ALLSEEN ALLIANCE

AllJoyn Security 2.0

Feature Test Plan Draft

February 11, 2015

Contents

Revision History

Revision	Date	Change Log	Owner
1	01/09/2015	Initial Version	narasubr
2	01/19/2015	Review comments	narasubr
3	02/11/2015	Review comments and addition of test areas	narasubr

Reference Documents

Reference Number	Document Name	Document Number	Document Path/Location/Link
1	Security 2.0 design		https://wiki.allseenalliance.org/core/security_enhancements

Acronyms

Acronym/term	Description
FR	Factory Reset
MC	Membership Certificate
ASC	AllJoyn Standard Client
ATC	AllJoyn Thin Client

Approvals

	Title	Print Name	Signature	Date
1	Development Lead	Phil Nguyen		

1 Introduction

Security 2.0 is an enhancement feature that will allow an application to validate access to secure interfaces or secure objects based on policies installed by the owner. The controlee will have access to Secure Interfaces or Secure Objects based on these policies. In addition to the encrypted messaging (using AES CCM) between the peers, the Security 2.0 Permission management module manages a database of access credentials and Access Control Lists (ACLs). The AllJoyn Core Permission Management component will also do enforcement including concept of mutual

authorization before any message action can be taken. It will not be up to the user to enforce permission but user can dictate how application will perform based on the access control lists (ACLs) defined for the application.

The end user Security manager is an optional service that helps user with key management and permission rule building. The security Manger is optional because the permissions can be installed directly into the application. The end user Security manager is, as of now out of the scope of this document as it is being developed by other alliance members.

This document outlines the test plan and test execution strategy for the security enhancements for the 15.04 release. Prominent developments/enhancements of this feature include:

- Enhancements to existing ECDHE_DSA mode of authentication to enforce peer trust.
- Claiming of devices
- Policy management
- Access control
- Enhancements to keystore file to store identity certificates, policies, and membership certificates.

The intent of this test plan is to create a standard, mutually agreed upon document, which will enable the test team to make an informed decision about the overall test scope and coverage will be for this product. Once a standard has been established; the success or failure of the features within the scope can be quantitatively measured, and appropriate feedback can be provided the stake holders.

1.1 Test Process

The testing activity will comprise of development of detailed test cases and executing them. The functional testing will be conducted on the Linux platform for AllJoyn Standard client and AllJoyn thin client libraries. Some end-end-end tests will be conducted across opposite endianness machines to test over-the-wire messaging between buses.

Discrepancies or potential enhancements identified while testing on the feature branch will be reported in AllSeen JIRA.

1.1.1 Test effort estimate and schedule

Table 1

#	**Feature**	**Estimated time**
1.	Writing test cases	4 weeks
2.	Test execution	TBD

1.2 Test Entrance and Exit Criteria

1.2.1 Entrance Criteria

- Feature branch should be signed off as development complete and ready for testing by the development lead for the feature. `
- Test applications should be available for testing. This includes unit tests, end-end-end system test applications.
- CI builds and automated tests (if any) should pass on the feature branch.

1.2.2 Exit Criteria

Testing will conclude after all the test case results have been agreed upon by the stakeholders.

1.2.3 Assumptions

- Security Manager Module is not in the scope of testing this feature.
- After Claiming a device, "No Policy" or "empty Policy" means "implicit deny" for all.
- Security 2.0 works only with ECDHE_DSA. Other authentication modes like SRP, RSA, etc. are not supported.
- Test plan covers only the code contributed by QCE. Please look at the feature matrix as of 2/4/2015 for more details.
- Set up TV by Dad, Son and its use cases are covered by the security manager. They are not included in this test plan. Please look at the feature matrix (IDs #34 - #39) as of 2/4/2015 for more details.
- **Test plan is subject to change as the HLD of the feature undergoes any change.**

2 Test Strategy

The test cases are divided into the following areas:

- Claiming of devices
- Policy management
- Access control
- Manifest
- End-to-End test
- Memory profiling

The test cases (as specified in Section 3) will be run for Standard client and thin client. The functional tests are platform and binding independent and hence will be run on Linux platform only for C++ bindings. An end-to-end test will be run on different endianness machines (OpenWRT and Linux). The term “device bus” means that it is a BusAttachment for both- ASC and ATC.

3 Test Coverage

High level overview of the test cases is presented below.

3.1 Claiming of devices

Test Objective	Procedure and Verification	Applicable for Standard client and Thin client
Device in FR mode.	Configure a device with “Claimable state”. The device is in FR condition. The device emits a PermissionMgmt.Notification. Verify that claimableState = ‘claimable’, policySerialNum = 0. Store the public key of the device. Another bus calls GetPublicKey() on the device bus. Compare the public key returned by the API with that coming from the notification. They should match.	Yes
Claim a device using ECDHE_NULL mode	Configure a device with "Claimable" state. The device is in FR condition. Claim the device by the admin bus using ECDHE_NULL mode. Check for PermissionMgmt.Notification. Check for new GUID (The new GUID is the GUID set by the admin bus which should be different than the device's keystore file), claimableState='claimed' , policySerialNum=0. Call GetPublicKey() on the claimed device from another bus and verify the public key of the claimed device. Call GetIdentity() on the claimed device by the above bus and verify the Identity certificate. The identiy cert returned by the API should be the same as that generated by the admin bus.	**Yes**

