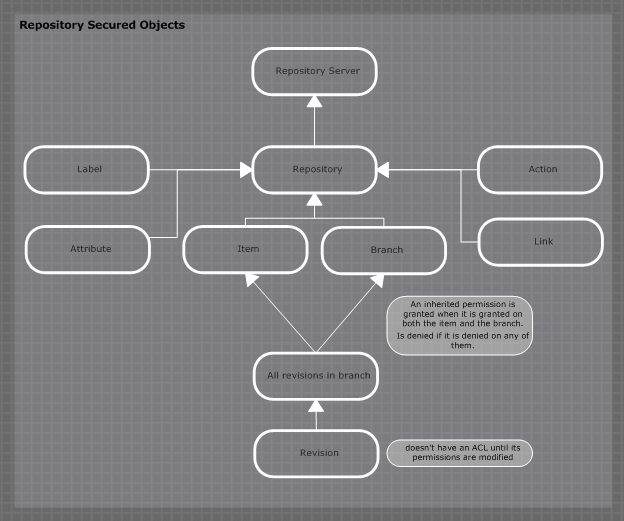


Figure 1. Permission inheritance hierarchy on repositories

## ACL structure

An ACL consist on the following:

* Entries: each user (SEcured IDentifier or SEID in Plastic terms) or group who has granted or denied permissions will have an entry on a given ACL. Each entry will have *allowed* and *denied* permissions.
* Inheritance entries: pointers to ACLs inherited by the current ACL.

## Available permissions

Figure 3 shows a list of all the available permissions on Plastic SCM and how they are applied to different objects.

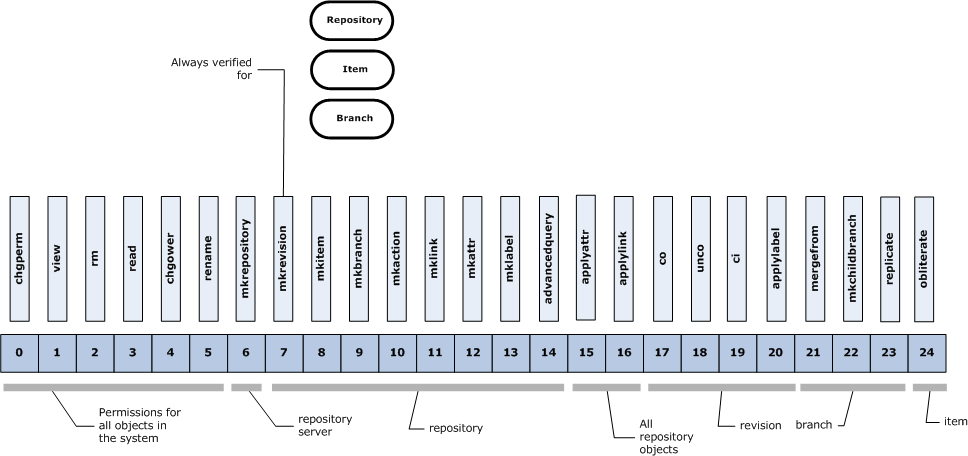


Figure 3. List of available permissions

Table 1 explains all the permissions in detail.

|  |  |  |
| --- | --- | --- |
| Applies to | Name | Description |
| All | chgperm | Allows or denies changing the permissions to a given object. Set by default. |
| view | Permission to view a given object. For instance viewing a branch is the ability to retrieve it on a list but reading is actually being able to access its data. For labels view is the capability of getting it listed and read the permission checked when the label is going to be applied. Set by default. |
| rm | Delete a given object. Set by default. |
| read | Read data from a given object. Set by default. |
| chgowner | Change the owner of an object. Set by default. |
| rename | Change the name of an object.  For workspaces: required to be able to change workspace location too. Set by default. |
| Repository server | mkrepository | Allows or denies creating repositories on a given repository server. Set by default. |
| Repository | mkrevision | *mkrevision* is an special permission that allows denying all kinds of revision creation on a repository. It is *always* checked for repository, branch and item. It overrides the checkout permission. Set by default. |
| mkitem | Creating items on a given repository. It is checked during add operations. Set by default. |
| mkbranch | Creating branches on the repository. Set by default. |
| mkaction | Creating actions (triggers) on the repository. Set by default. |
| mklink | Creating links. Set by default. |
| mkattr | Creating attributes. Set by default. |
| mklabel | Creating labels. Set by default. |
| advancedquery | Running advanced queries on the repository. Not set by default. |
| All repository objects | applyattr | Permission to get an attribute applied. Set by default. |
| applylink | Permission that allows an object to be linked. Set by default. |
| Revision / All revisions in branch | co | Check out. Permission to check out a given revision, creating a new one. Set by default. |
| unco | Permission to undo a check out. Set by default. |
| ci | Permission to check in a given revision. Set by default. |
| applylabel | Permission to get labels applied. Set by default. |
| Branch | mergefrom | Permission to merge from a specific branch. Set by default. |
| mkchildbranch | Permission to create child branches of a specific one. Set by default. |
| replicate | Permission to replicate branches from a repository to another. Set by default. It allows getting data from the source of the replication. If the user has not permissions in the destination repository the replication operation will fail. Also it is necessary to have set the mkbranch, mkchildbranch, mklabel, mkrevision and applylabel permissions to perform the replication. |
| ItItem | obliterate | Removes the item permanently from the system, removing every revision of the items in the databases. The item cannot be recovered after this operation again. Not set by default. |

