

How to prevent type-flaw guessing attacks on password protocols*

Sreekanth Malladi and Jim Alves-Foss
Center for Secure and Dependable Systems
University of Idaho
Moscow, ID - 83844
{msskanth, jimaf}@cs.uidaho.edu
<http://www.cs.uidaho.edu/~msskanth>

Abstract

A message in a protocol is said to have a type-flaw if it was created with some intended type, but is later received and treated as a different type. A type-flaw guessing attack is an attack where a password is guessed and verified by inducing type-flaws in a password protocol. Heather et al. [HLS00] prove that attacks that use type-flaws can be prevented if honest agents tag messages with their intended types. However, they do not consider type-flaw “guessing attacks” in their proofs. Further, their tagging scheme cannot be used in a password protocol since it allows a guess to be directly verified using the tags inside password encryptions. In this paper we prove that, by following a modification of Heather et al.’s scheme, type-flaw guessing attacks can still be prevented.

1 Introduction

It is well-known that password protocols can be subject to *guessing attacks* where a poorly chosen password is guessed and verified. As an example, consider the following message in a protocol:

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

(read as a sends b a nonce na and na encrypted with the shared password $passwd(a, b)$)

Now, if the user name is “Arnold Schwarzenegger”, “terminator” wouldn’t be a bad guess for $passwd(a, b)$. An attacker can encrypt na with the guess (like $\{na\}_{terminator}$), and compare it to $\{na\}_{passwd(a,b)}$. If they match, the guess was correct. Otherwise, she can make another guess and repeat the process.

Recently, in joint work with Ricardo Corin and Sandro Etalle, we developed a new tool to find guessing attacks and demonstrated some new guessing attacks that use type-flaws called *type-flaw guessing attacks* [CMAFE03]¹. As an example, consider the following protocol **P1** presented in [GLMS93]:

$$\begin{aligned} \text{Msg 1. } a \rightarrow b &: \{c, n\}_{pk(b)} \\ \text{Msg 2. } b \rightarrow a &: \{f(n)\}_{passwd(a,b)}. \end{aligned}$$

where c is a redundant random number called “confounder”; n is an integer value; f is an easily invertible function; and $pk(b)$ is the public-key of b .

Let **P1** be implemented so that the first message is $\{n, c\}_{pk(b)}$ instead of $\{c, n\}_{pk(b)}$. Call this **P2**:

$$\begin{aligned} \text{Msg 1. } a \rightarrow b &: \{n, c\}_{pk(b)} \\ \text{Msg 2. } b \rightarrow a &: \{f(n)\}_{passwd(a,b)}. \end{aligned}$$

*A preliminary version of this paper appeared in a workshop without published proceedings (FCS’03).

¹On-line demo of tool at <http://wwwes.cs.utwente.nl/24cget/guessing.html>

The apparently inconsequential change in message 1 leads to an attack when **P1** and **P2** are combined:

On-line phase: Msg P1.1. $a \rightarrow b : \{c, n\}_{pk(b)}$
 Msg P1.2. $b \rightarrow a : \{f(n)\}_{passwd(a,b)}$
 Msg P2.1. $I(a) \rightarrow b : \{c, n\}_{pk(b)}$
 Msg P2.2. $b \rightarrow I(a) : \{f(c)\}_{passwd(a,b)}$.

Off-line phase: guesses: [passwd(a,b)], verifier: [c,n] * pk(b)
 verification trace: sdec(c + passwd(a,b)), sdec(n + passwd(a,b)),
 keylookup(pk(b)), pair([c,n]), penc([c,n] * pk(b)).

Attack: In the on-line phase, attacker can replay msg 1 of **P1** in msg 1 of **P2** inducing a type-flaw. After b sends msg 2 in **P2**, attacker now has $\{f(n)\}_{passwd(a,b)}$ and $\{f(c)\}_{passwd(a,b)}$. In the off-line phase, she can decrypt both $\{f(n)\}_{passwd(a,b)}$ and $\{f(c)\}_{passwd(a,b)}$ using a guess for $passwd(a,b)$ 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.

Heather et al. in [HLS00] proved that attacks involving type-flaws can be prevented if all messages are tagged with their types. However, Heather et al. did not consider “guessing attacks” that use type-flaws (but only attacks on secrecy and authentication). Further, even if Heather et al.’s type-tagging scheme can prevent type-flaw guessing attacks, there is another problem. Consider the message:

$$\{na\}_{passwd(a,b)}$$

If this message is type-tagged following Heather et al.’s scheme of type-tagging as,

$$\{\text{nonce}, na\}_{passwd(a,b)}$$

the attacker can decrypt it with a guess and see if there is the tag “nonce” in it. If so, that would directly verify the guess. He doesn’t even need to know $na!$ ². Therefore, Heather et al.’s solution against type-flaw attacks cannot be used in password protocols.

In this paper, we address this problem by modifying the tagging scheme. We prove that by following Heather et al.’s scheme, but avoiding type-tags inside terms encrypted with passwords, type-flaw guessing attacks can still be prevented. Further, Heather et al.’s scheme also cannot be used “as is” in password protocols to prevent attacks on secrecy and authentication. We will have more to say about this aspect in the Conclusion.

2 Proof Strategy

We introduce a modified version of Heather et al.’s tagging scheme that prevents all type-flaw guessing attacks and does not add redundancy that enables normal guessing attacks. We prove this claim following a model and proof structure very similar to Heather et al. [HLS00].

Our main aim is to prove the following:

Whenever there is a guessing attack on a protocol using our tagging scheme, there is an equivalent guessing attack when there are no type-flaws in the protocol.

Therefore, we prove that the attack does not utilize type flaws:

Whenever there is a guessing attack on a protocol using our tagging scheme, there is an equivalent guessing attack when all fields are correctly tagged.

An off-line guessing attack is characterized by two factors:

- The protocol run (an attacker can actively participate in the protocol run, inducing type-flaws, but doesn’t use a guess);

²This observation was originally made by Lowe [Low03].

- Guessing a key and making inferences from the set of messages in the protocol run that enable verification of the guess.

Therefore, in order to prove our main claim, we need to prove two things:

1. If an attacker participates in a protocol run C that uses our tagging scheme, then an equivalent protocol run C'' can be visualized in which every field is correctly tagged;
2. If the attacker can verify a guess from the set of messages in C , then he can also verify a guess from C'' .

We use the main result from [HLS00] for point 1 above. Our only modification to their model is the following: We consider all weak encryptions (terms encrypted with passwords) as if they were just another type of atomic element such as nonce, agent etc. We introduce our modified protocol model and state the main result of [HLS00] which still holds after our modification. This is covered in section 3.

