SANS Institute
InfoSec Reading Room

Integrate HMAC Capable Token into User Authentication Mechanism and Public Key Infrastructure

This article describes using a HMAC capable token in user authentication or public key infrastructure (PKI) to derive user private key or produce message digest for digital signature scheme. The unique hardware token will be linked together with the user password cryptographically to provide a more secure/stronger solution.

Integrate HMAC Capable Token into User Authentication Mechanism and Public Key Infrastructure

Abstract

This article describes using a HMAC capable token in user authentication or public key infrastructure (PKI) to derive user private key or produce message digest for digital signature scheme. The unique hardware token will be linked together with the user password cryptographically to provide a more secure/stronger solution.

Introduction to Hash function and HMAC

Hash function, H( ) is a one-way function which take variable length message, M as the input and produce a fixed length output value, h= H(M). Due to this characteristic it is also called *compression* function.

Hash function is usually used to produces "fingerprint" or "message digest" of the input [1], either a message, data stream or a file. In this article we are particularly interested in "dedicated design iterated cryptographic"[2] hash function, such as MD5, SHA-1 and RIPEMD. These function tend to be faster than the hash functions which built around block ciphers(This is one of the main reasons why we are interested in using hash and HMAC rather than block cipher). Besides that, a hash function must have following properties (adapted from a list in [3]):

a) H can be applied to a block of date in any length.
b) H produces a fixed-length output.
c) h=H(M) is relatively easy to compute for any given M, making both hardware and software implementations practical.
d) one-way property – it is computationally infeasible to find M from H(M)
e) Weak collision resistance – infeasible determination of another different message, M' such that H(M') = H(M) for any given M
f) Strong collision resistance – infeasible determination of message pair (M, M') with M'≠ M such that H(M)=H(M').

The length of the hash function output is very important in protecting the algorithm from brute force attack. An n-bit hash function result only requires $2^{n/2}$ evaluations of the function in a brute force collision search attack. Most of the modern hash functions which widely use nowadays like MD5(128 bits), RIPEMD-160(160 bits), SHA1(160 bits), SHA256(256 bits) have output size of at least 128 bits. However, MD5 hash been recently shown to be vulnerable to collision search attack[4], since it only requires 2^{64} or about 2.10^{19} evaluations (strong collision resistance), which can be done in a fairly short time as the computation speed has tremendously increased in recent years. But a 160-bit hash function is expected to be secure for the next 10 years or more[4].

Table 1.1 shows a comparison of well-know hash functions MD5, SHA-1 and RIPEMD-160. However comparison of performance and quality of these functions is out of the scope of

this article. In the viewpoint of cryptography, we can generally consider that a hash algorithm is safer or better if the digest length is longer than another algorithm.

Algorithm	MD5	SHA-1	RIPEMD-160
Digest length, L	16 bytes	20 bytes	20 bytes
Block size, B	64 bytes	64 bytes	64 bytes
Iteration	64(4 rounds of 16)	80(4 rounds of 20)	160(5 paired rounds of 16)

Table 1.1

Message authentication code (MAC) can be generated based on the use of symmetric block cipher-namely, the data authentication algorithm defined in FIPS PUB 113; or derive from a cryptographic hash function, HMAC. HMAC has been issued as RFC2104. HMACs have the same properties as the one-way hash function discussed previously, but also include a key[5]. The key must share by both the HMAC generator and verifier. In another words, HMAC can only be verified by someone who knows about the identical key which is used to generate the HMAC code. The cryptographic strength of HMAC depends on the properties of the underlying hash function[6]. Please refer to RFC2104 for more detail about the implementation of HMAC.

The length of HMAC key can be of any length up to the block size of the hash function. Any key with the length more than hash function block size will not improve the strength of HMAC from cryptography viewpoint. Besides, the key has to be hashed first and then use the result of the computation as the actual key to HMAC. As recommended in RFC2104, the key length must at least as long as the output size of the hash function, otherwise it would decrease the security strength of the function.