Table 1. Detailed permission description

## Permission calculation

To calculate the permissions of a given object all its permissions´ hierarchy is considered. Ancestor’s permissions are *combined* with the object’s ACL entries. *Combined* means that both the *allowed* and *denied* entries for each user or group are merged together with a logical OR operation: for instance, if a repository doesn’t have the *co* (check out) permission allowed or denied but the item object has it allowed, the revision will have the permission.

When a certain ACL in the chain inherits from two ancestors, a special *combine* operation happens: instead of just *merging* both ancestors together, the *allowed* list is combined with a logical *AND* operator while the *denied* list is still combined using a logical *OR* operation.

Once the permissions for a given user are calculated they are combined with the permissions of the group the user belongs to, plus the special *ALL USERS* group.

Figure 4 shows a sample permission calculation. First row are allowed permission, second row denied. Sample shows how permissions from different ancestors are combined.

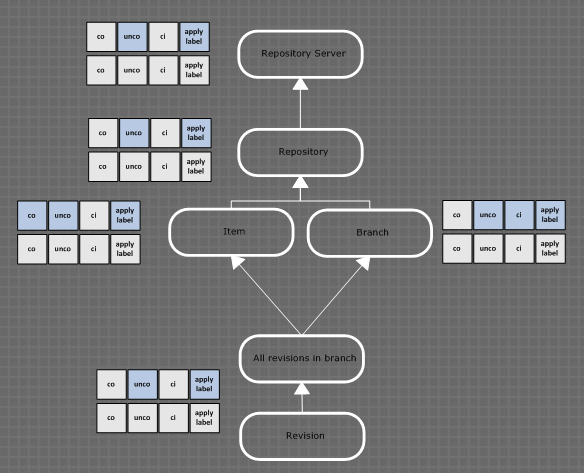


Figure 4. Permission calculation simple

# Users and groups

ACLs specify permissions granted and denied for different users and groups. When the permissions for a given user on a certain object are calculated Plastic first resolves inheritance for each SEID (user or group) and then combines them taking into account group and user relationships.

Figure 5 depicts a scenario in which a user is both a developer and an integrator at the same time. The permissions displayed refer to the main development line: branch br:/main. Integrators have full permissions to check out, check in and label on the main branch, but developers are not allowed to perform changes on the main branch, they don’t have the check in and apply label permission allowed. Note that instead of denying the permissions the sample doesn’t assign them to developers: denying is more restrictive and will always prevail in a multi-group scenario. Plastic will combine the permissions of developers and integrators to calculate the rights of the user who will finally be allowed to perform the operations on main branch because he belongs to the integrators group.

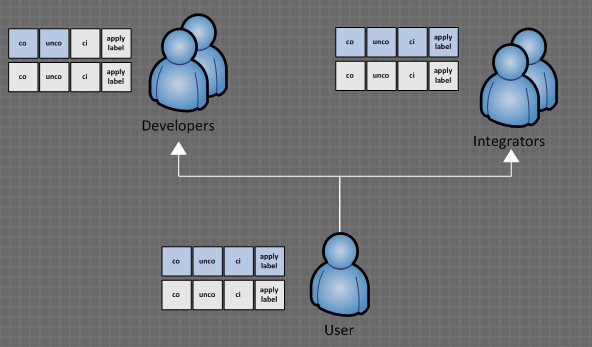


Figure 5. Security: user in two groups. Permissions on branch.

