IoTivity Secure Resource Manager
Design Specification and Notes v0.3.4
Q3 2015

Contributing Authors (alphabetically):
Sachin Agrawal
Nathan Heldt-Sheller
Sakthivel Samidurai
Shilpa A Sodani
# Table of Contents

- Revision History .................................................................................................................. 4
- Background .......................................................................................................................... 4
- Secure Resource Manager (SRM) Overview ........................................................................ 5
  - Request Filtering .................................................................................................................. 5
- Secure Virtual Resource Management .................................................................................... 6
  - SRM Sub-Module Composition .......................................................................................... 6
- Resource Manager (RM) ......................................................................................................... 6
- Policy Engine (PE) .................................................................................................................... 7
- Persistent Storage Interface (PSI) ............................................................................................. 7
  - SRM Persistent Storage Degradation and Fallback Behavior ............................................. 8
- Secure Virtual Resource Data Format ....................................................................................... 8
- Policy Engine (PE) Design ........................................................................................................ 8
  - PE Flowchart 1 .................................................................................................................... 9
- SRM Source and Header File Naming ....................................................................................... 10
- SRM Module Sequence Diagrams ........................................................................................... 10
  - Example Access Control List (ACL) flows ......................................................................... 13
- Public SRM API ....................................................................................................................... 17
- Appendix A: DTLS Interaction with SRM .............................................................................. 17
- Appendix B: BeachHead Definition ......................................................................................... 17
  - BeachHead Supported Use Cases ....................................................................................... 17
  - BeachHead Feature List from OIC Security Spec v0.96 errata 1 ....................................... 18
  - BeachHead Milestones and Delivery Schedule .................................................................... 18
- Appendix C – Gap Analysis with BeachHead Implementation vs. OIC Security Spec v0.96 errata 2 ... 19
- Appendix D – BaseCamp Feature Definition .......................................................................... 20
  - Features for Functional Correctness .................................................................................... 20
  - Features for Improved/Extended Functionality planned for BaseCamp ............................. 21
  - Features which Are NOT Slated for BaseCamp, but may be started in this time frame anyway .... 21
  - BaseCamp Milestones and Schedule ................................................................................ 21
### Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>3/4/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Initial revision</td>
</tr>
<tr>
<td>0.1b</td>
<td>3/9/2015</td>
<td>Samidurai, Sakhivel <a href="mailto:sakthivel.samidurai@intel.com">sakthivel.samidurai@intel.com</a>, Agrawal, Sachin (<a href="mailto:sachin.agrawal@intel.com">sachin.agrawal@intel.com</a>)</td>
<td>PSI API definition and diagram RM API definition</td>
</tr>
<tr>
<td>0.1c</td>
<td>3/13/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Misc updates to Design Opens Block Transfer Mode section</td>
</tr>
<tr>
<td>0.2</td>
<td>3/20/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added SRM SVR marshalling design agreement, and PE flowchart 1</td>
</tr>
<tr>
<td>0.2a</td>
<td>3/24/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Minor updates</td>
</tr>
<tr>
<td>0.2b</td>
<td>4/15/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added Appendix B: BeachHead definition and timeline</td>
</tr>
<tr>
<td>0.3</td>
<td>5/21/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added Appendix C: Gap Analysis with Spec v0.96 errata 2</td>
</tr>
<tr>
<td>0.3a</td>
<td>5/22/2015</td>
<td>Agrawal, Sachin (<a href="mailto:sachin.agrawal@intel.com">sachin.agrawal@intel.com</a>)</td>
<td>Updated ‘DTLS interaction with SRM’</td>
</tr>
<tr>
<td>0.3.1</td>
<td>5/28/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added Appendix D: BaseCamp feature definition</td>
</tr>
<tr>
<td>0.3.2</td>
<td>6/15/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added Bootstrap, Provisioning, and AMS sequence diagrams</td>
</tr>
<tr>
<td>0.3.3</td>
<td>6/252015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Elaborated on RM section to clarify SVR DB in memory vs Persistent Store</td>
</tr>
<tr>
<td>0.3.4</td>
<td>7/9/2015</td>
<td>Heldt-Sheller, Nathan <a href="mailto:nathan.heldt-sheller@intel.com">nathan.heldt-sheller@intel.com</a></td>
<td>Added note to document PolicyEngine behavior around /doxm.devOwner and &lt;resource&gt;.Rowner requests</td>
</tr>
</tbody>
</table>

### Background

The reader is assumed to be familiar with the OIC Specification, in particular the Security chapter. See the latest Security Spec at the OIC webpage, for example “OICSecurityTG-SecuritySpec-ProjA_v096_Errata2.docx”.
Secure Resource Manager (SRM) Overview

At a high level, Secure Resource Manager (SRM) has two roles: Request Filtering, and Secure Virtual Resource Management.