If an attacker want to attack HMAC based authentication, he will need to find out the secret key of the HMAC. We assume that there is no serious flaws in the collision behavior of the hash function discover. Take a HMAC with output size of 128 bits as the example, the attacker needs to acquired 2^{64} correct plain messages with the corresponding HMAC value (with the same key) to find out the right HMAC secret key. The attacker can only replace or generate fake massages and compute a good HMAC result if he know about the secret key. As explained in RFC2104, this is considered an impossible task in any realistic scenario (for a message block length of 64 bytes, this would take 250,000 years in continuous 1Gbps link, provided without changing the secret key during all this time). If we replace a strong HMAC with 160 bits output size then the number of original messages and HMAC code required should become 2^{80} which is even harder and impossible to attack.

Due to the cryptographic strength of HMAC algorithm, the solution that I propose in this article is considered strong and safe. There is another advantage of HMAC based solution, easy and fast in computation. The server side verification upon many messages or users at a same time can be done in fairly high efficiency. This will reduce the overhead of the server processor and will not affect too much on the server performance for other tasks.

Usage of HMAC capable Token in User Authentication

In this article, we are interested in hardware tokens with unique serial number, capacity of on-board HMAC computations and capacity to keep some hidden parameters (HMAC secret keys) inside the token which must not expose to outside world, except known the party who need to authenticate a message or an user. The token can be in any form factor like smart card or a hardware which can plug in or connect to PC, either with parallel port, serial port or USB port. There are some products available in the market now which have the properties and capabilities needed in this discussion.

The uniqueness of the serial number is a very important factor in this solution. It is also better if the serial number has longer bits, but the main reason is just to provide enough unique token for every single user in the particular group. Most of the token widely use today has serial number of 64 bits. In another words, we can have 2^{64}, about $1.8x10^{19}$ unique tokens guarantee by the manufacturer. However, since we will use the serial number as an input parameter to HMAC function, it could be better if there is a token with serial number of size at least the output size of the HMAC.

We usually only rely on password solution in the process of user authentication. However this is not good enough since end user always choose poor password (so that easy to remember), which can be easily guessed or vulnerable to dictionary search. There are even many software which can help attacker to launch a brute force attack to find out a password. What I am proposing in this article is using a HMAC capable token in conjunction with user input password (knowledge based proof) to provide a more secure solution.

To simplify this discussion, I will only consider a simple client server authentication scenario at here. Client is the end user who might be logon to a system or send an authenticated message to a server. The server must be able to check out if the client is an authorized user or the message it received is a legitimate message from the sender.

Before get into the point, first of all we have to know about the process of keying a token during user registration by Trusted Authority (TA). Before TA insert an *identification HMAC key* to a token, denote as $K_{identification}$, he must first select a random generated *master identity secret* (for identification purpose), denote as $S_{identity}$, which must also be known by the server which need to authenticate a user or incoming message. The length of $S_{identity}$ can be in any length up to the size of HMAC block size. However, any $S_{identity}$ with the length longer than the HMAC block size may not increase the function strength. It is also very important that the token has a unique serial number, denote as *i*, because we are using this serial number as the index to a specific user. We denote HMAC(a , b) as applying HMAC using "a" as the key and "b" is the data we want to compute the digest. So, TA need to compute $K_{identification}$ = HMAC($S_{identity}$, *i*) and then insert this key in to the token. This is the secret key keep in side the token which should not expose to outside. Since only TA and authentication server know about $S_{identity}$ and always be able to compute HMAC($S_{identity}$, *i*), which means server can authenticate any message (*M*) receive from client consist of digest HMAC($K_{identification}$, *M*). It is also a very important feature that even an intruder try to get a token with same serial number, he still cannot fake a token with the correct key, since he doesn't know about the *master identity secret* to generate *a identification HMAC token key* by TA. So, the serial number, *i* can be publicly known. Up to this point, the solution is still not good enough since we haven't involved the client yet! To link the client to the whole

system, we must acquire a proof from the client, which is knowledge based proof – password. In the process of registration a user, we must let user choose an authentication password (P). Then the server only need to record user's ID (or name), corresponding token serial number and his password in database, and the password must save in encrypted form (other information can be known publicly and will not threaten the security of this solution).