Figure 6 shows how denied permissions are applied in a multi-group scenario. In contrast with disabled (not allowed or denied) permissions, denied ones will always prevail in combination operations. The sample displays repository permissions: Group 1 has the *mkbranch* permission denied: every user in Group 1 won’t be able to create branches on the repository. Group 2 has the same permissions allowed. The user in the sample belongs to both groups so Plastic will deny the *mkbranch* permission to the user: combining both groups will make the denied permission *prevail*.

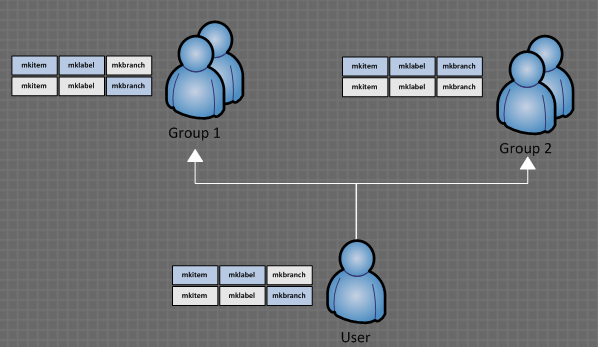


Figure 6. Security sample: denied permissions on groups

## Special users

Plastic SCM security system manages two kind of special users which help setting security policies.

### ALL USERS

Refers to all the users recognized by Plastic SCM. By setting permissions to this special group all the users will be affected. By default the system is totally open so the *all users* group has all the permissions allowed.

It is important to note that denying permission to the *all users* group will affect all the users overriding any other permission.

### OWNER

The *owner* user is a special user used to allow or deny permissions to the owner (normally the creator, although it can be changed) of a specific object. By setting permissions to the special *owner* object the user customizes what any owner can or can’t do in Plastic.

For instance, it can be used to prevent any other user than the owner to perform undo checkout operations, so only the user who checked out a file or directory can undo the operation.

# Permissions on the GUI tool

Permissions can be set from the GUI tool using the *permissions dialog* which is able to display and edit the ACL of any object stored under Plastic SCM.

## Setting basic permissions

Figure 7 depicts the main elements of the permissions dialog. The dialog consists on two *tabs*, the first one showing the users and groups and their permissions, and the second one both advanced information and advanced operations. In the upper side of the dialog a full specification of the object which ACL is being edited is displayed.

The *permission* *tab* shows the following information:

* A *list* of the users and groups present on the ACL. Each of them will have a list of permissions allowed, denied or disabled.
* A list of the permissions and their status is displayed just below the *users and groups* list. Each time a user or group gets selected on the *users and groups* *list* the contents of the *permissions list* get updated.
* The information about the owner of the object is displayed at the bottom of the *permission tab*.
* The *add* and *remove* buttons placed below the *users and groups* list allow adding or removing users and groups from the ACL. When a user or group is *inherited* from a parent ACL the system won’t allow it to be removed.
* The *owner* button at the bottom of the *permission tab* lets the user to change the owner of the object.

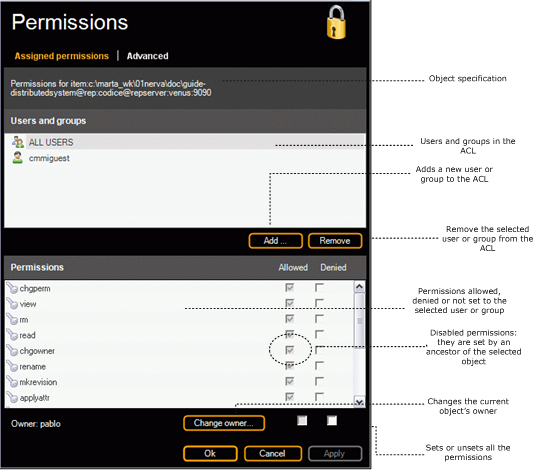


Figure 7. Permissions dialog

## Advanced permission information

The *advanced options* *tab* displays additional information about the ACL. Figure 8 shows a sample ACL and all its advanced information. The *inherited permissions* list details information about each specific permission present on the ACL. It shows which user has the permission, whether it is allowed or denied and the object from which it is inherited if appropriate.

