Rapport de projet

BERTHO Maxime (bertho.max@gmail.com)
EDGINTON-KING Leyton (leyton.edginton-king@etudiant.univ-rennes1.fr)
FOUCAULT Antoine (antoine.fouc@gmail.com)

Université de Rennes 1

March 1, 2016

Superviseur : **ALESSIO** Davide (Amossys)

Contents

Introduction

Ce rapport vise a présenter le travail que notre groupe a réalisé suite à l'étude que nous avons mené dans notre module de veille technologique. La première partie de ce rapport décrit les objectifs que nous nous sommes fixés pour la réalisation de ce projet. Dans la deuxième partie du rapport nous présenterons le travail que nous avons réalisé. Enfin, la troisième et dernière partie de ce rapport abordera la critique et les perspectives futures de notre travail.

1 Objectifs visés

Après les soucis causés par le fait que TOR messenger soit sorti en bêta durant notre projet et propose le support du protocole OTR dans le noyau ChatCore de Mozilla, nous avons dû redéfinir notre projet. Les implémentations de OTR étant alors présentes pour la plupart des moyens de communication classiques (XMPP, IRC, MSN, plusieurs messageries web, TOR messenger pour Mozilla...) nous avons alors fait le choix de nous intéresser au protocole lui même afin de le modifier et nous l'approprier sous deux grands axes :

- Modernisation et ajout de généricité pour les primitives cryptographiques
- Création d'un chat modulaire basé sur cette nouvelle version

1.1 Modernisation

Le protocole a été présenté pour la première fois en 2004 et sa version 3 sortie en 2012 souffre maintenant de quelques faiblesses et vetustées (Utilisation de l'algorithme de hashage SHA-1, taille de clés importante...). Le fait de moderniser les primitives du protocole nous as alors permis à la fois d'apporter un coup de neuf à celui-ci mais surtout de nous familiariser avec les nouveaux algorithmes et nouvelles primitives de chiffrement/signature. De cette façon nous avons revu l'implémentation du protocole tel qu'il a été spécifié et avons évité de réaliser une énième implémentation de celui-ci.

1.2 Modularité des primitives cryptographiques

La modernisation du protocole passe aussi par la modularité de celui-ci. Permettre de pouvoir facilement modifier ses primitives cryptographiques nous permet via le cloisonnement du code d'avoir une librairie facilement modernisable et configurable pour diverses utilisations. De cette façon le code peut-être retro-compatible avec la version classique de OTR. La modularité de notre implémentation fait qu'elle pourra également être réutilisée et peut-être suivre les évolutions futures que pourraient subir le protocole.

2 Implémentation

Notre projet a abouti à la création d'une preuve de concept correspondant à un chat peer-to-peer se basant sur une version modernisée d'OTR afin permettre aux interlocuteurs de communiquer de manière confidentielle et authentifiée.

2.1 Choix du langage de programmation

La librairie officielle du protocole OTR, libOTR, est codée en C++. Cependant, programmer en C++ nécessite une grande rigueur concernant la gestion de la mémoire. Afin de nous affranchir de cette tâche présentant des risques de sécurité évidents, nous avons choisi de programmer notre chat en Python. Ce choix réside également dans le fait que chaque membre du groupe connaissait déjà ce language. Programmer en Python a donc permis de développer plus rapidement tout en étant davantage confiant en notre code.

2.2 Pure Python OTR

Afin d'améliorer le protocole, nous avons cherché une implémentation de celui-ci en Python. Il s'avère que la seule implémentation disponible à cette date sur internet est pure-python-otr.[1] Cette librairie implémente le protocole jusqu'à la version 2. Bien qu'on pourrait imaginer qu'il est dérangeant de ne pas avoir accès à la dernière version du protocole pour apporter nos améliorations, cela ne pose en réalité pas de problème car la version 3 d'OTR n'apporte que des améliorations externes au coeur du protocole. Son fonctionnement reste donc identique à la version 2 sur les fonctionnalités de base.

[1] `https://github.com/python-otr/pure-python-otr`

2.3 DH vers ECDH

OTR fait usage de l'échange de clé Diffie-Hellman dans l'AKE ainsi que pour la rotation des clés. Dans l'objectif de moderniser le protocole nous avons remplacé cet échange de clé par ECDH qui se base sur les courbes elliptiques. Prenons Alice et Bob pour expliquer cet échange de clé :

Les deux interlocuteurs commencent par générer chacuns une clé secrète k (kA pour Alice et kB pour Bob) qui correspond à un nombre aléatoire modulo l'ordre de la courbe choisie pour l'échange.