In this scenario, there are three entities involved, token (with a secret key inside), the client (who know the password) and the server (which has all the required information needed to verify challenge response value or incoming message). These three independent entities need not to fully trust each other. In another words, an attacker can steal a legitimate token and pretend to be the client by guessing the password, or try to fake a token (he cannot do so since he doesn't know about the master identity secret to generate a legitimate HMAC key to insert to token) and use the password he get from the client (in a illegal way) to cheat the server. In either case, the attacker cannot compromise this system. The only way to be authenticated is own the legitimate token (registered and key inserted by the TA) and know exactly the client's password.

How is the mechanism of user or message authentication in this solution? Whenever the server want to authenticate a user, the server can send a random generated challenge value (C) to the client, then the client will have to response by compute $v = HMAC(K_{identification}, C)$ using the token and then compute $R_{client}=HMAC(P, v)$. R_{client} is the response value which the client has to sent back to server for verification. At the server side, it must know which user is trying to logon, then it will retrieve the required parameters from the database (i.e. the user specific token serial number and password), and compute the correct response value R_{server} locally (since server know about the *master identity secret*). Then, server can compare R_{client} with R_{server} to determine if the user is a legitimate user and own the corresponding legitimate token. An attacker with only either a stolen token or just knowing the user password is not enough to compromise the system. He cannot compute a correct response value without both of the components.

For generating or authenticating a message from a user, the same computation is carried out at both client and server as previously described, except instead of challenge value, now replace by the original message. Then the client sends both the original message together with the HMAC digest to server.

From the performance basis, implementation of this solution should be quite efficient, since we only use HMAC computation at both the client and server sides.

Usage of HMAC capable Token to derive Private Key in Public Key Infrastructure

We should understand that the HMAC result has the characteristic of pseudorandom like. This actually tells us that we can also use the token side HMAC computation to derive random cryptographic key from some other parameters like user ID and password. I must emphasize at here that this is only applicable to algorithm which use random key (i.e., DSA, Schnorr or ElGammal digital signature scheme and symmetric block cipher) rather than random derive key (i.e., RSA).

The question comes again is how to do that? It is fairly simple. Now we must introduce to another *key-generating secret*, S_{gen_key} during the process of key insertion to the token. The purpose of this secret key is just to generate a random and unique *key-generating HMAC key*. This secret has the same requirements as the *master identity secret*, $S_{identity}$ in previous section. We generate this secret randomly and with the suitable length (please refer to previous section about the requirements of master secret). However, this secret should be forgotten by the Trusted Authority(or the token issuer), after the process of token *key-generating HMAC key* insertion (this is important to ensure that even the TA does not able to reproduce the user private key it in the future). The secrecy of user private key is also relied on the secrecy of user password. So, first of all TA has to compute a unique *key-generating HMAC key*, $K_{pri\_key\_gen} = HMAC(S_{gen\_key}, i)$, again *i* is the token serial number. $K_{pri\_key\_gen}$ is the secret HMAC key insert to token for generating the user private key whenever needed. TA can just forget about S_{gen_key} after this step, he can never reproduce $K_{pri\_key\_gen}$ in future without S_{gen_key}.

After that we can compute the user private key, PriKey = HMAC($K_{pri\_key\_gen}$, user ID || password). (note: symbol "||" means concatenation). Then we need to compute the corresponding user public according to which kind of algorithm we are using (different public key cryptography algorithm has different formula to compute the public key). The server just has to keep record about the user public key. The client will feel even safer because no one else knows about his private key, even the TA (since it generate S_{gen_key} randomly and forget about it after the *key-generating HMAC key* insertion to the token). The privacy of client private key is important to prevent a client from repudiation in any transaction.

To recover the private key, the client need to provide user ID and password to the token in order to compute the private key at token side. Anyone provide a wrong user ID or password will get a wrong private key comes out from the token. Again, we have linked the token with the client's knowledge-based proof cryptographically to derive the correct private key. The private key can be used for any purpose in public key infrastructure including signing messages and symmetric key exchanging.

However, there is also a requirement on the private key length being derived from the token. The HMAC algorithm will only produce a value as long as its output length. Most of the hash functions have output block of 128 bits (absolutely fine for most symmetric block cipher) or 160 bits. Since elliptic curve cryptography (ECC) has come into the place, this algorithm offer equal security for a far smaller bit size as compare to conventional public key cryptographic algorithm. The key length of 128 bits or 160 bits can fulfill the basic safety requirement for the ECC algorithm. More over, ECC can be applied to most of the discrete logarithms algorithm by only applying some modification to the particular algorithm.

