UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

SÉCURITÉ DES SERVICES WEB : RESTAURATION D'UN MESSAGE SOAP APRÈS DÉTECTION D'UNE ATTAQUE PAR ENVELOPPEMENT SUR UN ÉLÉMENT SIGNÉ

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

IBRAHIM A. KEITA

HIVER 2010

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS
Département d'Informatique et d'Ingénierie

Ce mémoire intitulé

SÉCURITÉ DES SERVICES WEB : RESTAURATION D'UN MESSAGE SOAP APRÈS DÉTECTION D'UNE ATTAQUE PAR ENVELOPPEMENT SUR UN ÉLÉMENT SIGNÉ

Présenté par

Ibrahim A. Keita

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des
personnes suivantes :

Dr. Kamel Adi .....Directeur de recherche
Dr. Michal Iglewski ..... co-directeur de recherche
Dr. Karim El Guemhioui ..... Président du jury
Dr. Luigi Logrippo .....membre du jury

Mémoire accepté le : 11 mars 2010

À mes parents qui n'ont cessé de m'aider, de m'encourager et de me soutenir durant toute ma vie.

À mon épouse auprès de qui, je trouve la force et la motivation pour aller de l'avant.

REMERCIEMENTS

Je saisis l’occasion pour remercier le corps professoral du Département d’informatique et d’ingénierie, pour le savoir qu’il m’a inculqué. En particulier, je remercie mes superviseurs M. Kamel Adi et M. Michal Iglewski, pour leurs conseils et leur patience. Je remercie également M. Luigi Logrippo et M. Karim El Guemhioui, pour leur travail d’évaluation et leurs remarques pertinentes qui ont contribuées à améliorer ce travail.

Merci !

RÉSUMÉ

Les services web (SW) sont des applications hétérogènes et réparties exposant leurs fonctionnalités sous forme de services exécutables à distance. Les SW sont basés sur des technologies qui favorisent l'interopérabilité, l'extensibilité, l'indépendance vis-à-vis des plates-formes et des langages de programmation, et bénéficient d'une panoplie de spécifications pour la prise en charge de divers aspects. Les SW échangent à l'aide d'un type de message appelé message SOAP dont la syntaxe et la sémantique sont définies par la spécification SOAP. Même si leur sécurité a été améliorée par les spécifications plus ou moins récentes, les SW n'ont pas encore atteint la maturité à ce niveau. L'attaque par enveloppement (APE) en est une preuve. Une APE consiste à envelopper un élément d'un document XML par un autre élément, dans le but d'annuler son traitement. Typiquement, un autre traitement, une reproduction de l'élément enveloppé avec un nouveau contenu, est fourni par le pirate à l'emplacement original de ce dernier. Malheureusement, il n'y a pas de solution concrète à ce problème, ou bien les solutions proposées ne marchent pas dans tous les cas, et/ou elles nécessitent des données et d'opérations cryptographiques supplémentaires dans le message SOAP. En fait, toutes ces solutions se limitent à la simple détection des APE, sans traiter la restauration du message altéré après détection de l'attaque.

Dans ce mémoire, nous proposons une solution pour la restauration d'un message SOAP altéré par une APE. Puisqu'une restauration nécessite au préalable une détection, nous proposons également une technique de détection des APE. Notre solution est basée sur la recherche de motifs que nous avons établis sur la base de scénarios d'attaque réalistes. La solution est également basée sur certaines informations établies par le concepteur sur son service. Notre solution ne nécessite pas de données supplémentaires ni d'opérations cryptographiques dans le message SOAP. Elle permet l'économie de la bande passante grâce à la restauration qui évite la retransmission du message SOAP altéré. Par ailleurs, notre solution est compatible avec les spécifications des SW. L'implémentation de notre solution a montré des résultats intéressants. À notre connaissance, c'est la première solution pour la restauration d'un message SOAP altéré par une APE.

ABSTRACT

Web services (WS) are heterogeneous and distributed applications that expose their functionalities as remote services over a network. WS are built under technologies that promote interoperability, scalability and independence over operating systems and programming languages. WS exchange information over such a message called SOAP message which syntax and semantic are specified by a SOAP specification. Although WS take advantage of various security specifications; they still remain vulnerable to signed element wrapping attack (EWA). A EWA consists of wrapping an element in another element; often unknown to SOAP node, with the aim of cancelling its treatment. Typically, another treatment (a replication of the wrapped element with new content) is supplied by the attacker in the original location of the wrapped element. Unfortunately, the current works which addressed this problem do not provide any complete solution or the proposed solution do not addressed all EWAs and require additional data and cryptographic operations on SOAP message. In fact, these works are only focused on the EWAs detection without paying attention to the issue of the SOAP message restoration when detecting attack.

In this thesis, we propose a solution for SOAP message restoration when detecting an EWA. Since restoration requires prior detection, we also provide a solution for EWAs detection. Our solution is pattern-based analysis. It is also based on some information set up by the designer about his service. Patterns are established from realistic attack scenarios. Our solution does not require additional data or cryptographic operations within the SOAP message. It helps to save bandwidth with the restoration that prevents the transmission of altered SOAP message. Our solution is consistent with the specifications of WS. The implementation of our model has shown interesting results. To the best of our knowledge, this is the first solution for SOAP message restoration when detecting an EWA on a signed element.

Table des matières

Liste des algorithmes

Liste des figures

INTRODUCTION GÉNÉRALE

Les services web (SW) représentent le nouveau socle technologique de l'Architecture Orientée Service (AOS) qui, à son tour, est un concept et une approche de mise en œuvre des architectures réparties. Les SW permettent l'interaction des applications distantes et, comme toute implémentation des AOS, ils sont centrés sur la relation de service et la formalisation de cette relation dans un *contrat de service*[1]. Les SW ne sont pas une révolution en soi. Plusieurs initiatives avaient déjà été introduites pour définir un tel modèle d'architecture, et certaines d'entre-elles étaient fonctionnelles. Cependant, ces initiatives n'ont jamais atteint le statut de standard à cause de leur implémentation complexe et rigide qui ne garantie pas l'interopérabilité. Parmi ces technologies, nous citons CORBA (Sun, Oracle, IBM, …), DCOM (Microsoft) et RMI (Sun).

Les SW reposent sur les technologies standards qui sont SOAP, WSDL et UDDI. SOAP est un protocole d'échange de message. WSDL (Web Services Description Language) est un langage de description des SW. UDDI (*Universal Description, Discovery and Integration*) est un outil utilisé pour l'enregistrement, la publication et la découverte de services. À côté de celles-ci, d'autres spécifications ont été développées pour divers aspects tels que la sécurité, la coordination des processus métier, la fiabilité des messages, les modalités transactionnelles, etc. Toutes ces spécifications se basent sur le format universel XML, offrant ainsi un haut niveau d'interopérabilité ainsi qu'une grande flexibilité.

Les spécifications de sécurité incluent WS-Security, XML-Signature, XML-Encryption, WS-Policy, XML-SecurityPolicy, WS-Trust, XKMS, SAML, XACML, etc. Cette panoplie de spécifications permet une granularité appréciable et améliore grandement la sécurité. Malgré tout, les SW n'ont pas encore atteints la maturité au niveau de la sécurité. L'attaque par enveloppement (APE) en est une preuve. Une APE consiste à envelopper un élément d'un document XML par un autre élément, dans le but d'annuler son traitement. Typiquement, un autre traitement, une reproduction de l'élément enveloppé avec un nouveau contenu, est fourni par le pirate à l'emplacement original de ce dernier. Malheureusement, les solutions proposées à ce problème ne sont pas satisfaisantes et se limitent à la détection des APE sans aborder la restauration du message altéré après détection de l'attaque.

Dans le cadre de notre recherche, nous proposons une solution pour la restauration d'un message SOAP après détection d'une APE. Puisqu'une restauration nécessite au préalable une détection, nous proposons également une technique de détection des APE. C'est une technique basée à la fois sur la recherche de motifs tirés de scénarios d'attaque réalistes, et des informations établies par le concepteur sur son service.

Ce document est structuré de la manière suivante: le chapitre 1 introduit l'AOS et présente le fondement technologique ainsi que le fonctionnement des SW. Le chapitre 2 est consacré à la sécurité des SW. Dans le chapitre 3, nous abordons les failles de sécurité connues des SW. Dans le chapitre 4, sont présentés l'état de l'art des APE ainsi que nos objectifs, démarche et méthodologie concernant le déroulement ce travail. Le chapitre 5 est consacré à notre technique, avec un exemple à l'appui. Dans le chapitre 6, nous présentons l'implémentation de notre technique ainsi qu'une analyse de sa performance. Enfin, nous concluons par les contributions de notre technique ainsi que les travaux futurs à envisager.

[1] Un contrat de service est un document décrivant essentiellement les fonctions du service, l'interface du service ainsi que la qualité du service.

CHAPITRE 1: INTRODUCTION AUX SERVICES WEB

Dans ce chapitre, nous introduisons l'Architecture Orienté Service. Nous abordons ensuite les fondements technologiques des services web, à savoir les spécifications SOAP, WSDL et UDDI. Nous terminons par présenter le fonctionnement d'un service web.

1.1 L'ARCHITECTURE ORIENTÉE SERVICE (AOS)

L'AOS est un modèle d'exécution des applications logicielles distantes. L'AOS se caractérise par le concept de *service* entre applications qui échangent à travers des messages. Un service est le plus petit traitement qu'une application offre à une autre, généralement à la demande de celle-ci. L'application qui fournit un service est appelée *fournisseur ou prestataire de service* et l'application qui utilise le service est appelée *client de service*. Une application peut jouer à la fois le rôle de fournisseur de service et le rôle de client de service. L'accomplissement d'un service par un fournisseur de service à la demande d'un client de service est régi dans un contrat de service. Le contrat de service est spécifié par le fournisseur de service. Il s'agit d'un document qui définit:

- les fonctions du service;
- l'interface du service;
- la qualité du service.

On désigne par fonction de service une description abstraite de l'offre de service. L'interface de service est une description des mécanismes et des protocoles de communication avec le prestataire de service. On désigne par qualité de service les détails de fiabilité, de disponibilité, de robustesse, etc.

L'AOS correspond à un réseau de relation de service dans lequel un service peut être gratuit, payant, troquée ou mixe. Un service gratuit ne requiert aucune forme de rémunération. Un service payant est un service dont l'utilisation requiert une forme de rémunération définie dans les modalités de paiement. Dans un tel service, il est aussi défini les modalités de facturation ainsi que d'éventuelles pénalités en cas de non satisfaction des clauses du contrat. Un service troqué est un service offert en échange d'un autre, c'est la notion du troc. Un service mixte est un service qui se prête à la fois aux deux précédents, par exemple: une partie de la rémunération sous forme numéraire et l'autre sous forme d'échange de service.

Lors de la conception d'une AOS, il peut être adopté une stratégie d'agrégation de service ou de dissémination de service. La technique d'agrégation de service consiste à une fédération de service dans laquelle un service offre des fonctionnalités qui proviennent d'autres services encapsulés. La dissémination de service est l'approche contraire qui consiste à implémenter des services spécifiques qui étaient préalablement encapsulés avec d'autres pour former un tout.

1.2. LE FONDEMENT DES SERVICES WEB

Minimalement, le fonctionnement d'un SW requiert uniquement les spécifications de base qui sont: SOAP et WSDL. Lorsque l'utilisation du SW nécessite d'abord sa découverte par un utilisateur, alors la spécification UDDI est également requise afin de publier le service. À côté de celles-ci, d'autres spécifications existent pour des aspects divers. Une liste complète des spécifications des SW se trouve dans la référence [1].

Remarque
Par souci de lisibilité, les exemples de message SOAP contenus dans ce document ne sont pas dans la forme normative. Seule la partie pertinente est illustrée. Les éléments ou attributs non pertinents sont remplacés par des pointillés.

1.2.1 Protocole SOAP v1.2

Le protocole SOAP [2] est un standard introduit en 1998 comme moyen d'échange de messages entre applications hétérogènes et réparties. Les paquets SOAP prennent place au sein d'un protocole de transport, et ils sont échangés suivant un certain style de communication, par exemple: style RPC (Remote Procedure Call), style document, …

SOAP était manifestement victime de son acronyme « Simple Object Access Protocol » qui n'indique en rien sa fonction. Certains chercheurs ont manifesté leur embarras quant au sens de cet acronyme, et ont souhaité quelque chose en relation avec son vrai rôle, notamment: « *Service Oriented Access Protocol* ». Finalement, W3C a adopté SOAP comme un nom propre depuis la sortie de sa version 1.2.

SOAP repose sur le format XML. Le format XML est utilisé pour définir le contenu du message SOAP ainsi que sa structure, le tout est alors acheminé à l'aide d'un protocole de transport. À ce niveau, le protocole HTTP reste le plus utilisé, mais tout autre protocole de transport[2] de messages tels que SMTP, POP, … sont utilisables.

XML est un format universel de description de données (structurées ou non). Depuis sa sortie, XML est une technologie de plus en plus outillée dont la manipulation est assurée par plusieurs langages de programmation. HTTP est un protocole simple pour le transport de message. HTTP est utilisable pratiquement sur n'importe quel type de connexion. L'utilisation du protocole HTTP permet au protocole SOAP de faire abstraction de la problématique de gestion des ports; les messages SOAP traversent les pare-feux et les proxys sans aucune nécessité de changer les règles de filtrage. SOAP peut ainsi garantir un haut niveau d'interopérabilité tout en restant indépendant des spécificités des systèmes, des plates-formes ainsi que les langages de programmation.

[2] En réalité, les protocoles HTTP, SMTP et POP sont des protocoles d'application selon le modèle OSI. Cependant, dans le contexte d'échange de message SOAP, on les désigne sous le nom de protocole de transport car ils servent effectivement à transporter les messages SOAP

1.2.1.1 Quelques concepts importants

a. Message SOAP

Un message SOAP est le type de message que les SW utilisent pour échanger. La spécification SOAP exige qu'un message SOAP soit formé d'une enveloppe (`SOAP Envelope`) qui contient un en-tête optionnel (`SOAP Header`) et un corps (`SOAP Body`).

Voici un exemple simplifié de message SOAP:

```
<Envelope xmlns="http://www.w3.org/2001/06/soap-envelope">
  <Header ...>
    <MsgControl>
        <priority>1</priority>
        <expiry>2001-06-22T14:00:00-05:00</expiry>
    </MsgControl>
  </Header>
  <Body ...>
      <notify>
         <msg>Pick up Mary at school at 2pm</msg>
     </notify>
  </Body>
</Envelope>
```

Cet exemple illustre un message de notification. L'information à transmettre se trouve dans la partie `Body` tandis que les détails du traitement, notamment la date d'expiration ainsi que la priorité du message, se trouvent dans la partie `Header`.

b. Nœud SOAP

Un nœud SOAP est un agent qui a la capacité de transmettre, de recevoir et/ou de traiter un message SOAP. Il peut s'agir d'un émetteur, un intermédiaire ou d'un destinataire.

c. Module SOAP

Chaque nœud SOAP possède ce que l'on appelle un module SOAP qui implémente la syntaxe et la sémantique associée, des entrées d'en-tête destinées à son nœud SOAP. Une entrée d'en-tête est un sous-élément direct de l'élément `Header`, tel que l'élément `MsgControl` de l'exemple précédent. On dit qu'un nœud SOAP reconnaît une entrée d'en-tête si cette dernière est conforme à une syntaxe et sémantique spécifiée dans son module SOAP.

d. Émetteur SOAP

Un émetteur SOAP est un nœud SOAP capable de générer un message SOAP à transmettre à un destinataire pour traitement.

e. Intermédiaire SOAP

Un intermédiaire est un nœud SOAP qui a la capacité de recevoir un message SOAP et de le retransmettre sur la chaîne de transmission. Certains intermédiaires font bien plus que le simple routage des messages. Ils sont désignés sous le nom d'intermédiaire actif, car ils ont la responsabilité de traiter certaines entrées d'en-tête du message. Il peut s'agir d'un service de sécurité, d'un service de transformation du contenu, d'un service d'annotation, etc.

f. Destinataire SOAP

Un destinataire est un nœud SOAP à l'intention de qui, un émetteur envoie un message qui peut éventuellement passer par d'autres nœuds SOAP appelés intermédiaires. Le destinataire du message a la charge de traiter la partie Body. Il peut également traiter les entrées d'en-têtes qui lui sont adressés.

g. Chaîne de transmission

Une chaîne de transmission est le chemin pris par un message SOAP depuis son envoi par un émetteur jusqu'à sa réception par son destinataire. Ce parcours peut comprendre zéro ou plusieurs intermédiaires.

h. Processeur SOAP

Chaque nœud SOAP possède son propre processeur SOAP dont le rôle est de traiter les messages SOAP qui passe par lui suivant un modèle de traitement présenté dans la section 1.2.1.4.

1.2.1.2 Fonctionnement du protocole SOAP

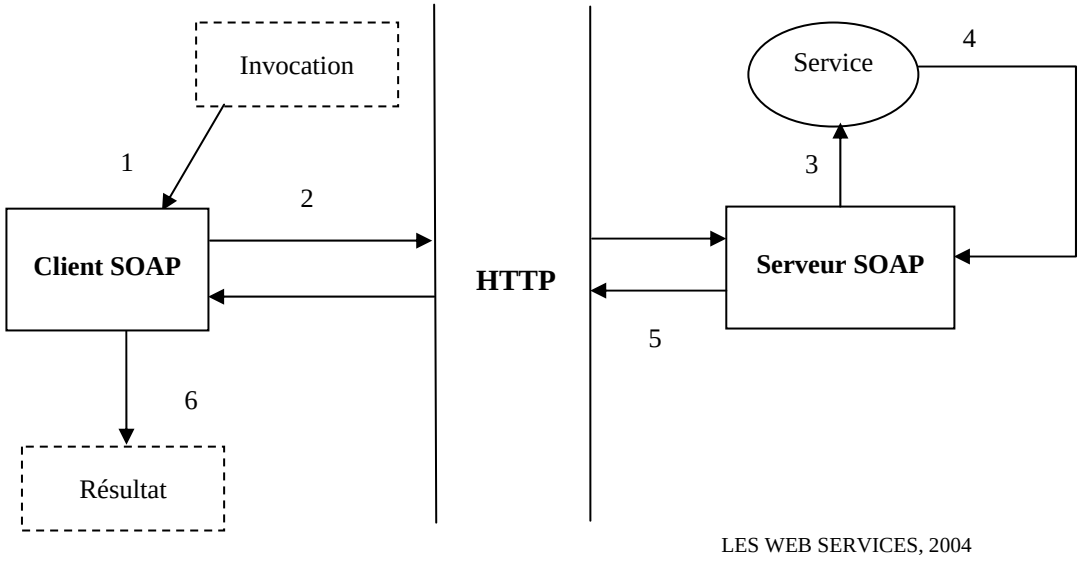


Figure 1 : Fonctionnement du protocole SOAP

Explication

Un demandeur de service invoque un service par l'entremise d'un agent logiciel, généralement un navigateur web. Le message est rassemblé par le client SOAP qui le met sous format SOAP avant de l'acheminer via HTTP au serveur SOAP. La requête SOAP arrive au niveau du serveur qui effectue une analyse syntaxique sur le document XML afin de vérifier sa cohérence avant de donner la main au service invoqué. À ce niveau, si l'analyse syntaxique du serveur échoue, la demande est rejetée et un message d'erreur est envoyé au client. Si l'analyse syntaxique réussit, le service invoqué traite le message et renvoie le résultat au serveur SOAP qui le met sous format SOAP avant de le retransmettre vers le client SOAP toujours via HTTP. Le client SOAP reçoit la réponse SOAP et accomplit son traitement applicatif.

1.2.1.3 Structure d'un message SOAP

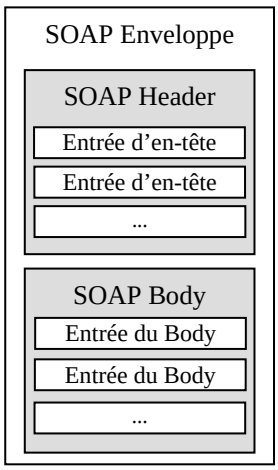


Figure 2 : Structure d'un message SOAP

a. SOAP Envelope

C'est l'élément racine d'un message SOAP. Il doit avoir le nom d'espace *http://www.w3.org/2003/05/soap-envelope* qui indique qu'il s'agit de l'élément `SOAP Envelope`. `SOAP Envelope` peut utiliser d'autres noms d'espace si cela est requis. Par exemple, il doit inclure les noms d'espace de WS-Security, XML-Signature, XML-Encryption si le message contient une déclaration de sécurité, de signature et de chiffrement. *`SOAP Envelope`* possède un attribut `encodingStyle` qui informe les règles d'encodage du message SOAP. `SOAP Envelope` possède deux sous-éléments: `SOAP Header` et `SOAP Body`.

Rappel sur les préfixes d'éléments XML

Dans un document XML, un élément peut être préfixé. Par exemple, dans le nom de balise « soap:Envelope », le préfixe est « soap », l'élément est Envelope. Un préfixe sert à associer les éléments et leurs attributs au nom d'espace dans lequel ils doivent être considérés. Un préfixe peut également contribuer à la lisibilité/compréhension d'un document XML. Par exemple, pour les éléments et attributs régis par SOAP, on utilise souvent le préfixe « soap » pour les distinguer des autres éléments d'un message SOAP. Parfois, on utilise aussi « s11 » ou « s12 » pour préciser la version SOAP. Pour XML-Signature, on utilise généralement « ds », pour WS-Security, c'est « wsse », « wsse10 » ou « wsse11 », pour XML-Encryption, c'est « xenc ». L'utilisation d'un préfixe est facultative, et son choix est arbitraire. D'autres détails supplémentaires sur les préfixes peuvent être trouvés dans la spécification XML.

b. SOAP Header

`SOAP Header` est optionnel. S'il est présent, il doit être le premier sous-élément de `SOAP Envelope`. `SOAP Header` donne des directives, à un intermédiaire ou au destinataire, concernant le traitement du message. Ces directives concernent généralement la sécurité : déclaration d'assertions, déclaration de signature, déclaration de chiffrement, information sur une clé cryptographique, etc. *`SOAP Header`* contient un ou plusieurs blocs appelés également entrées d'en-têtes. Une entrée d'en-tête se compose d'un ou plusieurs espaces de nom, d'un nom local et de quatre attributs: *encodingStyle*, *role*, *Relay* et *mustUnderstand*.

L'attribut encodingStyle

L'attribut encodingStyle, assigné à une entrée d'en-tête, indique le style d'encodage utilisé pour sérialiser cette entrée. Cette même information est utilisée pour sérialiser l'entrée lors de son traitement.

L'attribut role

L'attribut `role`, assigné à une entrée d'en-tête, sert à designer un nœud SOAP responsable de son traitement. La valeur de `role` est un URI; la spécification en définit trois valeurs spéciales. La valeur `next` désigne tous les nœuds SOAP responsables du traitement de cette entrée d'en-tête. La valeur `none` indique qu'aucun nœud SOAP n'est responsable du traitement de cette entrée d'en-tête. Une telle entrée d'en-tête (avec `role` à `none`) pourrait contenir des informations requises dans le traitement d'autres entrées. La valeur `ultimateReceiver` de `role` désigne le destinataire responsable du traitement de cette entrée d'en-tête. L'absence de l'attribut `role` équivaut à sa présence avec la valeur `ultimateReceiver`.

Nom du rôle	**Valeur réelle**
ultimateReceiver	http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver
next	http://www.w3.org/2003/05/soap-envelope/role/next
none	http://www.w3.org/2003/05/soap-envelope/role/none

Par simplicité, la valeur réelle de ***role*** *sera remplacée dans nos exemples par un trois points suivi d'un slash et du nom du rôle. Par exemple, pour le* ***role none*** *on écrira* ***…/none****.*

L'attribut mustUnderstand

L'attribut `mustUnderstand`, assigné à une entrée d'en-tête, indique si une erreur doit être générée au cas où aucun traitement n'est prévu pour cette entrée chez le nœud SOAP responsable de son traitement. Si le traitement échoue pour une entrée d'en-tête ayant `mustUnderstand` à `true`, une erreur est générée et envoyée à l'émetteur, puis le traitement s'arrête. Si le traitement échoue dans le cas d'une entrée d'en-tête ayant `mustUnderstand` à `false`, le reste du traitement suit son cours sans génération d'erreur. L'absence de cet attribut équivaut à sa présence avec la valeur `false`.

L'attribut relay

L'attribut `relay` est utilisé pour rendre transmissible une entrée d'en-tête lorsque celle-ci n'a pas été traitée par son nœud responsable. Plus précisément, lorsqu'un intermédiaire n'arrive pas à traiter une entrée d'en-tête qui lui est destinée, alors l'intermédiaire doit le réintroduire dans le message avant de le retransmettre si l'entrée possédait l'attribut `Relay` à `true`. L'absence de cet attribut équivaut à sa présence avec la valeur `false`.

Voici quelques exemples d'entrées d'en-tête:

Exemple 1

```
...
<soap:Header ...>
  <MsgControl soap:role=".../next"
     soap:mustUnderStand="true">
    <expiry>2001-06-22T14:00</expiry>
  </MsgControl>
</soap:Header>
...
```

Dans cet exemple, l'entrée MsgControl est destinée à tous les nœuds SOAP jusqu'au destinataire final. De plus, son traitement est obligatoire.

Exemple 2

```
...
<soap:Header ...>
  <MsgControl soap:mustUnderStand="true">
    <expiry>2001-06-22T14:00</expiry>
  </MsgControl>
</soap:Header>
...
```

Dans cet exemple, l'entrée MsgControl est destinée au destinataire final, et son traitement est obligatoire.

Exemple 3

```
...
<soap:Header ...>
  <MsgControl>
    <expiry>2001-06-22T14:00</expiry>
  </MsgControl>
</soap:Header>
...
```

Dans cet exemple, l'entrée MsgControl est destinée au destinataire final, mais son traitement est facultatif.

c. SOAP Body

SOAP `Body` est l'élément contenant les données utiles à transmettre. Chacune de ses entrées contient des informations applicatives que le destinataire doit traiter. L'élément `Body` est également utilisé pour transmettre un message d'erreur dans le cas où une erreur survient.

1.2.1.4 Modèle de traitement du message SOAP

Lorsqu'un nœud SOAP reçoit un message SOAP, qu'il soit intermédiaire ou destinataire, il procède comme suit:

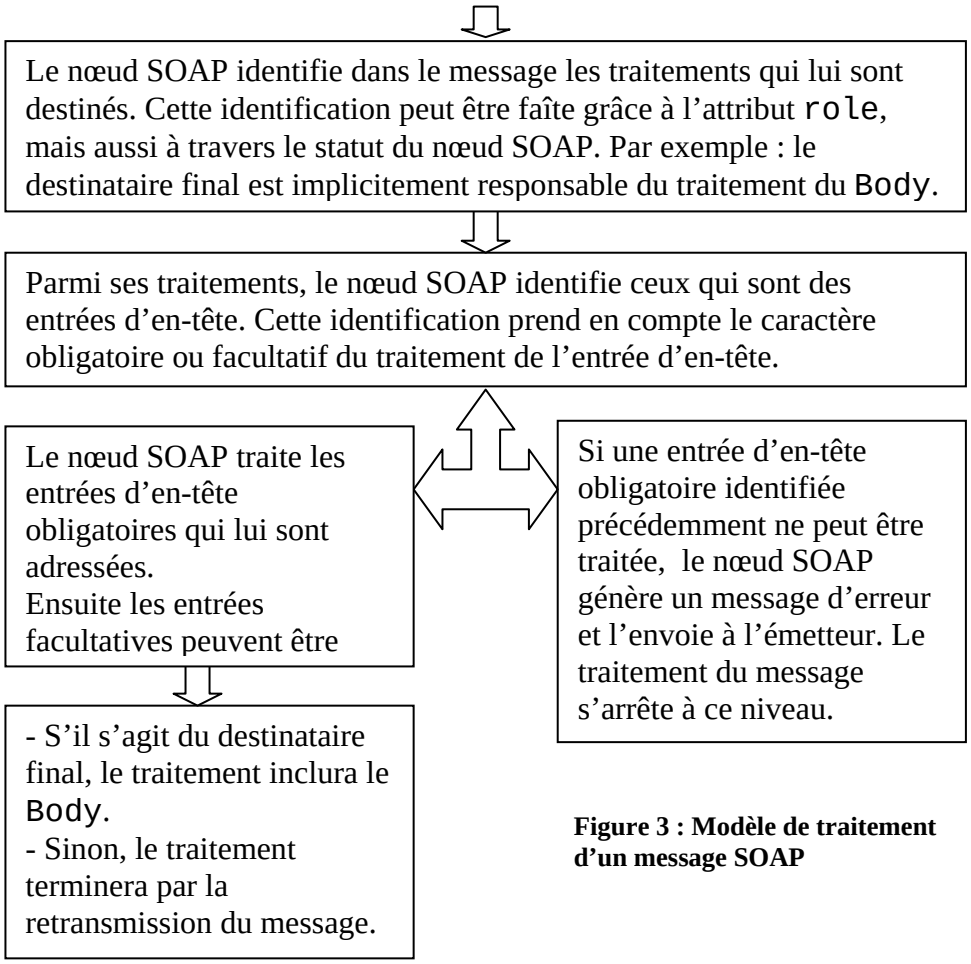


Figure 3 : Modèle de traitement d'un message SOAP

1.2.1.5 Protocole SOAP et les protocoles de transport

SOAP se prête de façon flexible à divers modes de transport et de styles d'échange. D'abord, il supporte les styles d'échange de message unique (appelé encore messagerie unidirectionnelle) et de message requête/réponse. Le style requête/réponse admet deux modes, à savoir le mode RPC (message contenant des paramètres d'entrée et des valeurs de retour) et le mode document (message contenant des documents). Le style de liaison ne se limite pas à la messagerie unidirectionnelle et à la requête/réponse, un tout autre style peut être mis en œuvre au gré du concepteur.

SOAP peut être utilisé conjointement avec HTTP, SMTP ou tout autre protocole de transport de message.

1.2.2. WSDL: WEB SERVICES DESCRIPTION LANGUAGE

WSDL [3] est un standard et le langage de description des SW. Cette description inclut le point d'accès du service, le type de liaison accepté, les fonctionnalités (ou méthodes) offertes par le service, les types de données utilisés dans les messages, etc. La description est faite dans le format XML, ce qui le rend exploitable par un grand nombre de systèmes et de langages de programmation.

1.2.2.1 Les six éléments de base d'un document WSDL

Un document WSDL est formé des six éléments suivants:

- l'élément `types`: définition des types de données utilisés lors de l'échange;
- l'élément `message`: représentation abstraite du contenu d'un message (données à transmettre, valeur de retour);
- l'élément `portType`: définition d'un ensemble d'opérations offertes par un service web;
- l'élément `binding`: définition d'un protocole de transport et le format des messages;
- l'élément `service`: ensemble de ports (lien url) associés à une opération donnée;
- l'élément `port`: adresse d'une liaison définissant le point d'accès associé à un service.

1.2.2.2 Squelette d'un document WSDL

Voici le squelette d'un document WSDL[3]

```
1. <wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

2.    <import namespace="uri" location="uri"/>*

3.    <wsdl:types> ?
         <wsdl:documentation .... />?
         <xsd:schema .... />*
      </wsdl:types>
4.    <wsdl:message name="nmtoken"> *
         <wsdl:documentation .... />?
```

[3] Les symboles ? et * présents dans la grammaire ci-dessus ont la même signification que dans les expressions régulières : ? signifie *apparaît au plus une fois*, * signifie *apparaît 0 ou plusieurs fois*.

```
         <part name="nmtoken" element="qname"? type="qname"?/> *
      </wsdl:message>
5.    <wsdl:portType name="nmtoken">*
         <wsdl:documentation .... />?
         <wsdl:operation name="nmtoken">*
            <wsdl:documentation .... /> ?
            <wsdl:input name="nmtoken"? message="qname" ...>?
                <wsdl:documentation .... /> ?
            </wsdl:input>
            <wsdl:output name="nmtoken"? message="qname" ...>?
                <wsdl:documentation .... /> ?
            </wsdl:output>
            <wsdl:fault name="nmtoken" message="qname"> *
                <wsdl:documentation .... /> ?
            </wsdl:fault>
         </wsdl:operation>
      </wsdl:portType>
6.    <wsdl:binding name="nmtoken" type="qname">*
         <wsdl:documentation .... />?
         <wsdl:operation name="nmtoken">*
           <wsdl:documentation .... /> ?
           <wsdl:input> ?
               <wsdl:documentation .... /> ?
           </wsdl:input>
           <wsdl:output> ?
               <wsdl:documentation .... /> ?
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <wsdl:documentation .... /> ?
           </wsdl:fault>
         </wsdl:operation>
      </wsdl:binding>
7.    <wsdl:service name="nmtoken"> *
         <wsdl:documentation .... />?
         <wsdl:port name="nmtoken" binding="qname"> *
            <wsdl:documentation .... /> ?
         </wsdl:port>
      </wsdl:service>

  </wsdl:definitions>
```

Ce document constitue le squelette d'un document WSDL. La plupart des éléments de ce document autorisent des extensions pour des besoins spécifiques. Alors il ne sera pas rare de voir dans un vrai document WSDL de nouveaux éléments ou attributs qui n'auraient pas été mentionnés sur ce présent document.

1.2.2.3 Description de la structure d'un document WSDL

Pour décrire un document WSDL, nous prenons en exemple le SW StockQuoteService qui est l'un des exemples de SW qui vient avec Axis2, un OpenSource de Apache.

Parenthèse

Axis2 est un moteur de SW qui s'exécute en tant qu'application web sous Jakarta Tomcat, le moteur de Servlet de Apache. Axis2 permet le déploiement automatique des SW. Avec Axis2, on peut afficher la liste des services disponibles ainsi que le fichier wsdl associé à chaque service. La version courante de Axis2 (1.4.1) supporte les spécifications suivantes: SOAP 1.1 et 1.2, SAAJ (SOAP with Attachments API for Java), WSDL 1.1, WS-Addressing, WS-Policy. La version future de Axis2 prévoit le support de WS-Security/Secure-Conversation, de WS-Trust, de WS-Reliable Messaging et de WS-Eventing. Axis2 supporte plusieurs moyens de transport qui sont: HTTP, SMTP, JMS (Java Messaging Service), TCP. Pour une

information complète sur Axis2, le lecteur peut consulter sa documentation qui se trouve sur le site de Apache.

Remarque
XML-Encryption et XML-Signature sont pris en charge par le projet XML Security de Apache.

Comme mentionné précédemment, le service StockQuoteService est celui que nous allons décrire à l'aide d'un document WSDL. Il s'agit d'une classe Java qui fournit des méthodes pour mettre à jour et interroger le prix d'un produit.

Service web StockQuoteService

```
...
public class StockQuoteService {
    private HashMap map = new HashMap();

    public double getPrice(String symbol) {
        Double price = (Double) map.get(symbol);
        if(price != null){
            return price.doubleValue();
        }
        return 42.00;
    }

    public void update(String symbol, double price) {
        map.put(symbol, new Double(price));
    }
}
```

a. L'élément definitions

L'élément `definitions` est la racine d'un document WSDL. Cet élément peut optionnellement avoir les attributs `name` et `targetNamespace` qui permettent respectivement de lui associer un nom et un nom d'espace. Ces deux attributs permettent à un document WSDL d'être importé par d'autres documents de définitions (schéma ou document WSDL) au cas où ceux-ci l'utilisent en tant qu'extension à leur propre définition. Cette façon permet la réutilisation de définition de service, et le point numéro 2 du squelette d'un document WSDL montre sa mise en œuvre. D'autres noms d'espace peuvent être utilisés s'ils sont requis dans la définition du service. Les éléments `documentation` sont optionnels et peuvent être utilisés sous n'importe quels éléments d'un document WSDL afin de leur associer une explication en langage naturel.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://quickstart.samples/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                   xmlns:axis2="http://quickstart.samples/"
                   xmlns:ns="http://quickstart.samples/xsd" ...>
    <wsdl:documentation>StockQuoteService</wsdl:documentation>
             ...
</wsdl:definitions>
```

b. L'élément types

L'élément `types`, grâce à l'importation d'un schéma ou à la définition de structures de données, permet de spécifier le type (l'encodage) des données qui sont utilisées lors de l'échange des messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
     ...
    <wsdl:types>
        <xs:schema ...>
            <xs:element name="getPrice">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="symbol" nillable="true"
                        type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="getPriceResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="return" nillable="true"
                         type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="update">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="symbol" nillable="true"
                         type="xs:string"/>
                        <xs:element name="price" nillable="true"
                         type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </wsdl:types>
          ...
</wsdl:definitions>
```

c. L'élément message

L'élément `message` sert à donner une représentation abstraite du contenu d'un message qui peut être une invocation ou une réponse. L'élément `message` peut se répéter plusieurs fois et possède un attribut `name` qui lui sert de référence parmi les autres éléments `message` du même document WSDL. L'élément `message` comprend une ou plusieurs sous-éléments `part` appelés également parties logiques. L'élément `part` peut être vue comme des paramètres d'entrée et de sortie d'une fonction. L'élément `part` a trois attributs qui sont `name`, `type` et `element`. Seul l'attribut `name` est obligatoire et permet d'associer un nom unique à un élément `part` parmi les autres éléments `part` du même élément *message*. L'attribut `type` permet de définir le type de donnée de l'élément `part`. L'attribut `element` est une alternative syntaxique pour spécifier un type complexe pour l'élément `part`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
            ...
    <wsdl:message name="getPriceMessage">
        <wsdl:part name="part1" element="ns:getPrice"/>
    </wsdl:message>
    <wsdl:message name="getPriceResponseMessage">
        <wsdl:part name="part1" element="ns:getPriceResponse"/>
    </wsdl:message>
    <wsdl:message name="updateMessage">
        <wsdl:part name="part1" element="ns:update"/>
    </wsdl:message>
            ...
</wsdl:definitions>
```

d. L'élément portType

L'élément `portType` permet de définir un ensemble d'opérations (ou fonctions). Il est optionnel et peut se répéter plusieurs fois dans un document WSDL. Son attribut `name` lui sert d'identifiant unique parmi les autres éléments `portType` d'un même document WSDL. Le sous-élément de `portType` est l'élément `operation` qui a aussi un attribut `name` qui lui sert d'identifiant non nécessairement unique (surcharge de fonction par exemple). Les sous-éléments de l'élément `operation` sont `input` (un message d'entrée), `output` (un message de sortie) ou `fault` (un message d'erreur). La présence ou non de ces trois éléments dépend du type de transmission utilisé dans l'élément `binding`[4]. Le sous-élément `fault` est optionnellement utilisé pour indiquer un rapport d'erreur dans les cas de transmission à deux sens. Les trois sous-éléments de l'élément `operation` ont un attribut optionnel `name` et un attribut obligatoire `message`. L'attribut `name` sert d'identifiant unique, et l'attribut `message` permet aux éléments `operation` de se référer aux éléments `message`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
      ...
    <wsdl:portType name="StockQuoteServicePortType">
        <wsdl:operation name="getPrice">
            <wsdl:input message="axis2:getPriceMessage"
              wsaw:Action="urn:getPrice"/>
            <wsdl:output message="axis2:getPriceResponseMessage"
              wsaw:Action=".../getPriceResponse"/>
        </wsdl:operation>
        <wsdl:operation name="update">
            <wsdl:input message="axis2:updateMessage"
              wsaw:Action="urn:update"/>
        </wsdl:operation>
    </wsdl:portType>
      ...
</wsdl:definitions>
```

[4] Il y'a quatre types de transmission qui sont **One-way** : réception d'un message, mais pas d'envoi, **Request-response** : réception d'un message et envoi d'une réponse, **Solicit-response** : envoi d'un message et réception d'une réponse, **Notification** : envoi d'un message, mais pas de réponse. Un élément **operation** indique en réalité l'une de ces quatre types de transmission.

e. L'élément binding

L'élément `binding` définit un protocole de transport et le format des données pour les opérations et les messages pour chaque élément `portType`. Il est optionnel et peut être présent plusieurs fois. Plusieurs éléments `binding` peuvent être définis pour un même élément `portType`. Un élément `binding` comprend deux attributs obligatoires `name` et `type`. L'attribut `name` lui sert de référence unique parmi d'autres éléments `binding` du même document WSDL. L'attribut `type` sert de référence à l'élément `portType` auquel il est lié. Un élément `binding` peut contenir plusieurs sous-éléments `operation`. Ce sous-élément a un attribut `name` qui permet de le faire correspondre à un élément `operation` défini dans `portType`. Puisque la valeur d'un attribut `name` d'un élément `operation` dans `portType` peut ne pas être unique, dans ce cas on spécifiera les éléments `input`, `output` et `fault` correspondants à l'élément `operation` que l'on veut référencer. À noter la présence sous le sous-élément `body` l'attribut obligatoire `use` avec la valeur `literal`. L'autre valeur possible de `use` est `encoded`. La valeur `literal` de `use` sous un élément indique que lors de la génération du message SOAP, les données textuelles de cet élément ne renfermeront pas l'information d'encodage spécifiée soit dans `types` ou via l'importation d'un schema XML. La valeur `encoded` de `use` indique que les informations d'encodage spécifiées seront incorporées à l'élément dans le message SOAP généré.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
            ...
    <wsdl:binding name="StockQuoteServiceSOAP11Binding"
                  type="axis2:StockQuoteServicePortType">
        <soap:binding transport=".../soap/http" style="document"/>
        <wsdl:operation name="getPrice">
            <soap:operation soapAction="urn:getPrice"
                 style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="update">
            <soap:operation soapAction="urn:update"
               style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:binding name="StockQuoteServiceSOAP12Binding"
          type="axis2:StockQuoteServicePortType">
        <soap12:binding transport=".../soap/http" style="document"/>
        <wsdl:operation name="getPrice">
            <soap12:operation soapAction="urn:getPrice"
           style="document"/>
            <wsdl:input>
                <soap12:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap12:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="update">
            <soap12:operation soapAction="urn:update"
```

```
                style="document"/>
            <wsdl:input>
                <soap12:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
            ...
</wsdl:definitions>
```

f. L'élément service et l'élément port

L'élément `service` permet de grouper ensemble une multitude de ports. Il est optionnel et peut se répéter plusieurs fois. L'élément `service` possède un attribut `name` qui lui sert de référence unique parmi d'autres éléments `service` du même document WSDL. Il possède un sous-élément `port` qui indique l'adresse d'un point terminal pour un élément `binding`. L'élément `port` a deux attributs obligatoires qui sont `name` et `binding`. L'attribut `name` lui sert de référence unique parmi d'autres éléments `port` du même document. L'attribut `binding` permet d'associer un élément `port` à un élément `binding`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
            ...
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServiceSOAP11port"
             binding="axis2:StockQuoteServiceSOAP11Binding">
            <soap:address location=".../StockQuoteService"/>
        </wsdl:port>
        <wsdl:port name="StockQuoteServiceSOAP12port"
            binding="axis2:StockQuoteServiceSOAP12Binding">
            <soap12:address location=".../StockQuoteService"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

1.2.2.4 Document WSDL intégral du service StockQuoteService

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
    <wsdl:documentation>StockQuoteService</wsdl:documentation>
    <wsdl:types>
        <xs:schema ...>
            <xs:element name="getPrice">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="symbol" nillable="true"
                        type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="getPriceResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="return" nillable="true"
                         type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="update">
```

```
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="symbol" nillable="true"
                         type="xs:string"/>
                        <xs:element name="price" nillable="true"
                         type="xs:double"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </wsdl:types>
    <wsdl:message name="getPriceMessage">
        <wsdl:part name="part1" element="ns:getPrice"/>
    </wsdl:message>
    <wsdl:message name="getPriceResponseMessage">
        <wsdl:part name="part1" element="ns:getPriceResponse"/>
    </wsdl:message>
    <wsdl:message name="updateMessage">
        <wsdl:part name="part1" element="ns:update"/>
    </wsdl:message>
    <wsdl:portType name="StockQuoteServicePortType">
        <wsdl:operation name="getPrice">
            <wsdl:input message="axis2:getPriceMessage"
              wsaw:Action="urn:getPrice"/>
            <wsdl:output message="axis2:getPriceResponseMessage"
                        wsaw:Action=".../getPriceResponse"/>
        </wsdl:operation>
        <wsdl:operation name="update">
            <wsdl:input message="axis2:updateMessage"
                        wsaw:Action="urn:update"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="StockQuoteServiceSOAP11Binding"
                  type="axis2:StockQuoteServicePortType">
        <soap:binding transport=".../soap/http" style="document"/>
        <wsdl:operation name="getPrice">
            <soap:operation soapAction="urn:getPrice"
                 style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="update">
            <soap:operation soapAction="urn:update"
               style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:binding name="StockQuoteServiceSOAP12Binding"
      type="axis2:StockQuoteServicePortType">
        <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>
        <wsdl:operation name="getPrice">
            <soap12:operation soapAction="urn:getPrice" style="document"/>
            <wsdl:input>
                <soap12:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
```

```
                <soap12:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="update">
            <soap12:operation soapAction="urn:update" style="document"/>
            <wsdl:input>
                <soap12:body use="literal"/>
            </wsdl:input>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServiceSOAP11port"
             binding="axis2:StockQuoteServiceSOAP11Binding">
            <soap:address location=".../StockQuoteService"/>
        </wsdl:port>
        <wsdl:port name="StockQuoteServiceSOAP12port"
            binding="axis2:StockQuoteServiceSOAP12Binding">
            <soap12:address location=".../StockQuoteService"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

1.2.3. UDDI: Universal Description Discovery and Integration

UDDI [4] est un annuaire conçu pour les SW. Il donne une description d'un SW (en langage naturel), du fournisseur de service ainsi que des informations techniques relatives aux points d'accès du service. Ainsi, UDDI est consultable selon trois types de visions qui sont:

- les pages blanches:

Les pages blanches donnent une description des entreprises fournisseurs de service, telles que leur nom, leur adresse, leur description, ainsi que toute autre information susceptible de les identifier;

- les pages jaunes:

Les pages jaunes donnent une description d'un service suivant une catégorie;

- les pages vertes:

Les pages vertes renseignent sur les informations techniques nécessaires à l'accès et à l'exécution d'un service.

1.2.3.1 Structure UDDI

La phase de description d'un service comprend la description du fournisseur de service, la description du service lui-même ainsi que les informations techniques relatives au service. Pour ce faire, UDDI se base sur les éléments suivants: `BusinessEntity`, `BusinessService`, `BindingTemplate`, `tModel`, `Publisher Assertions`. Voici l'information véhiculée par chacun de ces éléments:

- `BusinessEntity:` donne des informations sur l'activité ou l'organisation offrant le service;

- `BusinessService:` donne des informations sur les services offerts par une organisation décrite dans `BusinessEntity`;

- `BindingTemplate:` donne des informations techniques nécessaires à l'utilisation d'un service en particulier;

- `tModel(`*`technical models`*`):` donne une représentation du service en modèle technique en utilisant des concepts réutilisables tels que: le type du SW, le protocole utilisé par le SW, la catégorie du système, etc;

- `Publisher Assertions:` décrit la relation de partenariat d'une `BusinessEntity` avec une autre dans le cadre de l'accomplissement d'un service.

À présent, regardons de plus près la structure de chacun de ces éléments.

a. BusinessEntity

L'élément `businessEntity` a trois attributs et sept sous-éléments. Le sous-élément `discoveryURL` indique un URL vers la définition de d'autres façons d'utiliser le service. Le sous-élément `name` désigne le nom de l'organisation. Le sous-élément `description` donne la description de l'organisation. Le sous-élément `contacts` représente les informations de contact de l'organisation. Le sous-élément `businessServices` donne la liste des services offerts par l'organisation. Le sous-élément `identifierBag` représente d'autre information d'identification de l'organisation. Le sous-élément `categoryBag` représente les catégories auxquelles l'organisation appartient. L'attribut `businessKey` sert d'identifiant unique pour `businessEntity`. L'attribut `operator` désigne le nom du site de l'opérateur UDDI qui publie ce service. L'attribut `authorizedName` désigne le nom de la personne ayant publié le service.

b. BusinessService

L'élément `businessService` a deux attributs et quatre sous-éléments. Le sous-élément `name` est le nom de la personne ayant enregistré le service. Le sous-élément `description` est la description du `businessService`. Le sous-élément `bindingTemplates` représente les modèles de liaison pour la description technique du service. Le sous-élément `categoryBag` représente les catégories auxquelles le service appartient. L'attribut `serviceKey` est un identifiant unique pour `businessService`. L'attribut `businessKey` est un identifiant unique pour `businessEntity`.

c. BindingTemplate

L'élément `bindingTemplate` a deux attributs et trois sous-éléments. Le sous-élément `description` donne la description du `bindingTemplate`. Le sous-élément `tModelInstanceDetails` représente des structures d'information `tModel`. Les sous-éléments `accesspoint` et `hostingRedirect` sont mutuellement exclusifs. Le sous-élément `accesspoint` indique un point d'accès à l'information sur le service. Le sous-élément `hostingRedirect` est un pointeur vers un autre modèle de liaison si `accesspoint` est absent. L'attribut `bindingKey` identifie de manière unique un `bindingTemplate`. L'attribut `serviceKey` identifie le service qui utilise ce `bindingTemplates`.

d. tModel

L'élément `tModel` a trois attributs et cinq sous-éléments. Le sous-élément `name` désigne le nom du `tModel`. Le sous-élément `description` donne la description du `tModel`. Le sous-élément `overviewDoc` indique les documents relatifs au `tModel`. Le sous-élément `identifierBag` indique le numéro d'identification du `tModel`. Le sous-élément `categoryBag` désigne les catégories du `tModel`. L'attribut `tModelKey` sert d'identifiant pour le `tModel`. L'attribut `operation` désigne le nom du site de l'opérateur UDDI où se trouvent les `tModel`. L'attribut `authorizedName` désigne les entités ayant enregistré les `tModel`.

e. Publisher Assertions

L'élément `publisherAssertion` a trois sous-éléments qui sont `fromKey`, `toKey` et `keyedReference`. L'élément `fromKey` désigne la première `businessEntity` impliquée dans l'assertion. L'élément `toKey` désigne la deuxième `businessEntity` impliquée dans l'assertion. Et enfin L'élément `keyedReference` désigne le type de relation liant les deux `businessEntity`.

1.3. FONCTIONNMENT D'UN SERVICES WEB

1.3.1. Scénario 1: services web non publics

- Services web non publiés dans un annuaire UDDI
- Services web dont le point d'accès est connu des utilisateurs du service
- Généralement des SW intranet, des SW de type B2B (Business to Business), …

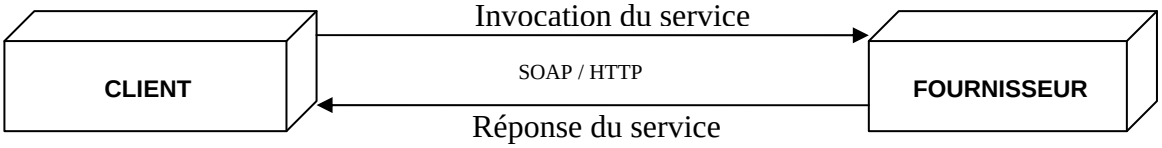


Figure 4 : Fonctionnement d'un SW à accès non public

1.3.2. Scénario 2: services web publics

- Services web publiés dans un annuaire UDDI
- Généralement des SW de type B2C (Business to Consumer), ex: agence de voyage, …
- Le SW et l'annuaire UDDI qui le publie peuvent ne pas résider sur la même machine

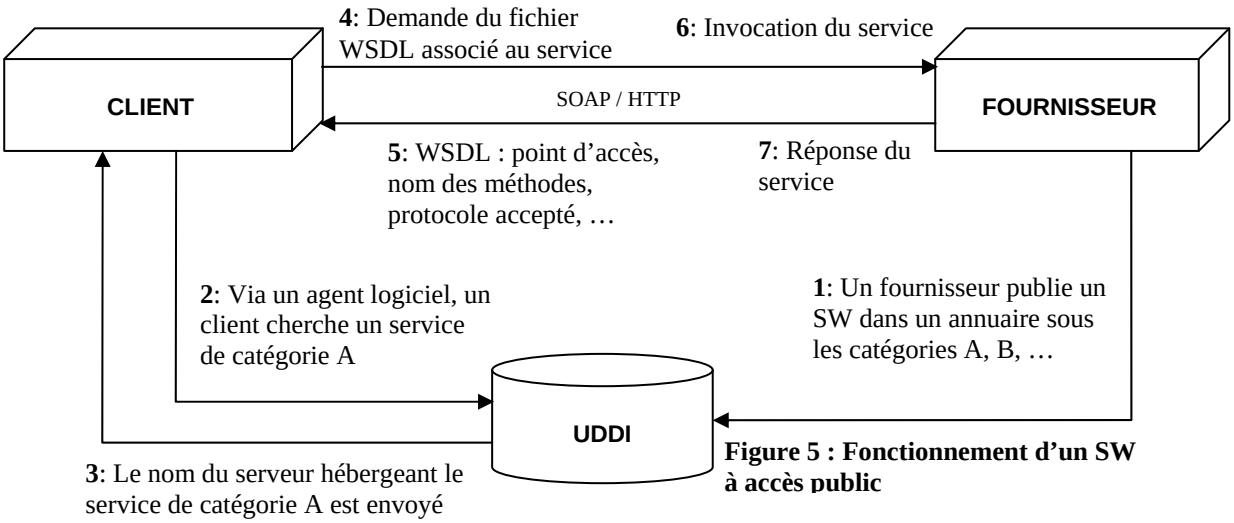


Figure 5 : Fonctionnement d'un SW à accès public

CHAPITRE 2: SÉCURITÉ DES SERVICES WEB

Dans ce chapitre, nous passons en revue les spécifications de sécurité des services web. Pour les spécifications de sécurité pertinentes à notre travail de recherche telles que WS-Security ou XML-Signature, nous montrons l'utilisation à travers des exemples simples.

2.1. INTRODUCTION

En général, l'effort pour sécuriser un système dépend de son environnement d'opération. Cela signifie que bien avant de mettre en place une quelconque stratégie de sécurité, il convient de déterminer les menaces auxquelles le système peut faire face dans son contexte précis. Par exemple, un système qui a un accès sur l'Internet n'aura pas les mêmes exigences de sécurité qu'un système isolé. De la même manière, deux systèmes ayant accès au réseau Internet peuvent avoir des exigences de sécurité différentes dépendamment du contexte dans lequel chaque système opère.

Les SW communiquent via des messages SOAP. De ce fait, l'altération des messages constitue la principale menace à laquelle doivent faire face les concepteurs de SW. D'une part, lorsqu'un SW envoie un message à un autre service, la confidentialité du message doit être assurée si celle-ci est requise. Si nécessaire, le destinataire du message doit être en mesure de déterminer que le message provient effectivement de l'émetteur qui le revendique, et qu'il n'a subi aucune altération. D'autre part, l'autorisation d'accès est un défi de taille puisque les SW peuvent permettre l'accès à distance à des ressources souvent très importantes: base de données bancaires, base de données de compagnie d'assurance, etc.

La sécurité d'un message SOAP doit être assurée de bout en bout[5], et non de nœud en nœud. Pour tenter de répondre à ces besoins de sécurité, une panoplie de spécifications de sécurité, pour la plupart basées sur XML, ont vu le jour. Il s'agit entre autres de XML-Signature, XML-Encryption, WS-Policy, WS-SecurityPolicy, WS-Trust, XKMS (*XML Key Management Specification*), SAML (*Security Assertions Markup Language*), XACML (*eXtensible Acces Control Markup Language*), etc. Ces spécifications sont complémentaires, et contribuent collectivement à atteindre des objectifs de sécurité. L'arrivée de la spécification de sécurité WS-Security a grandement améliorée la sécurité des SW. Cette spécification offre une infrastructure de sécurité assez complète en se basant exclusivement sur les autres spécifications de sécurité.

2.2. SPÉCIFICATION DE SÉCURITÉ DES SERVICES WEB

2.2.1. WS-Security

La spécification WS-Security [5] est très importante. C'est elle qui permet d'utiliser, dans un message SOAP, la plupart des autres spécifications de sécurité.

[5] La sécurité de bout en bout signifie un contexte de sécurité établi depuis l'émetteur du message jusqu'au destinataire, incluant d'éventuels intermédiaires.

Dans un message SOAP, une déclaration de sécurité commence par l'élément `Security`. Cet élément peut se retrouver plusieurs fois dans un même message SOAP, chacun étant destiné à un nœud SOAP distinct à travers la chaîne de transmission.

Voici un fragment d'un document sécurisé:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="...">
  <soap:Header ...>
     ...
     <wsse:Security soap:role="..." soap:mustUnderstand="...">
     ...
     </wsse:Security>
     ...
   </soap:Header>
    ...
</soap:Envelope>
```

L'élément `Security` étant une entrée d'en-tête, il peut donc utiliser les attributs `role` (ou `actor` dans la version SOAP 1.1) et `mustUnderStand`. Nous rappelons que l'attribut `role` permet de désigner un nœud SOAP responsable du traitement de l'entrée correspondante. L'attribut `mustUnderStand` indique si une erreur doit être générée lorsqu'aucun traitement n'est prévu pour l'entrée correspondante.

L'élément `Security` est extensible par ajout de nouveaux éléments ou attributs, pourvus qu'ils soient définis dans un schéma. Si l'identification d'un nouvel élément ou attribut échoue, le nœud SOAP désigné doit générer une erreur SOAP.

2.2.1.1 Type de jeton de sécurité proposé par Security

L'élément `Security` propose les types de jetons de sécurité suivants:

- UsernameToken: cet élément, à travers son sous-élément `Username`, permet de passer un nom d'utilisateur comme jeton de sécurité.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="...">
  <soap:Header ...>
    ...
    <wsse:Security soap:role="..." soap:mustUnderstand="...">
      <wsse:UsernameToken>
         <wsse:Username>Zoe</wsse:Username>
       </wsse:UsernameToken>
   </wsse:Security>
   ...
 </soap:Header>
...
</soap:Envelope>
```

- BinarySecurityToken: cet élément permet de passer des jetons de sécurité de format binaire. Son attribut `Id` permet sa référence. L'attribut `EncodingType` spécifie le type d'encodage utilisé, par exemple: Base64Binary. Et l'attribut `ValueType` indique le jeton de sécurité en question, par exemple: un ticket Kerberos.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="...">
  <soap:Header ...>
    ...
    <wsse:Security soap:role="..." soap:mustUnderstand="...">
```

```
        <wsse:BinarySecurityToken wsu:Id="..."
                        EncodingType="..." ValueType="..."/>
      </wsse:Security>
      ...
    </soap:Header>
   ...
</soap:Envelope>
```

Dans certains cas, le destinataire peut exiger que les informations sur le jeton de sécurité soient chiffrées. L'élément `EncryptedData` peut être utilisé à cet effet pour contenir le jeton de sécurité chiffré. Cet élément fait partie de la spécification XML-Encryption qui est abordée plus loin.