Claim a device using ECDHE_PSK mode	Configure a device with "Claimable" state. The device is in FR condition. Claim the device by the admin bus using ECDHE_PSK mode. Check for PermissionMgmt.Notification. Check for new GUID (The new GUID is the GUID set by the admin bus which should be different than the device's keystore file), claimableState='claimed' , policySerialNum=0. Call GetPublicKey() on the claimed device from another bus and verify the public key of the claimed device. Call GetIdentity() on the claimed device by the above bus and verify the Identity certificate. The identiy cert returned by the API should be the same as that generated by the admin bus.	**Yes**
Claim a device using SRP mode	Configure a device with "Claimable" state. The device is in FR condition. Claim the device by the admin bus using SRP mode. Check for PermissionMgmt.Notification. Check for new GUID (The new GUID is the GUID set by the admin bus which should be different than the device's keystore file), claimableState='claimed' , policySerialNum=0. Call GetPublicKey() on the claimed device from another bus and verify the public key of the claimed device. Call GetIdentity() on the claimed device by the above bus and verify the Identity certificate. The identiy cert returned by the API should be the same as that generated by the admin bus.	**No**
Claim an an already claimed device.	Claim a device using ECDHE_NULL mode. Try to claim it again using the same admin bus. Claim method-call should fail. Error should indicate that the device is already claimed.	**Yes**
Claim an already claimed device by a different non-admin bus.	Try to claim an already claimed device by a different non-admin bus, get an Error. Error should indicate that the device cannot be claimed non-admin.	**Yes**
Claim a “Non- claimable” device.	Configure a device with "NotClaimableState". The device is in FR condition. Claim the device by the admin bus using ECDHE_NULL mode. Claim should get an appropriate error indicating the device cannot be claimed.	**Yes**
Claim and Reset	Configure a device with "Claimable" state. The device is in FR condition. Claim a device. Check the PermissionMgmt.Notification for Check for new GUID (The new GUID is the GUID set by the admin bus which should be different than the device's keystore file), claimableState='claimed' , policySerialNum=0 Reset the device by the above bus. Check the PermissionMgmt.Notification for Check for GUID (The GUID should be same as above), claimableState='Unclaimed' policySerialNum=0 Claim the device again from a new bus. Check the PermissionMgmt.Notification . Check for new GUID (The new GUID is the GUID set by the admin bus which should be different than the device's keystore file), claimableState='claimed' , policySerialNum=0	**Yes**

Reset the device by non-admin bus.	Reset the device by a non-admin bus, get an Error. Error should indicate that the device cannot be reset by non-admin.	**Yes**
Claim a device using an incorrect identity certificate.	Admin generates an identity certificate based out a a different public key. This public key is not the actual public key of the device. Claim the device. Claim should return an error. Error should indicate that an incorrect identity certicate was passed.	**Yes**
Identity certifcate is signed by admin using a different public key.	Admin generates an identity certifcate signed by a different key. The key used for signing is different than the public key of the admin. Claim the device. Claim should return an error. Error should indicate an incorrect identity certifcate.	**Yes**

3.2 Installation of policy, Membership certificates

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
InstallPolicy is called by a non-admin bus.	InstallPolicy is called by a non-admin bus. It should return error. Error should indicate that the Method can be called by the admin bus only.	Yes
InstallPolicy and GetPolicy	Create a policy, note the serial number. call InstallPolicy using the admin bus. MethodCall should return success. Check the PermissionMgmt.Notification.policySerialNumber. It should match with the above serial number. call GetPolicy using a non-admin bus. It should return error. Error to indicate a permission problem. call GetPolicy using an admin bus. Method should be successfull. Compare the policy returned with the policy created. They should match.	**Yes**
RemovePolicy is called by a non-admin bus.	RemovePolicy is called by a non-admin bus. It should return error. Error should indicate that the Method can be called by the admin bus only.	**Yes**
Install a policy without a serial number.	Setup a policy and ensure that the serial number is not set. InstallPolicy is called with the above policy. Method call should fail. The error should indicate invalid args.	**Yes**
Install a policy with the same serial number.	The device bus where the policy is being installed should return an error. This is because if the policy is installed, another Notification will be sent with the same serial number. Error should indicate “Policy with the same serial number already exists.)	**Yes**
Install a policy with the serial number lesser than the previous policy.	InstallPolicy should return an error. A new policy installed should have a serial number greater than the previous.	**Yes**

InstallPolicy, RemovePolicy	Install a policy on a device bus. Check for PermissionMgmt.Notification.policySerialNumber. It should match with the policy serial number. Remove the policy on the device bus. Check for PermissionMgmt.Notification.policySerialNumber. It should be 0. Call GetPolicy on the device bus. It should fetch empty policy set.	**Yes**
Membership certificate is not signed by the admin.	InstallMembership is called by any bus but it uses a membership certificate not signed by the admin or guild authority. Method should fail with Permission denied.	**Yes**
Membership certifcate is non X509 DER format.	InstallMembership should return error at the sender side indicating “Invalid certificate format”	**Yes**
InstallMembership and RemoveMembership	InstallMembership is called by admin bus. The membership certificate is signed by the admin. The MC belongs to GUILD1. Methodcall is successful. InstallMembership is called again. The membership certificate is signed by the admin. The MC belongs to GUILD2. Methodcall is successful. RemoveMembership is called using the serial number related to GUILD2. Methodcall is successful. RemoveMembership is called using the serial number related to GUILD1. Methodcall is successful.	**Yes**
RemoveMembership is called by a non-admin bus.	RemoveMembership is called by a non-admin bus. It should return error (permission denied.)	**Yes**
RemoveMembership called on an invalid serial number.	RemoveMembership called on an invalid serial number. Methodcall should return error. Error indicating that there is no such certificate installed on the device.	**Yes**
InstallMembershipAuthData is called with a serial number not repesenting the membership certificate.	InstallMembershipAuthData is called on the device bus with a serial number not representing the membership certificate. Methodcall should return error. Error indicating “certificate not found”.	**Yes**
InstallMembershipAuthData is called on an existing serial number but for a different membertship certificate.	admin installs Membership certificate 1 with (GUILD1, serial1) admin installs Membership certificate 2 with (GUILD2, serial2) admin calls InstallMembershipAuthData with serial1 but for Membership certificate 2. The method call should return error indicating "Invalid hash digest"	**Yes**
InstallIdentity is called by non-admin bus.	InstallIdentity is called by non-admin bus. Methodcall should return error (permission denied.)	**Yes**