One of the methods to generate a longer bit length key, we can reuse the result of PriKey(0) = HMAC($K_{pri\_key\_gen}$, user ID || password) to compute PriKey(1)= HMAC($K_{pri\_key\_gen}$, Prikey(0)) ……PriKey(n) = HMAC($K_{pri\_key\_gen}$, PriKey(n-1)). Finally, we concatenate all PriKey(0) || PriKey(1) || ……|| PriKey(n) and use the result as the private key. Attacker still cannot find out the user private key so long as he doesn't have the token as well as know the user ID and password.

Usage of HMAC capable Token to compute Message Digest in Digital Signature

On board HMAC can also be used to compute the intermediate value when we want to digital sign a message. In any public key digital signature algorithm, when we want to sign a message there must be a step which we need to compute the digest of the message before comes out with the digital signature value. We usually will only use a cryptographic hash function in order to do so. Since now we have a HMAC capable token, and yet with a unique secret key keeps in side the token, we can replace the normal hash function by applying HMAC($K_{identification}$, message) computation using the token to produce the digest of the message. The server also has the same capacity to compute the same digest since it has all the parameters (the *master identity secret*, $S_{identity}$ and the token serial number, *i*) to reproduce the same *identification HMAC key* as inside the token - by only compute HMAC($S_{identity}$, *i*) = $K_{identification}$, then apply HMAC using the key to compute the message digest for verification. This alternative method can be apply to any digital signature scheme (since all algorithms need to compute the message digest), regardless of algorithm requirement for random key or random derive key.

Conclusion

The solution propose in this article has some advantages and a few matters which should be aware of when implement the solution. This solution has been very strongly linking the token together with user knowledge based proof (password) by applying cryptographic understanding about the features and characteristics of HMAC algorithm. Instead of store user private key inside the token storage, we are using the token HMAC generation capacity to derive the private key whenever we need the key. The strength of the solution rely on the secrecy of *master identity secret* (which should only known by the Trusted Authority to register a user token and authentication server), the *key-generation secret*, *secret HMAC keys* inserted to the token (for both identification and private key generating purpose), and also the user password. The requirements on the master secret should follow the recommendation as in RFC2104 document. Though there is a requirement on the uniqueness of token serial number, but even an attacker can have a token with same serial number, it is impossible to recreate a fake token with legitimate HMAC keys in the token without knowing the master secret known by the Trusted Authority. We also depend very much on the cryptographic strength of the hash function used in the implementation. Another advantage of this solution is that HMAC can be computed in a very high speed both at PC side or token side as compare to other algorithm like block cipher MAC or RSA. So the performance of the whole protocol should be fairly efficient.

Finally, I must say that the client has the full responsibility to protect both his token and password. User should use a strong password rather than a password which can be easily guessed by attacker. The attacker can only compromise the system with both token and the password.

References

[1] R. Rivest, "The MD5 Message-Digest Algorithm", MIT Laboratory for Computer Science and RSA Data Security, Inc. April 1992. http://theory.lcs.mit.edu/~rivest/rfc1321.txt

[2] RSA Laboratory, "What other hash functions are there?", Crypto FAQ. http://www.rsasecurity.com/rsalabs/faq/3-6-11.html

[3] Stallings, William Cryptography And Net Security, Principles and Practice, NJ: Prentice-Hall, Second Edition, page 253.

[4] Antoon Bosselaers, "The hash function RIPEMD-160", 17 August 1999 http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html

[5] Bruce Schneier, Applied Cryptography , Protocols, Algorithms, and Source Code in C, second edition, page 455.

[6] M. Bellare, H.Krawczyk, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", February 1997. ftp://ftp.isi.edu/in-notes/rfc2104.txt

[7] Aviel D. Rubin, "Kerberos Versus the Leighton-Micali Protocol" http://www.ddj.com/articles/2000/0011/0011a/0011a.htm

[8] ElGamal, T. "A Public-Key Crytosystem and a Signature Scheme Based on Discrete Logarithms." IEEE Transactions on Information Theory, July 1985.

