Guess what? Here is a new tool that finds some new guessing attacks

Ricardo Corin[1], Sreekanth Malladi[2], Jim Alves-Foss[2], and Sandro Etalle[1]

[1] Faculty of Computer Science, University of Twente,
P.O.Box 217, 7500AE Enschede,
The Netherlands. Fax - (31 53)-489-4590
{corin,etalle}@cs.utwente.nl
http://www.cs.utwente.nl/∼corin

[2] Center for Secure and Dependable Systems, University of Idaho,
Moscow, ID - 83844, USA. Fax - (208)-885-9052
{msskanth,jimaf}@cs.uidaho.edu
http://www.cs.uidaho.edu/∼msskanth

Abstract. If a protocol is implemented using a poor password, then the password can be guessed and verified from the messages in the protocol run. This is termed as a *guessing attack*. Published design and analysis efforts always lacked a general definition for guessing attacks. Further, they never considered possible type-flaws in the protocol runs or using messages from other protocols. In this paper, we provide a simple and general definition for guessing attacks. We explain how we implemented our definition in a tool based on constraint solving. Finally, we demonstrate some new guessing attacks that use type-flaws and multiple protocols which we found using our tool.

1 Introduction

Guessing attacks on protocols implemented using poor passwords are well-known [LGSN89,GLNS93]. As an example, consider the following message in a protocol:

$$a \rightarrow s \ : \ \{na\}_{passwd(a,s)}$$

(read as 'a' sends 's' a nonce 'na' encrypted with the password '$passwd(a,s)$').

An attacker can guess a password, decrypt $\{na\}_{passwd(a,s)}$ getting na' out, obtain na in another way (possibly from a different message) and compare na and na' to verify the guess. If the guess is incorrect, the attacker can try again. In this paper we only consider this type of *off-line* guessing attacks, where guesses are not used in direct communication with the server.

Past efforts to address guessing attacks in terms of design or analysis lack a general definition for guessing attacks, in the sense that one definition captures all guessing attacks and avoids case analysis. Further, they always assume that the protocols will be implemented without type-flaws and without interaction from other protocols. However, these assumptions have led to the development of succesful attacks against published protocols.

A *type-flaw* occurs when a message of one type is received by a party, interpreting it as a different type. For example, receiving an agent identity and treating it as a nonce.

We use the term *multiple protocols* to mean two or more protocols having different sets of messages. eg. SSH, SSL or EKE, IKE and so on, but not merely two different runs of the same protocol.

Some design techniques that prevent attacks involving type-flaws and multiple protocols on properties like secrecy and authentication have been reported in the literature [HLS00,GT00]. However, if these techniques are used when a protocol is using poor passwords, we have found that they may actually *facilitate* a guessing attack (we elaborate more on this in section 4).

In this paper, we address the type flaw and multiprotocol issues of guessing attacks. The main contributions are:

- A new simple and general definition for guessing attacks. This in turn helps in designing a general approach to find guessing attacks. In particular, we have extended the constraint solving technique of [MS01,CE02] to find guessing attacks, using our new definition[3];
- Demonstration of the effect of type-flaws and multiple protocols on guessing attacks, through *type-flaw guessing attacks* and *multi-protocol guessing attacks* [MAFM02].

The rest of this paper is organized as follows. In section 2 we formally define guessing attacks and illustrate the definition using examples. In section 3 we show how we implemented an analysis technique using this definition. In section 4 we discuss how conventional techniques to prevent type-flaw attacks and multi-protocol attacks actually facilitate a guessing attack. In section 5 we show some examples of type-flaw and multi-protocol guessing attacks which can exist when type-flaws and multiple protocols cannot be detected. We conclude with a discussion of related work and future work.

2 Defining guessing attacks

Lowe analyzes protocols for guessing attacks in [Low02]. His definition goes,

> A guessing attack consists of the attacker guessing a value g, and then verifying that guess in some way. The verification will be by the intruder using g to produce a value v, which we call the *verifier*; the verifier will demonstrate that the guess was correct, i.e. an incorrect guess would not have led to this value. This verification can take a number of different forms: (1) the attacker knew v initially, or has seen v during the protocol run; (2) the attacker produced v in two different ways; or (3) v is an asymmetric key, and the attacker's knows the inverse of v from somewhere.