For point 2 above, we use the definition for guessing attacks from [CMAFE03] and show that, whenever a guess is verifiable from C , it is also verifiable from C'' . This is covered in section 4.

3 Proof Part 1: Heather et al.'s Protocol Model and Main Result

In this section we reiterate the model and main results of Heather et al.'s [HLS00] tagging scheme in the context of our modification.

3.1 Message Structure

3.1.1 Tags, Facts and Taggedfacts

The main message element is a *taggedfact*. It is a combination of a *tag* and *fact*, written as $(tag, fact)$. The idea is that the tag represents the “type” of the fact. The message grammar is as follows:

$$\begin{aligned}
 TaggedFact & ::= Tag \times Fact \\
 Tag & ::= agent \mid nonce \mid wenc \mid \dots \mid pair \mid enc \ Tag^* \ Tag \\
 Fact & ::= Atom \mid PAIR \ TaggedFact \ TaggedFact \mid ENCRYPT \ Tag \ TaggedFact \ Fact
 \end{aligned}$$

Message structures are divided into atoms, pairs and encryptions. An atom is an indivisible element. Sets of atoms are grouped together as *Agent*, *Nonce*, *Pubkey* and so on. The tags for elements of these sets are given obvious names such as agent, nonce etc. An atomic fact a of type “agent”, associated with the corresponding tag agent is written as $(agent, a)$. In our modification we add the set *Wenc* to *Atoms* to represent “weak encryptions”. The corresponding tag is “wenc”, treating weak encryptions as an “abstract type”. We will talk more about this set as we progress in the paper.

A pair tag is associated with concatenation of two tagged facts. The tag enc is associated with encryptions together with the collection of tags for the elements inside the encryption and a tag for the key. The pairing, PAIR $tf_1 \ tf_2$ is written as (tf_1, tf_2) . When this is associated with its corresponding tag, “pair”, this is written as $(pair, (tf_1, tf_2))$. PAIR PAIR $tf_1 \ tf_2 \ tf_3$ should actually be $((tf_1, tf_2), tf_3)$; but it is simply written as (tf_1, tf_2, tf_3) in order to avoid notational clutter, since it is unambiguous.

A tagged fact tf encrypted with a key k using an algorithm kt is written as $\{tf\}_k^{kt}$. A tag for an encryption, going by the grammar, would look like $enc \ \langle t_1, \dots, t_n \rangle \ kt$ where t_1, \dots, t_n are the collection of tags for the facts inside the encryption and kt is the tag for the key. This tag is written in a simpler notation as $\{[t_1, \dots, t_n]\}_{kt}$. It is assumed that the tag for the key contains enough information regarding the type of the key (public-key or shared-key etc.) and the encryption algorithm used (RSA, DES etc.)

Although we treat weak encryptions as an atomic type, we do consider them to have an “internal structure” for the fields inside the encryption as below:

$$\begin{aligned}
 SubWenc & ::= Atom \mid PAIR \ Subwenc \ Subwenc \mid ENCRYPT \ Tag \ TaggedFact \ Fact \\
 Wenc & ::= ENCRYPT \ Subwenc \ WeakKey
 \end{aligned}$$

By defining such a structure, we imply that no top-level or unencrypted fact inside a weak encryption is associated with a tag. We will call the set of all such facts as *Subwenc*. We assume that honest agents follow such a structure before encrypting with a weak key (fairly realistic since otherwise, as explained before, the tags themselves would verify a guess).

We split keys into sets called *Strongkeys* and *Weakkeys*, depending on the application of the function to generate the keys. For example, application of *Passwd* gives a weak key. In contrast a function *PublicKey* gives rise to a strong key. We will talk more about function applications in section 3.2. We denote a fact $f \in \text{Subwenc}$, encrypted with a weakkey $w \in \text{Weakkeys}$ as $\{f\}_w$.

Projections are defined on tagged facts as:

$$(t, f)_1 \hat{=} t, (t, f)_2 \hat{=} f.$$

A version of the perfect encryption assumption is assumed for strong encryptions, whereby honest agents are capable of knowing if they decrypted an encryption correctly [MCJ97] (called “explicit redundancy” in [Gon90]).

We do not assume any sort of explicit redundancy inside weak encryptions. Usually, the required redundancies for those encryptions is provided by the protocol itself (called “implicit redundancy” [Gon90]).

3.1.2 Subtaggedfacts

In the following definition, we introduce the *subfact* relation denoted by ‘ \sqsubset ’ to refer to *subtaggedfacts* of a tagged fact.

Definition 1. The *subfact* relation is the smallest relation on tagged facts such that:

1. $tf \sqsubset tf$;
2. $tf \sqsubset (t, (tf_1, tf_2))$ iff $tf \sqsubset tf_1 \vee tf \sqsubset tf_2$;
3. $tf \sqsubset (t, \{tf'\}_k)$ iff $tf \sqsubset tf'$.

Such a relation is also lifted to refer to sub-untagged-facts of a tagged fact. i.e. $f \sqsubset tf$ if $(t, f) \sqsubset tf$ for some tag t .

3.1.3 Correct Tagging

A tagged fact is said to be correctly tagged if its tag represents the true type of the associated fact. A function “well-tagged” is defined inductively over the structure of tags to represent correct tagging:

$$\begin{aligned} \text{well-tagged}(\text{agent}, x) &\Leftrightarrow x \in \text{Agent}, \\ \text{well-tagged}(\text{nonce}, x) &\Leftrightarrow x \in \text{Nonce}, \\ \text{well-tagged}(\text{wenc}, x) &\Leftrightarrow x \in \text{Wenc}, \\ &\dots \\ \text{well-tagged}(\text{pair}, x) &\Leftrightarrow \exists tf_1 tf_2 : \text{TaggedFact} . \\ &\quad x = \text{PAIR } tf_1 tf_2 \wedge \text{well-tagged } tf_1 \wedge \text{well-tagged } tf_2, \\ \text{well-tagged}(\{ts\}_{kt}, x) &\Leftrightarrow \exists tf : \text{TaggedFact}; \\ &\quad k : \text{Fact} . x = \{tf\}_k^{kt} \wedge \text{well-tagged}(tf) \\ &\quad \wedge \text{well-tagged}(kt, k) \wedge ts = \text{get-tags } tf. \end{aligned}$$

where *get-tags* returns the collective sequence of tags inside an encryption, defined as:

$$\begin{aligned} \text{get-tags}(\text{pair}, (tf_1, tf_2)) &= \text{get-tags } tf_1 \hat{\ } \text{get-tags } tf_2, \\ \text{get-tags}(t, f) &= \langle t \rangle, \text{ for } t \neq \text{pair}. \end{aligned}$$

