MOZ2-8: J-PAKE Implementation Allows Session Key Retrieval Without Knowing The Shared Secret

General

Title:	J-PAKE Implementation Allows Session Key Retrieval Without Knowing The Shared Secret
Update ID:	MOZ2-8
Notification date:	1/24/2011
Contributors:	
Report type:	vendor report

Summary

The J-PAKE implementation in Firefox browser's Sync client doesn't seem to check if the other party has provided *g^x2* equal to 1 (which would mean that *x2* is 0). Additionally, neither Firefox Sync nor Firefox Home seem to set the upper bound for *g^x2* (or perform an initial modulo *p* operation on it) to prevent the other party from providing number `p+1` which is in effect identical to 1.

This allows an active attacker (rogue server) to masquerade as a legitimate peer without knowing the shared secret. As a result, a rogue server can first extract the Sync credentials from the "sender", and then immediately pass them on to the "receiver", thus avoiding the legitimate J-PAKE parties from noticing anything suspect.

ⓘ Problem type: **Actual security problem**

Affected Components

- Sync Client (Firefox)
- Firefox Home (iPhone)

Demonstrations

No demonstration is available for this security problem.

Analysis

The J-PAKE protocol (http://eprint.iacr.org/2010/190.pdf, page 10) requires both parties to check, at the end of round 1, that the other party's *g^x2* (or *g^x4*) is not equal to 1. Namely, if it were equal to 1, it would mean that *x2* (or *x4*) would be 0, and the commonly generated keying material *K*, which has both *x2* and *x4* in the exponent, would thus be 1 regardless of the shared secret (i.e., the 8-char J-PAKE password).

We found that while Firefox Home J-PAKE code (`jpake.c`) implements a check that the other party's *g^x2* (or *g^x4*) is not 1 (function `JPAKE_STEP1_process`):

```
/* g^xd != 1 */
if(BN_is_one(received->p2.gx))
{
 JPAKEerr(JPAKE_F_JPAKE_STEP1_PROCESS, JPAKE_R_G_TO_THE_X4_IS_ONE);
 return 0;
}
```

we found no such check in the JavaScript implementation of J-PAKE in Firefox Sync client (`jpakeclient.js`). We did find a check for `s != 0` in `nsSyncJPAKE.cpp`, but no check for the other party's (received) *g^x2* (or *g^x4*). Note that it is not enough for a communicating party to verify its own *x2* to be larger than 0 (or *g^x2* to be larger than 1) - this is important but would only provide an extremely unlikely and rare leverage to a passive attacker. It is much more important **to prevent the other party from using *x2* (or *x4*) equal to 0**, because this would allow an active attacker to successfully, and reliably, perform the J-PAKE protocol without knowing the shared secret.

An attack could be mounted by an active man-in-the-middle, in our case a rogue Sync server (since the communication with the server is SSL-protected). In particular, the attack scenario would go like this:

1. User initiates the J-PAKE protocol on the receiver Sync client ("receiver").
2. Receiver requests a new channel on the server and posts the first J-PAKE message with *g^x1*, *g^x2* and the zero-knowledge proofs for *x1* and *x2*. Receiver starts polling the server for sender's message 1.
3. User types the shared secret (8-char password) to sender Sync client ("sender").
4. Server accepts sender's request for receiver's message 1, but provides its own message 1 instead, generated from its own *x3* and *x4* where *x4* is 0 and thus *g^x4* is 1. (The actual receiver is taken out of the loop and just keeps polling the server for sender's message 1, while the server takes over his role.)
5. Sender gets the server's (malicious) message 1 with *g^x4* = 1, and provides its own message 1 to server.
6. Server computes, as specified by the protocol, *A* = *g^((x1+x3+x4)*x2*s')* and sends it to the sender. Note that server doesn't know the shared secret *s* (which is only known to sender and receiver), so it uses an arbitrary *s'* instead or simply 0.
7. Sender computes, as specified by the protocol, *B* = *g^((x1+x2+x3)*x4*s)*, and sends it to the server.
8. Sender and server both calculate the shared keying material *K* as specified by the protocol, but since both formulas have *x4* in the exponent, and *x4* is 0, *K* will inevitably be 1 regardless of the difference between shared secret *s* (which is unknown to server) and server's arbitrary *s'*.
9. Both sender and server calculate the session key *k* by hashing the keying material *K* (which they both know is 1).
10. Sender now requires server to prove the knowledge of *k* by having the server encrypt the control string "0123456789ABCDEF" with *k*. Since *k* is known by server, it is able to provide this proof.
11. Sender, having verified that server knows *k*, now encrypts user's Sync credentials with *k* and sends it to server. This concludes the attack: the rogue server now has user's

