Secure Enterprise Services Consumption for SaaS Technology Platforms

Yuecel Karabulut, Ike Nassi

SAP Research Center Palo Alto
USA

yuecel.karabulut@sap.com
ike.nassi@sap.com

***Abstract*— Over recent years there has been increased level of discussion on utility pricing for software. The focus of these discussions is to create new operating cost models where the unit costs are directly tied to the business operations to which they contribute. While creating a fine-grained operating cost model is very important for software solutions such as SaaS, the anticipated technology platforms will need to rely on a set of security mechanisms in order to provide a secure and trustworthy service consumption environment. We present an architecture for secure enterprise services consumption management system and a protocol for secure service consumption for service-oriented technology platforms. Our approach is performance sensitive and utilizes a novel combination of asymmetric and symmetric cryptography, and capability based access control. Access to technology platform services is regulated based on the permissions encoded in cryptographic capability tokens. In this paper we report a work in progress.**

I. Introduction

Today, software is increasingly being delivered on the SOA model. SOA represents a model in which functionality is decomposed into services, which can be distributed over a network and can be combined together and reused to create (composite) applications. Current license-based pricing (perpetual licensing) for software is mainly based on per processor or per registered user. Changes in technology (e.g. SOA) and in usage (e.g. Internet-based vs. intranet-based) are causing software vendors to look at other models for charging for their software. In particular, software-as-a-service (SaaS) customers are putting increased pressure on application vendors to come up with more flexible pricing models that meet their business needs. Over recent years there has been increased level of discussion on utility pricing for software. The focus of these discussions is to create new operating cost models where the unit costs are directly tied to the business operations to which they contribute. Simply put, customers would pay for what they use. For instance, business software vendor would set his utility price on a per-enterprise service or business process basis, or a search engine vendor might base his pricing on a per-search basis. While creating a fine-grained operating cost model is very important, the real world implementations of technology platforms offering such models will require a set of security mechanisms in order to create a secure and trustworthy service consumption environment.

We argue that current service-oriented technology platforms used to build Software-as-a-Service (SaaS) applications need to be extended with a secure enterprise services consumption management system which will include at least following key components: Metering and Billing Service, Configuration Service, License Token Service and Security Token Service. In this document we present an architecture and a protocol for secure service consumption. Our approach utilizes a combination of asymmetric and symmetric cryptography, and capability based access control [9][10][11]. Access to platform services is regulated based on the capabilities encoded in cryptographic capability tokens [11].

The rest of the document is structured as follows. Section 2 discusses the problem domain. In Section 3, we outline the high-level architecture. Section 4 discusses an overview and detailed view of the secure service consumption protocol. In section 5, we discuss performance and security related issues. Finally, section 6 concludes the paper.

II. Problem Domain And Requirements

SaaS applications take advantage of the benefits of centralization through a single-instance, multi-tenant architecture. The emergence of SaaS as an effective software-delivery mechanism will create many opportunities. As a SaaS application is provided as a hosted service and accessed over the Internet, a SaaS offering will need to have a set of security mechanisms to keep sensitive data safe in transmission and storage. Most of SaaS security discussions focussed on the tenant data isolation and data encryption [13]. While providing a secure data isolation approach is the key for the success of SaaS solutions, it is also crucial to provide a security solution for tenant authentication, authorization and message level data confidentiality.

More concretely, we consider following fundamental security requirement: The *authorizers (i.e. the SaaS provider)* autonomously follow a security policy which ensures that *requested service* is delivered only to appropriate *requestors (i.e. the SaaS consumer)*. In order to achieve this goal, requestors have to provide evidence that they are eligible for requested service, and authorizers have to maintain mechanisms to inspect such evidence and to decide whether and which services are performed. Furthermore, an authorizer has to ensure that service is actually usable to only that requestor which provided the appropriate credentials.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.150

IEEE computer society

III. HIGH-LEVEL ARCHITECTURE

We assume at least following main actors in a SaaS environment:

- SC – Service Consumer
- SP – Service Provider

For generality we assume that SP provides and maintains an Enterprise Service Repository (ESR), a Service Registry (SR), and corresponding service implementations in different Backend Systems (BS). ESR is a central repository in which service interfaces and enterprise services are modeled and their metadata is stored. SR constitutes yellow pages of services. It supports publishing, classifying and discovering services.

SP hosts various BSs and a central Enterprise Services Consumption Management System (ESCOMAS). It includes following key components, as illustrated in Figure 1:

- CS – Configuration Service: Supports the configuration of contracts, tariff plans and pricing rules. The CS is used by the Service Level Agreement (SLA) administrator.
- MBS – Metering and Billing Service: Calculates charges and associates each charge with an account.
- LTS – License Token Service: Provides functionality for issuance and validation of cryptographic license tokens.
- STS – Security Token Service: Provides functionality for issuance, exchange, and validation of cryptographic tokens.

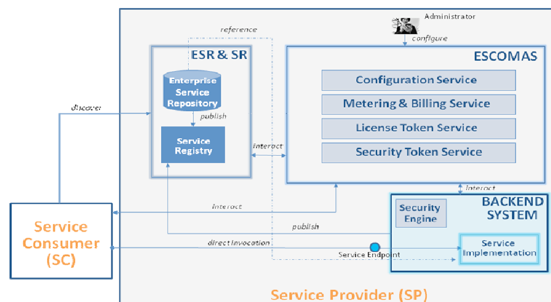

Figure 1. High-level ESCOMAS Architecture

The BS hosts service implementations and a security engine which is responsible for handling BS's all security related operations such as access management and encryption.

We make following assumptions regarding the underlying security infrastructure:

- Each entity (SC, LTS, STS and BS) is represented by (one of) its public key(s)[1].
- SC, LTS, STS and BSs have the required security mechanisms which provide asymmetric and symmetric cryptography operations.
- STS and BSs are able to grant capability tokens and to provide other operations associated with the processing of capability tokens.
- Each BS follows a security policy that is expressed in terms of capabilities.

[1] It is often thought best to use separate asymmetric key pairs for encrypting and signing. For the sake of simple presentation, we assume that each entity uses the same key pair for encrypting and signing. A commercial implementation of our proposal could use separate key pairs for the mentioned purposes.

IV. SECURE ENTERPRISE SERVICES CONSUMPTION PROTOCOL

In this section we present the Secure Enterprise Services Consumption (SESCO) protocol. As discussed in Section 1, we use a combination of asymmetric and symmetric cryptography. Due to performance reasons we limit the use of asymmetric cryptography and mainly use faster symmetric cryptography. Asymmetric cryptography is only used for initial authentication step and Enterprise Service request step, as discussed below.

A. *Notation*

SC: Service Consumer
SC: The id of the Service Consumer
SP: Service Provider
SR: Service Registry
LTS: License Token Service
LTS: The id of the License Token Service
STS: Security Token Service
STS: The id of the Security Token Service
BS: Backend System
BS: The id of the Backend System
ES: Enterprise Service
ES: The id of the requested Enterprise Service
CA: Certification Authority
CA: The id of the Certification Authority
MBS: Service Metering and Billing Service
LN: Unique Service Consumer License Number
LT: License Token
CT: Capability Token
DT: Delegation Token
($\mathbf{PubK_P}$, $\mathbf{PrivK_P}$): Public/Private key pair of the principal P
$\mathbf{PubKCert_P}$: Public key certificate of the principal P.
$\mathbf{SecK_{P1\text{-}P2}}$: Symmetric key shared by the principals P1 and P2
$\{m\}\mathbf{SecK_{P1\text{-}P2}}$: The encryption of message m with symmetric key $\mathbf{SecK_{P1\text{-}P2}}$ which is shared by the principals P1 and P2.
$\{\{m\}\}\mathbf{PubK_P}$: The encryption of message m with public key $\mathbf{PubK_P}$
$[m]\mathbf{PrivK_P}$: The digital signature of message m with private key $\mathbf{PrivK_P}$ which belongs to principal P.

B. *Basic SESCO Protocol Overview*

The basic SESCO protocol consists of following phases (see Figure 2):

- Capability delegation phase (BS → STS)
- Authentication & License Token request phase (SC → LTS)
- License Token delivery phase (LTS → SC)
- Capability Token request phase (SC → STS)

- Capability Token delivery phase (STS → SC)
- Enterprise Service request phase (SC → BS)
- Service Result delivery phase (BS → SC)
- Service Metering phase (BS → MBS)

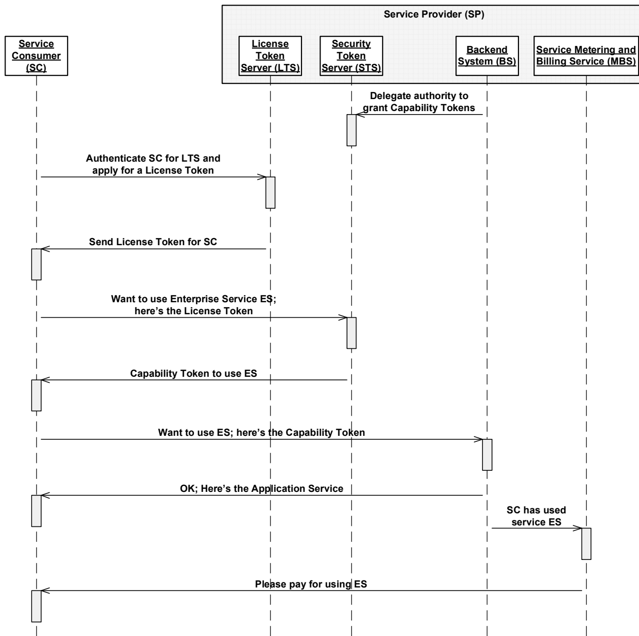

Figure 2. Basic SESCO Protocol Interactions

Our protocol assumes that SC and SP communicated over a secure channel and signed a SLA which includes the contractual details (e.g. subscription model) regarding the usage of Enterprise Services.

Each time a Backend System BS registers an Enterprise Service at the Service Registry, the BS issues a Delegation Token to the Security Token Service STS. The Delegation Token grants the STS the authority to delegate the encoded capabilities to eligible Service Consumers. The Delegation Token is signed by the private key of the Backend System. Instead of issuing Delegation Tokens for individual Enterprise Services belonging to BS, the BS can also issue a collective Delegation Token which encodes a collection of registered Enterprise Services. An Enterprise Service is uniquely identified by its URL.

The Service Consumer's goal is to be able to access Enterprise Services provided by Backend Systems. For this purpose a Service Consumer needs to prove to a Backend System that Service Consumer owns capabilities which authorize him to access requested Enterprise Services. This is done by obtaining a License Token from a License Token Service and then presenting this to a Security Token Service in order to obtain a service specific "Capability Token", the credential that Service Consumer uses to prove Backend System that Service Consumer is eligible to access an Enterprise Service. The Service Consumer can now use his Capability Token and invoke Enterprise Services, which are encoded in his Capability Token. Upon receiving the Capability Token and other required data (such as an encrypted timestamp as an authenticator) from the Service Consumer, the Backend System verifies the received Capability Token and these data. In the positive case, the Backend System executes the requested Enterprise Service and sends the computed result back to the Service Consumer. As shown below, in our protocol, any interaction between two principals is encrypted with the symmetric session keys shared between those principals.

A License Token encapsulates a symmetric key (Service Consumer session key) intended for secure communication between Service Consumer and Security Token Service when applying for service specific capability tokens. The License Token also includes other information such as Service Consumer's unique id. The contents of the License Token are encrypted with a symmetric key shared between the Security Token Service and the issuing License Token Service.

A Capability Token is a digitally signed credential that expresses that the owner of the holder public key has a possibly conditional permission to access an Enterprise Service within a given validity period.

C. *Detailed View of the SESCO Protocol*

In this section we provide an elaborated view of each protocol phase discussed above and depicted in Figure 2.

In the SESCO protocol, SC, LTS, STS and BS possess independent public/private key pairs, ($\mathbf{PubK_{SC}}$, $\mathbf{PrivK_{SC}}$), ($\mathbf{PubK_{LTS}}$, $\mathbf{PrivK_{LTS}}$), ($\mathbf{PubK_{STS}}$, $\mathbf{PrivK_{STS}}$), and ($\mathbf{PubK_{BS}}$, $\mathbf{PrivK_{BS}}$), respectively. We assume that at least the principals SC and LTS hold public key certificates $\mathbf{PubKCert_{SC}}$ and $\mathbf{PubKCert_{LTS}}$ issued by certification authorities $\mathbf{CA_1}$ and $\mathbf{CA_2}$, respectively. These certificates are used to testify the binding between each principal (i.e. SC and LTS) and its purported public key. In our protocol, we assume that digital signatures are unforgeable.

We assume that LTS and STS already share a symmetric secret key. We also assume that STS and each BS share a symmetric secret key, and exchanged each other public key over a secure channel. As LTS, STS and BSs are hosted in the same SP's security realm, we assume that there is symmetric key distribution infrastructure within this security realm.

1) *Capability delegation phase*

In this phase, as illustrated in Figure 3, the BS grants a delegation token to the STS, the content of which roughly means "STS can speak for the issuing BS, the owner of the Enterprise Service". For the sake of simplicity we assume that STS and BS already exchanged each other's public key over a secure channel. The backend system BS generates a timestamp [2] $\mathbf{tstamp_{BS}}$ with his local clock and issues a delegation token $\mathbf{DT_{BS\text{-}STS}}$ to STS. The BS encrypts the

[2] To prevent replay attacks, steps must be taken to ensure the freshness of messages. Common techniques include using nonces and/or timestamps. In order to use timestamps, the receivers's clock and sender's clock need to be fairly synchronized. We also make an assumption about fairly synchronized clocks when validity time periods are specified in tokens.

sequence of BS's id **BS** and $\mathbf{tstamp_{BS}}$ with the symmetric key $\mathbf{SecK_{STS\text{-}BS}}$ shared between STS and BS. This sequence will be used as authenticator by the STS. The BS also encrypts the delegation token with $\mathbf{SecK_{STS\text{-}BS}}$. Th BS then sends both encrypted message parts to STS. The delegation token $\mathbf{DT_{BS\text{-}STS}}$, signed by BS's private key $\mathbf{PrivK_{BS}}$, consists of following fields:

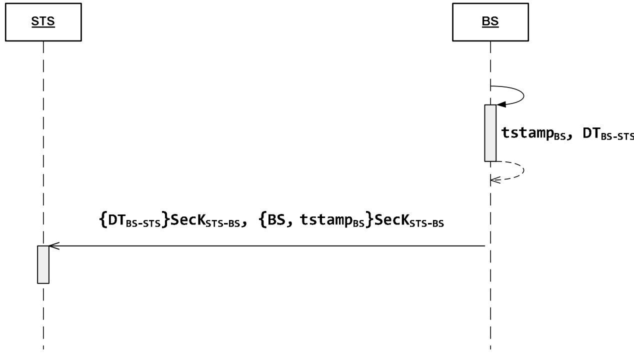

Figure 3. Capability delegation phase

- **Issuer** The BS public key $\mathbf{PubK_{BS}}$ that is delegating the authority to grant authorization to eligible service consumers.
- **Holder** The STS public key $\mathbf{PubK_{STS}}$ that is receiving the authority to grant authorization to eligible service consumers.
- **Capabilities** A set of unique identifiers, each of which represents the unique address (e.g. URL) of an Enterprise Service whose implementation is hosted by the backend system.
- **Validity** A time period during which the holder is allowed to grant authorizations to eligible service consumers.

2) Authentication & License Token request phase delivery phase

In the Authentication & License Token request and delivery phases the principals SC and LTS perform following steps (see Figure 4):

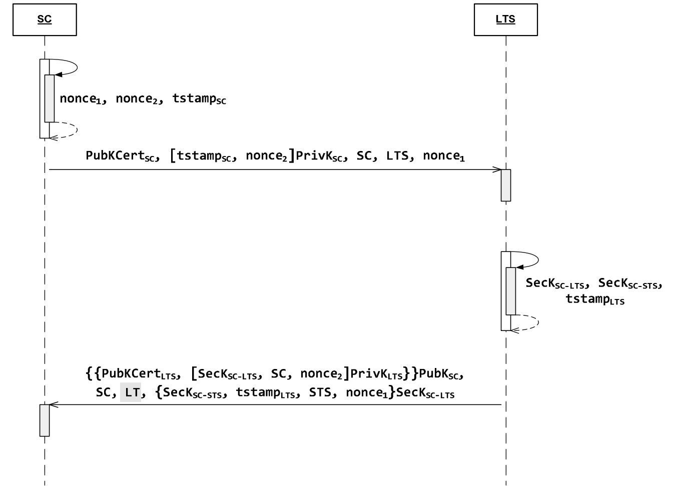

Figure 4. Authentication & License Token request and delivery phase

- The service consumer SC generates two nonces $\mathbf{nonce_1}$, $\mathbf{nonce_2}$ and a timestamp $\mathbf{tstamp_{SC}}$.
- SC then sends a request <$\mathbf{PubKCert_{SC}, [tstamp_{SC}, nonce_2]PrivK_{SC}, SC, LTS, nonce_1}$> to LTS and applies for a License Token LT. The request consists of following parts:
  - $\mathbf{PubKCert_{SC}}$: SC's public key certificate
  - $\mathbf{[tstamp_{SC}, nonce_2]PrivK_{SC}}$: SC's local time $\mathbf{tstamp_{SC}}$ and a nonce $\mathbf{nonce_2}$, signed by SC's private key $\mathbf{PrivK_{SC}}$.
  - **SC**: SC's id
  - **LTS**: LTS's id
  - $\mathbf{nonce_1}$: a nonce generated by SC
- The LTS responds by generating a fresh key $\mathbf{SecK_{SC\text{-}LTS}}$ for use between SC and LTS, and another fresh key $\mathbf{SecK_{SC\text{-}STS}}$ for use between SC and STS, and a timestamp $\mathbf{tstamp_{LTS}}$, containing LTS's local time. The LTS then sends the following response to SC: <$\mathbf{\{\{PubKCert_{LTS}, [SecK_{SC\text{-}LTS}, SC, nonce_1]PrivK_{LTS}\}\}PubK_{SC}, SC, LT, \{SecK_{SC\text{-}STS}, tstamp_{LTS}, STS, nonce_1\}SecK_{SC\text{-}LTS}}$>**.** The response consists of following parts:
  - $\mathbf{\{\{PubKCert_{LTS}, [SecK_{SC\text{-}LTS}, SC, nonce_i]PrivK_{LTS}\}\}PubK_{SC}}$: LTS builds the sequence $\mathbf{SecK_{SC\text{-}LTS}}$, **SC**, $\mathbf{nonce_2}$ and signs this sequence with his private key $\mathbf{PrivK_{LTS}}$. The LTS then encrypts a message with the public key of LTS, where the message contains LTS's public key certificate $\mathbf{PubKCert_{LTS}}$ and the signature of the signed sequence in the previous step. This part of LTS's response guarantees integrity protection as SC's name **SC** and $\mathbf{nonce_2}$ (which was generated by SC) appears inside a component signed by LTS. This defends

against a man-in-the-middle attack since SC can verify that the LTS generated the received License Token for SC and not for another principal. The presence of the nonce **$nonce_1$** uniquely identifies which of the requests of SC this reply corresponds to. Alternatively, the LTS could have also included **$nonce_2$** in the signed sequence. Important here is to include a nonce generated by SC.

   - **SC**: SC's name
   - **LT**: License Token issued by LTS for SC and encrypted with a long-term symmetric key **$SecK_{LTS-STS}$** shared between LTS and STS. **LT** contains SC's id (name) **SC**, SC's network address, SC's license unique license number, service subscription type (e.g. annual, monthly, one-time), license validity period, and the SC-STS session key **$SecK_{SC-STS}$**.
   - **{$SecK_{SC-STS}$, $tstamp_{LTS}$, STS, $nonce_1$}$SecK_{SC-LTS}$**: This part contains the session key **$SecK_{SC-STS}$** for use between SC and STS, LTS's local time **$tstamp_{LTS}$**, the name of the token service **STS**, and **$nonce_1$** generated by SC, and encrypted with the symmetric key **$SecK_{SC-LTS}$** shared between SC and LTS for message confidentiality.

3) Capability Token request and delivery phase

In the Capability Token request and delivery phases the principals SC and STS perform following steps (see Figure 5):

- The service consumer SC browses [3] the Service Registry and retrieves [4] the URL of the Enterprise Service, represented by **ES**.
- The service consumer SC generates a nonce **$nonce_3$** and a timestamp **$tstamp_{SC}$**.
- SC sends a request **<LT, {SC, $tstamp_{SC}$}$SecK_{SC-STS}$, SC, ES, $nonce_3$>** to STS and applies for a Capability Token which grants SC the authorization to invoke an Enterprise Service represented by **ES**. SC's request is composed of following parts:
   - **LT**: SC's License Token issued by LTS.
   - **{SC, $tstamp_{SC}$}$SecK_{SC-STS}$**: SC's name **SC** and local time **$tstamp_{SC}$** encrypted with the session key **$SecK_{SC-STS}$**.
   - **SC**: SC's name
   - **ES**: The id of the requested Enterprise Service
   - **$nonce_3$**: a nonce generated by SC
- Upon receiving SC's request, the STS decrypts **LT** using the **$SecK_{LTS-STS}$** secret key. This gives STS the SC-STS session key **$SecK_{SC-STS}$**.Using this key the STS decrypts the message part **{SC, $tstamp_{SC}$}$SecK_{SC-STS}$** and retrieves the combination of **SC** and **$tstamp_{SC}$**, which is the authenticator part of SC's request. The STS grants a Capability Token **CT_{STS-SC}**. The STS then sends the following response to the SC: **<SC, {$SecK_{SC-BS}$, DT_{BS-STS}, CT_{STS-SC}}$SecK_{STS-BS}$, {$SecK_{SC-BS}$, ES, $nonce_3$}$SecK_{SC-STS}$>**. This response is composed of the following message parts:

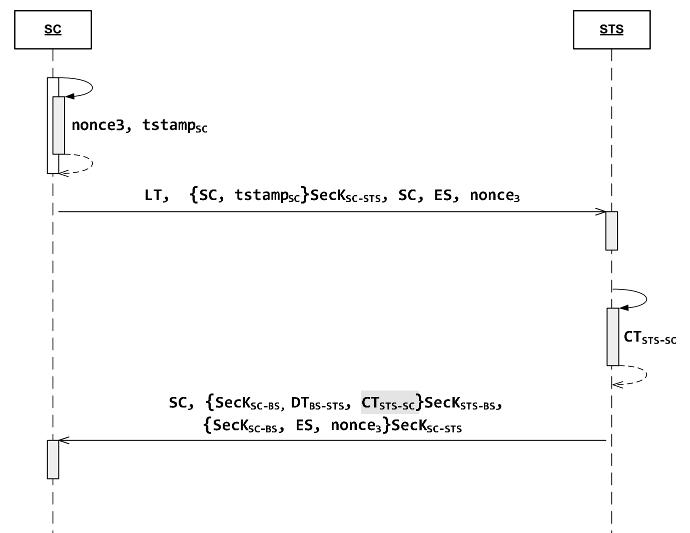

Figure 5. Capability Token request and delivery phase

   - **SC**: SC's id
   - **{$SecK_{SC-BS}$, DT_{BS-STS}, CT_{STS-SC}}$SecK_{STS-BS}$**: SC-BS session key **$SecK_{SC-BS}$**, a Capability Token and a Delegation Token, encrypted with the symmetric key **$SecK_{STS-BS}$** shared between STS and BS. The Capability Token **CT_{STS-SC}** expresses a possibly conditional permission to access an enterprise service. The **CT_{STS-SC}** is signed by STS using **$PrivK_{STS}$** and includes following fields:
      - **Issuer:** The public key **$PubK_{STS}$** of the issuing STS.
      - **Holder:** The public key **$PubK_{SC}$** of the SC (which now acts as the holder of the capability).
      - **Capabilities:** The id **ES** of the granted enterprise service.
      - **Validity period:** A time period during which the holder is allowed

[3] Note that this step is most probably automated by using a service discovery process.
[4] If a selected service constitutes a composed service consisting of at least two services, only the URL of the composed service will be included into SC's service shopping cart.

to access the granted enterprise service.

- **Service Consumer Id:** The id of the SC.
- **Service Consumer network address:** The network address of the SC.
- **Service Consumer license number:** The unique service consumer license number **LN**.

- **{SecK$_{SC-BS}$, ES, nonce$_3$}SecK$_{SC-STS}$**: SC-BS session key, the id of the granted Enterprise Service and a nonce, encrypted with the session key **SecK$_{SC-STS}$**.

4) Enterprise Service request phase and Service Result delivery phase

Upon receiving the response in the Capability Token delivery phase, the SC has now enough information to authenticate itself to BS and prove that he holds a capability which authorizes him to access an enterprise service. In this phase SC and BS perform following steps (see Figure 6):

- The service consumer SC generates a timestamp **tstamp$_{SC}$**.
- The SC then sends the following request to BS: <**{SecK$_{SC-BS}$, DT$_{BS-STS}$, CT$_{STS-SC}$}SecK$_{STS-BS}$,{SC, tstamp$_{SC}$}SecK$_{SC-BS}$**>. The first part of the message is from the previous phase (see Figure 5) and contains the SC-BS session key **SecK$_{SC-BS}$**, the Delegation Token **DT$_{BS-STS}$** and the Capability Token **CT$_{STS-SC}$**, encrypted with the symmetric key **SecK$_{STS-BS}$** shared between STS and BS. The second part of the message is an authenticator which contains SC's id **SC** and a timestamp **tstamp$_{SC}$**, encrypted with the session key **SecK$_{SC-BS}$**.
  The BS decrypts the first part of the message from the previous step using the symmetric key **SecK$_{STS-BS}$** to retrieve the SC-BS session key **SecK$_{SC-BS}$**, the delegation token **DT$_{BS-STS}$** and the capability token **CT$_{STS-SC}$**. To determine whether SC's request for a protected enterprise service should be honored, the BS verifies the request's "proof of authenticity" and "proof of authorization". The proof of authenticity is the authenticator which contains SC's id **SC** and a timestamp **tstamp$_{SC}$** encrypted with the session key **SecK$_{SC-BS}$**. The proof of authorization is a token chain which consists of **DT$_{BS-STS}$** and **CT$_{STS-SC}$**. To verify the proof of authorization, the BS checks whether the token chain constitutes a chain of authorization originating from the BS himself. For this purpose the BS performs following steps: The BS verifies the signature and validity period of the Delegation Token. After a positive evaluation of the Delegation Token, the BS then checks whether **CT$_{STS-SC}$** is signed by an issuer who is also the holder of the **DT$_{BS-STS}$**. In the positive case the BS checks whether the capability "to invoke the enterprise service **ES**" is included in the capabilities encoded in the delegation token **DT$_{BS-STS}$** and whether the validity period in the capability token **CT$_{STS-SC}$** is within the range of the validity period contained in the delegation token. In the positive case the BS issues a new capability token **CT$_{BS-SC}$** signed by the private key **PrivK$_{BS}$** of the BS. The new capability token includes following fields:
  - **Issuer:** The public key **PubK$_{BS}$** of the BS
  - **Holder:** The public key **PubK$_{SC}$** of the SC
  - **Capabilities:** The id **ES** of the granted enterprise service. This is calculated by computing the intersection of the capabilities included in the delegation token and the capabilities contained in the previous capability token **CT$_{STS-SC}$**.

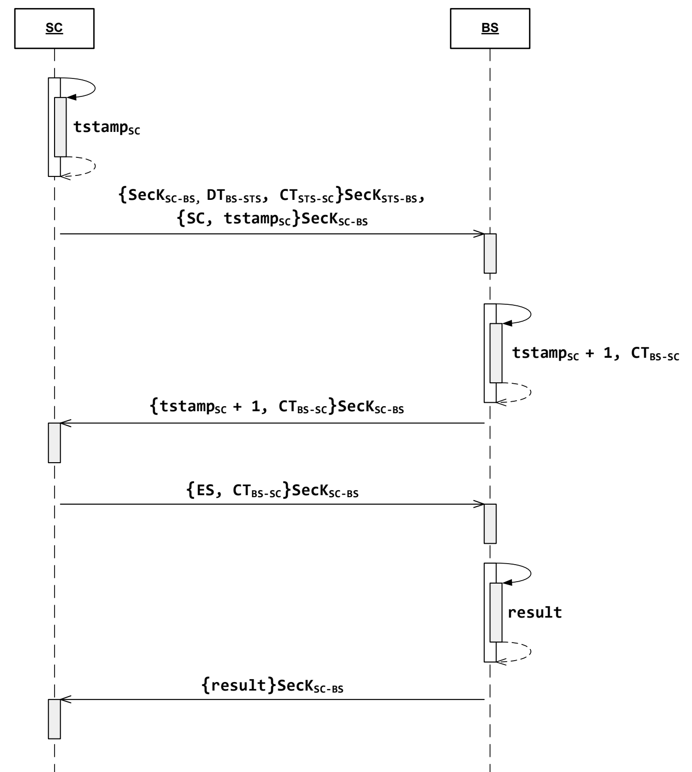

Figure 6. Enterprise Service request phase and Service Result delivery phase

  - **Validity period:** A time period during which the holder is allowed to access the granted enterprise service. This validity period is calculated by computing the intersection of validity periods contained in the delegation token and the validity period contained in the previous capability token **CT$_{STS-SC}$**.
  - **Delegation bit:** If this bit is set to **true**, this means that the Service Consumer is

authorized to delegate the encoded capabilities to another principal. Otherwise, the Capability Token can only be used by the Service Consumer which possesses the private key corresponding to the encoded public key $\mathbf{PubK_{SC}}$. The default value of this bit is **false**.

- The BS sends the new capability token $\mathbf{CT_{BS\text{-}SC}}$ and the timestamp $\mathbf{tstamp_{SC}}$ found in the SC's authenticator plus 1, encrypted with the session key $\mathbf{SecK_{SC\text{-}BS}}$. With this response the BS grants authorization via capability token $\mathbf{CT_{BS\text{-}SC}}$, confirms his true identity and willingness to serve the SC.
- The SC decrypts the capability token $\mathbf{CT_{BS\text{-}SC}}$ and confirmation using the SC-BS session key $\mathbf{SecK_{SC\text{-}BS}}$ and checks whether the timestamp $\mathbf{tstamp_{SC}}$ is correctly updated. If so, then the SC can trusts the BS and can start submitting services requests to the BS, where each service request contains the id of the requested enterprise service and the corresponding capability token, encrypted with the session key $\mathbf{SecK_{SC\text{-}BS}}$.
- Upon receiving the service request, the BS decrypts the request by using the shared session key, verifies the included capability token. In the positive case, the BS executes the requested enterprise service (operation). The BS encrypts the result with the SC-BS session key and returns the encrypted message to SC.

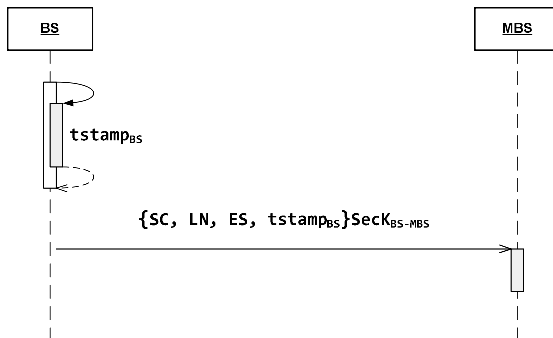

Figure 7. Service Metering phase

5) Service Metering phase

Upon successful submission of the computed service result in the previous phase, the BS informs the Metering & Billing Service MBS about the consumption of the Enterprise Service **ES** by the Service Consumer **SC**. For this purpose, the BS sends the following message to the MBS (see Figure 7): $\mathbf{<\{SC, LN, ES, tstamp_{BS}\}SecK_{BS\text{-}MBS}>}$. The message contains the id of the Service Consumer, the SC's license number, the id of the consumed Enterprise Service, and BS's local time. The message is encrypted with the long-term symmetric key $\mathbf{SecK_{BS\text{-}MBS}}$ shared between BS and MBS. Upon receiving the message, the MBS decrypts the message and checks the timestamp.

The MBS then updates its records about the usage of the service by the given Service Consumer. Depending on the agreed service subscription type (e.g. monthly, annual), the MBS sends a bill to SC and asks him to pay for the consumed Enterprise Service.

V. Performance Considerations

Threats can arise at many different points in modern networked systems and compromise one or more security dimensions, including confidentiality, integrity and availability. Our approach utilized a combination of symmetric cryptography and public key cryptography. The latter requires a public key infrastructure. Neither public key cryptography nor public key infrastructure comes for free. Adding cryptographic mechanisms and procedures to applications and systems does not come inexpensively. Security increases the total cost of computer-system ownership. Security is about trade-offs, rather than absolutes, and that we should strive for good-enough security, not for more security than necessary [1]. We limit the usage of public key cryptography due to performance reasons. Symmetric key encryption is orders of magnitude faster than public key encryption. For example, encrypting a 128-byte block using a public key of 512 bits takes 3.5 milliseconds on a Pentium-II 266 MHz3 whereas symmetric key encryption using AES takes less than one microsecond on the same machine [2]. Therefore, our approach exploited a hybrid scheme by amalgamating the convenience of asymmetric cryptography with the efficiency of symmetric cryptography.

Here are some thoughts on some issues which will have impact on the security and performance of a real-world implementation of our protocol:

- Longer keys provide an increased level of security but at a performance cost. Bruce Schneier [3] indicates that, a public key of 1,280 bits should be sufficient to protect against attacks from individuals, but a 1,536-bit key will be needed to protect from attacks originated in large corporations. A 2,048-bit key would be the best protection against attacks from the government. Each of these selections carries a certain performance cost. Therefore, when implementing our approach, a careful quantitative analysis of the performance impacts of security protocols must be combined with an analysis of potential threats so that good-enough security mechanisms are deployed for each situation.
- For both security and performance reasons, most digital signature algorithms specify that only the digest of the message be "signed", not the entire message. Our protocols assume that a real-world implementation of our approach would utilize such digital signature algorithms.
- Intel [4] provides cryptography library functions which are optimized for performance on the Itanium processor family. This library functions may be utilized for an implementation of our approach.

- When processing cryptographic functions, hardware modules [4] may be used because they are more secure and provide higher performance than software security modules. Hardware modules are optimized to generate the random numbers for encrypting and decrypting keys and messages and producing keys and hash values.

Note that our solution may not require a new public key infrastructure. Application vendors strive to build public key infrastructures for realizing their next generation Single-Sign-On solutions based on industry standards such as SAML [7], WS-Trust [8]. Our architectural components such as STS rely on such standards and, therefore, may utilize these existing security infrastructures like Kerberos [6].

VI. CONCLUSIONS

To best of our knowledge, this is the first paper focusing on authentication and authorization for SOA based SaaS applications. We presented an architecture and a security protocol for secure enterprise services consumption for service-oriented SaaS technology platforms. Our approach utilizes a combination of asymmetric and symmetric cryptography, and capability based access control. According to our approach, access to platform services is regulated based on the permissions encoded in cryptographic capability tokens. We argue that our approach can be used as a reference proposal for real-world implementations by software vendors, although the real-world implementations may need to modify our protocol and architecture slightly, depending on the existing capabilities of the technology platforms and envisioned industry standards such as SAML and WS-Trust, and supported security protocols like Kerberos. This is a work in progress. We plan to validate our approach by developing a research prototype.

REFERENCES

[1] R. Sandhu. Good-Enough Security, IEEE Internet Computing, vol. 7, no. 1, 2003, pp. 66–68.

[2] B. Schneier et al. Performance Comparison of the AES Submissions, Proc. 2nd AES Conf., Nat’l Inst. Standards and Technology, 1999, pp. 15–34;

[3] B. Schneier. Applied Cryptography, 2nd ed., John Wiley & Sons, 1996.

[4] http://softwarecommunity.intel.com/articles/eng/3443.htm

[5] http://en.wikipedia.org/wiki/Hardware_Security_Module

[6] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks, *IEEE Communications*, 32(9):33-38. September 1994.

[7] SAML. http://saml.xml.org/

[8] WS-Trust. http://docs.oasis-open.org/ws-sx/ws-trust/200512

[9] Y. Karabulut. Secure Mediation Between Strangers in Cyberspace, Dissertation Thesis, 2002, https://eldorado.uni-dortmund.de/bitstream/2003/2560/2/karbulut.ps.

[10] C. Altenschmidt, J. Biskup, U. Flegel and Y. Karabulut, Secure Mediation: Requirements, Design and Architecture, Journal of Computer Security, Volume 11, Issue 3, March 2003.

[11] D. Clarke, J.-E. Elien, C. Ellison, M. Freddette, A. Morcos, R. R. Rivest, Certificate Chain Discovery in SPKI/SDSI, Journal of Computer Security, Volume 9, Issue 4, January 2001.

[12] J. Altmann, Integrating PKI with Kerberos, 5th Annual PKI R&D Workshop, Gaithersburg, MD, USA, April 2006.

[13] F. Chong, G. Carraro, R. Wolter. Multi-Tenant Data Architecture. June 2006. http://msdn.microsoft.com/en-us/library/aa479086.aspx