A well-tagged fact represents a taggedfact which is correctly tagged and has every subtaggedfact in it, correctly tagged. In contrast, a fact is characterized as top-level-well-tagged when it is correctly tagged at the outer-most level. This means, for example, a taggedfact is indeed a pair of tagged facts when its tag equals *pair*, even if the two tagged facts may not be well-tagged.

$$\begin{aligned}
\text{top-level-well-tagged}(\text{agent}, x) &\Leftrightarrow x \in \text{Agent}, \\
\text{top-level-well-tagged}(\text{nonce}, x) &\Leftrightarrow x \in \text{Nonce}, \\
\text{top-level-well-tagged}(\text{wenc}, x) &\Leftrightarrow x \in \text{Wenc}, \\
&\dots
\end{aligned}$$

$$\begin{aligned}
\text{top-level-well-tagged}(\text{pair}, x) &\Leftrightarrow \exists tf_1, tf_2 : \text{TaggedFact} . x = \text{PAIR } tf_1 \text{ } tf_2, \\
\text{top-level-well-tagged}(\{|ts|\}_{kt}, x) &\Leftrightarrow \exists tf : \text{TaggedFact}; k : \text{Fact} . x = \{tf\}_k^{kt} \wedge ts = \text{get-tags } tf.
\end{aligned}$$

3.2 The framework

In the previous section, the structure of messages in a protocol and their properties were introduced. In this section, we introduce the framework on which messages are used to build protocol runs.

The framework is derived from the strand space model of [FHG99]. A *strand* is a sequence of communications represented as $\langle \pm tf_1, \pm tf_2, \dots, \pm tf_n \rangle$. $+tf$ indicates sending tf and $-tf$ indicates receiving tf . Each send or receive event is a *node*. A transition from consecutive nodes n_i and n_{i+1} on the same strand is represented as $n_i \Rightarrow n_{i+1}$. A transmission of a tagged fact from n_i on one strand, followed by a reception in n_j on another strand is represented as $n_i \rightarrow n_j$.

A *bundle* represents a partial or complete protocol run. It is an acyclic digraph using edges \rightarrow and \Rightarrow such that, whenever a tagged fact is received, the bundle also includes a transmission of the tagged fact. Further, a bundle holds the history of the network from the start of the communication.

A node is said to be an *entry point* to a set of tagged facts if no previous node has uttered an element of that set. A taggedfact is said to be *originating* on a node if the node is an entry point for the set to which the taggedfact belongs. A taggedfact is said to be *uniquely originating* if there is no other node in the bundle that utters an element of the set to which the tagged fact belongs.

3.3 Honest strands

Honest strands represent execution traces of honest agents. Since roles of honest agents are dictated by the protocol (in terms of sending and receiving messages), it makes sense to have some set of “templates” that dictate the actions of those roles in the protocol. Therefore strand templates are defined which specify the message structure of honest agents under ideal conditions. These contain variables that would be instantiated to generate honest strands.

Each taggedfact in an honest strand corresponds to an instantiation of a “tagged template” in a strand template. Tagged templates are defined by the following grammar:

$$\begin{aligned}
\text{TaggedTemplate} &::= \text{Tag} \times \text{Template} \\
\text{Template} &::= \text{Var} \mid \text{APPLY } F_n \text{ Var}^* \mid \text{PAIR } \text{TaggedTemplate} \text{ TaggedTemplate} \mid \\
&\quad \text{ENCRYPT } \text{Tag} \text{ TaggedTemplate} \text{ TaggedTemplate}
\end{aligned}$$

Here Var represents atomic variables, which upon instantiation output atomic facts. $\text{APPLY } F_n \text{ Var}^*$ means that a function identifier F_n is being applied to a collection of atomic variables. This application is the basis to generate keys, hashes of messages etc. For example, in $\text{PublicKey}(A)$, $F_n = \text{PublicKey}$. Note that this specification allows to model constructed keys, not just atomic keys, which is important for ‘real-world’ protocols such as SSL 3.0. (Atomic keys refer to the keys possessed by participants which are handled by exhaustive substitution of agents’ identities. Constructed keys are keys that are produced from just about any random bitstring formed using different message elements).

The next step is to consider how tagged templates are instantiated to form taggedfacts. This is accomplished by defining an instantiation function sub to substitute facts for variables:

$$\text{sub} : \text{Var} \rightarrow \text{Fact}$$

The properties of this function are defined below in order to instantiate all possible tagged templates:

$$\begin{aligned}
sub(t, v) &= (t, sub(v)) \text{ for } v \in Var, \\
sub(t, g(v_1, \dots, v_n)) &= (t, g(sub(v_1), \dots, sub(v_n))), \\
&\text{where } g \in Fn, \text{ and } Fn \text{ is the} \\
&\text{set of function identifiers.} \\
sub(pair, (tt_1, tt_2)) &= (pair, (sub(tt_1), sub(tt_2))), \\
sub(\{|ts|\}_{tk}, \{tt\}_k) &= (\{|ts|\}_{tk}, \{sub(tt)\}_{sub(tk, k_2)}), \\
&\text{where } k = g(v_1, \dots, v_n) \text{ and } g \in Fn \text{ represents a key} \\
&\text{type using a particular keying algorithm.}
\end{aligned}$$

For the third and fourth clauses above, there is a little change from the same expressions given in [HLS00]. (They use tf_1, tf_2 and tf in place of tt_1, tt_2 and tt . However, since sub is an instantiation of variables and not facts, we feel it is proper to apply it on templates instead of facts. This change however, wouldn't affect their results in any way).

There are two assumptions on strand templates and instantiating templates:

Assumption 1. For every strand template, there is some ideal tag environment ρ defined as:

$$\rho : (Var \rightarrow Tag) \cup (Fn \rightarrow Tag^* \times Tag)$$

The idea is that ρ returns the tags for each variable in a template. This is to ensure that the same tags are always given to the same variables in a template. (For the exact properties of ρ , please refer [HLS00].)

Assumption 2. If a taggedfact tf originates on a honest strand, then top-level-well-tagged(tf).

This means, it is assumed that honest agents always tag messages correctly. However, since it is impossible to distinguish between random bitstrings, it is probably more appropriate to say, whenever a bitstring is substituted for a variable next to a tag in a template, then the bitstring is automatically added to the set corresponding to that tag. (For example instantiating N_A in (nonce, na) would result in N_A being added to the set $Nonce$.) The bitstring is treated to be of that type from then onwards.

3.4 Penetrator strands

The penetrator is considered to have standard Dolev-Yao attacker capabilities [DY83], i.e. she can overhear messages on a network, construct messages, split them, send her own messages and so on. She is also assumed to possess some set K_P of keys and produce some texts T of her choice. These capabilities are listed in the following definition.