Install a new identity and GetIdentity.	Admin claims a device. Admin calls GetIdentity on the device. Admin matches the issuer of the certficate with itself using GetIdentity. Admin generates an identity certificate for the device and he changes some fields. (The subject field and the issuer filed still remains the same.) Admin bus calls InstallIdentity with the new certificate. InstallIdentity is successful. A new bus calls GetIdentity on the device bus to check that the identity cert of the device has changed.	**Yes**
InstallIdentity is called using a non X.509 DER format.	The method call should return error at the sender side indicating "Invalid certificate format"	**Yes**
PolicyInstallation works only with ECDHE_DSA, does not work with SRP mode.	Admin claims a device bus using SRP mode of authentication. Admin and device bus clear keys of remote peers. Establish a SRP based session between admin and device bus. Admin calls Install policy on device bus. Error should indicate both sides cannot verify trust establishment with each other.	**No**
PolicyInstallation works only with ECDHE_DSA, does not work with RSA mode.	Admin claims a device bus using RSA mode of authentication. Admin and device bus clear keys of remote peers. Establish a RSA based session between admin and device bus. Admin calls Install policy on device bus. Error should indicate both sides cannot verify trust establishment with each other.	**No**
PolicyInstallation works only with ECDHE_DSA. Keystore not cleared.	Admin claims a device bus using SRP mode of authentication. Admin calls Install policy on device bus. (The keys have not expired and thus an SRP based session is established.) Error should indicate both sides cannot verify trust establishment with each other.	**No**
Claim using SRP mode of authentication, but InstallPolicy using ECDHE_DSA mode.	Admin claims a device bus using SRP mode of authentication. Admin and device bus clear keys of remote peers. Establish a ECDHE_DSA based session between admin and device bus. Admin calls Install policy on device bus. This should be successful.	**No**

3.3 Access Control

*X= Failed to send/receive signal

*V = Sent or Received the signal

Sender policy peer type	Action mask for the signal in policy	Action mask in the GUILD	Receiver policy peer type	Action mask for the signal in the policy	Action mask in the GUILD	Sent	Rcvd
ANY_USER	DENY		ANY_USER	PROVIDE		X*	X*
ANY_USER	PROVIDE		ANY_USER	PROVIDE		X	X
ANY_USER	MODIFY		ANY_USER	PROVIDE		V	V
ANY_USER	OBSERVE		ANY_USER	PROVIDE		V	V
ANY_USER (The policy is not GUILD specific, so don't bother checking the GUILD authorization data. The DENY action mask will not take effect.)	OBSERVE	DENY	ANY_USER	PROVIDE	DENY	V	V
SPECIFC_USER (The destination is the specific user)	DENY		ANY_USER	PROVIDE		X	X
SPECIFC_USER (The destination is the specific user)	PROVIDE		ANY_USER	PROVIDE		X	X
SPECIFC_USER (The destination is the specific user)	MODIFY		ANY_USER	PROVIDE		V	V
SPECIFC_USER (The destination is the specific user)	OBSERVE		ANY_USER	PROVIDE		V	V
SPECIFIC_USER (The destination is a non-specific user.)	DENY		ANY_USER	PROVIDE		X	X
SPECIFIC_USER (The destination is a non-specific user)	PROVIDE		ANY_USER	PROVIDE		X	X
SPECIFIC_USER (The destination is a non-specific user)	MODIFY		ANY_USER	PROVIDE		X	X
SPECIFIC_USER (The destination is a non-specific user)	OBSERVE		ANY_USER	PROVIDE		X	X
No policies installed.			ANY_USER	PROVIDE		X	X
Empty policies (no rules)			ANY_USER	PROVIDE		X	X
ANY_USER	OBSERVE		ANY_USER	DENY		V	X
ANY_USER	OBSERVE		ANY_USER	PROVIDE		V	V
ANY_USER	OBSERVE		ANY_USER	MODIFY		V	X
ANY_USER	OBSERVE		ANY_USER	OBSERVE		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the specific user)	DENY		V	X
ANY_USER	OBSERVE		SPECIFIC_USER	PROVIDE		V	V

			(The source is the specific user)				
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the specific user)	MODIFY		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the specific user)	OBSERVE		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the non-specific user.)	DENY		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the non-specific user.)	PROVIDE		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the non-specific user.)	MODIFY		V	X
ANY_USER	OBSERVE		SPECIFIC_USER (The source is the non-specific user.)	OBSERVE		V	X
ANY_USER	OBSERVE		No policies installed.			V	X
ANY_USER	OBSERVE		Empty policies (no rules)			V	X
GUILD (guild1)	OBSERVE	-	ANY_USER, belongs to guild1	PROVIDE	OBSERVE	V	V
GUILD (guild1)	OBSERVE	-	ANY_USER, belongs to guild1	PROVIDE	DENY	X	X
GUILD (guild1)	OBSERVE	-	ANY_USER, belongs to guild1	PROVIDE	MODIFY	V	V
GUILD (guild1)	OBSERVE	-	ANY_USER, belongs to guild1	PROVIDE	PROVIDE	X	V
GUILD (guild1) (Guilds are not matching, sender won't be able to send.)	OBSERVE	-	ANY_USER, belongs to guild2	PROVIDE	OBSERVE	X	X
GUILD (guild1) (Guilds are not matching, DENY's deny, sender able to send)	DENY	-	ANY_USER, belongs to guild2	PROVIDE	OBSERVE	V	V
GUILD (guild1) (Remote peer auth. data is empty, sender cannot send.)	OBSERVE	-	ANY_USER, belongs to guild1	PROVIDE	-	X	X
ANY_USER, belongs to guild1	OBSERVE	PROVIDE	GUILD (guild1)	PROVIDE		V	V
ANY_USER, belongs to guild1	OBSERVE	DENY	GUILD (guild1)	PROVIDE		V	X
ANY_USER, belongs to guild1	OBSERVE	OBSERVE	GUILD (guild1)	PROVIDE		V	X
ANY_USER, belongs to guild1	OBSERVE	MODIFY	GUILD (guild1)	PROVIDE		V	X
ANY_USER, belongs to guild1 (No auth. data for the sender side. Receiver won't accept the signal.)	OBSERVE	-	GUILD (guild1)	PROVIDE		V	X

