



ANDRES ANDREU

Cracker les SSHA sous LDAP

Degré de difficulté



Dans le domaine des applications Web, les données utilisateur sont stockées de manière à être accessibles par les personnes qui y seront autorisées. Les données utilisateur contiennent les logins et mots de passe, ces derniers (et éventuellement les noms d'utilisateurs et d'autres attributs) sont stockés pour une utilisation future.

Pour se protéger contre une multitude d'attaques, ce qui inclut les attaques par injection de code malveillant, l'exposition de données archivées, de données utilisateur, mais aussi de mots de passe, ces derniers sont stockés dans une forme non réversible et dans un texte brouillé. Chose intéressante, cette façon de penser et cette technique de stockage a bénéficié de l'interface du bureau pour se loguer avec des logins liés à Active Directory et LDAP.

Stocker des données utilisateur telles que des mots de passe en texte brut représente un certain risque. En cas de piratage, les crackers qui ont accès à vos données via les failles d'un logiciel (comme une mauvaise validation des données entrées) pourraient avoir un accès non autorisé à divers systèmes. Ces jours-ci le risque est beaucoup plus élevé que par le passé en raison de l'évolution de l'Internet/Web qui se base sur des mots de passe uniques et des technologies à authentification unique (SSO). Ceci peut avoir pour conséquence une augmentation des activités malveillantes de n'importe quel utilisateur lambda, avec les permissions de celui-ci. Les conséquences de ces actions ne sont limitées que par votre imagination et les accès autorisés de l'application cible. Pour diminuer ce risque l'industrie s'est généralement basée sur un stockage des mots de passe résultant d'un algorithme de hachage. Malgré leur

sophistication, ces algorithmes ont perdu de leur efficacité. Le besoin d'avoir des données aléatoires, qui remonte aux techniques utilisées à l'époque d'Unix, devenait indispensable. Ces besoins ont trouvés une réponse grâce à des hashs de type salt permettant d'ajouter un caractère aléatoire à l'affichage des données. Ce caractère aléatoire rend la charge de travail importante pour le succès d'un crack.

Un hachage à sens unique est une valeur binaire calculée de longueur fixe qui est normalement représentée en notation hexadécimale ou en Base64.

L'idée derrière l'utilisation de hachages à sens unique est l'identification (validité) du texte brut (grâce à l'emploi de comparaisons). Certains experts considèrent cela comme une empreinte digitale de texte.

Un salt est une donnée aléatoire calculée pour modifier l'affichage d'un algorithme de hachage à sens unique (dans le cadre de cet article). Ces ensembles de données se présentent dans des blocs de 4 ou 8 octets. En ce qui concerne l'aspect aléatoire du salt, il a été critiqué maintes et maintes fois (véritablement aléatoire ?), cela dépasse de loin l'article. Il est dit en général que la plupart du code qui se base sur le web génère des salts aléatoires en utilisant des fonctions pseudo aléatoires.

Cet article n'est pas une attaque des fondements mathématiques sur l'aléatoire dans

CET ARTICLE EXPLIQUE...

Comment les salts de hachage SHA (SSHA) sont structurés sous LDAP.

Comment exploiter les outils actuels pour cracker les hashs SSHA sous LDAP.

Pourquoi les hashs SSHA sous LDAP devraient être traités comme des données en texte clair.

CE QU'IL FAUT SAVOIR...

Des connaissances de base dans la compilation de codes sources en C sous Linux (basé sur : x-86).

Une compréhension basique des scripts (du code et/ou extraits donnés en Python, Ruby & PHP).

Des connaissances de base en cryptage de données binaires.

Des connaissances sur les concepts de stockage de mots de passe utilisateur.

les environnements web, l'aspect aléatoire de la valeur du salt n'apporte rien de pertinent dans les exemples que nous traiterons ici.

La raison pour laquelle on utilise un salt en conjonction avec un hachage des données à sens unique devrait être une évidence.

Des technologies avancées comme PKI, les certifications X509 côté client, et les solutions biométriques sont de plus en plus utilisées, mais la réalité de l'industrie informatique est autre, celle-ci se base toujours sur la combinaison nom d'utilisateur/mot de passe qui est le mode d'authentification le plus répandu, en particulier dans le domaine des applications web et les technologies connexes telles que les VPN qui se basent sur SSL. Un acteur majeur du stockage de données et des services d'authentification/autorisation tel qu'il se fait actuellement est *Lightweight Directory Access Protocol* (LDAP).

LDAP est un protocole qui peut être supporté par différentes technologies, telles que XML ou les bases de données traditionnelles.

Dans la mesure du possible, les schémas utilisés par LDAP et les implémentations de certaines classes d'objets sont acceptés comme un standard dans l'industrie, et ils peuvent stocker des données sensibles telles que les mots de passes. La classe d'objet qui peut illustrer cela est *inetOrgPerson* décrit dans le RFC-2798 qui possède une propriété nommée *userPassword*. N'importe quel standard dont l'implémentation se base sur le protocole LDAP supportera la classe objet. Les ingénieurs logiciels peuvent écrire du code qui interagit avec ces propriétés et être confiants sur le bon fonctionnement du code pour un produit qui se base sur

des serveurs LDAP respectant certaines normes.