Definition 2. An *on-line penetrator strand* is one of the following:

M	<i>Text message</i>	$\langle +(t, x) \rangle$ with well-tagged(t, x) and $x \in T$.
F	<i>flushing</i>	$\langle -tf \rangle$.
T	<i>Tee</i>	$\langle -tf, +tf, +tf \rangle$.
C	<i>Concatenation</i>	$\langle -tf, -tf', +(\text{pair}, (tf, tf')) \rangle$.
S	<i>Separation</i>	$\langle -(\text{pair}, (tf, tf')), +tf, +tf' \rangle$.
K	<i>Key</i>	$\langle +(tk, k) \rangle$ with well-tagged(tk, k) and $k \in K_P$.
E	<i>Encryption</i>	$\langle -(tk, k), -tf, +(\{ ts \}_{tk}, \{tf\}_k^k) \rangle$, where $ts = \text{get-tags}(tf)$.

- D** *Decryption* $\langle -(tk', k'), -(\{|t|\}_{tk}, \{tf\}_k^{tk}), +tf \rangle$,
 where tk and tk' are tags representing inverse key types,
 and k' is the corresponding decrypting key of k
 with both being of the type tk and tk' respectively.
- R** *Retagging* $\langle -(t, f), +(t', f) \rangle$.

The retagging strand captures the concept of receiving a message of one type and sending it, with a claim of a different type. In this paper, we will denote the set of on-line penetrator strands as X_{on} . In Section 4 we will add some “off-line” strands to the above capabilities to model off-line guessing attacks.

Note that we treat weak encryptions as an “abstract type”. This is a major modification in our paper. We do not allow the attacker to perform any operations on it during the on-line communication, i.e. we assume that guessing the password and deducing the contents inside the encryption is done entirely off-line. Lastly, we consider only those attacks in which the attacker is able to learn a password shared by honest agents by attempting an off-line guessing attack. In other words, we do not consider attacks wherein a password is learned by breaching secrecy.

3.5 Transforming arbitrarily tagged bundles to well-tagged bundles

An arbitrarily tagged bundle represents a bundle with or without type-flaws. Since a tag in a taggedfact indicates the type of it’s fact, a correctly tagged fact indicates that the fact is *indeed* the type indicated by it’s tag. Generally speaking, a well-tagged bundle represents that all it’s tagged facts are correctly tagged. This in turn means that there are no type-flaws in a well-tagged bundle. The main result in [HLS00] states that any bundle that uses the tagging scheme can be changed into an equivalent well-tagged bundle.

To prove this hypothesis, Heather et al. define a *renaming function* that changes any arbitrarily tagged bundle to a well-tagged bundle. The main idea behind such a transformation being possible is that, if an honest agent is willing to accept an ill-tagged fact (t, f) , then it should accept any value in place of f . Naturally, this includes the fact f' such that $\text{well-tagged}(t, f')$.³

Below is the definition and properties of the renaming transformation:

Definition 3.

$$\phi : \text{TaggedFact} \rightarrow \text{TaggedFact}$$

is a *renaming function* having the following properties:

1. ϕ preserves top-level tags:

$$\phi(t, f) = (t', f') \Rightarrow t = t';$$

2. ϕ returns well-tagged terms: $\text{well-tagged}(\phi(tf))$;

3. ϕ is the identity function over well-tagged terms:

$$\text{well-tagged}(tf) \Rightarrow \phi(tf) = tf;$$

4. ϕ distributes through concatenations that are top-level-well-tagged:

$$\phi(tf_1, tf_2) = (\phi(tf_1), \phi(tf_2));$$

5. ϕ distributes through encryptions that are top-level-well-tagged:

$$\phi(\{|ts|\}_{kt}, \{tf\}_k^{tk}) = (\{|ts|\}_{kt}, \{\phi(tf)\}_{\phi(tk, k)_2}^{tk}) \text{ if } ts = \text{get-tags}(tf);$$

6. ϕ respects inverses of keys: if (tk, k) and (tk', k') are inverses of each other, then so are $\phi(tk, k)$ and $\phi(tk', k')$, tk and tk' being their types;

³There seems to be a typo in [HLS00] in stating the same.

7. When ϕ is applied to a top-level-ill-tagged fact (t, f) of C , such that $\phi(t, f) = (t, f')$, then $f' \in T$;
8. When ϕ is applied to a top-level-ill-tagged fact tf of C , it produces a fact that has an essentially new value. i.e., a fact that has no sub-untagged-fact in common with $\phi(tf')$ for any other fact tf' of C :

$$\forall tf \in facts(C) . \neg \text{top-level-well-tagged}(tf) \wedge f \sqsubset \phi(tf) \Rightarrow \forall tf' \in facts(C) . tf \not\sqsubset tf' \Rightarrow f \not\sqsubset \phi(tf').$$

where $facts(C)$ represents all the facts and sub-untagged-facts of nodes in C .

This establishes an injectivity property for ϕ over facts of C .

Merely defining such a renaming transformation neither proves that all possible taggedfacts in C are covered by ϕ nor proves that the $\phi(C)$ is a bundle by definition. Therefore, there are essentially four things to be proven:

1. Given a bundle C , there is some renaming function ϕ for C . (Refer [HLS00, Lemma 3].)
2. If $temp$ is a template for an honest agent and $sub(temp)$ is an instantiation of the template, then $\phi(sub(temp))$ corresponds to an instantiation of the same template using some other function sub' . i.e.

$$\phi(sub(temp)) = sub'(temp).$$

This means if $sub(temp)$ is an honest strand, then $sub'(temp)$ is also an honest strand. (Refer [HLS00, Lemma 4].)

3. The penetrator is “equally capable” in C and $\phi(C)$. In other words, if X is a penetrator strand in C , then X is also a penetrator strand in $\phi(C)$ with every tagged tf in X replaced by $\phi(tf)$. (Refer [HLS00, Section 3.3].)
4. Protocol security is entirely based on values that originate uniquely, such as nonces and short term keys. Therefore, it is important to ensure that the transformed bundle doesn’t contain nodes that “duplicate” such values. To this end, a bundle C'' is produced from $\phi(C)$ such that, facts in C'' are uniquely originating if they were uniquely originating in C . (Refer [HLS00, section 3.4].)

Since our modification only defines a new subset of atoms, the proofs for the above points presented in Heather et al. [HLS00] still hold.

3.6 Main Result of [HLS00]

The main result of Heather et al. ([HLS00, Theorem 1]) follows from the concepts explained in the previous section:

Theorem 1. If C is a bundle (under the tagging scheme) then there is a renaming function ϕ and a bundle C'' , such that:

- C'' contains the tagged facts of C (considered as a set), renamed by ϕ ;
- C'' contains the same honest strands as C , modulo some renaming;
- facts are uniquely originating in C'' if they were uniquely originating in C ;
- all tagged facts in C'' are well-tagged.