username, password and Sync key and can thus obtain and decrypt his encrypted data on Sync storage servers.

This attack could be enhanced by passing the stolen Sync credentials on to the legitimate receiver (which is still waiting, polling the server for the sender's message 1). Using the same trick, the server can establish a shared key, again without knowing the actual secret *s*. By doing so, the user would not notice anything suspect as the Sync credential migration would indeed succeed.

This is a similar flaw to the one discovered in OpenSSL and OpenSSH in September 2010 (http://seb.dbzteam.org/crypto/jpake-session-key-retrieval.pdf): these products did check that the received *g^x2* (or *g^x4*) were not equal to 1 but failed to account for the possibility of these values being *p*+1, 2*p*+1, 3*p*+1 etc., which all equal to 1 in the modulo *p* calculation. A rogue other party could thus send *g^x2* (or *g^x4*) equal to *p*+1, knowing that it will be understood as 1 in the modulo *p* calculation. If the receiving party didn't make sure to calculate the received *g^x* modulo *p* before comparing it to 1, it could be tricked into effectively accepting *g^x2* (or *g^x4*) equal to 1.

That said, we found no such preliminary modulo *p* calculation of the received numbers in either Firefox or Firefox Home, and no check for the received numbers being less than *p*, thus we conclude that both must be susceptible to such "*g^x* = *p*+1" attack.

Granted, the attacker would either have to break into Mozilla's Sync server `setup.services.mozilla.com`, be a rogue administrator of that server or be able to set up a fake server with a trusted SSL server certificate and perform a DNS spoofing attack to redirect users' clients to that server. All these options seem very unlikely, but since one of Firefox Sync's competitive advantages compared to Google Chrome bookmarks sync is the vendor's guarantee of user data privacy, any attack that might be mounted by Mozilla should also be addressed.

Recommendations

We recommend that the Firefox Sync client, like the Firefox Home, checks the received *g^x2* (or *g^x4*) to be not equal to 1. In order to avoid the vulnerability described in http://seb.dbzteam.org/crypto/jpake-session-key-retrieval.pdf, the received *g^x2* (or *g^x4*) must either be checked not to exceed *p* or be reduced modulo *p* immediately upon receipt (since *p*+1, 2*p*+1, etc. are also identical to 1 in the context of this calculation). Finally, the received *A* (or *B*) must also be not equal to 1.

Glossary

Security Problem Type specifies whether the reported security issue currently presents a real risk to the product's users ("Actual Security Problem") or merely has the potential to evolve into such a risk in the future ("Potential Security Problem"). An example of the former is a buffer overflow vulnerability that can be remotely exploited for executing malicious code on the user's computer. An example of the latter is the same buffer overflow vulnerability which is only present in the debug build of the product: the vulnerable code might get copied to release parts of the code in the future, or a debug build could be mistakenly dispatched to the users.

Discoverability defines the likelihood that an independent third party will discover the vulnerability in the foreseeable future using publicly obtainable information about the product (including its source code in case of an open-source product). Following the worst-case principle, it is assumed that whoever should discover this vulnerability, would either announce it to the public (e.g. on security-related mailing lists) before a fix was available, thus causing damage to vendor's reputation and putting users at risk, or enable its covert exploitation, again causing damage to vendor's customers and reputation. Discoverability is quantified between 1 (very low) and 5 (very high), where 1 means "very difficult to discover" and 5 means "trivial to discover."

