

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

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

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 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, their tagging scheme can not 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 following a modification of Heather et al.'s scheme most type-flaw guessing attacks can still be prevented.

1 Introduction

Numerous protocols have been introduced to initialize security services for protocol users. One of the goals of these protocols is authentication of a sender's identity. There exists a class of protocols called password protocols, that use user chosen passwords for authentication. If these protocols are not designed well, they may be subject to *guessing attacks* [GLNS93]; here an attacker can learn the password by guessing it and verifying the guess using the messages in the protocol.

A message in a protocol is said to have a *type-flaw* if it was created with an intended type but is later received and treated as a different type. For example, receiving a nonce and treating it as though it was an agent's identity. A *type-flaw guessing attack* is an attack where a type flaw is induced in a protocol to enable a password guessing attack.

*We dedicate this paper to Late Professor Roger Needham. This work was funded in part by DARPA under grant no. F30602-2-1-0178.

1.1 Type-Flaw Guessing Attacks

Consider the following protocol we discussed in [?]:

Msg 1. $A \rightarrow B : \{\{K\}_{pk(B)}, \{\{K\}_{pk(B)}\}_k\}_{pab}$
Msg 2. $B \rightarrow A : \{NB, \{K_2\}_K\}_{pab}$
Msg 3. $A \rightarrow B : \{NB\}_{K_2}$.

A type-flaw guessing attack is possible against this protocol. 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. $A \rightarrow I(B) : \{\{k\}_{pk(B)}, \{\{k\}_{pk(B)}\}_k\}_{pab}$
Msg 2. $I(B) \rightarrow A : \{\{k\}_{pk(B)}, \{\{k\}_{pk(B)}\}_k\}_{pab}$
Msg 3. $A \rightarrow I(B) : \{\{k\}_{pk(B)}\}_k$.

In message 2, attacker replays message 1 back to A , causing a type-flaw. A cannot detect this type-flaw and hence responds by sending message 3.

Attack: The attacker goes off line and begins guessing values for pab . Using a guess he decrypts message 1 with the guess, splits it, and takes the first part ($\{k\}_{pk(B)}$) out of it. He can then decrypt message 3 with this value to obtain it $\{k\}_{pk(B)}$ again, thereby verifying the guess. This guessing attack is possible because the attacker tricked A into sending redundant information in message 3 [?].

Observe that this attack is not possible if A can detect the type-flaw in message 2. Tagging messages 1 and 2 will enable this detection, but can also enable the guessing attack unless we are careful.

1.2 Tagging to Prevent Type Flaw Attacks

Heather et al. in [HLS00] proved that attacks involving type-flaws can be prevented if all messages are tagged with their types. For example, in their scheme, a nonce na should be tagged as (nonce, na), an agent's identity a as, (agent, a) and so on.

However, there is a problem with Heather et al.'s solution. Consider the message:

$$\{na\}_{passwd(a,b)}^1$$

An attacker can attempt a guessing attack by guessing the password $passwd(a, b)$. For example, if the user name is “Arnold Schwarzenegger”, “terminator” wouldn’t be a bad guess for $passwd(a, b)$. If the attacker knows na , he can decrypt $\{na\}_{passwd(a,b)}$ with “terminator” to see if it matches the na he knows. If so, that verifies the guess. Otherwise, he can try using another guess.

Note that this attack is not feasible if the attacker does not know na initially. But consider the same message using Heather et al.’s scheme of type-tagging:

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

The attacker can decrypt with the 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 ! Therefore, Heather et al.’s solution against type-flaw attacks cannot be used in password protocols.

1.3 Tagging to Prevent Type-Flaw Guessing Attacks

We have run into a classic security problem: one security solution, tagging to prevent type-flaw attacks, introduces a new problem, enabling of non type-flaw guessing attacks.

In this paper, we address this problem by modifying the tagging scheme. We prove that if we follow Heather et al.’s scheme but avoid type-tags inside terms encrypted with passwords, most the type-flaw guessing attacks can still be prevented.

The only type-flaw that our modified scheme fails to prevent is the following: a password encrypted term, say $\{m1\}_{passwd(a,b)}$ being received, expecting to be of the form $\{m2\}_{passwd(a,b)}$ with $m1$ and $m2$ having different types (ideally). We will have more to say about this case in the Conclusion.