The object specification displayed as *parent* of the ACL inheritance is calculated in the following way: consider you are displaying the permissions of a given item, as the Figure 8 does. It shows that the permission *read* is allowed for *ALL USERS* and it is inherited from the *repository server*. What does it mean? The item inherits its permissions from the repository and the repository from the repository server. So the higher object in the hierarchy defining the permission will be the one displayed on the dialog.

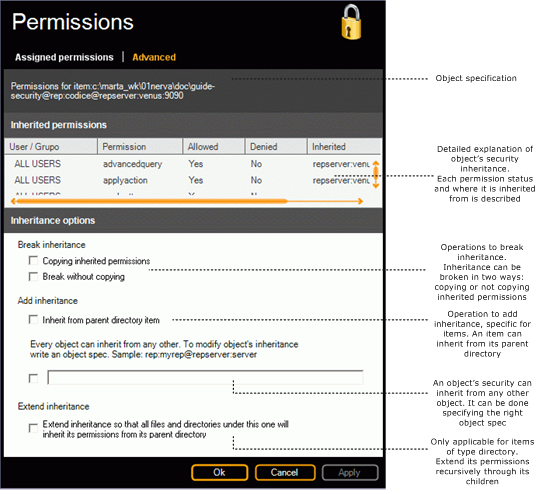


Figure 8. Permissions dialog advanced options

## Break inheritance

The bottom side of the dialog is focused on inheritance operations. An object’s ACL can inherit from the ACL of any other object, but this inheritance can be broken too. Inheritance can be broken in two ways: copying or not copying the permissions.

Break inheritance copying: when this option is set, the inheritance entries in the object’s ACL will be removed, but the inherited permissions will be assigned to the ACL. For instance: if the *read* permission is inherited from the repository server and assigned to *ALL USERS*, after breaking inheritance copying the permission will be present in the object’s ACL.

Break inheritance without copying: if this option is set the inheritance entries will be removed from the object’s ACL but the inherited permissions won’t be set to the ACL. If the ACL had its own entries it will keep them, otherwise it will be left empty.

## Reapply inheritance

Inheritance can also be *reapplied* or added using the *add inheritance* options in the advanced dialog tab. If the ACL being edited corresponds to an item, the *inherit from parent directory* option will be enabled, and the user will be able to restore a file or directory security making it inherit from its containing directory item.

The second option is specifying a complete object specification from which security will be inherited.

## Extend inheritance

The *extend inheritance* is available for directory item permissions. The user will be able to extend the permissions of any given directory item to all of its contained files and subdirectories recursively, replacing their actual permissions by the ones set in the directory. The permissions are not actually copied; instead each object’s ACL gets cleared and inherits from its parent directory.

The sample depicted on Figure 9 shows a directory structure in which two items have their own assigned permissions which don’t inherit from their parents´ ACL.



Figure 9. Sample directory structure before extending inheritance

After expanding inheritance from the root directory, the ACLs of the contained files and directories will directly inherit from their parent’s directory ACL. Permissions are then set to the ones of the parent, and the permission hierarchy is reestablished. Figure 10 shows the situation after the extend inheritance operation is executed from the top directory.



Figure 10. Sample directory structure after extending inheritance

If a file or directory inside the tree which is being *reassigned* has entries in its ACL and it, in turn, inherits from another element, the entries are cleared and the inheritance is established to the parent directory. Figure 11 shows a complete example depicting the mentioned situation.



Figure 11. Full sample showing how an ACL with inheritance is removed

# Setting security from the command line client

Security can be also set from the command line client using the *cm acl* and *cm sa* commands. To list all the available permissions the command *cm showpermissions* or *cm sp* must be used.

To obtain help on the security commands type *cm help acl*, *cm help sa* or *cm help showpermissions*.

## Displaying permissions

The *cm showacl* command, or *cm sa* for short, displays the permissions of any given object specification. Figure 12 shows a sample displaying the permissions of the item *01nerva*. The *cm sa* command tells the user which permissions are available, to which users or groups, and where they come from considering the object’s hierarchy.



Figure 12. Command to display the permissions of an item

The *–extended* modifier allows the user to obtain more information about how the permissions and ACLs are actually defined. Figure 13 shows a full explained sample of the *–extended* option.



