Analysing Security Protocols Using Refinement in iUML-B

Colin Snook, Thai Son Hoang, and Michael Butler

ECS, University of Southampton, U.K.
{cfs,t.s.hoang,mjb}@ecs.soton.ac.uk

Abstract. We propose a general approach based on abstraction and refinement for constructing and analysing security protocols using formal specification and verification. We use class diagrams to specify conceptual system entities and their relationships. We use state-machines to model the protocol execution involving the entities' interactions. Features of our approach include specifying security principles as invariants of some abstract model of the overall system. The specification is then refined to introduce implementable mechanisms for the protocol. A gluing invariant specifies why the protocol achieves the security principle. Security breaches arise as violations of the gluing invariant. We make use of both theorem proving and model checking techniques to analyse our formal model, in particular, to explore the source and consequence of the security attack. To demonstrate the use of our approach we explore the mechanism of a security attack in a network protocol.

Keywords: Virtual LAN, Security, Event-B, iUML-B.

1 Introduction

Ensuring security of protocols is a significant and challenging task in the context of autonomous cyber-physical systems. In this paper, we investigate the use of formal models of protocols in order to discover and analyse possible security threats. In particular, we are interested in the role of formal models in identifying security flaws, exploring the nature of attacks that exploit these flaws and proposing measures to counter flaws in systems that are already deployed.

Our contribution is a general approach based on abstraction and refinement for constructing and analysing security protocols. The approach is suitable for systems containing multiple conceptual entities (for example, data packets, devices, information tags, etc.). We use class diagrams to specify the relationships between entities and state-machines to specify protocols involved in their interactions. Security principles are defined as constraints on the system entities and their relationships. We use refinements of these models, to gradually introduce implementation details of the protocols that are supposed to achieve these security properties. The use of abstract specification and refinement allows us to separate the security properties from the protocol implementation. In particular, possible security flaws are detected as violations of the gluing invariants that link

the abstract and concrete models. Further analysis helps to pinpoint the origin and nature of attacks that could exploit these flaws. The approach has been developed within the Enable-S3 project [4] which aims to provide cost-efficient cross-domain verification and validation methods for autonomous cyber-physical systems. Within Enable-S3, we are applying the approach on case studies in the avionics and maritime domains. The case-studies involve secure authentication and communications protocols as part of larger autonomous systems.

We illustrate our approach with an analysis of *Virtual Local Area Network* (VLAN) operation including the principle of tagging packets. We explore a known security flaw of these systems, namely double tagging. We use the Event-B method and iUML-B class diagrams and state-machines as the modelling tool.

The rest of the paper is structured as follows. Section 2 gives some background on the case study, the methods and tools that we use. The main content of the paper is in Section 3 describing the development using iUML-B and analysis of the VLAN model. Finally, we summarise our approach in Section 4 and conclude in Section 5. For more information and resources, we refer the reader to our website: `http://eprints.soton.ac.uk/id/eprint/403533`. The website contains the Event-B model of the VLAN.

2 Background

2.1 VLAN tagging

A *Local Area Network* (LAN) consists of devices that communicate over physical data connections that consist of multiple steps forming routes via intermediate network routing devices called switches. The 'trunk' connections between switches are used by multiple routes. A VLAN restricts communication so that only devices that share the same VLAN as the sender, can receive the communication thus providing a way to group devices irrespective of physical topology. In order to achieve this, switches attach a tag to message packets in order to identify the sender's VLAN. The tag is removed before being sent to the receiving device. Typically, a system uses one VLAN identity to represent a default VLAN. This is known as the *native* VLAN. A packet intended for the native VLAN does not require tagging. The IEEE 802.1Q standard [6] is the most common protocol for ethernet-based LANs and includes a system for VLAN tagging and associated handling procedures. The standard permits multiple VLAN tags to be inserted so that the network infrastructure can use VLANs internally as well as supporting client VLAN tagging. A well-known security attack exploits double tagging by hiding a tag for a supposedly inaccessible VLAN behind a tag for the the native VLAN. The receiving switch sees the unnecessary native VLAN tag and removes it before sending the packet on to the next switch. This switch then sees the tag for the inaccessible VLAN and routes the packet accordingly so that the packet infiltrates the targeted VLAN. Double tagging attacks can be avoided by not using (i.e. de-configuring) the native VLAN.