Request Filtering
When performing this role, SRM receives a request (e.g. GET, PUT, etc.) from Connectivity Abstraction layer, and either:

1. Grants the request (e.g. a read by an authorized Subject)
2. Denies the request (e.g. a write by an un-authorized Subject)
3. Responds to the request directly (e.g. write to a Secure Virtual Resource such as an Access Control List (ACL))

*Design Note: for the first version, because SRM doesn't know about Resources (outside of its own SVRs), we don't support:*

- Collections (except via a collection-specific URI which is treated as a separate resource)
- Attribute-level access control
Disambiguating Post(CREATE) from Post(PUT)
  o Design Open: This needs to be addressed!

Secure Virtual Resource Management
When performing this role, the SRM manages a database of Secure Virtual Resources, keeping them in memory and persisting them across restarts.

SRM Sub-Module Composition
The SRM has a handful of sub-modules to encapsulate functional blocks. These are the Resource Manager (RM), the Policy Engine (PE) and the Persistent Storage Interface (PSI). Further information on each sub-module can be found in the above-linked chapters dedicated to each module.

Block Transfer Mode Operation
At times, the SRM may need to respond to a Request for a SVR with a data payload that is larger than the UDP packet size, and thus may need to initiate Block Transfer. In this case the SRM will use the RI layer’s BT mode APIs to initiate a Block Transfer, but will then fill the data payload itself.

Design Dependency: this implies that the RI layer BT APIs must be exposed appropriately for this use model.

Also note that Resource Requests that are handled via BT will still be checked for access by the SRM. Once a BT Request is granted, the RI layer should notify the SRM when the BT is complete, so that the SRM can appropriately monitor access request state (e.g. if access should expire during a BT).

Design Dependency: this implies the RI layer should expose an API for the SRM to revoke a previously-granted request. The RI layer should also include a notification to the SRM when the BT is complete.

TODO [SachinA?]: follow up with RI design and ensure that these requirements make sense at the next level of detail, and that they are followed in the RI implementation.

Resource Manager (RM)
The Resource Manager is responsible for:

1. Loading the Secure Virtual Resources to and from persistent storage using the PSI
2. Supplying the Policy Engine (PE) with Resources upon request
3. Responding to requests for SVRs

As mentioned, the RM will load SVRs from persistent storage using the PSI. The RM is responsible for maintaining the privacy and integrity of the SVRs. Therefore it is recommended that the RM encrypt and integrity check the SVRs before storing them to the platform persistent store. In other words, it is suggested that the SVR DB in memory (i.e. the SRM’s “active working set” of SVRs at any given time) be the only plaintext version of the SVR DB kept by the SRM. This will be particularly important if the SRM is hardened using a TEE.
The RM will initially use JSON format to marshal SVR data structures. When CBOR implementation is complete within IoTivity Core, the SRM will migrate to CBOR as a more efficient marshalling format. This choice was agreed upon (as opposed to using a raw binary data format) for a few reasons:

1. It is preferable to use a standard format for marshaled structures, so that an authoring tool, or other policy-controlling application (such as a web-interface) can easily read/write ACLs and other SVRs.
2. Because the oic.sec.* resources (a.k.a. Secure Virtual Resources or SVRs) vary in content and length, it is necessary to have some way of knowing when a given optional and/or empty field is empty so that the marshaling code can properly handle the structs.
   a. Given this issue, the binary format approach would require a “length” field for each data field, and a Variable Length Array approach (where each data field is at the end of a struct and declared e.g. “uint8_p data[]”) forces use to use a struct layout different from the OIC Spec, which makes it very difficult to reconcile against the Spec for correctness.