2 Proof Strategy

We introduce a modified version of Heather et al.’s tagging scheme that prevents most 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.

¹Here na is a nonce; $\{na\}_{passwd(a,b)}$ represents na encrypted with $passwd(a, b)$.

Therefore, we prove that:

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);
- Attacker inferences from the set of messages in the protocol run that enable him to verify a 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 elements such as *nonce*, *agent* etc. We associate a generic tag “wenc” for weak encryptions. We introduce their protocol model and state their main result in section 3.

For point 2 above, we use the definition for guessing attacks from [?] and show that, whenever a guess is verifiable from C , then 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.

$$TaggedFact ::= Tag \times Fact.$$

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. In our modification we add the set *Wenc* to *Atoms* to represent “weak encryptions”. The corresponding tag is *wenc*. We treat 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.

$$Tag ::= agent \mid nonce \mid wenc \mid \dots \mid pair \mid enc \ Tag^* \ Tag$$

$$Fact ::= Atom \quad \left| \begin{array}{l} PAIR \ TaggedFact \ TaggedFact \\ ENCRYPT \ Tag \ TaggedFact \ Fact \end{array} \right.$$

An atomic fact *a* of type “agent”, associated with the corresponding tag *agent* is written as (*agent*, *a*). 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* < *t*₁, *t*₂, . . . , *t*_{*n*} > *kt* where *t*₁, *t*₂, . . . , *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 $\{[t_1, t_2, \dots, t_n]\}_{kt}$. 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.)

We extend this message structure by defining the structure of atoms of type *wenc* as below:

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

By defining such a structure, we imply that no facts 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.

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, whereby honest agents are capable of knowing if they decrypted an encryption correctly [?].

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:

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

$$\begin{aligned} \text{well-tagged}(\text{pair}, x) & \Leftrightarrow \exists tf_1, \\ & \quad tf_2 : TaggedFact \bullet x = PAIR \ tf_1 \ tf_2 \wedge \\ & \quad \text{well-tagged} \ tf_1 \wedge \text{well-tagged} \ tf_2, \end{aligned}$$

$$\begin{aligned} \text{well-tagged}(\{[ts]\}_{kt}, x) & \Leftrightarrow \exists tf : TaggedFact; \\ & \quad k : Fact \bullet x = \{tf\}_k^{kt} \wedge \text{well-tagged}(tf) \\ & \quad \wedge \text{well-tagged}(kt, k) \wedge ts = \text{get-tags} \ tf. \end{aligned}$$

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 a fact is correctly tagged at the outer-most level. This means, for example, a taggedfact is indeed a pair of tagged facts when it's tag equals pair, even if the two tagged facts may not be well-tagged.

$$\begin{aligned}
\text{top-level-well-tagged}(\text{agent}, x) &\Leftrightarrow x \in \text{Agent}, \\
\text{top-level-well-tagged}(\text{nonce}, x) &\Leftrightarrow x \in \text{Nonce}, \\
\text{top-level-well-tagged}(\text{wenc}, x) &\Leftrightarrow x \in \text{Wenc}, \\
&\dots \\
\text{top-level-well-tagged}(\text{pair}, x) &\Leftrightarrow \exists \text{tf}_1, \\
&\text{tf}_2 : \text{TaggedFact} . x = \text{PAIR } \text{tf}_1 \text{tf}_2,
\end{aligned}$$

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

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 [?]. A *strand* is a sequence of communications represented as $\langle \pm \text{tf}_1, \pm \text{tf}_2, \dots, \pm \text{tf}_n \rangle$. $+\text{tf}$ indicates sending tf and $-\text{tf}$ 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 starting 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 is 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 output 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:

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

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 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:

$$\begin{aligned}
\text{sub}(t, v) &= (t, \text{sub}(v)) \text{ for } v \in \text{Var}, \\
\text{sub}(t, g(v_1, \dots, v_n)) &= (t, g(\text{sub}(v_1), \dots, \text{sub}(v_n))), \\
&\text{where } g \in F_n, \text{ and } F_n \text{ is the} \\
&\text{set of function identifiers.} \\
\text{sub}(\text{pair}, (tt_1, tt_2)) &= (\text{pair}, (\text{sub}(tt_1), \text{sub}(tt_2))), \\
\text{sub}(\{|ts|\}_{tk}, \{tt\}_k) &= \{|ts|\}_{tk}, \{\text{sub}(tt)\}_{\text{sub}(tk, k)_2}, \\
&\text{where } k = g(v_1, \dots, v_n) \\
&\text{and } g \in F_n \text{ represents a key} \\
&\text{type using a particular keying} \\
&\text{algorithm.}
\end{aligned}$$

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:

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].)