[3] A demo of the implementation is at `http://wwwes.cs.utwente.nl/24cqet/guessing.html`

Let us leave (3) aside for the moment. We can combine (1) and (2) as follows:

The attacker produced v in two ways, and at least one of these two ways was not possible before using the guess.

Now, whether an attacker can produce v in two different ways can be determined by simply masking that occurrence of v with some fresh constant (say v') and see if he can produce v and v' again. This observation leads to our following new definition of guessing attacks.

Let $\vdash$ be the (attacker's) inference relation: $T \vdash t$ means that the attacker is able to produce the value t using his knowledge T (T is a set of terms). We use the standard Dolev-Yao inference rules as the attacker capabilities [DY83]. Let v be a subterm of a term in T and g denote a guess. Then, we say that:

Definition 1. *g is* verifiable *wrt T and v is a* verifier *for g iff:*

1. $T' \cup \{g\} \vdash v \ \wedge \ T' \cup \{g\} \vdash v'$; *and*
2. $\neg(T' \vdash v \ \wedge \ T' \vdash v')$.

where v' is a fresh constant and T' is a set of terms obtained by replacing the particular occurrence of v in T, with v'.

Below we illustrate our definition on some examples.

Example 2.1: Let $T = \{na, \{na\}_{pab}\}$; (pab is $passwd(a, b)$). Let g be pab, and pick the leftmost occurrence of na as the verifier (v). Then, $T' = \{v', \{na\}_{pab}\}$. It is straightforward to check that $T' \vdash v'$, $T' \nvdash na$ (satisfying condition 2 of Definition 1), and $T' \cup \{g\} \vdash v'$ and $T' \cup \{g\} \vdash na$ (satisfying condition 1 of Definition 1). Hence, that occurrence of na is a verifier for g and there is a guessing attack.

Example 2.2: Let $T = \{\{na\}_{pab}, \{na, nb\}_{pab}\}$. Again make $g = pab$ and $v = na$ (in $\{na\}_{pab}$). So, $T' = \{\{v'\}_{pab}, \{na, nb\}_{pab}\}$. Both na and v' are not derivable from T' (satisfying condition 2), while they are both derivable from $T' \cup \{g\}$ (satisfying condition 1). Thus, that occurrence of na is a verifier for g and there is an attack.

Example 2.3: Let $T = \{\{na\}_{pk(b)}, \{na\}_{pab}\}$. Let $g = pab$, and pick $v = \{na\}_{pk(b)}$. Then, $T' = \{v', \{na\}_{pab}\}$. Now, $T' \vdash v'$, $T' \nvdash \{na\}_{pk(b)}$ (satisfying condition 2), and $T' \cup \{g\} \vdash v'$ and $T' \cup \{g\} \vdash \{na\}_{pk(b)}$ (satisfying condition 1). Hence, $\{na\}_{pk(b)}$ is a verifier for g, and there is a guessing attack. However, consider $v = na$ (in $\{na\}_{pab}$). $T' \cup \{g\} \vdash v'$ but $T' \cup \{g\} \nvdash v$ (not satisfying condition 1). Hence, the particular v cannot be a verifier and it was a wrong choice.

The definition also implicitly includes another special case of Lowe, where the protocol itself gives $\{g\}_g$ to the attacker. In this case, we select v as g (inside the encryption). After guessing, v and v' can be obtained from $\{v'\}_g \cup \{g\}$ (satisfying condition 1), but not before guessing (satisfying condition 2).

The only case in Lowe's definition not captured by our definition is his condition (3). However, this is really an implementation dependent attack. We introduce some special predicates in our actual implementation of our definition, and generalize such implementation dependent attacks.

3 Implementation

If the number of parties executing protocols on a network is finite, then the question "Is an attack possible in this scenario?" can be answered using a simple *constraint satisfaction procedure*. The idea is to first consider all possible message interleavings of the finite number of participants. Now if there is an attack on a particular interleaving, then the intruder must be able to send all participants with the messages that they expect to receive. For this, he should be able to synthesize those messages using his knowledge and actions. These are called *constraints*. If the constraints are solvable, it means that the intruder can produce those messages and there is indeed an attack on that interleaving. This process is repeated for all the possible interleavings.

This technique of protocol analysis, called *constraint solving*, was first introduced by Millen and Shmatikov [MS01]. It is a terminating, sound and complete procedure and has been implemented in a tool called *constraint solver*. Unlike previously developed tools for protocol analysis, the constraint solver is very easy to use, available for free, simple (a three page Prolog program) and a natural way of protocol analysis.

We used an improved version of the original constraint solver called CoProVe [CE02]. Unlike the original constraint solving, CoProVe tries to construct an attack scenario 'on-the-fly', by finding a corresponding execution sequence of the agents' events. This avoids the necessity to consider many 'useless' interleavings, resulting in a much faster implementation. Moreover, the tool is capable of finding attacks involving partial runs and has a monotonic behavior.

We extended CoProVe to find guessing attacks, by implementing Definition 1. Let $m\ :\ T$ represent a constraint stating that attacker should be able to derive message m from a set of terms T. Now, in our notation, $m\ :T$ is solvable iff $T \vdash m$. Therefore, the implementation is straightforward: put the conditions in Definition 1 in a constraint form as:

1. $v\ :\ T' \cup \{g\}\ \wedge\ v'\ :\ T' \cup \{g\}$ and
2. $\neg(v\ :\ T'\ \wedge\ v'\ :\ T')$.

As explained above, we first consider a scenario with finite number of participants. We then 'execute' the scenario using constraint solving; for each possible way of executing the scenario, we obtain a knowledge that. Using the two conditions above, the knowledge can be tested for the existence of a guessing attack.

We also added to the implementation some new reduction rules that permit operations like guessing secrets, Vernam encryption, and explicit use of secret keys. In addition, we povided some new predicates, that can be specified in the input to the solver:

1. keylookup($true/false$), to turn on/off public-key lookup;
2. obtainable(t), to specify terms that the attacker knows, even before the protocol executes (for instance, from a file);
3. checkable(t), to specify terms that the attacker can immediately recognize when he sees them (e.g. words in English). This predicate combined with obtainable() allows us to take care of Lowe's condition (3).

Similar predicates can be easily added to detect more such possible implementation dependent attacks. The implementation is a five-page Prolog program. There is also an option to find only one or all the attacks on a protocol.

However, there are a couple of limitations to our implementation. Due to the free algebra assumptions in the original constraint solver, our use of Vernam encryption was restricted. It was modelled similar to symmetric key encryption, without associative and commutative properties (however, it appears that this restriction is not needed anymore, thanks to a recent work of Millen and Shmatikov [4]). Also, protocols can use explicit secret keys only as a construction operator (just like hash).

Results obtained with our tool were surprising and exciting. Apart from a number of attacks on toy protocols and known attacks (including Lowe's examples), we also found some new attacks on published protocols.

4 Type-flaw and Multi-protocol guessing attacks

As mentioned earlier, some techniques to prevent attacks involving type-flaws and multiple protocols may actually facilitate a guessing attack. In [HLS00], Heather et al. provide a method to prevent type-flaw attacks, by tagging the message fields with their intended types. However, type-tagging should not be implemented when the protocols are using poor passwords. To see why, consider again the message $\{na\}_{passwd(a,s)}$ shown in the introduction. Now, an attacker has to decrypt it with a guess, obtain na in another way and compare to verify the guess. But if the message is typed, like $\{\mathsf{nonce},\ na\}_{passwd(a,s)}$, after decryption with the guess, presence of the tag 'nonce' would itself verify the guess. He does not even need to know na!

Guttman et al. proved that attacks involving multiple protocols can be avoided if all the protocols are implemented with disjoint sets of encrypted messages [GT00]. However, if disjoint encryption is implemented by inserting 'protocol-identifiers' into messages, then the identifier itself would reveal the guess (as in the case of type-tagging). On the other hand, disjoint encryption using disjoint key sets is an expensive requirement due to the high cost of certified keys. Hence, users are unlikely to follow it.

However, in the absence of any mechanisms to detect type-flaws and use of multiple protocols, protocols may be vulnerable to new kinds of guessing attacks called, *type-flaw guessing attacks* and *multi-protocol guessing attacks* [MAFM02]. We show examples for these in the next section.

[4] Vitaly Shmatikov, private communication, Feb 2003.

4.1 Examples

Example 4.1 Consider the following protocol:

Msg 1. $a \to b : \{\{k\}_{pk(b)}, \{\{k\}_{pk(b)}\}_k\}_{pab}$
Msg 2. $b \to a : \{nb, \{k_2\}_k\}_{pab}$
Msg 3. $a \to b : \{nb\}_{k_2}$

We tested this protocol in our tool and found only one attack, involving a type-flaw. During the on-line phase of the attack, the attacker performs the following communication with a (we write $I(x)$ when the attacker impersonates honest agent x):

Msg 1. $a \to I(b) : \{\{k\}_{pk(b)}, \{\{k\}_{pk(b)}\}_k\}_{pab}$
Msg 2. $I(b) \to a : \{\{k\}_{pk(b)}, \{\{k\}_{pk(b)}\}_k\}_{pab}$
Msg 3. $a \to I(b) : \{\{k\}_{pk(b)}\}_{\{k\}_{pk(b)}}$

In message 2, attacker replays message 1 back to a.

Now the attacker moves to the off-line phase; the relevant events in the tool's output are[5]:

```
guesses:[pab], verifier: k * pk(b)
verification trace: sdec([k * pk(b),k * pk(b) + k] + pab),
split([k * pk(b),k * pk(b) + k]),sdec(k * pk(b) + k * pk(b))
```

Attack: The attacker guesses *pab*, decrypts the first message with the guess, splits it, and takes the first part ($\{k\}_{pk(b)}$) out of it. He can then decrypt the third message with this to obtain it $\{k\}_{pk(b)}$ again, thereby verifying the guess.

Example 4.2 Lomas et al. presented the following protocol in [LGSN89] (say **P1**):

Msg 1. $a \to b : \{c, n\}_{pk(b)}$
Msg 2. $b \to a : \{f(n)\}_{pab}$

We tested **P1** in our tool and found a simple attack [6]:

Msg 1. $I(a) \to b : \{c, n\}_{pk(b)}$
Msg 2. $b \to I(a) : \{f(n)\}_{pab}$

The attack trace:

```
guesses:[k], verifier: n
verification trace: guess(k), sdec(n + k).
```

Attack: The attacker spoofs as b and sends message 1 by encrypting his own c, n with b's public key. After receiving message 2, he decrypts it with a guess. Since f is known and invertible, he applies f^{-1} to $f(n)$ and obtains n. If it matches n sent in message 1, the guess was correct. (We also found a similar attack on

[5] '*' and '+' indicate asymmetric and symmetric key encryption operators respectively.

[6] We are grateful to Sarvar Patel for pointing out this possibility.

a slightly more complicated version of this protocol, presented in [GLNS93]. See Appendix).

However, we also found an attack involving a type-flaw and multi-protocols on **P1**. We changed **P1** so that the first message is $\{n,c\}_{pk(b)}$ instead of $\{c,n\}_{pk(b)}$.

Msg 1. $a \to b : \{n,c\}_{pk(b)}$
Msg 2. $b \to a : \{f(n)\}_{pab}$

Call this **P2**. When **P1** and **P2** are combined, the scenario looks like:

Msg P1.1. $a \to b : \{c,n\}_{pk(b)}$
Msg P1.2. $b \to a : \{f(n)\}_{pab}$
Msg P2.1. $I(a) \to b : \{c,n\}_{pk(b)}$
Msg P2.2. $b \to I(a) : \{f(c)\}_{pab}$

The apparently inconsequential change in message 1 now leads to an attack:

```
guesses:[pab], verifier: [c,n] * pk(b)
verification trace: sdec(c + pab), sdec(n + pab), keylookup(pk(b)),
pair([c,n]), penc([c,n] * pk(b)).
```

Attack: Attacker can replay msg 1 of **P1** in **P2**. After b sends msg 2 in **P2**, attacker now has $\{f(n)\}_{pab}$ and $\{f(c)\}_{pab}$. She can decrypt both to get $f(n)$ and $f(c)$ and apply f^{-1} to get n and c. Lastly, she can construct message 1 in **P1** or **P2** with c, n and $pk(b)$ to verify the guess.

Example 4.3 Consider another protocol presented by Lomas et al. in the same paper (say, **P3**):

Msg 1. $a \to s : \{a, b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pk(s)}$
Msg 2. $s \to b : \; a, b$
Msg 3. $b \to s : \{a, b, nb1, nb2, cb, \{tb\}_{passwd(b)}\}_{pk(s)}$
Msg 4. $s \to a : \{na1, na2 \oplus k\}_{passwd(a)}$
Msg 5. $s \to b : \{nb1, nb2 \oplus k\}_{passwd(b)}$
Msg 6. $a \to b : \{ra\}_k$
Msg 7. $b \to a : \{f1(ra), rb\}_k$
Msg 8. $a \to b : \{f2(rb)\}_k$

Here, $\oplus$ represents Vernam encryption (bitwise XOR).

We did not find any attacks on **P3** in isolation. However, we mixed it with **P1** above and found an attack. The scenario looks like:

Msg P3.1. $a \to b : \{a, b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pk(b)}$
Msg P1.1. $I(a) \to b : \{a, b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pk(b)}$
Msg P1.2. $b \to I(a) : \{b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pab}$

(we take $f(n) = n$, since f is arbitrary).

Attack: Msg 1 in **P3** is replayed to b in **P1**. If b in **P1** is the server in **P3**, b would send msg 2 in **P1**. Attacker can then decrypt it with the guess of $passwd(a)$, obtain $ca, na1, na2, \{ta\}_{passwd(a)}$ and construct msg 1 of **P3** again to verify the guess. Other similar attacks on **P3** can be found in the on-line demo.

An important point about type-flaw and multi-protocol guessing attacks is that, protocol runs need not be contemporaneous (i.e need not be executing parallely). Hence, they seem much more powerful than previously known type-flaw and multi-protocol attacks [AF98].

An interesting question is: "Are there attacks on these protocols without type-flaws and without using multiple protocols?". On some protocols that we tested, the guessing attacks we found always required a type-flaw or multiple protocols or both.

5 Conclusion

In this paper, we presented a new simple definition of guessing attacks and implemented it in a tool based on constraint solving. We also demonstrated some type-flaw and multi-protocol guessing attacks.

5.1 Related Work

Our technique and implementation have a number of advantages over Lowe's method.

1. It is a simple, yet general definition of guessing attacks avoiding a case analysis. This would be helpful in designing attack-prevention strategies;
2. In our definition, the attacker need not keep track of his derivations to ensure different ways of deriving the same term. In contrast, Lowe's definition implies that the attacker must do so;
3. Because we mark verifiers first and do a backward search, we do not "create" incorrect verifiers while finding the verifiers in different ways. Thus, we get rid of the tedious "undoes" relation in Lowe's method;
4. Due to the above three facts, our implementation turns out to be very fast;
5. It is easy to model multiple protocols in our tool and it implicitly looks for type-flaws. Lowe did not discuss the use of multiple protocols anywhere in his paper. And although Lowe's technique may find guessing attacks involving type-flaws, he does not mention it anywhere in his theory or examples. Besides, it is difficult to model type-flaws in Casper [Low98]. Further, Casper does not allow varying message lengths in a type-flaw;
6. Equipped with the obtainable() and checkable() predicates, the implementation allows to find more varieties of attacks than Lowe's.

5.2 Future work

Our definition of guessing attacks is good only when standard Dolev-Yao attacker inference rules are used. Lowe's definition seems more general in this sense, because it is independent of the attacker's inference rules. Also, we consider only subterms of the attacker's knowledge as potential verifiers, since we believe that any verifier with standard inference rules is always a subterm of the initial attacker knowledge. In contrast, Lowe considers terms that are not necessarily subterms of the initial attacker knowledge, as possible verifiers.

Consider, for instance, the inference rule $\{\{m, n\}_k \rightarrow \{m\}_k\}$, and the knowledge $T = \{\{\{m, n1\}_k\}_{passwd(a,b)}, \{\{m, n2\}_k\}_{passwd(a,b)}\}$. Now $\{m\}_k$ can be a verifier since it can be obtained in two different ways after guessing $passwd(a, b)$ [7]. However, since $\{m\}_k$ is not a subterm and since the inference rule is not in the standard set, it cannot be found using our definition. In contrast, it can be found using Lowe's definition. It would be interesting to extend our definition to make it equivalent to Lowe's so that it is independent of the inference rules, at the same time keeping it simple and general. We believe that our definition is equivalent to Lowe's when the standard inference rules are adopted. We are currently investigating this point.

It is our conjecture that having type-tags or protocol identifiers only inside terms encrypted with strong keys would prevent all type-flaw and multi-protocol guessing attacks respectively (such tagging seems to work atleast for the examples shown in this paper). Proving this conjecture formally appears to be a challenging task.

Patel presents some implementation dependent attacks on versions of EKE and other protocols [Pat97]. Some of them use redundancies such as typing inside encrypted messages. However, in all those cases, the redundancies only aid those attacks but do not form the root cause. Patel points out some solutions such as making type tags longer as possible ways of avoiding those attacks. Formalization and implementation issues regarding the use of redundancies in encrypted messages are important, but little work has been done in that direction.

There are a number of facets to password guessing attacks. Lot of effort seems to have been put into using 'provable security' [KOY01] for analysis and some effort into studying the implementation dependent issues. However, application of formal methods, which has been extensively used for cryptographic protocol analysis, was not used in a great deal for password protocols. This is probably due to the complicated nature of guessing attacks, whose analysis involves complications similar to combining cryptanalytical and abstract protocol analysis.

Acknowledgments. We are grateful to the anonymous referees of this paper whose comments helped us correct some critical errors. This research was funded in part by (i) DARPA under grant F30602-02-1-0178 and (ii) LicenseScript project.

References

[AF98] J. Alves-Foss. Multi-protocol attacks and the public key infrastructure. In *Proc. National Information Systems Security Conference*, pages 566–576, October 1998.

[7] We are grateful to an anonymous reviewer for pointing this out to us.

[CE02] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. *9th Int. Static Analysis Symp. (SAS)*, LNCS 2477:326–341, September 2002.
[DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.
[GLNS93] L. Gong, T. M. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.
[GT00] J. D. Guttman and F. J. Thayer. Protocol independence through disjoint encryption. *13th IEEE Computer Security Foundations Workshop*, pages 24–34, July 2000.
[HLS00] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings, 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, July 2000.
[KOY01] J. Katz, R. Ostrovsky, and M Yung. Practical password-authenticated key exchange provably secure under standard assumptions. In *Advanced in Cryptology-EUROCRYPT*, 2001.
[LGSN89] T. M. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham. Reducing risks from poorly chosen keys. *Operating Systems Review*, 23(5):14–18, 1989.
[Low98] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.
[Low02] G. Lowe. Analyzing protocols subject to guessing attacks. *Workshop on Issues in the Theory of Security (WITS'02)*, January 2002.
[MAFM02] S. Malladi, J. Alves-Foss, and S. Malladi. What are multi-protocol guessing attacks and how to prevent them. *Proceedings of Eleventh IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2002*, June 2002.
[MS01] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communication Security*, volume Proc. 2001, pages 166–175. ACM press, 2001.
[Pat97] S. Patel. Number theoretic attacks on secure password schemes. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, 236–247, 1997.

Appendix 1. Attack on another version of P1

We presented a previously unpublished attack on **P1** in Section 4.1 (Example 4.2). Here we present a similar attack on a slightly modified version of **P1** presented in [GLNS93]. The protocol is described as follows,

Msg 1. $a \to b : \{c1, c2, n\}_{pk(b)}$
Msg 2. $b \to a : \{c2 \oplus f(n)\}_{pab}$

The attack scenario,

Msg 1. $I(a) \to b : \{c1, c2, n\}_{pk(b)}$
Msg 2. $b \to I(a) : \{c2 \oplus f(n)\}_{pab}$

Attack: The attacker spoofs as b and sends message 1 by encrypting his own $c1, c2, n$ with b's public key. After receiving message 2, he constructs $\{c2 \oplus n\}_g$ with a guess, since he knows $c2$ and n. A match with message 2 verifies the guess.