FlexTLS : des prototypes à l'exploitation de vulnérabilités dans TLS

Benjamin Beurdouche [1] et Jean-Karim Zinzindohoué [2,1]
benjamin.beurdouche@inria.fr
jean-karim.zinzindohoue@inria.fr

[1] Inria Paris-Rocquencourt
[2] École des Ponts ParisTech

Résumé *Transport Layer Security (TLS)* est l'un des protocoles cryptographiques les plus utilisés pour protéger les communications sur internet. Cependant au vu des nombreuses attaques publiées, les tests réalisés sur les différentes implémentations se révèlent être insuffisants. Dans cet article, nous présentons FlexTLS, une librairie de source ouverte, permettant de créer rapidement des prototypes de nouvelles fonctionnalités du protocole et de tester le handshake. FlexTLS permet également de découvrir et d'exploiter des vulnérabilités en scriptant des scénarios concis reposant sur les primitives cryptographiques formellement vérifiées de miTLS.

1 Introduction

Il existe de nombreux protocoles de sécurité destinés à communiquer via internet. L'un des principaux, Transport Layer Security (TLS) [4], a pour objectif de sécuriser l'échange de données entre deux parties. Très largement utilisé, le protocole de chiffrement HTTPS [5] repose sur ce dernier. TLS est donc une cible privilégiée pour un adversaire souhaitant accéder à des données confidentielles. Nous introduisons ici FlexTLS, une librairie développée en F# et basée sur les primitives cryptographiques de miTLS [2], une implémentation de référence, formellement vérifiée, de TLS. FlexTLS a une API propre lui permettant d'implémenter de manière flexible et concise des scénarios de *handshake* sans réutiliser le *state-machine* de miTLS.

La spécification de TLS est volumineuse et en constante évolution. Dans le cadre de sa mise à jour vers la version 1.3 [6], le *TLS Working Group* de l'IETF est en train de procéder à des changements radicaux sur certains aspects du protocole : par exemple, la suppression du mécanisme d'échange de clés RSA ou encore le chiffrement opportuniste du *handshake*. Cela implique des modifications parfois profondes dans les librairies existantes et tend à introduire de nouveaux problèmes dans le protocole et ses

implémentations. Du fait de la complexité du phénomène, ces erreurs ne sont pas nécessairement détectées par les tests de fonctionnement usuels.

2 Création rapide de prototypes et implémentation robuste de nouvelles fonctionnalités dans TLS

Certains aspects problématiques de la spécification ne sont détectés qu'au moment d'écrire le code. Or, du fait de l'instabilité actuelle de celle-ci, aucun outil ne permettait, à notre connaissance, de réaliser une évaluation de la sécurité, de l'interopérabilité et du comportement fonctionnel des implémentations de TLS et à fortiori, pas dans de courts délais. Par ailleurs, les fonctionnalités encore en cours de maturation sont rarement implémentées du fait de l'investissement en temps à consentir.

Il est donc nécessaire pour les industriels et les académiques d'avoir accès à un outil permettant de répondre aux problématiques suivantes :

— Vérifier l'absence d'écarts anormaux entre la spécification et les implémentations en élaborant un prototype de ces nouveautés
— Établir le bon fonctionnement et l'impact de ces nouvelles fonctionnalités sur le code déjà existant
— Déterminer le nouvel état des propriétés de sécurité associées à ces changements à l'aide de la vérification formelle

L'objectif de FlexTLS est de répondre à l'ensemble de ces besoins.

Afin de garantir une meilleure cohérence entre la spécification du protocole et ses implémentations et d'accroître le niveau de sécurité de celles-ci, nous pouvons désormais accompagner l'émergence d'un nouveau concept quasiment en temps réel grâce au prototypage par FlexTLS.

Scenario	n° de msgs	n° de lignes de code
TLS 1.2 RSA	9	18
TLS 1.2 DHE w/ Client Auth	13	23
TLS 1.3 1-RTT	10	24
CCS Injection	17	29
Triple Handshake	14	44
FREAK Attack	12	22
ClientHello Fragmentation	3	8
Alert Attack	3	7

Table 1. Bilan des appels aux fonctions de FlexTLS par scénarios

D'autres implémentations existent et auraient pu être utilisées comme base pour FlexTLS ; cependant, notre expérience à mis en évidence le fait que les librairies écrites dans un style impératif et non modulaire sont mal adaptées aux usages décrits dans cet article. La puissance de notre outil réside dans sa capacité à exploiter les fonctions internes à miTLS pour rédiger des scénarios. Ceux-ci décrivent en quelques lignes un échange de messages de type *handshake* qui peuvent être, suivant les besoins, bien ou mal formés. FlexTLS se pose alors au choix comme un client ou comme un serveur afin d'interagir avec une implémentation contre laquelle il déroule son scénario. L'analyse des résultats (messages acceptés, réception d'une alerte, coupure TCP, etc.) retournés par FlexTLS permet alors à l'utilisateur de vérifier si l'implémentation visée est correcte vis-à-vis du standard et ainsi d'exploiter et de signaler les écarts éventuels.

Ci-dessous un scénario de TLS 1.3 écrit avec FlexTLS :

```
let tls13Client (address:string, cn:string, port:int) : state =
  let cfg = { defaultConfig with maxVer = TLS_1p3;
              negotiableDHGroups = DHE2432; DHE3072; DHE4096] } in
  let ch = { FlexConstants.nullFClientHello with pv = cfg.maxVer;
              suites = [TLS_DHE_RSA_WITH_AES_128_GCM_SHA256] } in

  (* Start a TCP connection to the server *)
  let st,_ = FlexConnection.
    clientOpenTcpConnection(address,cn,port,cfg.maxVer) in

  (* Start the handshake flow *)
  let st,nsc,ch   = FlexClientHello.send(st,ch,cfg) in
  let st,nsc,cks  = FlexClientKeyShare.send(st,nsc) in
  let st,nsc,sh   = FlexServerHello.receive(st,ch,nsc) in
  let st,nsc,sks  = FlexServerKeyShare.receive(st,nsc) in

  (* Switch to the next security context *)
  let st           = FlexState.installReadKeys st nsc in
  let st,nsc,scert = FlexCertificate.receive(st,Client,nsc) in

  (* Compute the log up to here *)
  let log = ch.payload @| cks.payload @| sh.payload @| sks.payload @
      | scert.payload in
  let st,scertv = FlexCertificateVerify.receive(st,nsc,sigAlgs_ALL,
      log=log) in
  let log = log @| scertv.payload in
  let st,sf = FlexFinished.receive(st,logRoleNSC=(log,Server,nsc))
      in

  (* Switch to the next security context *)
  let st    = FlexState.installWriteKeys st nsc in

  (* Update the log, and send the Finished message *)
  let log   = log @| sf.payload in
  let st,cf = FlexFinished.send(st,logRoleNSC=(log,Client,nsc)) in
  st
```

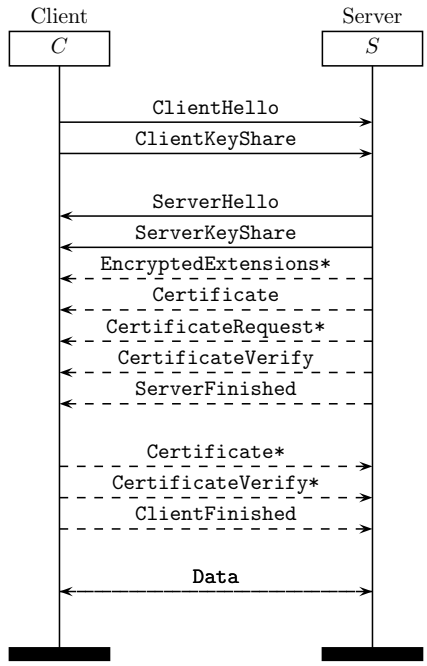

Figure 1. Spécification de TLS 1.3 *draft 5*

La rédaction de ce scénario ainsi que des fonctions nécessaires au *parsing* et à l'exploitation des nouveaux formats de messages dans miTLS représente environ deux heures de développement. À titre de comparaison, dans NSS, il aura fallu plus de 20 heures pour une machine à états complète. Ces deux heures sont tout le temps nécessaire au prototypage d'une nouvelle fonctionnalité du protocole, et permettront par la suite d'évaluer différentes implémentations. Par ailleurs, la facilité de prototypage permet de déceler très tôt certaines erreurs de conception mises en lumière par les contraintes du développement. FlexTLS étant multi-rôles nous pouvons également exploiter des vulnérabilités.

```
let st,nsc,ch = FlexClientHello.send(st,ch,cfg,fp=All(5)) in
(* finish the handshake as usual ... *)
```

Cette ligne montre la facilité avec laquelle un utilisateur de FlexTLS peut créer un scénario et implémenter la preuve de concept d'une vulnérabilité. Ici, FlexTLS applique une fragmentation très simple en blocs de cinq octets. La fragmentation n'étant pas authentifiée dans TLS, un adversaire peut, sans violer l'intégrité du message, forcer la connexion à négocier

TLS 1.0 dans les versions antérieures à OpenSSL 1.0.1h du fait d'une erreur de *parsing* dans la librairie. Implémenter cette attaque est simple mais certains scénarios sont plus complexes (cf. Table 1). Ainsi, la seule implémentation complète de l'attaque « Triple Handshake » [3] connue de ses auteurs utilise FlexTLS, seul outil permettant la rédaction d'un code clair et concis ainsi que la manipulation des clés cryptographiques ou des états internes au protocole, et ce pour plusieurs connexions.

3 L'avenir de FlexTLS

Plusieurs évolutions concernant le développement de notre outil sont en cours. La robustesse des tests réalisés à l'aide de FlexTLS repose entre autres choses sur l'utilisation de miTLS, formellement vérifiée. Nous étudions donc actuellement la possibilité d'étendre FlexTLS en tirant d'avantage parti des propriétés de sécurité vérifiées par miTLS. C'est pourquoi, nous travaillons avec nos collègues sur la transposition de miTLS et FlexTLS vers le langage FStar [7]. Ceci nous permettra de bénéficier d'un nouvel outil afin d'exprimer directement dans FlexTLS les pré- et post-conditions nécessaires à la vérification du code. Par ce biais, il sera prochainement possible d'évaluer rapidement la sécurité, au niveau formel, de certaines fonctionnalités en cours d'élaboration à l'IETF. FStar permettra par ailleurs d'obtenir une traduction certifiée du code vers d'autres langages comme OCaml ou JavaScript pour une meilleure portabilité et ce tout en conservant la validité des spécifications formelles.

Ne sont pas traités dans cet article les aspects d'automatisation des tests ni la génération heuristique de nouveaux scénarios pour la découverte automatique et l'exploitation de vulnérabilités. Un outil tel que FlexTLS permet cependant de s'assurer qu'une implémentation ne contient pas un certain nombre de failles et qu'elle se plie au standard (ex. Alertes dans TLS). Des discussions sont en cours avec plusieurs industriels afin de mettre en place FlexTLS dans les chaînes d'intégration continue des navigateurs web par exemple. Les écueils rencontrés permettent à la fois de corriger les implémentations en cours de développement, et de cerner les difficultés liées à leurs évolutions futures. A cette fin, il est déjà possible de créer une batterie de scénarios de tests à exécuter afin de rechercher d'éventuelles nouvelles vulnérabilités. Enfin, FlexTLS permet de tester les implémentations à mesure que de nouvelles failles sont découvertes [1] ainsi que leur résilience envers la réapparition d'anciens bugs.

4 Conclusions

Nous constatons régulièrement que les implémentations de TLS, dont la sécurité est pourtant cruciale, sont d'une qualité inégale. Elles ne sont, souvent, pas auditées correctement et testées uniquement à un niveau purement fonctionnel. Les méthodes de test actuellement employées ne sont pas adaptées et ne permettent pas de déceler certaines failles liées à la mise en œuvre du protocole dans son intégralité.

FlexTLS nous permet de contribuer à la sécurité du protocole et des librairies, en implémentant et testant de nouvelles fonctionnalités dès leur conception tout en surveillant que les bugs précédemment recensés ne réapparaissent pas. A l'heure actuelle, FlexTLS a déjà permis de communiquer de manière précoce à l'IETF certaines problématiques telles que la gestion concurrente des pairs de clés asymétriques par le client ou les risques d'attaques des serveurs par déni-de-service. Avec la création rapide de scripts démontrant des vulnérabilités telles « FREAK », nos collègues et nous [1] avons pu constater l'efficacité de FlexTLS à effectuer un large panel des tâches nécessaires à l'amélioration de la sécurité de TLS. Nous pensons que cette méthodologie devrait être étendue à l'ensemble des protocoles cryptographiques (ex. SSH, IPsec...) et à leurs implémentations afin de prémunir les utilisateurs contre des classes d'attaques qui, dès lors, pourront être évitées facilement.

Références

1. Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub et Jean Karim Zinzindohoue. A Messy State of the Union : Taming the Composite State Machines of TLS. *IEEE Symposium on Security & Privacy*, 2015. To appear.
2. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti et Pierre-Yves Strub. Implementing tls with verified cryptographic security. *IEEE Symposium on Security & Privacy*, pp. 445–462, 2013.
3. Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti et Pierre Yves Strub. Triple handshakes and cookie cutters : Breaking and fixing authentication over TLS. *IEEE Symposium on Security & Privacy*, pp. 98–113. 2014.
4. T. Dierks et E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.
5. E. Rescorla. HTTP over TLS. RFC 2818 (Informational), 2000.
6. E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2015.
7. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, et Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4) :402–451, 2013.