4 Proof part 2 : Off-line guessing attacks

4.1 Off-line penetrator capabilities

In this section we will introduce our notion of an attacker engaging in off-line guessing and verification. We assume a set G of guesses that a penetrator possesses. In the off-line phase, a penetrator can guess a password, use it to encrypt and decrypt weak encryptions, concatenate and split facts, tag facts and untag tagged facts. We add some new penetrator strands (in addition to the capabilities in Definition 2) to capture such off-line capabilities:

\mathbf{D}_g	<i>Decryption_using_Guess</i> $\langle - \{f\}_g , -g, +f \rangle$ with $g \in G$.
\mathbf{E}_g	<i>Encryption_using_Guess</i> $\langle -f, -g, + \{f\}_g \rangle$ with $g \in G$.
\mathbf{C}_f	<i>Concatenating_facts</i> $\langle -f, -f', +(f, f') \rangle$.
\mathbf{S}_f	<i>Separating_facts</i> $\langle -(f, f'), +f, +f' \rangle$.
\mathbf{T}_g	<i>Tagging</i> $\langle -t, -f, +(t, f) \rangle$.
\mathbf{Utg}	<i>Untagging</i> $\langle -(t, f), +f \rangle$.

Note that in the off-line phase, the penetrator can still use some of his on-line capabilities. However, since there wouldn't be any network events during the off-line phase, we will remove the corresponding strands ($\mathbf{M}, \mathbf{F}, \mathbf{T}, \mathbf{K}, \mathbf{R}$) from X_{on} before adding them to off-line capabilities:

Definition 4. The set of *off-line penetrator strands*, denoted as X_{off} is defined as:

$$X_{\text{off}} = X_{\text{on}} \setminus \{\mathbf{M}, \mathbf{F}, \mathbf{T}, \mathbf{K}, \mathbf{R}\} \cup \{\mathbf{D}_g, \mathbf{E}_g, \mathbf{C}_f, \mathbf{S}_f, \mathbf{T}_g, \mathbf{Utg}\}$$

or

$$X_{\text{off}} = \{\mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{D}, \mathbf{D}_g, \mathbf{E}_g, \mathbf{C}_f, \mathbf{S}_f, \mathbf{T}_g, \mathbf{Utg}\}.$$

We now introduce a simple inference relation \vdash . If S is a set of tagged facts, we write $S \vdash_X tf$ if the strand X can be constructed such that, for every $tf' \in S$, tf' occurs on a '-' node in X , and tf is a tagged fact on any '+' node of X . For example, a \mathbf{C}_f strand, $\langle -na, -nb, +(na, nb) \rangle$ is written as $\{na, nb\} \vdash_{\mathbf{C}_f} (na, nb)$.

Using \vdash , we define a relation *deducible* such that, tf is deducible from a bundle C , if there is a valid sequence of penetrator strands that yield tf from C .

Definition 5. Let C be a bundle. Then, tf_n is *deducible* from C , or:

$$C \models_{tr} tf_n$$

if $tr = \langle S_1 \vdash_{X_1} tf_1, S_2 \vdash_{X_2} tf_2, \dots, S_n \vdash_{X_n} tf_n \rangle$ such that, for $i = 1 \dots n$, $X_i \in X_{\text{off}} \setminus \{\mathbf{D}_g, \mathbf{E}_g\}$ and $S_{i+1} \subseteq \text{Taggedfacts}(C) \cup \{tf_1, \dots, tf_i\}$, where $\text{Taggedfacts}(C)$ is the set of taggedfacts on all the nodes in C .

Example: We write

$$\{na, nb, g\} \models_{\langle \{na, nb\} \vdash_{\mathbf{C}_f} (na, nb), \{na, nb, g\} \vdash_{\mathbf{E}_g} \{(na, nb)\}_g \rangle} \{(na, nb)\}_g.$$

when $\{(na, nb)\}_g$ is deducible from $\{na, nb, g\}$.

We will tend to drop the subscript tr when it is obvious. Note that guesses will not be present in C , and have to be added to construct \mathbf{D}_g and \mathbf{E}_g strands from C .

We now prove a useful lemma from which follows a useful corollary.

Lemma 1. Let C and C'' be two bundles defined as in section 3. Then,

$$C \cup \{g\} \models_{tr} tf \Rightarrow C'' \cup \{g\} \models_{\phi(tr)} \phi(tf).$$

Proof. We need to show that, for every possible inference $S \vdash_X tf$ in tr , there is an equivalent $\phi(S) \vdash_{\phi(X)} \phi(tf)$ in $\phi(tr)$. This inturn implies we need to show that for every possible strand $X \in X_{\text{off}}$ from C , there is an equivalent strand from C'' such that, $\phi(X) \in \phi(X_{\text{off}})$.

It is proven in [HLS00, section 3.3] that for each of the penetrator strands in C , equivalent penetrator strands in C'' can be constructed. Therefore, for every $X \in \{\mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{D}\}$, there is a corresponding $\phi(X) \in \{\phi(\mathbf{C}), \phi(\mathbf{S}), \phi(\mathbf{E}), \phi(\mathbf{D})\}$.

We now prove that for every $X \in \{\mathbf{D}_g, \mathbf{E}_g, \mathbf{C}_f, \mathbf{S}_f, \mathbf{T}_g, \mathbf{Utg}\}$ there is a corresponding $X' = \phi(X)$ such that, $X' \in \{\phi(\mathbf{D}_g), \phi(\mathbf{E}_g), \phi(\mathbf{C}_f), \phi(\mathbf{S}_f), \phi(\mathbf{T}_g), \phi(\mathbf{Utg})\}$. For the items below, note that $\phi(t, f)$ is defined only when $f \in \text{Facts}$ (by definition). When $f \notin \text{Facts}$ (for example, if $f \in \text{SubWenc}$), then we replace $\phi(t, f)$ with f itself.

- If X is a \mathbf{D}_g strand, then

$$X' = \langle -\phi(\text{wenc}, |\{f\}_g|)_2, -g, +f \rangle,$$

which is a \mathbf{D}_g strand because, when $(\text{wenc}, |\{f\}_g|)$ originates on a regular node in C , then $\text{well-tagged}(\text{wenc}, |\{f\}_g|)$ (by Assumption 2) and therefore, $\phi(\text{wenc}, |\{f\}_g|) = (\text{wenc}, |\{f\}_g|)$.