Figure 13. Extended ACL information

The two previous examples have focused on item permissions, but the command line tool can work with any object stored under Plastic SCM. Figure 14 displays the permissions of a certain revision of a directory. To select a given revision a revision specification is used.



Figure 14. Command line tool displaying permissions of a given revision

Figure 15 includes a detailed explanation on how permissions are set to a revision, considering the full ACL hierarchy involved.



Figure 15. Detailed explanation of a revision's permissions

## Setting permissions

To modify permissions from the command line use the *cm acl* command. Figure 16 gives a detailed explanation of the modifiers available for the *cm acl* command. Using the *acl* command it will be possible to add or remove users or groups to an ACL and allow, unallow, deny or undeny permissions. Inheritance can also be manipulated using the *–cut*, *-cutncpy* and *–inherit* options.



Figure 16. Explanation of the *acl* command syntax

Taking the same sample used in the previous topic, Figure 17 denies the *co* permission on the main branch, and checks how the permissions of revision *1* of the directory *01nerva* gets modified.



Figure 17. Denying check out permission on a branch

Figure 18 illustrates the sample in more detail making use of the *–extended* option of the *cm sa* command.

Now the branch associated ACL which previously had only an inheritance entry, introduces its own ACL entry for *ALL USERS* specifying the system a denied *co* permission.

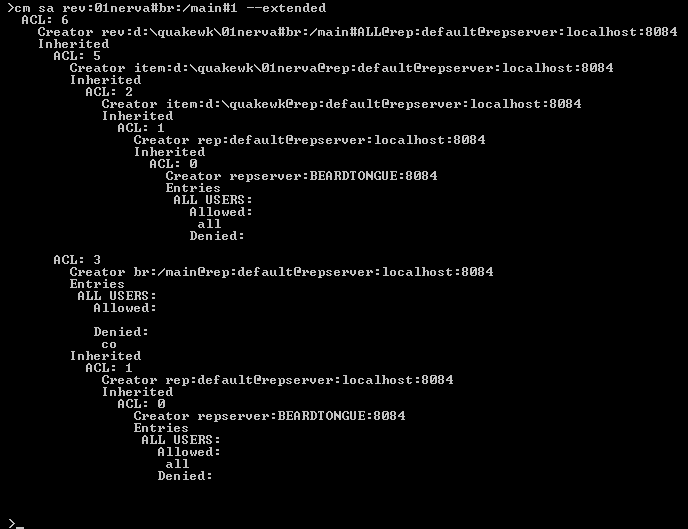


Figure 18. Extended information

## Manipulating inheritance

Inheritance can be manipulated using the *cm acl* command, and the *–cut, -cutncpy* and *–inherit* modifiers.

Figure 19 shows a sample where the branch br:/main inherits its permissions from the repository server and also has a specific permission denied for all users. After cutting inheritance copying, both entries are *non-inherited* and correctly displayed by the *cm sa* command.



Figure 19. Cut inheritance copying simple

It is also possible to add inheritance to an object, making its permissions inherit from any other object in the system (considering hierarchy restrictions: an object can never inherit from another object which is below in the hierarchy, for instance: a repository can’t inherit from a revision).

Figure 20 shows how inheritance can be added again to the branch br:/main used in the previous example.



Figure 20. Add inheritance scenario

# Securing a Plastic SCM system

When Plastic SCM is first installed the default security set up is entirely open. This means that any user authenticated by the system will have full access granted.

If your environment requires security restrictions, whether they are to prevent unwanted access or enforce certain development policies, you should consider the following hints:

* Consider changing the permissions to the repository server as the first step to set up a security policy. Changing the permissions to the top level element in the security hierarchy will ensure that all the rest of the objects get secured.
* Define which users and groups will have access to the Plastic SCM system, and give them the right access in the repository server. Later on you can customize specific privileges to repositories, branches and even items if required.
* Understand correctly the meaning of each of the available permissions and their impact on the system, carefully reading the present guide.

# Case studies

## Case 1: Restrict integration branches

Developers use several branches to work following a *branch per task* branching pattern. There are two different groups involved in development: integrators and developers. Certain developers can act as integrators too. The security policy to enforce is the following: developers can’t check out, check in or label on the integration branches, only integrators can perform these operations.

To solve the scenario an administrator will do the following:

* Break inheritance copying on integration branches. This way the integration branches will continue having the same permissions as before, but now it will be possible to modify the previously inherited permissions.
* Disable the check out, check in and apply label permissions to developers (not deny but disable). Integrators will have these permissions allowed, but developers won´t be able to execute the mentioned operations. A user belonging to both groups will have the operations allowed: combining a disabled permission with an allowed one results in an allowed permission.

## Case 2: Restricted access to source directories

There are two different teams on the company: developers and analysts. Developers will have full access to the source code directory tree, but analysts will have only *read* access.

The steps to set up the scenario are:

* Ensure the permissions under the source code tree are correctly defined by *expanding* permissions from the root of the tree (there is a specific option in the advanced tab of the permissions form).
* Cut inheritance copying in the source code root directory item, if needed.
* Disable the co, unco, ci and applylabel permissions in the source code root directory item for the analysts group. Keep all permissions allowed for developers.

## Case 3: Restricted access to branches

A development team has the following participants:

* A system administrator
* A development group
* An integrators group

Developers can only work on development branches.

Integrators can only make changes on integration branches.

Administrators have full access.

If a user belongs to several groups, he will have all the combined benefits (if a user is both a developer and administrator, he should be given full access).

Steps to set up the scenario:

* The *mkbranch* permission will be disabled at the repository level for both integrators and developers.
* Development branches: integrators will have *co*, *ci*, *mkrev* and *applylabel* permissions disabled. Developers will have all permissions allowed.
* Integration branches: developers will have *co*, *ci*, *mkrev* and *applylabel* permissions disabled. Integrators will have all permissions allowed.

## Case 4: Guest consultants

When external consultants work together with the development group during a quality assessment (e.g. a CMMi evaluation), it is normally required to give them read access to repositories, but ensure that they can’t perform any modification to the code or the documentation.

To set up this scenario using Plastic SCM security, consider the following steps:

* Create a *guests* group on your operating system (or set any suitable name).
* Add the newly created group to the required repositories, and deny the permissions *mkrevision*, *mkbranch*, *mkchildbranch*, *mkitem*, *co* and *ci*.

## Case 5: Set a repository as readonly for a given set of users

To restrict access to a given set of users to a certain repository so that they can just have read access but not being able to create revisions or any other object, consider the following steps:

* Add the group to the repository.
* Deny (or disable, depending on whether you need to support users being member of several groups and having the less restrictive set of permissions or not) the following permissions: *chgpermission*, *mkrevision*, *mkbranch*, *mkchildbranch*, *mkattr*, *applylabel*, *mklabel*, *unco*, *ci*.

## Case 6: Only integrators can check in to the main branch

In the proposed case study only the integrators can perform a check in operation to the main branch, although developers would be able to check out files and directories.

To set up this scenario consider the next step:

* Disable the *ci* permission to the *developers* group on the main branch. This way a member who is both developer and integrator will have *ci* permission allowed, but users who are just developers won’t be able to perform check in operations on the main branch.

## Case 7: Only integrators can modify the main and the maintenance branches

The purpose of the case is restricting the access to the revisions stored on both a main line branch and a maintenance branch. Only integrators will be allowed to perform changes on those branches. Developers will only be able to *read* data from those branches, but won’t be allowed to perform any change.

In order to implement the scenario consider the following steps:

* Disable all permissions, except *view* and *read* to the developer’s group.

## Case 8: Only the owner of a check out can undo the check out

This scenario is useful to prevent a user to *undo* a check out to another user. Normally in a secured scenario only the owner and the administrator will be allowed to perform an *uncheckout* operation.

Setting up this behavior is very simple with Plastic SCM: disable the *unco* permission for ALL USERS at the repository or repository server level, and leave the permission enabled just for the OWNER special user and the *administrator’s* group.

## Case 9: Open a repository to a given group

Consider a company which develops several projects. Each of them is run by a certain development group which is assigned a given repository to work with.

Each repository can be thought as a *component* in which certain functionality is implemented. So, at a given moment in time, a second development group will need *read access* to a repository not directly handled by them so they can use it as a *subproduct* (a dependency or a library) for their project.

Setting this scenario with Plastic SCM will involve the following considerations:

* The two repositories involved will be named *rep00* and *rep01*.
* The two groups will be *grp00* and *grp01*.
* *grp00* has full access to *rep00* and *grp01* has full access to *rep01*.
* grp01 will need read access to rep00.
* The administrator will give *read* and *view* permissions to the *rep00* main branch for *grp01*.