Security Protocols and Application — Final Exam
Solution

Philippe Oechslin and Serge Vaudenay

17.6.2013

- allover exam duration: 2h00
- no document allowed
- a pocket calculator is allowed
- communication devices are not allowed
- the exam invigilators will not answer any technical question during the exam
- the answers to each exercise must be provided on separate sheets
- readability and style of writing will be part of the grade
- do not forget to put your name on every sheet!

Both exercises have the same weight in the final grade. The exercise grades follow a linear scale in which each question has the same weight.

1 Security Analysis of a Multi-Factor Authenticated Key Exchange Protocol

We recall that the MF-AKE authentication protocol works over a group of large prime order q generated by some g. We use two random group elements u and v. The protocol is depicted on the following diagram:

Client		**Server**
$W'_1,\dots,W'_N,x,\mathsf{pwd}$		$(g^{r_1},h^{r_1}g^{W_1}),\dots,(g^{r_N},h^{r_N}g^{W_N}),h=g^x,\mathsf{pwd}$
$b\in\mathbf{Z}_q, B=g^b, B^*=Bv^{\mathsf{pwd}}$	$\xrightarrow{B^*}$	$r'_i\in\mathbf{Z}_q, g^{s_i}=g^{r'_i}g^{r_i}, h^{s_i}g^{W_i}=h^{r'_i}h^{r_i}g^{W_i}, i=1,\dots,N$
for $i=1,\dots,N$:	$\xleftarrow{g^{s_1},\dots,g^{s_N},A^*}$	$a\in\mathbf{Z}_q, A=g^a, A^*=Au^{\mathsf{pwd}}$
$K_C=\left(\frac{A^*}{u^{\mathsf{pwd}}}\right)^b, K_C^i=(g^{s_i})^x g^{W'_i}$		
$K'_i=g^{s_1},\dots,g^{s_N}\|A^*\|B^*\|K_C^i\|K_C\|\mathsf{pwd}\|i$		
$H(K'_i)=\alpha'_i\|\beta'_i\|k'_i$	$\xrightarrow{\alpha'_1,\dots,\alpha'_N}$	for $i=1,\dots,N$:
		$K_S=\left(\frac{B^*}{v^{\mathsf{pwd}}}\right)^a, K_S^i=h^{s_i}g^{W_i}$
		$K_i=g^{s_1},\dots,g^{s_N}\|A^*\|B^*\|K_S^i\|K_S\|\mathsf{pwd}\|i$
		$H(K_i)=\alpha_i\|\beta_i\|k_i$
$K'=$ first k bits k'_i s.t. $\alpha_i=\alpha'_i$	$\xleftarrow{\beta_1,\dots,\beta_N}$	$K=$ first k bits k_i s.t. $\alpha_i=\alpha'_i$
if $\#\{i;\beta_i\neq\beta'_i\}>t$, reject		if $\#\{i;\alpha_i\neq\alpha'_i\}>t$, reject

Q.1 What is the role of x, h, pwd, and the W_i's?

x is the secret key, h is the public key, pwd *is a (low-entropy) password, and the W_i's are the bits of a biometric template.*

Q.2 What is the difference between W_i and W'_i?

W_i is from the reference template (capture at enrolment). W_i' is the template which is captures during authentication.

Q.3 Why don't we store W_i instead of $(g^{r_i}, h^{r_i} g^{W_i})$ on the server side?

We want to have privacy-by-design. Namely, we want to avoid that the server can deduce the reference template in clear. This is why each bit is encrypted using the ElGamal cryptosystem.

Q.4 Explain how the server computes g^{s_i} and $h^{s_i} g^{W_i}$.

g^{s_i} is the product of g^{r_i} (from the database) and g raised to the power r_i', a random $\mathbf{Z}_q$ element.
$h^{s_i} g^{W_i}$ is the product of $h^{r_i} g^{W_i}$ (from the database) and h raised to the power r_i'.

Q.5 If the protocol succeeds, explain why we have $K = K'$.

We assume that the α_i's and β_i's are large enough. If we have $\alpha_i = \alpha_i'$, we must thus have $K_i = K_i'$, due to the collision resistance of H restricted to the α part. So, we must have $\beta_i = \beta_i'$ as well. The converse is similar. So, the set of i's such that $\alpha_i = \alpha_i'$ and the set of i's such that $\beta_i = \beta_i'$ are identical.

Q.6 Show that by selecting $s_i \in \mathbf{Z}_q$ at random, $i = 1, \ldots, N$, an adversary who knows pwd and h and tries to play the role of the server can recover the W_i''s.

By selecting the s_i's, the adversary can compute g^{s_i} and send to the client. The client would return some α_i''s. Since W_i' is a bit, the adversary knows that K_i' can take two different values which are h^{s_i} (if $W_i' = 0$) and $h^{s_i} g$ (if $W_i' = 1$). He can compute both and find two possible values for $H(K_i')$. The one with the matching α_i' tells which values has the bit W_i'.

2 Hash DoS attacks

AWK is simple scripting language available on all unix platforms. The language uses a hash fonction to implement arrays of strings. In the latest version of GNU awk (4.1.0) we can find the following comments in the source code:

```
/*
 * Even more speed:
 * #define HASHC   h = *s++ + 65599 * h
 * Because 65599 = pow(2, 6) + pow(2, 16) - 1 we multiply by shifts
 *
 * 4/2011: Force the results to 32 bits, to get the same
 * result on both 32- and 64-bit systems. This may be a
 * bad idea.
 */
```

The hash function iterates through all characters of a string. It multiplies the current value of the hash h with the constant 65599 and adds the next character s to the hash. Initially h is zero.

Q.1 This hash function can be subjected to an equivalent substring attack. Explain how this attack works and why this hash is vulnerable to it.

The attack consists in finding sets of substrings that have identical hashes and creating collisons by creating all combining of substrings from different sets. Assume that the string is composed of two substrings. The calculation of the partial hash of each substring does not depend on the characters of the other substring.

Q.2 You want to mount the equivalent substrings attack using strings made of an alphabet of 64 different characters (2^6). You want to find one pair of colliding substrings. How long must your substrings be such that you can be reasonably sure that you will find one pair of colliding substrings? Explain the assumptions that you made to find your result.

From the birthday paradox we know that we need about $\sqrt{n}$ elements to have a 50% chance of collision in a set of n possible values. Thus we need about 2^{16} substrings. Two characters is too short (2^{12} combinations), thus we need three (2^{28}).

Q.3 If the maximum length of the parameter you can pass to an AWK script is 64 characters long, how many coliding strings can you construct using two colliding substrings ?

We can fit 21 substrings of 3 characters in a parameter. We thus have 2^{21} colliding strings of length 63.

Q.4 What is the number of operations that AWK will have to carry out to insert all these strings into an array ?

It is $O(N^2)$ which is $O(2^{42})$

Q.5 Now assume the script only accepts parameters of up to nine characters. A colliding substring attack will not be efficient and we have to try a meet-in-the-middle attack.
Which property must the hash function have to make a meet-in-the-middle attack possible ?

It must be possible to calculate the hash backwards.

Q.6 Explain the operations that have to be carried out to mount a meet-in-the-middle attack against a hash function h.

Divide the strings in suffix and prefix. Find the inverse function of the hash function for the suffix. From a given hash calculate backwards the intermediate hash value for a number of suffixes and store the results in a table. For a number of prefixes calculate the intermediate hash value and look it up in the table. If a match is found the concatenation of the prefix and the suffix is a colliding string.

Q.7 Find the parameters of the meet-in-the-middle attack that will generate 2^{21} colliding 9-character strings with the smallest possible number of operations while keeping the memory usage low. Give the number of operations and the memory needed for your attack.

We chose a prefix of length 5 and a suffix of length 4. We inverse all possible suffixes to generate about 2^{24} entries in a table. Generate 2^{29} prefixes (out of the 2^{30} possible ones). The chance of hitting an entry in the table is 2^{-8} yielding about 2^{21} collisions. The number of operations is thus $O(2^{29})$ and the memory usage $O(2^{24})$.

Q.8 The meet-in-the-middle attack appears to be better since it can create large numbers of collisions on shorter strings than the equivalent substring attack. Assuming the hash function is vulnerable to both types of attack, can you describe a case were the equivalent substring attack has an advantage over the meet-in-the-middle attack ?

To mount an equivalent substring attack we only need one pair of colliding substrings. If collisions are hard to find (e.g. with 64 bit hashes) then the equivalent substring attack is the only way to create large numbers of collisions.

Q.9 Siphash is a 64 bit hash function that was designed to replace the hash functions that are vulnerable to hash dos attacks. Give two reasons why Siphash is superior to a cryptographic hash function like SHA1, truncated to 64 bits.

Siphash is faster. Siphash has a 128 bit key that the attacker must know in order to be able to find any collisions off-line.