GUILD (guild1), belongs to guild2	OBSERVE	PROVIDE	GUILD (guild2), belongs to guild1	PROVIDE	OBSERVE	V	V

*XXX = Failed in sending or receiving method calls, get property and set property

*VVV = Sent or Received method calls, get property and set property

Sender policy peer type	**Action mask for the signal in policy**	**Action mask in the GUILD**	**Receiver policy peer type**	**Action mask for the signal in the policy**	**Action mask in the GUILD**	**Sent**	**Rcvd**
ANY_USER	DENY		ANY_USER	MODIFY		XXX	XXX
ANY_USER	PROVIDE		ANY_USER	MODIFY		VVV	VVV
ANY_USER	MODIFY		ANY_USER	MODIFY		XXX	XXX
ANY_USER	OBSERVE		ANY_USER	PROVIDE		XXX	XXX
ANY_USER (The policy is not GUILD specific, so don't bother checking the GUILD authorization data. The DENY action mask will not take effect.)	PROVIDE	DENY	ANY_USER	MODIFY	DENY	VVV	VVV
SPECIFC_USER (The destination is the specific user)	DENY		ANY_USER	MODIFY		XXX	XXX
SPECIFC_USER (The destination is the specific user)	PROVIDE		ANY_USER	MODIFY		VVV	VVV
SPECIFC_USER (The destination is the specific user)	MODIFY		ANY_USER	MODIFY		XXX	XXX
SPECIFC_USER (The destination is the specific user)	OBSERVE		ANY_USER	MODIFY		XXX	XXX
SPECIFIC_USER (The destination is a non-specific user.)	DENY		ANY_USER	MODIFY		XXX	XXX
SPECIFIC_USER (The destination is a non-specific user)	PROVIDE		ANY_USER	MODIFY		XXX	XXX
SPECIFIC_USER (The destination is a non-specific user)	MODIFY		ANY_USER	MODIFY		XXX	XXX
SPECIFIC_USER (The destination is a non-specific user)	OBSERVE		ANY_USER	MODIFY		XXX	XXX
No policies installed.			ANY_USER	MODIFY		XXX	XXX
Empty policies (no rules)			ANY_USER	MODIFY		XXX	XXX
ANY_USER	PROVIDE		ANY_USER	DENY		VVV	XXX
ANY_USER	PROVIDE		ANY_USER	PROVIDE		VVV	VXX
ANY_USER	PROVIDE		ANY_USER	MODIFY		VVV	VVV
ANY_USER	PROVIDE		ANY_USER	OBSERVE		VVV	XVX
ANY_USER	PROVIDE		SPECIFIC_USER	DENY		VVV	XXX

			(The source is the specific user)				
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the specific user)	PROVIDE		VVV	XXX
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the specific user)	MODIFY		VVV	VVV
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the specific user)	OBSERVE		VVV	XVX
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the non-specific user.)	DENY		XXX	XXX
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the non-specific user.)	PROVIDE		VVV	XXX
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the non-specific user.)	MODIFY		VVV	XXX
ANY_USER	PROVIDE		SPECIFIC_USER (The source is the non-specific user.)	OBSERVE		VVV	XXX
ANY_USER	PROVIDE		No policies installed.			VVV	XXX
ANY_USER	PROVIDE		Empty policies (no rules)			VVV	XXX
GUILD (guild1)	PROVIDE	-	ANY_USER, belongs to guild1	MODIFY	PROVIDE	VVV	VVV
GUILD (guild1)	PROVIDE	-	ANY_USER, belongs to guild1	MODIFY	DENY	XXX	XXX
GUILD (guild1)	PROVIDE	-	ANY_USER, belongs to guild1	MODIFY	MODIFY	XXX	XXX
GUILD (guild1)	PROVIDE	-	ANY_USER, belongs to guild1	MODIFY	OBSERVE	XVX	XVX
GUILD (guild1) (Guilds are not matching, sender won't be able to send.)	PROVIDE	-	ANY_USER, belongs to guild2	MODIFY	PROVIDE	XXX	XXX
GUILD (guild1) (Guilds are not matching)	DENY	-	ANY_USER, belongs to guild2	MODIFY	PROVIDE	XXX	XXX
GUILD (guild1) (Remote peer auth. data is empty, sender cannot send.)	PROVIDE	-	ANY_USER, belongs to guild1	MODIFY	-	XXX	XXX
ANY_USER, belongs to guild1	PROVIDE	MODIFY	GUILD (guild1)	MODIFY		VVV	VVV

ANY_USER, belongs to guild1	PROVIDE	DENY	GUILD (guild1)	MODIFY		VVV	XXX
ANY_USER, belongs to guild1	PROVIDE	OBSERVE	GUILD (guild1)	MODIFY		VVV	XVX
ANY_USER, belongs to guild1	PROVIDE	PROVIDE	GUILD (guild1)	MODIFY		VVV	XXX
ANY_USER, belongs to guild1 (No auth. data for the sender side. Receiver won't accept the signal.)	PROVIDE	-	GUILD (guild1)	MODIFY		VVV	XXX
GUILD (guild1), belongs to guild2	PROVIDE	MODIFY	GUILD (guild2), belongs to guild1	MODIFY	PROVIDE	VVV	VVV