Policy Engine (PE)
The Policy Engine takes a Request and responds with either “ACCESS_GRANTED” or “ACCESS_DENIED”. It also may include a reason code with the response (see source code for more details). In the nominal request case, the PE will consult the appropriate Access Control List (ACL), find the best Access Control Entry (ACE) that applies to the request, and compare the ACE rights to the Request rights to make a determination.

Finding the best ACE is non-trivial and will be discussed in the Policy Engine (PE) Design chapter.

Persistent Storage Interface (PSI)
The PSI supplies the RM with at least five file APIs:

1. fopen() – opens or creates a file
2. fread() – reads the contents of a file
3. fwrite() – writes to a file
4. fclose() – closes a file
5. unlink() – deletes an existing file

These APIs were chosen to permit maximum flexibility to the PM on managing its own memory and storage utilization.

Because persistent storage is platform (OS and hardware) specific, the Application is required to provide these functions to the PSI. The PSI then exposes them to the RM.

Design Decision: The Application might have supplied the APIs at init-time (e.g. “push” the function ptr structure), or it might have provided them upon request via a pre-defined callback (e.g. “pull”). To stay in keeping with existing App-to-RI interfaces, the “push” model was chosen.
SRM Persistent Storage Degradation and Fallback Behavior
If at any point during SVR load or store operations, the SRM is unable to successfully complete a file operation, the SRM will assume that the Device state has been corrupted. It will reset to a non-provisioned state and await discovery and provisioning as if it were a previously-un-provisioned (e.g. newly deployed) device.

*Design note: We discussed whether a richer fallback behavior was warranted for IoTivity. We decided that OEMs may (and probably will) want to define their own “fallback” behavior. We can’t guess what they might want to do and didn’t want to spend a lot of time developing a rich failure-mode behavior.*

Secure Virtual Resource Data Format
For the database of SVRs (e.g. ACLs, keys, creds, etc.) there will be four categories of formatting.

1. **In Memory** (the SVRs are in use by the SRM) - database in memory will be internal format (e.g. proprietary data structures defined by SRM)
2. **Marshaled** (the SVRs are being prepared for storage via PSI, or perhaps for crossing memory boundaries within the IoTivity Device stack) - database in flight will be marshaled into JSON format. When CBOR is merged into IoTivity Core, JSON will be replaced with CBOR. See Resource Manager (RM) chapter for discussion on this choice.
3. **In Persistent Store** - database in storage will be signed and encrypted with OS or Platform-provided (as opposed to app-provided) wrapping key. Note that database will have already been marshaled prior to being place in storage. In the case of an OS-provided key, the Application layer must not be able to observe the wrapping key.

In **Flight** (traveling off-Device) - database in flight will be signed and encrypted with protocol-specific wrapping key (e.g. DTLS). Note that database will have already been marshaled prior to be sent off-Device.

Policy Engine (PE) Design
The function of the PE is to take a Resource Request from the SRM interface layer (usually being passed from the CA layer, to the SRM, and from there into the PE), and use the ACLs in the SVR database to respond to the Request with either “ACCESS_GRANTED” or “ACCESS_DENIED” (and an optional reason code to explain the denial).

The PE’s primary complexity is finding not just any Access Control Entry (ACE) which matches the request, but rather, to find the **best** ACE to match a given request. For example, if a GET Request for “Resource1” comes in from the CA with SubjectID “SubjectA”, there may be more than one ACE which could apply.

It is assumed that the Policy Author who created the ACLs for this Device would use the principle of Least Privilege, and therefore, that the more specific policy takes precedence. For example it could be that the above Request matches one ACE for SubjectID Anonymous, a second ACE for GroupA to which SubjectA belongs, and a third ACE for SubjectA itself. In this case, the access decision would be made based on the third ACE, as it is the more specific of the three. This would usually be the correct decision,
unless for some reason Least Privilege had been violated, and Anonymous (for example) had great privilege than SubjectA. This is considered an error in Policy Authoring and would be ignored by the PE.

Exceptions to the “more specific policy takes precedence” rule are documented here; however, for the first version of the PE, there are no exceptions to the rule.

