Olivier Heen, Thomas Genet, Erwan Houssay – firstname.lastname@irisa.fr

VOTRE PROTOCOLE EST-IL VÉRIFIÉ ?

mots clés : *protocoles / vérification formelle / animation de spécifications*

Comme l'indiquent plusieurs articles de ce dossier, des techniques de fuzzing peuvent être appliquées aux protocoles. Elles permettent alors de détecter des vulnérabilités sur des implémentations particulières. Mais qu'en est-il de la détection des erreurs de conception ? Comment s'assurer qu'un nouveau protocole est sain d'un point de vue purement logique avant même de disposer de son implémentation ? Les techniques récentes de vérification automatique de protocoles s'avèrent à cet égard très utiles. Plusieurs outils d'animation et de vérification existent : tout au long de cet article, nous utilisons conjointement SPAN (www.irisa.fr/lande/genet/span) et AVISPA (www.avispa-project.org). D'une certaine manière, l'animation est à la spécification ce que le fuzzing est à l'implémentation.

1. Introduction

Dans cet article, nous montrons comment les outils actuels de spécification et de vérification formelles révèlent très tôt des vulnérabilités qui seraient difficiles à corriger au moment de l'implémentation. Avec un outil d'animation, il est possible de « jouer » avec la spécification du protocole pour varier les scénarios d'exécution, et même tenter des exécutions a priori idiotes. Ce faisant, on retrouve une démarche de type essai/erreur proche du *fuzzing*, mais appliquée au modèle du protocole plutôt qu'à son implémentation : du **fuzzing de spécification**.

Tout d'abord, quels avantages peut-on attendre de la vérification automatique ?

1.1 Retrouver des vulnérabilités connues

Cela peut sembler inutile a priori : si des vulnérabilités sont connues, pourquoi se donner la peine de les retrouver automatiquement ? Nous voyons pourtant deux bonnes raisons de le faire : augmenter la confiance dans l'outil, éviter la réapparition d'anciennes vulnérabilités. On trouve aujourd'hui encore des protocoles souffrant de vulnérabilités pourtant parfaitement connues à l'époque de leur conception (pour prendre un exemple lié à l'implémentation ; le **ping of death** est bien connu contre les piles ICMP des ordinateurs et pourtant on le retrouve maintenant sur certains téléphones IP). Seule l'utilisation systématique d'outils de vérification peut empêcher ce type de réapparitions.

1.2 Trouver des variantes d'attaques

Les attaques retrouvées automatiquement n'apparaissent pas toujours sous leurs formes les plus connues. Cela peut amener les concepteurs à réfléchir autrement ou à mieux évaluer les conséquences d'une attaque. Nous illustrons ce point sur le protocole d'échange de clé Diffie-Hellman décrit en figure 1.

Ce protocole souffre d'une vulnérabilité connue à l'encontre du secret des communications et exploitable par l'attaque de **l'homme du milieu** décrite en figure 2. Comme résultat, les communications de A et de B ne sont plus confidentielles vis-à-vis de l'attaquant.

g et p sont des valeurs numériques adaptées à Diffie-Hellman connues de tous les participants. On note g^x pour $g^x \pmod p$. A et B sont deux participants honnêtes. Na et Nb sont deux entiers aléatoires frais (des nonces).
$A \hookrightarrow B : g^{Na}$
$B \hookrightarrow A : g^{Nb}$
Fin : A et B sont d'accord sur la clé $(g^{Na})^{Nb} = (g^{Nb})^{Na}$.

1 *Échange de clés Diffie-Hellman*

I est l'attaquant, N_i est un nonce choisi par lui.
$A \hookrightarrow I : g^{Na}$
$I \hookrightarrow B : g^{Ni}$
$B \hookrightarrow I : g^{Nb}$
$I \hookrightarrow A : g^{Ni}$
Fin : A et I sont d'accord sur la clé $(g^{Na})^{Ni}$, B et I sont d'accord sur la clé $(g^{Nb})^{Ni}$, I comprend les communications entre A et B.

2 *Attaque de l'homme du milieu*

Lancés sur une spécification formelle du protocole Diffie-Hellman, certains outils exhibent plutôt l'attaque de la figure 3. Il s'agit en fait d'un cas très particulier de la première moitié de l'attaque de l'homme du milieu, mais l'attaquant renvoie g plutôt que g^{Ni}. Un point intéressant est que quiconque observe l'attaque peut capturer la valeur g^{Na} envoyée par A, constater que l'attaquant renvoie g et en déduire que A va dorénavant utiliser la clé g^{Na} pour ses communications. Les communications de A vers B ne sont plus confidentielles du tout : n'importe quel participant peut y accéder.

N.B. : cette attaque fonctionne en fait avec toute valeur g^k renvoyée par l'attaquant telle que g^k est facile à identifier (en particulier g^1, g^2, g^3…). Il ne suffit donc pas à A de vérifier qu'il reçoit autre chose que g pour s'en prémunir.

1.3 Trouver de nouvelles vulnérabilités

Dans certains cas, les plus recherchés en fait, l'outil de vérification peut exhiber des attaques correspondant à des vulnérabilités encore inconnues (voir par exemple [1]).

$A \hookrightarrow I : g^{Na}$
$I \hookrightarrow A : g$
Fin : A et I sont d'accord sur la clé g^{Na}.

3 *Une autre attaque contre Diffie-Hellman*

En pratique, il s'agit rarement de vulnérabilités totalement nouvelles, mais plutôt de combinaisons que les experts n'avaient pas réussi à détecter ou à expliciter totalement.

1.4 Vérifier l'utilité des contre-mesures

Les protocoles complexes incluent souvent plusieurs contre-mesures afin de se prémunir de certaines menaces. Il s'agit par exemple de l'ajout de valeurs aléatoires dans des messages pour compliquer leur rejeu. De telles contre-mesures compliquent les protocoles et peuvent elles-mêmes ajouter des vulnérabilités. Il est donc très intéressant de pouvoir vérifier leur utilité : détecte-t-on réellement des attaques lorsqu'on supprime certaines contre-mesures ? Le cas échéant, les attaques sont-elles bien celles qu'on attendait ou s'agit-il de variantes ? (cf. [2] pour un exemple récent).

1.5 Vérifier des cas non prévus

Les concepteurs font très souvent des hypothèses sur le contexte d'utilisation du protocole et sur les capacités de l'attaquant. Or, les protocoles ont souvent des usages inattendus et sont exposés à des classes d'attaquants plus larges que prévu. À titre d'exemple, le protocole de transmission de données sur câble USB fait l'hypothèse que le câble est sûr, mais cette hypothèse devient fausse si la transmission de données est émulée sur IP ou WiFi.