- If X is a $\mathbf{E_G}$ strand, then

$$X' = \langle -\phi(t, f)_2, -g, +|\{\phi(t, f)_2\}_g| \rangle,$$

which is an $\mathbf{E_G}$ strand. (Note that $\phi(t, f)$ is not defined for all $f \in \text{Subwenc}$, in which case, we replace $\phi(t, f)_2$ with f itself.)

- If X is a $\mathbf{C_f}$ strand, then

$$X' = \langle -\phi(t, f)_2, -\phi(t, f')_2, +(\phi(t, f)_2, \phi(t, f')_2) \rangle,$$

which is a $\mathbf{C_f}$ strand.

- If X is a $\mathbf{S_f}$ strand, then

$$X' = \langle +(\phi(t, f)_2, \phi(t, f')_2), +\phi(t, f)_2, +\phi(t, f')_2, \rangle,$$

which is a $\mathbf{S_f}$ strand.

- If X is a \mathbf{Tg} strand, then

$$X' = \langle -t, -\phi(t, f)_2, +\phi(t, f) \rangle.$$

Now $\phi(t, f) = (t, f')$ for some f' such that $\text{well-tagged}(t, f')$. Therefore, we can rewrite the above expression as $X' = \langle -t, -f', +(t, f') \rangle$, which is a \mathbf{Tg} strand.

- If X is a \mathbf{Utg} strand, then

$$X' = \langle -\phi(t, f), +\phi(t, f)_2 \rangle.$$

Again, since $\phi(t, f)_2 = (t, f')$ for some f' such that $\text{well-tagged}(t, f')$, this can be rewritten as $X' = \langle -(t, f'), +f' \rangle$, which is a \mathbf{Utg} strand.

□

Corollary 1.

$$C \not\vdash_{tr} tf \Rightarrow C'' \not\vdash \phi(tf).$$

Now the above expression is equivalent to,

$$C'' \models_{\phi(tr)} \phi(tf) \Rightarrow C \models_{tr} tf.$$

Above we proved that every possible inference $\phi(S) \vdash_{\phi(X)} \phi(tf)$ in $\phi(tr)$ from C'' corresponds to an equivalent $S \vdash_X tf$ in tr from C . Hence, the result.

4.2 Formalizing guessing attacks

A guessing attack consists of a *verifier* – any message term which proves that the guess was correct. This can take the following forms:

1. A term that is initially known to the attacker. In our example in the introduction, na is such a verifier. Gong et al. [GLMS93] have defined *only* such terms as verifiers;
2. A term derived in two distinct ways. For example, an attacker can decrypt $\{na\}_{passwd(a,b)}$ and $\{na, nb\}_{passwd(a,b)}$ with a guess, and match na in both. If na obtained in each case are equal, that proves that the guess was correct. Although Gong et al. give an example for such an attack, Lowe was the first to consider it in his formal analysis [Low02].

To make both cases simpler and general, 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 \hat{v}) and see if he can produce v and \hat{v} again. This observation lead to the following new definition of guessing attacks in [CMAFE03].

Definition 6. Let v be a subfact of a fact in a bundle C and g denote a guess. Then, we say that: g is *verifiable* wrt C and v is a *verifier* for g iff:

1. $\hat{C} \cup \{g\} \models v \wedge \hat{C} \cup \{g\} \models \hat{v}$; and
2. $\neg(\hat{C} \models v \wedge \hat{C} \models \hat{v})$.

where \hat{v} is a fresh constant and \hat{C} is a set of terms obtained by replacing the particular occurrence of v in C , with \hat{v} .

Below we illustrate the definition on some examples. Let T denote the set of facts in the bundle C .

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, $\hat{T} = \{\hat{v}, \{na\}_{pab}\}$. It is straightforward to check that $\hat{T} \models \hat{v}$, $\hat{T} \not\models na$ (satisfying condition 2 of Definition 6), and $\hat{T} \cup \{g\} \models \hat{v}$ and $\hat{T} \cup \{g\} \models na$ (satisfying condition 1 of Definition 6). 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, $\hat{T} = \{\{\hat{v}\}_{pab}, \{na, nb\}_{pab}\}$. Both na and \hat{v} are not derivable from \hat{T} (satisfying condition 2), while they are both derivable from $\hat{T} \cup \{g\}$ (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, $\hat{T} = \{\hat{v}, \{na\}_{pab}\}$. Now, $\hat{T} \models \hat{v}$, $\hat{T} \not\models \{na\}_{pk(b)}$ (satisfying condition 2), and $\hat{T} \cup \{g\} \models \hat{v}$ and $\hat{T} \cup \{g\} \models \{na\}_{pk(b)}$ (satisfying condition 1). Hence, $\{na\}_{pk(b)}$ is a verifier for g , and there is a guessing attack. However, consider $v = na$ (in $\{na\}_{pab}$). $\hat{T} \cup \{g\} \models \hat{v}$ but $\hat{T} \cup \{g\} \not\models v$ (not satisfying condition 1). Hence, the particular v cannot be a verifier and it was a wrong choice.

Below we do a simple remodeling of Definition 6 to suite the terminology in this paper.

Definition 7. Let C and g be as defined above. Let sub be an instantiation function for a template $temp$ such that $sub(temp) \in C$ and tt be a tagged template in $temp$; Also let $tf = sub(tt)$. Then, g is *verifiable* from C and tf is a *verifier* for g iff:

$$\hat{C} \cup \{g\} \models tf \wedge \hat{C} \cup \{g\} \models \hat{tf}; \tag{1}$$

and

$$\hat{C} \not\models tf \vee \hat{C} \not\models \hat{tf}. \tag{2}$$

where \hat{tf} is a fresh constant and \hat{C} is obtained by replacing the particular occurrence of tf in C , with \hat{tf} .

4.3 The main result

Our main aim is to show that, whenever there is a guessing attack on C , there is also a guessing attack on C'' . If there is a guessing attack on C , by definition, a guess $g \in G$ is verifiable in C with a verifier $sub(tt)$. Therefore, we frame our main theorem as,

Theorem 2. Whenever $g \in G$ is verifiable from C , g is also verifiable from C'' .

Proof. Let sub'' be defined as in section 3.3:

$$sub''(tt) = \phi(sub(tt)).$$

Let C''' be denoted as \mathcal{C} and $\phi(tf)$ as tf'' . From Lemma 1, $C \cup \{g\} \models tf \Rightarrow \mathcal{C} \cup \{g\} \models \phi(tf)$. Further, from Corollary 1, $C \not\models tf \Rightarrow \mathcal{C} \not\models \phi(tf)$. Therefore, 1 and 2 in Definition 7 can be written as,

$$\hat{C} \cup \{g\} \models tf'' \wedge \hat{C} \cup \{g\} \models t\hat{f}''; \quad (3)$$

and

$$\hat{C} \not\models tf'' \vee \hat{C} \not\models t\hat{f}'' \quad (4)$$

Also, $\phi(tf) = \phi(sub(tt)) = sub''(tt)$. Therefore, g is verifiable in C'' with a verifier $sub''(tt)$. □

4.4 An example

Firstly, observe that we considered type-flaw guessing attacks even in multi-protocol scenarios. The term ‘‘multi-protocol’’ means that two or more protocols are being run simultaneously (not merely two different runs of the same protocol). Our example type-flaw guessing attack in the Introduction is infact based on type-flaws from multiple protocols. Both Heather et al.’s and our results⁴ are applicable even when considering multi-protocol scenarios provided that the same tagging implementation is used across all the protocols under consideration.