La propriété *userPassword* supporte les données de type mots de passe dans une des formes suivantes :

- CLEAR – littéralement c'est du texte brut,
- BASE64 – Représentation codée du texte brut en Base64,
- MD5 – utilise l'algorithme de hachage à sens unique Message Digest 5,
- SHA – utilise l'Algorithme de Hachage Sécurisé à sens unique.

Une donnée en texte brut est évidemment non sécurisée et en Base64 ce n'est pas mieux.

Chacune de ces méthodes n'apporte rien pour la protection des données utilisateur si un attaquant accède à la source de vos données LDAP. Les algorithmes de hachage à sens unique sont venus au secours des données stockées du fait qu'ils ne soient pas réversibles. Malheureusement ils opèrent de la même manière ainsi un attaquant pourrait voir que les utilisateurs dans votre source de données LDAP ont tous les mêmes mots de passes basés sur des hachages identiques. Le Listing 1 vous montre du code en Python qui générera un hachage SHA1 de texte brut. Vous pouvez exécuter ce script en Python autant de fois que vous le souhaitez sur votre texte brut et observer que l'affichage ne change pas. C'est un sujet de préoccupation, car de nombreuses entités utilisent un mot de passe par défaut pour l'ensemble des nouveaux et/ou utilisateurs temporaires dans leur infrastructure. Dans les opérations qui utilisent des hachages de type SHA1 vous trouverez du code qui effectue ce type d'actions et affiche les propriétés (en général *userPassword*).

Listing 1. Un simple code pour générer des hachages non salt (unsalted) de type SHA1 de données en texte brut

```
import sys, sha, base64
ctx = sha.new( sys.argv[1] )
hash = "\n{SHA}" + base64.b64encode( ctx.digest() )
print hash

En présumant que le code ci-dessus est sauvegardé dans un fichier nommé genSHA.py alors une exécution (en utilisant la chaîne de texte "test" en clair) ressemblerait à ceci :
$ python genSHA.py test
{SHA}qUqP5cyxm6YcTAhz05Hph5gvu9M=
```

Pour rendre les choses plus difficiles aux équipes de sécurité informatique, il existe des services en ligne qui cherchent à identifier une collision de données de hachage (de nombreuses instances de MD5 sont disponibles en ligne), des crackers par technique de force brute pour des hachages spécifiques, des cracks de type rainbow et d'énormes *rainbow tables* qui peuvent être mis à mal avec les technologies de type torrent. C'est assez inquiétant car il devient difficile de nos jours de protéger les données sensibles des utilisateurs.

La suggestion des experts en sécurité est d'utiliser un salt avec des algorithmes de hachage en sens unique. Dans le domaine de la recherche en sécurité les MD5 et SHA1 ont connu des collisions qui ont abouti avec succès. (Cf. Les liens fournis à la fin de cet article). Cela ne veut pas dire que personne ne puisse produire une telle situation mais cela demande un niveau de connaissances similaire à celui d'un expert et une grande maîtrise de l'informatique (même si les projets de collisions se basant sur BOINC réduisent les connaissances dont il faudrait disposer). Les algorithmes de hachage de type fort couramment utilisés aujourd'hui sont MD5, SHA et la famille d'algorithmes SHA2 en combinaison avec des salts aléatoires, ainsi dans des implémentations LDAP plus sophistiquées vous rencontrerez les identifiants suivants pour les données stockées dans les propriétés *userPassword* :

- SMD5 – MD5 salt,
- SSHA – SHA1 salt,
- SSHA256 – SHA256 salt,
- SSHA384 – SHA384 salt,
- SSHA512 – SHA512 salt.

Cette utilisation d'un salt signifie que pour chaque objet utilisateur un hachage unique est mis en oeuvre pour stocker les données de type mot de passe. Si deux utilisateurs ont le même mot de passe et qu'un salt est utilisé alors une analyse classique ou l'œil humain ne pourra pas faire la correspondance, étant donné que les hachages stockés ne correspondent pas visuellement.

Il est important qu'un salt soit produit avec le plus haut niveau possible d'entropie dans le but d'optimiser l'utilisation de cette technologie.

Le Listing 2 vous fournit un script en Python 2.5 qui génère des hachages salt de type SHA, ce script particulier rassemble l'ensemble des algorithmes de hachage de la famille SHA2.

Si vous exécutez de nombreuses fois ce script avec la même chaîne de texte vous obtiendrez toujours des

affichages uniques des hachages qui en résultent. Cet affichage émule ce qu'un environnement plus sophistiqué ferait si il était sous LDAP.

Le Salt est toujours disponible

Il n'y a pas de magie noire qui permette de réaliser ces techniques de hachage de type salt. La partie salt d'un hachage où l'on a effectuée un salt doit être disponible pour le code/le(s)

application(s) qui interagît avec pour qu'il puisse fonctionner correctement. Sinon il serait impossible pour une application de vérifier les données qui lui sont soumises.

Dans un scénario d'authentification lors de l'interaction avec des hashs de type salt, par exemple, une demande suivra généralement les étapes suivantes (celles-ci sont spécifiques aux normes basées sur les serveurs LDAP mais le concept peut également s'appliquer à d'autres scénarios) :

Listing 2. Un simple code en Python2.5 pour générer des hachages SHA1 et SHA2 de données en texte brut.

```
import hashlib, binascii, sys
from base64 import b64encode
from random import randrange
str = sys.argv[1]
saltsize = int(sys.argv[2])
if saltsize <> 4 and saltsize <> 8:
    print "Tenons nous en à ce qui se fait en général, 4 ou
          8 octets pour la taille d'un salt
          ...\\n\\n"
    sys.exit(0)
print "generating simple random salt of %d bytes...\\n" % saltsize
salt = ''
for n in range(saltsize/2):
    salt += chr(randrange(256))
salt = binascii.hexlify(salt)
print "SHA1"
m = hashlib.sha1()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA}" + w
print "Codé en Base64:\\n%s" % w
print "%s" % wo
print
print "SHA256"
m = hashlib.sha256()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA256}" + w
print "Codé en Base64:\\n%s" % w
print "%s" % wo
print
print "SHA384"
m = hashlib.sha384()
m.update(str)
m.update(salt)
h = m.digest()
print "In Hex:\\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA384}" + w
print "Codé en Base64:\\n%s" % w
print "%s" % wo
print
print "SHA512"
m = hashlib.sha512()
m.update(str)
m.update(salt)
```

```

h = m.digest()
print "In Hex:\n%s" % binascii.hexlify(h)
w = b64encode( h + salt )
wo = "{SSHA512}" + w
print "Codé en Base64:\n%s" % w
print "%s" % wo
print

En présumant que le code ci-dessus est sauvegardé dans un fichier
genSSHA.py alors une exécution sommaire (en utilisant la
chaîne de texte brut "test") ressemblerait à ceci :
$ python genSSHA.py test 4
Génère un salt aléatoire de 4 octets...
SHA1
En Hex :
98f161a269d8d3b5567749420f8024a27a9844c0
Codé en Base64 :
mPFhomnY07VWd01CD4AkongYRMBjNTg1
{SSHA}mPFhomnY07VWd01CD4AkongYRMBjNTg1
SHA256
En Hex :
a1efb0cc52ed95dca4536d6d21d68044ca742fec269326992f5d9279e7cccf48
Codé en Base64 :
oe+wzFLtldykU21tIdaARMp0L+wmkyazL12SeefMz0hjNTg1
{SSHA256}oe+wzFLtldykU21tIdaARMp0L+wmkyazL12SeefMz0hjNTg1
SHA384
En Hex :
4034f8cecd3b59e44810c113b88c7b04475193aeab6629034994b1c71e8213392
    bd5f07d25e1b2d42547150b7679618c
Codé en Base64 :
QDT4zt07WeRIEMETuIx7BEDRk66rZikDSZSxxx6CEzkrlfb9JeGy1CVHFQt2eWGM
    YzU4NQ===
{SSHA384}QDT4zt07WeRIEMETuIx7BEDRk66rZikDSZSxxx6CEzkrlfb9JeGy1CVH
    FQt2eWGMYzU4NQ==

SHA512
En Hex :
ff24c1b3cf119bf449478c1931a645d240b3454213531ee3fd1lebe2d24a15017c
    7aacdebdæ4d181b6d62696dc1fb20046684096bf2ae71bf1fd20409
    ca3dfb
Codé en Base64 :
/yTBS88Rm/RJR4wZMaZF0kCzRUITUx7j/R6+LSShUBfHqs3r2uTRgbWJpbcsfsgB
    GaECWvyrnG/H9IECc9+2M1ODU=
{SSHA512}/yTBS88Rm/RJR4wZMaZF0kCzRUITUx7j/R6+LSShUBfHqs3r2uTRgb
    bWJpbcsfsgBGaECWvyrnG/H9IECc9+2M1ODU=

On voit nettement une différence dans la taille de l'octet des
hashs générés et les plus importants représentent
un facteur de travail plus grand pour réussir le crack.
Bien qu'ils nécessitent un effort plus important, ils
peuvent néanmoins être crackés via des collisions tant que
le cracker/attaquant sait où se trouve le salt et peut
l'extraire.

```

- `getSaltedHash` (d'un stockage interne – LDAP, base de données, ...),
- `detectHashingAlgorithm` (analyser le hachage de type salt permet de déterminer l'algorithme à sens unique utilisé – dans LDAP il y a généralement un préfixe identifiant comme {SSHA} ;

- il y a aussi le fait que ces hachages à sens unique affichent des données dont la taille est statique),
- `extractSalt` (en connaissant l'algorithme, le code peut extraire le salt à partir du hachage qu'il vient de stocker),

Listing 3. Un petit script en ruby illustrant le processus des données utilisées par un algorithme de hachage à sens unique de type salt

```
#!/usr/bin/env ruby
# A des fins pédagogiques nous avons utilisé une chaîne de texte clair et un salt
require 'sha1'
require 'base64'
salt = 'SALT'
pass = 'testing'
conc = pass+salt
sha = Digest::SHA1.digest(conc)
puts "SHA1 Digest"
puts "In Binary: #{sha}"
puts "Length of Binary: #{sha.length}"
puts "\nIn Hex: #{sha.unpack('H*').to_s}"
puts "Length of Hex: #{(sha.unpack('H*')).to_s.length}"
puts "\nSalt\nIn ASCII: #{salt}"
puts "In Hex: #{salt.unpack('H*')}"
concsalt = sha+salt
puts "\nSHA1 Hash plus salt (RAW): #{concsalt}"
puts "SHA1 Hash plus salt (RAW - Length): #{concsalt.length}"
puts "SHA1 Hash plus salt (Hex): #{concsalt.unpack('H*')}"
puts "SHA1 Hash plus salt (Hex - Length): #{(concsalt.unpack('H*')).to_s.length}"
hash = "[SSHA]" + Base64.encode64(concsalt).chomp!
puts "\nSalted SHA1 Hash(Base64 Encoded): #{hash}"
Une exécution de ce script génère l'affichage suivant:
$ ruby genSSHA.rb
SHA1 Digest
In Binary: yP?`x&%u??V?Cf9M
Length of Binary: 20
In Hex: 790250aa1e6078262575a2c6991856ec4366394d
Length of Hex: 40
Salt
In ASCII: SALT
In Hex: 53414c54
SHA1 Hash plus salt (RAW): yP?`x&%u??V?Cf9MSALT
SHA1 Hash plus salt (RAW - Length): 24
SHA1 Hash plus salt (Hex): 790250aa1e6078262575a2c6991856ec4366394d53414c54
SHA1 Hash plus salt (Hex - Length): 48
Salted SHA1 Hash(Base64 Encoded): {SSHA}eQJQqh5geCY1daLGmRhW7ENmOU1TQUxU
```

Listing 4. Extrait de code de SSHA Attack décrivant le processus qui extrait le salt à partir d'un salt de hachage obtenu d'une implémentation LDAP

```
// prend le salt à partir de temp & cpy à tempSalt
if (strcmp(hashtype, "SHA1") == 0) {
    strcpy(tempSalt, temp + 20);
} else if (strcmp(hashtype, "SHA224") == 0) {
    strcpy(tempSalt, temp + 28);
} else if (strcmp(hashtype, "SHA256") == 0) {
    strcpy(tempSalt, temp + 32);
} else if (strcmp(hashtype, "SHA384") == 0) {
    strcpy(tempSalt, temp + 48);
} else if (strcmp(hashtype, "SHA512") == 0) {
    strcpy(tempSalt, temp + 64);
}
```

A la fin de cet extrait du code le tableau tempSalt contiendra la valeur du salt à partir du hash. Prenez note de la façon dont une bonne connaissance des tailles des hashs est utilisée pour calculer le démarrage de l'extraction du salt. Avec cette donnée, on peut démarrer le crack. Il est évident que ce salt générera des hashs en texte clair basés sur la méthode de crack que vous utiliserez.

- `getClearTextData` (ce sont les données entrées par l'utilisateur ou bien une application, dans une requête d'authentification c'est le mot de passe),
- `combineClearTextDataAndSalt` (combine les données entrées avec le salt que l'on vient de récupérer),
- `applyAlgorithm` (applique l'algorithme de hachage au résultat de l'étape précédente),
- `compareValues` (compare le hash de base des données stockées en interne avec le hash et le salt soumis dans le code).

Un extrait de code PHP qui réalise cela sur des données de type salt SHA1 pourrait ressembler à ceci :

```
$encrypted = substr($encrypted, 6);
// on obtient à partir du {SSHA}
// quelque chose comme ceci {SSHA}
// VyVW450scSQLHTj+3XIXIbewGLVmYWJ1

$hash = base64_decode($encrypted);
// $hash obtention donnée binaire

$salt = substr($hash, 20);
// extrait le salt de la donnée
// binaire

if ($hash == mHash(MHASH_SHA1,
    $cleartext . $salt)) {
    return true;
}
```

Même si cet article se focalise sur les hachages LDAP de type salt et sur leur utilisation courante dans l'industrie, les concepts décrits s'appliquent à toute technique similaire pour le stockage de ce type de données. Certaines solutions stockent toutes les données dans une

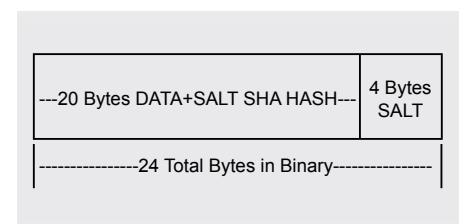


Figure 1. Structure Hash SSHA

– Représentation visuelle de la structure de données de hachage (SHA1 – Salt de 4 octets) détaillé dans l'affichage du code au Listing 3

table d'une base de données où un champ possède le hachage de type salt et un autre champ possède sa valeur.

Ce qu'il faut bien comprendre, c'est que le salt est quelque part et un attaquant essayera d'y accéder. Si le salt est compromis alors les techniques par force brute où les attaques par dictionnaire deviennent possibles comme vous allez le voir.

La structure des données de type Hash

Dans le cas des hachages LDAP de type salt la structure de donnée hachée finale ressemble à cela (encore une fois, ceci est spécifique aux hachages salt de type SHA1 avec un salt de 4 octets mais essayez d'avoir une vision d'ensemble).

La valeur d'un salt se présente sous forme binaire. Ce salt se compose de

données binaires aléatoires de 4 octets avec une notation hexadécimale (Base16 ou 8 octets). Le hash final de type salt est d'une longueur de 20 octets représenté en binaire (40 octets si vous le regardez en hexadécimal). L'algorithme SHA1 génère au final une chaîne de 160 bits. Avec 8 bits par octet cela équivaut à 20 octets. La Figure 1 devrait vous en donner une illustration simple et claire.

Lorsque vous êtes confrontés à ce type de données (ayant déjà été hachées) vous devez bien en comprendre la structure. Notre but est de désassembler cette donnée stockée afin d'obtenir le salt et des données stockées qui peuvent être utilisées pour la comparaison des hashs, ainsi le hash stocké doit être découpé. Dans le cas du SHA1 le but est de découper le hash d'origine en 2 tableaux d'octets distincts, l'un pour les 20 octets restants (0 – 20 y compris le `NULL`) et l'autre pour les autres données. Les 0 à 20 octets à gauche seront les valeurs binaires de type salt que nous utiliserons pour une comparaison octet par octet avec le nouveau texte en clair pour la vérification. La chaîne de texte à vérifier devra également être de type salt. Les octets restants (21 – 32) représenteront le salt aléatoire qui lors du déchiffrage montrera les caractères en hexadécimal qui composaient l'élément généré aléatoirement.

Regardez le script en Ruby du Listing 3, ça affiche des détails intéressants sur le processus de création de salt SHA1. Ça vous permet de bien comprendre ce qui se passe à l'intérieur du code qui génère ce type de hashs. En cas réel, le hash résultant que l'on peut voir en dernière ligne de l'affichage est ce qui serait stocké dans l'attribut de LDAP (dans le Listing 3 ce serait : `{SSHA}eQJQqh5geCYldalGmRhW7ENmoU1TQUxu`). La Figure 1 devrait renforcer votre compréhension par une illustration visuelle du hash binaire de type salt.

Cracker le Hash

En se basant sur ce que vous avez appris, vous verrez que tout le nécessaire pour casser un salt de hachage SHA sur LDAP est facilement disponible. Des outils ont été écrits pour accomplir cette

Listing 5. C Snippet from SSHA attacks generate hash function

```
...
EVP_MD_CTX_init(&mdctx);
EVP_DigestInit_ex(&mdctx, md, NULL); // Initialise digest
EVP_DigestUpdate(&mdctx, // Ajouter le mot de passe en texte clair à digest value,
    unsigned int strlen(value));
if(salt) {
    EVP_DigestUpdate(&mdctx, // Si on a un salt, ajouter également à digest salt,
        unsigned int strlen(value));
}
EVP_DigestFinal_ex(&mdctx, // Créer le hash
    md_value,
    &md_len);
EVP_MD_CTX_cleanup(&mdctx); // Nettoyer
for(i = 0; i < md_len; i++) {
    sprintf(&buffer[i*2], "%02x", md_value[i]);
// copier les valeurs hexadécimales dans le buffer
}
...

```

Listing 6. Déclaration d'utilisation de SSHA Attack

```
Usage: ./ssha_attack -m mode [-d attack_dictionary_file |
    [-n min] -u max -a alphabet | -a 20 -c custom_alphabet] -s SSHA_hash_string
-m C'est le mode sous lequel doit opérer le programme. Les modes actuellement pris
en charge sont "dictionary" et "brute-force". Ce switch est requis.
-d Cette option doit être utilisée pour le mode "dictionary". Le dictionnaire est un
fichier texte avec une entrée par ligne. Les données de ce fichier seront celles
utilisées comme les données en texte clair qui se verront appliquer le salt.
-l Nombre de caractères au minimum pour lancer l'attaque.
-u Nombre maximum de caractères à utiliser pour l'attaque. Si -l n'est pas utilisé le
traitement débutera avec une taille de 1
-a Index numérique (alphabet) à utiliser pour l'attaque:
1. Nombres uniquement
2. casse minuscule hex
3. CASSE MAJUSCULE HEX
4. casse minuscule caractères alphabétiques
5. CASSE MAJUSCULE caractères alphabétiques
6. casse minuscule caractères alphanumériques
7. CASSE MAJUSCULE CARACTERES ALPHANUMERIQUES
8. casse minuscule & CASSE MAJUSCULE caractères ALPHANUMERIQUES
9. casse minuscule & CASSE MAJUSCULE caractères ALPHANUMERIQUES
10. Ensemble des caractères ASCII que l'on peut imprimer
11. casse minuscule & CASSE MAJUSCULE caractères alphanumériques, et aussi:
    !"#$%&*()_+=[]{}'#@~,.<>?/
20. Alphabet personnalisé - doit être utilisé avec le switch -c
-c Alphabet personnalisé à utiliser, par exemple abcABC123!
Notez que cela oblige à lancer un processus de permutation ainsi, plus l'alphabet est
grand plus le processus est long. De même, lorsqu'il est combiné au switch -a 20,
mais pas le switch -u, les permutations se basent toutes sur la taille de l'alphabet
soumis. En utilisant l'exemple ci-dessus toutes les permutations auraient une longueur
de 10 caractères. Ceci peut impliquer une attaque incrémentale avec l'utilisation du
switch -n
-s La chaîne de hachage SHA qu'on attaquera. Ceci doit être une chaîne chiffrée en
Base64.
Ce switch est requis.
```

tâche. John the Ripper, ou John comme il est communément appelé, dispose d'un

Tableau 1. Pour la taille 1

A	b	c	d
---	---	---	---

Tableau 2. Pour la taille 2

Aa	ba	ca	da
Ab	bb	cb	db
Ac	bc	cc	dc
Ad	bd	cd	dd

Tableau 3. Pour la taille 3

Aaa	baa	caa	daa
Aba	bba	cba	dba
Aca	bca	cca	dca
Ada	bda	cda	dda
Aab	bab	cab	dab
...
Adc	bdc	cdc	ddc
Aad	bad	cad	dad
Abd	bbd	cbd	dbd
Acd	bcd	ccd	dcd
Add	bdd	cdd	ddd

Tableau 4. Pour la taille 4

Aaaa	baaa	caaa	daaa
Abaa	bbaa	cbaa	dbaa
Acaa	bcaa	ccaa	dcaa
Adaa	bdaa	cdaa	ddaa
Aaba	baba	caba	daba
...
Adcd	bcdc	cdcd	ddcd
Aadd	badd	cadd	dadd
Abdd	bbdd	cbdd	dbdd
Acdd	bcdd	ccdd	dcdd
Addd	bddd	cddd	dddd

Tableau 5. Utiliser la fonction d'alphabet personnalisé avec un alphabet abcd

Aaaa	Bbbb	cccc	dddd
Abcd	Abdc	acbd	acdb
Adcb	Adbc	bacd	badc
Bcad	Bcda	bdca	bdac
Cbad	Cbda	cabd	cadb
Cdab	Cdba	dbca	dbac
Dcba	Dcab	dacb	dabc

patch qui lui permet de cracker les salts de hachage SHA1. John est un utilitaire

de crack multi-tâches très puissant, mais il est quelque peu limité en raison du fait qu'il ne prenne pas en charge les algorithmes de la famille SHA2. Dans cet article on se focalise sur un type particulier de hachage qui se base sur la famille SHA (sauf pour le SHA224 car il ne semble pas répandu dans l'industrie) jusqu'à SHA512. SSHA Attack est un outil écrit pour faire cela, il supporte les attaques sur les salts de hachage : SHA1, SHA256, SHA384 & SHA512 qui sont communément utilisés dans des banques de données que l'on accède via LDAP.

SSHA Attack est écrit en C pour optimiser ses performances. Il utilise le concept d'authentification expliqué plus haut pour le crack d'un hash donné. Si vous y réfléchissez simplement, un crack de cette nature n'est autre que le même processus de la requête d'authentification une fois qu'on a les salts de hachage.

Cette technique ne s'attaque pas à l'algorithme de hachage, elle l'utilise seulement pour la comparaison des hachages, nous attaquons uniquement les données en sortie de ces algorithmes.

Technique mise à part, il est essentiel de comprendre la structure de données que l'on a expliquée auparavant. L'extraction du salt depuis les salts de hachage est le cœur de notre attaque et cela a un impact direct sur le crack d'un hachage. Analysez l'extrait de code dans le Listing 4, il vous montre où SSHA Attack extrait le salt à partir d'un salt de hachage SHA selon son type.

Avec le salt en main, SSHA Attack l'applique aux données en texte clair. Dans le cas d'une attaque avec SSHA Attack le texte en clair proviendrait soit d'une attaque par force brute ou par dictionnaire à l'exécution.

Dans le code source, ces étapes sont les suivantes :

```
strcpy(finalRequestPW, requestPW);
// copiez requestPW dans un tableau
// non signé

strcat(finalRequestPW, tempSalt);
// faire un cat du salt binaire en
// tableau binaire
```

```
GenerateHash(hashtype, finalRequestPW,
    NULL, buffer);
// génère un salt de hachage sha
```

La fonction `GenerateHash` utilise les librairies OpenSSL sur un système Linux pour générer le hachage approprié. Le `hashtype` a déjà été obtenu dynamiquement et il est passé comme premier paramètre à `GenerateHash`. Dans la fonction `GenerateHash` vous trouverez le code suivant. Comme vous pouvez le voir (Listing 5) le dernier paramètre passé

à `GenerateHash` (nommé : `buffer`) finira avec le salt de hachage binaire après que l'algorithme à sens unique ait réalisé sa magie. Cette opération a lieu à chaque fois pour une chaîne de texte claire qui provient soit du dictionnaire ou d'une méthode par force brute. La vérification finale qui établie une requête basée sur les 2 éléments qui permettront de savoir si le crack a réussi ou non ressemble à ceci :

```
// réaliser la comparaison de
// formattedPW et du buffer
```

Listing 7. Un exemple qui permet de générer des salts de hachage, de style LDAP, puis de les cracker.

```
SHA1
Chiffré en Base64: XjW3J0gbK+nkHDwCdLsksYxx/50wYmJm
SHA256
Chiffré en Base64: DP8Qwmb5LP1Br1H3EoJ/F7MXJwY9IPt8w3MiDm9r72QwYmJm
SHA384
Chiffré en Base64:
Yn19q3hVFGN8xUkfvfbCfZg7cZ6d3wqN2v199Ezuxjd9M0N4y8s6LN+ihIAxWV2tMGJiZg==
SHA512
Chiffré en Base64:
G1kSnef8EObDZdm1SHh0911J8TWP5eL0jGctHbG83NNhpWtV34fv8wuF3gOP/N37+RM0dbr8TP28ZQ1kxKr0r
    DBiYmJm=
On utilisera ces hashs comme exemple avec SSHA Attack pour découvrir les collisions,
et au final, cracker le texte clair représenté par un salt de hachage.
./ssha_attack -m brute-force -u 8 -a 9 -s XjW3J0gbK+nkHDwCdLsksYxx/50
    wYmJm

Algorithme de Hachage DéTECTé: SHA1
Essai Longueur Mot: 1
Aucun résultat pour Longueur de Mot: 1
...
Essai Longueur Mot: 4
Il y a une correspondance pour la valeur "T35t"
Temps écoulé en seconde pour une attaque avec succès: 22
./ssha_attack -m brute-force -u 8 -a 9 -s DP8Qwmb5LP1Br1H3EoJ/F7MXJwY9IPt8w3MiDm9r72Q
    wYmJm

Algorithme de Hachage DéTECTé: SHA256
Essai Longueur Mot: 1
Aucun résultat pour Longueur de Mot: 1
...
Essai Longueur Mot: 4
Il y a une correspondance pour la valeur "T35t"
Temps écoulé en seconde pour une attaque avec succès: 29
./ssha_attack -m brute-force -u 8 -a 9 -s Yn19q3hVFGN8xUkfvfbCfZg7cZ6d3wqN2v199Ezuxjd9
    M0N4y8s6LN+ihIAxWV2tMGJiZg==

Algorithme de Hachage DéTECTé: SHA384
Essai Longueur Mot: 1
Aucun résultat pour Longueur de Mot: 1
...
Essai Longueur Mot: 4
Il y a une correspondance pour la valeur "T35t"
Temps écoulé en seconde pour une attaque avec succès: 33
./ssha_attack -m brute-force -u 8 -a 9 -s G1kSnef8EObDZdm1SHh0911J8TWP5eL0jGctHbG83NNh
    pWtV34fv8wuF3gOP/N37+RM0dbr8TP28ZQ1kxKr0rDBiYmJm

Algorithme de Hachage DéTECTé: SHA512
Essai Longueur Mot: 1
Aucun résultat pour Longueur de Mot: 1
...
Essai Longueur Mot: 4
Il y a une correspondance pour la valeur "T35t"
Temps écoulé en seconde pour une attaque avec succès: 35
```

```
if(strcmp(formattedPW, buffer) == 0) {
    return 1;
// correspondance des mots de passes
}
```

Utiliser SSHA Attack

Le lien pour télécharger SSHA Attack peut être trouvé dans la partie *Sur le Net*. Une fois le tarball téléchargé, il faut faire un `untar`. On y trouve un `Makefile` qui vous permet de réaliser l'essentiel de la compilation et les instructions attachées. La véritable instruction pour exécuter le programme est : `gcc -O3 functions.o ssha_attack.o -lssl -o ssha_attack`. Cela suppose que vous ayez déjà compilé les 2 fichiers nommés `functions.c` et `ssha_attack.c` en fichiers objets. L'instruction de compilation ressemble à ceci : `gcc -O3 -c -o functions.o functions.c`. Mais vous pouvez uniquement utiliser `make` sur une distribution Linux. Une fois que vous avez exécuté la commande `make` avec le fichier inclut `Makefile` vous devriez obtenir un programme exécutable se situant dans le même répertoire où vous avez extrait les fichiers sources. Cela veut dire que vous devriez être capable d'invoquer SSHA Attack avec une notation standard point slash, c'est à dire `./ssha_attack` soit le même répertoire où vous aviez lancé l'utilitaire `make`.

Exécuter SSHA Attack avec l'option `-help` vous donne de plus amples informations sur son utilisation. Le Listing 6 vous montre l'affichage obtenu après une telle action. Déterminez votre plan d'attaque, vous avez globalement 2 choix, soit dictionnaire ou force brute.

Une attaque par dictionnaire se passe d'explications, vous devrez ne pas oublier d'indiquer le fichier dictionnaire avec la commande `-a`. Un dictionnaire est essentiellement une liste de chaînes (une par ligne) qui se verront appliquer le salt puis un hachage avec l'algorithme approprié.

Le mode force brute est un peu plus compliqué étant donné que vous devez indiquer au programme quel alphabet vous souhaitez utiliser.

Vous pouvez choisir un ensemble déjà présent en utilisant la commande `-a switch` ou proposer la vôtre avec la commande `-c`. Cette dernière technique

est intéressante pour les Tiger Teams (groupement de crackers) qui ont déjà une idée sur les possibles caractères utilisés (se basant sur l'analyse des navigations ou une compréhension des habitudes personnelles ou de l'historique).

En utilisant un alphabet préconstruit on générera une combinaison (comme dans *l'Algorithme du Produit Cartésien*) des données à utiliser.

Par exemple, en utilisant un alphabet se basant sur abcd et une combinaison de min – max et 1- 4 donnera le texte clair qui suit à utiliser avec le salt déjà extrait : Tabele 1, Tabele 2, Tabele 3, Tabele 5.

L'utilisation d'un alphabet personnalisé permet de générer l'ensemble des permutations possibles des données que l'on a renseignées manuellement.

Par exemple, par l'utilisation d'un alphabet abcd, le texte généré serait le suivant.

Pour vous montrer à quoi ressemble une véritable exécution on va d'abord générer des hachages.

Pour cet exemple voici l'affichage produit par un script en Python qui génère de multiples familles de hachage SHA. Pour rester simple j'ai utilisé (long de 4 caractères) une chaîne de texte clair : `t35t`. Une fois que ces hachages sont générés on utilisera SSHA Attack. Dans un cas réel nous ne connaîtrions pas la valeur du texte en clair, mais

ceci n'est qu'un exemple à des fins pédagogiques.

```
python genSSHA_py25.py t35t 4
```

Lorsqu'on analyse le facteur travail pour ces collisions, on comprend que nos exemples sont exécutés sur des biprocesseurs (Pentium(R) D 2.8 GHz) avec plate-forme Linux basée sur VMWare avec 768 MO de RAM. Pendant l'exécution une instance de l'utilitaire Linux `top` a montré que l'utilisation de la mémoire du programme SSHA Attack ne dépassait jamais 0.1% alors que les capacités du CPU allaient bien au-delà de 90%.

Le tableau qui suit résume le temps d'exécution en corrélation avec le type/taille du hash (Table 6).

Conclusion

L'utilisation de salts aléatoires lors du stockage de données sensibles est une bonne pratique et c'est un élément judicieux dans une architecture de défense mais ce n'est pas la panacée.

Un attaquant peut être futé dans sa manière d'extraire les valeurs de salts soit à partir de méthodes intégrées à partir du modèle LDAP qu'on a pu voir tout au long de cet article ou par d'autres techniques de stockage. Le salt doit être disponible pour une utilisation légale au sein d'une application, les salts de hachage sont sensibles aux attaques par cracking telles que celles présentées dans l'article.

Étant donné que les outils deviennent de plus en plus sophistiqués, la protection des mots de passe et de données sous forme de texte en clair deviendra de plus en plus difficile. Il y a des outils sur le web qui permettent de cracker aisément et rapidement des hachages unsalt à sens unique. Aujourd'hui une nouvelle génération d'outils apparaît et ceux-ci ciblent les zones les plus complexes des données sensibles.

Ne soyez pas surpris si ces outils commencent à utiliser des techniques de programmation basées sur le calcul distribué pour accroître de façon exponentielle leur efficacité.

Un parfait exemple de ceci est le projet BOINC qui recherche des collisions avec des hachages unsalt de famille SHA-1.

Le projet BOINC apporte au monde entier sa puissance dans le calcul distribué. C'est un projet open source qui a pris vie grâce à des volontaires fournissant de la puissance de calcul pour la résolution de calculs longs et complexes.

Vous pouvez obtenir de plus amples informations sur ce projet à : <http://boinc.berkeley.edu/>. Les communautés qui devaient servir de ces travaux étaient à la base des scientifiques, mais la communauté informatique a réalisée les apports que pouvait avoir une architecture en grid pour des calculs complexes comme le crack de schémes de cryptage.

En liaison avec cet article : le projet de Recherche de Collisions SHA-1, pour de plus amples informations sur ce projet : http://boinc.iaik.tugraz.at/sha1_coll_search/.

Ceci n'est qu'un exemple et vous pouvez en trouver beaucoup d'autres sur : <http://distributedcomputing.info/ap-crypto.html>

À propos de l'auteur

Andres Andreu travaille dans le milieu de l'ingénierie logicielle depuis plusieurs années en concevant des solutions basées sur le web pour des organismes du gouvernement américain. Il s'implique fortement dans la sécurité des applications web et dans le domaine du pen-testing et est l'auteur des programmes OWASP WSFuzzer et SSHA Attack ainsi que du livre *Professional Pen Testing for Web Applications* (ISBN-13: 978-0471789666).

Tableau 6. Résumé de l'exécution pour le Listing 7

Algorithme SHA	Temps (en secondes) pour des collisions de 4 octets
SHA1	22
SHA256	29
SHA385	33
SHA512	35

Sur Internet

- http://cryptography.hyperlink.cz/MD5_collisions.html,
- <http://www.msccs.dal.ca/~selinger/md5collision/>,
- <http://www.stachliu.com.nyud.net:8090/collisions.html>,
- http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html,
- <http://www.rsa.com/rsalabs/node.asp?id=2927>,
- https://www.iaik.at/research/krypto/collision/SHA1Collision_Description.php,
- <http://sourceforge.net/projects/ssha-attack>.