Les interlocuteurs génèrent ensuite leurs clé publique $pubX$ en faisant le produit scalaire de kX par le générateur de la courbe G. Ils s'échangent ensuite ces clés publiques.

L'étape suivante consiste à calculer P, le point partagé par les deux interlocuteurs en faisant le produit scalaire de $pubX$ par kY (avec $(X = B, Y = A)$ pour Alice et inversement pour Bob).

Afin d'obtenir le secret partagé S correspondant à un nombre dans notre implémentation, nous calculons un hash du point P de cette manière :

$$S = int(HASH(hexstring(P.x) + hexstring(P.y))$$

Choix des courbes elliptiques

Le choix de la courbe elliptique utilisée pour le chiffrement est important car tous les courbes n'offrent pas la même niveau de securité. Par exemple, la courbe NIST P-256 utilise une forme de courbe et des parametres choisis pour améliorer l'efficacité calculatoire, mais ces choix nuisent à sa securité[2]. L'utilisation de certaines techniques algorithmiques peuvent également donner des courbes vulnérables à des attaques telle que la courbe secp256k1 avec Montgomery Ladder, qui est vulnérable à une attaque par fautes qui révèle l'exposant privé[1].

Malgré l'éxistance de ce style d'attaque contre la courbe secp256k1, nous l'avons utilisé pour faire notre ECDH car nous avons trouvé des vecteurs de test qui nous ont permis de vérifier notre implémentation. Étant donné la modularité de notre code il ne sera cependant pas difficile de remplacer cette courbe par une autre moins vulnérable par la suite.

Pour implémenter ECDSA nous avons choisi la courbe NIST P-384 qui a quelques faiblesses théoriques mais sur laquelle il n'existe pas encore d'attaque connue. Cependant, cette courbe fait partie d'une suite de courbes conseillées par la NSA, ce qui peut mettre en doute sa securité ainsi que sa robustesse.

2.4 DSA vers ECDSA

Le protocole OTR fait usage de l'algorithme de signature à clé publique DSA lors de l'échange de clé authentifié (AKE). Bien que DSA ne soit pas déprécié, nous avons choisi d'utiliser la variante ECDSA basée sur les courbes elliptiques afin de rajeunir le protocole. En effet, cette variante est intéressante sur deux points :

- Elle demande moins de calculs pour générer une signature
- Elle génère des signatures plus courtes ce qui permet de limiter le traffic réseau

L'implémentation d'ECDSA dans le projet a été réalisée via l'import du module python-ecdsa [2] qui est une implémentation complètement codée en Python. Ce module est testé de manière sérieuse notamment via la comparaison des signature du module avec celle fournies par OpenSSL.

2.5 SHA-1 vers SHA-3

Le protocol OTR de base utilise SHA-1 comme fonction de hashage. Les révélations de "The SHAppening" [5] ont montré qu'il était possible de trouver des collisions pour la fonction de compression de SHA-1. Par conséquent SHA-1 ne peux plus être considérée comme sécurisée.

Les acteurs du numérique l'ont bien compris : Microsoft a acceleré le depreciation de la signature du code en utilisent SHA-1, passant d'un retrait en 2017 à un retrait en Janvier 2016[3]. En ce qui concerne le web, Microsoft, Google, et Mozilla se sont mis d'accord sur l'arrêt le 1 Janvier 2016 de l'apparition de nouveaux certificats SSL utilisant SHA-1, et de ne plus faire confiance aux certificats utilisant SHA-1 à partir du 1er Janvier 2017[4].

Au vu des risques de securité concernant SHA-1, nous avons fait le choix de remplacer SHA-1 par SHA3. SHA3 est depuis le 5 Août 2015 standardisée par le NIST, non pas pour remplacer le SHA-2 (qui ne présente pas encore de risques de

[2] `https://github.com/warner/python-ecdsa`

securité), mais en tant qu'alternative. SHA3 peut calculer des hashs de longueur allant de 224 bits jusqu'à 512 bits. Nous avons choisi de prendre SHA3-256, mais vu la modularité de notre code il sera facilement possible de passer à SHA3-512 dans le futur.
Pour l'implementation de SHA3, nous avons utilisé la librarie python-sha3[3].

3 Usage

Pour faire fonctionner notre programme il est nécessaire d'installer les modules "python-ecdsa", "python-sha3" et "pure-python-otr-v4" avant tout lancement. Notre repository contient des fichiers "INSTALL" indiquant à l'utilisateur comment procéder pour réaliser ces installations.

Notre programme fonctionne selon le principe client/serveur. Pour démarrer un chat il est donc nécessaire que l'un des utilisateur se mette en écoute tandis que l'autre initie la connexion.

Voici la commande pour lancer le programme en mode serveur :

python ./chat.py -l PORT

Voici la commande pour lancer le programme en mode client :

python ./chat.py -c IP PORT

[3] `https://github.com/bjornedstrom/python-sha3`

4 Difficultés

Lors de ce projet nous avons été confronté à plusieurs obstacles. Tout d'abord de part le fait que les primitives cryptographiques modernes sont souvent peu éprouvées de part leur minorité dans le paysage informatique actuel. Il est donc difficile de faire des choix en ce qui concerne les technologies à utiliser pour moderniser le protocole sans pour autant risquer de l'affaiblir.

Par exemple, les vecteurs de tests pour HMAC-SHA3 n'étant pas encore disponibles (Début janvier W.Ehrhardt publie ses vecteurs de tests [4] qui ne sont pas vérifiés par plusieurs implémentations), nous avons donc choisi de ne pas implémenter cette primitive pour le moment. Cette pratique était également déconseillée dans la documentation de la librairie pysha3 [5].

La librairie Pure Python OTR s'est rapidement imposé à nous car elle était la seule implémentation qui correspondait à ce que l'on souhaitait moderniser. Soit une librairie OTR en python que l'on pourrait nous approprier mais celle-ci ayant été quelque peu délaissée ces derniers temps ne contient ni documentation, ni commentaires, ni tests. Ce qui nous a poussé à faire un travail en amont de "reverse-engineering" et d'appropriation du code afin de pouvoir la réutiliser.

5 Perspectives

Notre projet s'étant finalement plus cantonné à une preuve de concept et un travail sur les primitives cryptographiques d'un protocole sécurisé qu'est OTR, il est donc difficile d'imaginer de réelles perspectives de futur quant à celui-ci, notre modernisation du protocole ne respectant pas les spécification faites d'OTR, mais des perspectives d'évolutions sont possibles.

[4] `http://wolfgang-ehrhardt.de/hmac-sha3-testvectors.html`
[5] `https://pypi.python.org/pypi/pysha3/`

5.1 Documentation

Tout d'abord la documentation nous ayant fait défaut en ce qui concerne la librairie pure python OTR, nous pourrions nous intéresser à documenter et commenter le reste de la librairie et pas seulement la partie que nous avons ajouté ou modifiée. De cette façon, même si notre version ne correspond pas aux spécification du protocole OTR, celle-ci est configurable pour l'être et cette documentation pourrait donc servir aux personnes qui souhaiteraient reprendre ou réutiliser cette librairie qui n'est plus tenue à jour.

5.2 OTRv3

Notre projet s'étant fondé sur la librairie pure python OTR qui est resté au stade de la version 2 du protocole, notre projet est donc une dérivation de cette version. La version 2 et la version 3 ne présentant pas de différences en ce qui concerne la cryptographie mais plus en ce qui concerne le multi-session. Cette fonctionnalité n'étant pas réellement nécessaire pour notre projet nous n'avons donc pas passé la librairie en version 3 modifiée. Mais dans l'idée d'une poursuite du projet c'est une direction qui serait prise, afin, dans le cas d'une configuration avec les primitives classiques d'être compatible avec les autres clients OTR.

5.3 HMAC-SHA3

Etant donné qu'il n'existe pas encore de vecteurs de test ni d'implementation ou bibliotheque verifiée pour HMAC-SHA3, nous avons preferé ne pas utiliser SHA3 pour les HMACs. Cependant, dès qu'il y a un implementation ou une bibliothèque de reference, il serait facile de l'incorporer grace à la modularité que nous avons introduit au code.

5.4 XMPP

De part le fait que notre librairie est un bloc autonome, il pourrait être intéressant de tenter une intégration dans un outil ou plugin déjà existant afin de pouvoir tester son utilisation à un plus haut niveau que le chat très basique que nous avons développé. Par exemple la librairie python pure OTR dont nous sommes parti pour notre implémentation a été utilisée[6] une application de communication via le protocole XMPP. Notre librairie OTR modernisée pourrait alors remplacer celle existante.

[6] `https://github.com/mikegogulski/python-otrxmppchannel`

5.5 Courbes elliptiques

Comme nous avons evoqué lors de la partie 2.3, les courbes que nous avions choisi ne sont pas 100% securisées. Afin d'obtenir une securité optimale, nous pourrions remplacer les courbes utilisées dans les primtives cryptographiques ECDH et ECDSA par des courbes considerées comme 100% securisées.

6 Deroulement du projet

6.1 Gestion des sources

Pour la gestion de notre code source, nous avons decidé d'utiliser GitHub. Ce choix nous a permis de se passer d'avoir notre propre serveur git et nous a facilité le partage de code ainsi que la mise en commun de notre travail.
Nos sources sont disponibles sur GitHub : `github.com/El-gitano/otr_new`

6.2 Test du code

Afin de vérifier le fonctionnement de nos implémentations nous avons systématiquement testé celles-ci. Cette partie vise à décrire comment chaque primitive cryptographique a été implémentée.

6.2.1 SHA-3

Pour tester le module que nous avons utilisé pour importer la fonction SHA-3 nous avons récupéré les vecteurs de test fournis par le NIST [7]. En exécutant les tests nous nous sommes aperçu que notre implémentation ne fournissait pas les résultats attendus. En effet celle-ci se basait sur le "brouillon" de la spécification SHA3. Nous avons donc changé le module Python pour un autre qui respectait les spécifications officielles. Les tests sont de deux sortes :

- Des tests avec des entrées de faible longueur (maximum 137 octets)
- Des tests avec des entrées de grande longueur (minimum 411 octets)

De plus, un test vérifie que la fonction générique HASH utilisé dans notre version de pure-python-otr correspond bien à SHA3.

[7] `http://csrc.nist.gov/groups/STM/cavp/secure-hashing.html`

6.2.2 Courbe secp256k1

La courbe secp256k1 est utilisée dans notre projet afin de réaliser des échanges de clé ECDH. L'implémentation de cette courbe est tirée d'un code disponible sur GitHub Afin de vérifier le fonctionnement de cette implémentation nous avons utilisé des vecteurs de tests provenant de deux sources :

- Le développeur de hxBitcoin [8]
- Un développeur qui a publié ses vecteurs sur un forum

Cette série de tests consiste à effectuer la multiplication scalaire d'un nombre avec le point de génération de la courbe puis de comparer les coordonnées du point résultant avec celles de référence.

Dans un second test nous avons vérifié que la multiplication du point de génération de la courbe avec un nombre correspondant à l'ordre de la courbe donne bien en résultat l'infini.

La dernière série de tests consiste à répéter 100 fois les instructions suivantes :

- Générer a et b entiers aléatoires inférieur à l'ordre de la courbe et $c = a + b$
- Calculer p et q et c tel que $p = aG$, $q = bG$ avec G le générateur de la courbe
- Vérifier que $p + q == cG$

6.2.3 HMAC-SHA512

Les tests de HMAC-SHA512 se sont basés sur les vecteurs de test du NIST. De plus, un test vérifie que la fonction générique HMAC de notre implémentation correspond bien à HMAC-SHA512.

[8] `http://hxbitcoin.com/`

Temps de travail

	Antoine	Maxime	Leyton	Total (h)
Temps d'étude (h)	12	13	13.2	38.2
Temps de programmation (h)	12	8	8	28
Temps de rédaction (h)	6	6.11	5.5	17.61
Temps de préparation pour la soutenance (h)	5	4.6	5	14.6
Total (h)	35	31.71	31.7	98.41

Conclusion

Notre projet qui était tout d'abord très orienté vers une utilisation concrète car répondant à un besoin réel (Bugbounty...) s'est finalement changé en un projet plus théorique sur l'étude d'un protocole sécurisé qu'est Off-The-Record, des primitives cryptographiques modernes ainsi que la ré-appropriation d'un protocole et le travail sur une librairie.

Ce projet ayant alors plus une ambition pédagogique et n'étant plus vraiment orienté vers la résolution d'un besoin est devenu moins attrayant qu'il ne pouvait l'être. Celà nous as cependant permis de nous intéresser de près à la fois à un protocole, mais aussi aux nouveautés en terme de primitives cryptographiques qui n'ont pas forcément pu être vue dans le cadre de nos cours.

References

[1] Pierre-Alain Fouque et al., *Fault Attack on Elliptic Curve with Montgomery Ladder Implementation*, FDTC '08, IEEE, 2008.

[2] Daniel J. Bernstein and Tanja Lange, *SafeCurves: choosing safe curves for elliptic-curve cryptography*, http://safecurves.cr.yp.to

[3] Windows Enforcement of Authenticode Code Signing and Timestamping, Microsoft, http://aka.ms/sha1

[4] Phasing Out Certificates with SHA-1 based Signature Algorithms Mozilla Security Blog, https://blog.mozilla.org/security/2014/09/23/phasing-out-certificates-with-sha-1-based-signature-algorithms/

[5] The SHAppening: freestart collisions for SHA-1, https://sites.google.com/site/itstheshappening/