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. A *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> | $\langle -(tk, k), -tf, +(\{ts\}_{tk}, \{tf\}_k^{tk}) \rangle$, where $ts = \text{get-tags}(tf)$. |
| D | <i>Decryption</i> | $\langle -(tk', k'), -(\{t\}_{tk}, \{tf\}_k^{tk}), +tf \rangle$, |

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 Section 4 we will later add some more strands to the above capabilities to model off-line guessing attacks.

Note that, we treat weak encryptions as an ‘‘abstract type’’. i.e. we do not allow the attacker to perform any operations on it during the on-line communication. We also 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 learnt 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 well-tagged(t, f').²

Below is the definition and properties of the renaming transformation:

Definition 3.

$$\phi : TaggedFact \rightarrow TaggedFact$$

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: well-tagged($\phi(tf)$);

²There seems to be a typo in [HLS00] in stating the same.

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:

$$\phi(tf1, tf2) = (\phi(tf1), \phi(tf2));$$

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

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

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;

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 $\phi(tf')_2$ for any other fact tf' of C :

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

where $\text{facts}(C)$ 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. Since our modification only defines a new subset of the atoms, the proofs presented in Heather et al. [HLS00] still hold and are summarized below:

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].)

3.6 Main Result

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 the above 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 : Guessing attacks

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 an attacker possesses. We add some more penetrator strands to the capabilities in definition 2 to capture capabilities in the off-line phase:

- D_G** *Decryption_using_Guess* $\langle -(wenc, f), -g, +f' \rangle$
with $g \in G, f \in Wenc \cdot f = \{f'\}_w$
 $\wedge w \in Weakkeys \cdot w = g.$
- E_G** *Encryption_using_Guess* $\langle -f, -g, +\{f\}_g \rangle$
with $g \in G, f \in Subwenc.$
- Cf** *Concatenating_facts* $\langle -f, -f', +(f, f') \rangle.$
- Sf** *Separating_facts* $\langle -(f, f'), +f, +f' \rangle.$
- Tg** *Tagging* $\langle -t, -f, +(t, f) \rangle.$
- Utg** *Untagging* $\langle -(t, f), +f \rangle.$

We prove that basically the same strands can be constructed from C'' . Let X be a penetrator strand from C and X' , the corresponding strand from C'' . If X is a **D_G** strand, define

$X' = \langle -\phi(\text{wenc}, f), -g, +f' \rangle$, which is a valid \mathbf{D}_G strand.

If X is a \mathbf{E}_G strand, define

$X' = \langle -\phi(t, f), -g, +\{\phi(t, f)_2\}_g \rangle$, which is a valid \mathbf{E}_G strand.

If X is a \mathbf{Cf} strand, define

$X' = \langle -\phi(t, f)_2, -\phi(t, f')_2, +(\phi(t, f)_2, \phi(t, f')_2) \rangle$, which is a valid \mathbf{Cf} strand.

If X is a \mathbf{Sf} strand, define

$X' = \langle +(\phi(t, f)_2, \phi(t, f')_2), +\phi(t, f)_2, +\phi(t, f')_2, \rangle$, which is a valid \mathbf{Sf} strand.

If X is a \mathbf{Tg} strand, define

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

Now $\phi(t, f) = (t, f')$ for some f' such that well-tagged(t, f'). Therefore, we can rewrite the above expression as

$X' = \langle -t, -f', +(t, f') \rangle$, which is a valid \mathbf{Tg} strand.