[9] Schnorr, C. "Efficient Signatures for Smart Card." Journal of Cryptology, No.3, 1991.

[10] Schneier, Bruce, Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in. New York, NY: John Wiley & Sons, 1996.

[11] RSA Laboratories, "Cryptography FAQ What is a hash function" http://www.rsasecurity.com/rsalabs/faq/2-1-6.html

[12] RSA Laboratories, "What are some of the basic types of cryptanalytic attack?" http://www.rsasecurity.com/rsalabs/faq/2-4-2.html

[13] RSA Laboratories, "What are some techniques against hash functions?" http://www.rsasecurity.com/rsalabs/faq/2-4-6.html

[14] RSA Laboratories, "What are the most important attacks on MACs?" http://www.rsasecurity.com/rsalabs/faq/2-4-8.html

[15] Handbook of Applied Cryptography, chapter 9, "Hash Functions and Data Integrity"
http://www.cacr.math.uwaterloo.ca/hac/about/chap9.pdf

[16] Rainbow Technologies, iKey 1000 product product specifications.
http://www.rainbow.com/ikey1000/ikey1_specs.html

[17] Rainbow Technologies, iKey 1000 Overview.
http://www.rainbow.com/ikey1000/index.html

Last Updated: March 18th, 2018

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Munich March 2018	Munich, DE	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Pen Test Austin 2018	Austin, TXUS	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Secure Canberra 2018	Canberra, AU	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Boston Spring 2018	Boston, MAUS	Mar 25, 2018 - Mar 30, 2018	Live Event
SANS 2018	Orlando, FLUS	Apr 03, 2018 - Apr 10, 2018	Live Event
SANS Abu Dhabi 2018	Abu Dhabi, AE	Apr 07, 2018 - Apr 12, 2018	Live Event
Pre-RSA® Conference Training	San Francisco, CAUS	Apr 11, 2018 - Apr 16, 2018	Live Event
SANS Zurich 2018	Zurich, CH	Apr 16, 2018 - Apr 21, 2018	Live Event
SANS London April 2018	London, GB	Apr 16, 2018 - Apr 21, 2018	Live Event
SANS Baltimore Spring 2018	Baltimore, MDUS	Apr 21, 2018 - Apr 28, 2018	Live Event
Blue Team Summit & Training 2018	Louisville, KYUS	Apr 23, 2018 - Apr 30, 2018	Live Event
SANS Seattle Spring 2018	Seattle, WAUS	Apr 23, 2018 - Apr 28, 2018	Live Event
SANS Riyadh April 2018	Riyadh, SA	Apr 28, 2018 - May 03, 2018	Live Event
SANS Doha 2018	Doha, QA	Apr 28, 2018 - May 03, 2018	Live Event
SANS SEC460: Enterprise Threat Beta Two	Crystal City, VAUS	Apr 30, 2018 - May 05, 2018	Live Event
Automotive Cybersecurity Summit & Training 2018	Chicago, ILUS	May 01, 2018 - May 08, 2018	Live Event
SANS SEC504 in Thai 2018	Bangkok, TH	May 07, 2018 - May 12, 2018	Live Event
SANS Security West 2018	San Diego, CAUS	May 11, 2018 - May 18, 2018	Live Event
SANS Melbourne 2018	Melbourne, AU	May 14, 2018 - May 26, 2018	Live Event
SANS Northern VA Reston Spring 2018	Reston, VAUS	May 20, 2018 - May 25, 2018	Live Event
SANS Amsterdam May 2018	Amsterdam, NL	May 28, 2018 - Jun 02, 2018	Live Event
SANS Atlanta 2018	Atlanta, GAUS	May 29, 2018 - Jun 03, 2018	Live Event
SANS Rocky Mountain 2018	Denver, COUS	Jun 04, 2018 - Jun 09, 2018	Live Event
SANS London June 2018	London, GB	Jun 04, 2018 - Jun 12, 2018	Live Event
DFIR Summit & Training 2018	Austin, TXUS	Jun 07, 2018 - Jun 14, 2018	Live Event
SANS Milan June 2018	Milan, IT	Jun 11, 2018 - Jun 16, 2018	Live Event
SEC487: Open-Source Intel Beta One	OnlineVAUS	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced