Protocal

Présentation du sujet

Sujet

Création d'un protocole d'échange d'une requête commune et synchrone venant deux serveurs vers un serveur maître. Ce protocole est défini en utilisant le langage et les outils de la suite AVISPA.

Noms des rôles

Pour répondre au sujet, nous avons créé deux serveurs que nous avons nommés "shell" et "code" et un serveur maître "master".

Création d'un plugin pour faciliter le développement

Nous avons remarqué qu'il y avait de nombreuses erreurs de compilation lors de l'écriture du protocol. En effet, il est difficile de ne pas se tromper dans la syntaxe avec un éditeur de texte sans coloration.

Avec Visual Studio Code (VSC) nous pouvons facilement créer un plugin pour obtenir de la coloration de nos fichiers. Grâce à la documentation du projet Avispa décrivant la syntaxe du langage hlpsl, nous avons pu créer un plugin et nous faciliter l'écriture des protocoles et mieux comprendre la grammaire hlpsl.

L'extension est présente dans le market de VSC sous le nom de **hlpsl**.

Première version du protocole (V1)

Cette première version est la première définition de notre protocole, il s'agit d'une représentation naïve du sujet. Nous verrons donc par la suite comment nous avons amélioré cette version.

```
S ------- {Ns.S}_PKc ------------> C
S <------ {Ns.Nc}_PKs ------------ C
S ------- {Nc}_PKc --------------> C
S <------ {OK}_PKs --------------- C
S ------- {Mess1}_PKc -----------> C
S <------ {Mess1.Mess2}_PKs ------ C
S ------- {OK.Mess1.Mess2}_PKc --> C
S -------------- {Mess1.Mess2}_PKm ------------> M
S <------------- {OK.rep}_PKs ------------------ M
```

Cette première version est présente dans le fichier *protocol_v1.hlpsl*

Choix pour la première version

Nous avons choisi de faire communiquer d'abord les deux premiers serveurs Shell et Code (S et C) pour construire le message.

```
S ------- {Ns.S}_PKc ------------> C
S <------ {Ns.Nc}_PKs ------------ C
S ------- {Nc}_PKc --------------> C
```

Les deux serveurs commencent par s'identifier à l'aide de nonces et des clefs publiques.

```
S <------ {OK}_PKs --------------- C
S ------- {Mess1}_PKc -----------> C
S <------ {Mess1.Mess2}_PKs ------ C
S ------- {OK.Mess1.Mess2}_PKc --> C
```

Lorsque les deux serveurs se sont identifiés, c'est-à-dire à la réception du OK venant de C, ils construisent ensemble le message à envoyer au serveur Master (M).

```
S -------------- {Mess1.Mess2}_PKm ------------> M
S <------------- {OK.rep}_PKs ------------------ M
```

C'est le serveur S qui est chargé de communiquer avec le serveur M. S envoie donc le message complet créé précédemment et attend la réponse de M.

Points à améliorer à la v1

- Les premiers échanges peuvent être intégrés à la construction du message
- Ajouter des "goals" : secret, authentication, ...

- Nous ne nous sommes pas encore posé toutes les questions de sécurité puisque nous n'avons pas encore de session et donc pas d'intrus.

Deuxième version du protocole (V2)

Dans la deuxième version de notre protocole, nous avons pris en compte les différents points à améliorer de la V1.

Nous avons donc commencé par intégrer les nonces à la construction du message à envoyer au serveur. Comme les nonces permettent de garantir la fraîcheur de la communication, il est plus judicieux de les intégrer dans l'échange de messages.

```
S ------- {Mess1.Ns.S}_PKc ------------> C
S <------ {Mess1.Mess2.Ns.Nc}_PKs ------ C
S ------- {OK.Mess1.Mess2.Nc}_PKc -----> C
S -------------- {Mess1.Mess2}_PKm ------------> M
S <------------- {OK.rep}_PKs ------------------ M
```

Création du rôle session

Nous avons créé le rôle session qui est composé de shell et code :

```
role session(
    S, C: agent,
    PKs, PKc, PKm: public_key,
    Mess1, Mess2: text,
    Snd, Rcv: channel(dy)
) def=

composition
    shell(S,C,PKs,PKc,PKm,Mess1,Snd,Rcv)
    /\ code(S,C,PKs,PKc,PKm,Mess2,Snd,Rcv)

end role
```

Nous pouvons donc tester notre protocole avec un intrus :

```
master(m,pkm,Snd,Rcv)
/\ session(s,c,pks,pkc,pkm,mess1,mess2,Snd,Rcv)
/\ session(s,i,pks,pki,pkm,mess1,mess2,Snd,Rcv)
```

Nous noterons que le rôle 'master' est défini directement dans l'environnement ce qui nous permet d'avoir un unique serveur 'master' pour toutes les sessions.

Création d'un goal

Nous avons créé un goal pour garantir le secret des nonces :

```
goal

    secrecy_of sns, snc % Keep secret the nonces
    authentication_on shell_code_nc
    authentication_on code_shell_ns

end goal
```

Nous avons trouvé une faille lors de l'ajout d'un intrus. Nous verrons dans la V3 comment nous avons corrigé cela.

Modifications mineures

Lors de la relecture de notre première version, nous avons découvert quelques incohérences comme la présence de la variable 'M' dans les rôles shell et code alors que cette dernière n'est pas utilisée.

Points à améliorer à la v2

- Le protocole n'est pas safe sur un test simple (avec une session comprenant un intrus)
- Faire plus de tests (avec des sessions différentes)
- Il faut créer un deuxième goal (authentication forte par exemple)

Troisième version du protocole (V3)

La description des messages de la V3 n'a pas beaucoup changé par rapport à la V2. Nous devions ajouter un détail pour corriger le problème "UNSAFE" vu précedemment.

Correction du protocole

Nous avons simplement fait signer le nonce par son emmetteur. Nous nous retrouvons donc avec les messages suivants :

```
S ------- {Mess1.Ns.S}_PKc ------------> C
S <------ {Mess1.Mess2.Ns.Nc.C}_PKs ---- C
S ------- {OK.Mess1.Mess2.Nc}_PKc -----> C
S -------------- {Mess1.Mess2}_PKm ------------> M
S <------------- {OK.rep}_PKs ------------------ M
```

Ajout de goals

Nous avons créé un goal pour garantir le secret des nonces :

```
goal

    secrecy_of sns, snc
    authentication_on shell_code_nc
    authentication_on code_shell_ns

end goal
```

Nous avons ajouté deux goals pour l'authentification des utilisateurs. Nous nous sommes reporté sur la documentation pour faire le choix entre weak et strong athentication.

L'authentification forte nous permet d'être sûrs qu'une fois que Shell a fini le protocole avec Code, alors c'est que Code a auparavant terminé le même protocole.

Avec l'authentification forte il y a une notion temporelle qui nous permet de savoir que C a fini le même protocol **récemment** :

Les nonces nous permettent de garantir la fraîcheur des échanges. Ce sont des valeurs fraîchement créées lors de l'éxecution du protocole. On est donc sûr que le tour de protocole du deuxième acteur est après le premier, car les valeurs sont fraîches.

Améliorations possibles

Pour la partie synchronisation des deux serveurs, nous avons cherché des alternatives au simple échange initial entre C et S. Notre idée étant de rajouter une sorte d'horloge interne qui restreint les possibilités de communication entre les deux serveurs. Plus simplement, un acteur ne peut envoyer qu'un seul message tous les x ticks de l'horloge afin d'assurer la synchronicité. La mise en place de cette amélioration réduit grandement la facilité de compréhension de notre protocole et n'apporte pas un réel bénéfice dans le cadre de la modélisation sous AVISPA.

Nous avons aussi réfléchi à rendre la requête commune secrète de telle sorte que l'intrus ne puisse pas lire le contenu de cette requête. Ce qui aurait pu être intéressant, c'est de s'approcher de l'implémentation d'HTTPS (HTTP over TLS) qui est un protocole de communication protégeant les acteurs des attaques de type man-in-the-middle.

En pratique, l'implémentation d'HTTPS nécessite l'intervention d'une autorité de certification, des acteurs possédant des certificats valides, des clefs publiques, privées et une clef de session générée pour l'encryption des messages. L'ensemble de ces prérequis est modélisable sous AVISPA mais le protocole final obtenu serait beaucoup plus complexe, ce qui n'était pas forcément l'objectif de ce projet.