If X is a \mathbf{Utg} strand, define

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

again, since $\phi(t, f)_2 = (t, f')$ for some f' such that well-tagged(t, f'), this can be rewritten as

$X' = \langle -(t, f'), +f' \rangle$, which is a valid \mathbf{Utg} strand.

4.1 Defining guessing attacks

Before giving a formal definition for guessing attacks, 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 .

Firstly, we introduce a simple inference relation \vdash . If S a set of tagged facts, we write $S \vdash_X tf$ if the strand X can be constructed such that, for every tf' on a ‘-’ node in X , $tf' \in S$ and tf is a tagged fact on any ‘+’ node of X .

Definition 4. Let C be a bundle. Then, tfn is *deducible* from C , or:

$C \models_{tr} tfn$, where $tr = \langle S1 \vdash_{X1} tf1, S2 \vdash_{X2} tf2, \dots, Sn \vdash_{Xn} tfn \rangle$, and for $i = 1 \dots n$, $Si+1 \subseteq \text{Taggedfacts}(C) \cup \{tf1, \dots, tfi\}$, where $\text{Taggedfacts}(C)$ is the set of taggedfacts on all the nodes in C .

We will tend to drop the subscript tr when it is obvious.

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

$$C \cup \{g\} \models_{tr1} tf \Rightarrow C'' \cup \{g\} \models_{tr2} \phi(tf).$$

Proof. In order to prove the above proposition, we need to show that, for every possible inference $S \vdash_X tf$ in $tr1$, there is an equivalent $\phi(S) \vdash_X \phi(tf)$ in $tr2$. This inturn implies we need to show that for every possible strand in X from C , there is an equivalent strand in C'' .

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. In section 4 we proved that, for every penetrator strand used on C in the off-line phase, an equivalent penetrator strand can be constructed from C'' .

Hence, the result. \square

Lemma 2. Let C and C'' be two bundles defined as in section 3. Then,

$$C \not\models tf \Rightarrow C'' \not\models \phi(tf).$$

Proof. We proceed as in the previous lemma. For every possible strand from C , there will be an equivalent strand possible from C'' . Observe from Heather et al.’s results that, every tagged fact tf in C has an equivalent $\phi(tf)$ in C'' which is well-tagged.

There are two cases when it may be possible to construct a penetrator strand from C'' but from C :

1. There is a tagged fact tf such that $tf \in \text{Taggedfacts}(C'')$, but $tf \notin C$;
2. When a key k cannot be used in a \mathbf{D} strand in C but $\phi(k)$ can be used in an equivalent strand from C'' .

However,

1. From [HLS00, Theorem 1], unique origination is preserved in C'' obtained from C ;
2. By condition 6 of definition 3, ϕ respects inverses. Therefore, it is not possible to construct a \mathbf{D} strand from C'' which was not possible from C .

\square

Using the above formalism, we give a simple definition for a guessing attack. We say that a *guessing attack* is possible on a bundle C , if a guess $g \in G$ is *verifiable* in C .

In short, we try to see if the attacker can derive a tagged-fact in *atmost* one way before guessing, but in more than one way after guessing. To find if there are two different ways to derive a taggedfact, we ‘mask’ the first occurrence with some random value and then look for another occurrence of it.

Definition 5. 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:

1. $\hat{C} \cup \{g\} \models tf \wedge \hat{C} \cup \{g\} \models \hat{t}f$; and
2. $\hat{C} \not\models tf \vee \hat{C} \not\models \hat{t}f$.

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

4.2 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 C and $\phi(tf)$ as tf' . Now if g is verifiable in C , by definition 5,

1. $\hat{C} \cup \{g\} \models tf' \wedge \hat{C} \cup \{g\} \models \hat{t}f'$; and
2. $\hat{C} \not\models tf' \vee \hat{C} \not\models \hat{t}f'$.

From Lemma 4.2, $C \cup \{g\} \models tf \Rightarrow C \cup \{g\} \models \phi(tf)$. Further, from Lemma 4.3, $C \not\models tf \Rightarrow C \not\models \phi(tf)$. Therefore, (1) and (2) above can be rewritten as,

- 1'. $\hat{C} \cup \{g\} \models tf' \wedge \hat{C} \cup \{g\} \models \hat{t}f'$; and
- 2'. $\hat{C} \not\models tf' \vee \hat{C} \not\models \hat{t}f'$.