2.2.1.2. Identification et référencement du jeton de sécurité

La signature numérique et le chiffrement nécessitent tous deux la spécification d'une clé qui peut être à l'intérieur ou à l'extérieur du message. Un jeton de sécurité, inclus ou non dans un message, peut être référencé grâce à l'élément `SecurityTokenReference` dont la syntaxe est la suivante:

```
<wsse:SecurityTokenReference wsu:Id="..." wsse11:TokenType="..."
     wsse:Usage="...">
</wsse:SecurityTokenReference>
```

L'élément `SecurityTokenReference` est localisé à l'intérieur d'une déclaration de signature (XML-Signature), d'encryption (XML-Encryption), etc pour indiquer le jeton de sécurité. L'attribut `Id` indique le libellé de la référence. L'attribut `TokenType` permet de spécifier, à travers un URI, le type du jeton référencé. Le dernier attribut `Usage` sert à définir un type d'usage de l'élément `SecurityTokenReference`. Malheureusement, nous n'avons pas trouvé de détail sur l'utilisation de l'attribut `Usage`.

Un référencement direct dans l'élément `SecurityTokenReference` est possible:

```
<wsse:SecurityTokenReference wsu:Id="...">
     <wsse:Reference URI="..." ValueType="..."/>
</wsse:SecurityTokenReference>
```

L'élément `Reference` sous `SecurityTokenReference` permet de faire une référence directe à un jeton de sécurité. Son attribut `URI` sert à localiser le jeton et son attribut `ValueType` indique le type du jeton.

À défaut d'utiliser une référence directe dans `SecurityTokenReference`, un identifiant de clé en format binaire peut être utilisé:

```
<wsse:SecurityTokenReference>
   <wsse:KeyIdentifier wsu:Id="..."
      ValueType="..."
      EncodingType="...">
     ...
    </wsse:KeyIdentifier>
</wsse:SecurityTokenReference>
```

L'attribut `Id` permet de nommer l'identifiant. L'attribut `ValueType` indique le type de l'identifiant et l'attribut `EncodingType` indique le format d'encodage de l'identifiant.

Dans certains cas, la référence peut être enchâssée dans `SecurityTokenReference` grâce à l'élément `Embedded` qui admet un attribut `Id` qui lui sert de référence pour la signature ou le chiffrement.

L'exemple suivant montre le référencement d'un jeton de sécurité à l'aide d'assertions basées sur SAML (Security Assertions Markup Language). SAML [13] est une spécification de sécurité qui permet l'échange d'information d'authentification et d'autorisation entre les SW:

```
<wsse:SecurityTokenReference>
   <wsse:Embedded wsu:Id="tok1">
      <saml:Assertion xmlns:saml="...">
         ...
      </saml:Assertion>
    </wsse:Embedded >
</wsse:SecurityTokenReference>
```

2.2.1.3. Exemples d'utilisation de WS-Security

L'exemple suivant illustre l'utilisation de XML-Signature avec WS-Security.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
 <soap:Header ...>
 <wsse:Security>
   <wsse:BinarySecurityToken ValueType="..."
    EncodingType="..."  wsu:Id="X509Token">
    MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
   </wsse:BinarySecurityToken>
  <ds:Signature>
    <ds:SignedInfo>
       <ds:CanonicalizationMethod Algorithm="..."/>
       <ds:SignatureMethod Algorithm= "..."/>
       <ds:Reference URI="#myBody">
           <ds:Transforms>
              <ds:Transform Algorithm= "..."/>
           </ds:Transforms>
           <ds:DigestMethod Algorithm= "..."/>
           <ds:DigestValue>EULddytSo1...</ds:DigestValue>
       </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>BL8jdfToEb1l/vXcMZNNjPOV...</SignatureValue>
    <ds:KeyInfo>
      <ds:SecurityTokenReference>
        <ds:Reference URI="#X509Token"/>
      </ds:SecurityTokenReference>
   </ds:KeyInfo>
 </ds:Signature>
</wsse:Security>
</soap:Header>
<soap:Body id="myBody">
  <StockSymbol>QQQ</StockSymbol>
</soap:Body>
</soap:Envelope>
```

L'exemple suivant illustre l'utilisation conjointe de XML-Encryption et WS-Security:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:wsu="..."
xmlns:ds="..." xmlns:xenc="...">
  <soap:Header>
    <wsse:Security>
      <xenc:ReferenceList>
```

```
        <xenc:DataReference URI="#bodyID"/>
      </xenc:ReferenceList>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <xenc:EncryptedData Id="bodyID">
      <ds:KeyInfo>
        <ds:KeyName>CN=Hiroshi Maruyama, C=JP</ds:KeyName>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```

2.2.2. XML-Signature

XML-Signature [6] est la spécification qui offre les services d'intégrité, d'authenticité du message, et/ou de l'émetteur. Une signature peut être appliquée aussi bien sur un objet contenu dans un document XML que sur un objet distant. Dans tous les cas, la référence de l'objet signé doit résider dans le document XML contenant la signature. XML-Signature permet également d'attacher une signature aussi bien sur le contenu intégral de l'objet que sur une partie. Ainsi la spécification admet un traitement sélectif. Il faut quant même reconnaître que la signature du contenu au complet d'un objet ne constitue pas toujours une solution envisageable, notamment pour des raisons de performance.

2.2.2.1 Structure de XML-Signature

```
<Signature ID?>
      <SignedInfo>
            <CanonicalizationMethod/>
            <SignatureMethod/>
            (<Reference URI? >
                  (<Transforms>)?
                  <DigestMethod>
                  <DigestValue>
            </Reference>)+
      </SignedInfo>
      <SignatureValue>
      (<KeyInfo>)?
            (<Object ID?>)*
</Signature>
```

Figure 6 : Structure de XML-Signature

L'élément `Signature` déclare une signature dans un document XML. Il peut contenir les sous-éléments `SignatureValue`, `SignedInfo`, `KeyInfo` et `Object`. Les deux premiers sont obligatoires tandis que les deux derniers sont optionnels. Les éléments `Signature` sous différents éléments `Security` d'un même message sont identifiés grâce à leur attribut `Id`. Cet attribut reste optionnel.

L'élément `SignatureValue` contient le résultat (la signature de l'objet) de l'encodage des données à l'aide de l'algorithme de signature, et elle est toujours encodée suivant la base64.

L'élément `SignedInfo` donne des informations spécifiques sur la signature telles que la référence à l'objet signé, la valeur du hash de l'objet, etc. Cet élément peut avoir un attribut

optionnel `Id` et les sous-éléments suivants: `CanonicalizationMethod`, `SignatureMethod`, `Reference`. L'élément obligatoire `CanonicalizationMethod` spécifie quel algorithme de canonisation est appliqué sur l'élément `SigneInfo` avant le calcul de la valeur de signature. L'élément obligatoire `SignatureMethod` identifie l'algorithme qui est utilisé pour générer et valider la signature. L'élément obligatoire `Reference` fournit l'algorithme utilisé pour calculer le hash de l'objet à signer, la valeur du hash ainsi que la référence à l'objet. L'élément `Reference` peut également contenir une liste de transformations à appliquer avant le calcul du hash (« digest »).

L'élément `KeyInfo` permet au nœud SOAP désigné de récupérer la clé requise pour la validation de la signature. À ce niveau plusieurs méthodes de gestion de clés peuvent être utilisées. La récupération de la clé peut nécessiter la fourniture de certaines informations supplémentaires sur le signataire. Ainsi L'élément `Object` et `SignatureProperties` peuvent être utilisés à cet effet.

2.2.2.2 Fonctionnement de XML-Signature

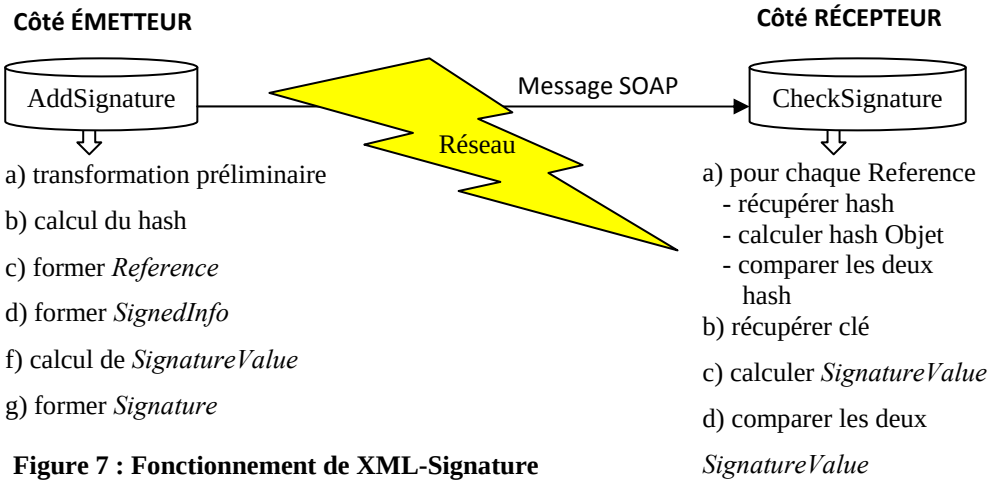


Figure 7 : Fonctionnement de XML-Signature

a. Procédure pour signer un objet

1) Application, s'il y'a lieu, des transformations préliminaires sur le contenu à signer:
Ces transformations peuvent être l'opération de canonisation[6], d'encodage/décodage, des transformations XSLT, d'application d'expressions XPath, de validation par XML schema. Typiquement les transformations permettent de mettre le contenu à signer dans une forme standard qui permet sa signature. Par exemple, si l'objet à signer renferme la signature elle-même, une transformation peut être utilisée pour exclure la signature lors de sa génération. Si aucune transformation n'est requise, on passe directement au point 2).
2) Calcul du hash (`DigestValue`):
Cette valeur est calculée à l'aide d'un algorithme spécifié par `DigestMethod`.

3) Création de l'élément `Reference`:
Former l'élément `Reference` avec ses éventuels attributs et sous-éléments.

[6] La « canonisation » permet d'appliquer sur un document XML une série de règle visant à le mettre sous une forme standard permettant son traitement par tous les outils XML. La « canonisation » permet, par exemple, d'établir une équivalence entre documents XML ayant un contenu syntaxiquement différente. Pour savoir en quoi consistent les règles de « canonisation », veuillez consulter la référence [7].

```
<Reference URI? >
    <Transforms>)?
    <DigestMethod>
    <DigestValue>
</Reference>
```

Ces opérations sont répétées pour chaque objet à signer (un élément référence pour chaque objet).

4) Création de l'élément `SignedInfo`:
L'élément `SignedInfo` contiendra l'élément `Reference` précédemment créé, et les deux éléments `CanonicalizationMethod` et `SignatureMethod`.

```
<SignedInfo>
     <CanonicalizationMethod/>
     <SignatureMethod/>
     <Reference URI? >
            (<Transforms>)?
             <DigestMethod>
             <DigestValue>
      </Reference>
</SignedInfo>
```

5) Calcul du format canonique (sérialisation) de l'élément `SignedInfo`:
Le format canonique de l'élément `SignedInfo` est calculé à l'aide de l'algorithme spécifié dans `CanonicalizationMethod`.

6) Calcul de la valeur de la signature (`SignatureValue`):
Il s'agit d'appliquer sur le hash (`DigestValue`), l'algorithme de signature spécifié sous `SignatureMethod` après avoir obtenu la clé de signature dans `KeyInfo`.

7) Création de l'élément `Signature`:
L'élément `Signature` contient le format canonique de `SignedInfo`, `SignatureValue`, `KeyInfo` et probablement un ou plusieurs éléments `Object`. L'élément `Object` peut contenir des informations supplémentaires telles que le temps de signature.

b. Procédure pour valider la signature

Pour vérifier une signature, XML-Signature prévoit les étapes inverses:

a) pour chaque `Reference` dans `SignedInfo`:
   - récupérer le hash (`DigestValue`)
   - retrouver l'objet original et calculer son hash à nouveau avec l'algorithme spécifié dans `DigestMethod`
   - comparer ce résultat avec la valeur du `DigestValue` dans `Signature`, si différente alors échec;

b) récupérer la clé de validation, et calculer à nouveau la signature de l'objet à l'aide de l'algorithme spécifié dans `SignatureMethod`.

L'algorithme terminera avec succès si les deux signatures correspondent.

2.2.2.3 Exemple d'un message signé

Soit le message SOAP suivant:

```
<soap:Envelope xmlns:soap="...">
  <soap:Body>
       <GetAccountBalance>
            <symbol>DIS</symbol>
       </GetAccountBalance >
  </soap:Body>
</soap:Envelope>
```

Le contenu de l'élément `Body` est l'objet que nous aimerons signer. En appliquant sur cet élément les sept étapes de signature susmentionnées, on obtient le message signé ressemblant au prochain listing:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
      <wsse:Security>
          <ds:Signature>
              <ds:SignedInfo>
                  <ds:CanonicalizationMethod Algorithm="..." />
                  <ds:SignatureMethod Algorithm=..." />
                  <ds:Reference URI="#body">
                     <ds:DigestMethode Algorithm="..." />
                     <ds:DigestValue>7ULp/H24b=…</ds:DigestValue>
                  </ds:Reference>
              </ds:SignedInfo>
              <ds:SignatureValue>8Gn40C7qT+rr …</ds:SignatureValue>
              <ds:KeyInfo>
                 ...
              </ds:KeyInfo>
          </ds:Signature>
      </wsse:Security>
  </soap:Header>
  <soap:Body id="body">
       <GetLastTradePrice>
            <symbol>DIS</symbol>
       </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

2.2.3. XML-Encryption

XML-Encryption [8] indique comment chiffrer (crypter) les données d'un document XML. Comme nous l'avons dit précédemment, lors de l'échange SOAP à travers une chaîne de transmission, un intermédiaire reçoit un message, traite éventuellement une partie et l'achemine à son tour. Dans un tel contexte, il est utile de pouvoir chiffrer uniquement certaines parties du document XML (l'information sensible) et laisser les autres telles qu'elles. XML-Encryption supporte cette granularité.

La spécification XML-Encryption peut être utilisée conjointement avec la spécification XML-Signature avec qui elle est parfaitement compatible.

La déclaration d'un chiffrement dans un document XML se fait à l'aide de l'élément `EncryptedData`. Cet élément a un sous-élément `CipherData` qui contient le résultat du chiffrement en base16 par l'intermédiaire de son sous-élément `CipherValue`. L'élément

`CipherReference` est le deuxième sous-élément de `CipherData`, et il permet d'identifier la ressource chiffrée si l'élément `CipherValue` n'est pas fourni.

Voici un exemple de chiffrement tiré du site de W3C. Il s'agit des informations de paiement d'une personne fictive nommée John Smith.

Document original (avant chiffrement)

```
<?xml version="1.0"?>
<PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <CreditCard Limit="5,000" Currency="USD">
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
 </PaymentInfo>
```

Chiffrement d'un élément XML au complet

Normalement l'information sur la carte de crédit est sensible. Avant d'envoyer une telle information, une option « pessimiste » est de chiffrer l'élément `CreditCard` au complet:

```
<?xml version="1.0"?>
<PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
</EncryptedData>
</PaymentInfo>
```

Chiffrement du contenu d'un élément XML

Au lieu de chiffrer tout le contenu de l'élément `CreditCard`, une meilleure option est de chiffrer uniquement le numéro de la carte de crédit et laisser les autres informations en clair:

```
<?xml version="1.0"?>
  <PaymentInfo xmlns="http://example.org/paymentv2">
    <Name>John Smith</Name>
    <CreditCard Limit="5,000" Currency="USD">
      <Number>
        <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
         Type="http://www.w3.org/2001/04/xmlenc#Content">
          <CipherData>
            <CipherValue>A23B45C56</CipherValue>
          </CipherData>
        </EncryptedData>
      </Number>
      <Issuer>Example Bank</Issuer>
      <Expiration>04/02</Expiration>
    </CreditCard>
  </PaymentInfo>
```

Chiffrement de données arbitraires et de documents XML
Évidemment les exigences de chiffrement dépendent des situations. Certaines applications peuvent exiger que le document XML soit crypté en entier:

```
<?xml version="1.0"?>
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
 MimeType="text/xml">
  <CipherData>
    <CipherValue>A23B45C56</CipherValue>
  </CipherData>
</EncryptedData>
```

2.2.4. Survol des autres spécifications de sécurité

À présent, nous passons brièvement à travers d'autres spécifications de sécurité pouvant être utilisées avec WS-Security. Il s'agit de:

- WS-SecureConversation [9]:
  Cette spécification permet la dérivation de clés de session ainsi que l'établissement et le partage de contexte de sécurité.

- WS-Authorization [1]:
  Toujours en cours de développement, cette spécification sera dédiée à la gestion des autorisations sur les données ainsi que les entités qui les manipulent.

- WS-SecurityPolicy [10]:
  Cette spécification permet de définir des assertions sur des propriétés de sécurité relatives aux messages SOAP. Par exemple, on peut spécifier quel élément du message doit être signé ou chiffré, quel type de jeton de sécurité est accepté, quel l'algorithme de signature est accepté, etc.

- WS-Policy [11]:
  Cette spécification définit le modèle et la syntaxe associée pour exprimer des politiques de sécurité relatives aux entités dans un environnement des SW: accès à un répertoire, accès à un système, etc.

- WS-Trust [12]:
  Cette spécification définie une syntaxe pour la réclamation, la distribution et la validation de jetons de sécurité.

- WS-Privacy [1]:
  Toujours en cours de développement, cette spécification décrira comment incorporer des politiques de confidentialité dans WS-Policy, et comment WS-Security peut être utilisé pour réclamer des identifiants de confidentialité associés à un message.

- SAML (Security Assertions Markup Language) [13]:
  La spécification SAML permet l'échange d'information d'authentification et d'autorisation entre les SW.

- XKMS (XML Key Management Specification) [14]:

XKMS propose une infrastructure de gestion de clés publiques. Elle a été conçue pour être utilisée harmonieusement avec XML-Signature.

- XACML (eXensible Access Control Markup Language) [15]:

XACML permet de décrire des politiques de contrôle d'accès qui peuvent être formulées dans les messages SOAP.

CHAPITRE 3: FAILLES DE SÉCURITÉ DANS LES SERVICES WEB

Dans ce chapitre, nous présentons les types d'attaques auxquelles les services web peuvent faire face. Parmi ces attaques, nous nous focalisons sur les attaques par enveloppement. Nous verrons pourquoi ce type d'attaque réussit malgré une signature sur l'élément attaqué. Finalement, nous présentons quelques scénarios d'attaques.

3.1. INTRODUCTION

Les SW bénéficient d'une panoplie de technologies, chacune conçue pour une raison spécifique. L'aspect sécurité est pris en charge par les spécifications de sécurité parmi lesquelles nous citons WS-Security, XML-Signature, XML-Encryption, etc. Ces spécifications peuvent se révéler assez complexes et difficiles à mettre en œuvre de façon parfaitement sécuritaire. Il est possible qu'un message SOAP utilisant ces spécifications reste toujours vulnérable à certains types d'attaques, car la sécurité a été mise en place de façon inadéquate. Certaines de ces spécifications ont été conçues pour permettre des extensions futures. D'autres offrent une grande flexibilité au niveau de la syntaxe. Tous ces détails apportent leur lot de problème comme nous le verrons dans la suite de ce chapitre.

Dus à leur nature, mais surtout à leur immaturité, les SW sont exposés à des menaces de sécurité qui sont très variées. En effet, les SW sont des programmes repartis qui communiquent via des messages SOAP, et qui sont le plus souvent exposés à des attaques au delà des frontières physiques de l'organisation. Théoriquement, n'importe qui peut composer un message pour l'envoyer à un service web. Un message SOAP est un document XML qui est manipulé par un outil ou un processeur XML. Ces outils ou processeurs peuvent faire l'objet d'attaque afin de perturber leur fonctionnement. De plus, un message SOAP, grâce au protocole HTTP, passe à travers les pare-feux et les proxys sans nécessité de changer les règles de filtrage. Cette « facilité » de transport monte d'un cran le risque d'infiltration des barrières de sécurité établies par les organisations.

3.2. LES ATTAQUES INHÉRENTES À XML

XML est le langage de description de données. XML est extensible et très expressif. Son expressivité réside dans sa capacité à décrire pratiquement tout type de donnée. XML a acquis une grande popularité depuis sa sortie grâce en partie à son format (un texte brut) qui simplifie les modifications du fichier, et peut être lu et analysé facilement. Il est devenu l'un des langages les plus outillés et les plus supportés par les langages de programmation ainsi que les systèmes d'exploitation. Malheureusement, XML lui-même peut constituer une menace pour les systèmes.

Il est possible de fabriquer un document XML dont l'analyse et/ou le traitement est susceptible de causer des dommages plus ou moins sérieux à un système [16,17]. Il s'agit:

- d'envoi de messages avec des noms d'élément ou d'attribut excessivement longs pour entraîner un débordement de mémoire lors du traitement;

- d'envoi d'un nombre très élevé de messages contenant des clés publiques d'une très grande taille. Un tel message peut accaparer une grande ressource lors des vérifications cryptographiques;
- d'envoi de messages avec un virus XML (X-Virus) en attachement;
- d'envoi à travers un tag CDATA d'un message des codes malicieux ou des commandes systèmes dont l'exécution peut compromettre la stabilité du système;
- d'envoi de messages avec des noms d'élément récursifs de sorte à compliquer le traitement;
- d'injection d'expression XPath, XSL ou SQL dans le code applicatif;
- d'inclusion, dans un message, d'un outil XML visant à perturber son traitement;
- de corruption d'outils XML tels que XML Schema, fichiers XSL;
- d'étude d'un fichier WSDL pour tenter d'avoir accès à un service non listé (service non public).

3.3. LES ATTAQUES PAR ENVELOPPEMENT (APE)

L'attaque décrite dans cette section exploite une flexibilité de XML-Signature et de SOAP. Nous avons vu qu'une signature dans un message SOAP est spécifiée par l'élément `Signature` qui comprend typiquement la référence, la signature et le hash de l'élément signé, et également les algorithmes de signature ainsi que les informations sur la clé de signature. Un élément signé est référencé par un URI sous l'élément `Reference`. Pour vérifier une signature, l'élément référencé dans l'élément `Reference` est récupéré, ensuite son hash est calculé puis comparé à celui dans `Signature`. Si cette vérification passe, alors on récupère la clé de validation, puis on calcule la signature. La validation réussit si cette nouvelle signature correspond à celle dans `Signature`. Malheureusement, il est possible de changer l'emplacement de l'élément signé sans que cela n'altère sa signature. De plus, avec l'attribut `mustUnderStand` à `false`, SOAP permet d'ignorer un élément inconnu pour le nœud SOAP désigné. À noter que le nœud SOAP désigné est celui à qui est destiné l'élément signé. D'autre part, avec l'attribut `role` à `none`, SOAP autorise l'interdiction du traitement d'un élément par un nœud SOAP. Ces possibilités peuvent être exploitées par un pirate pour faire un accès non autorisé à des ressources, ou faire des bris de confidentialité comme nous le verrons par la suite.

Scénario 1

Soit le message SOAP suivant:

```
<soap:Envelope xmlns:soap="...">
  <soap:Body>
       <GetLastTradePrice>
           <symbol>DIS</symbol>
       </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 1 : message SOAP avant signature de l'élément Body

Cet exemple met en évidence une requête SOAP à un service GetLastTradePrice avec un paramètre nommé `symbol` ayant la valeur `DIS`. `DIS` est l'identifiant d'un produit reconnu par le service. À la réception de ce message, le module SOAP procède à l'identification des traitements qui lui sont adressés telle que décrite dans la section 1.2.1.4. L'entrée

`GetLastTradePrice` du `Body` est la seule fonction identifiée, et elle est traitée par la suite.

Maintenant, nous allons appliquer une signature sur l'élément `Body` de notre message. Pour des raisons de lisibilité, les informations sur la clé de signature ont été omises dans le message. De toute façon, ces informations n'ont pas d'impact sur l'attaque, et elles ne sont pas nécessaires à sa compréhension. La valeur du hash (`DigestValue`) ainsi que celle de la signature (`signatureValue`) sont fictives. Le message est donc transformé comme suit:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    <ds:DigestMethod Algorithm="..." />
                    <ds:DigestValue>6TB+7UnlLp/H24p= ...</ds:DigestValue>
               </ds:Reference>
             </ds:SignedInfo >
             <ds:SignatureValue>juf/rbrVGn40CapgB ...</ds:SignatureValue>
             <ds:KeyInfo>
                ...
             </ds:KeyInfo >
        </ds:Signature>
   </wsse:Security>
  </soap:Header>
  <soap:Body id="body">
      <GetLastTradePrice>
        <symbol>DIS</symbol>
      </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 2 : message de l'exemple 1 après signature de l'élément Body

À la réception de ce message, le module SOAP procède une fois de plus à l'identification de ses traitements. Il en identifie deux: la vérification d'une signature et un traitement `GetLastTradePrice` dans `Body`. En principe, le traitement d'une signature précède celui de l'objet signé. Ainsi, la main est passée au module de vérification de la signature. Au cas où celle-ci est valide, le traitement suivra son cours comme précédemment.

Maintenant, nous simulons une APE sur l'élément signé `Body`. Modifions le message précédent de la manière suivante:

- envelopper l'élément `Body` dans un élément fictif `WrapperElement`;
- déplacer l'élément fictif `WrapperElement` dans l'élément `Header` pour en faire une entrée d'en-tête;
- assigner à `WrapperElement` les attributs `role` et `mustUnderStand` avec respectivement les valeurs `none` et `false`;
- introduire un nouvel élément `Body` invoquant le même service, mais avec un paramètre différent.

Suite à ces altérations, notre message ressemblera à celui-ci:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    <ds:DigestMethod Algorithm="..." />
                    <ds:DigestValue>6TB+7UnlLp/H24p= ...</ds:DigestValue>
               </ds:Reference>
             </ds:SignedInfo >
             <ds:SignatureValue>juf/rbrVGn40CapgB ...</ds:SignatureValue>
             <ds:KeyInfo>
                ...
             </ds:KeyInfo >
        </ds:Signature>
     </wsse:Security>
     <WrapperElement soap:role=".../none" soap:mustUnderStand="false">
       <soap:Body id="body">
         <GetLastTradePrice>
           <symbol>DIS</symbol>
         </GetLastTradePrice>
       </soap:Body>
    </WrapperElement>
  </soap:Header>
  <soap:Body>
      <GetLastTradePrice>
        <symbol>XIF</symbol>
      </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 3 : message de l'exemple 2 après altération par un pirate

À la réception de ce message, comme précédemment le module SOAP procède à l'identification des traitements qui lui sont adressés. Deux traitements sont alors identifiés: la vérification d'une signature et l'invocation d'un service. Puisque le contenu de l'objet référé par la signature, c'est-à-dire l'élément `Body` ayant l'attribut `id` à `body`, n'a pas changé, alors il n'y a pas de raison pour que sa signature soit invalide. Ainsi la vérification de la signature réussit. Ensuite le service `GetLastTradePrice` est invoqué, mais cette fois-ci, avec le nouveau paramètre `XIF` introduit par le pirate (à l'instar de `DIS`, `XIF` est l'identifiant d'un produit reconnu par le service). Il est très probable que le résultat de cette nouvelle requête soit différent de celui de la précédente.