Now consider the following, ‘‘Demonstration protocol’’ presented in [GLMS93]:

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

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

Some type-flaw guessing attacks are possible on this protocol [CMAFE03]. We can now use our main result in the paper and conclude that this protocol remains secure against all type-flaw guessing attacks when it uses our tagging scheme. Specifically, the protocol should be implemented as below, following the tagging scheme in this paper⁴:

- Msg 1. $a \rightarrow s : \{\text{pair}, (\text{pair}, (\text{agent}, a), (\text{agent}, b)), (\text{pair}, (\text{pair}, (\text{nonce}, na1), (\text{nonce}, na2))), (\text{pair}, (\text{con}, ca), (\text{wenc}, \{ta\}_{passwd(a)}))\}_{pk(s)}$
- Msg 2. $s \rightarrow b : a, b$
- Msg 3. $b \rightarrow s : \{\text{pair}, (\text{pair}, (\text{agent}, a), (\text{agent}, b)), (\text{pair}, (\text{pair}, (\text{nonce}, nb1), (\text{nonce}, nb2))), (\text{pair}, (\text{con}, cb), (\text{wenc}, \{tb\}_{passwd(b)}))\}_{pk(s)}$
- Msg 4. $s \rightarrow a : \{na1, na2 \oplus k\}_{passwd(a)}$
- Msg 5. $s \rightarrow b : \{nb1, nb2 \oplus k\}_{passwd(b)}$
- Msg 6. $a \rightarrow b : \{\text{randomno}, ra\}_k$
- Msg 7. $b \rightarrow a : \{\text{pair}, (f1(\text{randomno}), f1(ra)), (\text{randomno}, rb)\}_k$
- Msg 8. $a \rightarrow b : \{f2(\text{randomno}), f2(rb)\}_k$

⁴We remove all top-level-tags and tags outside encryptions since they can easily be changed by an attacker and do not guarantee any protection.

The implementation of the tagging scheme using bit strings can be referred from [HLS00].

5 Conclusion

5.1 Summary

In this paper we have considered type-flaw guessing attacks on password protocols. We modified Heather et al.'s existing solution in [HLS00] to prevent type-flaw attacks and proved that such modification prevents all type-flaw guessing attacks on password protocols. Our proof strategy was built on Heather et al.'s proof structure with a minor change: We considered all weak encryptions as atoms. This was possible since we disallowed any attacker operations on such terms during the on-line phase.

Our proof proceeded in two stages:

1. The on-line communication: here we proved that basically the same protocol run is obtained when all messages are correctly tagged, if it was obtained by adopting our tagging scheme. Most of this result was already established by Heather et al. A renaming function is applied on an arbitrarily tagged bundle so that the resulting bundle has every message correctly tagged. Such a renaming is realistic because, if an honest agent is willing to accept an ill-tagged message, it should accept any value in its place;
2. We showed that a guessing attack is possible on the correctly tagged bundle, if it was possible on the original bundle. This indirectly proves that the attack was not based on a type-flaw but on some other mechanism.

In the following section we will discuss some interesting issues together with directions towards future work.

5.2 Discussion and Future work

Observe that our proof (or for that matter Heather et al.'s proof) is highly dependent on the way a type-flaw is defined. For example, if we define that sending an atom of one type, claiming it as an atom of another type is not a type-flaw, then the tag structure would appear as follows:

$$Tag ::= atom | pair | enc Tag^* Tag.$$

Such a tagging would prevent sending an atom as a pair or as a (strong) encryption allow but allows, for example, sending a key, claiming it as an agent's identity.

Similarly, we identified all weak encryptions, regardless of their structure, as belonging to a unique type, *wenc*. Therefore, it would allow weak encryptions having different structures to be replayed in place of one another. For example, a message $\{na, k, nb\}_{passwd(a)}$ can be replayed, claiming it to be structurally identical to $\{k, na, ts\}_{passwd(a)}$ (*na*, *nb* are nonces. *k* is a key and *ts* is a timestamp). However, this problem can be easily overcome by using different tags for weak encryptions having different structures. i.e. tags can be specified as *wenc*₁, *wenc*₂ etc., splitting weak encryptions having different structures inside the encryptions, although they would all be grouped under a common type, *wenc*.

5.2.1 Secrecy, authentication and multi-protocol guessing attacks

Heather et al. conjecture that all tags inside encryptions can be combined into a single *component number*. Such a transformation would be *fault-preserving* in the sense of Hui and Lowe [HL99]: That means, if there is an attack on the component numbering scheme, there would also be an attack on the original tagging scheme. Turning it around, if a protocol is secure under the original tagging scheme, so it is under the component numbering scheme.

The same statements also hold for our proofs. Further, those results on component numbering have a useful implication: they also prevent one encrypted component from being replayed into another ! However, since we disallow tags inside weak encryptions and consequently component numbers, we should seek some other option to achieve the same result that component numbering does. We believe this can be achieved by ensuring that weak encryptions are non-unifiable in the protocol.

Observe that, although Heather et al.'s scheme works against secrecy and authentication in protocols where every key is strong, it cannot be used in password protocols since the tags inside weak encryptions would directly verify a guess. However,

password protocols are designed for basically the same purpose as non-password protocols such as key establishment and authentication. Again, to address this problem, we must adopt “non-unifiable” weak encryptions in the protocol to achieve the same effect as using component numbers. This ensures that password protocols adhering to the restriction are free from type-flaw attacks on secrecy and authentication as well.

Multi-protocol guessing attacks are those where messages from two or more different protocols are mixed to launch a guessing attack. Although Guttman et al. prove in [GT00] that disjoint encryption by using “protocol-identifiers” or disjoint key sets makes protocols independent, their solution suffers from the same problems as Heather et al.’s — they do not consider multi-protocol “guessing attacks”; and protocol identifiers should not be used inside weak encryptions as they would directly verify a guess similar to type-tags. Disjoint encryption through disjoint key sets is also hard to achieve in password protocols since users — even after repeated warnings — tend to use the same password for different applications.