Further, $\phi(tf) = \phi(sub(tt)) = sub'(tt)$.

Therefore, g is verifiable in C with a verifier $sub'(tt)$. \square

5 Conclusion

In this paper we have considered type-flaw guessing attacks on password protocols. We modified Heather et al.'s existing solution to prevent type-flaw attacks and proved that such modification prevents 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.

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.

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

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

5.1 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. i.e. 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:

$$Tag ::= atom \mid pair \mid enc \, Tag^* \, Tag$$

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

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). Such type-flaws may be used in attacks but can neither be prevented by our tagging scheme nor our proof establishes that they cannot be used in attacks.

However, in practice, many times such replays can be avoided. For example, consider the following messages in Gong et al.'s popular, "Demonstration protocol" [?]:

$$\begin{aligned} \text{Msg 1. } a &\rightarrow s : \{a, b, na1, na2, ca, \{ta\}_{passwd(a)}\}_{pk(s)} \\ \text{Msg 4. } s &\rightarrow a : \{na1, na2 \oplus k\}_{passwd(a)} \end{aligned}$$

Here ca is a redundant random number. $pk(s)$ is the public-key of s . Under some assumptions about message structures, a type-flaw guessing attack is possible on this protocol. An attacker can use Msg 4 in a legitimate run between a and s as follows:

Msg 1. $I(a) \rightarrow s$: $\{a, b, nI1, nI2, ca,$
 $\{na1, na2 \oplus k\}_{passwd(a)}\}_{pk(s)}$
 Msg 4. $s \rightarrow I(a)$: $\{nI1, nI2 \oplus k'\}_{passwd(a)}$

$I(a)$ denotes attacker I pretending as a . The attacker creates his own nonces $nI1$ and $nI2$ together with Msg 4 of the previous run to construct Msg 1 and sends it to s . After he gets back Msg 4 from s as a response, he decrypts it with a guess and matches the first part ($nI1$) with his $nI1$ to verify the guess.

The other messages of the protocol are irrelevant in this attack.

Now this attack can be prevented if there is a tag for the time stamp ts in Msg 1. This type tag would not directly verify a guess because it is protected by another layer of encryption under a strong key ($pk(s)$).

Some replays cannot be avoided. For example, $\{f\}_{passwd(a)}$ can be replayed in $\{f'\}_{passwd(a)}$ provided f and f' can be “unified”. However, in most cases, the possibility of such unification itself means that a guessing attack is possible: since unification implies that constants in f and f' should match, whenever f and f' are textually distinct (except for the positions of the constants), the constants would themselves verify a guess. For example, $\{na, K, NB\}_{passwd(a)}$ can be unified with $\{na, Ts, K\}_{passwd(a)}$ (na is constant, K, NB are variables). However, na can be obtained from both messages in two different ways, by using a guess; this verifies the guess even before unification!

Observe that in the tagging scheme, tags not protected by encryption can be safely removed while achieving the same results. Further, the tags inside encryptions can be combined into a single *component number*. As Heather et al. argue, this simplification is *fault-preserving* in the sense of Hui and Lowe [HL01]: That means, if there is an attack on the component numbering scheme, there was also an attack on the original tagging scheme.

Such component numbering ensures that encrypted components can not be replayed in place of one another. Above we argued (although yet to prove formally), that weak encryptions should as well be non-replayable (i.e. non-unifiable). Therefore, a protocol following this numbering suggestion, along with the component numbering scheme, ensures that no replays of encrypted components are possible. Such a result in protocol analysis has already been shown in numerous occasions as holding the key to protocol security [AN94, ?]. Fairly recently, it was also shown to ensure decidability for security protocols in the context of secrecy [?]. (Secrecy is a security property that specifies that an attacker should not be able to learn a secret value from a protocol run.)

We also believe that the result regarding component numbering makes it easy to prove that “protocol number-

ing” inside encrypted components would prevent *multi-protocol guessing attacks* [MAFM02, ?] if we can find a way to enforce the numbering. (A multi-protocol guessing attack works by replaying encrypted components from one protocol into a different protocol.)