Les outils de vérification automatique ne font pas d'hypothèse a priori sur la sécurité des canaux de communication (nous reviendrons sur ce point). Ainsi, la vérification permet éventuellement de trouver des attaques peu réalistes dans le contexte nominal, mais possibles en théorie. C'est alors au concepteur de décider ou non de la mise en place d'une contre-mesure ou, à défaut, d'un avertissement sur les conditions d'utilisation.

2. Le cœur du métier

Une spécification formelle de protocole est, essentiellement, une description en langage mathématique des échanges de messages effectués entre les différents intervenants (ou agents) du protocole. Si ces spécifications étaient encore extrêmement génériques et théoriques dans les années 90, elles sont maintenant beaucoup plus proches de la réalité des protocoles et décrites dans un langage dédié, comme le langage de ProVerif [3] ou le langage HLPSL pour *High Level Protocol Specification Language* [4]. Dans cet article, nous nous intéressons aux spécifications formelles HLPSL. Celles-ci

font abstraction des détails d'implémentation du protocole et se concentrent sur les mécanismes qui établissent les propriétés de sécurité. En conséquence, si une vulnérabilité est trouvée sur la spécification formelle, il y a peu de chances qu'elle puisse être corrigée au moment de l'implémentation. La spécification déclare également les propriétés attendues sur le protocole considéré : essentiellement le secret d'une donnée, l'authentification d'un message ou d'un agent.

Une fois la spécification rédigée, elle peut être automatiquement analysée par un outil de vérification formelle (nous utilisons AVISPA) afin d'en découvrir les vulnérabilités ou au contraire d'en garantir la sécurité. L'outil peut produire une trace d'attaque invalidant les propriétés déclarées et révélant donc une vulnérabilité dans la spécification. En plus d'être vérifiées formellement, les spécifications HLPSL restent proches d'un langage de programmation et peuvent donc être exécutées et simulées. Cette faculté est centrale dans la confiance que l'on peut accorder à une spécification formelle. Comme le dit Donald Knuth, « un algorithme doit être vu pour être cru ». Il en va de même pour les spécifications formelles et leur simulation.

...Si une vulnérabilité est trouvée sur la spécification formelle, il y a peu de chances qu'elle puisse être corrigée au moment de l'implémentation...

2.1 Des hypothèses réductrices, mais pas trop

Pour prendre en compte un intrus dans la vérification, il est nécessaire de modéliser également celui-ci, c'est-à-dire de définir son comportement. La modélisation de l'intrus est un problème délicat, car, en définissant son comportement, on le limite nécessairement. À ce niveau, les hypothèses communément utilisées sont regroupées sous le nom de modèle de Dolev-Yao **[5]**. Ce modèle s'articule autour de deux hypothèses centrales qui peuvent être résumées de la façon suivante : **le chiffrement est parfait** et **l'intrus est le réseau**. Le chiffrement parfait est une restriction sur les capacités de déchiffrement de l'intrus : on le considère incapable de récupérer ne serait-ce qu'un bit d'information d'un message chiffré s'il ne dispose pas de la clé de déchiffrement. La seconde hypothèse, au contraire, approche par le haut les capacités de l'intrus : dire que l'intrus est le réseau signifie concrètement que les agents lui envoient directement leurs messages. Il peut, par la suite, les transmettre ou non à leur destinataire, choisir un autre destinataire, les dupliquer, les détruire... Il est également capable de déchiffrer des messages s'il a appris, par ailleurs, la clé de déchiffrement. Enfin, il peut expédier à n'importe quel agent un message produit à partir de la somme de ses connaissances et le chiffrer avec les clés qu'il détient.

On peut revenir sur la première hypothèse et se demander si elle est raisonnable. L'hypothèse du chiffrement parfait, très forte, fait totalement abstraction du message à chiffrer, du type d'algorithme de chiffrement ou de la taille de la clé utilisée. En pratique, il est possible de rendre cette hypothèse plausible en choisissant, lors de l'implémentation du protocole, des messages, algorithmes et clés de chiffrement résistant à des attaques cryptanalytiques[1]. Ceci est en accord avec la stratégie de recherche d'attaques utilisée ici qui se focalise plus sur les vulnérabilités liées à une mauvaise conception que celles dues à un mauvais choix d'implémentation. En outre, des travaux récents montrent que cette vue dégradée de la cryptographie est recevable également en théorie [6]. Ces travaux montrent en particulier, qu'en choisissant bien les schémas de chiffrement cryptographiques, les preuves réalisées dans le modèle de Dolev-Yao sont valides dans le modèle calculatoire[2].

2.2 Le prix à payer

Actuellement, même si certains efforts vont dans ce sens **[12]**, l'extraction d'une spécification formelle de protocole directement à partir de son code source reste difficile. Il revient donc au concepteur du protocole de réaliser ce modèle à la main en faisant les abstractions nécessaires.

Il existe plusieurs langages de spécification. Certains langages sont basés uniquement sur la description des messages du protocole. D'autres langages sont basés sur une description complète des états et des transitions des agents participant au protocole. Le langage HLPSL que nous montrons ici fait partie de la seconde catégorie. Nous l'utilisons pour spécifier le protocole Diffie-Hellman de la figure 1. Il y a deux **rôles** dans la spécification, appelés `alice` et `bob`. Dans chaque rôle, le langage impose une définition explicite des messages, des transitions et des objets utilisés comme les clés, les nonces, les états... Les transitions s'écrivent sous la forme générale `évènements =|> réactions`. Le symbole `/\` permet la conjonction de plusieurs évènements ou réactions.

```
role alice(A,B:agent, G:text, Snd,Rcv:channel(dy)) played_by A def=
    local State:nat, Na,Nsecret:text, X,K:message
    init State:=1
transition
1.  State=1 /\ Rcv(start) =|> State':=2 /\ Na':=new() /\ Snd(exp(G,Na'))
2.  State=2 /\ Rcv(X') =|> State':=3 /\ K':=exp(X',Na) /\ Nsecret':=new()
    /\ Snd({Nsecret'}_K')
end role
```

[1] On peut par exemple suivre les résultats de concours de factorisation de nombres RSA pour savoir quelle taille de clés RSA choisir afin se placer au-delà de ce qui est actuellement déchiffrable par cryptanalyse.

[2] Le modèle utilisé par les cryptographes pour garantir la sécurité des algorithmes de chiffrement dans lequel on relie une propriété de sécurité avec un problème calculatoirement difficile.

On peut remarquer la présence de plusieurs variables, avec ou sans prime. Dans la première transition par exemple, la réaction `Na':=new()` signifie que la variable `Na` prend une nouvelle valeur aléatoire. Dans la première transition, le terme `Snd(exp(G,Na'))` représente l'envoi du message g^{Na}. Dans la seconde transition, l'évènement `Rcv(X')` signifie que la variable `X` prend pour nouvelle valeur le contenu du message reçu par l'agent. Cette valeur `X'` est alors utilisable dans la partie réaction : l'expression `K':= exp(X',Na)` signifie ainsi que la clé `K` prend pour nouvelle valeur l'exponentielle de `X'` par la valeur courante de `Na`.

Le rôle `bob` est défini de la même manière. Il contient aussi deux transitions. La première est déclenchée à la réception d'un message de la part d'`alice` avec pour réaction le calcul de la clé `K'`. La seconde transition n'est là que pour montrer la possibilité de recevoir un message secret chiffré par `alice` avec la clé `K`. En HLPSL, ceci est noté `Rcv({Nsecret'}_K)` ce qui signifie que la variable `Nsecret` prend pour nouvelle valeur le résultat du déchiffrement par `K` du message reçu. Ce point illustre une partie de la puissance du langage : de manière très statique, il est possible de nommer le résultat d'un déchiffrement, alors même qu'on ne sait pas si la clé sera la bonne au moment de l'exécution. En effet, si la clé `K` n'est pas exactement celle qu'`alice` a utilisé (par exemple en cas d'attaque), alors le terme `{Nsecret'}_K` ne définit aucune valeur pour `Nsecret'`.

```
role bob(B,A:agent, G:text, Snd,Rcv:channel(dy)) played_by B def=
    local State:nat, Y,K:message, Nb,Nsecret:text
    init State:=1
transition
1.  State=1 /\ Rcv(Y') =|> State':=2 /\ Nb':=new() /\ K':=exp(Y',Nb') /\
    Snd(exp(G,Nb'))
2.  State=2 /\ Rcv({Nsecret'}_K) =|> State':=3
end role
```

Une fois définis, tous les rôles du protocole peuvent être composés en sessions. Pour chaque session, de la connaissance peut être partagée entre les différents rôles. Dans notre exemple, la valeur de `G` est ainsi connue de tous les participants. Voici comment s'écrit une session simple mettant en présence un agent `alice` et un agent `bob`.

```
role session (A,B:agent, G:text) def=
    local SND_A,RCV_A,SND_B,RCV_B:channel(dy)
composition
    alice(A,B,G,SND_A,RCV_A) /\ bob(B,A,G,SND_B,RCV_B)
end role
```

Il reste à définir un environnement pour l'exécution de la (ou des) session(s). Cet environnement contient systématiquement un agent supplémentaire noté `i` pour intrus. L'environnement définit en particulier la connaissance initiale de l'intrus, le jeu de valeurs initiales et les sessions à exécuter.

```
role environment() def=
    const a,b:agent, g:text
    intruder_knowledge={g,a,b}
composition
    session(a,b,g,Snd,Rcv)
end role
```

Le tout dernier point consiste en l'expression formelle des propriétés de sécurité à vérifier. HLPSL permet d'exprimer deux types de propriétés : le secret de valeurs et l'authentification de participants.

Pour le secret, il s'agit d'indiquer quelles valeurs doivent être connues de quels agents, et surtout d'eux seuls ! La forme générale pour exprimer ceci est `secret(T,t,{A,B,...})` où `T` est la valeur que seulement `{A,B,...}` doivent connaître. Le littéral `t` est simplement une façon d'identifier cette propriété au moment de la recherche d'attaque, sous la forme `secrecy_of t`. Ainsi, dans notre spécification, nous pouvons ajouter `secret(Nsecret',t,{A,B})` juste après la création de la valeur `Nsecret'` par `alice`. Au moment de vérifier une session, nous ajoutons le but `secrecy_of t`.

Attention, si jamais l'attaquant arrive à jouer le rôle de `A` dans une session, alors il arrivera à connaître `T` et le but `secrecy_of t` ne sera pas atteint. C'est typiquement ce qui arrive dans les attaques par usurpation d'identité.

```
role alice(A,B:agent, G:text, Snd,Rcv:channel(dy)) played_by A def=
    local State:nat, Na,Nsecret:text, X,K:message
    init State:=1
transition
1.  State=1 /\ Rcv(start) =|> State':=2 /\ Na':=new() /\ Snd(exp(G,Na'))
2.  State=2 /\ Rcv(X') =|> State':=3 /\ K':=exp(X',Na) /\ Nsecret':=new() /\
    Snd({Nsecret'}_K')
    /\ secret(Nsecret',t,A,B)
end role

role bob(B,A:agent, G:text, Snd,Rcv:channel(dy)) played_by B def=
    local State:nat, Y,K:message, Nb,Nsecret:text
    init State:=1
transition
1.  State=1 /\ Rcv(Y') =|> State':=2 /\ Nb':=new() /\ K':=exp(Y',Nb') /\
    Snd(exp(G,Nb'))
2.  State=2 /\ Rcv({Nsecret'}_K) =|> State':=3
end role

role session (A,B:agent, G:text) def=
    local SND_A,RCV_A,SND_B,RCV_B:channel(dy)
composition
    alice(A,B,G,SND_A,RCV_A) /\ bob(B,A,G,SND_B,RCV_B)
end role

role environment() def=
    const a,b:agent, g:text
    intruder_knowledge={g,a,b}
composition
    session(a,b,g,Snd,Rcv)
end role

goal secrecy_of t end goal
environment()
```

4 *Spécification HLPSL du protocole Diffie-Hellman.*

Pour vérifier l'authenticité de certains participants, il est nécessaire d'exprimer des propriétés supplémentaires. Les mots-clés `request`, `witness` et `authentication_on` permettent de le faire. La forme générale `request(A,B,abk,K)` signifie que `A` veut s'assurer que la valeur `K` a bien été créée par `B` spécialement pour cette occasion (en particulier, la valeur n'est pas rejouée d'une session précédente). Chaque `request` est apparié à un `witness` de la forme `witness(B,A,abk,K')` indiquant que `B` fournit la valeur fraîche `K'` à `A` comme témoin de son identité. Pour une paire `request / witness`, on peut demander la vérification du but `authentication_on abk`.

Au final, on obtient une spécification complète du protocole augmentée des définitions des propriétés de sécurité, des sessions et des buts relatifs aux propriétés de sécurité. Dans l'exemple de Diffie-Hellman, l'ensemble fait environ 40 lignes de code, indiquées en figure 4, page précédente.

À partir de ce point, deux questions importantes se posent, auxquelles nous répondons dans les paragraphes suivants. La spécification correspond-elle **vraiment** au protocole ? Les propriétés de sécurité sont-elles toujours vérifiées ?

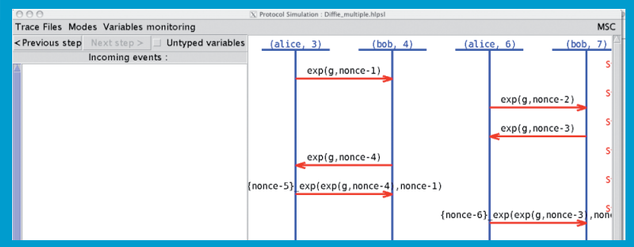
3. La spécification correspond-elle au protocole ?

Tous les diagrammes indiqués dans cette section sont réalisés avec l'outil SPAN.

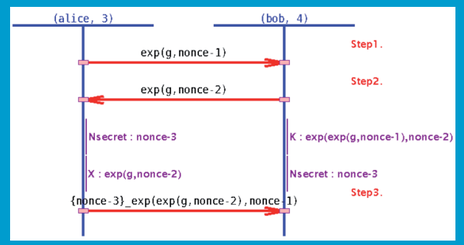
La visualisation et l'animation des spécifications représentent une étape importante du processus de vérification. En premier lieu, il s'agit d'être certain que la spécification rédigée correspond bien à la réalité du protocole. Comme en programmation, une petite faute de frappe peut changer complètement le sens de la spécification. Le langage HLPSL est extrêmement permissif. Cela participe à sa puissance, mais augmente aussi le risque d'erreur non détectée. SPAN prend en entrée une spécification HLPSL, par exemple celle de la figure 4, et propose à l'opérateur de déclencher une transition parmi toutes les transitions possibles dans la session simulée. L'opérateur déclenche une transition, SPAN calcule toutes les conséquences, propose un choix parmi les nouvelles transitions possibles s'il y en a, et ainsi de suite. Ainsi, l'opérateur s'assure que sa spécification produit bien le cas nominal d'utilisation du protocole. C'est ce que montre la figure 5. Il est également possible de visualiser les valeurs des variables entre les différentes transitions, figure 6.

Une fois persuadé que le cas nominal est correctement atteint, l'opérateur peut essayer des sessions plus complexes dans des combinaisons variées. Dans notre exemple, nous remplaçons la session unique déclarée dans la section `environment` de la spécification de la figure 4 par

5 *Début d'animation de la spécification HLPSL de Diffie-Hellman. L'outil a automatiquement instancié l'agent représentant Alice en (alice,3) et l'agent représentant Bob en (bob,4). Dans le volet « incomming event », on voit que l'opérateur peut encore déclencher une transition (alice,3) -> (bob,4).*



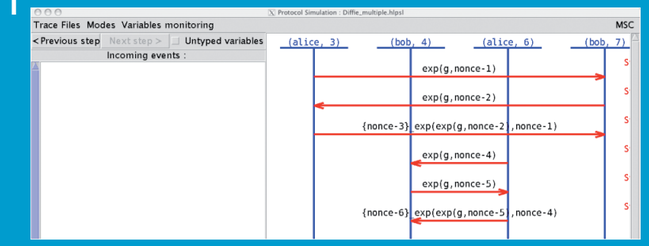
7 *Entrelacement de sessions session(a,b,g,Snd,Rcv) /\ session(c,d,g,Snd,Rcv). Avec appariement de a avec b et c avec d. L'outil a automatiquement instancié a=(alice,3) b=(bob,4) c=(alice,6) d=(bob,7).*



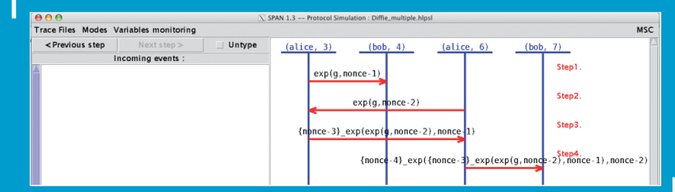
6 *La clé de session K est bien détenue par bob. Le secret Nsecret choisi par alice a bien été reçu par bob.*

`session(a,b,g,Snd,Rcv) /\ session(c,d,g,Snd,Rcv)`, où `c` et `d` sont des nouveaux agents. On peut obtenir ainsi les figures 7 et 8 avec divers entrelacements de sessions Diffie-Hellman.

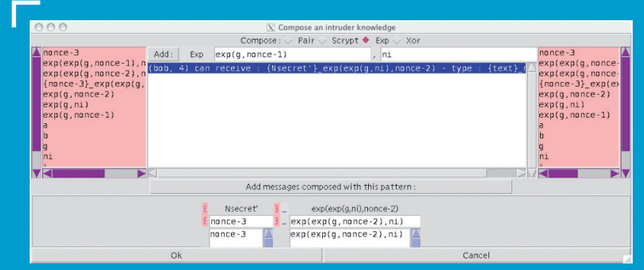
On peut enfin, et c'est là un grand intérêt de l'outil, essayer des cas non prévus. Il ne s'agit pas encore de simuler l'intrus, mais plutôt de rechercher des cas autorisés par la spécification formelle qui ne correspondent à aucune réalité.



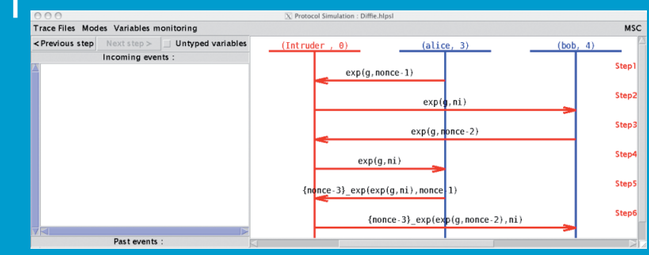
8 *Entrelacement de sessions session(a,b,g,Snd,Rcv) /\ session(c,d,g,Snd,Rcv). Avec appariement de a avec d et c avec b. L'outil a automatiquement instancié a=(alice,3) b=(bob,4) c=(alice,6) d=(bob,7).*



9 *Un exemple de « fuzzing de spécification ». Des transitions possibles ont été déclenchées aléatoirement.*



10 *Interface de construction de messages par l'intrus. Toute la connaissance accumulée est disponible et peut être composée avec les opérateurs pair, Scrypt, Exp, Xor selon des patterns attendus par les agents participant au protocole.*



11 *L'attaque complète de l'homme du milieu : l'intrus transmet le secret nonce-3 à bob*

Par exemple, la figure 9 montre l'entrelacement de sessions Diffie-Hellman où l'agent `(alice,3)` confond l'invite de `(alice,6)` avec une réponse correcte de la part d'un agent `bob`. C'est ici qu'on peut retrouver les réflexes du fuzzing : secouer un peu les spécifications pour en extraire des comportements dont il faudra peut-être se prémunir par la suite. L'analogie va même plus loin lorsque l'opérateur choisit aléatoirement la prochaine transition dans l'ensemble des transitions possibles. On obtient alors des diagrammes d'exécution valides en regard des spécifications, mais parfois totalement incohérents en regard du comportement obtenu.

L'outil offre un mode **simulation d'intrus** qui permet d'ajouter explicitement l'intrus dans les transitions possibles. L'intrus peut à tout moment capter un message ou forger n'importe quel message à partir de sa connaissance initiale et de tout ce qu'il a capté. Pour ce faire, l'outil propose une interface de construction de messages reproduite en figure 10. La figure 11 montre comment l'intrus capture un secret d'`alice` et l'utilise pour forger un message à l'intention de `bob`.

Simuler explicitement l'intrus donne énormément d'information à l'opérateur qui peut ainsi mieux se rendre compte de la faisabilité d'une attaque, trouver de nouvelles attaques à la main, rejouer des attaques trouvées par les outils automatiques ou trouvées dans la littérature, chercher des variantes d'attaques connues…

Le côté pratique est important : il est possible de sauver des traces d'exécution ou d'attaque et de les rejouer sur les spécifications en questions. Ainsi, on peut conserver les traces et tenter de les rejouer plus tard sur des versions corrigées des protocoles.

4. Les propriétés de sécurité sont-elles vérifiées ?

4.1 Preuves d'insécurité

AVISPA est un ensemble d'outils pour la recherche d'attaques et la validation de protocoles. Les outils actuellement disponibles sont : OFMC, CL-AtSe, SATMC et TA4SP. Nous ne détaillons pas plus leurs spécificités ici. Les propriétés de sécurité actuellement accessibles à ces outils sont soit liées au secret de valeurs, soit liées à l'authentification des participants. C'est un axe de recherche important que d'étendre ces propriétés à, par exemple, l'anonymat, la non-répudiation, la simultanéité d'évènements...

La figure 12 montre le résultat d'AVISPA sur notre spécification de Diffie-Hellman. On retrouve bien là l'attaque de la figure 3, différente de l'attaque de l'homme du milieu classique de la figure 2.

```
SUMMARY
 UNSAFE
DETAILS
 ATTACK_FOUND
PROTOCOL
 Diffie-Hellman.if
GOAL
 secrecy_of_t
BACKEND
 OFMC
STATISTICS
 parseTime: 0.00s
 searchTime: 0.01s
 visitedNodes: 1 nodes
 depth: 1 plies
ATTACK TRACE
i -> (a,3): start
(a,3) -> i: exp(g,Na(1))
i -> (a,3): g
(a,3) -> i: {Nsecret(2)}_(exp(g,Na(1)))
```

12 *Sortie d'AVISPA sur la spécification de Diffie-Hellman.*

Plusieurs protocoles ont ainsi été spécifiés et vérifiés. Ils sont disponibles sur le site d'AVISPA **www.avispa-project.org** dans la rubrique *The AVISPA Library of Protocols*. On dénombre ainsi une cinquantaine de protocoles IETF vérifiés dont TLS, ssh-transport ou encore diverses parties de IKEv2. On compte également une vingtaine de protocoles non-IETF, dont notamment SET pour le paiement électronique. Sur l'ensemble de ces 70 protocoles, une douzaine d'attaques ont été trouvées ou retrouvées.

Pour illuster cet article, nous avons choisi des protocoles réels, mais suffisamment simples pour que leurs spécifications HLPSL restent de l'ordre de la centaine de lignes. Les protocoles de paiement bancaire par carte à puce sont relativement simples, leur fonctionnement est bien connu et leurs vulnérabilités aussi. Nous nous intéressons ici uniquement aux protocoles hors lignes utilisés, par exemple, chez les commerçants. Pour la majorité des transactions, le terminal du commerçant doit vérifier la validité de la carte et des informations bancaires reçues sans communiquer avec un central bancaire externe.

Le protocole initial, tel qu'il existait à l'époque de l'attaque de Serge Humpich et des premières Yescard, se nomme B0'. Selon [7], il peut être décrit de la façon suivante, cf. figure 13.

Card ↪ *Term: Data,*$\{hash(Data)\}_{KBank}^{-1}$
Term ↪ *User: Enter PIN?*
User ↪ *Term: 4567*
Term ↪ *Card: 4567*
Card ↪ *Term: ok*

13 *Protocole bancaire B0'*

où *User* est le détenteur la carte *Card* qui lui a été remise par sa banque *Bank* (pour simplifier) et *Term* est le terminal du commerçant. Le protocole débute lorsque *User* introduit sa carte dans *Term*. Dans le premier message du protocole, *Card* envoie à *Term* ses données bancaires *Data= (nom,prénom,numéro de carte, date de validité)* suivies de la valeur de signature $\{hash(Data)\}_{KBank}^{-1}$. Cette valeur, calculée et enregistrée dans la puce au moment de la création de la carte, consiste en un hachage de *Data* signé par la clé privée de *Bank*, notée ici K_{Bank}^{-1}. Le terminal *Term* dispose, lui, de la fonction *hash* et de la clé publique de la banque, notée K_{Bank}. Quand il reçoit le premier message, *Term* peut calculer, d'une part, une valeur *x* en appliquant *hash* à *Data* et, d'autre part, une valeur *y* en déchiffrant la valeur de signature $\{hash(Data)\}_{KBank}^{-1}$ avec la clé publique K_{Bank}. Pour vérifier que *Card* a bien émis des informations bancaires *Data* signées par *Bank*, il suffit au terminal de vérifier qu'il a bien *x=y=hash(Data)*. Si c'est le cas, il émet le second message à destination de *User* l'invitant à saisir son code. *User* tape son code qui est transmis par *Term* à *Card*. Si *Card* reconnaît son code, elle émet un message d'acquittement figuré ici par 'ok'.

La spécification HLPSL du protocole B0' fait moins de 60 lignes. Elle est donnée en figure 14.

Dans cette spécification, les capacités de l'intrus sont volontairement plus limitées que dans le cas d'un intrus Dolev-Yao général. Une analyse de ce protocole dans le cas où l'intrus a effectivement un contrôle total sur toutes les communications aurait été possible. Cela aurait supposé que l'intrus ait par

```
role user(User,Term:agent, Data,PIN:message,KSUT:(symmetric_key) set,
    SND,RCV:channel(dy)) played_by User def=
    local State:nat,K:symmetric_key
    init State:=0
transition
0.  State=0 /\ RCV(enter_pin) =|> State':=1 /\ SND({PIN}_K') /\ K':=new() /\ KSUT':=cons(K',KSUT)
    /\ witness(User,Term,data_ut,Data)
end role

role term(Term,User,Card:agent, Kbank:public_key, Hash:function, KSUT,KSTC:(symmetric_key) set,
    SND,RCV:channel(dy)) played_by Term def=
    local State:nat,Data, PIN:message, KU,KC:symmetric_key
    init State:=0
transition
1.  State=0 /\ RCV(Data'.{Hash(Data')}_inv(Kbank)) =|> State':=1 /\ SND(enter_pin)
2.  State=1 /\ in(KU',KSUT) /\ RCV({PIN'}_KU') =|> State':=2 /\ KC':=new() /\ SND({PIN'}_KC')
    /\ KSTC':=cons(KC',KSTC)
3.  State= 2 /\ RCV(ok) =|> State':=3 /\ request(Term,User,data_ut,Data)
end role

role card(Card,Term:agent, Kbank:public_key, Hash:function, Data,PIN:message,
    KSTC:(symmetric_key) set, SND,RCV:channel(dy)) played_by Card def=
    local State:nat, K:symmetric_key
    init State:=0
transition
0.   State=0 /\ RCV(start) =|> State':=1 /\ SND(Data.{Hash(Data)}_inv(Kbank))
1.   State= 1 /\ in(K',KSTC) /\ RCV({PIN}_K') =|> State':=2 /\ SND(ok)
end role

role session(User,Term,Card:agent, Kbank:public_key, Hash:function,
    KSUT,KSTC:(symmetric_key) set, Data,PIN:message) def=
    local SndU,RcvU,SndT,RcvT,SndC,RcvC:channel(dy)
composition
    user(User,Term,Data,PIN,KSUT,SndU,RcvU) /\ term(Term,User,Card,Kbank,Hash,KSUT,KSTC,SndT,RcvT)
 /\ card(Card,Term,Kbank,Hash,Data,PIN,KSTC,SndC,RcvC)
end role

role environment() def=
    local Ksut1,Kstc,Ksut2:(symmetric_key) set
    const enter_pin,ok,data1,code1,data2,code2:message, kbank:public_key,
    fhash:function,u1,u2,t,c1,c2:agent, data_ut:protocol_id
    init Ksut1:={} /\ Kstc:={} /\ Ksut2:={}
    intruder_knowledge={u1,u2,t,c1,c2}
composition
    session(u1,t,c1,kbank,fhash,Ksut1,Kstc,data1,code1)
 /\ session(u2,t,c2,kbank,fhash,Ksut2,Kstc,data2,code2)
end role

goal authentication_on data_ut end goal
environment()
```

14 *Spécification HLPSL de B0'*

exemple accès aux communications entre *User* et *Term*. Sous ces hypothèses, on trouve des attaques beaucoup plus simples, puisque l'intrus a accès, en particulier, au code secret tapé par *User*. Nous avons donc choisi de nous placer dans un cas plus réaliste où il existe des canaux de communication inaccessibles à l'intrus : un canal privé entre *User* et *Term* – l'intrus ne peut pas lire le code de *User* par dessus son épaule ou taper sur le clavier à sa place – et un autre entre *Term* et *Card* – il ne cherche pas à attaquer le matériel pour lire ou brouiller les communications entre la carte et le terminal. Dans la spécification, le canal privé entre *User* et *Term* est représenté à l'aide d'un ensemble `Ksut` partagé entre ces deux agents contenant des clés de chiffrements renouvelées à chaque nouveau message expédié sur le canal. Ce codage permet d'assurer la confidentialité et le non-rejeu des messages échangés sur le canal. On utilise un codage similaire pour le canal entre *Term* et *Card*.

Dans la spécification HLPSL de la figure 14, nous nous intéressons au scénario de vérification suivant : deux sessions du protocole en parallèle concernant deux *User* différents, deux *Card* différentes et un même *Term*. Cette spécification utilise les mots-clés `witness` et `request` pour exprimer l'authentification de l'utilisateur par le terminal (au travers de sa carte). La figure 15, page suivante, montre le résultat d'AVISPA sur notre spécification de B0'.

Dans ce résultat d'attaque, `i` représente l'intrus, `(c1,5)` et `(c2,9)` les deux cartes, `(u1,3)` l'utilisateur détenteur de la carte `(c1,5)`. Enfin, `(t,4)` et `(t,8)` représentent le terminal impliqué dans deux sessions différentes. En bref, cette attaque montre d'abord que les données bancaires `data2` de la deuxième carte `(c2,9)` sont acceptées par le terminal `(t,4)` (4ème bloc du script de l'attaque). Ensuite, le code `code1` tapé par `(u1,3)` (5ème bloc) est transmis à la carte `(c1,5)` (8ème bloc). La carte émet son signal d'acquittement 'ok' (9ème bloc) et il est accepté par le terminal `(t,4)` (10ème bloc). Cette attaque, même si elle semble difficile, était réalisable en pratique sur certains terminaux à l'aide d'un dispositif physique empêchant la détection du retrait d'une carte. Dans cette attaque dite « **par interversion de cartes** », il était ainsi possible de faire authentifier la première carte, de la retirer, d'insérer la seconde et de saisir son code sur la seconde. À l'issue, la première carte était débitée en utilisant le code de la seconde. Cette vulnérabilité a été exploitée de façon plus radicale dans les Yescards qui utilisaient soit des valeurs de signature existantes (recopiées sur

```
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
  TYPED_MODEL
PROTOCOL
  CB-3d.if
GOAL
 Authentication attack on (t,u1,data_ut,data2)
BACKEND
  CL-AtSe
STATISTICS
  Analysed  : 878 states
  Reachable : 310 states
  Translation: 0.13 seconds
  Computation: 0.02 seconds
ATTACK TRACE
  i -> (c2,9): start
  (c2,9) -> i: data2.{{data2}_fhash}_(inv(kbank))

  i -> (c1,5): start
  (c1,5) -> i: data1.{{data1}_fhash}_(inv(kbank))

  i -> (t,8): data1.{{data1}_fhash}_(inv(kbank))
  (t,8) -> i: enter_pin

  i -> (t,4): data2.{{data2}_fhash}_(inv(kbank))
  (t,4) -> i: enter_pin

  i -> (u1,3): enter_pin
  (u1,3) -> i: {code1}_n1(K)
                & Witness(u1,t,data_ut,data1);

  i -> (t,4): {code1}_n1(K)

  (t,4) -> i: {code1}_n4(KC1)

  i -> (c1,5): {code1}_n4(KC1)

  (c1,5) -> i: {ok}_n10(K2)

  i -> (t,4): {ok}_n10(K2)

  (t,4) -> i: ()
                & Request(t,u1,data_ut,data2);
```

15 *Sortie d'AVISPA sur la spécification de B0'*

des cartes dérobées), soit des valeurs de signature générées aléatoirement[3].

Les groupements bancaires ont répliqué par plusieurs protocoles, dont EMV/DDA (*Dynamic Data Authentication*). EMV/DDA est censé équiper 98% des terminaux en France **[11]**. La figure 16 montre son fonctionnement, tel que déduit de spécifications de www.emvco.com.

Card ↪ *Term*: $\{K_{Bank}\}K_{CA}^{-1}$, $\{K_{Card}\}K_{Bank}^{-1}$,*Data*,$\{hash(Data)\}K_{Bank}^{-1}$
Term ↪ *Card*: N_{Term}
Card ↪ *Term*: $\{N_{Term}\}K_{Card}^{-1}$
Term ↪ *User*: *Enter PIN?*
User ↪ *Term*: *4567*
Term ↪ *Card*: $\{4567\}K_{Card}$
Card ↪ *Term*: *ok,tc*

16 *Protocole bancaire EMV/DDA*

Les nouveautés par rapport au protocole B0' sont les suivantes :

- Chaque carte dispose d'un couple de clés RSA K_{Card} et K_{Card}^{-1}.
- Les cartes sont suffisamment puissantes pour effectuer des chiffrements/déchiffrements RSA en utilisant ces clés.
- Le terminal ne dispose que de la fonction *hash* et de la clé publique K_{CA} d'une autorité de certification.
- Le premier message, lorsqu'il est reçu par le terminal, lui permet de récupérer la clé K_{Bank} (signée par *CA*), puis à l'aide de cette dernière de récupérer K_{Card} (signée par *Bank*) et de vérifier l'authenticité de la valeur de signature $\{hash(Data)\}_{KBank}-1$ (signée par *Bank*).
- Le terminal s'assure que la carte détient bien la clé privée K_{Card}^{-1} en lui envoyant un nombre aléatoire N_{Term} et en vérifiant que la carte répond bien par $\{N_{Term}\}_{KCard}^{-1}$ dont il peut vérifier l'authenticité en le déchiffrant à l'aide de K_{Card}.
- Le code secret est expédié chiffré par K_{Card} depuis le terminal vers la carte.
- Le message d'acquittement 'ok' a une valeur statique. Il est suivi d'un enregistrement 'tc', relatif au déroulement de la transaction. Cependant, d'après **[8]**, 'tc' est chiffré avec une clé partagée entre la carte et l'émetteur de la carte (la banque pour simplifier) et cette clé n'est pas connue du terminal. Ce dernier ne peut donc rien en faire, à part le stocker pour vérification future par la banque.

La spécification HLPSL de ce protocole prend environ 200 lignes, nous ne la reproduisons pas ici. Comme le montre la figure 17, les outils d'AVISPA ne retrouvent pas l'attaque par interversion de cartes légales dont souffrait B0'. C'est grâce au chiffrement du code PIN par K_{Card} que l'attaque n'est plus possible. En effet, le code PIN est chiffré avec la clé K_{Card} de la première carte et la carte substituée ne sait pas déchiffrer cette valeur. Il

[3] La clé publique RSA K_{Bank}, de longueur insuffisante à l'époque, pouvait être factorisée, révélant ainsi la clé privée K_{Bank}^{-1}. À l'aide de cette dernière et de la fonction *hash*, il était possible de produire des valeurs de signature pour des données bancaires *Data* quelconques, acceptées par les terminaux.

```
SUMMARY
 SAFE

DETAILS
 BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
 CB-DDA-cartes-legales.if
GOAL
 As Specified
BACKEND
 CL-AtSe
STATISTICS

 Analysed  : 89786 states
 Reachable : 29856 states
 Translation: 0.14 seconds
 Computation: 2.55 seconds
```

17 *Sortie d'AVISPA sur la spécification d'EMV/DDA*

peut néanmoins exister d'autres attaques contre EMV/DDA. Nous en montrons une plus loin.

Dans la section suivante, nous voyons comment l'absence d'attaque détectée par certains outils peut-être vue comme une preuve de sécurité de la spécification.

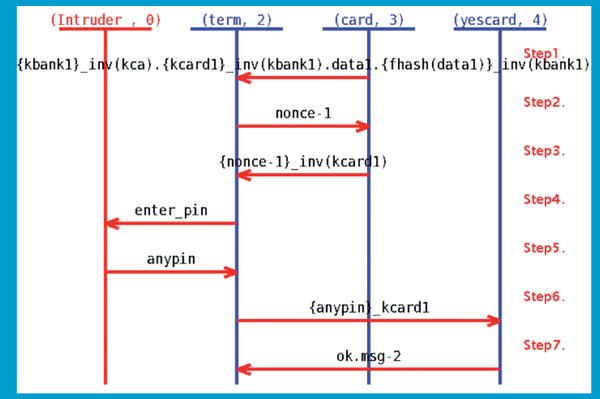
4.2 Preuves de sécurité

Il ne faut pas s'y tromper, les attaques découvertes par AVISPA ne sont pas obtenues en utilisant des heuristiques astucieuses comme cela est fait, par exemple, dans le fuzzing. L'objectif des outils de vérification d'AVISPA est de prouver formellement sur le protocole les propriétés de sécurité déclarées dans la spécification. Lorsque cette preuve automatique échoue, elle produit **systématiquement** un contre-exemple sous la forme d'une trace d'attaque. Ces outils offrent donc une forme de complétude que n'offrent pas des méthodes de génération de test ou des heuristiques de recherche d'attaque. En clair, s'il existe une attaque **sur la spécification formelle**, alors celle-ci sera découverte. À l'inverse, s'il n'existe pas d'attaque sur les sessions déclarées dans la section `environment` de la spécification, ceci pourra également être démontré **[9]**. En effet, l'existence d'une attaque, propriété indécidable en général, devient décidable dans le cas d'un intrus de Dolev-Yao et si l'on se limite à un nombre borné de sessions **[10]**.

...Les outils de vérification offrent donc une forme de complétude que n'offrent pas des méthodes de génération de test ou des heuristiques de recherche d'attaque...

Si l'on revient au protocole DDA, le résultat obtenu plus haut est donc une preuve que la **spécification formelle** de DDA ne contient pas, a priori, de vulnérabilité liée à l'interversion de deux cartes légales. Cette preuve n'est bien sûr valable que dans le cas d'un intrus respectant le modèle de Dolev-Yao et sur un nombre borné de sessions.

On peut bien sûr continuer à jouer avec cette spécification et tester d'autres hypothèses. Par exemple, on peut supposer que l'intrus dispose d'une Yescard spécifique dont le rôle est **uniquement** d'envoyer un couple 'ok,tc' quel que soit le message reçu. On suppose que le 'tc' peut être généré aléatoirement de façon a être accepté par le terminal car, celui-ci ne semblant pas en mesure de le déchiffrer, il ne pourra pas l'authentifier. Dans ce cas, il est possible de retrouver automatiquement une attaque basée sur un échange entre une carte légale et la Yescard spécifique (voir Figure 18).



18 *Trace SPAN d'une attaque théorique sur DDA avec interversion d'une carte légale et d'une Yescard*

note

La réussite de cette attaque dépend grandement des conditions de réalisation. En effet, l'attaque nécessite une fausse carte et une substitution au bon moment. La mise en œuvre de ces deux points peut s'avérer difficile.

Améliorations et conclusion

Plusieurs points restent bien évidemment à améliorer. Nous avons déjà évoqué l'extension des propriétés de sécurité vérifiables automatiquement. Ce point essentiel fait aujourd'hui l'objet de nombreuses recherches.

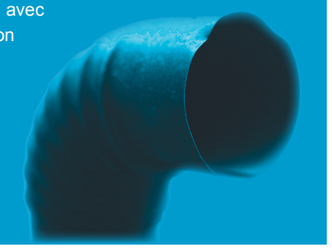
Il serait également très utile de disposer de canaux non entièrement accessibles à l'attaquant. De tels canaux permettraient aux outils de passer outre la détection des attaques les plus immédiates au profit des attaques complexes. En effet, la plupart des outils considèrent qu'aucun canal n'est sûr. C'est un excellent principe de sécurité, mais cela peut provoquer l'arrêt des outils sur une attaque liée à un canal particulier alors qu'il y a peut-être encore d'autres attaques à révéler.

Pour reprendre l'exemple du protocole bancaire B0', les outils détectent d'abord une attaque sur la confidentialité du PIN code. Cette attaque bien connue (dite « *shoulder surfing* ») est liée au manque de confidentialité du canal entre l'utilisateur et le terminal de paiement. Pour autant, une fois cette attaque détectée, on aimerait considérer le canal comme confidentiel et continuer la détection.

Les outils actuels n'offrant pas cette possibilité, il faut procéder à la main, typiquement en pré-distribuant des clés pour sécuriser certains canaux. Dans l'exemple B0' de la figure 14, il s'agit des clés stockées dans les ensembles KSUT et KSTC. De telles modifications sont dangereuses, car elles risquent de faire disparaître d'autres attaques.

Même si elles restent améliorables, les techniques de vérification automatique de protocoles sont déjà très utilisables en pratique. Elle permettent de détecter très tôt des vulnérabilités impossibles à corriger dans des phases ultérieures : on ne peut pas patcher une erreur de conception. L'effort pour arriver à une spécification vérifiable est de plus en plus raisonnable. Avec un peu de pratique, un langage comme HLPSL devient facile à manipuler. Sa puissance d'expression permet de spécifier de nombreux protocoles. Un outil d'animation comme SPAN facilite également le contrôle de la bonne rédaction de la spécification et permet de « secouer » la spécification exactement comme un fuzzer « secoue » une implémentation. Aussi, nous pensons que ce type de vérification devrait avoir lieu systématiquement.

Au final, il devient possible de publier un protocole avec sa preuve de correction et ainsi de répondre à la question posée comme titre de cet article : **Votre protocole est-il vérifié ?**



Références

[1] SANTIAGO (J.) et VIGNERON (L.), « *Optimistic non-repudiation protocol analysis* », LNCS 4462:90–101, mai 2007.

[2] HEEN (O.), GENET (T.), GELLER (S.) et PRIGENT (N.), « *An industrial and academic joint experiment on automated verification of a security protocol* », in *Mobile and Wireless Networks Security*, pages 39–53, 2008.

[3] BLANCHET (B.), « *Proverif: Cryptographic protocol verifier in the formal model* », **www.proverif.ens.fr/**.

[4] AVISPA Team, *HLPSL tutorial*, **www.avispa-project.org/package/tutorial.pdf**.

[5] DOLEV (D.) et YAO (A.), « *On the security of public key protocols* », in *Proc. IEEE Transactions on Information Theory*, pages 198–208, 1983.

[6] CORTIER (V.) AND WARINSCHI (B.), « *Computationally Sound, Automated Proofs for Security Protocols* », in Proc. of ESOP'05, volume 3444 of LNCS, pages 157–171, Springer, 2005.

[7] PATARIN (J.), « La cryptographie des cartes bancaires », dossier *Pour la science*, 36, juillet 2002.

[8] VAN HERREWEGHEN (E.) et WILLE (U.), « *Risks and potentials of using EMV for internet payments* », in Proc. of USENIX Workshop WOST'99, ACM, 1999.

[9] TURUANI (M.), *Security of Cryptographic Protocols: Decidability and Complexity*, *PhD thesis*, Université of Nancy 1, 2003.

[10] CORTIER (V.), « Vérifier les protocoles cryptographiques », *Techniques et Sciences Informatique*, 24(1):115–140, 2005.

[11] « DDA : Les cartes bancaires de 3ième génération », **www.cartes-bancaires.com/spip.php?article32**

[12] GOUBAULT-LARRECQ (J.)et PARRENNES (F.), « *Cryptographic protocol analysis on real C code* », in Proc. of VMCAI'05, volume 3385 of LNCS, pages 363–379. Springer, 2005.

Cédric Llorens & Denis Valois – cedric.llorens@wanadoo.fr – denis.valois@laposte.net

PROTÉGER DES SERVICES PAR TOPOLOGIE SUR UN CŒUR DE RÉSEAU MPLS

mots clés : ***réseau / sécurité / MPLS / pseudo-wire / VPLS / VRF***

Nous présentons dans cet article quelques techniques de construction de topologie réseau de niveau 2 et 3, sur un cœur de réseau MPLS, permettant d'assurer une protection par isolation.

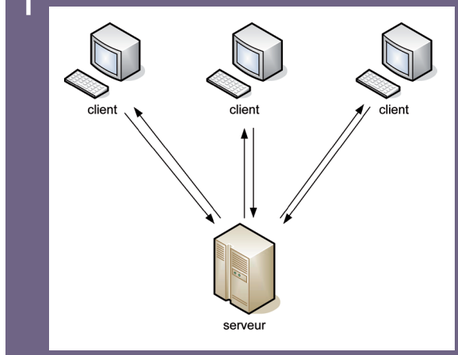
1. La problématique de la sécurité des services

Par le passé, la séparation entre les couches 2 et 3 du modèle OSI a forcé les opérateurs à construire et opérer des infrastructures séparées pour offrir différents services réseau. De nos jours, une infrastructure unique et partagée permet généralement d'offrir des services d'accès à ces mêmes couches comme l'illustre le tableau suivant :

3	Réseau	ex. IP (IPv4 ou IPv6)
2	Liaison	ex. Ethernet, HDLC, Frame Relay, ATM
1	Physique	ex. techniques de codage du signal (électronique, radio, laser, etc.) pour la transmission des informations sur les réseaux physiques

Reposant sur des techniques de *tunneling* ou d'émulation de lien point à point sur des cœurs de réseau, des topologies réseau sont construites tout en étant non visible et non atteignable directement par une couche de niveau 3 extérieure (telle que le réseau Internet).

Nous décrivons dans cet article trois techniques fondées sur l'isolation permettant de construire de telles topologies afin d'assurer un premier niveau de sécurité. Nous construirons plus précisément des topologies de type client-serveur dans lesquelles un client ne voit que le serveur et pas les autres clients comme l'illustre la figure 1.



Topologie client(s)-serveur(s)