X*= Failed to Send or Receive GetAllProps

V*= Successfully Sent or Received GetAllProps

Sender policy peer type. Sender interface has two properties prop1, prop2	**Action mask for prop1, prop2**	**Action mask in the GUILD for prop1, prop2**	**Receiver policy peer type. Receiver interface has two properties prop1, prop2**	**Action mask for prop1, prop2**	**Action mask in the GUILD for prop1, prop2**	**Sent**	**Rcvd**
ANY_USER	PROVIDE, DENY		ANY_USER	OBSERVE, OBSERVE		X*	X*
ANY_USER	PROVIDE, PROVIDE		ANY_USER	OBSERVE, OBSERVE		V	V
ANY_USER	PROVIDE, PROVIDE		ANY_USER	OBSERVE, DENY		V	X
GUILD (guild1)	PROVIDE, PROVIDE		ANY_USER, belongs to guild1	OBSERVE, OBSERVE	PROVIDE, PROVIDE	V	V
GUILD (guild1)	PROVIDE, PROVIDE		ANY_USER, belongs to guild1	OBSERVE, OBSERVE	PROVIDE, DENY	X	X
ANY_USER, belongs to guild1	PROVIDE, PROVIDE	OBSERVE, OBSERVE	GUILD (guild1)	OBSERVE, OBSERVE		V	V
ANY_USER, belongs to guild1	PROVIDE, PROVIDE	OBSERVE, DENY	GUILD (guild1)	OBSERVE, OBSERVE		V	X

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
delegation with flag=true and new rules are subset of old rules. The auth. data is for the sender, but checked by the receiver.	sender supports signal1, signal 2 and signal 3. admin claims mid-user bus, consumer bus. mid-user bus claims sender bus. admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=true, gives PROVIDE access to signal 1, signal 2	Yes

	mid-user bus installs membership certificate on sender bus for 'guild1', gives PROVIDE access to signal1 only. sender has a ANY_USER policy, OBSERVE for signal1, signal2 and signal 3. Sender belongs to guild1. Consumer has a GUILD based policy for guild1, PROVIDE for * bus objects. sender bus emits signal1. It is received successfully by the consumer. sender bus emits signal2. It wont be received by the consumer.	
delegation with flag=false and new rules are subset of old rules. The auth. data is for the sender, but checked by the receiver.	admin claims mid-user bus, consumer bus. mid-user bus claims sender bus. admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=false, gives PROVIDE access to signal 1, signal 2 mid-user bus installs membership certificate on sender bus for 'guild1', gives PROVIDE access to signal1 only. sender has a ANY_USER policy, OBSERVE for signal1, signal2 and signal 3. Sender belongs to guild1. Consumer has a GUILD based policy for guild1, PROVIDE for * bus objects. sender bus emits signal1. It won't be received by the consumer. sender bus emits signal2. It won't be received by the consumer. (When the signal is received by the consumer, it validates the membership certs and the auth. data. At that time, it finds that the delegation flag was false. Thus it discards the cert. chain as a security violation. Thus the consumer has no membership auth. data for either signal 1 or signal 2. Thus, both will be rejected.)	Yes
delegation with flag=true and new rules are NOT subset of old rules. The auth. data is for the sender, but checked by the receiver.	sender supports signal1, signal 2 and signal 3. admin claims mid-user bus, consumer bus. mid-user bus claims sender bus. admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=false, gives PROVIDE access to signal 1, signal 2 mid-user bus installs membership certificate on sender bus for 'guild1', gives PROVIDE access to signal1 and signal 3 only. sender has a ANY_USER policy, OBSERVE for signal1, signal2 and signal 3. Sender belongs to guild1. Consumer has a GUILD based policy for guild1, PROVIDE for * bus objects. sender bus emits signal1. It is received successfully by the consumer. sender bus emits signal2. It wont be received by the consumer. sender bus emite signal3. It wont be received by the consumer. (Signal 3 should be authorized by all the members in the cert chain. It was not authorized by admin bus, thus it wont be received.)	Yes
delegation with flag=true and new rules are subset of old rules. The auth. data is for the receiver, but checked by the sender.	sender supports signal1, signal 2 and signal 3. admin claims mid-user bus, sender bus. mid-user bus claims consumer bus.	Yes

	admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=true, gives OBSERVE access to signal 1, signal 2 mid-user bus installs membership certificate on consumer bus for 'guild1', gives OBSERVE access to signal1 only. consumer has a ANY_USER policy, PROVIDE for signal1, signal2 and signal 3. Consumer belongs to guild1. sender has a GUILD based policy for guild1, OBSERVE for signal1, signal 2 and signal 3. sender bus emits signal1. It is received successfully by the consumer. sender bus emits signal2. It wont be sent by the sender.	
delegation with flag=false and new rules are subset of old rules. The auth. data is for the receiver, but checked by the sender.	sender supports signal1, signal 2 and signal 3. admin claims mid-user bus, sender bus. mid-user bus claims consumer bus. admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=false, gives OBSERVE access to signal 1, signal 2 mid-user bus installs membership certificate on consumer bus for 'guild1', gives OBSERVE access to signal1 only. consumer has a ANY_USER policy, PROVIDE for signal1, signal2 and signal 3. Consumer belongs to guild1. sender has a GUILD based policy for guild1, OBSERVE for signal1, signal 2 and signal 3. sender bus emits signal1. It wont be sent by the sender. sender bus emits signal2. It wont be sent by the sender. (When the signal is about to be sent, the sender validates the membership certs and the auth. data. At that time, it finds that the delegation flag was false. Thus it discards the cert. chain as a security violation. Thus the sender has no membership auth. data for either signal 1 or signal 2. Thus, both will be rejected at the sender side.)	Yes
delegation with flag=true and new rules are NOT subset of old rules. The auth. data is for the receiver, but checked by the sender.	sender supports signal1, signal 2 and signal 3. admin claims mid-user bus, sender bus. mid-user bus claims consumer bus. admin installs membership certificate on mid-user bus for 'guild1' with delegate flag=false, gives OBSERVE access to signal 1, signal 2 mid-user bus installs membership certificate on consumer bus for 'guild1', gives OBSERVE access to signal1 and signal3 only. consumer has a ANY_USER policy, PROVIDE for signal1, signal2 and signal 3. Consumer belongs to guild1. sender has a GUILD based policy for guild1, OBSERVE for signal1, signal 2 and signal 3. sender bus emits signal1. It is received successfully by the consumer. sender bus emits signal2. It wont be sent by the sender. sender bus emite signal3. It wont be sent by the sender. (Signal 3 should be authorized by all the members in the cert chain. It was not authorized by admin bus, thus it wont be sent.	Yes