Also worth noting is that (per Security Specification behavior) the Device Owner (/doxm.devOwner) has full CRUDN access to all SVRs, and will be automatically granted access on request for any SVR. Similarly, the Resource Owner (e.g. /pstat.Rowner or /cred.Rowner) has full access to the specific resource(s) that is owns.

**PE Flowchart 1**

![Flowchart](chart.png)

*Figure 2 - SRM Policy Engine Flowchart*
Note that if access has expired (due to the duration of the Permission object for example) this will result in an “ACCESS_DENIED | INSUFFICIENT_PERMISSION” response. In the case of an Observer who is initially ACCESS_GRANTED, but then expires, the SRM will send a notification that access is no longer allowed. For example the final message might be “ACCESS_DENIED | INSUFFICIENT_PERMISSION”, or we might add another reason code or access response.

TODO [NathanHS] Implementation Open: should we re-use “ACCESS_DENIED | INSUFFICIENT_PERMISSION” or add a new response code of some sort for Observers whose access expires?

TODO [NathanHS] Update above PE flowchart for allowing ‘device owner’ all access without checking ACL’s.

SRM Source and Header File Naming
Files will be in all lower case and placed in the csdk/security folder (or subfolders therein). Examples include “/src/secureresourcemanager.c”, “/include/secureresourcetypes.h”, etc.

SRM Module Sequence Diagrams

Client-Driven Just-Works Device On-Boarding
In Figure 3, a Provisioning Tool takes ownership of a New Device using Client-Driven mode (i.e. the state advancement is driven by the Provisioning Tool running on the Client Device), and “Just Works” Ownership Transfer Method.
"Just works" ownership transfer method

Client-Driven ACL Provisioning
In Figure 4, a Provisioning Tool installs an /oic/sec/acl resource on a Server Device which it has previously taken ownership of.
**General REST Get/put/delete/post/observe request flow**

In Figure 5, a Client issues a get, put, delete, post or observe request to a Server. Server refers to the ACL database, finds an Access Control Entry (ACE) for the request, and grants/denies the request:
Example Access Control List (ACL) flows

**Resource1 Read request from Anonymous is allowed via ACL**

In Figure 6, Anonymous Client issues a request to Read “Resource1” from Server. Server refers to the ACL database, finds an Access Control Entry (ACE) for Anonymous with R-only permissions, and grants the Read request:
**Resource1 Read request from Anonymous is allowed via ACL**

In Figure 6, Anonymous Client issues a request to Read “Resource1” to Server. Server refers to the ACL database, finds an Access Control Entry (ACE) for Anonymous with R-only permissions, and grants the Read request:

**Resource1 Write request from Anonymous is denied via ACL**

In Figure 7, Anonymous Client issues a request to Write “Resource1” to Server. Server refers to the ACL database, finds an Access Control Entry (ACE) for Anonymous with R-only permissions, and denies the Write request:

**Resource2 Read request from SubjectA is allowed via ACL**

In Figure 8, Client previously identified as SubjectA issues a request to Read “Resource2” from Server. Server refers to the ACL database, finds an Access Control Entry (ACE) for SubjectA with full CRUD permissions, and grants the Read request:
Resource2 Read request from SubjectB is denied via missing ACL

In Figure 9, Client previously identified as SubjectB issues a request to Read “Resource2” from Server. Server refers to the ACL database, cannot find an Access Control List for SubjectB, and denies the Read request:

Server Device refers to Access Management Service (AMS) to determine access rights

In Figure 10, a Server device cannot find a local /acl which corresponds to a resource request, but it does find an /amacl resource, and uses the referenced AMS to resolve the access rights for the resource request.
Figure 10 – SRM refers to AMS for access decision

Note: for clarification, the Figure 10 step which is labeled as “SRM looks up AMACL corresponding to resource “Resource1”, and finds /amacl containing a oic.sec.ams, ...” entails the following steps:

1. SRM calls into /amacl Entity Hander “GetNextAmaclByResource(<R1>)” and gets back a list of /amacl objects whose Resource match the <R1> arg.
2. SRM dereferences the first /amacl.Ams to get a /oic/sec/svc object (the rest of the list are for redundancy)
3. SRM opens the /oic/sec/svc object, and reads the <ServerDeviceld>, <ServerCredId> and <ClientCredId> Properties
4. SRM would then call into /cred EH using method like “GetCredentialById(<ServerCredId>)” and likewise for the <ClientCredId>
5. SRM discovers the <ServerDeviceld> Device using multicast message (I know we have an open here on exactly what the multicast looks like... so there is a TBD in this step)
6. SRM connects to the AMS via the IP, port, and /cred objects from steps 4 & 5
Public SRM API

**SRMRegisterHandler**

Resource Introspection (RI) layer call this API to register ‘request’ and ‘response’ callbacks. When stack is compiled for ‘SECURED’ mode, SRM caches these callback pointers and register SRM’s callback methods (SRMRequestHandler and SRMResponseHandler) with Connection Abstraction (CA) layer via CARegisterHandler API.

**SRMRegisterPersistentStorageHandler**

Design Decision: It was considered whether RI should provide the PSI API to the SRM, or whether the SRM should call the App directly. It was decided that the RI should provide the API to the SRM, so that the RI layer can make use of PSI if needed, and the SRM continues to be disconnected from the App. It was also determined that the performance overhead of this additional layer would be nominal as it would only be incurred at setup time, after which the function ptrs for Persistent Storage would be invoked directly.

**Appendix A: DTLS Interaction with SRM**

DTLS interface in CA layer maintains a mapping of ‘Subject ID’ and ‘IP address + Port number’. This mapping is generated when Iotivity stack initiates a DTLS handshake with another device. ‘SubjectID’ is exchanged between two DTLS end-points along with ‘ClientKeyExchange’ and ‘ServerKeyExchange’ handshake messages. When a REST (GET/PUT/POST/DEL) request is received at CA layer in Server, it looks up this mapping table for the corresponding ‘Subject ID’ and attaches it with the ‘request’. Policy Engine uses this ‘Subject ID’ to look up the corresponding ACL in the ACL resource to make a decision regarding access control for this ‘request’.

**Appendix B: BeachHead Definition**

BeachHead is the codename for the initial functional implementation of SRM and Provisioning Tool scheduled for delivery in Q2 2015.

BeachHead is meant to supply a minimum set of functionality to enable the following high-level Use Cases. Note that for these use cases, we assume a Linux Server Device and a Linux Provisioning Tool. Subsequent versions plan to support Arduino Server Device and Android Provisioning Tool, among others.

**BeachHead Supported Use Cases**

1. Onboarding (aka Take Ownership) of New Server Device
   a. The Provisioning Tool discovers and takes ownership of a newly-deployed Server Device
2. Provision New Device
   a. The Provisioning Tool installs at least one Access Control List (/oic/sec/acl) and one Credential (/oic/sec/cred) to the New Server Device
3. Deny Access Request
   a. The newly-provisioned ACL is used to respond “ACCESS_DENIED” to at least one REST request which is not allowed via the ACL
4. Grant Access Request
   a. The newly-provisioned ACL is used to respond “ACCESS_GRANTED” to at least one REST request which is allowed via the ACL

**BeachHead Feature List from OIC Security Spec v0.96 errata 1**

BeachHead is not intended to be SpecA compliant. BeachHead will not support, among other things, Server Device driven provisioning, or Multi-Service provisioning. BeachHead will also not support a variety of Secure Virtual Resource types. It is more concise to list what will be supported from Spec v0.97:

1. Device and Resource level access control, as described in section 5.1 (but not Group or Property level)
2. OIC Security Onboarding model, as outlined by section 5.2 (see below for supported Owner Transfer methods)
3. OIC Security Provisioning model, as described by section 5.3
4. Create/Read/Update/Delete/Notify (CRUDN) permissions support for Access Control as described in section 6
   a. As mentioned previously in this document, POST(CREATE) and POST(WRITE) can only be disambiguated for Secure Virtual Resources. For application-defined Resources, Create and Update privileges are effectively the same and are equivalent to one another.
5. “Just Works” Owner Transfer method, as described in section 7.2.3
6. Device Owner Transfer Method SVR “/oic/sec/doxm”, as described in section 7.2.4
7. Credential provisioning, as described in section 7.3.3
8. Access Control List (ACL) provisioning, as described in section 7.3.4
9. Provisioning Status SVR “/oic/sec/pstat”, as described in section 7.5
   a. only Operation Mode 0b11 is supported, i.e. Client-driven, Single-service
   b. CommitHash will be placeholder, always computed to 0x0 so that it’s always “correct” and passes the comparison test
10. Session Protection with DTLS, as described in section 8
    a. TLS-PSK-AES-CCM-128 ciphersuite, as described in section 8.1.1
11. Device Credential SVR “/oic/sec/cred”, as described in section 8.2.1
12. Simple Access Control, as described in section 9
13. Local Access Control SVR “/oic/sec/acl”, as described in section 9.4
14. ACL Evaluation, as described in section 9.6

In addition, the BeachHead “Provisioning Tool” will support the onboarding and provisioning flow as described in “Provisioning Flow for Beach Head version v0.1.pptx” (which is being added to the next version of the OIC Security Spec, as well).

**BeachHead Milestones and Delivery Schedule**

BeachHead implementation has begun already as of 4/15/2015. The **code-complete milestone is April 24th**, which includes:
1. All SRM and related code (including unit tests) pushed to Gerrit, peer-reviewed, and merged to “security-M3” branch (Intel deliverable)
2. All Provisioning Tool and related code (including unit tests) pushed to Gerrit, peer-reviewed, and merged to “security-M3” branch (Samsung deliverable)

The end-to-end functional milestone is scheduled for May 5th, which includes integration of #1 and #2 from code-complete milestone, and the ability to demonstrate all use cases defined in BeachHead Supported Use Cases above.

Appendix C – Gap Analysis with BeachHead Implementation vs. OIC Security Spec v0.96 errata 2

The following list of gaps have been identified between the BeachHead implementation of IoTivity, and the OIC Security Spec v0.96 errata 2. The page and line number are listed, along with at some times a section reference. Gaps without a particular page/line are listed as “Spec (general)” and come from the v0.96 errata 2 Spec in one or more places. Additional gaps have been included from speculative Spec B analysis and are marked “Spec B”.

<table>
<thead>
<tr>
<th>Title</th>
<th>Requirement Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add /oic/sec/am acl to supported Resource Types (including Entity Handler)</td>
<td>p.14#23 (5.4)</td>
</tr>
<tr>
<td>Add /oic/sec/sa cl to supported Resource Types (including Entity Handler)</td>
<td>p.15#1 (5.4)</td>
</tr>
<tr>
<td>Add /oic/sec/svc to supported Resource Types (including Entity Handler)</td>
<td>p.15#3 (5.4)</td>
</tr>
<tr>
<td>Update all SVRs to use /oic/sec/svc Resource Types where appropriate</td>
<td>Spec (general)</td>
</tr>
<tr>
<td>Complete missing elements in /acl, /cred, /doxm and /pstat</td>
<td>Spec (general)</td>
</tr>
<tr>
<td>Review SVR update policy for supported provisioning and oxm types</td>
<td>Spec (general)</td>
</tr>
<tr>
<td>Add Delete support to SRM Entity Handlers</td>
<td>Spec (general)</td>
</tr>
<tr>
<td>Add /ams logic to SRM</td>
<td>Spec (general)</td>
</tr>
<tr>
<td>Implement Pre-provisioned Device PIN ownership transfer method</td>
<td>p18#26</td>
</tr>
<tr>
<td>Add &quot;role&quot; support to SRM</td>
<td>p41#4 (8.1.3)</td>
</tr>
<tr>
<td>Dynamic key provisioning and Pair-wise Key Generation</td>
<td>p43#11 (8.3)</td>
</tr>
<tr>
<td>Add Fluffy Key Mgmt support for Device-&gt;Device (pairwise) to SRM</td>
<td>Spec B</td>
</tr>
<tr>
<td>Add Fluffy Key Mgmt support for GSKs to SRM</td>
<td>Spec B</td>
</tr>
<tr>
<td>Implement Mode Switch ownership transfer method</td>
<td>p18#26</td>
</tr>
<tr>
<td>Implement Random Device PIN ownership transfer method</td>
<td>p18#26</td>
</tr>
<tr>
<td>256-bit key support</td>
<td>p19#32</td>
</tr>
<tr>
<td>JustWorks Cipher Suite: TLS_ECDH_ANON_WITH_AES_256_CBC_SHA</td>
<td>p20#10</td>
</tr>
<tr>
<td>JustWorks Cipher Suite: TLS_ECDH_ANON_WITH_AES_128_CBC_SHA256</td>
<td>p21#1</td>
</tr>
<tr>
<td>JustWorks Cipher Suite: TLS_ECDH_ANON_WITH_AES_256_CBC_SHA256</td>
<td>p21#2</td>
</tr>
</tbody>
</table>
Device Class 0-2 OPTIONAL Cipher Suite:
TLS_ECDH_ANON_WITH_AES_128_CBC_SHA256 p21#9

Device Class 0-2 OPTIONAL Cipher Suite: TLS_ECDH_ANON_WITH_AES_256_CBC_SHA p21#10

Device Class 0-2 OPTIONAL Cipher Suite:
TLS_ECDH_ANON_WITH_AES_256_CBC_SHA256 p21#11

Device Class 3+ and higher MANDATORY Cipher Suite:
TLS_ECDH_ANON_WITH_AES_128_CBC_SHA256 p21#13

Device Class 3+ and higher MANDATORY Cipher Suite:
TLS_ECDH_ANON_WITH_AES_256_CBC_SHA p21#14

Device Class 3+ and higher MANDATORY Cipher Suite:
TLS_ECDH_ANON_WITH_AES_256_CBC_SHA256 p21#15

Leverage Exposed Platform-hardened Key/Cred/etc storage p40#3

Encrypt PSI storage (general)
Implement CBOR support in SRM Spec B
Add Fluffy RA support to SRM Spec B
Binding between device context and platform implementing device p.11#15 (5.3)
Permission check at time of issuing Slow Response or Observe Notification Spec (general)
Apply policy to presence notifications Spec (general)

Appendix D – BaseCamp Feature Definition
BaseCamp is the next major release of IoTivity Security slated for end of July 2015, and is meant to close the remaining “must-have” gaps between Error! Reference source not found. and OIC Security Spec 0.96 errata 2 (the latest official OIC Spec version at the time of BaseCamp definition).

To close the gaps identified in Error! Reference source not found., BaseCamp will implement the following features:

Features for Functional Correctness
1. Add /oic/sec/amacl to supported Resource Types (including Entity Handler)
   a. Source: Spec p.14#23 (5.4)
2. Add /oic/sec/sacl to supported Resource Types (including Entity Handler)
   a. Source: Spec p.15#1 (5.4)
3. Add /oic/sec/svc to supported Resource Types (including Entity Handler)
   a. Source: Spec p.15#3 (5.4)
4. Update all SVRs to use /oic/sec/svc Resource Types where appropriate
   a. Source: Spec general
5. Complete missing fields in /acl
   a. Source: Spec general
6. Complete missing fields in /cred
   a. Source: Spec general
7. Complete missing fields in /doxm
   a. Source: Spec general
8. Complete missing fields in /pstat
   a. Source: Spec general
9. Review SVR update policy for supported provisioning and oxm types
   a. Source: Spec general
10. Add DELETE support to SRM Entity Handlers
    a. Source: Spec general
11. Add /ams logic to SRM
12. Source: Spec general

Features for Improved/Extended Functionality planned for BaseCamp

1. Implement Pre-provisioned Device PIN ownership transfer method
   a. Source: Spec p18#26
2. <TBD> on completion of OSWG Security TG discussions on Spec B features for BaseCamp

Features which Are NOT Slated for BaseCamp, but may be started in this time frame anyway
There are a handful of high priority “Advanced Functionality” or Spec B features, which are currently viewed as critical for the next version after BaseCamp, and which may be complex enough that they should be started during BaseCamp dev cycle.

1. Add Fluffy Key Mgmt support for Pairwise Keys to SRM
   a. Source: Spec B
2. Integrate CBOR into SRM
   a. Source: Spec B

BaseCamp Milestones and Schedule
BaseCamp is slated to be feature complete and merged with IoTivity Master by July 10th. This leaves a few weeks for Integration/Testing before the IoTivity 0.9.2 release date of end of July.