Remarque

Bien que l'entrée WrapperElement contienne l'élément `Body` *ayant été signé, son identification échouera parce qu'il n'est pas un élément légitime (on supposera qu'il n'y a pas d'élément légitime avec un nom similaire du côté destinataire). Ainsi aucun traitement n'est prévu pour WrapperElement. Rappelons que la valeur* `false` *de* `mustUnderStand` *sous une entrée d'en-tête indique qu'aucune erreur ne sera générée au cas où aucun traitement n'est prévu pour cette entrée. Ainsi l'arborescence complète de* `WrapperElement` *est ignorée.*

Cette attaque peut être faite sur un autre élément différent du `Body` comme on va le voir dans l'exemple suivant:

```
<soap:Envelope xmlns:soap="..." xmlns:wsa="...">
  <soap:Header>
      <wsa:To>http://serviceweb.com/service1</wsa:To>
      <wsa:ReplyTo>
        <wsa:Address>http://serviceweb.com/service1</wsa:Address>
      </wsa:ReplyTo>
  </soap:Header>
  <soap:Body>
     <GetLastTradePrice>
        <symbol>DIS</symbol>
     </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 4 : message de l'exemple 1 après ajout d'une nouvelle entrée ReplyTo

Il s'agit du message de l'exemple 1 dans lequel on a ajouté deux points d'accès, l'un pour le service à invoquer, l'autre pour désigner un nœud où envoyer la réponse à la requête.

Le message suivant est le résultat de l'application d'une signature sur les éléments `To`, `ReplyTo` et `Body` du message de l'exemple 4.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="..."
xmlns:wsa="...">
  <soap:Header>
     <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    ...
               </ds:Reference>
                <ds:Reference URI="#to">
                    ...
               </ds:Reference>
                <ds:Reference URI="#replyTo">
                    ...
               </ds:Reference>
             </ds:SignedInfo >
             <ds:SignatureValue>...</ds:SignatureValue>
             <ds:KeyInfo>
                ...
             </ds:KeyInfo >
        </ds:Signature>
    </wsse:Security>
    <wsa:To id="to">http://serviceweb.com/service1</wsa:To>
    <wsa:ReplyTo id="replyTo">
      <wsa:Address>http://serviceweb.com/service1</wsa:Address>
   </wsa:ReplyTo>
  </soap:Header>
<soap:Body>
     <GetLastTradePrice>
        <symbol>DIS</symbol>
     </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 5 : message de l'exemple 4 après signature des éléments To et ReplyTo

Les entrées d'en-tête `To` et `ReplyTo` sont vulnérables à des APE telle qu'on a vu pour le cas du `Body`.

Scénario 2

Considérons le message SOAP de l'exemple 5, enveloppons l'entrée `ReplyTo` dans un élément fictif `WrapperElement` avec respectivement les attributs `role` et `mustUnderStand` à `none` et `false`.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="..."
xmlns:wsa="...">
  <soap:Header>
     <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    ...
               </ds:Reference>
                <ds:Reference URI="#to">
                    ...
               </ds:Reference>
                <ds:Reference URI="#replyTo">
                    ...
               </ds:Reference>
             </ds:SignedInfo >
             <ds:SignatureValue>...</ds:SignatureValue>
             <ds:KeyInfo>
                ...
             </ds:KeyInfo >
        </ds:Signature>
    </wsse:Security>
    <wsa:To id="to">http://serviceweb.com/service1</wsa:To>
   <WrapperElement soap:role=".../none" soap:mustUnderStand="false">
     <wsa:ReplyTo id="replyTo">
       <wsa:Address>http://serviceweb.com/service1</wsa:Address>
     </wsa:ReplyTo>
   </WrapperElement>
  </soap:Header>
  <soap:Body>
     <GetLastTradePrice>
        <symbol>DIS</symbol>
     </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 6 : message de l'exemple 5 après attaque de l'élément ReplyTo

Côté destinataire, l'entrée `WrapperElement` ne sera pas disponible à cause de la valeur `none` de `role` qui indique que l'entrée n'est destinée à aucun nœud SOAP. De plus, la valeur `false` de `mustUnderStand` permet d'ignorer l'entrée en cas de problème. Tout se passe comme si le message ne contient pas le sous arbre `WrapperElement`. Dans ce type d'attaque, il faut remarquer que l'élément attaqué (ici `ReplyTo`) doit forcement être un élément optionnel du message.

Scénario 3

Considérons le message SOAP de l'exemple suivant. Il s'agit d'une version modifiée de notre exemple 2. Nous avons, cette fois-ci, introduit une nouvelle entrée d'en-tête nommée `TimeStamp`.
L'élément `Timestamp` peut être utilisé pour indiquer la date de création et/ou d'expiration de l'élément `Security` dans lequel il apparaît. Sa syntaxe est définie dans la spécification WS-Security. Une autre APE consiste à déplacer `Timestamp` dans un nouvel élément `Security` sous lequel on assigne les attributs `role` et `mustUnderStand` avec respectivement les valeurs `none` et `false`. Cela a pour conséquence de dissocier l'élément `Timestamp` de son parent original. Ainsi d'autres attaques, notamment l'attaque par retransmission (replay attack), peuvent être basées sur celle-ci. L'exemple 8 montre le message SOAP altéré suite à ce scénario d'attaque.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="..."
xmlns:wsu="...">
  <soap:Header>
     <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    ...
               </ds:Reference>
                <ds:Reference URI="#time">
                    ...
               </ds:Reference>
             </ds:SignedInfo >
             <ds:SignatureValue>...</ds:SignatureValue>
             <ds:KeyInfo>
                ...
             </ds:KeyInfo >
       </ds:Signature>
       <wsu:Timestamp id="time">
         <wsu:Created>2005-05-29T08:45:00Z</wsu:Created>
         <wsu:Expires>2005-05-29T09:00:00Z</wsu:Expires>
      </wsu:Timestamp>
   </wsse:Security>
  </soap:Header>
  <soap:Body>
     <GetLastTradePrice>
        <symbol>DIS</symbol>
     </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Exemple 7 : message SOAP de l'exemple 2 après assignation de l'élément Timestamp

CHAPITRE 4: DÉTECTION DES APE ET RESTAURATION DES MESSAGES ALTÉRÉS

Dans ce chapitre, nous explorons l'état de l'art des APE. À ce titre, nous nous focalisons sur les travaux les plus significatifs. Plus loin, nous présentons les motifs des APE et des informations importantes sur l'élément enveloppant et l'élément enveloppé. Pour finir, nous abordons brièvement notre technique de détection et de restauration qui est détaillée dans le chapitre 5.

4.1. INTRODUCTION

Parmi les spécifications de sécurité des SW, XML-Signature est celle conçue pour assurer l'intégrité du message ainsi que son authenticité, et/ou celle de l'émetteur. XML-Signature applique correctement une signature sur un objet. En effet, sans connaître la bonne clé, il est impossible de modifier le contenu de l'objet signé sans compromettre l'intégrité de la signature. Toutefois, lors de la vérification de la signature, l'information permettant de retrouver l'objet signé peut se révéler insuffisante pour détecter une délocalisation de l'objet signé suite à une APE. Cette délocalisation est rendue non détectable à cause de certaines flexibilités de XML-Signature et de SOAP.

Parvenir à détecter les APE constitue déjà une avancée. Mais en regardant de près, nous constatons qu'il est possible de restaurer, dans certains cas, le message original suite à la détection de son altération par une APE. Cette idée vient du fait qu'il est possible de dégager des motifs aux APE. En général, une APE consiste à envelopper un élément par un autre élément. Parfois, une nouvelle instance de l'élément attaqué, mais avec un nouvel contenu, est fournie par le pirate et mise à l'emplacement initial de l'élément attaqué. Ainsi, nous avons pensé qu'une investigation devait être menée à ce niveau, car une telle restauration apporte plusieurs avantages dont une économie sur la bande passante, et un meilleur temps de traitement, car elle prévient toute retransmission du message altéré.

4.2. ÉTAT DE L'ART

Dans la littérature, les APE ont été peu étudiées. Les travaux que nous avons consultés proposent des solutions basées soit sur une technique formelle, soit sur l'analyse de la structure du message SOAP. Malheureusement, ces travaux ne proposent pas de solutions complètes, ou proposent des solutions qui ne sont pas efficaces, et/ou nécessitent des données et d'opérations cryptographiques supplémentaires dans le message SOAP. En fait, tous ces travaux se limitent à la simple détection des APE [19-24], sans aborder la question de restauration du message altéré après détection de l'attaque. Nous croyons être les premiers à entreprendre une telle investigation.

4.2.1. Prévention basée sur les outils XML existants

4.2.1.1 XML-Encryption

On peut crypter l'élément signé afin de le rendre illisible lors de la communication réseau. Cela permet d'empêcher la lecture de l'information utile lorsque le message est intercepté par un pirate. Cependant, cette solution présente des problèmes de performance dans le cas de l'envoi de gros messages à cause des opérations cryptographiques qu'elle engendre. La solution présente également des problèmes de performance lorsque le message passe par plusieurs intermédiaires qui doivent accomplir un traitement sur l'élément signé après avoir vérifier sa signature.

4.2.1.2 Utilisation de Schema XML

On peut fixer la syntaxe d'un message à un certain nombre d'éléments définis dans un Schema XML. Malgré tout, cette solution à elle seule n'est pas suffisante pour détecter certains APE. Par exemple, le Schema XML peut ne pas détecter ces cas d'APE:

- l'élément signé est remplacé par un élément légitime différent
- l'élément enveloppant est une entrée d'en-tête (sous-élément légitime de `soap:Header`), et il utilise l'attribut `role` mis à `none`.

4.2.1.3 Référencement de l'objet signé à l'aide d'un filtre de transformation XPath

Dans la spécification XML-Signature, nous avons vu que l'élément `Reference` permet de localiser l'objet signé grâce à son attribut `URI`. Cependant, il est possible d'imposer la localisation basée sur un filtre de transformation XPath [18].

Considérons le message SOAP suivant dans lequel l'entrée d'en-tête `ReplyTo` a été référencée via un filtre de transformation XPath.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="..."
xmlns:wsa="...">
  <soap:Header>
     <wsse:Security>
        <ds:Signature>
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="..." />
                <ds:SignatureMethod Algorithm="... " />
                <ds:Reference URI="#body">
                    ...
               </ds:Reference>
                <ds:Reference>
                   <ds:Transforms>
                       <ds:Transform Algorithm="...">
                         <ds:XPath ...>
                          /soap:Envelope/soap:Header/wsa:ReplyTo
                         </ds:XPath>
                       <ds:Transform>
                   </ds:Transforms>
              </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>...</ds:SignatureValue>
            <ds:KeyInfo>
               ...
            </ds:KeyInfo>
```

```
        </ds:Signature>
    </wsse:Security>
    <wsa:ReplyTo id="replyTo">
      <wsa:Address>http://serviceweb.com/service1</wsa:Address>
   </wsa:ReplyTo>
  </soap:Header>
  <soap:Body id="body">
     <GetLastTradePrice>
        <symbol>DIS</symbol>
     </GetLastTradePrice>
  </soap:Body>
</soap:Envelope>
```

Dans le cas de la localisation via un filtre de transformation XPath, l'attribut `Reference/@URI` est vide, et cela permet d'identifier la racine du document XML comme nœud contexte auquel est appliquée la transformation XPath. Même s'il arrive que deux éléments signés portant un même nom de balise soient voisins, il est toujours possible de les distinguer via une expression XPath, par exemple en spécifiant la valeur de leur attribut `id` qui est unique. Par conséquent, cette solution semble la mieux adaptée. Pour des raisons que nous ignorons aucun auteur n'a vraiment insisté sur cette solution.

4.2.2. Détection basée sur WS-SecurityPolicy

Le travail de M. McIntosh *et al.* [19] fut le premier à présenter les APE. Les auteurs proposent des idées de solution basées sur WS-SecurityPolicy pour contrer quelques scénarios d'attaque. Il est possible de détecter certaines APE grâce à l'utilisation de WS-SecurityPolicy qui permet de définir des politiques de sécurité relatives au contenu d'un message SOAP. Par exemple, avec WS-SecurityPolicy on peut spécifier que la présence de tel élément est obligatoire ou non, quel type de jeton de sécurité sera utilisé lors de l'échange, quel élément du message doit être signé ou chiffré, quel algorithme doit être utilisé pour la signature, etc. Ce travail a beaucoup de mérite, cependant il ne fournit pas de solution complète aux APE. Par ailleurs, utiliser WS-SecurityPolicy pour détecter une APE exige la maîtrise de WS-SecurityPolicy dont la syntaxe pour exprimer certains faits peut se révéler très complexe, et engendre dans le message SOAP une grande quantité de données non nécessaires. C'est la principale limitation de WS-SecurityPolicy.

4.2.3. Détection basée sur le travail SOAP Account

Les chercheurs M. A. Rahaman *et al.* [20-21] ont abordé le problème des APE du point de vue de la structure du message SOAP, et ils ont élaboré une solution nommée `SOAP Account`. Les auteurs pensent que l'APE est la conséquence d'une absence d'information sur la structure du message SOAP. Ces informations sont le nombre d'éléments sous `Envelope`, le nombre d'entrées d'en-tête, le nombre de référence(s) dans chaque élément `Signature`, et enfin les informations de parenté et de voisinage pour chaque élément signé.

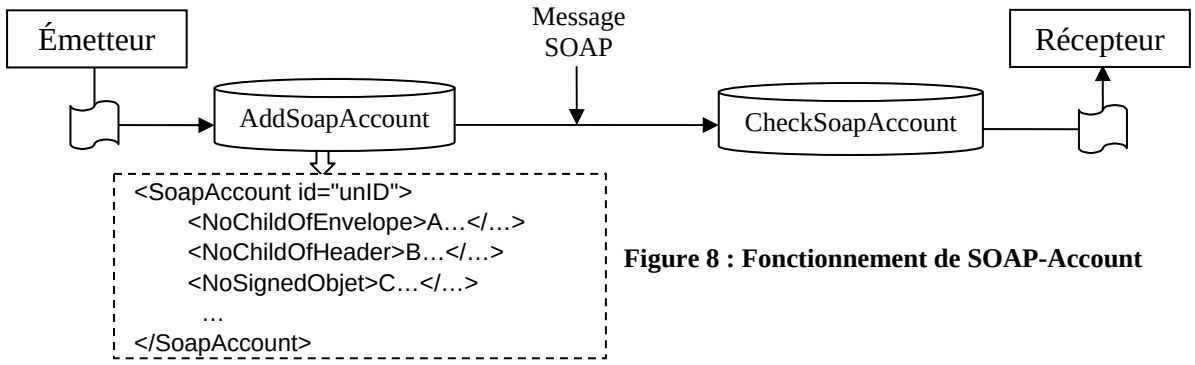


Figure 8 : Fonctionnement de SOAP-Account

`SOAP Account` fonctionne de la manière suivante: avant l'envoi d'un message, un module appelé AddSoapAccount insère un élément `SoapAccount` contenant des informations sur la structure XML telles que mentionnées précédemment. À la réception du message, un module similaire appelé CheckSoapAccount vérifie que le contenu du message est fidèle aux informations contenues dans l'élément `SoapAccount`.

Ainsi selon les auteurs, l'utilisation de `SOAP Account` permet au récepteur de détecter une APE si celle-ci survient lors de l'envoi d'un message. Cependant, il a été prouvé que cette technique ne détecte pas toutes les APE [22-24]. De plus, cette technique peut présenter des problèmes de performance dans les cas suivant:

- la taille du message est importante;
- le message passe par des intermédiaires qui ajoutent leur signature au message. Ainsi le nombre d'éléments `SoapAccount` dans le message peut devenir très grand. Cette technique peut rapidement compromettre la performance dans le cas d'une architecture faisant intervenir plusieurs SW.

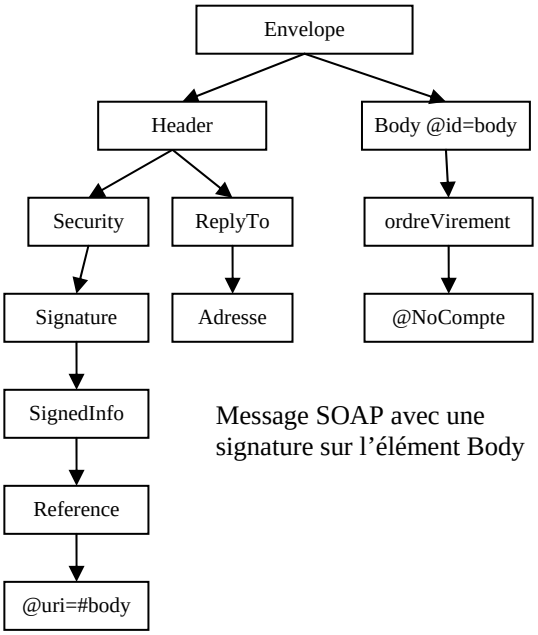
Il faut également mentionner que `SOAP Account` exige des opérations cryptographiques supplémentaires sur le message SOAP.

4.2.4. Détection basée sur la valeur de retour de la fonction de Signature

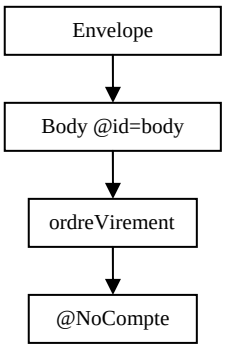
Le travail de S. Gajek *et al.* [22] dévoile les faiblesses du travail publié dans [20-21], puis aborde le problème des APE en insistant sur la valeur de retour du module de vérification de la signature. Les auteurs proposent deux idées:

a. Retourner le sous arbre de SOAP Envelope avec uniquement l'objet signé
Lorsque la vérification d'une signature réussie, au lieu de retourner la valeur booléenne `true`, le module de vérification de la signature retourne au processus applicatif l'arborescence complète de l'élément signé.

Soit le message signé suivant:



Message SOAP avec une signature sur l'élément Body



Valeur retournée par le module de vérification de la signature au processus applicatif dans le cas d'une vérification réussie

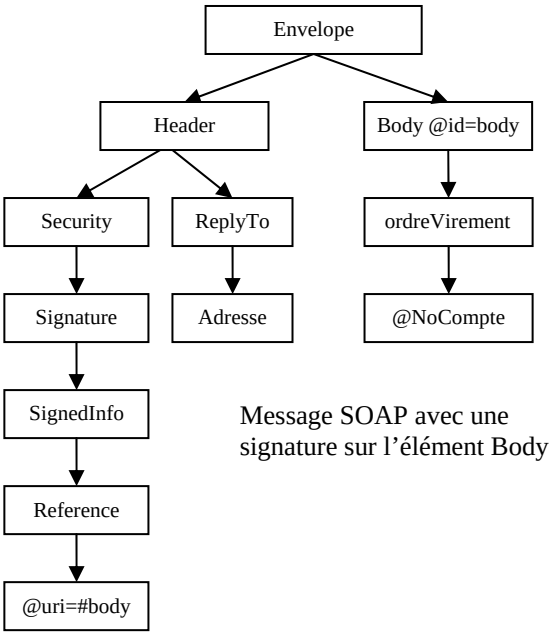
Figure 9 : Modèle de solution basé sur la valeur de retour de la fonction de signature

b. Retourner la location de l'objet signé

Au lieu de retourner l'arborescence de l'objet signé depuis la racine, une autre solution est de retourner le chemin XPath de l'objet signé depuis la racine, voir prochaine illustration.

Pour ce qui est de la première solution, il semble évident que retourner l'arborescence complète de l'élément signé depuis la racine posera un problème de performance dans le cas de gros document XML avec une grande profondeur. La deuxième idée, qui ressemble à l'utilisation du filtre de transformation XPath, semble plus intéressante. Cette solution, une fois implémentée, peut détecter les APE. Cependant, en cas d'APE il n'y a aucun moyen de restaurer le message SOAP. Par conséquent, cette solution exige la retransmission du message SOAP altéré. C'est la limite de ce travail.

Soit le message signé suivant:



Message SOAP avec une signature sur l'élément Body

Dans le cas d'une vérification réussie, cette valeur est retournée au processus applicatif :

```
/Envelope/Body[id=body]
```

4.2.5. Détection basée sur l'exploitation de la profondeur et de la parenté de l'élément signé

Le travail de A. Benameur *et al.* [23] expose les faiblesses des spécifications de sécurité WS-Security, WS-Signature, WS-Policy, WS-SecurityPolicy ainsi que le travail publié dans [20-21]. Les auteurs proposent d'incorporer dans une signature, la profondeur ainsi que les informations de parenté de l'élément signé. Puisque deux éléments dans un message SOAP pourraient porter le même nom de balise, ce qui rend l'information sur la parenté insuffisante, les auteurs proposent également d'utiliser des identifiants « id » afin d'identifier de façon absolue chaque parent d'un élément signé. Selon les auteurs les informations combinées sur la profondeur et sur la parenté sont suffisantes pour détecter toute forme d'APE. Malheureusement, les auteurs ne soutiennent pas leur idée par un exemple concret. Il n'y a pas de précision sur comment intégrer les informations citées ci-haut dans la signature d'une quelconque façon. De plus, l'ajout d'informations supplémentaires dans le message SOAP peut avoir des incidences négatives sur la performance, car ces informations augmentent avec le nombre d'élément(s) à signer.

4.2.6. Détection basée sur une technique formelle

Le travail de S. Kumar Sinha *et al.* [24] utilise une approche basée sur une méthode formelle. Les auteurs pensent que l'absence de contexte de signature explique la faille de XML-Signature. Ainsi, ils définissent le concept de signature non contextuelle (Context-Free Signature, CFS) et le concept de signature contextuelle (Context-Sensitive Signature, CSS). Au niveau du CFS, la vérification d'une signature d'un message nécessite deux choses: le message lui-même ainsi que sa signature. De ce fait, il modélise à l'aide du CFS un message SOAP signé à l'aide de XML-Signature, car ce dernier n'utilise pas de contexte.

La signature non contextuelle (Context-Free Signature, CFS)

Au niveau du CFS, un message signé prend la forme $\{\partial_M\}_{SA}$, où ∂_M est le résultat d'une fonction de hachage à sens unique sur un message M, $\{\partial_M\}_{SA}$ est le chiffrement du hash ∂_M par la clé sécrète d'un agent A; il s'agit de la signature. Pour vérifier la signature du message, deux étapes sont nécessaires:

1) $\{\{\partial_M\}_{SA}\}_{PA}$: déchiffrer le message à l'aide de la clé publique de A afin d'obtenir ∂_M;

2) Calculer le hash de M soit ∂'_M et le comparer à ∂_M.

Si $\partial'_M = \partial_M$, par conséquent on en conclut que ce message provient de A. Dans le contexte d'une signature non contextuelle, il n'y a pas une grande relation entre le message lui-même et sa signature à part le fait que le message est signé par la clé sécrète du signataire. Cette technique simpliste a été adoptée afin d'éviter les grosses opérations cryptographiques coûteuses sur le message au complet. Ainsi au lieu de signer le message au complet, on signe uniquement son empreinte (son hash ou digest). L'inconvénient de la signature non contextuelle est l'absence de relation entre la signature et le message, ce qui constitue justement la raison pour laquelle les APE sont possibles.

La signature contextuelle (Context-Sensitive Signature, CSS)

Au niveau du CSS, un message signé porte, en plus de sa signature, le contexte de signature. Ici un message signé prend la forme $\{\partial_M,\partial_C,t\}_{SA}$, où ∂_M est le résultat d'une fonction de hachage à sens unique sur un message M, ∂_C est le résultat d'une fonction de hachage à sens unique sur le contexte de signature au temps t, t représente le temps durant lequel la signature

a été générée, et enfin$\{\partial_M,\partial_C,t\}_{SA}$ est l'application de la signature à l'aide de la clé sécrète d'un agent A. Cette fois-ci, pour vérifier la signature du message, trois étapes sont nécessaires:

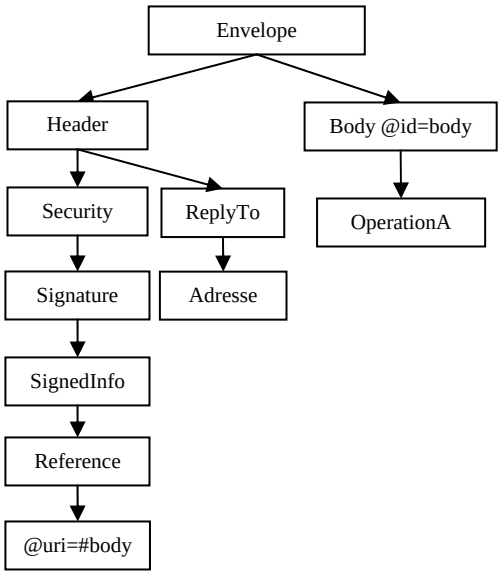
1) $\{\partial_M,\partial_C,t\}_{SA}$: déchiffrer le message à l'aide de la clé publique de A afin d'obtenir ∂_M, ∂_C et t;

2) Calculer le hash de M soit ∂'_M et le comparer à ∂_M;

3) Générer C' à partir de M et t, ensuite calculer le hash de C' soit ∂'_C, et le comparer à ∂_C;

Si $\partial'_M = \partial_M$ et $\partial'_C = \partial_C$, par conséquent on en conclut que le message a été signé au temps t, et qu'il provient bel et bien de A.

À chaque contexte de signature inclut un contexte de sécurité. Pour illustrer le fonctionnement de cette solution, nous prenons un exemple. Considérons le message SOAP de la prochaine figure.



Ici, l'élément signé est *Body*. Le contexte de signature est :
Envelope ← (*Header,* [*Body*])

Le contexte de sécurité correspondant est:
Envelope ← (*Header,Body*)
Header ← (*Security,ReplyTo*)
Security ← (*Signature*)
Signature ← (*SignatureInfo*)
SignatureInfo ← (*Reference,* [*Reference*])

Figure 10 : Exemple de contexte de signature

Le contexte de signature et le contexte de sécurité correspondant sont des règles de dérivation depuis l'élément racine `Envelope`, le premier menant vers l'élément signé et le second menant vers la référence à l'élément signé. Ces informations sont hachées et stockées dans l'élément `signedInfo`. La signature est donc calculée à partir du hash de l'élément signé et du hash du contexte de signature.

Ainsi lors de la réception du message, le destinataire doit régénérer la signature et son contexte afin de les vérifier. Mais il y'a un problème. Le contexte de signature dépend fortement de la structure du message au moment de sa signature. Toute manipulation du message, même par un agent légitime, risque fort de détruire le contexte de signature. Par exemple, imaginons que ce message passe par un intermédiaire qui doit ajouter un nouveau traitement sous l'élément `Header`, disons un élément nommé `OtherLegalProcess`. La règle de dérivation vers l'élément signé reste le même, mais le contexte de sécurité devient :
Envelope ← (*Header,Body*)
Header ← (*Security,ReplyTo,OtherLegalProcess*)
Security ← (*Signature*)
Signature ← (*SignatureInfo*)

SignatureInfo ← (*Reference,* [*Reference*])

Cette opération légale d'un intermédiaire légitime a modifié le contexte de signature généré par l'émetteur. Remarquons qu'un intermédiaire, au lieu d'ajouter un nouvel élément, peut consommer un élément existant avant de réacheminer le message vers son destinataire. Consommer un élément signifie le traiter et l'enlever du message. En fait, plusieurs situations peuvent nécessiter une modification de la structure du message original. Ainsi, les auteurs proposent un contexte de signature adaptif qui consiste pour un intermédiaire de régénérer le nouveau contexte de signature reflétant la nouvelle structure du message. Malheureusement, la façon dont le contexte de signature est généré par un intermédiaire n'est pas illustrée par un exemple concret. Plus important, cette technique permet de vérifier une signature en prenant uniquement en compte le chemin XML entre l'élément racine `Envelope` et l'élément signé ainsi que sa référence dans `SignInfo`. Par conséquent, l'introduction par le pirate d'un nouvel élément en dehors de ces chemins ne sera pas détectée : typiquement en bas de tous les éléments signés. Si ce nouvel élément est reconnu, alors il est possible qu'il soit traité. Même si cette attaque peut ne pas constituer une APE, elle prouve que cette technique ne détecte pas toutes formes de réécritures d'un message SOAP, contrairement à ce que les auteurs le prétendent.

Cette solution offre une piste exploitable pour contrer les APE. Mais en présence d'intermédiaire, elle semble mal adaptée. Par exemple, pour qu'un intermédiaire puisse modifier le contexte de signature, il doit connaître la clé de vérification de la signature, cela même si le traitement de l'intermédiaire ne touche pas l'élément signé. Nous pensons qu'une clé de vérification de la signature doit être connue uniquement des agents qui agissent directement sur l'élément signé, et non pas n'importe quel agent qui peut modifier la structure du message. De plus, la solution ne supporte pas la restauration. Toute détection d'APE nécessite une retransmission du message altéré. Par ailleurs, nous pensons que les données que la solution requière dans le message SOAP peuvent être évitées.

4.3. OBJECTIFS

L'objectif général de ce travail consiste à améliorer les techniques de détection des APE, afin de proposer une solution pour la restauration d'un message SOAP altéré par une APE. Toujours dans le même sens, nous nous fixons comme objectifs spécifiques:

1) de catégoriser les APE afin de mieux les traiter;
2) d'évaluer les possibilités de détection et de restauration pour chaque catégorie;
3) de proposer un algorithme de détection et un algorithme de restauration pour chaque catégorie d'attaque, si cela est possible. Dans le cas contraire, fournir les raisons pour lesquelles de telles actions ne sont pas faisables;
4) de montrer, à travers des exemples, le fonctionnement des algorithmes de détection et de restauration pour chaque catégorie que nous aurons identifiée à l'étape 2.

4.4. MÉTHODOLOGIE

Afin d'atteindre nos objectifs, nous comptons adopter une approche analytique.

1) Catégorisation des APE
La catégorisation vise à mieux identifier les APE, et aussi à faire d'un problème difficile une multitude de problèmes plus facile à traiter. Grâce à notre connaissance des APE, et aussi aux travaux portant sur ce type d'attaque, nous dresserons une liste de scénarios d'attaque réalistes. Suite à cela, une analyse de chaque cas est effectuée. Cette analyse tient compte d'un certain nombre d'informations qui seront établies au fur et à mesure que nous détaillons le problème.

2) Évaluation de la possibilité de détection et de restauration
La restauration ne vise pas forcement à reproduire syntaxiquement le message original, mais un message ayant la même sémantique, et dont le traitement donnera le même résultat. Tous les messages altérés ne sont pas récupérables. Pour qu'un message soit récupérable, il faudrait préalablement identifier un certain nombre de critères. Ainsi l'évaluation de la possibilité de restauration se déroulera suivant l'exécution d'un ensemble de tests à effectuer sur un certain nombre de critères.

3) Algorithme de détection et de restauration
Nos algorithmes de détection et restauration sont basés sur l'analyse syntaxique du message SOAP, il est donc important qu'ils aient accès aux informations les plus fiables qui soient sur le contenu des messages SOAP qu'ils sont sensés traiter. Ces informations incluent le nom des éléments (nom de balise) du message SOAP, le caractère obligatoire ou optionnel de chaque élément, la classe de chaque élément ainsi que l'emplacement exact de chaque élément. Tout ceci est abordé en détail au chapitre 5.

4.4.1. Un mot sur notre approche

Nous pouvons distinguer deux situations dans un environnement d'exécution des SW: la situation dans laquelle il n'y a pas d'intermédiaires actifs dans la chaîne de transmission et la taille des messages est relativement petite, et la situation dans laquelle il y'a des intermédiaires actifs sur la chaîne de transmission et/ou la taille des messages est importante.

Pour ce qui est de la simple détection des APE, la signature et/ou le chiffrement du message au complet est la solution idéale dans la première situation. Dans la deuxième situation, nous ne pouvons envisager la signature et/ou le chiffrement du message au complet en raison d'un souci de performance ou de peur à rendre impossible la déclaration de toute nouvelle signature dans le message. Dans ce cas, une solution de prévention/détection efficace des APE est nécessaire. Cependant, pour ce qui est de la restauration du message SOAP altéré par une APE, il n'existe pas de solution prenant en charge cette question. Par conséquent, nous pensons que notre solution est pertinente dans l'une ou l'autre des situations.

4.4.2. Analyse des APE

Une APE est caractérisée par l'élément enveloppant et l'élément enveloppé. L'élément enveloppé est l'élément attaqué, et il peut être:

- un élément obligatoire ou optionnel (du point de vue de sa présence dans le message);
- un élément à traitement obligatoire ou optionnel;
- un élément à position fixe ou non fixe au sein de son parent;
- un élément à parent unique ou à parent multiple ;

En fait, l'élément enveloppant peut ne pas être destiné au nœud SOAP désigné ou peut ne pas être reconnu par ce dernier. Cela constitue la raison pour laquelle l'attaque réussit. L'élément enveloppant n'est pas toujours un élément fictif tel qu'illustré dans les scénarios d'attaque 1 et 2 de la section 3.3. Effectivement, il peut arriver que l'élément enveloppant porte un nom XML connu du nœud SOAP désigné. Le scénario d'attaque 3 de la section 3.3 présente une telle situation. Parfois, l'élément enveloppé est reproduit à son emplacement original, avec un nouveau contenu fourni par le pirate. Toutefois, dans les scénarios d'attaque 2 et 3, on remarque que l'élément enveloppé n'est pas reproduit ailleurs dans le message SOAP. Cela est possible parce que l'élément enveloppé est un élément optionnel du message SOAP. Notons que l'élément enveloppé représente toujours une opération à part entière, c'est-à-dire qu'il contient les informations suffisantes pour un traitement spécifique du côté destinataire.

Comment rendre un élément inconnu à un nœud SOAP ?

Il suffit de choisir pour l'élément un nom de balise qui n'existe pas dans la grammaire du nœud SOAP.

Comment rendre le traitement d'un élément optionnel ?

Il suffit de lui assigner un attribut `mustUnderStand` avec la valeur `false.`

Comment ne pas destiner un élément à un nœud SOAP ?

Il suffit de lui assigner un attribut `role` avec une valeur différente du rôle du nœud SOAP. Lorsqu'un nœud SOAP n'a pas le même role que la valeur de l'attribut `role` d'un élément, le nœud SOAP ne doit pas le traiter. La valeur spéciale `none` de `role` sous un élément signifie qu'aucun nœud SOAP ne doit traiter cet élément.

Par conséquent, l'élément enveloppant doit être une entrée d'en-tête afin de pouvoir utiliser les attributs `role` et `mustUnderStand` de SOAP. À ce niveau, il faut s'assurer que, non seulement l'élément enveloppant n'est pas destiné au nœud SOAP désigné, mais aussi qu'il n'est destiné au traitement d'aucun élément de ce nœud SOAP. Ainsi sa présence dans le message, surtout sa présence en tant qu'élément parent de l'élément enveloppé, lui qui est destiné au nœud SOAP désigné, est suspecte. Par ailleurs, nous rappelons que certains éléments peuvent avoir l'attribut `role` à `none`, mais utilisés dans le traitement d'autres éléments tel que cela a été mentionné dans le point b) de la section 1.2.1.3. La spécification SOAP permet la présence de tels éléments.

D'après les scénarios d'attaque réalistes, une APE consiste à:

a) reproduire l'élément enveloppé avec un nouveau contenu,

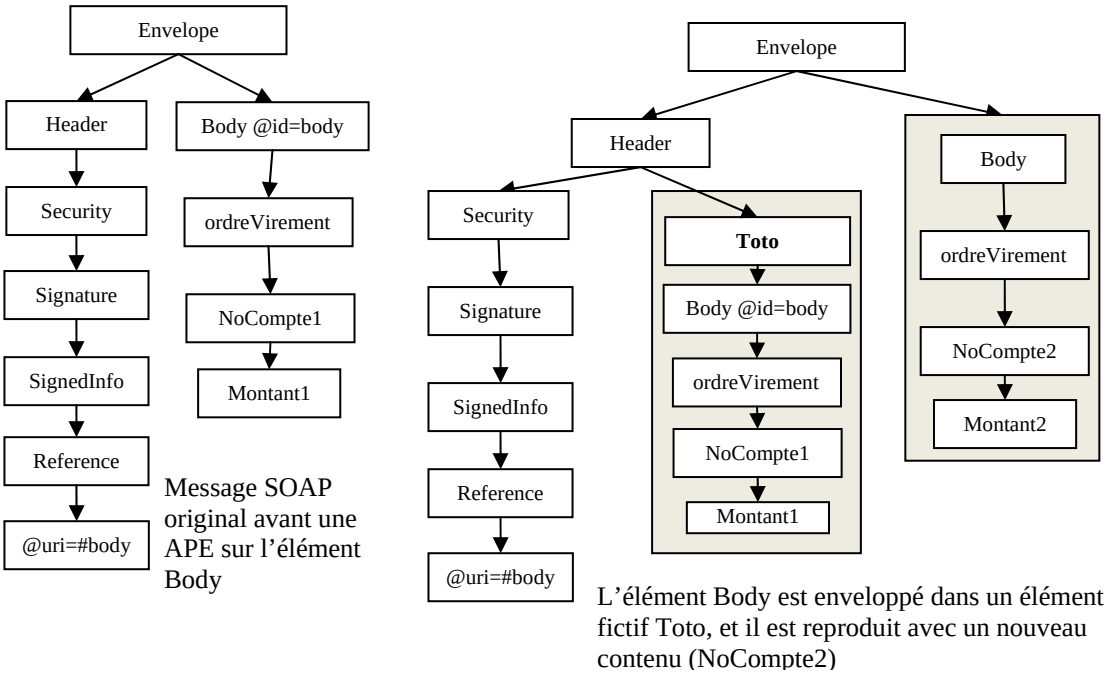


Figure 11 : APE – scénario d'une reproduction d'élément avec nouveau contenu

b) ignorer un élément optionnel (sans le reproduire)

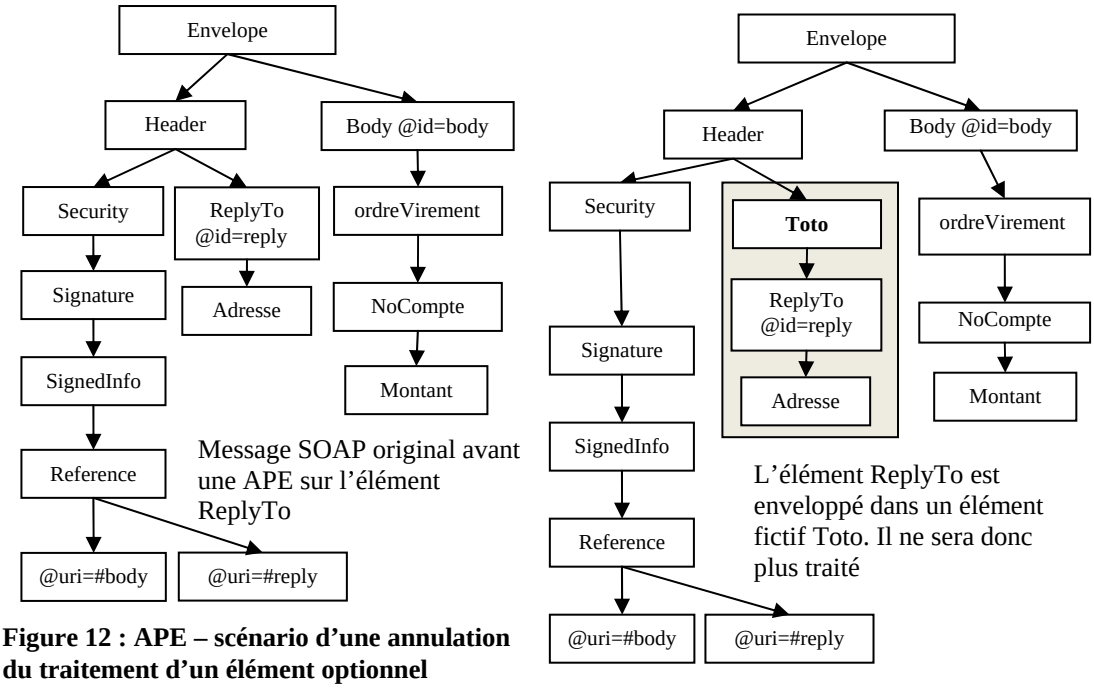


Figure 12 : APE – scénario d'une annulation du traitement d'un élément optionnel

4.4.3. Définitions et notations

Soit A = {A1,…,An} une liste non ordonnée de nœuds XML de M, M étant un message SOAP original ou le message SOAP résultant d'une APE. Soit N = {N1,…,Nm} un ensemble ordonné fini de nœuds SOAP sur la chaîne de transmission de M, de telle sorte que Ni < Nj si Ni vient avant Nj dans la chaîne de transmission.

Notation 1: Ai $\leftrightarrow$ Aj signifie que Ai et Aj ont le même nom de balise.

Notation 2: Ai = Aj signifie que Ai est identique à Aj du point de vue syntaxique (nom de balise, noms d'attributs et contenu).

Notation 3: Aj $\wr$ Ai signifie que Aj est le parent direct de Ai. On écrira également: Aj $\wr_x$ Ai pour exprimer le fait que Ai est le $x^{ème}$ enfant de Aj, x étant un entier positif.

Notation 4: Aj $\otimes$ Ai signifie que Ai est le seul enfant de Aj. Par conséquent, Aj $\otimes$ Ai implique Aj $\wr_1$ Ai.

Définition 1: on dit que Aj est destiné à Nk si Aj a été généré par Ni pour être traité par Nk, avec Ni < Nk, k et i étant des entiers positifs avec k supérieur à i (Ni vient avant Nk sur la chaîne de transmission).

Notation 5: Ni[A1,A2,..,Ak] signifie que A1, A2,..,Ak sont destinés à Ni.

Définition 2: on dit que Ni reconnaît Aj si Ni a prévu un traitement pour Aj.

Notation 6: Ni⟨[A1, A2,..,Ak]⟩ signifie que A1, A2,.., Ak sont destinés à Ni, et ce dernier les a reconnus.

Notation 7: *Sig*(Ai) signifie que Ai est signé.

Notation 8: *Optional*(Aj,Ni) signifie que Aj, dans le contexte de Ni, est un élément optionnel du point de vue de sa présence dans M.

Notation 9: *OptionalProcess*(Aj,Ni) signifie que le traitement de Aj par Ni est optionnel, autrement dit, si Ni ne parvient pas à traiter à Aj, peu importe la raison, Ni peut l'ignorer sans générer d'erreur.

Notation 10: *SYNTAX*(Ai,…,Ak) désigne l'ensemble des constructions syntaxiques possibles de Ai,…,Ak. On écrit: *Syntax*(…) pour désigner une construction particulière de SYNTAX(…). Par exemple:

- *Syntax*(Ai $\wr$ Aj) désigne les constructions syntaxiques de *SYNTAX*(Ai, Aj) dans lesquelles Ai est le parent de Aj, ou encore
- *Syntax*(*Sig*(Ai)) désigne les constructions syntaxiques de *SYNTAX*(Ai,…) dans lesquelles Ai est signé.

Notation 11: écrire: Nx[*Syntax*(…)] signifie que Nx reconnaît cette construction syntaxique particulière de *SYNTAX*(…). Par exemple, Nx[*Syntax*(*Sig*(Ai))] signifie que Nx reconnaît les constructions syntaxiques de SYNTAX(Ai, …) dans lesquelles Ai est signé.

Notation 12: l'opérateur de négation est dénoté par le signe « $\neg$ » devant un autre opérateur ou un prédicat. Par exemple, on écrira: Ai $\neg\leftrightarrow$ Aj, si Ai n'a pas le même nom de balise que Aj, ou encore $\neg$*Sig*(Ai) signifie que Ai ne porte pas de signature.

Définition 3: on dit que Ai est un élément ayant un parent fixe dans le contexte de Nx, si pour toute construction syntaxique *Syntax*(Aj $\wr$ Ai) dans *SYNTAX*(Ai, Aj, …), on a toujours: Nx[*Syntax*(Aj $\wr$ Ai)] et Nx[*Syntax*(Ap $\wr$ Ai)] implique Aj $\leftrightarrow$ Ap, pour tout Ap $\in$ A.

Définition 4: on dit que Ai est un élément à parent multiple dans le contexte de Nx, s'il existe dans *SYNTAX*(Ai, Ap, Aj, …), au moins deux constructions syntaxiques *Syntax*(Ap ≀ Ai) et *Syntax*(Aj ≀ Ai) telles que Nx[*Syntax*(Ap ≀ Ai)] et Nx[*Syntax*(Aj ≀ Ai)], avec Aj ¬↔ Ap.

Définition 5: on dit que Ai est un élément à position fixe dans le contexte de Nx, si pour toute construction syntaxique *Syntax*(Aj ≀ Ai) dans *SYNTAX*(Ai, Aj, …), on a toujours: Nx[*Syntax*(Aj $\wr_k$ Ai)], avec k étant un entier naturel constant.

Définition 6: on dit que Ai est un élément à position non fixe dans le contexte de Nk, s'il existe dans *SYNTAX*(Ai, Aj, …) au moins deux constructions syntaxiques *Syntax*(Aj ≀ Ai) et *Syntax*(Ap ≀ Ai) telles que Nk[*Syntax*(Aj $\wr_x$ Ai)] et Nk[*Syntax*(Ap $\wr_z$ Ai)], x et z étant des entiers naturels différents, et Ap appartenant à A.

Définition 7: on dit que Aj représente une opération pour Ni lorsque la présence de l'arborescence de Aj nécessite un traitement du côté Ni. L'arborescence de Aj doit être suffisante et minimale. La seule restriction sur Aj est qu'il ne doit pas être une déclaration de signature (élément `Signature`). La raison pour cela est parce que nous allons exiger une signature sur toutes les opérations de notre service. Or nous n'avons pas besoin de signer à nouveau une signature. La prochaine figure illustre quelques exemples d'opérations.

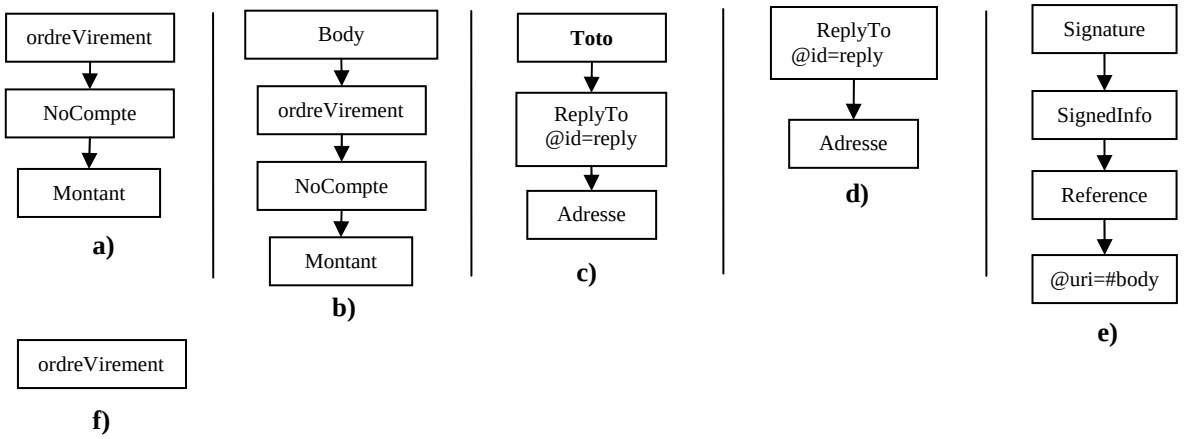


Figure 13 : Exemples d'éléments représentant une opération

En considérant le message SOAP de la figure 12, les éléments **a)** et **d)** de la figure 13 sont des opérations. Les éléments **b)**, **c)**, **e)** et **f)** ne sont pas des opérations parce que respectivement : la présence de l'élément `Body` n'est pas nécessaire, l'élément `Toto` est inconnu, nous ne considérons pas une déclaration de signature comme étant une opération, l'élément `ordreVirement` tout seul n'est pas suffisant. Dans ce dernier cas, il faut noter que même si `Toto` avait été reconnu, le résultat sera le même pour la même raison que l'élément **b)**.

4.4.4. Configuration de notre modèle

Les conditions suivantes doivent être observées pour que notre solution fonctionne correctement. Lorsqu'un nœud SOAP, recevant un message SOAP, constate le non-respect de l'une des conditions, il doit rejeter le message:

1. un nœud SOAP désigné pour le traitement d'un élément signé doit pouvoir trouver une déclaration de signature (pour cet élément) qui lui soit adressée. En fait, quelque soit la

solution de signature utilisée, assurer l'intégrité d'une information nécessite cette condition. Si le destinataire n'exige pas de signature sur l'information qu'il traite, la tâche d'un pirate consiste à simplement supprimer la signature de l'élément avant d'acheminer le message;

2. un nœud SOAP désigné pour le traitement d'un élément signé doit pouvoir déterminer l'identité de l'émetteur, par exemple, à partir de sa clé de signature. À cet effet, les spécifications de sécurité permettent à l'émetteur de proposer plusieurs modes de récupération de clé à travers des méthodes sûres. Cette condition est également nécessaire, car elle empêche le pirate d'utiliser n'importe quelle clé pour signer des éléments du message SOAP après les avoir modifié à sa guise;
3. tout élément représentant une opération doit être signé en entier. Il est logique de préserver l'intégrité d'une opération de façon globale;
4. un élément enveloppé (élément signé) ne doit pas être inconnu pour le nœud SOAP désigné.

4.4.5. Détection des APE : les motifs de l'attaque

Sur la base de scénarios d'attaque réalistes, nous avons retenu deux catégories d'APE: la reproduction d'élément avec nouveau contenu, ou l'annulation du traitement d'un élément optionnel.

4.4.5.1 Motif de la reproduction d'élément avec un nouveau contenu

Soient M un message SOAP; $Nx \in N$ un nœud SOAP qui reçoit M; et Ai, Aj, Ap, $Am \in A$ des éléments XML de M. M a subi une APE consistant à une reproduction d'élément avec un nouveau contenu si Nx constate dans M:

- $Sig(Ai) \wedge Nx[Ai] \wedge Nx\langle[Ai]\rangle$
- $Ai \leftrightarrow Ap \wedge Ai \neg= Ap \wedge \neg Sig(Ap) \wedge Nx\langle[Ap]\rangle$
- $Aj \wr Ai \wedge \neg Sig(Aj)$
- $\neg Nx[Aj] \vee (Nx[Aj] \wedge \neg Nx\langle[Aj]\rangle \wedge OptionalProcess(Aj,Nx))$

Sémantique

M a subi une APE consistant en la reproduction d'élément avec un nouveau contenu, si Nx constate dans M que:

- il existe un élément Ai signé, destiné à Nx et reconnu par celui-ci
- il existe un élément Ap ayant le même nom de balise que Ai, mais avec un contenu différent. Ap est destiné à Nx et reconnu par celui-ci
- Ai a un parent Aj qui:
  - n'est pas destiné à Nx ni à aucun autre traitement de Nx, ou
  - est destiné à Nx, mais n'est pas reconnu par ce dernier, et il a un traitement optionnel
- Aj et Ap ne sont pas des éléments signés.

4.4.5.2 Motif d'un élément optionnel ignoré

Soient M un message SOAP; $Nx \in N$ un nœud SOAP qui reçoit M; et Ai, Aj, Ap, $Am \in A$ des éléments XML de M. M a subi une APE consistant à ignorer le traitement d'un élément optionnel si Nx constate dans M:

- $Sig(Ai) \wedge Nx[Ai] \wedge Nx\langle[Ai]\rangle \wedge Optional(Ai,Nx)$
- $Aj \wr Ai \wedge \neg Sig(Aj)$
- $\neg Nx[Aj] \vee (Nx[Aj] \wedge \neg Nx\langle[Aj]\rangle \wedge OptionalProcess(Aj,Nx))$

Sémantique

M a subi une APE consistant à ignorer le traitement d'un élément optionnel si Nx constate dans M que:

- il existe un élément optionnel Ai signé, destiné à Nx et reconnu par celui-ci
- Ai a un parent Aj qui n'est pas signé et qui :
  - n'est pas destiné à Nx ni à aucun autre traitement de Nx, ou
  - est destiné à Nx, mais n'est pas reconnu par ce dernier, et il a un traitement optionnel.

4.4.6. Restauration du message SOAP après détection d'une APE

Nous venons de voir comment reconnaitre une APE à travers un motif. Maintenant, nous présentons la démarche à suivre afin de restaurer un message SOAP altéré par une APE.

4.4.6.1 Étapes de restauration

La restauration d'un message SOAP suite à la détection d'une APE consiste en ces étapes:
a) déplacer l'élément enveloppé vers son emplacement original;
b) enlever du message l'élément enveloppant, et éventuellement l'élément reproduit dans le cas d'une reproduction d'élément.

Le point b) est facile à réaliser dès lors que l'on identifie l'élément enveloppant, et éventuellement l'élément reproduit. Cependant, le point a) peut se révéler délicat. Avant de pouvoir déplacer l'élément enveloppé vers son emplacement original, il convient d'identifier d'abord sa classe. Une fois sa classe identifiée, on met en œuvre la stratégie de replacement associée à cette classe, si toutefois la restauration est possible.

4.4.6.2 Les classes d'éléments

La classe d'un élément indique si sa position est fixée ou pas dans son parent, et s'il peut ou non avoir plusieurs parents possibles dans la grammaire. Quatre cas sont possibles.

a) Élément de classe 1 : élément à position fixe et à parent unique

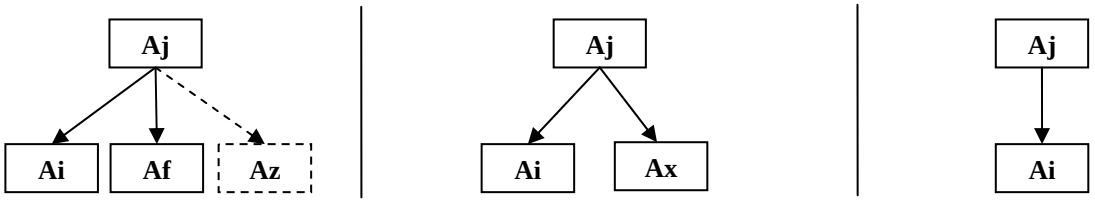


Figure 14 : Exemple d'élément de classe 1

L'élément Ai est de classe 1: il a un parent unique dans la grammaire du service (Aj), et une position fixe sous son parent dans la grammaire du service (position 1).

b) Élément de classe 2 : élément à position non fixe et à parent unique

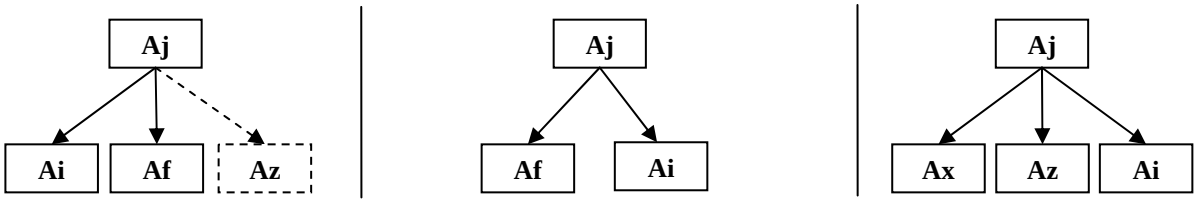


Figure 15 : Exemple d'élément de classe 2

L'élément Ai est de classe 2: il a un parent unique dans la grammaire du service (Aj), mais n'a pas de position fixe sous son parent dans la grammaire du service (tantôt Ai apparait à la position 1, tantôt à la position 2, etc).

c) Élément de classe 3 : élément à position fixe et à parent multiple

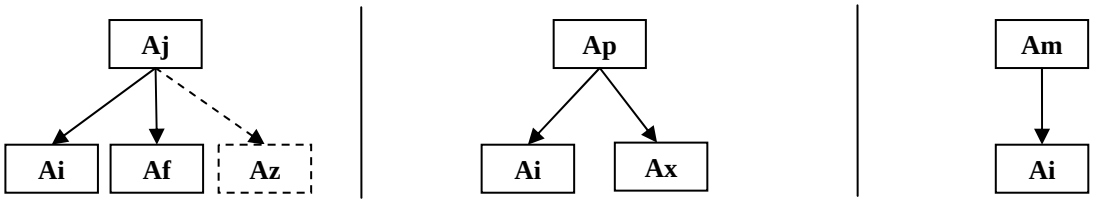


Figure 16 : Exemple d'élément de classe 3

L'élément Ai est de classe 3: il a des parents multiples dans la grammaire du service (Aj, Ap, Am), et une position fixe sous ses parents dans la grammaire du service (position 1).

d) Élément de classe 4 : élément à position non fixe et à parent multiple

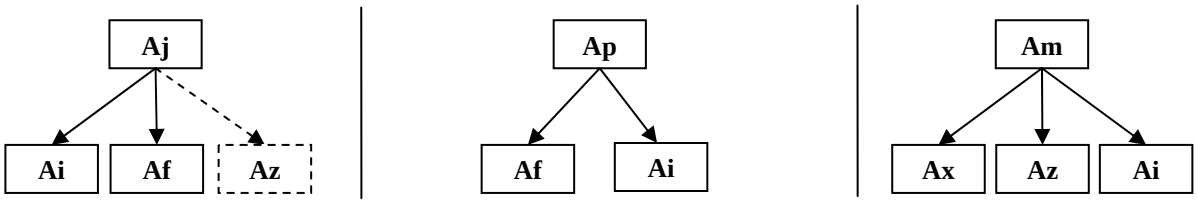


Figure 17 : Exemple d'élément de classe 4

L'élément Ai est de classe 4: il a des parents multiples dans la grammaire du service (Aj, Ap, Am), et n'a pas de position fixe sous ses parents dans la grammaire du service (tantôt Ai apparait à la position 1, tantôt à la position 2, etc).

4.4.6.3 Stratégie de replacement de l'élément enveloppé

Un élément enveloppé est replacé selon sa classe par une stratégie décrite dans le prochain tableau. Le replacement des éléments de classe 1 et 2 est simple puisqu'ils n'ont qu'un parent unique. Un élément de classe 3 et 4 est celui qui peut légitimement être sous-élément de plus qu'un élément. Dans ce cas, la détermination de l'emplacement original de l'élément enveloppé est nécessaire, et dépend du nombre d'emplacements possibles trouvés dans le message. Si le message contient un seul élément sous lequel l'élément enveloppé peut apparaître, alors l'élément enveloppé est déplacé:

- sous cet élément à sa position exacte s'il est de classe 3, ou
- sous cet élément à la dernière position s'il est de classe 4.

Lorsque le message contient plus qu'un élément sous lequel l'élément enveloppé peut apparaître, alors la restauration n'est pas faisable.

Classe	Élément	Stratégie de replacement
1	Position fixe et parent unique	Déplacer sous son parent à sa position exacte
2	Position non fixe et parent unique	Déplacer sous son parent à n'importe quelle position. Par souci d'uniformité, nous considérerons la dernière position
3	Position fixe et parent multiple	Replacement conditionnel
4	Position non fixe et parent multiple	Replacement conditionnel

Stratégie de replacement de l'élément enveloppé

4.4.6.4 Exemples de restauration d'un message SOAP altéré

Soit Ni un nœud SOAP recevant un message SOAP, identifie un élément enveloppant Aj et un élément enveloppé Ax. Ni restaure le message altéré de la manière suivante:

a) S'il est de classe 1 : élément à position fixe et à parent unique

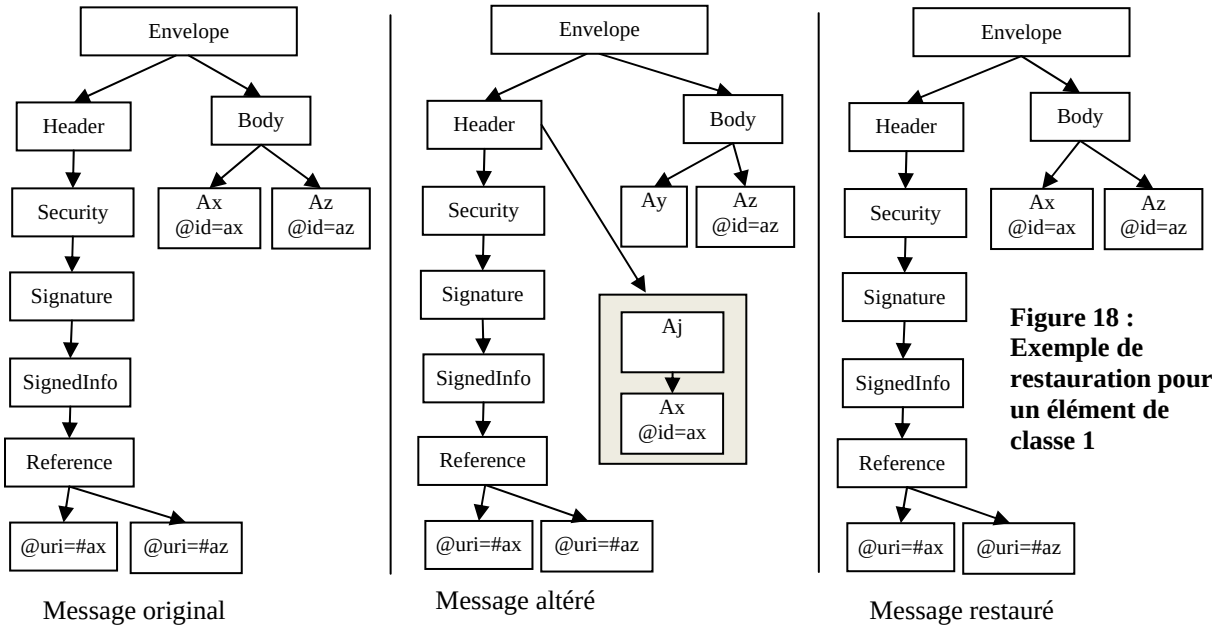


Figure 18 : Exemple de restauration pour un élément de classe 1

De gauche à droite, nous avons le message SOAP original, le message altéré par une APE, et le message restauré. L'élément enveloppé Ax est remis sous son parent `Body` et à sa position exacte, soit la position 1.

b) Élément de classe 2 : élément à position non fixe et à parent unique

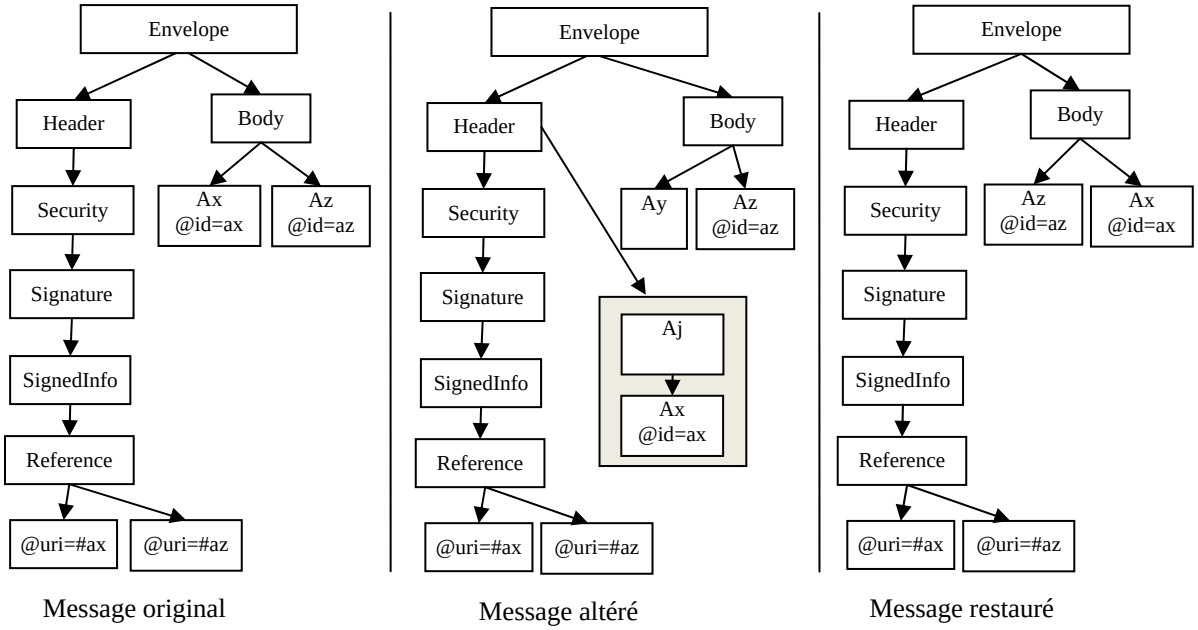


Figure 19 : Exemple de restauration pour un élément de classe 2

Dans ce cas, l'élément enveloppé Ax est remis sous son parent `Body` et à la dernière position puisqu'il n'a pas une position fixe.

c) Élément de classe 3 : élément à position fixe et à parent multiple

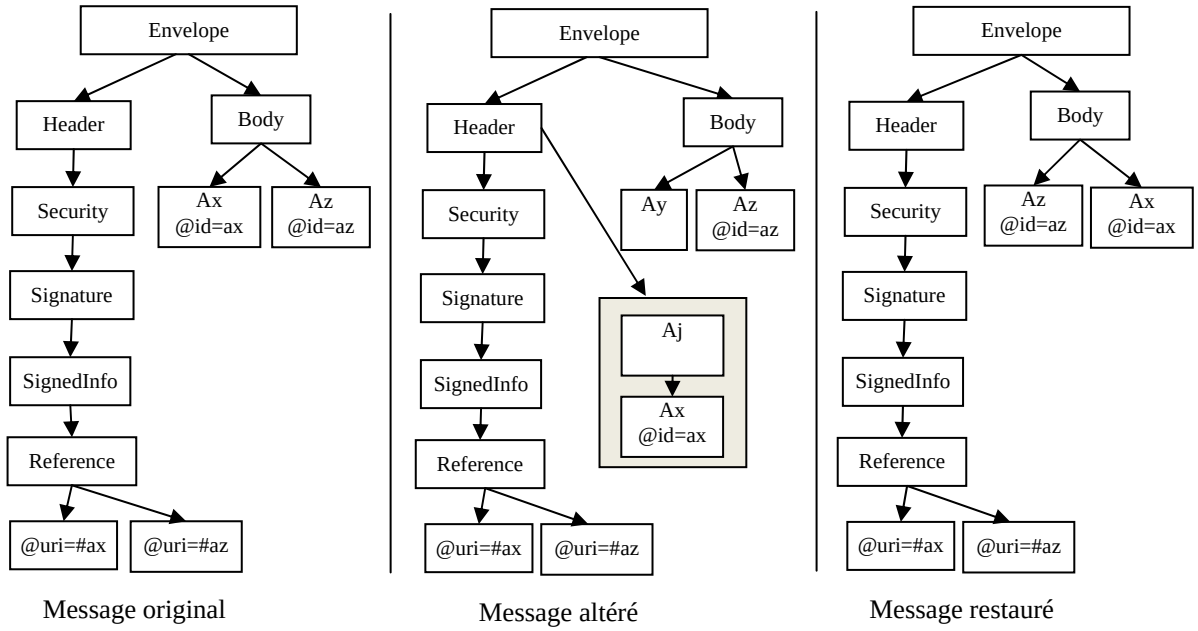


Figure 20 : Exemple de restauration pour un élément de classe 3

Si dans le message SOAP courant, l'élément `Body` est le seul parent possible de l'élément enveloppé Ax, alors Ax est remis sous `Body` à la première position. En plus de l'élément `Body`, s'il existe dans le message SOAP un autre élément, par exemple l'élément `Header`, pouvant être le parent de Ax, alors la restauration est impossible. Dans ce cas, un message d'erreur est généré.

d) Élément de classe 4 : élément à position non fixe et à parent multiple

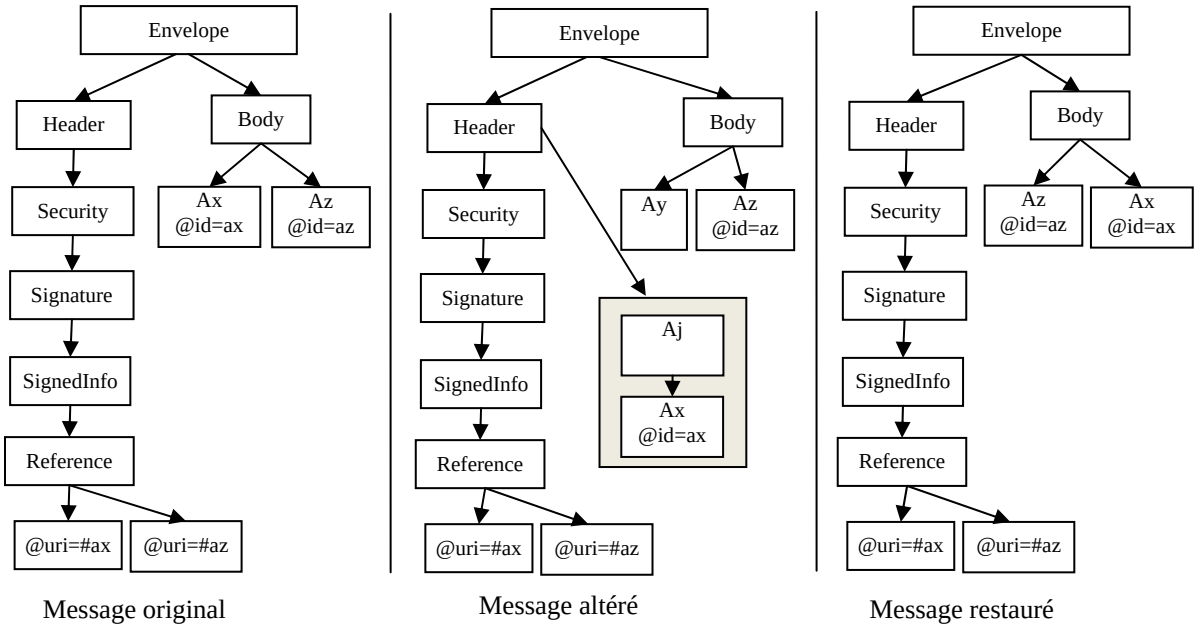


Figure 21 : Exemple de restauration pour un élément de classe 4

Si dans le message SOAP courant, l'élément `Body` est le seul parent possible de l'élément enveloppé Ax, alors Ax est remis sous `Body` à la dernière position. En plus de l'élément `Body`, s'il existe dans le message SOAP un autre élément, par exemple l'élément `Header`, pouvant être le parent de Ax, alors la restauration est impossible.

4.4.7. Fondement de notre technique

4.4.7.1 La technique de détection des APE

Notre technique de détection est fondée sur l'idée selon laquelle tout élément signé d'un message SOAP doit être traité; aucun élément signé ne peut exister pour rien. En réalité, cette hypothèse est également valable pour tout élément non signé, mais cela est hors sujet.

a) Identification de l'élément enveloppant :

Rappelons qu'une APE réussit pour l'une des deux raisons suivantes:

- l'élément enveloppant a un traitement optionnel et il est inconnu au nœud SOAP désigné, ou ;
- l'élément enveloppant n'est pas destiné au nœud SOAP désigné (bien qu'il contienne un élément destiné à ce dernier). Dans ce cas, l'élément enveloppant peut ou non être reconnu par le nœud SOAP désigné, cela n'a pas d'importance.

Les seules formes qu'un élément enveloppant peut prendre étant désormais connues, un nœud SOAP recevant un message SOAP n'a plus qu'à vérifier si parmi les éléments qui lui sont adressés, il existe un élément signé enveloppé par un élément possédant l'une de ces formes d'enveloppement.

b) Identification de l'élément enveloppé :

Nous avons vu qu'un élément enveloppé est celui enveloppé par un élément enveloppant. Il peut être optionnel ou obligatoire, reproduit sous un parent légal ou ne pas être reproduit du tout. Les formes exploitables d'un élément enveloppé étant ainsi réduites, cela garantie

l'identification de tout élément enveloppé quelque soit la catégorie d'APE qui survient sur un message SOAP.

c) Identification de la reproduction de l'élément enveloppé :

La reproduction est l'élément par lequel le pirate remplace l'élément enveloppé. Cet élément porte généralement le même nom que l'élément enveloppé, mais avec un contenu différent. Si tel est le cas, l'identification de la reproduction est simple. Mais on peut imaginer qu'un pirate, après avoir étudié la grammaire de nos messages, remplace l'élément enveloppé par un élément légitime de nom différent. Dans un tel cas, l'identification de la reproduction ne marchera pas puisqu'il s'agit de deux opérations distinctes, par exemple remplacer l'opération `getProductStock` par `addToStock`. Cependant, notre modèle exige que toute opération de nos messages soit signée, sinon le message SOAP est d'office rejeté.

Remarque :

Notre modèle propose une technique de détection des APE et de restauration du message SOAP altéré par une APE. Cependant, rien n'interdit de l'utiliser uniquement pour la détection des APE. Dans une telle circonstance, seules les contraintes 1 et 2 de la section 4.4.4 sont nécessaires, les autres peuvent être supprimées.

4.4.7.2 La technique de restauration

La technique de restauration est fondée sur la grammaire des messages traités par le service web. Normalement, la grammaire d'un service web est établie depuis la conception du service. Lorsqu'une APE est détectée, les identités de l'élément enveloppant et de l'élément enveloppé sont connues, ainsi que l'éventuelle reproduction de l'élément enveloppé. Puisque la grammaire renseigne sur la structure du message SOAP, alors le ou les parents possibles de l'élément enveloppé sont d'office connus. Nous observons deux cas pour la remise de l'élément enveloppé à son emplacement original (son parent original). L'élément enveloppé a :

- un parent unique dans la grammaire (classe 1 ou 2) ou ;
- des parents multiples dans la grammaire (classe 3 ou 4).

Dans l'un ou l'autre des cas, l'élément enveloppé est remis sous son seul parent possible qui a été préalablement identifié à travers le message SOAP. Si le nombre de parents possibles pour l'élément enveloppé, dans le message, est différent de 1, le message est rejeté.

Si l'élément enveloppé a pu être remis sous son parent, il ne reste plus qu'à supprimer du message SOAP l'élément enveloppant et éventuellement la reproduction de l'élément enveloppé.

Puisque toute autre opération non signée entraine le rejet du message, alors la restauration, une fois confirmée, préserve la sémantique du message original.

Remarque importante :

Selon notre modèle, la restauration peut ne pas être possible dans le cas où l'emplacement d'un élément enveloppé n'a pas pu être déterminé de façon unique. Cependant, dans la réalité la restauration est possible dans la majorité des cas. En effet, un message SOAP admet deux types de traitements : les traitements applicatifs et les directives. Les traitements applicatifs sont des sous éléments directs de l'élément `Body`*. Les directives sont des traitements informant un nœud SOAP sur la façon de traiter certains éléments. Les directives concernent tout autre traitement non applicatif, par exemple : une déclaration de signature, de chiffrement, d'assertions, etc. Une directive est soit un sous élément direct de l'élément*

`Header`, `Security` *ou parfois* `Body`. *Le nombre d'élément Body et Header dans un message SOAP est au plus un. Cela est une exigence de la spécification SOAP[2]. Le nombre d'élément* `Security` *pouvant être destiné à un nœud SOAP en particulier est au plus un. Cela est une exigence de la spécification WS-Security[5]. Un élément enveloppé est toujours un traitement : soit une directive ou un traitement applicatif. Par conséquent, dans la grande majorité des cas, le parent original d'un élément enveloppé est soit* `Header`, `Body` *ou* `Security`*, des éléments toujours uniques dans le contexte de chaque nœud SOAP.*

CHAPITRE 5: NOTRE MODÈLE

Dans ce chapitre, nous abordons notre modèle. Nous présentons les algorithmes ainsi que les informations qu'ils requièrent. Par la suite, nous illustrons la mise en œuvre de notre solution par un exemple complet. Nous terminons par une présentation de l'implémentation de notre solution.

5.1. INTRODUCTION

Notre modèle, illustré par la prochaine figure, propose à la fois la détection des APE, et la restauration d'un message SOAP après détection de l'attaque.

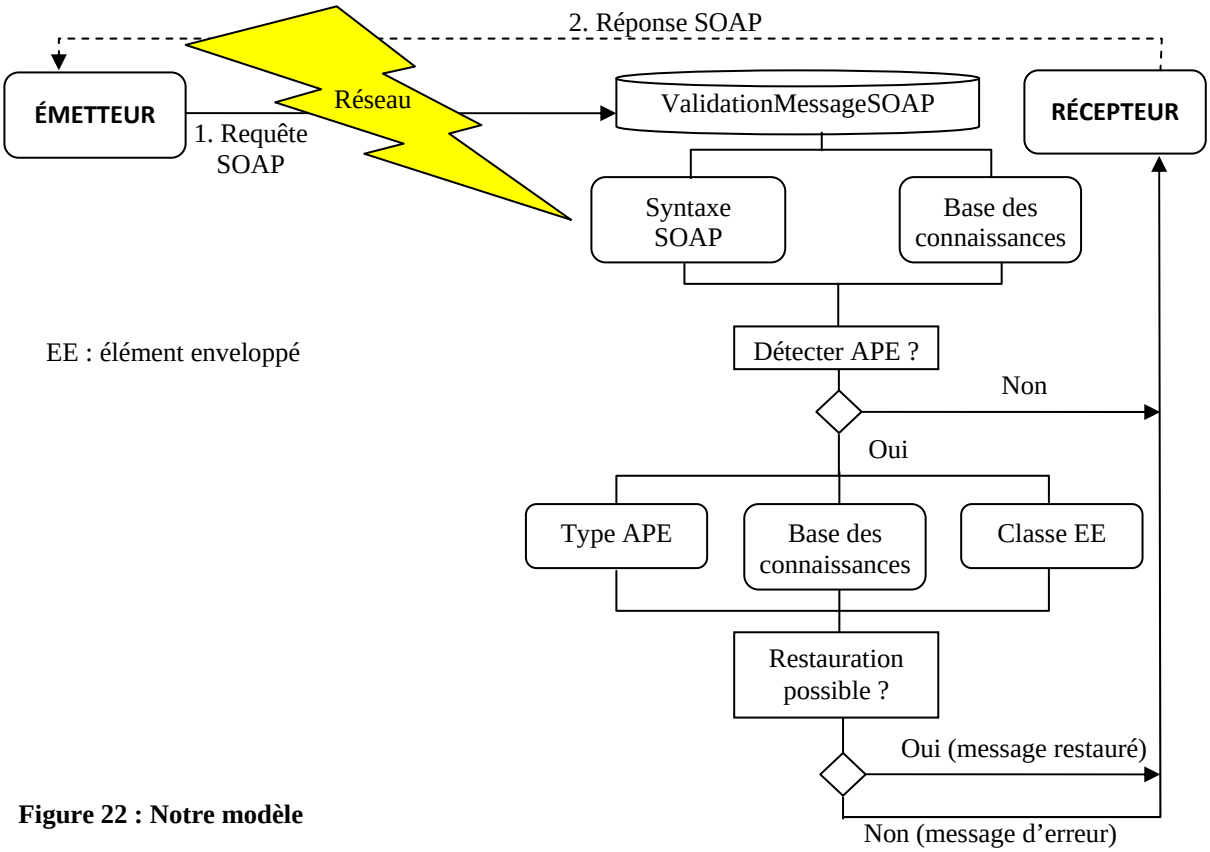


Figure 22 : Notre modèle

À sa réception, le message SOAP passe par notre module qui lui applique une série de vérifications à travers notre algorithme de détection. Cet algorithme qui est présenté plus loin, a besoin de certaines informations établies sur le contenu des requêtes SOAP traitées par le service web. En cas d'absence d'APE, le message est passé au prochain gestionnaire SOAP. Lorsqu'une APE est détectée, l'algorithme de restauration prend le relais. En plus du type d'APE, l'algorithme de restauration a besoin d'informations telles que: la classe de l'élément enveloppé et son emplacement exact (original). Lorsque la restauration est possible, et qu'elle ait été effectuée avec succès, le message restauré est passé au prochain gestionnaire de message SOAP. Si la restauration n'est pas faisable, alors un message d'erreur est généré.

5.2. LES CONNAISSANCES RÉQUISES

Dans cette section, nous parlons des informations nécessaires à nos algorithmes de détection et de restauration. Nos algorithmes étant basés sur l'analyse syntaxique du message SOAP, alors ils doivent avoir accès à des informations exactes sur les messages SOAP qu'ils traitent. Ces informations incluent le nom d'élément (nom de balise) des messages SOAP, le caractère obligatoire ou optionnel de chaque élément, la classe de chaque élément ainsi que l'emplacement exact de chaque élément. Du point de vue de la conception, un message SOAP est constitué exclusivement de deux catégories d'éléments (et attributs): les éléments « communs » et les éléments spécifiques. Les éléments communs sont ceux qui peuvent se retrouver dans tous les messages SOAP, peu importe le type de service. Il s'agit d'éléments issus des spécifications des SW: SOAP, WS-Security, XML-Signature, etc. Ces éléments sont caractérisés par leur syntaxe et leur sémantique qui ne changent pas d'un service à un autre. Par contre, les éléments spécifiques, quant à eux, sont particuliers à chaque service. Ils représentent généralement le traitement applicatif du message à travers l'élément `Body`. Les éléments spécifiques se distinguent par leur syntaxe et leur sémantique qui sont choisies au gré du concepteur de service.

5.2.1. Les éléments communs d'un message SOAP

Tel que mentionné précédemment, les éléments communs sont ceux relatifs aux spécifications des SW. Les informations sur ces éléments peuvent constituer une quantité de donnée vaste. Heureusement que ces dernières sont durables, voire permanentes pour les spécifications déjà établies. Pour un concepteur de service web, il est rare d'avoir à utiliser toutes les spécifications des SW disponibles dans un même service, car pour bon nombre de services, seule la sécurité relative au chiffrement, à la signature ainsi qu'à la distribution de clé suffisent. De plus, certaines spécifications n'autorisent pas d'extension, c'est-à-dire l'utilisation d'éléments hors mis ceux déjà prévus entraîne automatiquement l'échec du traitement. Il n'est donc pas nécessaire de s'intéresser à toutes les spécifications des SW, mais seulement celles dont on fait usage et qui soient vulnérables à une APE.

5.2.2. Les éléments spécifiques d'un message SOAP

Les informations relatives aux éléments communs sont nécessaires pour nos algorithmes de détection et de restauration, mais ces informations ne sont pas suffisantes. Pour mener à bien une détection et une restauration, nous avons également besoin d'informations relatives aux éléments spécifiques des messages. Nous avons déjà dit que ces éléments sont généralement des sous-éléments de `Body`. Mais il peut aussi s'agir d'éléments au niveau des entrées d'en-tête (`Header`), afin de donner des directives spéciales sur comment le message doit être traité. Les informations sur les éléments spécifiques peuvent constituer une quantité de données plus petite que celle des éléments communs, mais de nature beaucoup plus changeante à cause de l'évolution des pratiques.

L'ensemble de ces informations constituent la grammaire du service web. À partir de maintenant, nous désignons sous le nom de base des connaissances ces informations une fois établies. Le lecteur peut, s'il le désire, jeter un coup d'œil à la section 5.4.4 pour avoir une idée sur une structure d'une telle base des connaissances.

5.3. LES ALGORITHMES

Dans cette section, nous présentons nos algorithmes de détection et de restauration, leur fonctionnement est illustré dans les sections 5.4.6 et 5.4.7. Afin d'aider à mieux comprendre l'algorithme de détection, nous commençons par fournir un algorithme spécifique pour chaque catégorie d'APE. Plus loin, un algorithme général pour la détection des APE est fourni. Pour ne pas rallonger les algorithmes, et aussi pour ne pas répéter certains traitements, nous avons regroupé certaines parties des algorithmes dans des fonctions. Parmi ces fonctions, seules les plus importantes sont détaillées.

Les variables suivantes sont utilisées par nos deux algorithmes et bon nombre des fonctions. Le lecteur peut se familiariser avec ces variables avec leur rôle associé:

- `m` = message soap
- `w[]` = liste des éléments signés destinés au nœud SOAP
- `a` = catégorie d'APE
- `e` = éventuel élément enveloppé
- `n` = éventuel élément enveloppant
- `eCopied` = éventuelle reproduction de e
- `eClass` = classe de e
- `eStatus` = statut de e
- `K` = Base des connaissances
- `header` = élément SOAP Header
- `body` = élément SOAP Body
- `Role` = identifiant de l'attribut role
- `This_Node_Role` = role du nœud soap courant
- `MustUnderStand` = identifiant de l'attribut mustUnderStand

Remarque:
Certaines contraintes exprimées sur notre modèle dans la section 4.4.4 ne sont pas prises en compte dans les algorithmes pour des raisons de simplicité.

5.3.1. Reproduction d'élément avec nouveau contenu

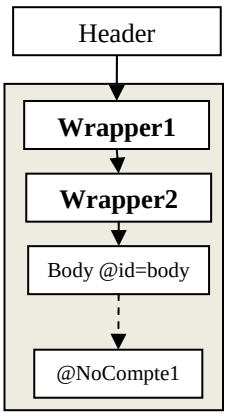
À présent, voyons l'algorithme de détection des APE consistant à la reproduction d'élément avec nouveau contenu. Afin de mieux voir le lien entre l'algorithme et l'APE à laquelle il se rapporte, le lecteur peut se reporter soit à la section 4.4.4 pour voir le motif de cette attaque, soit à la section 4.4.2 où la figure 10 représente un message SOAP original (à gauche), et le même message SOAP altéré par une APE consistant à la reproduction d'élément avec nouveau contenu (à droite):

`[1]` En premier, nous récupérons la liste des éléments signés qui sont adressés au nœud SOAP courant. Ces valeurs sont stockées dans la variable `w[]`.

`[2-22]` Ensuite, nous entamons une série de vérifications en entrant dans une boucle `foreach` qui parcourt chacun des éléments de `w[]`. À chaque élément `e` de `w[]`, on fait les traitements suivants:

- ligne 3-4: les variables `a` et `n` sont initialisées respectivement à 0 (absence d'attaque) et `null`;

- ligne 5-9: l'élément enveloppant, s'il existe, est identifiée à ce stade. Il s'agit du parent inconnu de `e`. Un élément est considéré comme inconnu si au moins l'une des conditions suivantes est satisfaite:
  a) son traitement est facultatif et la syntaxe utilisée (nom de balise, contenu, parenté) n'est pas conforme à celles spécifiées dans la base des connaissances;
  b) l'élément n'est pas destiné au nœud SOAP courant (ni au traitement d'aucun élément de ce dernier). L'élément inconnu n'est pas forcement le parent direct de l'élément enveloppé, car en effet, il peut y avoir plusieurs éléments enveloppants, tel que illustré dans la prochaine figure. Dans cette dernière situation, le parent inconnu le plus haut dans l'arborescence sera retourné (`Wrapper1` dans cet exemple).



Ici, les éléments `Wrapper1` et `Wrapper2` sont tous inconnus. Dans ce cas, c'est `Wrapper1` qui est retourné. La flèche pointillée signifie que certains éléments ont pu être omis.

Figure 23 : Cas d'APE où l'élément enveloppant est multiple

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: message SOAP original
Sortie: message SOAP résultant du traitement de l'algorithme
CheckForElementReproduction(m)
1 : w[] ← GetSignedElts(m)
2 : foreach e in w[] do
3 :   a ← 0 /* 0 = absence d'APE, 1 = APE-reproduction, 2 = APE-annulation */
4 :   n ← null
5 :   t ← GetCurrentParent(e,m)
6 :   while t != header and t != body do
7 :     if (t not in K and IsOptProc(t)) or !IsTargetedTo(t,m) then
8 :       n ← t
       endif
9 :     t ← GetCurrentParent(t,m)
     endwhile
10:  if n != null and n not signed then
      /* À ce stade, nous savons qu'il existe un élément enveloppant et un élément
      enveloppé */
11:    eCopied ← FindCopied(e,m)
12:    if eCopied != null then /* eCopied peut être une liste d'éléments */
13:      a ← 1
14:      r ← Restore(a,n,e,m,eCopied)
15:      if r != null then /* message restauré */
16:        m ← r
17:      else
18:        m ← error
19:        break
         endif
       endif
     endif
    endforeach
20: return m
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Détection des APE consistant à la reproduction d'élément

- ligne 10-19: si l'élément enveloppant existe (`n` différent de `null`), alors nous recherchons une éventuelle reproduction (variable `eCopied`) de `e`. Nous confirmons une APE consistant à la reproduction d'élément si `eCopied` et `n` sont différents de `null` et ne sont pas signés. Dans ce cas, a est instanciée à 1, l'indicateur de reproduction d'élément, puis l'algorithme de restauration est appelé avec les paramètres: `a`, `n`, `e`, `m` et `eCopied`. Lorsque la restauration réussit le message restauré est retourné, sinon un message d'erreur est généré et l'algorithme s'arrête;

Remarque :
La condition while t != header and t != body a pour but d'arrêter la recherche d'un éventuel élément enveloppant lorsque l'élément légitime Header ou Body est rencontré. Alors la vérification doit inclure le nom de balise de l'élément t, mais également sa profondeur puisque l'élément enveloppé ou l'élément enveloppant peut être nommé Header ou Body. À noter qu'un élément légitime Header ou Body apparaît toujours comme sous élément direct de l'élément racine Envelope.

À présent, voyons chacune des fonctions utilisées dans cet algorithme.

Fonction GetSignedElts: cette fonction retourne tous les éléments signés du message, et qui sont destinés au nœud SOAP courant. Un élément est considéré comme signé s'il est référencé par un élément `Signature/SignedInfo/Reference`, ce dernier étant sous-élément de l'élément `Security` destiné au nœud SOAP courant, voir section 2.2.2.

Fonction IsTargetedTo: cette fonction détermine si l'élément passé en paramètre est destiné ou non au nœud SOAP courant. Plus exactement, cette fonction vérifie à travers le message si l'élément en question, ou un ancêtre, possède un attribut `role` avec une valeur correspondant au `role` du nœud SOAP courant. Dans ce cas, la fonction retourne `true`. Rappelons que l'absence de cet attribut équivaut à sa présence avec la valeur `ultimateReceiver`. Cette fonction retourne également `true` si l'élément possède l'attribut `role` à `none`, mais est destiné au traitement d'un autre élément du nœud SOAP courant. Si un élément est destiné au traitement d'un autre élément, alors ce dernier doit exister dans le message et être destiné au nœud SOAP courant. L'information consistant à déterminer si un élément participe dans le traitement d'un autre élément, se trouve dans la base des connaissances.

Fonction IsOptProc: cette fonction détermine si l'élément passé en paramètre a un traitement optionnel ou non. Il ne faut pas confondre un élément optionnel avec un élément dont le traitement est optionnel. Un élément optionnel est celui dont la présence n'est pas requise dans le message, contrairement à un élément obligatoire dont l'absence provoquera une erreur. Un élément dont le traitement est optionnel est celui, ou un ancêtre, qui possède un attribut `mustUnderStand` à `false` (voir point b. section 1.2.1.3). Nous rappelons que l'absence de cet attribut équivaut à sa présence avec la valeur `false`. Nous rappelons également que le caractère obligatoire ou non du traitement d'un élément peut être déterminé à partir du statut du nœud SOAP courant. Par exemple: le traitement de `Body` est obligatoire pour le destinataire.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: m
Sortie: w[]
GetSignedElts(m)
1 :  w[] ← null
2 :  sec ← null /* sec = élément Security destiné au nœud soap courant */
3 :  se[] ← GetSecurityElements(m) /* se[] = éléments Security de m */
4 :  foreach s in se[] do
```

```
5 :    if IsTargetedTo(s,m) then
6 :      sec ← s
7 :      break
       endif
     endforeach
8 :  ref[] ← GetReferenceElts(sec) /* ref[] = éléments Reference de sec */
9 :  i ← 0;
10: foreach r in ref[] do
11:   w[i] ← RetrieveSignedElt(r)
12:   i ← i + 1
    endforeach
13: return w[]
```

Fonction GetSignedElts : Récupération des éléments signés du

```
Entrée: n
Sortie: re (true / false)
IsTargetedTo(n,m)
1 : re ← false
2 : attR ← GetAttNode(Role,n)
3 : if attR = This_Node_Role then
4 :   re ← true
5 : else if attR = none then
6 :   if IsUseInOtherProc(n) then
7 :     ps[] ← GetProcElts(n,K)
8 :     foreach x in ps[] do
9 :       if x in m and IsTargetedTo(x,m) then
10:         re ← true
11:         break
          endif
        endforeach
      endif
    endif
12: return re
```

Fonction IsTargetedTo : Détection d'un élément destiné au nœud SOAP courant

```
Entrée: n
Sortie: re (true / false)
IsOptProc(n)
1 : re ← false
2 : atM ← GetAttNode(MustUnderStand,n)
3 : if (atM = false or atM = null) and
     (This_Node_Role != ultimateReceiver and n != body) then
4 :   re ← true
    endif
5 : return re
```

Fonction IsOptProc : Détection d'un élément à traitement optionnel

Fonction FindCopied: cette fonction retourne l'élément (la reproduction), s'il existe, qui est destiné au nœud SOAP courant et ayant le même nom de balise que celui passé en paramètre, mais avec un contenu différent. S'il existe plusieurs éléments de ce genre, alors ils seront retournés dans une structure de liste.

La reconnaissance de la syntaxe de l'élément reproduit est également une condition de sa validité. La fonction FindCopied agit à la fois sur le message et sur la base des connaissances pour s'assurer que l'emplacement de cette reproduction est bien conforme à une syntaxe

valide. Si cette fonction rencontre un élément ayant le même nom que l'élément en question, et qui est destiné au nœud SOAP courant, mais non reconnu par ce dernier, alors cet élément est automatiquement supprimé du message.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: e,m
Sortie: eCopied
FindCopied(e,m)
1 : eCopied[] ← null
2 : es[] ← FindSameNameElts(e,m)
3 : i ← 0
4 : foreach x in es[] do
5 :   if x not signed then
6 :     if x in K then
7 :       eCopied[i] ← x
8 :       i ← i + 1
9 :      else if IsTargetedTo(x,m) then
10:        m ← Remove(e,m) /* m doit être passé en référence */
        endif
      endif
    endforeach
11: return eCopied
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fonction FindCopied : Récupération de la reproduction de l'élément enveloppé

Fonction FindSameNameElts: cette fonction retourne tous les éléments du message ayant le même nom de balise que l'élément passé en paramètre, mais avec un contenu différent de ce dernier. Ces éléments doivent être destinés au même nœud SOAP que l'élément passé en paramètre.

Fonction IsUseInOtherProc: cette fonction détermine si l'élément passé en paramètre est utilisé dans le traitement d'un autre élément du nœud SOAP courant. Cette vérification est importante afin de distinguer un élément légitime ayant l'attribut `role` à `none`, d'un élément fictif ayant l'attribut `role` à `none`. Les informations nécessaires à cette opération se trouvent dans la base des connaissances.

Fonction Restore: cette fonction représente notre algorithme de restauration qui sera abordé dans la section consacrée à ce sujet.

5.3.2. Annulation du traitement d'un élément optionnel

Cet algorithme diffère du précédent algorithme de détection par l'absence de la reproduction d'élément, et du caractère optionnel de l'élément enveloppé.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: message SOAP original
Sortie: message SOAP résultant du traitement de l'algorithme
CheckForOptionalElementIgnored(m)
1 : w[] ← GetSignedElts(m)
3 : foreach e in w[] do
4 :   a ← 0 /* 0 = absence d'APE, 1 = APE-reproduction, 2 = APE-annulation */
5 :   n ← null
6 :   t ← GetCurrentParent(e,m)
7 :   while t != header and t != body do
8 :     if (t not in K and IsOptProc(t)) or !IsTargetedTo(t,m) then
9 :       n ← t
       endif
```

```
10:     t ← GetCurrentParent(t,m)
      endwhile
11:   if n != null and n not signed then
12:      eSatus ← GetStatus(e,K)
13:      if eSatus = 0 then
14:         a ← 2
15:         t ← Restore(a,n,e,m,null) /* ici la reproduction est null*/
16:         if r != null then /* message restauré */
17:           m ← r
18:         else
19:           m ← error
20:           break
            endif
          endif
      endif
    endforeach
21: return m
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Détection des APE consistant à ignorer le traitement d'un élément optionnel

Voyons à présent les fonctions utilisées dans cet algorithme.

Fonction GetStatus: cette fonction retourne le statut de l'élément passé en paramètre. Le statut d'un élément est soit 1 (obligatoire) ou 0 (optionnel). Cette information se trouve dans la base des connaissances.

Les autres fonctions ont déjà été présentées dans les sections précédentes.

Fonction FindPotentialParent: cette fonction retourne le seul parent possible de l'élément passé en paramètre. La fonction récupère d'abord tous les emplacements possibles de l'élément à partir de la base des connaissances. Ensuite la fonction calcule combien parmi ces emplacements récupérés existe dans le message. L'élément parent courant de l'élément en question n'est pas pris en compte dans ce calcul. Si ce nombre est égal à 1, alors la fonction retourne cet emplacement, sinon la fonction retourne la valeur `null`.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: e,n,m /* n est le parent actuel de e */
Sortie: pt
FindPotentialParent(e,n,m)
1 : pt ← null
2 : ps[] ← GetPotentialParents(e,K)
3 :   i ← 0
4 :   foreach x in ps[] do
6 :     if n != x then
7 :       pt ← x
8 :       i ← i + 1
9 :     endif
      endforeach
10:   if pt != null and i > 1 then
11:     pt ← null
      endif
    endif
12: return pt
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Fonction FindPotentialParent : Identification de l'élément parent potentiel

Fonction GetPotentialParents: cette fonction retourne, à partir de la base des connaissances, tous les éléments parent potentiels de l'élément passé en paramètre.

5.3.4. L'algorithme de détection des APE

À présent, nous présentons l'algorithme général pour la détection des APE. Cet algorithme est issu des précédents algorithmes de détection des APE. Cet algorithme prendra en compte le cas où un élément enveloppé obligatoire est identifié, mais n'a pas été reproduit sous un parent légal.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée: message SOAP original
Sortie: message SOAP résultant du traitement de l'algorithme
CheckForElementWrappingAttack(m)
1 : w[] ← GetSignedElts(m)
2 : foreach e in w[] do
3 :   a ← 0 /* 0 = absence d'APE, 1 = APE-reproduction, 2 = APE-annulation, 3 = APE –
          enveloppement d'un élément obligatoire sans le reproduire */
4 :   n ← null
5 :   t ← GetCurrentParent(e,m)
6 :   while t != header and t != body do
7 :     if (t not in K and IsOptProc(t)) or !IsTargetedTo(t,m) then
8 :       n ← t
       endif
9 :     t ← GetCurrentParent(t,m)
     endwhile
10:  if n != null and n not signed then
11:    eSatus ← GetStatus(e,K)
12:    eCopied ← FindCopied(e,m)
13:    if eCopied != null then
14:      a ← 1
15:    else if eSatus = 0 then
16:      a ← 2
17:    else if eSatus = 1 then
       /* Il s'agit d'un élément enveloppé obligatoire, mais non reproduit. Alors on
       restaure le message comme s'il s'agissait d'une APE consistant à annuler le
       traitement d'un élément optionnel */
18:      a ← 3
       endif
19:    if a != 0 then
20:      r ← Restore(a,n,e,m,eCopied)
21:      if r != null then
22:        m ← r
23:      else
24:        m ← error
25:        break
         endif
       endif
     endif
    endforeach
26: return m
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Algorithme 1 : Détection des APE

5.3.5. L'algorithme de restauration

L'algorithme de restauration est un sous algorithme de celui de la détection. La restauration est entamée suite à la détection d'une catégorie d'APE. Par conséquent, la catégorie d'APE, l'élément enveloppant, l'élément enveloppé, le message SOAP et éventuellement la reproduction d'élément sont communiqués à l'algorithme de restauration.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Entrée : a,n,e,m,eCopied
Sortie : message restauré ou valeur nulle
Restore(a,n,e,m,eCopied)
1 : r ← null
2 : ps[] ← null
3 : eClass ← GetClass(e,K)
4 : if eClass = 1 or eClass = 2 then
5 :   p ← GetParentElt(e,K)
6 :   ps[] ← FindSameNameElts(p,m)
7 : else
8 :   ps[] ← FindPotentialParent(e,m)
    endif
9 : if count(ps[]) = 1 and ps[0] != null then
10:   m ← Relocate(e,ps[0],m)
11:   if !IsMultipleSignedEltsWrapping(e,n)
12:     m ← Remove(n,m)
      endif
13:   if a = 1 then /* Si c'est la reproduction d'élément, alors supprimer la reproduction */
14:     m ← Remove(eCopied,m)
      endif
15:   r ← m
    endif
16: return r
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Algorithme 2 : Restauration d'un message SOAP suite à la détection d'une APE

Nous initialisons à `null` les variables `r et p`, représentant respectivement le résultat de la restauration et le parent original de `e`. La variable `eClass` est initialisée à la classe de `e`. Lorsque `eClass` vaut 1 ou 2, c'est-à-dire `e` a un parent fixe dans la grammaire, alors nous récupérons son parent puis nous stockons dans `ps[]`la liste d'éléments ayant le même nom que le parent de `e`. Lorsque `eClass` vaut 3 ou 4, c'est-à-dire `e` peut avoir différents éléments parents dans la grammaire, nous stockons dans `ps[]`son seul parent possible, s'il existe un. Lorsque `ps[]` contient exactement un élément et celui est différent de `null`, alors le parent de `e` a été déterminé avec succès. Dans ce cas, `e` est déplacé sous son parent, puis `p` est supprimé s'il ne contient pas un autre élément signé. Lorsque l'opération de relocalisation de `e` a eu lieu avec succès, alors on supprime également `eCopied` s'il s'agit d'une reproduction d'élément du message. Le résultat de la restauration est retourné à travers la variable `r`.

Discutons à présent sur les fonctions importantes de notre l'algorithme de restauration.

Fonction GetClass: cette fonction retourne la classe de l'élément passé en paramètre. Cette information se trouve dans la base des connaissances. À ce propos, rappelons que la base des connaissances associe à chaque élément une classe particulière parmi quatre. Chaque classe a une signification particulière quant aux possibilités d'emplacement d'un élément, voir section 4.4.5.2. Pour un élément de classe 1 ou 2, il est possible d'identifier son emplacement (dans le message SOAP) simplement en consultant la base des connaissances (fonction GetParentElt). Pour un élément de classe 3 ou 4, il est nécessaire d'évaluer le contenu du message SOAP afin de déterminer les possibilités de son emplacement. Si cette possibilité d'emplacement est plus grande que 1, il n'est pas nécessaire d'en trouver davantage, car la restauration ne sera pas possible.

Fonction GetParentElt: cette fonction permet de retourner l'emplacement (l'élément parent) d'un élément de classe 1 ou 2, directement à partir de la base des connaissances.

Fonction Relocate: cette fonction permet de remettre un élément dans l'arborescence du message, à un emplacement spécifié après avoir récupéré la position exacte de l'élément (voir attribut `depth` de l'élément `path` dans la section 5.4.4) s'il s'agit d'un élément à position fixe. La fonction retourne le message résultant suite à cette opération.

Fonction Remove: cette fonction enlève du message, l'arborescence complète d'un élément ou la liste d'élément passée paramètre. La fonction retourne le message résultant suite à cette opération.

Fonction IsMultipleSignedEltsWrapping: cette fonction vérifie si l'élément enveloppant contient un autre élément signé différent de celui en cours de traitement. Cela permet de s'assurer que l'élément enveloppant est supprimé seulement lors du traitement du dernier élément signé qu'il enveloppe.

Les autres fonctions importantes ont été déjà présentées dans les sections précédentes.

5.4. EXEMPLE DE MISE EN ŒUVRE

Pour illustrer la mise en œuvre de notre modèle, nous prenons comme exemple un service de gestion de stock.

5.4.1. Description de notre service

Notre service de gestion de stock offre les fonctionnalités suivantes:

a) consultation du stock d'un produit (getStock)
b) mise à jour du stock d'un produit à une date spécifique (addToStock)

Ces fonctionnalités sont appelées des opérations. Pour le besoin de notre exemple, nous considérons comme optionnelle la date pour l'opération addToStock. Ainsi si une date est spécifiée, l'opération prendra effet à celle-ci, sinon l'opération prend effet à la date courante.

5.4.2. Messages SOAP (requêtes) traités par notre service

Dans ce point, nous présentons nos messages SOAP contenant uniquement le traitement applicatif. Plus loin, nous appliquons une signature sur les informations importantes qu'ils contiennent. Pour des raisons de simplicité, nous limitons aux requêtes SOAP. Les réponses SOAP n'ont pas grand intérêt pour la compréhension de notre modèle.

Opération addToStock

```
<soap:Envelope xmlns:soap="...">
  <soap:Header>
    <opDate>2005-05-29T09:00:00Z</opDate>
  </soap:Header>
  <soap:Body>
    <addToStockRequest>
       <account>7545</account>
       <amount>250</amount>
    </addToStockRequest>
  </soap:Body>
</soap:Envelope>
```

Opération getStock

```
<soap:Envelope xmlns:soap="...">
  <soap:Body>
    <getStockRequest>
       <account>7545</account>
    </getStockRequest>
  </soap:Body>
</soap:Envelope>
```

5.4.3. Génération de la signature sur nos messages

Afin d'assurer l'intégrité de nos messages, les éléments suivants seront signés:

Opération	**Élément(s) signé(s)**
getStock	getStockRequest
addToStock	addToStockRequest, opDate

Requête SOAP getStock

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="..."
xmlns:wsu="...">
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..." />
          <ds:SignatureMethod Algorithm="... " />
          <ds:Reference URI="#thisProduct">
          ...
          </ds:Reference>
        </ds:SignedInfo >
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
        ...
        </ds:KeyInfo >
      </ds:Signature>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <getStockRequest id="thisProduct ">
      <noProduct>7545</noProduct>
    </getStockRequest>
  </soap:Body>
</soap:Envelope>
```

Requête SOAP addToStock

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..." />
          <ds:SignatureMethod Algorithm="... " />
          <ds:Reference URI="#toStock">
          ...
          </ds:Reference>
          <ds:Reference URI="#date">
          ...
          </ds:Reference>
        </ds:SignedInfo >
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
        ...
        </ds:KeyInfo >
      </ds:Signature>
    </wsse:Security>
    <opDate id="date">2005-05-29T09:00:00Z</opDate>
  </soap:Header>
```

```
  <soap:Body>
    <addToStockRequest id="toStock">
      <noProduct>7545</noProduct>
      <amount>15</amount>
    </addToStockRequest>
  </soap:Body>
</soap:Envelope>
```

5.4.4. Établissement de notre base des connaissances

Tel que mentionné précédemment, la base des connaissances contient des informations sur deux types d'éléments: les éléments communs et les éléments spécifiques. Les éléments communs qui sont utilisés dans nos messages sont les éléments de SOAP, de WS-Security et de XML-Signature, soient: `Envelope`, `Header`, `Body`, `Security`, `BinarySecurityToken`, `Signature`, `SignedInfo`, `CanonicalizationMethod`, `SignatureMethod`, `Reference`, `DigestMethod`, `DigestValue`, `SignatureValue`, `KeyInfo`, `SecurityTokenReference`. Les éléments spécifiques sont: `getStockRequest`, `addToStockRequest`, `noProduct`, `amount` et `opDate`. Nous rappelons que les informations à collecter sur les éléments sont leur nom de balise, leur caractère obligatoire ou optionnel, leur classe et leur emplacement exact.

À présent, considérons notre base des connaissances qui se trouve sur la figure 14. À première vue, la structure de la base des connaissances ressemble à celle d'un message SOAP. Nous avons un élément `Envelope` contenant les éléments `Header` et `Body`, qui à leur tour contiennent les éléments qu'ils peuvent contenir dans un message SOAP. Un autre élément nommé `procElts` regroupe les éléments destinés au traitement d'autres éléments. Ces éléments se caractérisent par leur attribut `role` à `none`. La spécification SOAP a mentionné la présence possible de tels éléments sans illustrer leur utilisation par un exemple concret. Pour rester dans la généralité, nous avons fait cette distinction afin d'identifier un élément légitime ayant l'attribut `role` à `none`, d'un élément fictif ayant l'attribut `role` à `none`.

Chaque élément de notre base peut avoir les attributs `@statut`, `@class` et `@deph` qui renseignent respectivement le statut, la classe de l'élément et la position de l'élément au sein son élément parent dans le cas où l'élément a une position fixe. Typiquement, la valeur de `@deph` est supérieure à 0. Cependant, certains éléments doivent être le dernier sous-élément de leur parent. Par exemple, la spécification SOAP stipule implicitement que `Body` soit le dernier sous-élément de son parent `Envelope`. `Header` étant optionnel, alors `Body` peut légitimement être le 1er sous-élément de `Envelope` (si `Header` est absent), ou être le 2ème sous-élément (si `Header` est présent). C'est pour éviter l'ambigüité que nous avons introduit pour `@depth` la valeur spéciale -1 qui signifie un emplacement correspondant au dernier sous-élément de son élément parent.

Cette structure offre un bon avantage en termes de simplicité de traitement, et améliore sa lisibilité. En jetant simplement un coup d'œil sur la structure, on peut voir quel élément est le parent de quel autre. Par exemple, l'élément `noProduct` est à la fois un sous-élément des éléments `getStockRequest` et `addToStockRequest`, qui à leur tour sont des sous-éléments de l'élément `Body`.

```
<knowledge>
  <Envelope status="1" class="1">
    <Header status="0" class="1">
      <Security status="1" class="2">
```

```
        <BinarySecurityToken status="1" class="2"/>
        <Signature status="1" class="2">
          <SignedInfo status="1" class="1" depth="1">
            <CanonicalizationMethod status="1" class="2"/>
            <SignatureMethod status="1" class="2"/>
            <Reference status="1" class="2">
              <DigestMethod status="1" class="2"/>
              <DigestValue status="1" class="2"/>
            </Reference>
          </SignedInfo>
          <SignatureValue status="1" class="2"/>
          <KeyInfo status="1" class="2">
             <SecurityTokenReference status="1" class="2">
               <Reference status="1" class="2"/>
             </SecurityTokenReference>
          </KeyInfo>
         </Signature>
       </Security>
      <opDate status="0" class="1" depth="1"/>
    </Header>
    <Body status="1" class="1" depth="-1">
     <getStockRequest status="1" class="2">
       <noProduct status="1" class="4"/>
     </getStockRequest>
     <addToStockRequest status="1" class="2">
        <noProduct status="1" class="4"/>
        <amount status="1" class="4"/>
      </addToStockRequest>
    </Body>
  </Envelope>
  <procElts/><!— éléments utilisés dans le traitement d'autres éléments -->
</knowledge>
```

Figure 24 : La base des connaissances

5.4.5. Simulation des APE sur nos messages

Maintenant que la base des connaissances est établie, simulons des APE sur différents éléments de nos messages SOAP.

5.4.5.1 Reproduction d'élément avec nouveau contenu

Une APE consistant à la reproduction d'élément survient sur un élément dont le pirate veut assigner un nouveau traitement. Par conséquent, le pirate peut raisonnablement être intéressé à chacune des opérations offertes par notre service. Par exemple, le pirate peut vouloir intercepter une opération de mise à jour de stock, et modifier soit le numéro de produit ou la quantité de stock spécifiée. Cependant, pour rester simple, nous simulons une reproduction d'élément uniquement sur l'opération d'interrogation de stock (getStock). Le message altéré résultant d'une telle attaque est illustré dans la figure suivante:

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..." />
          <ds:SignatureMethod Algorithm="... " />
          <ds:Reference URI="#thisProduct">
          ...
          </ds:Reference>
        </ds:SignedInfo >
```

```
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
        ...
        </ds:KeyInfo >
      </ds:Signature>
    </wsse:Security>
    <WrapperElement>
      <getStockRequest id="thisProduct">
        <noProduct>7545</noProduct>
      </getStockRequest>
    </WrapperElement>
  </soap:Header>
  <soap:Body>
    <getStockRequest id="otherProduct">
      <noProduct>1545</noProduct>
    </getStockRequest>
  </soap:Body>
</soap:Envelope>
```

Figure 25 : Requête getStock altéré par une APE consistant à la reproduction d'élément

5.4.5.2 Annulation du traitement d'un élément optionnel

Seul l'élément `opDate` de l'opération addToStock peut faire l'objet d'une APE consistant à annuler son traitement, il s'agit de l'unique élément optionnel de nos messages SOAP.

```
<soap:Envelope xmlns:soap="..." xmlns:wsse="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="..." />
          <ds:SignatureMethod Algorithm="... " />
          <ds:Reference URI="#toStock">
          ...
          </ds:Reference>
          <ds:Reference URI="#date">
          ...
          </ds:Reference>
        </ds:SignedInfo >
        <ds:SignatureValue>...</ds:SignatureValue>
        <ds:KeyInfo>
        ...
        </ds:KeyInfo >
      </ds:Signature>
    </wsse:Security>
    <WrapperElement soap:role=".../none">
      <opDate id="date">2005-05-29T09:00:00Z</opDate>
    </WrapperElement>
  </soap:Header>
  <soap:Body>
    <addToStockRequest id="toStock">
      <noProduct>7545</noProduct>
      <amount>15</amount>
    </addToStockRequest>
  </soap:Body>
</soap:Envelope>
```

Figure 26 : Requête addToStock altéré par une APE consistant à annuler un traitement optionnel

5.4.6. Exécution des algorithmes

Dans cette section, nous montrons la trace d'exécution de l'algorithme de détection, et l'algorithme de restauration qui est un sous algorithme du précédent. Pour ce faire, nous prenons l'un après l'autre les messages altérés des figures 25 et 26. Voyons en premier le message de la figure 25.

5.4.6.1 Reproduction d'élément

À la ligne 1, la variable `w[]` stocke l'élément `getStockRequest` (id="thisProduct"), il s'agit du seul élément signé du message. De la ligne 5 à 9, l'élément WrappindElement est identifié comme étant le parent inconnu de `getStockRequest` (id="thisProduct"). Un élément ayant le même nom de balise que `getStockRequest` (id="thisProduct"), mais avec un nouveau contenu a été identifié, il s'agit de l'élément `getStockRequest` (id="otherProduct"). Dans ce cas, il s'agit d'une APE consistant à la reproduction d'élément avec un nouveau contenu. Les éléments `WrappindElement` et `getStockRequest` (id="thisProduct"), le type d'APE (1), le message SOAP ainsi que la reproduction `getStockRequest` (id="otherProduct") sont passés à l'algorithme de restauration. Ce dernier remet `getStockRequest` (id="thisProduct") sous l'élément `Body` qui a été préalablement identifié comme étant son seul emplacement possible. Il faut noter que `getStockRequest` est remis exactement à la dernière position de `Body`, car il s'agit d'un un élément à position non fixe (classe 2). Ensuite, `WrappindElement` est supprimé du message, car il ne contient pas un autre élément signé. Puisqu'il s'agit d'une reproduction d'élément, alors `getStockRequest` (id="otherProduct") est également supprimé du message. Le message résultant est retourné à l'algorithme de détection. Ce résultat n'étant pas `null`, il donc est affecté à l'ancien message. Tous les éléments signés ont été traités, le message restauré est retourné et l'algorithme se termine.

5.4.6.2 Annulation du traitement d'un élément optionnel

Maintenant voyons la trace de notre algorithme de détection des APE sur notre deuxième message (figure 26).

À la ligne 1, nous identifions les éléments `addToStockRequest` et `opDate` comme étant les éléments signés du message. L'élément `addToStockRequest` est le premier à être traité. L'identification de son parent inconnu échoue puisque son parent est l'élément légitime `Body`. Nous passons au prochain élément signé `opDate` dont le parent inconnu est identifié comme étant `WrappindElement`. L'élément `opDate` est optionnel (son statut est 0) et n'a pas été reproduit ailleurs. Dans ce cas, il s'agit d'une APE consistant à l'annulation du traitement d'un élément optionnel. Les éléments `WrappindElement` et `opDate`, le type d'APE (2) ainsi que le message sont passés à l'algorithme de restauration. L'élément `Header` est identifié comme étant le seul parent possible de `opDate`. L'élément `opDate` est remis à cet emplacement à la dernière position, car `opDate` est un élément à position non fixe (classe 2). Par la suite, `WrappindElement` est supprimé du message, car il ne contient pas un autre élément signé. Le message résultant est retourné à l'algorithme de détection. Ce résultat n'étant pas `null`, il est donc affecté à l'ancien message. Tous les éléments signés ont été traités, le message restauré est retourné et l'algorithme se termine.

CHAPITRE 6: IMPLÉMENTATION ET ANALYSE DES RÉSULTATS

Dans ce chapitre, nous discutons de l'implémentation de nos algorithmes et leur performance. Nous analysons les données que nous avons recueillies lors des tests, et nous discutons de la complexité de nos algorithmes. Pour terminer, nous présentons un tableau comparatif de notre technique avec d'autres.

Notre technique nécessite la manipalution de données XML. Parmi les technologies contribuant au traitement XML, XPath est le plus populaire. La plupart des autres technologies, notamment XQuery, XPointer et XSLT, se basent sur elle pour offrir encore plus de possibilités. La syntaxe de XPath est relativement simple. Une expression (ou requête) XPath est typement composée :

- d'un axe de localisation (axe de parenté, axe de voisinage, axe des attributs, etc);
- un motif (nœud XML contenant tel caractère, nœud XML à telle position, nœud XML possédant ou non tel attribut, etc).

La complexité des requêtes XPath est une donnée importante pour notre technique. Cette complexité a été un sujet assez bien étudié dans la littérature. Dans le pire cas, la complexité des requêtes XPath est connue d'être polynomiale pour la version 1.0, et exponentielle pour la version 2.0 [27, 28]. Dans certains cas, la complexité est linéaire pour certaines types de requêtes XPath 1.0.

Puisque nos algorithmes utilisent uniquement des requêtes XPath 1.0, alors une première impression nous conduit à une estimation de sa compléxité dans l'ordre polynomial. Cependant, nous avons procédé à une série de test pour observer l'évolution du temps de traitement d'un message SOAP en fonction du nombre d'éléments signés dans le message SOAP, et de la taille de la base des connaissances.

6.1. IMPLÉMENTATION

Afin d'illustrer la faisabilité de notre technique, nous avons développé un prototype en utilisant les langages XHTML, PHP et XML. Le XHTML est utilisé pour l'affichage. Les nœuds XML sont manipulés en tant qu'objet SimpleXML de PHP. PHP fait le gros du traitement. Les messages SOAP ainsi que la base des connaissances sont dans un format de fichier XML.

L'application propose une liste de choix composés de plusieurs messages altérés, un message pour chacune des catégories d'APE, et d'autres messages pour couvrir différentes branches des algorithmes. Ces derniers messages illustrent :

- le cas de plusieurs catégories d'APE sur le même message;
- le cas où la restauration n'est pas possible du à la présence de plusieurs emplacements possibles pour l'élément enveloppé;
- le cas où il y a plusieurs éléments enveloppants;
- le cas où l'élément enveloppé est inconnu;

- le cas où il n'y a pas d'élément signé dans le message;
- le cas où l'élément enveloppé est obligatoire, mais n'a pas été reproduit ailleurs;
- le cas où l'élément référencé dans la signature est absent du message;
- le cas où l'élément enveloppant enveloppe plusieurs éléments signés;
- le cas où l'élément enveloppé est reproduit plusieurs fois;
- diverses autres situations où le contenu du message SOAP, soit n'est pas conforme aux exigences de notre service, soit n'est pas une donnée XML valide.

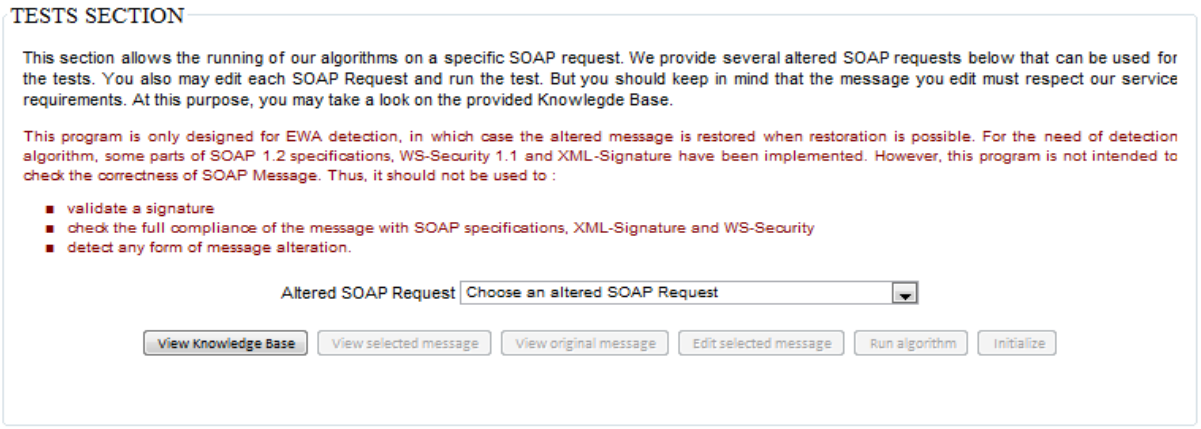


Figure 27: Application de détection des APE, et de restauration du message SOAP altéré

En tout temps, l'utilisateur peut consulter le contenu de la base des connaissances. À chaque fois qu'un message est sélectionné, l'application donne la possibilité de:

- visualiser le contenu du message;
- visualiser le contenu du message original (message avant l'attaque);
- éditer le message pour tenter une attaque plus spécifique;
- lancer l'algorithme de détection (sur le message sélectionné).

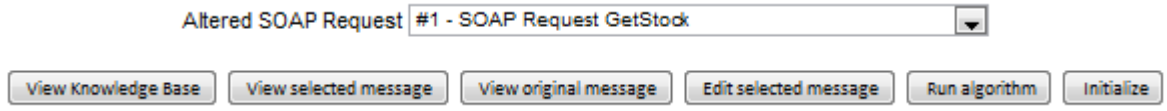


Figure 28: Fonctions disponibles lors de la sélection d'un message

Lorsque l'utilisateur sélectionne un message et clique sur le bouton **Run algorithm**, le message est passé à l'algorithme de détection pour vérification. La figure 29 montre le résultat de l'exécution de l'algorithme sur la requête SOAP altéré addToStock.

Ce résultat montre l'identification de deux éléments signés nommés respectivement `addToStockRequest` et `opDate`. L'élément signé `addToStockRequest` est le premier à être traité. Le résultat montre que cet élément n'a pas subi d'APE puisqu'il ne possède pas d'élément parent inconnu. Par contre, l'algorithme détecte un parent inconnu (`WrapperElement`) pour le second élément signé `opDate`. Le résultat montre qu'il s'agit d'une APE consistant à l'annulation du traitement d'un élément optionnel.

Lorsqu'une attaque est détectée sur un message SOAP, ce dernier est passé à l'algorithme de détection. La figure 31 montre le résultat de l'algorithme de restauration sur la requête SOAP `addToStock`. Ce résultat montre que le message a été restauré avec succès, et son contenu est également affiché. L'application offre également la possibilité de visualiser le contenu du message restauré dans un navigateur web. Ainsi le contenu du message restauré peut aisément

être comparé à celui du message original. En bas du résultat de restauration, est affiché le temps d'exécution total: étapes de détection et de restauration.

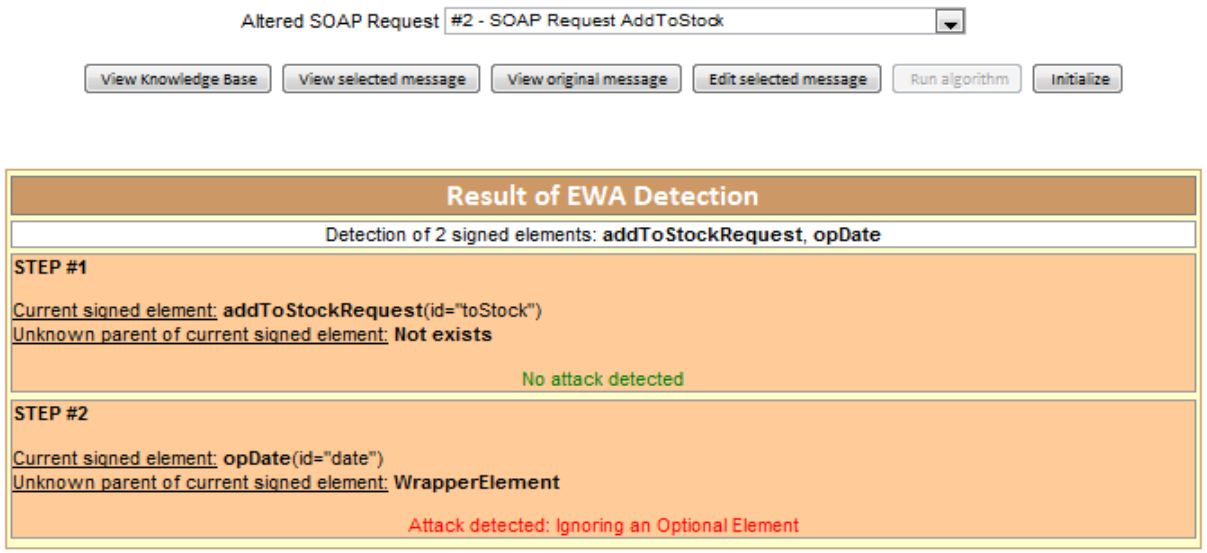


Figure 29: Résultat de l'exécution de l'algorithme sur la requête addToStock

La figure 30 montre le résultat de l'algorithme sur une requête formée des requêtes getStock et addToStock. Dans cette requête, nous avons expressément introduit les éléments `getStockRequest` et `addToStockRequest` sous deux éléments `Body` distincts (pour le besoin de l'exemple). Le résultat montre que l'APE sur `getStockRequest` est détectée, mais le nombre d'emplacements possibles pour cet élément est multiple. Ainsi, il n'est pas nécessaire de traiter les autres éléments signés restants, car la restauration est impossible; le message est simplement rejeté.

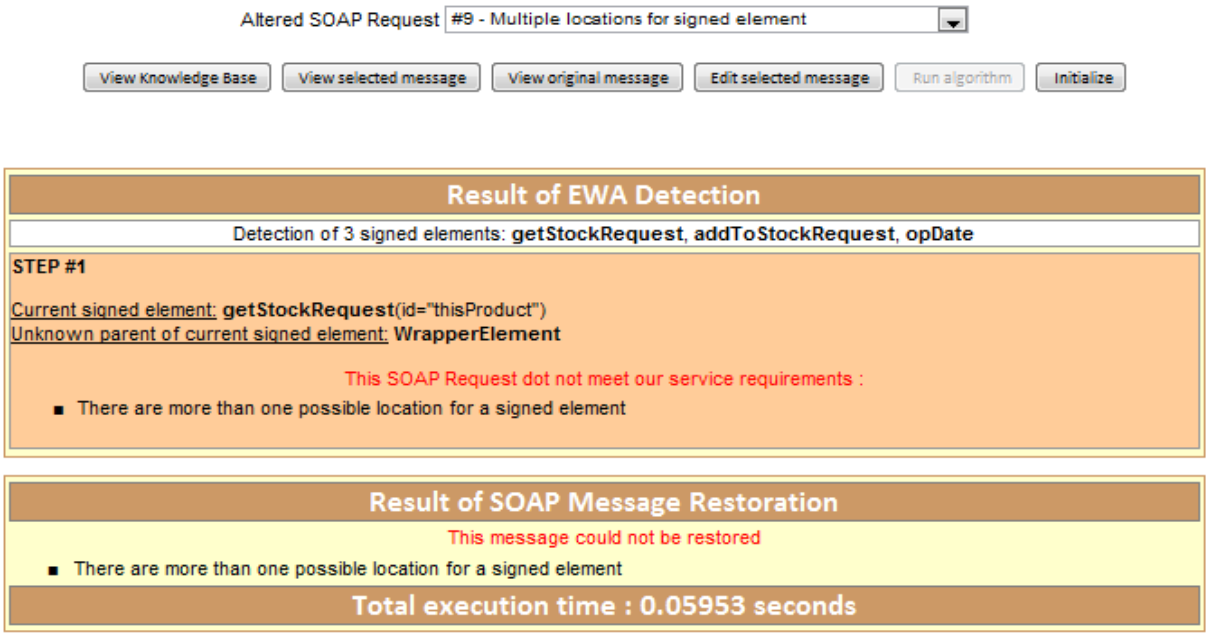


Figure 30 : Résultat dans le cas où le nombre d'emplacement est multiple

Figure 31 : Résultat de la restauration de la requête SOAP addToStock altéré

6.2. ANALYSE DE NOTRE TECHNIQUE

À présent, nous étudions la complexité de notre algorithme de détection qui contient l'algorithme de restauration. La complexité algorithmique permet de voir l'évolution du temps de traitement requis par un algorithme pour accomplir une tâche spécifique en fonction de la taille des données d'entrée. Dans notre cas, les données d'entrée sont le message SOAP et la base des connaissances. Notre algorithme comprend une boucle principale dans laquelle plusieurs autres fonctions sont invoquées. On peut évaluer la complexité d'un algorithme en analysant ses boucles et ses fonctions (ou sous-algorithmes) dans le pire cas. Dans notre cas, cela est très difficile dans la mesure où plusieurs facteurs rentrent en jeu tels que la taille de la base des connaissances. Une autre approche d'analyse de la complexité algorithmique est faite par expérimentation en calculant le temps moyen d'exécution à partir d'échantillons de données. Ainsi avons-nous procédé, et ce de deux façons :

1) un premier test visant à montrer l'évolution du temps moyen de traitement d'un message SOAP en fonction du nombre d'éléments signés qu'il contient, et ce dans les contextes suivants :
    a. le message contient à la fois des éléments signés altérés et des éléments signés non altérés. Les éléments signés sont générés dynamiquement, et la probabilité qu'un élément signé soit altéré ou non est la même;
    b. le message contient uniquement des éléments signés non altérés;
    c. et enfin, le message contient uniquement des éléments signés altérés.
2) un test visant à montrer l'évolution du temps moyen de traitement d'un message SOAP en fonction du nombre d'éléments contenus dans la base des connaissances.

Les tests ont été réalisés sur la même machine dans des conditions similaires, en occurrence un PC 64 bits avec processeur à triple cœur avec 4 Giga-octets de mémoire, tournant sous Windows 7.

Remarques importantes

- *Parce que les résultats obtenus lors des tests proviennent d'une configuration précise, les données recueillies varieront d'une machine à l'autre. En revanche, les données suivront sensiblement la même évolution en fonction de la taille des données d'entrée, et ce indépendamment de la machine sur laquelle le test est effectué.*
- *Seule l'APE consistant à la reproduction d'élément a été retenue lors des tests, car en effet cette attaque est la plus complexe, et requiert davantage de délai pour sa correction.*

6.2.1. Description des données lors des tests

Il est important d'avoir une idée de la structure d'un élément signé typique utilisé dans les messages SOAP lors des tests. Les éléments signés avaient la même structure que l'un des éléments `getStockRequest` ou `addToStockRequest` présentés dans le chapitre 5. Il s'agit d'élément contenant un ou deux sous-éléments. Le choix entre l'un ou l'autre est fait de façon aléatoire avec la même probabilité. Les éléments signés d'un message sont tous distincts. Leur nom et celui de chacun de leurs sous-éléments se terminent avec un chiffre correspondant au n-ième élément signé du message. Par exemple :

Pour le premier élément signé, nous avons :

```
<getStockRequest1 id="opt1">
  <noProduct1>7545</noProduct1>
</getStockRequest1>
```

ou

```
<addToStockRequest1 id="opt1">
  <noProduct1>7545</noProduct1>
  <amount1>15</amount1>
</addToStockRequest1>
```

Pour le dixième élément signé, nous avons:

```
<getStockRequest10 id="opt10">
  <noProduct10>7545</noProduct10>
</getStockRequest10>
```

ou

```
<addToStockRequest10 id="opt10">
  <noProduct10>7545</noProduct10>
  <amount10>15</amount10>
</addToStockRequest10>
```

Un élément est représenté dans la base des connaissances sous un format similaire.

```
      …
<getStockRequest status="1" class="2">
  <noProduct status="1" class="4"/>
</getStockRequest>
<addToStockRequest status="1" class="2">
  <noProduct status="1" class="4"/>
  <amount status="1" class="4"/>
</addToStockRequest>
      …
<getStockRequest20 status="1" class="2">
  <noProduct20 status="1" class="4"/>
</getStockRequest20>
<addToStockRequest20 status="1" class="2">
  <noProduct20 status="1" class="4"/>
  <amount20 status="1" class="4"/>
</addToStockRequest20>
```

Maintenant que nous avons une idée des données utilisées lors des tests, nous allons présenter les résultats obtenus dans les prochaines sections.

6.2.2. Temps moyen de traitement d'un message SOAP en fonction du nombre d'éléments signés

6.2.2.1 Tableau des données recueillies lors de ce test

Le tableau suivant résume le temps de traitement des messages M1, M2, M3 (voir légende) contenant un nombre d'éléments signés (ou opérations) allant de 100 à 1000, avec un incrément de 100.

	100				200				300				400				500			
	T1	T2	T3	**T**	T1	T2	T3	**T**	T1	T2	T3	**T**	T1	T2	T3	**T**	T1	T2	T3	**T**
M1	2	2	2	**2**	8	9	7	**8**	20	19	15	**18**	28	37	31	**32**	60	44	56	**53**
M2	0.4	0.4	0.5	**0.4**	2	2	2	**2**	4	4	5	**4**	6	6	7	**6**	11	10	10	**10**
M3	4	3	3	**3**	13	13	16	**14**	29	37	36	**34**	53	67	53	**58**	84	89	88	**87**

	600			700			800			900			1000		
	T1	T2	**T**	T1	T2	**T2**	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**
M1	66	91	**77**	98	95	**97**	133	128	**131**	172	179	**176**	218	217	**218**
M2	15	19	**17**	21	21	**21**	35	28	**32**	37	36	**37**	59	44	**52**
M3	119	120	**120**	181	172	**177**	218	215	**217**	318	316	**317**	389	464	**427**

Légende

M1 : le message contient à la fois des éléments signés altérés et des éléments signés intacts
M2 : le message contient uniquement des éléments signés intacts
M3 : le message contient uniquement des éléments signés altérés
T1 : série de test 1
T2 : série de test 2
T3 : série de test 3
T : moyenne des séries

Remarque

Afin de simuler une situation réelle, à chaque fois qu'un certain nombre d'éléments signés est généré dans le message SOAP, les mêmes éléments (et ses sous-éléments) sont également générés dans la base des connaissances. En effet, lorsque le message SOAP contient un élément qui n'existe pas dans la base des connaissances, il est rejeté et le traitement s'arrête tout de suite. Cette série de test montre donc l'évolution du temps de traitement en fonction du nombre d'éléments (opération), à la fois dans le message SOAP et dans la base des connaissances.

On peut observer que les résultats obtenus pour chacun des trois types de messages peuvent être approximés par une fonction polynomiale f(n) pour un entier n strictement positif :

- M1 : $f(n) = 2n^2 = 2, 8, 18, 32, 50, 72, 98, 128, 162, 200, \ldots$
- M2 : $f(n) = n^2 - 2n = -1, 0, 6, 8, 15, 20, 35, 48, 63, 80, \ldots$
- M3 : $f(n) = 4n^2 = 4, 16, 36, 64, 100, 144, 196, 256, 324, 400, \ldots$

6.2.2.2 Courbe illustrant l'évolution du temps moyen de traitement par rapport au nombre d'éléments signés

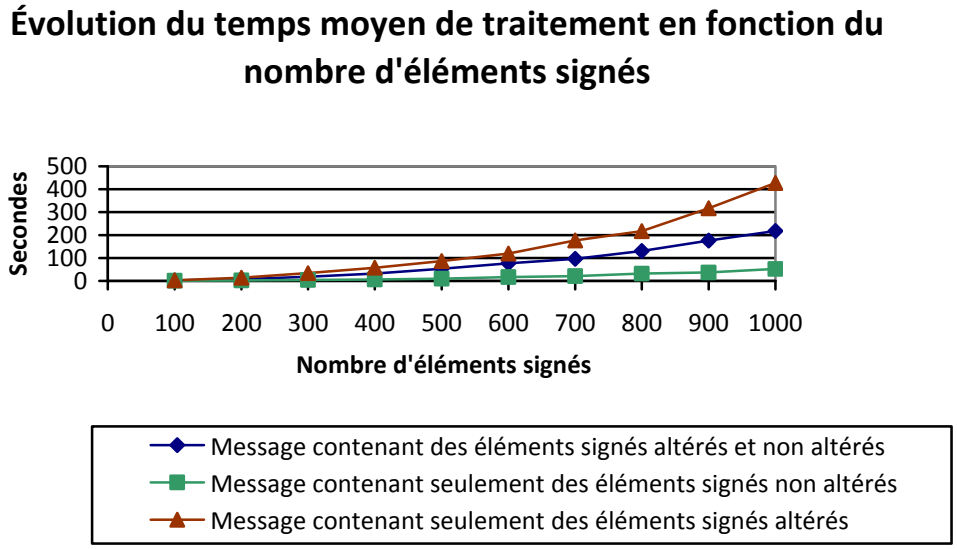


Figure 32 : Temps moyen d'exécution de notre algorithme en fonction du nombre d'éléments signés

6.2.3. Temps moyen de traitement d'un message SOAP en fonction de la taille de la base des connaissances

6.2.3.1 Tableau des données recueillies lors de ce test

Nous étudions à présent l'évolution du temps moyen d'exécution de notre algorithme par rapport à la taille de la base des connaissances. Plus exactement, nous voulons savoir pour un message SOAP ayant un certain nombre d'éléments signés, comment évolue le temps moyen de traitement lorsque la taille de la base des connaissances augmente successivement de 1000 à 10000 éléments (ou opérations), avec un incrément de 1000. Le tableau suivant montre les résultats obtenus.

	1000			2000			3000			4000			5000		
	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**
M1	0.8	0.8	**0.8**	5	1.8	**1.75**	3.58	3.66	**3.62**	5.93	5.97	**5.95**	9.12	9.06	**9.09**
M2	0.16	0.15	**0.16**	0.37	0.4	**0.39**	0.73	0.77	**0.75**	1.27	1.23	**1.25**	2.02	1.95	**1.98**
	6000			7000			8000			9000			10000		
	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**	T1	T2	**T**
M1	12.9	12.8	**12.8**	17.4	17.2	**17.3**	22.4	22.4	**22.4**	28.6	27.6	**28.1**	34.6	34.8	**34.7**
M2	2.81	2.82	**2.82**	3.74	3.65	**3.69**	4.88	4.78	**4.83**	5.99	6.07	**6.03**	7.46	7.56	**7.51**

Légende
M1 : le message contient 10 éléments signés dont 5 altérés
M2 : le message contient 10 éléments signés tous intacts
T1 : série de test 1
T2 : série de test 2
T : moyenne des séries

Remarque

Les éléments communs, en occurrence `Envelope`, `Header`, `Body`, *etc ne sont pas comptabilisés dans le nombre d'éléments dans la base des connaissances. Seuls les éléments applicatifs sont considérés. Cela signifie qu'en réalité le test a été effectué sur une base des connaissances qui contient une vingtaine d'éléments de plus que les chiffres mentionnés dans le tableau.*

6.2.3.2 Courbe illustrant l'évolution du temps moyen de traitement par rapport à la taille de la base des connaissances

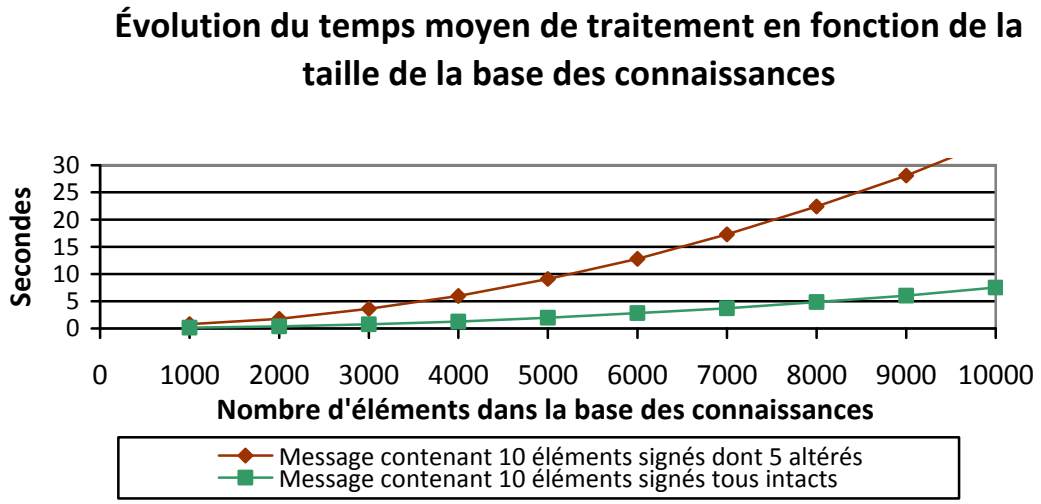


Figure 33 : Temps moyen d'exécution de notre algorithme en fonction de la taille de la base des connaissances

Ce graphique montre une influence de la taille de la base des connaissances sur le temps de traitement d'un message SOAP. Cette influence est très minime lorsque le message n'a pas subi d'altération. Ces données peuvent être approximées par une fonction polynomiale majorée par la fonction $f(n) = n^2$, où n est un entier strictement positif.

6.2.3.3 Discussion des résultats obtenus

Les résultats que nous avons obtenus montrent que notre algorithme, incluant les sous-algorithmes, fait son traitement en temps quadratique. D'une manière générale, nous situons sa complexité dans l'ordre polynomial : $O(n^P)$. En effet, cette complexité reflète le pire cas, c'est-à-dire lorsque tous ou quelques éléments signés du message SOAP sont altérés. En revanche, cette complexité est quasi-linéaire lorsque le message SOAP n'a subi aucune altération.

6.2.4. Comparaison avec d'autres techniques

Nous considérons les travaux [20,21] et [24] qui proposent, au delà de l'idée, une solution complète. À titre de rappel, le travail [20,21] propose d'inclure dans le message des informations sur sa structure lors de chaque signature. Le travail [24] propose d'inclure dans le message un contexte de signature. Le fonctionnement de ces techniques est expliqué dans la

section 4.2. Le tableau suivant montre le résultat de la comparaison de notre technique avec les deux autres.

Critère de comparaison	**Notre technique**	**[24]**	**[20,21]**
La technique nécessite une configuration du service web	Oui	Oui	Oui
Le message nécessite un traitement avant son envoi	Non	Oui	Oui
Le message nécessite un traitement à destination	Oui	Oui	Oui
Le message nécessite des informations de vérification (hors mis la signature)	Non	Oui	Oui
Le message nécessite des opérations cryptographiques supplémentaires	Non	Oui	Oui
Les informations de vérification du message augmentent avec le nombre de signature dans le message	Sans object	Oui	Oui
Les informations de vérification du message nécessitent leur mise à jour après une modification de la structure du message	Sans object	Oui	Oui
La complexité du traitement de vérification dépend de la taille des informations de vérification dans le message	Sans object	Oui	Oui
La complexité du traitement de vérification dépend du nombre de signatures dans le message	Oui	Oui	Oui
La taille et la structure du message influencent la complexité du processus de vérification	Parfois	Non	Non
La technique nécessite aux intermédiaires actifs[7] de connaitre la clé de signature de l'émetteur afin de mettre à jour les informations de vérification du message	Sans object	Oui	Oui
La retransmission du message est nécessaire après détection d'une APE	Très rarement (voir 4.4.7.2)	Oui	Oui
Détecte toutes les catégories d'APE	Oui	Oui	Non

Il est facile d'observer que notre technique offre plusieurs avantages par rapport aux autres techniques et ce, dans plusieurs contextes.

[7] Voir section 1.2.1.1 point d)

CONCLUSION

Les SW permettent l'interaction entre applications hétérogènes et réparties, et ce de façon souple et indépendante de leurs plates-formes et leurs langages de programmation. D'une part, les SW sont sollicités dans le domaine des affaires inter entreprises, un domaine dans lequel il existe un réel besoin d'intégration des processus métier. D'autre part, ils sont mis en œuvre par des compagnies dans le but de « subdiviser » de grosses et complexes applications en vue de mieux faire face aux évolutions futures. Quelque soit la raison pour laquelle les SW sont utilisés, le contexte d'échange de messages entre ces agents logiciels répartis nécessite une qualité de service satisfaisante, surtout au niveau de la sécurité.

Dans ce mémoire, nous abordons les services web (SW) tout en se focalisant sur l'aspect sécurité. Plus précisément, nous évoquons le problème des attaques par enveloppement(APE) qui reste toujours un sujet d'actualité. Nous abordons également la question de restauration d'un message SOAP suite à la détection d'une APE, un sujet qui semble jusqu'alors absente de la littérature. Nous présentons une solution à la fois pour la détection des APE, et la restauration du message altéré. Notre solution se base sur la recherche de motifs, et d'informations établies par le concepteur sur son service. Pour illustrer notre modèle, nous choisissons un contexte d'exécution de service où il n'y a pas d'intermédiaires. Cependant, notre solution ne fixe pas de contrainte sur le statut du nœud SOAP qui traite le message. Dans le contexte d'intermédiaires, il suffit d'établir une base des connaissances pour chaque intermédiaire.

Notre technique étant basée sur l'identification de motifs, elle ne nécessite pas de données ni d'opérations cryptographiques additionnelles au niveau du message SOAP. Notre technique n'a pas impact sur la bande passante. D'ailleurs, elle permet l'économie cette dernière grâce à la restauration qui prévient la retransmission du message SOAP altéré. Nous remarquons que l'utilisation de la plupart des autres technologies des SW (XML-Encryption, WS-Policy, WS-SecurityPolicy, etc) requiert l'ajout de données dans le message SOAP. En fait, cet aspect des SW constitue un désavantage surtout lorsque le service nécessite l'utilisation de plusieurs de ces technologies. En effet, la taille du message SOAP peut devenir très importante uniquement à cause de ces données de vérification. Or nous sommes dans une tendance où les applications consomment de plus en plus de données utiles. Dans cette optique, il est préférable de se pencher vers une solution de sécurité telle que la notre, plutôt que l'ajout de données supplémentaires, qui ne sera plus envisageable dans le message pour des raisons de performance et d'économie de la bande passante.

Afin d'étudier la faisabilité de notre technique, nous avons développé un prototype. Ensuite, nous avons procédé à une série de test pour analyser la performance de nos algorithmes par rapport au nombre d'éléments signés d'une part, et à la taille de la base des connaissances d'autre part. Les tests ont montré des résultats intéressants. En effet, le temps moyen de traitement d'un message nous permet de situer la complexité de nos algorithmes dans un temps polynomial. Nous pensons qu'il s'agit de la première technique pour la restauration d'un message SOAP altéré par une APE.

Nous suggérons quelques idées de travaux futurs qui peuvent être entrepris pour améliorer ce travail. Par exemple, à l'aide de Timestamp, on peut exiger pour chaque signature une date de création et d'expiration afin de détecter les messages périmés. Une autre amélioration envisageable est l'automatisation de la génération d'informations relatives aux éléments (communs ou spécifiques) de la base des connaissances à partir de fichier DTD, schéma XML

ou tout autre document de description de données disponible. Nous croyons qu'une telle entreprise est possible parce qu'un bon nombre d'informations requises dans la base des connaissances (nom de balise, statut, emplacement, etc) peuvent être obtenues à partir d'un tel document. Enfin notre modèle peut servir de point de départ pour l'étude de la restauration d'un message SOAP après toute forme d'altération, non pas seulement les éléments signés. Ce type d'attaque est encore plus général, et il est connu sous le nom d'attaque par réécriture (rewriting attacks), un autre problème d'actualité.

BIBLIOGRAPHIE

[1] Liste complète des spécifications relatives aux services web
http://fr.wikipedia.org/wiki/Liste_des_sp%C3%A9cifications_des_Services_Web_WS-%2A (consulté en hiver 2008)

[2] SOAP 1.2
http://www.w3.org/TR/soap12-part1 (consulté en automne 2008)

[3] WSDL 1.1
http://www.w3.org/TR/wsdl (consulté en automne 2008)

[4] UDDI 3.0.2
http://uddi.org/pubs/uddi_v3.htm (consulté en automne 2008)

[5] WS-Security
http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf (consulté en automne 2008)

[6] XML-Signature
http://www.w3.org/TR/xmldsig-core/ (consulté en automne 2008)

[7] Canonical XML Version 1.0
http://www.w3.org/TR/xml-c14n

[8] XML-Encryption
http://www.w3.org/TR/xmlenc-core/ (consulté en automne 2008)

[9] WS-SecureConversation
http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html (consulté en hiver 2008)

[10] WS-SecurityPolicy
http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html (site consulté en hiver 2008)

[11] WS-Policy
http://www.ibm.com/developerworks/library/specification/ws-polfram/ (consulté en hiver 2008)

[12] WS-Trust
http://www.ibm.com/developerworks/library/specification/ws-trust/ (consulté en hiver 2008)

[13] SAML http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf (site consulté en hiver 2008)

[14] XKMS http://www.w3.org/TR/xkms/ (site consulté en hiver 2008)

[15] XACML http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf (site consulté en hiver 2008)

[16] Journal sur la sécurité des services web
http://www.ibm.com/developerworks/websphere/techjournal/0603_col_h (consulté en hiver 2009)

[17] Web Services Architecture
http://www.w3.org/TR/ws-arch/ (consulté en hiver 2008)

[18] XML-Signature XPath Filter 1.0
http://www.w3.org/TR/xmldsig-filter2/

[19] Michael McIntosh, Paula Austel. *XML Signature Element Wrapping Attacks and Countermeasures*. Proceedings of the Secure Web Services Workshop 2005. Fairfax, USA, pp. 20-27

[20] M A. Rahama, A. Schaad, and M. Rits. *An Inline Approach for Secure SOAP Requests and Early Validation*. 2005.
http://www.owasp.org/images/4/4b/AnInlineSOAPValidationApproach-MohammadAshiqurRahaman.pdf

[21] A. Rahama, A. Schaad, and M. Rits. *Towards secure soap message exchange in soa*. In SWS '06: Proceedings of the 3rd ACM workshop on Secure web services, pages 77-84, New York, USA, 2006.

[22] S. Gajek, L. Liao, J. Schwenk. *Breaking and Fixing the Inline Approach*. In SWS'07, November 2, 2007, Fairfax, Virginia, USA.

[23] A. Benameur, F. Abdul Kadir, S. Fenet. *XML Rewriting Attacks: Existing Solutions and their Limitations*. In IADIS Applied Computing 2008. IADIS Press, Apr. 2008

[24] S. Kumar Sinha, A. Benameur. A *Formal Solution to Rewriting Attacks on SOAP Messages*. In *SWS'08,* October 31, 2008, Fairfax, Virginia, USA.

[25] Kadima, V. Monfort. 2003. *LES SERVICES WEB*. Paris (France): DUNOD, 407 pages.

[26] Maesano, C. Bernard, X. Le Galles. 2003. *Services web avec J2EE et .NET*. Paris (France): Group Eyrolles, 1041 pages.

[27] G. Gottlob, C. Koch, R. Pichler. *The Complexity of XPath Query Evaluation*. ACM *PODS* 2003, June 912, 2003, San Diego, CA

[28] G. Gottlob, C. Koch, R. Pichler. *Efficient Algorithms for Processing XPath Queries*. Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002