Impact is the "worst-case scenario" assessment of the damage the attacker could cause by exploiting the vulnerability, regardless of what special conditions would have to be met in order for this scenario to become possible. Impact is quantified between 1 (very low impact) and 5 (very high impact).

Required access refers to the access the attacker would need to obtain in order to be able to exploit a particular vulnerability. It defines the "likelihood" of attacker actually obtaining such access, and is quantified between 1 (very low) and 5 (very high) where 1 means "very few attackers could gain such access" and 5 means "every motivated attacker could gain such access".

Configuration dependence is an assessment of the likelihood that a specific configuration described in the attack scenario would be used in a real-world production system. For example, if the attack scenario requires the usernames to be of a specific, unlikely form (e.g., containing question marks), or that two servers must have IP addresses with matching last octet (which is a 1:255 chance in average), the configuration dependence would be very high. On the other hand, if the attack scenario assumes a default configuration or a configuration proposed by the product documentation, the configuration dependence would be very low. Configuration dependence is quantified between 1 (very low) and 5 (very high), and relates closely to the "Number of users affected" metric used by some other vulnerability assessment methods.

Simplicity defines how likely it would be for an attacker to actually carry out a successful attack by exploiting the vulnerability, given the required access to the target system. This attribute also includes the difficulty of crafting any special exploit tools, using special hardware or other equipment and performing social engineering on users. Simplicity is quantified between 1 (very low) and 5 (very high), where 1 means "very difficult to exploit" and 5 means "very easy to exploit."

Cost simply describes the estimated cost of a particular attack for the attacker (per the attack scenario) - in money and other resources. A very high cost, for example, would be assigned to an attack that required cracking a DES key - although this has been proven to be possible, it still requires special costly equipment. An attack that requires sending a billion requests to a web server over a telephone line would be assigned a high cost due to communication expenses. However, most of the attacks are usually quite inexpensive, thus having "very low" or "low" cost values. Cost is quantified between 1 (very low) and 5 (very high).

Traceability defines how likely a security-trained system administrator would be able to trace the origin of the attack, once it has been detected, and therefore also defines the attacker's exposure to system owner's legal or administrative retribution. Traceability is higher when the attack leaves traces of attacker's IP address, her username, targeted information, etc. on the attacked system. Following the worst-case assumption, any external logging facilities (e.g. firewall appliances, logging routers etc.) and non-default local logging facilities are not considered in the assessment. Traceability is quantified between 1 (very low) and 5 (very high), where 1 means "very difficult to trace" and 5 means "very easy to trace."

Severity is calculated with formula: **Imp** * **AVG** (**Acc** , (6 - **Conf**) , **Simp** , (6 - **Cost**) , (6 - **Det**) , (6 - **Trc**))/5, where **Imp** = Impact, **Acc** = Required access, **Conf** = Configuration dependence, **Simp** = Simplicity, **Cost** = Cost, **Det** = Detectability, **Trc** = Traceability and **AVG** = average of arguments. Rationale behind this formula is the following: (1) Impact affects the severity proportionally: if impact were 0, the severity would also be 0 regardless of other factors; (2) Required Access, Configuration dependence, Simplicity, Cost, Detectability and Traceability are "softer" attributes, more prone to subjective opinions. Therefore their average is calculated, normalized and used as a proportional factor in severity formula. Note that Configuration dependence, Cost, Detectability and Traceability need to be inverted (subtracted from 6) in order to correctly contribute to severity. According to the possible values of each attribute used in the formula the resulting severity can have a value between 0 and 5. We round this value to one decimal place to provide meaningful differentiation between severities of the discovered vulnerabilities. Note that the resulting severity is merely an orientation source: the assessments of input values are based on our team's experience, knowledge and expectations and would almost certainly be somewhat different were they evaluated by someone else.