SPaCIoS

Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D3.5
Methodology for Fault Localization based on Execution Traces

Abstract

This deliverable presents a methodology and technology for fault localization. This includes two main points: a method that employs static analysis for finding and repairing flaws in security protocols, and a technique that uses an identified attack trace in order to detect its root cause and to fix the protocol. Both methods are guided by a set of rules for good protocol design. The second method also employs a logic-based strategy for fault localization.

Deliverable details

Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *07.08.2013*
Due on: *31.07.2013*
Editors: *IeAT*
Total pages: *35*

Project details

Start date: *October 01, 2010*
Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

1 Introduction

Security protocols are notoriously difficult to design and develop correctly, which can be seen by the number of protocols shown to be flawed. Recent advances in using formal techniques to model and verify not only protocols, but also web applications, software components, etc., has led to an increased confidence in the security aspects of communication software.

However, while verification tools help finding different types of attacks to a modeled system, they usually lack support for actually localizing the flaw that made the attack possible. Under these circumstances, redesigning flawed security protocols for correctness remains almost just as hard as getting the design right from the first try.

Therefore, it is still hard to stop insecure systems from being designed and/or developed, and many protocols and web applications are still found to be vulnerable. While formal verification of such systems is able to uncover a large spectrum of possible attacks, the localization of actual faults behind these attacks is still a matter of manual inspection of the model. We aim to address this limitation and provide further automated support for good protocol design and development.

In this document, we describe the techniques developed in SPaCIoS for fault localization and repair. Some of these techniques take advantages from other components and resources present in the SPaCIoS platform. For instance, the trace-based fault localization method specifically addresses protocols described in the ASLan++ modeling language, and makes use of the model-checkers developed together with ASLan++.

In Section 2, we briefly present related work on fault localization and repair for security protocols and applications.

Section 3 describes a rule-based method that employs static analysis and specific protocol design principles to identify the flaws in security protocols. For some of these flaws, automatic repair can also be performed.

In Section 4, we describe our work on a trace-based technique for fault localization and repair. This technique employs the attack trace obtained during the verification of a protocol written in ASLan++, and several interaction patterns, in the attempt to identify the root cause of the attack. Once such a possible cause is localized, we try to repair the protocol. To see whether the performed fix was effective, we then re-verify it, and perform additional fixes if necessary. Thus, the verification and repair process has the structure of a feedback loop, and only stops when either no new attack was found, so the new protocol is finally correct, or no new opportunities for repair are identified.

Finally, Section 5 summarizes the results of our approach.

2 Related Work

Fundamental insight into the root causes of faults in security protocols is offered by the classic set of informal principles for correct protocol design proposed by Abadi and Needham [1]. However, while these design rules are general and practical, they do not provide an explicit causality for faults. As acknowledged from the start, the proposed principles are neither sufficient, nor always necessary to ensure the security of the designed system. Furthermore, with some exceptions, they are not only informal, but also too vague to formalize and prone to various interpretations (e.g., "Be clear as to why encryption is being done", or "The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not").

Pimentel et al. [9] have also developed an approach to automatically repair security protocols susceptible to replay attacks. Their technique uses a strand space formalism for protocol modeling and is based on detecting violations to several Abadi and Needham principles. The idea is that for each requirement, there is a collection of rules that transform a set of protocol steps violating the requirement into a set conforming to it. The method has been implemented in the tool SHRIMP and successfully tested on 36 flawed protocols. While our approach is clearly similar to their work, as it addresses specific cases such as the lack of agent naming in a message, or no session binding, it is also different in that it aims at a more general view on the fault localization and repair problem, as it also treats the more low-level case of accepting messages which are insufficiently constrained.

Other sets of rules have been proposed, aiming to provide stronger guarantees, such as the set introduced by Carlsen et al. [4], which says that, in order to avoid replay attacks, each message must contain the protocol-id, session-id, step-id, message subcomponent-id and primitive type of data items. Similarly, Aura et al. [2] suggest that several encryption solutions should be used in the same protocol. Although these latter sets of principles are useful, their requirements appear overburdening for many protocol designers, as they imply dealing with a large number of details.

Attempts to automatically generate correct protocols also exist, such as the AGP method [11], which uses the principles of [4]. However, most security protocols and communication systems are still designed by hand for reasons of both established practice and understandability. Thus, being able to localize and explain faults in protocols is still of significant practical importance.

Choo et al. [5] introduce a method for protocol repair using asynchronous product automata as formalism. The state space of the protocol is analyzed by a dedicated model-checker, and, when one of several types of attack is

found, the protocol is fixed using a generic patch corresponding to the identified attack. As it is specifically directed towards a set of known generic attacks, this approach, while providing useful insight in the studied problem, is less general then both the work of Pimentel et al. [9] and ours.

Basin et al. [3] verify the protocols that are part of the ISO/IEC 9798 Standard, and find several weaknesses. In particular, they reveal the possibility that a trusted third party can behave maliciously by playing different roles. After analyzing these weaknesses, they introduce two new design principles: positional tagging for non-ambiguous identification of all message fields, and the inclusion of all entities (with their identities) and their roles. By using these principles they succeed in repairing the faulty protocols. The correctness of the fixed protocols is then proved using a theorem prover. As we address neither type-flaw attacks, nor issues concerning the actual identity of the principals, we consider the two approaches complementary.

The authors of [14] researched leading web SSO providers (Facebook, Google, JanRain, PayPal) and high-profile websites (Freelancer, FarmVille, Sears) and revealed surprising flaws. They mechanically analyze the browser traffic in SSO schemes in order to identify the dataflow between messages and extract semantic information about the message contents. From here, they derive opportunities for exploits and at the same time identify the flaws that make the exploits possible.

Son et al. develop methods such as RoleCast [12] and FixMeUp [13] that use static analysis to find and repair missing security checks and similar faults in access control policies of web applications. They exploit software engineering patterns to identify roles and security checks [12] and synthesize access control templates which are used to insert repairs when the application logic is faulty and does not match them. The usability is demonstrated on real-life PHP and JSP applications. While this approach is related to our method for detecting and fixing insufficiently constrained receives, it differs in at least two aspects: it is static, while our technique uses a reported attack trace, and it works on the application's source code, while we employ a higher level of abstraction by analyzing and repairing ASLan++ models.

3 Rule-based Fault Localization

In this chapter we present two static analyses to localize security protocol faults. The first analysis detects violations of protocol design rules stated by Abadi and Needham [1] and indicates possibilities for protocol repair. The second is a dataflow analysis that can detect missing checks. We illustrate it on several Cashier-as-a-Service protocols reported as flawed in the literature.

These approaches attempt to localize potential sources of faults using static analysis only. They can be further refined with the techniques in Section 4 when an actual faulty execution trace is known.

3.1 Abadi and Needham's principle of explicit naming

Abadi and Needham's classic paper [1] presents a set of principles for cryptographic protocol design. As stated, they are "neither necessary nor sufficient", but constitute a set of sensible design rules. Their violation is a natural candidate as root cause for detected faults.

The baseline in [1] is given by two general principles. The first states that "every message should say what it means: the interpretation of the message should depend only on its content", and "all elements of this meaning should be explicitly represented in the message". The second basic principle states that "the conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not." These basic principles are not intended to be formalized or checked directly, but serve as starting point for more specific rules stated subsequently.

Principle 3 states that "if the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message". We focus on this rule since it can be implemented in a straightforward manner and its absence is a frequent culprit for protocol faults. It is also the rule most often employed for protocol repair in the study of Pimentel et al. [9].

A stronger form of this rule is discussed in [3] where it is proposed to always include information about the identities of *all* principals participating in the protocol. Thus it impacts protocols with more than two actors, requiring also the naming of the remaining principals besides the message sender and receiver. Because most protocols have a fixed number of roles, this principle can be implemented by positional tagging, i.e., including an ordered sequence of the identities involved in each cryptographic message component, such that the role of an agent can be inferred from its position in the sequence. Since positional tagging usually means a significant change

in the protocol, we restrict ourselves to the original naming rule of Abadi and Needham; any further flaw localization or repair is particularized according to the error trace (Section 4).

Our rule corresponding to principle 3 from [1] checks whether the name of the sender and that of the receiver is explicitly present in the message or implicitly determined by the employed key (the sender by the signing key and/or the receiver by encryption key). For symmetric encryption, we consider that one of the names is given implicitly by the key if the other one is explicit in the message (to avoid a reflection attack).

To state the rules formally, consider a message $S \to R : M$ and let $\mathcal{A}(M)$ the set of agent names inferred from the message. We define $\mathcal{A}(M)$ by structural inductions as follows:

- $\mathcal{A}(A) = \{A\}$ iff A is an agent name
- $\mathcal{A}(M) = \emptyset$ if M is an atomic message but not an agent name
- $\mathcal{A}(M_1.M_2) = \mathcal{A}(M_1) \cup \mathcal{A}(M_2)$
- $\mathcal{A}(\{M\}_{sk(S)}) = \mathcal{A}(M) \cup \{S\}$
- $\mathcal{A}(\{M\}_{pk(R)}) = \mathcal{A}(M) \cup \{R\}$
- $\mathcal{A}(\{M\}_{k(S,R)}) = \begin{cases} \mathcal{A}(M) \cup \{S\} \text{ if } R \in \mathcal{A}(M) \\ \mathcal{A}(M) \cup \{R\} \text{ if } S \in \mathcal{A}(M) \\ \mathcal{A}(M) \text{ otherwise} \end{cases}$

M is valid according to principle 3 of [1] if and only if $\{S, R\} \subseteq \mathcal{A}(M)$.

We have implemented a prototype checker for this rule for protocols written in the SPDL language of the Scyther verifier [7] from ETH Zürich. Similarly to ASLan++, protocols are described from the perspective of each role, thus the Alice-Bob notation $S \to R : M$ would translate to two statements, namely $send\_n(S, R, M)$ in role S and $recv\_n(S, R, M)$ in role R, where n is a (numeric) label indicating the desired pairing of send and receive.

Analyzing the full set of protocols from the Scyther distribution we found that 29 out of 38 of the protocols or specifically 245 out of 262 protocol messages (93.5% of messages) adhered to this principle. The protocols that do not adhere to this principle such as the unfixed version of the Needham-Schroeder protocol, the TMN (Telecommunications Management Network) protocol, the Wide-Mouthed-Frog (two versions) and the Yahalom-Lowe protocol involve messages exchanged between three principals in which either the name of the sender or the name of the receiver is missing.

The same implementation can also analyze protocol described in the Alice-Bob (AnB) notation employed in the AVANTSSAR toolset, which can be directly handled by the OFMC model checker or translated into ASLan format. This check can be used as a preliminary step for the trace-based fault identification of Section 4.

3.1.1 Protocol Repair

Our implementation also attempts to repair protocols that do not observe the third principle of Abadi and Needham by explicitly providing the name of the sender, receiver, or both, when these names are found to be missing. When modifying the protocol so it conforms to the naming principle we perform minimal changes (such as placing the names of the participating principals explicitly in the message) and make sure that these changes do not contradict other rules in order not to introduce new vulnerabilities, e.g., by modifying a message into one with a structure resembling that of a different message.

For signed and encrypted messages, the required principal name(s) are added inside the encryption, in order to avoid the risk of splitting and recombining message parts. This is done if the enclosing cryptographic operation (encryption of signature) is available to the sending principal; otherwise, the identifying name is simply appended the signed or encrypted message.

We have applied this simple repair pattern (see also, e.g., [9]) on those protocols from the Scyther distribution that do not fulfill their security claims (while also disobeying the naming rule). As expected, we obtain Lowe's fix for the Needham-Schroeder protocol. However, there are also cases where this approach alone brings no improvements (WMF-Lowe, TMN and ISO-IEC-9798 protocols).

3.2 Dataflow-based fault localization

Missing message fields are not the only flaws that can lead to execution faults. Another category is missing checks (omitted either in protocol design, or due to flawed implementations) that can lead to unintended executions.

A series of highly relevant real-world examples has been recently described in [15]. The authors studied popular merchant applications (Interspire and NopCommerce) used by online stores to allow payment through third-party cashiers (Amazon Payments, Google Checkout and PayPal) and found flaws that allowed a malicious shopper to purchase an item at an arbitrarily low price, shop for free after paying for one item, or even avoid payment.

We have modeled several of these interactions and describe an analysis that can be used to detect the causes of such attacks.

3.2.1 Protocol models in SPDL

We have described several Cashier-as-a-Service case studies from [15] in SPDL, the input language of the Scyther protocol verifier [7]. We have chosen this approach because SPDL has a simple syntax, and Scyther provides au-

tomatic graphical visualization of attacks. Much like in ASLan++, protocol roles are described individually (each exchanged message appears both from sender and receiver point of view). Moreover, SPDL descriptions contain an explicit syntactic pairing of a set with a matching receive. This allows us to also treat errors where one side of the protocol is wrongly implemented, or where the protocol can be exploited by a dishonest participant. Since the concepts carry over to ASLan++, an adaptation can be easily done.

In SPDL, each protocol is composed of roles that are in turn comprised of declarations of constants and/or variables which are used as building blocks for the messages that describe each step of the protocol. Roles can communicate with each other via pairs of send and receive events, each of these pairs having one equivalent message in Alice-Bob notation (i.e., $A \rightarrow B : M \Leftrightarrow send\_i(A, B, M) \in A \wedge recv\_i(A, B, M) \in B$ where i is the current step of the protocol).

Variables and constants in SPDL can have different types such as nonces, tickets (that can match anything), or user-defined types.

As an example, the model of the Needham-Schroeder protocol from the Scyther distribution is included below.

```
protocol ns3(I,R)
{
  role I
  {
    fresh ni: Nonce;
    var nr: Nonce;

    send_1(I,R, {ni,I}pk(R) );
    recv_2(R,I, {ni,nr}pk(I) );
    send_3(I,R, {nr}pk(R) );

    claim(I,Secret,ni);
    claim(I,Secret,nr);
  }
  role R
  {
    var ni: Nonce;
    fresh nr: Nonce;

    recv_1(I,R, {ni,I}pk(R) );
    send_2(R,I, {ni,nr}pk(I) );
    recv_3(I,R, {nr}pk(R) );

    claim(R,Secret,ni);
    claim(R,Secret,nr);
  }
}
```

3.2.2 Analyzing protocol dataflow

We address flaws where at the end of a protocol run two variables (in the same or different roles) are supposed to have the same value, but this condition

is violated. SPDL provides a construct $match(V_1, V_2)$ to specify an equality requirement; this is written directly as an equality check in ASLan++.

A variable in the description of a role is bound the first time it occurs in a *recv* event to the value of the corresponding message field. Occurrences of constants or already bound variables in a *recv* event imply a check that their value in the receiving role is equal to the corresponding message field. We employ a unification process and store the constraints and the association of variables with their corresponding values. Ultimately, all variables in the protocol must be eventually bound either directly or indirectly to a constant (or more generally, a ground term).

Figure 1 provides an example of these bindings and how variable values are traced back to the initial constants, for the cashier-as-a-service protocol in Figure 2 from [15].

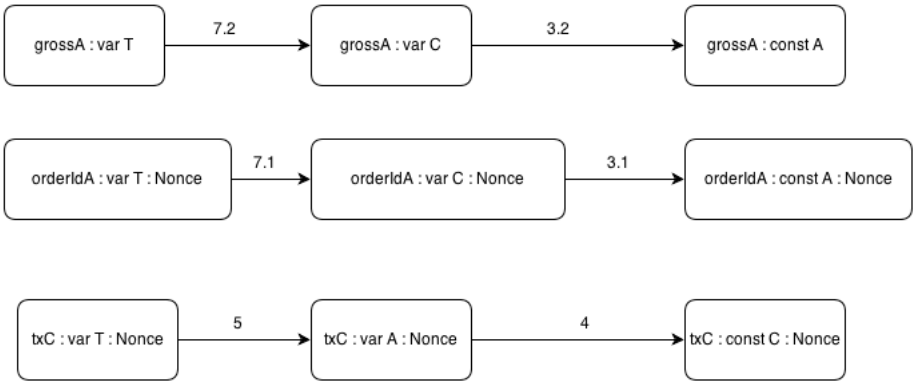


Figure 1: Data Flow

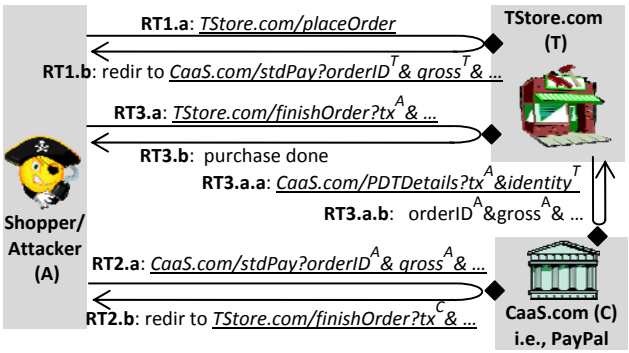
In the final step, the bindings for the variables required to have matching values are compared in the following way: if both are bound to the same constant terms, or to terms already constrained as being equal due to the protocol flow, the *match* constraint is considered to be satisfied. Otherwise, if the resulting constants are different, or not otherwise constrained to be equal, a potential violation of the protocol constraint is signaled.

If the equality claim is not satisfied we provide as counterexample a trace back to the initial constants. We can also determine the attacking role by assuming that the role with the inserted equality claim to be the potential victim (and therefore all its comprising declarations to be trustworthy). All other roles are considered to be untrusted. As a result we suspect that all constants and the variables ultimately bound to these constants in these other roles to be potentially hijacked or regarded as untrusted values that are under the control of the attacker.

3.2.3 Examples: Cashier-as-a-Service models

We have modeled several flawed cashier-as-a-service protocols described with their attacks in [15] and applied our analysis. The known flaws are reflected by omitting checks in the models. In a real analysis scenario, one would use mutation operators on the protocol specification to ascertain vulnerabilities if checks are omitted in the implementation.

Consider the integration of Paypal into NopCommerce (Figure 2), as described in [15]: In the first protocol step the customer A places an order.



$$
\begin{aligned}
A \to T &: \ orderId \\
T \to A &: \ orderId, grossT \\
A \to C &: \ orderId, grossA \\
C \to A &: \ tx \\
A \to T &: \ tx \\
T \to C &: \ tx, T \\
C \to T &: \ orderId, grossA
\end{aligned}
$$

Figure 2: Paypal integrated with NopCommerce from [15]

After inserting the order details into a database and marking the order as pending, the merchant T passes the order information ($orderId$ and $grossT$) back to the customer, and redirects his browser to the CaaS. The customer pays the CaaS, but according to the information ($orderId$ and $grossA$) sent by the customer. The CaaS stores the payment details and returns the transaction id to the customer, who finalizes the order with the store. The flaw in the code of the target store consists of not checking the gross amount received from the CaaS when getting the confirmation that payment has been finalized. Thus a dishonest customer can tamper with message 3 by sending $grossA$, and pay for a lesser amount that the cost of the item ordered.

```
protocol caas ( A, C, T )
{
  role A
  {
    const orderId : Nonce;
    const grossA;
    var txC: Nonce;
    var grossT;
    send_1(A, T, orderId );
    recv_2(T, A, orderId, grossT);
    send_3(A, C, orderId, grossA);
    recv_4(C, A, txC);
    send_5(A, T, txC);
  }
```

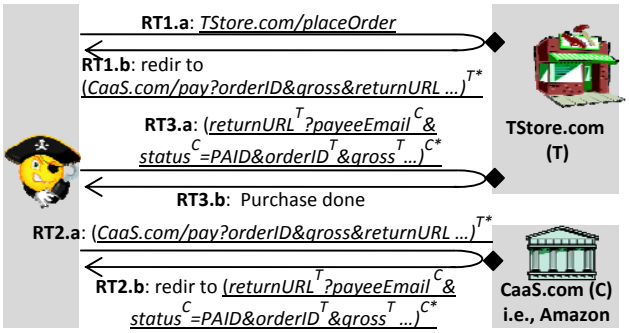
```
  role C
  {
    const txC : Nonce;
    var orderId : Nonce;
    var gross;
    recv_3(A, C, orderId, gross);
    send_4(C, A, txC);
    recv_6(T, C, txC, T);
    send_7(C, T, orderId, gross);
  }
  role T
  {
    const grossT;
    var grossA;
    var orderId, txC : Nonce;
    recv_1(A, T, orderId);
    send_2(T, A, orderId, grossT);
    recv_5( A, T, txC);
    send_6(T, C, txC, T);
    recv_7(C, T, orderId, grossA);
    match(grossT, grossA);
  }
}
```

Using our semantic analysis we are able to localize the source of the fault that allows the cutomer to pay less. By inserting an equality claim in the merchant code we can trace the values as they go back all the way to the client tampering with the gross amount (in message RT2.a, i.e., the third message, consequently reaching RT3.a.b which is message 7 in Figure 2). We illustrate the tool output in Figure 3. This leads us to hypothesis that the missing check for the gross amount in the merchant is the cause of the error. As expected, adding the check (by requiring the value $grossT$ on receipt of message 7) fixes the protocol.

```
EqualityClaim: grossT, grossA, 7, T
(tainting suspect) send(3,A,C,orderId,grossA); ->
(tainted) recv(3,A,C,orderId,gross); ->
(tainted) recv(7,C,T,orderId,grossA); ->
```

Figure 3: Example of Trace

We have also tested our analysis on other cashier-as-a-service protocols. Figure 4 depicts the integration of Amazon SimplePay into NopCommerce. This model is different in that all messages after initiating the order are signed and thus the shopper cannot tamper with the messages as in the previous protocol. The customer places an order and the merchant replies with a message that redirects the client's browser to the CaaS which verifies the merchant's signature before finalizing the transaction. Afterwards the CaaS redirects the client's browser to the merchant which checks the CaaS signature to confirm that the ordered item was indeed paid by the customer.



(1) $A \to T : orderId$
(2) $T \to A : \{orderId, gross, returnURL\}sk(T)$
(3) $A \to C : \{orderId, gross, returnURL\}sk(T)$ // *payee*
(4) $C \to A : \{returnURL, orderId, gross, payeeEmail, statusPaid\}sk(C)$
(5) $A \to T : \{returnURL, orderId, gross, payeeEmail, statusPaid\}sk(C)$

Figure 4: AmazonSimplePay integrated with NopCommerce from [15]

This apparently secure protocol is vulnerable when the malicious customer is also registered as a merchant. It is rather simple for anyone to open an account on Amazon and so after the attacker does this step he can also pay himself the money for the ordered purchase.

The protocol intent is for message (3) to be the redirection of message (2) which is signed by the merchant. However, as shown in the Scyther attack trace of Figure 5, it is possible for a malicious shopper to instead sign message (3) herself. Since the cashier C takes the signer to be the merchant to whom the payment is due, a malicious shopper who has previously registered as merchant can thus pay any purchase to herself. The cashier includes the payee's identity (e-mail) in the response (4), but if the merchant implementation does not use payeeEmail for security checks in (5), the fraud stays undetected and the merchant is convinced the order has been paid.

The error source can be localized if the protocol model is written making explicit that the signature in (3) represents the payee. In the shopper role:

```
send_!3(A,C,{orderId,gross,returnURL}sk(payee));
```

and in the cashier C:

```
recv_!3(A,C,{orderId,gross,returnURL}sk(payee));
send_4(C,A,{returnURL,orderId,gross,payee,statusPaid}sk(C));
```

By inserting the the equality claim *match(payeeEmail, T)* as desired specification in the merchant code, we can trace the value of `payeeEmail` all the way back to message (3), where it is bound to constant *payee*. Since this value

need not be T (it could be the identity of the shopper A), this constitutes the source of the error, and the inserted check is needed in the merchant code.

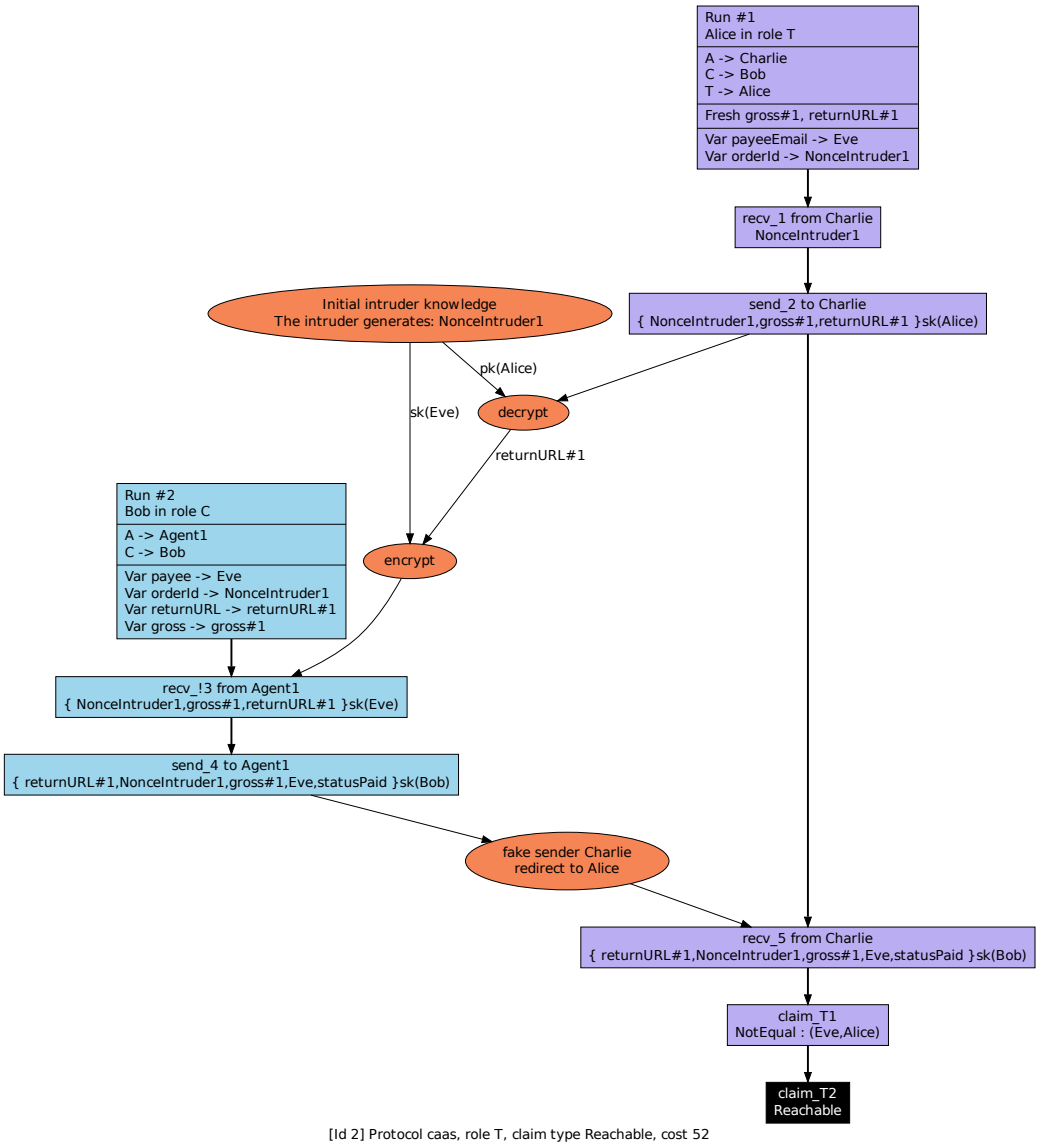


Figure 5: AmazonSimplePay integrated with NopCommerce attack

Concluding, this approach is useful in conjunction with applying mutations on an idealized model in order to suggest potentially flawed implementations (or variations due to a malicious agent not observing the protocol). The presented analysis can then be used to identify the missing checks that lead to the observed faults.

4 Trace-Based Fault Localization

In this section, we introduce our approach for fault localization in security protocols. It is a technique that analyzes the attack trace produced by the model-checker, followed by fault diagnosis and by attempts to repair the protocol in certain well-defined situations.

4.1 Overview

Consider a protocol P described in ASLan++. The protocol should conform to a security property ϕ, but its verification by model-checking finds an attack that violates the security goal. Thus, protocol P is faulty, but what exactly is the cause behind its failure and how could it be fixed?

We have developed a technique that aims at answering this question, at least in part. Our technique is based on a feedback loop with the phases: attack analysis, fault diagnosis, attempted repair and re-verification. In a nutshell, the method for fault localization and repair is described below.

Repeat the following steps until no new attack is found, or no diagnosis for the fault can be made:

- analyze the obtained attack trace together with the protocol P
- from this analysis, deduce possible root causes for the attack
- if a flaw is found, then
  - modify protocol P to repair the identified flaw
  - re-verify the repaired protocol against the security property ϕ
- if no root cause for the attack trace can be found, the algorithm stops (unsuccessfully).

Although the approach presented here is based on rules for protocol analysis, like the one in Section 3, there are several important differences between the two techniques. First of all, Section 3 uses the defined rules for a general static analysis and detection of potential flaws in the protocol, while this section uses the rules in a targeted way, to identify the root cause of an existing attack. Second, the approach here performs a feedback loop for repair and re-verification, attempting to fix the protocol until either no new attack is found, or no new repair opportunities are identified. In contrast, the techniques in Section 3 only perform one repair pass when encountering rule violations. Third, the only rule common to both methods is based on the third principle of Abadi and Needham [1], stating that explicitly naming sender and receiver in each message is essential for good protocol design.

4.2 Assumptions

Let V be the set of variables and W the set of constants used in the protocol, and let $n = |V|$, $m = |W|$. Let F be the set of functions known by the protocol, $F = F_1 \cup F_2 \cup ...$, where F_k is the set of k-ary function symbols.

Then, the set T of terms used in the protocol is defined as follows:

- each variable $v \in V$ is a term, thus $v \in T$
- each constant $w \in W$ is a term, thus $w \in T$
- for each k-ary function symbol $f \in F$, the application $f(t_1, t_2, ..., t_k)$ of this function to other terms, where $t_i \in T, i = \overline{1,k}$, is also a term, thus $f : T^k \rightarrow T$ and $f(t_1, t_2, ..., t_k) \in T$

Given a message M, a field of M is either a text or a primitive value that can be read by the message receiver, or an encrypted submessage of M which the receiver cannot decrypt. Let $Parts(M)$ be the set of all fields of M.

We define a session as a conversation, an interactive information exchange consisting of a sequence of message send and receive events between two or more actors. A session is considered established after the principals successfully exchange at least one message, and it ends when no message exchange takes place anymore. Each session has a unique session identifier, *SID*.

We introduce the following predicates:

- $send(S, R, SID, M)$: actor S, the sender, sends message M to actor R, the receiver, over the established session *SID*
- $recv(S, R, SID, M)$: actor R, the receiver, accepts message M from actor S, its sender, over the established session *SID*
- $knows(A, X)$: honest agent A knows X, which can be a message sent over the network, a message field, or any other term in the protocol
- $iknows(X)$: the intruder knows X, which can be a message sent over the network, a message field, or any other term in the protocol
- $fresh(X)$: X is a fresh value that was generated in the current run of the protocol
- $origin(S, M)$: actor S is the original sender of message M
- $signed(A, M)$: actor A has signed message M with his secret key
- $sym(A, B, M)$: message M is encrypted with a symmetric key used for the communication between A and B
- $asym(A, M)$: message M is encrypted with the public key of A
- $check(M, C, F)$: the receive condition C verifies the presence and value of field F in message M, i.e. the set $\mathcal{C}(C)$ of atomic constraints in condition C contains at least one test on the value of field F

A protocol P allows for a set of possible sequences of abstract, parameterized communication steps between the actors participating in the protocol

and/or between an actor and the intruder. A communication step implies sending and receiving a message between the two parties. An attack trace found by the model-checker is a sequence of concrete protocol steps that ends in a bad state, i.e. a state that violates the desired security property ϕ. Assume Θ is the translation of protocol P to a logic formula, as done in [6]. The model-checker employs a SAT solver to properly instantiate the variables in the protocol steps, by solving the conjunction of $\neg\phi$, the formula describing bad states, with formula Θ, thus obtaining the concrete attack trace.

4.3 Violation of non-injective agreement

The communication between actors is modeled by the two following patterns, where $X \hookrightarrow Y$ means that X is a precondition of Y:

1. $send(S, R, SID, M) \hookrightarrow iknows(M)$
2. $C \wedge iknows(M) \hookrightarrow recv(S, R, SID, M)$

Here, C is the message receive condition $C = c_1(x_1, x_2, ...) \wedge c_2(x_1, x_2, ...) \wedge ... \wedge c_k(x_1, x_2, ...)$, where all atomic conditions $c_i : T^{k_i} \rightarrow \{0, 1\}$ are predicates over the terms $x_i \in T$.

An honest, legitimate communication step must observe both rule 1 and rule 2, in that order. However, in a communication step where a message is replayed by an attacker, the message send event can be missing, as the received message is forwarded or generated by the intruder rather than coming from a send action performed by an honest participant.

We introduce predicate $replayed(M)$, which evaluates to true iff message M is replayed by the attacker. According to the observations above, we express message replay as follows:

$$recv(S, R, SID, M) \wedge \neg send(S, R, SID, M) \Rightarrow replayed(M)$$

The receive events in the attack trace for which the above condition holds will be referred to as vulnerable message receive events.

The above condition implies the violation of non-injective agreement [10], which is always a flaw in the protocol design, and must be fixed.

4.4 Agent naming

According to Abadi and Needham's third principle [1], a message should explicitly specify the names of all agents which are important for its meaning. This includes, but is not limited to, all its senders and receivers.

Let $\mathcal{A}(M)$ be the set of all agents A mentioned in message M, i.e., $A \in Parts(M)$. Let O be the origin of message M. In a system with two or more

agents, a message can sometimes be resent by other honest agents than its original sender, which is why we choose to distinguish between the message sender and the message origin. Then, one of the following should be true for all the conversation participants explicitly mentioned in M, and let this condition be encoded by the predicate $good(M)$:

- $sym(O, R, M)$:
  $recv(S, R, SID, M) \Rightarrow S \in \mathcal{A}(M) \wedge check(M, C, S) \vee R \in \mathcal{A}(M) \wedge check(M, C, R)$. The message is encrypted with the symmetric key of O and R; thus, if the key is not compromised only these two agents have access to its content. To avoid replay attacks, the message should explicitly include and check the name of either the sender or the receiver. While including a name, e.g. $S \in A(M)$ also implies $check(M, C, S)$ in Alice-Bob notation, this need not happen in ASLan++, where send and receive are specified separately from the viewpoint of each agent.
- $signed(O, M)$:
  $recv(S, R, SID, M) \Rightarrow R \in \mathcal{A}(M) \wedge check(M, C, R)$. If the original sender has signed the message with its secret key, then all its recipients should be mentioned in message M.
- $\neg signed(O, M) \wedge \neg sym(O, R, M)$:
  $recv(S, R, SID, M) \Rightarrow S \in \mathcal{A}(M) \wedge check(M, C, S) \wedge asym(R, M)$. If the message is not signed by its original sender, it should mention its sender (each sender, if it is resent), where M is encrypted with the receiver's public key.

When $good(M)$ is true we assume that all agents involved in the exchange of message M are explicitly specified in the message, and verified as such on message receive. However, this is a heuristic approximation, as the presence of an agent name A in M may be due to some other reason and have a different semantics than specifying the message sender or receiver. Since the intended reason is hard to determine, we do not address this situation; thus our approach is not complete.

In order to deduce and, whenever possible, repair the root cause of the attack, we identify the last message M for which $replayed(M) \wedge \neg good(M)$.

- if $sym(O, R, M) \wedge origin(O, M)$ occurs for a vulnerable event of the form $recv(A, R, SID, M)$, or $recv(S, A, SID, M)$, where agent A is either the message sender or its receiver, and if in both cases $A \in \mathcal{A}(M) \wedge \neg check(M, C, A)$, then, although symmetric cryptography is employed and the sender or the receiver is explicitly specified in the message, the field encoding the latter information is not tested when

message M is received (although this can not occur in Alice-Bob notation, it can happen in ASLan++).
Also, if $signed(O, M) \wedge origin(O, M) \wedge A \in \mathcal{A}(M)$ for the receive event $recv(S, A, SID, M)$, and $\neg check(M, C, A)$, the recipient A is not tested at the vulnerable receive event, although it is present in message M.
Likewise, if $\neg signed(O, M) \wedge A \in \mathcal{A}(M) \wedge asym(R, M)$ for the event $recv(A, R, SID, M)$, but we have $\neg check(M, C, A)$, it means that the sender has been specified in the message, but is not tested on receive.
In all three cases, we add the new constraint $c' = A \in \mathcal{A}(M)$ to the message receive condition C, which thus becomes $C' = C \wedge c'$ (as mentioned before, this situation and its fix apply to ASLan++ models, but they cannot be expressed in Alice-Bob notation).

- if $sym(O, R, M) \wedge origin(O, M)$ and for the vulnerable receive event $recv(A, R, SID, M)$ we have $A, R \notin \mathcal{A}(M)$, then, although the replayed message M is symmetrically encrypted, neither sender nor receiver name are included in M, which facilitates the attack.
  Also, if $signed(O, M) \wedge origin(O, M) \wedge A \notin \mathcal{A}(M)$ for the receive event $recv(S, A, SID, M)$, then, while the original sender of message M has signed message M, the deceived recipient of the message is not specified in M, which enables the intruder to replay M.
  Likewise, if for the replayed message $\neg signed(O, M) \wedge asym(R, M) \wedge \nexists A \in \mathcal{A}(M)$ s.t. $origin(A, M)$, then the original sender of message M has neither signed the message, nor is it specified in M, which enables the intruder to replay M.
  In all three cases, we apply the following fix: replace M by M′, so that $Parts(M') = Parts(M) \cup \{A\}$, and add constraint $c' = A \in Parts(M')$ to the receive condition C, which thus becomes $C' = C \wedge c'$. For the case of $sym(A, R, M)$, the name of the sender is the one chosen for inclusion in the repaired message.
- re-verify the protocol against property ϕ
- if another attack is found, then we have two possible situations:
  - the repaired message M is still replayed ⇒ analyze and repair the previous replayed message in the attack trace
  - the repaired message M is no longer replayed ⇒ repeat the procedure, until no attack is found, or there are no more replayed messages M with $\neg good(M)$

If an attack is still found for the protocol, although for all messages M for which $replayed(M)$, predicate $good(M)$ evaluates to true, then:

- the semantics of having $A \in Parts(M)$, for an agent A, may not be the

assumed one of mentioning the message sender or receiver, or it is also possible that A may be replaced or faked by the intruder
- or the attack may be due to another cause (e.g., lack of session binding, as treated subsequently, etc.)

4.5 Examples with lack of agent naming

4.5.1 NSPK

As a first example for the diagnosis and repair of this type of replay attacks, let us consider the classic NSPK protocol, as described below. For simplification, we present this protocol in Alice-Bob notation rather than ASLan++. This makes the example easier to read and understand.

$$\begin{aligned} A &\longrightarrow B : & \{N_a, A\}_{pk(B)} \\ B &\longrightarrow A : & \{N_a, N_b\}_{pk(A)} \\ A &\longrightarrow B : & \{N_b, A\}_{pk(B)} \end{aligned}$$

The attack found for the NSPK protocol is the well-known one:

$$\begin{aligned} A &\longrightarrow I : & \{N_a, A\}_{pk(B)} \\ I &\longrightarrow B : & \{N_a, A\}_{pk(B)} \\ B &\longrightarrow I : & \{N_a, N_b\}_{pk(A)} \\ I &\longrightarrow A : & \{N_a, N_b\}_{pk(A)} \\ A &\longrightarrow I : & \{N_b, A\}_{pk(I)} \end{aligned}$$

The diagnosis and repair process proceeds as follows. Let the first session between A and $I(B)$ be identified by SID=1, and the second one, between $I(A)$ and B, by SID=2.

The last replayed message is clearly $\{N_a, N_b\}_{pk(A)}$. We are in the following situation for message $M = \{N_a, N_b\}_{pk(A)}$:

$$replayed(M) \wedge \neg signed(B, M) \wedge recv(B, A, 1, M) \wedge B \notin \mathcal{A}(M)$$

As the message sender field is missing, the repaired message will contain this additional field and will be $\{B, N_a, N_b\}_{pk(A)}$, where the presence of the sender's name is also verified on message receive.

4.5.2 SAML-SSO

The second example presented is the SAML Single Sign-On protocol. Slightly more complex than the previous example, it is a 3-party protocol, having

principal *IDP*, the identity provider, as the trusted third party. Here, *IDP* signs the messages it sends, and the flaw allowing for the replay attack is the lack of the destination's name in the replayed message.

The SAML-SSO protocol in Alice-Bob notation is shown below:

$$
\begin{aligned}
C \longrightarrow SP: &\quad URI \\
SP \longrightarrow C: &\quad IDP, \{N, SP\}_{pk(IDP)}, URI \\
C \longrightarrow IDP: &\quad C, IDP, \{N, SP\}_{pk(IDP)}, URI \\
IDP \longrightarrow C: &\quad SP, pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
C \longrightarrow SP: &\quad pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
SP \longrightarrow C: &\quad URI, Resource
\end{aligned}
$$

We also give a fragment of the ASLan++ model for the SAML-SSO protocol:

Listing 1: SAML-SSO model fragment

```
%%%%%%%%%%%%%%%% CLIENT %%%%%%%%%%%%%%%%
entity Client (Actor, SP: agent, URI : uri) {
  symbols
    IdP          : agent;
    AReq, ARsp   : message;
    Resource     : text;
  body {
    %% C-SP (1)
    Actor  ->* SP     : httpReq (get, URI, Actor, nil);
    SP     *->  Actor : httpResp(redirect, ?IdP.?AReq, nil);

    %% C-IdP
    Actor *->* IdP    : httpReq (get, IdP, Actor.IdP.AReq, nil);
    IdP   *->* Actor  : httpResp(ok , nil, SP.?ARsp);

    %% C-SP (2)
    Actor  ->* SP     : httpReq (post, SP, nil, ARsp);
  channel_goal auth_C_on_uri:
    Actor *-> SP      : URI;
    SP    *->* Actor  : httpResp(ok, URI, ?Resource);
  channel_goal secrecy_and_auth_on_Resource:
    SP    *->* Actor  : Resource;
  }
}

%%%%%%%%%%%%%%%% IDENTITY PROVIDER %%%%%%%%%%%%%%%%
entity IdentityProvider (Actor: agent, TrustedSPs: agent set) {
  symbols
    C, SP: agent;
    ID: nat;
    URI : uri;
  body {
    %% is there another C in the message?
    ?C *->* Actor  : httpReq (get, Actor,
                     ?C.Actor.authReq(?ID, ?SP).?URI, nil);

    % The challenge by IdP and correct reponse by C to
    % authenticate C is abstracted away here,
```

```
    % but the auth. of C to IdP is modeled by using ?C *->...
    %% check whether SP is a trusted SP
    if (TrustedSPs->contains(SP)) {
      Actor *->* C : httpResp(ok, nil,
            SP.authResp(pk(Actor), ({C.Actor}_inv(pk(Actor)))).URI);
    }
  }
}

%%%%%%%%%%%%%%%% SERVICE PROVIDER %%%%%%%%%%%%%%%%
entity ServiceProvider (Actor, IdP: agent, URIs : uri set) {
  symbols
    C           : agent;
    URI         : uri;
    ID          : nat;
    AA          : message;
    ConsumedAAs : message set;
    Resource    : text;
  body {
    select {
      on(?C  ->* Actor : httpReq(get, ?URI, ?C, nil)
                       & URIs->contains(?URI)): {}
                 %% check whether the URI belongs to the SP
    }
    ID := fresh();
    Actor   *->  C   : httpResp(redirect, IdP.authReq(ID, Actor).URI, nil);
    select {
      on(?C  ->* Actor : httpReq (post, Actor, nil,
                                  authResp(pk(IdP), ?AA).?URI)
                    & ?AA = {?C.IdP  }_inv(pk(IdP))
                    & !ConsumedAAs->contains(?AA)):
                 %% check if the assertion has not been consumed
      channel_goal auth_C_on_uri:
        C *->  Actor : URI; {}
    }
    ConsumedAAs->add(AA); %% mark the assertion as consumed
    Resource := fresh();
    Actor   *->* C   : httpResp(ok, URI, Resource);
  channel_goal secrecy_and_auth_on_Resource:
    Actor   *->* C   : Resource;
  }
}
```

The attack found is as follows:

$$\begin{aligned} C \longrightarrow I : &\quad pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\ I \longrightarrow SP : &\quad pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\ SP \longrightarrow I : &\quad URI, Resource \end{aligned}$$

The intruder impersonates the client towards the service provider by replaying message $pk(IDP), \{C, IDP\}_{sk(IDP)}, URI$, and thus it gets to access the resource on the honest client's account. As the replayed message is signed by IDP, we are in the following situation, for $M = \{C, IDP\}_{sk(IDP)}$:

$$replayed(M) \wedge signed(IDP, M) \wedge recv(C, SP, SID, M) \wedge SP \notin \mathcal{A}(M)$$

As the replayed message does not specify its final destination, i.e., the name of the service provider, the repaired message will contain this additional

field, under the signature of IDP: $\{C, IDP, SP\}_{sk(IDP)}$, where the presence of the message recipient's name is also verified on message receive. After this repair, no new attack trace is found by the model-checker, therefore we conclude that the protocol was correctly fixed.

A fragment of the repaired protocol in ASLan++ is shown below:

Listing 2: SAML-SSO repaired model fragment

```
%%%%%%%%%%%%%%%% IDENTITY PROVIDER %%%%%%%%%%%%%%%%
entity IdentityProvider (Actor: agent, TrustedSPs: agent set) {
  symbols
    C, SP: agent;
    ID: nat;
    URI : uri;
  body {
    ?C *->* Actor  : httpReq (get, Actor,
                     ?C.Actor.authReq(?ID, ?SP).?URI, nil);

    if (TrustedSPs->contains(SP)) {
      Actor *->* C : httpResp(ok, nil,
            SP.authResp(pk(Actor), ({C.Actor.SP}_inv(pk(Actor)))).URI);
    }
  }
}

%%%%%%%%%%%%%%%% SERVICE PROVIDER %%%%%%%%%%%%%%%%
entity ServiceProvider (Actor, IdP: agent, URIs : uri set) {
  symbols
    C           : agent;
    URI         : uri;
    ID          : nat;
    AA          : message;
    ConsumedAAs : message set;
    Resource    : text;
  body {
    select {
      on(?C  ->* Actor : httpReq(get, ?URI, ?C, nil) &
                         URIs->contains(?URI)): {}
    }
    ID := fresh();
    Actor   *->  C   : httpResp(redirect,
                            IdP.authReq(ID, Actor).URI, nil);
    select {
      on(?C  ->* Actor : httpReq (post, Actor, nil,
                                  authResp(pk(IdP), ?AA).?URI)
                         & ?AA = {?C.IdP.Actor}_inv(pk(IdP))
                         & !ConsumedAAs->contains(?AA)):
                      %% check if the assertion has not been consumed
      channel_goal auth_C_on_uri:
        C *->  Actor : URI; {}
    }
    ConsumedAAs->add(AA); %% mark the assertion as consumed
    Resource := fresh();
    Actor   *->* C   : httpResp(ok, URI, Resource);
  channel_goal secrecy_and_auth_on_Resource:
    Actor   *->* C   : Resource;
  }
}
```

4.6 Session binding

A message replay attack can also be caused by the lack of session binding for one or several of its messages. This enables the attacker to reuse messages from a previous session in a replay attack.

In this subsection, we address the case in which the names of the participating entities are correctly specified in the message and checked on message receive, however the messages are still used for replay attacks. In such situations, we investigate whether the attack can be avoided by ensuring that all actors are alive, i.e., are executing the current protocol session.

Let M be a message for which $good(M)$ evaluates to true.

We define predicate $bound(M)$, meaning that message M is bound to a session, as follows: $\exists N \in Parts(M) \wedge (fresh(N) \vee (N = f(N') \wedge fresh(N')) \wedge (knows(S,N) \vee knows(S,N')) \wedge (knows(R,N)) \vee knows(R,N')) \wedge check(M,C,N)$. Thus, we say that message M is bound to a current session if it contains a field N which is a fresh value, so that either nonce N or a function of N is known by both the sender and the receiver of message M, and the receive condition C of message M verifies the value of N (or $f(N)$).

If $bound(M)$ is true, we assume that the fresh field N is used to bind message M to the current session; however, this is a heuristic notion of intent, as it is still possible that this field has some different semantics. Thus, we limit ourselves to fixing protocols where this design rule is not observed and $bound(M)$ is false.

Whenever $N = f(N')$, it is assumed that both parties S and R know the function f used to modify the exchanged nonce.

If the attack trace involves several sessions and condition $replayed(M) \wedge send(S, R, SID_1, M) \wedge recv(S, R, SID_2, M) \wedge SID_1 \neq SID_2$ is true, we must explore the possibility that M is not bound to the current session.

In this case, the attack diagnosis and protocol repair proceed as follows:

- start with the last replayed message M for which $bound(M)$ is false
- if the nonce N has already been exchanged between the participants and appears in the message M, but is not checked on receive, i.e. $\exists N \in Parts(M) \wedge (fresh(N) \vee (N = f(N') \wedge fresh(N'))) \wedge (knows(S, N) \vee knows(S, N')) \wedge (knows(R, N)) \vee knows(R, N')) \wedge \neg\, check(M, C, N)$, then we have a case of insufficiently constrained message receive.
  We force the verification of field N on receive, adding the new constraint $c' = N \in Parts(M)$ to the message receive condition C, which becomes $C' = C \wedge c'$.
- else, if message M has no such fresh field N, then it is not bound to the current session and the intruder may replay it in/from another session;

thus, in order to prevent the message replay attack we need to apply the following changes to the protocol:

  - introduce an authentication step prior to the exchange of message M, step in which S and R agree on a nonce N; if such a step already exists in the protocol, then move it prior to the exchange of M; if it already exists and it precedes the exchange of message M, then only apply the next modification
  - replace M by M' with $Parts(M') = Parts(M) \cup \{N\}$ (or $Parts(M') = Parts(M) \cup \{f(N)\}$), and for each event $recv(A, R, SID, M)$, update the message receive condition C with a new constraint c' that checks the value of the exchanged nonce: $c' = N \in Parts(M')$ (or $c' = f(N) \in Parts(M')$), and the new condition is $C' = C \wedge c'$.

- then, verify again the repaired protocol against property ϕ
- if another attack is found, then we have two possible situations:
  - the repaired message M is still replayed $\Rightarrow$ analyze and repair the previous replayed message in the attack trace
  - the repaired message M is no longer replayed $\Rightarrow$ repeat the above procedure until no attack is found, or there are no more replayed messages M for which $\neg\, bound(M)$

If an attack is still found for the repaired protocol, although all messages M for which $replayed(M)$ is true are now bound to the current session, then, similarly to the case of bad agent naming (Section 4.4), we have two situations:

- either for one such message M the semantics of having the nonce $N \in Parts(M)$ is different than the assumed one, or it is also possible that the nonce may be replaced or faked by the intruder, just like it can also happen for the agent name fields encountered in subsection Section 4.4
- or there is another cause behind the replay attack, for which we don't have a repair strategy, in which case the method stops unsuccessfully.

4.7 Example with lack of session binding

4.7.1 SAML-SSO

The lack of session binding is also to be found in the SAML Single Sign-On protocol. Below, we present this protocol again, in Alice-Bob notation:

$$
\begin{array}{ll}
C \longrightarrow SP: & URI \\
SP \longrightarrow C: & IDP, \{N, SP\}_{pk(IDP)}, URI \\
C \longrightarrow IDP: & C, IDP, \{N, SP\}_{pk(IDP)}, URI \\
IDP \longrightarrow C: & SP, pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
C \longrightarrow SP: & pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
SP \longrightarrow C: & URI, Resource
\end{array}
$$

The attack found is the same as before:

$$
\begin{array}{ll}
C \longrightarrow I: & pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
I \longrightarrow SP: & pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
SP \longrightarrow I: & URI, Resource
\end{array}
$$

Given the protocol and the attack, we observe that the replay can be blocked by a session binding repair step, as the following is true:

$$replayed(\{C, IDP\}_{sk(IDP)}) \wedge \neg bound(\{C, IDP\}_{sk(IDP)})$$

In this case, in order to repair the protocol, one must ensure a three-party authentication, by making the client C agree with the service provider SP and the identity provider IDP on the same nonce N (on which only SP and IDP agree on in the original, flawed protocol).

Thus, the fixed protocol is presented below:

$$
\begin{array}{ll}
C \longrightarrow SP: & URI \\
SP \longrightarrow C: & IDP, \{N, SP\}_{pk(IDP)}, URI, N \\
C \longrightarrow IDP: & C, IDP, \{N, SP\}_{pk(IDP)}, URI, N \\
IDP \longrightarrow C: & SP, pk(IDP), \{C, IDP\}_{sk(IDP)}, URI \\
C \longrightarrow SP: & pk(IDP), \{C, IDP\}_{sk(IDP)}, URI, N
\end{array}
$$

After the fix, no new attack trace is found by the model-checker.

A relevant fragment of the repaired protocol in ASLan++ is shown below:

Listing 3: SAML-SSO repaired model fragment

```
%%%%%%%%%%%%%%%% CLIENT %%%%%%%%%%%%%%%%
entity Client (Actor, SP: agent, URI : uri) {
  symbols
    IdP          : agent;
    AReq, ARsp   : message;
    Resource     : text;
    N: nat;
  body {
    %% C-SP (1)
    Actor  ->* SP    : httpReq (get, URI, Actor, nil);
    SP    *->  Actor : httpResp(redirect, ?IdP.?AReq, ?N);

    %% C-IdP
    Actor *->* IdP   : httpReq (get, IdP, Actor.IdP.AReq, N);
    IdP   *->* Actor : httpResp(ok , nil, SP.?ARsp);

    %% C-SP (2)
    Actor  ->* SP    : httpReq (post, SP, nil, ARsp.N);
  channel_goal auth_C_on_uri:
    Actor *-> SP     : URI;
    SP    *->* Actor : httpResp(ok, URI, ?Resource);
  channel_goal secrecy_and_auth_on_Resource:
    SP    *->* Actor : Resource;
  }
}
%%%%%%%%%%%%%%%% IDENTITY PROVIDER %%%%%%%%%%%%%%%%
entity IdentityProvider (Actor: agent, TrustedSPs: agent set) {
  symbols
    C, SP: agent;
    N: nat;
    URI : uri;
  body {
    %% is there another C in the message?
    ?C *->* Actor  : httpReq (get, Actor,
                              ?C.Actor.authReq(?N, ?SP).?URI.?N, nil);

    %% check whether SP is a trusted SP
    if (TrustedSPs->contains(SP)) {
      Actor *->* C : httpResp(ok, nil,
            SP.authResp(pk(Actor), ({C.Actor}_inv(pk(Actor)))).URI);
    }
  }
}
%%%%%%%%%%%%%%%% SERVICE PROVIDER %%%%%%%%%%%%%%%%
entity ServiceProvider (Actor, IdP: agent, URIs : uri set) {
  symbols
    C            : agent;
    URI          : uri;
    N           : nat;
    AA           : message;
    ConsumedAAs : message set;
    Resource     : text;
  body {
    select {
      on(?C  ->* Actor : httpReq(get, ?URI, ?C, nil) &
                         URIs->contains(?URI)): {}
                    %% check whether the URI belongs to the SP
    }
    N := fresh();
    Actor   *->  C   : httpResp(redirect, IdP.authReq(N, Actor).URI.N, nil);
```

```
    select {
      on(?C  ->* Actor : httpReq (post, Actor, nil,
                              authResp(pk(IdP), ?AA).?URI.N)
                       & ?AA = {?C.IdP  }_inv(pk(IdP))
                       & !ConsumedAAs->contains(?AA)):
            %% check if the assertion has not been consumed
      channel_goal auth_C_on_uri:
        C *->  Actor : URI; {}
    }
    ConsumedAAs->add(AA); %% mark the assertion as consumed
    Resource := fresh();
    Actor    *->* C    : httpResp(ok, URI, Resource);
  channel_goal secrecy_and_auth_on_Resource:
    Actor    *->* C    : Resource;
  }
}
```

4.8 Minimal unsatisfiable core

Assume the desired property ϕ is a safety property and that protocol P has at most k steps. Then, each term $x \in T$ has k different instances, x^1, x^2, ..., x^k, and we have k instances $T^1, T^2, ...T^k$ of the term set T.

Let $Vars(x)$ be the set of variables involved in the definition of term x. As the term x has k instances, the same happens for each variable $v \in Vars(x)$, and we have the variable instances v^1, v^2, ..., v^k, from the corresponding variable sets $V^1, V^2, ...V^k$, instances of the variable set V.

It is known that protocol P can be written as a boolean formula θ in conjunctive normal form $\theta = \bigwedge_i (\bigvee_j p_i^j(x, x', ...))$, where each predicate p_i^j is considered over the term instances in the union set $T^1 \cup T^2 \cup ... \cup T^k$. Similarly, the desired security property ϕ can also be expressed in a CNF, as $\phi = \bigwedge_i (\bigvee_j q_i^j(x, x', ...))$, where the predicates q_i^j are also over the set $T^1 \cup T^2 \cup ... \cup T^k$.

Both θ and ϕ are abstract, parameterized formulas. Solving the formula $\theta \wedge \neg\phi$ when it is satisfiable gives us the attack trace t that leads to a state in which formula $\neg\phi$ is true, and thus the desired security property is violated.

If we express the attack trace as a similar boolean formula α in CNF, we obtain a concrete formulawith no disjunctions: $\alpha = \bigwedge_i r_i(a, a', ...)$, with a, a', ... $\in T_C$, where T_C is the set of concrete terms obtained by assigning each variable instance in $V^1 \cup V^2 \cup ... \cup V^k$ with a concrete value during solving.

An unsatisfiable core of an unsatisfiable formula λ is a subset of the clauses in λ with their conjunction still unsatisfiable. A minimal unsatisfiable core of λ, denoted MUC(λ), is an unsatisfiable core for which the removal of any clause would result in a satisfiable conjunction of the remaining clauses [8].

Let us now consider the formula $\phi \wedge \alpha$, the conjunction of the desired property with the formula describing the attack trace. Since the attack

trace violates ϕ by definition, $\phi \wedge \alpha$ is unsatisfiable. This means we can compute a minimal unsatisfiable core MUC($\phi \wedge \alpha$) for it, which will contain a minimal subset of the conflicting clauses that have determined the formula's unsatisfiability. These conflicting clauses will provide us information on what is really wrong in the attack trace we have obtained.

We use the minimal unsatisfiable core MUC($\alpha \wedge \phi$) to diagnose the attack. The root cause deduction and repair method in this case works as follows:

Find all clauses $\gamma \in$ MUC($\alpha \wedge \phi$) for which $(\gamma \in \phi) \wedge (\neg\gamma \in \alpha)$. Let $Vars(\gamma)$ be the set of all variables in a clause γ. Then, we proceed as follows:

- for all variables $v \in Vars(\gamma)$ compute their corresponding sets of reaching definitions $RD(v)$
- choose variable v with the last reaching definition in the attack trace t
- if $RD(v)$ in the attack formula α is the event $recv(S, R, SID, M)$ with $v \in Parts(M)$, we conclude that the receive of M is underconstrained. Add γ to C so that $\gamma \wedge C \wedge iknows(M) \hookrightarrow recv(S, R, SID, M)$.
- otherwise, if $RD(v)$ corresponds to the value of the variable being internally generated or modified, then we mark that definition for manual inspection and continue with variable v', for which $RD(v')$ is the last executed in t prior to $RD(v)$
- verify the repaired protocol for executability, to see if a correct execution scenario, i.e. a scenario that conforms to ϕ, is still feasible
- if not, the performed repair is too restrictive and should be dropped, continue with variable v' whose reaching definition precedes the one of v on the attack trace t; else continue with the next step
- verify again the repaired protocol against property ϕ.
- if after re-verification, a new attack trace is found, this implies a new attack formula α', and the procedure is repeated for the minimal unsatisfiable core MUC($\alpha' \wedge \phi$) until no more attack is found.

The above procedure is specifically designed for local conditions, but γ might also be a global condition. A solution to enforce γ exists also in this case. This is done by introducing new message exchanges between the actors so that the state information relevant to γ is shared, and γ becomes a local condition, that can be verified by one of the participants.

Assume $v, v' \in Vars(\gamma)$ are local variables of agents A and A', respectively. Then, proceed as follows to enforce γ on v and v':

- assume variable v is defined after variable v' in the attack trace t
- we can force A' to share its relevant state, here represented by variable v', with A, so that the global condition can be checked in A. We do this by adding a message exchange $A' \longrightarrow A$: M' s.t. $v' \in Parts(M')$:

  - insert message event $send(A', A, SID, M')$ right after the reaching definition $RD(v')$ in the behavior description of actor A'
  - insert message event $recv(A', A, SID, M')$ right before the reaching definition $RD(v)$ in the behavior description of actor A
- add γ as a new constraint to the message receive condition C in the description of actor A, s.t. $\gamma \wedge C \wedge iknows(M) \hookrightarrow recv(S, A, SID, M)$

One particularly interesting case of the above global state condition may appear after the protocol is repaired with respect to a participant A. After fixing A, the other actors may be able to successfully finish their execution without being aware of the fact that A, due to the newly imposed constraints, has stopped. The solution is to introduce right after the constraining fix, in A, and as a last statement in the other actors, a message exchange between A and all other actors whose purpose is to let the other participants know whether A has successfully finished its execution.

The verification, diagnosis, repair and re-verification process ends when either no new attack is found, or we have exhausted all situations described above in which we have a repair strategy to apply.

Consider the following particular case:

$$\text{MUC}(\phi \wedge \alpha) \Rightarrow x_\phi \in D \wedge (x_\alpha = a \wedge a \notin D)$$

where a is a concrete value, D is a restricted domain for one or more of the instances x_i of variable x at protocol step i, x_α is the concrete value of x in the attack trace and x_ϕ is the symbolic value of x defined by property ϕ.

In this case, for each variable x for which this type of conflict exists, we apply the following method to resolve it:

- assume that in the attack trace the ϕ the violation of property ϕ is observed at step j of protocol P, thus the instance of x involved is x^j
- since for the initial receive condition C we have $C \not\Rightarrow x^j \in D$, add the new constraint $x \in D$ to C: $C = C \wedge \{x \in D\}$ to restrict the domain of x and avoid the attack
- verify the new protocol P', obtained after applying this repair, for executability (to ensure that the applied fix is not too restrictive)
- if the fix is too restrictive, then drop it and:
  - compute the set of reaching definitions $RD(step_j)$ in protocol P
  - $\forall d \in RD(step_j)$ for which $d : x = f(y)$, with $f : SD' \rightarrow SD$ an invertible function, $D \subseteq SD$ and $f^{-1}(D) \subseteq SD'$, the domain of y is restricted to $f^{-1}(D)$ to achieve restriction of x's domain to D.
  - for definitions with several independent variables, $d' \in RD(step_j)$ with $d' : x = f'(y_0, y_1, ..., y_{p-1})$, restricting the domain of x is very difficult, and thus we do not attempt it.

  - verify the new protocol for executability
  - if the new constraints on y are too restrictive, drop them and compute new set of $RD(d)$ and/or $RD(d')$, etc.
- else, keep the fix and further employ the new, fixed protocol
- continue with the next variable $x' \in \text{MUC}(\phi \wedge \alpha)$.

After applying these fixes for all variables in the minimal unsatisfiable core, verify the new protocol P'' against the original property ϕ. If a new attack trace is found, compute the new $\text{MUC}(\phi \wedge \alpha)$ and similarly eliminate its new conflicts. The repair-feedback loop continues until no new attack is found.

4.9 MUC-based fault localization example

4.9.1 CaaS: Cashier as a Service

Cashier as a Service is one of the real-life case studies presented in [15]. It is composed of three entities: an online shop, the online cashier service and a client. Here, the vulnerability is due to the lack of proper verification of the price field: the online shop doesn't check if the amount paid by the client to the cashier is the same as the price of the acquired item.

Below, we present an essential fragment from the ASLan++ description of the shop's behaviour in this system:

```
?Client -> Actor: corder(?OID);
Actor -> Client: morder(OID, price1);
Caas -> Actor: rorder(OID, ?Price);
assert okPayment: Price=price1;
```

The desired property ϕ specifically asks for the price of the item to be the same as the amount paid by the client:

$$\phi : Price = price1$$

The minimal unsatisfiable core found is

$$\text{MUC}(\alpha \wedge \phi) : Price = price1 \wedge \neg(Price = price1)$$

The last defined variable in the minimal unsatisfiable core is $Price$. The constraint to be added is then $Price = price1$ and the fix to be applied to the model is presented below. The new condition for the shop to receive the payment confirmation from the CaaS is for both the sent order ID (OID), and the paid amount to correspond to already known values.

```
Caas -> Actor:  rorder(OID, ?Price);
```

is replaced by:

```
Caas -> Actor:  rorder(OID, price1);
```

which fixes the protocol.

5 Conclusions

We have presented two methods to automatically localize and repair faults in security protocols, at model level.

The first described technique relies on static analysis and uses specific rules, such as the principles of Abadi and Needham for good protocol design, in order to identify potential security flaws. The flaws found can then be repaired in a pattern-based way, thus forcing the violated principle to hold. Also, the dataflow of the protocol can be analyzed to identify possible missing checks that might have allowed the unintended execution to occur. This analysis takes place backwards, starting from a violated embedded assertion.

The second method is based on a feedback loop with distinct phases of analysis, repair and verification. It uses a found attack trace, in conjunction with analyzing certain interaction patterns, to localize the root cause of the attack and to repair the protocol. Then a new verification of the repaired protocol is initiated, and the verification – diagnosis – repair cycle is repeated until either no new attack is found, or no new potential fixes can be identified. The technique addresses several specific situations, such as the violation of non-injective agreement and the issue of missing checks of necessary constraints upon message receipt.

We have also presented the application of our techniques on several examples from security literature.

References

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*, pages 122–136, 1994.

[2] T. Aura. Strategies against replay attacks. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 59–68, 1997.

[3] D. Basin, C. Cremers, and S. Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. In *Proceedings of the 1st International Conference on Principles of Security and Trust (POST)*, volume 7215 of *LNCS*, pages 129–148, 2012.

[4] U. Carlsen. Cryptographic protocol flaws: know your enemy. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW)*, pages 192–200, 1994.

[5] K.-K. R. Choo. An integrative framework to protocol analysis and repair: Bellare-Rogaway model + planning + model checker. *Informatica*, 18(4):547–568, 2007.

[6] L. Compagna. Sat-based model-checking of security protocols. *PhD Dissertation, Università di Genova and University of Edinburgh*, 2005.

[7] C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Proceedings of the 20th Conference on Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 414–418, 2008.

[8] N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *LNCS*, pages 36–41, 2006.

[9] J. C. López Pimentel, R. Monroy, and D. Hutter. On the automated correction of faulty security protocols susceptible to a replay attack. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS)*, volume 4734 of *LNCS*, pages 594–609, 2007.

[10] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 31–43, 1997.

[11] A. Perrig and D. Song. Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 64–76, 2000.

[12] S. Son, K. S. McKinley, and V. Shmatikov. RoleCast: finding missing security checks when you do not know what checks are. In *Proceedings, 26th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1069–1084, 2011.

[13] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[14] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy (SP)*, pages 365–379, 2012.

[15] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *IEEE Symposium on Security and Privacy (SP)*, pages 465–480, 2011.