5.2.2 Inference rules

In this paper we have considered the definition for guessing attacks given in [CMAFE03] which only considers verifiers that are subterms of the attacker’s initial knowledge. This definition is specifically tailored to the standard inference rules. In contrast, Lowe’s definition in [Low02] is stronger in this sense, because it can be used for any attacker inference set. (For example the rule $\{m, n\}_k \vdash \{m\}_k$ is not in the standard inference set, but holds when using Cipher Block Chaining.)

However, even Heather et al.’s solution works *only* when the standard inference rules (which imply the perfect encryption assumption) are assumed. We show that this is true by demonstrating an attack on the Woo and Lam authentication protocol π_1 presented in [HLS00]. It was shown to be secure when using the scheme in [HLS00] in that very paper.

Msg 1. $a \rightarrow b : a$
 Msg 2. $b \rightarrow a : nb$
 Msg 3. $a \rightarrow b : \{a, b, nb\}_{sh(as)}$
 Msg 4. $b \rightarrow s : \{a, b, \{a, b, nb\}_{sh(as)}\}_{sh(bs)}$
 Msg 5. $s \rightarrow b : \{a, b, nb\}_{sh(bs)}$

$sh(xy)$ represents a shared-key between agents x and y . Heather et al. present a type-flaw attack on this protocol:

Msg 3. $a \rightarrow b : nb$
 Msg 4. $b \rightarrow I_s : \{a, b, nb\}_{sh(bs)}$
 Msg 5. $I_s \rightarrow b : \{a, b, nb\}_{sh(bs)}$

The attack works by (i) using a type-flaw in message 3 (nb in place of $\{a, b, nb\}_{sh(as)}$) and (ii) replay of message 4 in message 5. Heather et al. argue that inserting unique component numbers inside encryptions prevents this attack and all possible type-flaw attacks on this protocol. In their scheme, the same protocol would be implemented as:

Msg 1. $a \rightarrow b : a$
 Msg 2. $b \rightarrow a : nb$
 Msg 3. $a \rightarrow b : \{a, b, nb, 1\}_{sh(as)}$
 Msg 4. $b \rightarrow s : \{a, b, \{a, b, nb, 1\}_{sh(as)}, 2\}_{sh(bs)}$
 Msg 5. $s \rightarrow b : \{a, b, nb, 3\}_{sh(bs)}$

But an attack is possible on the protocol even when it uses Heather et al.’s scheme if the CBC inference rule is considered:

Msg 1. $a \rightarrow b : a$
 Msg 2. $b \rightarrow a : nb$
 Msg 3. $I(a) \rightarrow b : (nb, 3) \quad /* \text{ In place of } \{a, b, nb, 1\}_{sh(as)} */$
 Msg 4. $b \rightarrow I(s) : \{a, b, (nb, 3), 2\}_{sh(bs)}$
 Msg 5. $I(s) \rightarrow b : \{a, b, nb, 3\}_{sh(bs)} \quad /* \text{ using CBC inf rule on Msg 4. } */$

This attack works because, an attacker can infer $\{a, b, nb, 3\}_{sh(bs)}$ from Msg 4 ($\{a, b, (nb, 3), 2\}_{sh(bs)}$) using the CBC inference rule.

Note that according to Heather et al., if there is an attack on a protocol using component numbering, there is also an attack on the protocol when using their original tagging scheme (although it is doubtful whether the result applies for inference rules outside the standard set).

- [Low02] Gavin Lowe. Analyzing protocols subject to guessing attacks. *Workshop on Issues in the Theory of Security (WITS'02)*, January 2002.
- [Low03] Gavin Lowe. Analyzing protocols subject to guessing attacks. *Journal of Computer Security*, To appear, 2003.
- [MCJ97] W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Available via URL: <http://dimacs.rutgers.edu/Workshops/Security/program2/program.html>.
- [MS03] Jon Millen and Vitaly Shmatikov. Symbolic protocol analysis with products and diffie-hellman key exponentiation. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, 2003.

It did not make sense to us to expect that an attacker who is restricted to standard inference rules in the on-line communication to have some other set of inference rules in the off-line phase. That is the reason why we used our definition of guessing attacks, which has the advantage that it is simpler and easier to reason about guessing attacks when using it.

Our definition may very well skip some attacks which can be found using Lowe's definition or some other definition. However, we used it to find *all* known attacks and even some new attacks on published protocols. Further, no definition of guessing attacks can be expected to be "complete". For example, refer Lowe's definition itself, which was changed from [Low02] to [Low03] to allow for additional possibilities.

In any case, of course, although we conjecture that our results hold even when using Lowe's definition, it would be interesting to see a formal proof of the same.

5.2.3 Miscellaneous issues

There are two other unsolved issues in Heather et al.'s scheme and hence in our scheme:

1. They do not consider all possible forms of constructed keys (but only those that result from application of a key function F_n to concatenation of sequence of atoms (f_1, \dots, f_n));
2. They do not consider cancellativity and other algebraic properties obeyed by message elements when using operations such as products and XOR (these operations are frequently used in real-world protocols [MS03]).

Lastly, we did not consider implementation dependent guessing attacks in this paper. For example, the password can be learned from $\{\textit{english_text}\}_{\textit{password}(a)}$ by decrypting it with a guess (even though *english_text* is not known initially).

We look forward to the future with all the issues pointed out in this section, which will keep us busy.

6 Acknowledgments

We would like to thank the guest editors in this special issue of JSAC and the anonymous referees in the workshop on Foundations of Computer Security (FCS 03) for insightful comments. Thanks are also due to Ricardo Corin for many helpful technical discussions. This work was funded in part by DARPA under grant no. F30602-2-1-0178.

References

- [CMAFE03] Ricardo Corin, Sreekanth Malladi, Jim Alves-Foss, and Sandro Etalle. Guess what? here is a new tool that finds some new guessing attacks. *Workshop on Issues in the Theory of Security (WITS'03)*, March 2003.
- [DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.
- [FHG99] F. J. Thayer Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2,3):191-230, 1999.
- [GLMS93] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648-656, 1993.
- [Gon90] L. Gong. A Note on Redundancy in Encrypted Messages. *ACM Computer Communication Review*, 20(5):18-22, October 1990.
- [GT00] Joshua D. Guttman and F. Javier Thayer. Protocol Independence through Disjoint Encryption. *13th IEEE Computer Security Foundations Workshop*, pages 24-34, July 2000.
- [HL99] Mei Lin Hui and Gavin Lowe. Safe Simplifying transformations for security protocols. In *12th Computer Security Foundations Workshop Proceedings*, pages 32-43. IEEE Computer Society Press, June 1999.
- [HLS00] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings, 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, July 2000.