Observe that we assume sufficient redundancy inside strong encryptions that allows honest agents to know if they decrypted them correctly. However, we did not allow such a redundancy in weak encryptions because that may verify a guess directly [Gon90]. In contrast, Lowe states that redundancy inside any encryptions (including strong) would aid in guessing attacks [Low02]. However, without the redundancies it is hard to see how honest agents can run protocols, satisfactorily.

Secrecy and guessing attacks seem to be quite more integrally related than what meets the eye. Halevi et al. have shown that security against guessing attacks can be reduced to the initial problem of establishing a secret between two unfamiliar parties [?]. (A corollary is that public key encryption is unavoidable to solve both the problems.) Thus, it is not entirely surprising that the same problems and solutions encountered in studying secrecy attacks on protocols also apply for guessing attacks.

Observe that learning a password through a guessing attack can result in breaches of secrecy not known to exist when analysing protocols for secrecy. For example, a successful guessing attack is possible on $\{na, nb\}_{passwd(a)}$ and na , but attacker also learns an otherwise secret nb .

Also observe that, like secrecy and authentication, guessing attacks should also be studied as a *trace property* (A trace property is a security property that can be verified by examining all possible traces or protocol runs within a scenario). Therefore, it would be interesting to see if the same results regarding decidability that were published for secrecy and authentication apply for guessing attacks as well (eg. [MS01, ?]).

The ideal tag environment ρ defined in section 3 assumes more importance than it may seem. A necessary condition for successful use of the tagging scheme is that *all* honest agents follow the same implementation. For example, agent a cannot run a protocol using value 001 for the tag nonce with b , who uses another value, say 101 for the same tag. This is also true when a itself is involved in different runs of the same protocol or if it is simultaneously engaging in runs from different protocols (eg. SSL 3.0 and SET concurrently). However, Heather et al.’s formal definition of ρ only specifies that each of the honest roles need to have tag values that are consistent within the same template; they do not specify that *all* honest agents follow the same tag values, which we believe is inadequate. Of course, it is also hard to have such “universally-agreed upon” tag values without having some sort of “international standards” for tagging schemes. And, there is no guarantee that malicious code

will use the wrong tag values to deliberately tailor a protocol to use for attacks [?, AF98]

In this paper we have considered the definition for guessing attacks given in [?] 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 [?] 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.) It would be interesting to see how this affects the results in this paper.

However, regardless of how such inference rules affect the results, they can be used in attacking Heather et al.’s original scheme as well (See Appendix for an attack on the Woo and Lam authentication protocol π_1).

There are two other unsolved issues in Heather et al.’s 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).

Lastly, we did not consider implementation dependent guessing attacks in this paper. For example, the password can be learned from $\{english_text\}_{passwd(a)}$ 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.

Acknowledgments

We would like to thank Iliano Cervesato and the anonymous referees for insightful comments. Thanks are also due to Ricardo Corin for many helpful technical discussions.

References

- [AN94] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 122–136, 1994.
- [DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2), 1983.
- [GLNS93] Li Gong, 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.
- [HL01] Mei Lin Hui and Gavin Lowe. Fault-preserving Safe Simplifying Transformations on Security Protocols. *Journal of Computer Security*, 9:3–46, 2001.
- [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.
- [Low02] Gavin Lowe. Analyzing protocols subject to guessing attacks. *Workshop on Issues in the Theory of Security (WITS’02)*, January 2002.
- [MAFM02] Sreekanth Malladi, Jim Alves-Foss, and Sreenivas Malladi. Preventing Guessing Attacks Using Fingerprint Biometrics. To Appear, *Proceedings of 2002 International Conference on Security and Management, SAM02*, June 2002.
- [MS01] Jonathan Millen and Vitaly 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.
- [WL94] T.Y.C. Woo and S. S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24–37, 1994.

Appendix 1 : Attack on Heather et al.’s scheme

Consider the Woo and Lam authentication protocol, π_1 [WL94]:

- 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. 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. $s \rightarrow b : \{a, b, nb, 3\}_{sh(bs)}$

However, Heather et al's results are valid *only* when assuming the standard inference rules. To see why, consider the inference rule $\{m, n\}_k \vdash \{m\}_k$ which would hold when using Cipher Block Chaining for encryption.

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