Remove Membership should remove membership authorization data.	After InstallMembership, verify that signal is authorized. After RemoveMembership, verify that the signal is not authoprized anymore.	Yes
Reset the device after Installing policies and Membership certificates**.**	Claim a device bus, Install policies. Install membership certficates, auth data. Check for PermissionMgmt.Notification. Check that policy serialNumber and claimed state. Reset the device bus. Check for PermissionMgmt.Notification. Check that policy serialNumber=0 and claimed state and "unclaimed"	**Yes**
Keystore size should not grow upon multiple claim/install polices/install membership certifctaes/reset.	Claim a device bus, Install policies. Install membership certficates, auth data. Check Keystore size. Reset the device bus. Check Keystore size. Repeat above two operations 1000 times. KeyStore size should not have a growth because of claim/reset operations.	**Yes**
Memory profiling test	Run unit tests under valgrind. No memory leaks should be found.	**Yes**
Peers using RSA mode of authentication should not be able to send messages IF policies are installed.	Two peers are claimed and are using security 2.0 with policies installed. Peers set up a RSA session between themselves. Peers cannot send or receive messages as the trust establishment is not enforced with the RSA mode of authentication.	**No**
Peers using RSA mode of authentication should be able to send messages after Reset.	Two peers are claimed and are using security 2.0 with policies installed. Peers set up a RSA session between themselves. Peers cannot send or receive messages as the trust establishment is not enforced with the RSA mode of authentication. Admin resets the buses. Peers set up a RSA session between themselves. Peers CAN send or receive encrypted messages.	**No**
Admin user has no restrictions.	A device bus implements a method call and registers for a signal. Admin bus claims the device bus and has an empty policy installed on it. Admin bus makes a method call. Method call should be successful. Admin bus emits a signal. The signal should be received by the device bus.	**Yes**
If a member or interface is not secure, there are no policy enforcements.	A device bus implements an unsecure interface that supports a method call, signal and property. A peer bus also implements the same unsecure interface. Both the buses have DENY action on a policy on the interface. They also have membership certificates that has a DENY action on the interface. device bus makes a method call, get property, set property on the peer bus. They should be successful and unencrypted. device bus sends a signal. The signal will be received by the peer bus. The signal will be unencrypted.	**Yes**

	Peer bus makes a method call, get property, set property on the device bus. They should be successful and unencrypted. Peer bus sends a signal. The signal will be received by the device bus. The signal will be unencrypted.	
Admin installs Membership cert. signed by a different bus (bus1).	On provider: Admin bus installs Membership cert signed by bus1. On consumer: Admin bus installs Membership cert signed by bus1. Provider and consumer should be able to talk. This is because they both are issued certificate by bus 1. Bus1 is NOT the admin.	**Yes**
Admin installs Membership cert. signed by a different buses(bus1 and bus2).	On provider: Admin bus installs Membership cert signed by bus1. On consumer: Admin bus installs Membership cert signed by bus2. Provider and consumer should NOT be able to talk. This is because they both are issued certificate by different buses. Bus1 and bus2 are not admin.	**Yes**

3.4 Priorities of the policy terms and their precedence, multiple rules

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
DENY takes precedence over anything, policy precedence on the sender side. (Single term, multiple rules)	Sender side: Term 1: ANY_USER: Method call, PROVIDE Method Call, DENY Receiver side: Term 1: ANY_USER: Method call, MODIFY Method call cannot be sent.	**Yes**
DENY takes precedence over anything, policy precedence on the sender side. (Single term, multiple rules)	Sender side: Term 1: ANY_USER: Method call, DENY Method Call, PROVIDE Receiver side: Term 1: ANY_USER: Method call, MODIFY Method call cannot be sent.	**Yes**
DENY takes precedence over anything, policy precedence on the receiver side. (Single term, multiple rules)	Sender side: Term 1: ANY_USER: Method call, PROVIDE Receiver side: Term 1: ANY_USER: Method call, MODIFY Method Call, DENY Method call will be sent, but won't be received.	**Yes**

DENY takes precedence over anything, policy precedence on the receiver side. (Single term, multiple rules)	Sender side: Term 1: ANY_USER: Method call, PROVIDE Receiver side: Term 1: ANY_USER: Method call, DENY Method Call, MODIFY Method call will be sent, but won't be received.	**Yes**
DENY takes precedence over anything, policy precedence on the sender side. (Multiple terms on sender side)	Sender side: Term 1: Specific_user: Method call, PROVIDE Term 2: Specific_user: Method Call, DENY Receiver side: Term 1: ANY_USER: Method call, MODIFY Method call to the 'specific user' cannot be sent.	**Yes**
DENY takes precedence over anything, policy precedence on the sender side. (Multiple terms on receiver side)	Sender side: Term 1: ANY_USER: Method call, PROVIDE Receiver side: Term 1: Specific_user: Method call, MODIFY Term 2: Specific_user: Method Call, DENY Method call will be sent but it won't be received from the 'specific_user'.	**Yes**
Specific user has higher precedence than GUILD or ANY_USER. Sender side.	Sender side: Term 1: ANY_USER: Method call, PROVIDE Term 2: GUILD (guild1) Rules: Method Call, PROVIDE Term 3: Specific_user Rules: Method Call, DENY Receiver side (guild1): Term 1: ANY_USER: Method call, MODIFY Auth. data: Method Call, PROVIDE The destination user belongs to the GUILD and is a specific user. The method call will be denied as specific_user takes precendence.	**Yes**

Specific user has higher precedence than GUILD or ANY_USER. Policy applied on the Sender side.	Sender side: Term 1: ANY_USER: Method call, DENY Term 2: GUILD (guild1) Rules: Method Call, DENY Term 3: Specific_user Rules: Method Call, PROVIDE Receiver side (guild1): Term 1: ANY_USER: Method call, MODIFY Auth. data: Method Call, PROVIDE The destination user belongs to the GUILD and is a specific user. The method call will be sent and received successfully.	**Yes**
GUILD user has higher precedence than ANY_USER. Sender side.	Sender side: Term 1: ANY_USER: Method call, DENY Term 2: GUILD (guild1) Rules: Method Call, PROVIDE Receiver side (guild1): Term 1: ANY_USER: Method call, MODIFY Auth. data: Method Call, PROVIDE The destination user belongs to the GUILD. The method call will be sent and received successfully.	**Yes**
GUILD user has higher precedence than ANY_USER. Sender side.	Sender side: Term 1: ANY_USER: Method call, PROVIDE Term 2: GUILD (guild1) Rules: Method Call, DENY Receiver side (guild1): Term 1: ANY_USER: Method call, MODIFY Auth. data: Method Call, PROVIDE The destination user belongs to the GUILD. The method call will not be sent as the GUILD term has a DENY rule.	**Yes**
Membership. Auth. data has conflicting rules. Deny will take precedence.	Sender side: Term 1: GUILD (guild1) Rules: Method Call, PROVIDE Receiver side (guild1): Term 1: ANY_USER: Method call, MODIFY Auth. data: Method Call, PROVIDE Method Call, DENY	**Yes**

	The destination user belongs to the GUILD. The method call will not be sent as the Auth. data has a DENY rule.	

3.5 Policy addition/removal, membership auth. data and enforcement

Verify that policies can be added and removed dynamically and they will be enforced accordingly.

he test has to be repeated three times for:

Device1	Device 2
Std. Client	Std. Client
Std. Client	Thin Client
Thin Client	Std. Client

Device1 and device2 implement the same method call, property, a signal.

Device 1 makes a method call, get property, set property call. Sending fails as no policies are installed on sender.
Device 1 sends a signal. Sending fails as no policies are installed on sender.

Install the following policies on device 1 and device 2.

Device 1:
Term 1: ANY_USER
Method call PROVIDE
Method call MODIFY
Property PROVIDE
Property MODIFY
Signal OBSERVE
Signal PROVIDE

Device 2:
Term 1: ANY_USER
Method call PROVIDE
Method call MODIFY
Property PROVIDE
Property MODIFY
Signal OBSERVE
Signal PROVIDE

Device 1 makes a method call, get property, set property call. They are successful.
Device 1 sends a signal and it is received successfully by device 2.

Install the following policies on device 1 and device 2.

Device 1:
Term 1: GUILD (guild1)
Method call PROVIDE
Property PROVIDE
Signal OBSERVE

Device 2:
Term 1: GUILD (guild2)
Method call MODIFY
Property MODIFY
Signal PROVIDE

Device 1 makes a method call, get property, sending fails as there is no membership auth. data for the receiver.
Device 1 sends a signal. Sending fails as there is no membership auth. data for the receiver.

Install membership certificate for guild1 on device2.

Device 1:
Method call PROVIDE
Property PROVIDE
Signal OBSERVE

Device 1 makes a method call, get property, sent successfully, but not received as no membership auth. data from the sender.
Device 1 sends a signal, sent successfully, but not received as no membership auth. data from the sender.

Yes

	Install membership certificate for guild2 on device1. Device 2: Method call MODIFY Property MODIFY Signal PROVIDE Device 1 makes a method call, get property, set property call. They are successful. Device 1 sends a signal and it is received successfully by device 2. "RemoveMembership" is called on device 1. Device 1 makes a method call, get property, sent successfully, but not received as no membership auth. data from the sender. Device 1 sends a signal, sent successfully, but not received as no membership auth. data from the sender. "RemoveMembership" is called on device 2. Device 1 makes a method call, get property, sending fails as there is no membership auth. data for the receiver. Device 1 sends a signal. Sending fails as there is no membership auth. data for the receiver. Install the following policies on device 1 and device 2. Device 1: Term1: GUILD (guild1) DENY for all Term 2: ANY_USER DENY for all Term 3: SPECIFIC_USER (Device 2's public key) Method call PROVIDE Property PROVIDE Signal OBSERVE Device 2: Term 1: GUILD (guild1) DENY for all Term 2: ANY_USER DENY for all Term 3: SPECIFIC_USER (Device 1's public key) Method call MODIFY Property MODIFY Signal PROVIDE Device 1 makes a method call, get property, set property call. They are successful. Device 1 sends a signal and it is received successfully by device 2.	

3.6 End-to-End test, Endianness and reboot based test

Test Objective	Procedure and Verification	Applicable for Standard client and Thin client
End-to-end test. This involves one OpenWRT (Big Endian) and 1 Linux (Little Endian) machine. The	OpenWRT runs an application1 that emits a signal. application1 has a membership cert. to emit the signal under GUILD type. Linux machine runs thin client application2 that emits a signal.	No

test also verifies that after reboot, the state of the system does not change as all the policies are stored in the keystore file.	application2 has a membership cert. to emit the signal under GUILD type. Linux machine runs admin app. Admin app claims application1, application2. Admin installs policies on application1, application2. Admin installs membership certificates on application1, application2. Linux machine runs application 3 that receives the signals sent by application1 and application2. application1 and application2 emits signals. Verify that application3 successfully receives both the signals. Reboot the Linux machine and the OpenWRT router. After reboot, Launch application1, application2 and application3. Do not launch admin application. application1 and application2 emits signals. Verify that application3 successfully receives both the signals.	

3.7 Application Manifest

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
Verify the PermissionConfigurator APIs	Load the PermissionConfigurator object using GetPermissionConfigurator() Call SetClaimableState(false); Call GetClaimableState and verify the state is false. Check for PermissionMgmt.Notification.claimable=Unclaimable SetClaimableState(true) Call GetClaimableState and verify the state is true. Check for PermissionMgmt.Notification.claimable=Claimable Call GetSigningPublicKey. Verify that the public key is not set. Call GenerateSigningKeyPair. check for ER_OK. Call GetSigningPublicKey. Verify that the public key is set now and is not empty.	**Yes**
SetManifest and GetManifest	Create a rule: e.g.: MODIFY action for a method call. SetPermissionManifest on the following rule. Call org.allseen.Security.PermissionMgmt.Notification.GetManifest on the above bus. Verify that the manifest fetched has the same rules that was set.	**Yes**

Reset the device bus using PermissionConfigurator	Claim a device bus, Install policies. Install membership certficates, auth data. check for PermissionMgmt.Notification. Check that policy serialNumber and claimed state. Reset the device bus using PermissionConfigurator. Check for PermissionMgmt.Notification. Check that policy serialNumber=0 and claimed state and "unclaimed"	**Yes**
Get the peer public key for an SRP based connection.	Establish a SRP session between two peers. Call GetConnectedPeerPublicKey on the other peer. This should return error. Error should indicate that the connection is not ECDHE_DSA based.	**Yes**
Get the peer public key for an ECDHE_NULL based connection.	Establish a ECDHE_NULL session between two peers. Call GetConnectedPeerPublicKey on the other peer. This should return error. Error should indicate that the connection is not ECDHE_DSA based.	**Yes**
Get the peer public key for an ECDHE_DSA based connection.	Establish a ECDHE_DSA session between two peers. Call GetConnectedPeerPublicKey on the other peer. Verify that the public key of the other peer was fetched successfully..	**Yes**

3.8 Existing security mechanisms test for unclaimed apps.

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
Test SRP mode of authentication between two peers. Will work only if NONE of the devices are claimed.	Two peers are NOT claimed. Peers set up an SRP session between themselves. Peers should be able to send/receive encrypted messages.	No
Test RSA mode of authentication between two peers. Will work only if NONE of the devices are claimed.	Two peers are NOT claimed. Peers set up an RSA session between themselves. Peers should be able to send/receive encrypted messages.	No
Test LOGON mode of authentication between two peers. Will work only if NONE of the devices are claimed.	Two peers are NOT claimed. Peers set up an LOGON session between themselves. Peers should be able to send/receive encrypted messages.	No
Test PINX mode of authentication between two peers. Will work only if NONE of the devices are claimed.	Two peers are NOT claimed. Peers set up an PINX session between themselves. Peers should be able to send/receive encrypted messages.	No
Test SRP mode of authentication between two peers after Claiming them.	Two peers are claimed but policies installed that will allow all mesaages. Peers set up an SRP session between themselves. Peers should NOT be able to send/receive encrypted messages.	No

3.9 Corrupt Keystore (ASC and ATC)

Procedure:

- Admin bus claims device bus, installs policies, installs membership certificates and auth data.
- Device bus is able to send and receive encrypted messages successfully from a peer bus.
- Establish ECDHE_DSA session between admin bus and device bus.
- Establish ECDHE_DSA session between non-admin bus and device bus.
- Add random bytes to the keystore file of the device bus.
- Admin calls "Claim" on device bus. It should return error indicating "invalid or corrupt keystore file"
- Admin calls "InstallPolicy" on device bus. It should return error indicating "invalid or corrupt keystore file"
- Admin calls "RemovePolicy" on device bus. It should return error indicating "invalid or corrupt keystore file"
- Admin calls "Reset" on device bus. It should return error indicating "invalid or corrupt keystore file"
- A non-admin bus calls "Claim" on device bus. It should return error indicating "invalid or corrupt keystore file"
- A non-admin bus calls "InstallPolicy" on device bus. It should return error indicating "invalid or corrupt keystore file"
- A non-admin bus calls "RemovePolicy" on device bus. It should return error indicating "invalid or corrupt keystore file"
- A non-admin bus calls "Reset" on device bus. It should return error indicating "invalid or corrupt keystore file"
- Verify that the device bus cannot send or receive messages to the peer bus. The error should indicate an "invalid or corrupt keystore file"

(The only way to recover is to delete the keystore externally and start all over again.)

3.10 Certificate expiry

Test Objective	**Procedure and Verification**	**Applicable for Standard client and Thin client**
Test Identity certificate expiry	Admin installs Identity certificate on device bus with a validity period of 5 minutes Admin calls installs policy on the device bus. Admin waits for 6 minutes. Admin calls RemovePolicy on the device bus. This should be successful as the peer keys have not expired and hence trust establishment is not trigerred. Admin clears the peer keys Admin calls install policy on the device bus. This should trigger an ECDHE_DSA session which should fail as the device bus identity certificate is not valid.	Yes

Test Membership certificate expiry	Admin installs membership certificate on device bus with a validity period of 5 minutes device bus establishes a secure connection with a peer bus using ECDHE_DSA. It should succeed. device bus sends a signal to peer bus using a GUILD. The signal will be successfully received by peer bus. Device bus tears down the connection. device bus waits for 6 minutes. device bus establishes a secure connection with a peer bus using ECDHE_DSA. It should succeed. (Succeed because the identity certificate is still valid.) device bus sends a signal to peer bus using a GUILD. The signal will be rejected by peer bus because the identity certificate of the bus has expired.	**Yes**