2.2 Event-B

Event-B [1] is a formal method for system development. Main features of Event-B include the use of *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* constraining the carrier sets and constants. Machines contain *variables* v, *invariants* $I(v)$ constraining the variables, and *events*. An event comprises a guard denoting its enabled-condition and an action describing how the variables are modified when the event is executed. In general, an event `e` has the following form, where t are the event parameters, $G(t, v)$ is the guard of the event, and $v := E(t, v)$ is the action of the event[1].

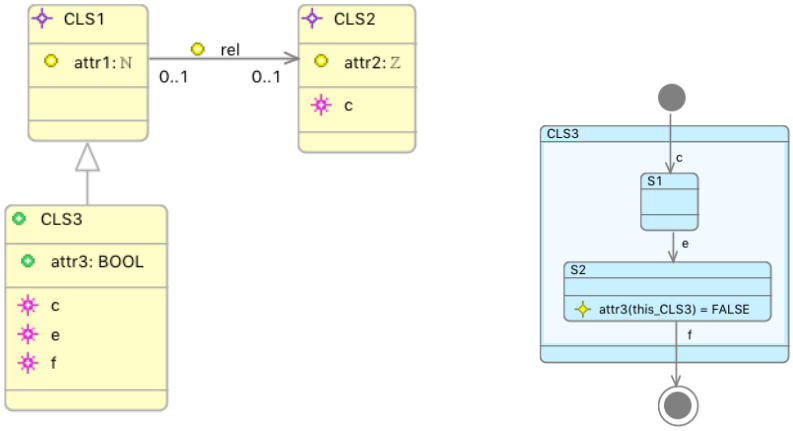
$$\texttt{e} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ G(t, v)\ \textbf{then}\ v := E(t, v)\ \textbf{end} \tag{1}$$

A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions. Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. Machine **M** can be *refined* by machine **N** (we call **M** the abstract machine and **N** the concrete machine). The state of **M** and **N** are related by a gluing invariant $J(v, w)$ where v, w are variables of **M** and **N**, respectively. Intuitively, any "behaviour" exhibited by **N** can be simulated by **M**, with respect to the gluing invariant J. Refinement in Event-B is reasoned event-wise. Consider an abstract event `e` and the corresponding concrete event `f`. Somewhat simplifying, we say that `e` is refined by `f` if `f`'s guard is stronger than that of `e` and `f`'s action can be simulated by `e`'s action, taking into account the gluing invariant J. More information about Event-B can be found in [5]. Event-B is supported by the *Rodin Platform* (Rodin) [2], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

2.3 iUML-B

iUML-B [8–10] provides a diagrammatic modelling notation for Event-B in the form of state-machines and class diagrams. The diagrammatic models are contained within an Event-B machine and generate or contribute to parts of it. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states while Event-B events are expected to already exist to represent the transitions. Transitions contribute further guards and actions representing their state change, to the events that they elaborate. An existing Event-B set may be associated with the state-machine to define its instances. In this case the state-machine is 'lifted' so that it has a value for every instance of the associated set. State-machines are typically refined by adding nested state-machines to states.

[1] Actions in Event-B are, in the most general cases, non-deterministic [5].



(a) Class diagram (b) State-machine

Fig. 1: Example iUML-B diagrams

Class diagrams provide a way to visually model data relationships. Classes, attributes and associations are linked to Event-B data elements (carrier set, constant, or variable) and generate constraints on those elements. For the VLAN we use class diagrams extensively to model the sets of entities and their relationships and we use state-machines to constrain the sequences of events and to declare state dependant invariant properties.

Figure 1 shows an abstract example of an iUML-B model to illustrate the features we have used in the VLAN. We give the corresponding translation into Event-B in Figure 2. In Figure 1a, there are three classes; *CLS1*, *CLS2*, which elaborate carrier sets, and *CLS3*, which is a sub-class of *CLS1* and elaborates a variable. An *attribute* or *association* of a class can have a combination of the following properties: *surjective*, *injective*, *total*, and *functional*. Attributes *attr1* of *CLS1* and *attr3* of *CLS3* are total and functional, while *attr2* of *CLS2* is functional. An injective association *rel* defined between *CLS1* and *CLS2* elaborates a constant. Figure 1b shows an example of a state-machine, which is lifted to the carrier set *CLS1* for its instances. This is also the instances set for the class *CLS1* and a state of the state-machine is named after its variable sub-class, *CLS3*. Further sub-states *S1* and *S2* are modelled as variable subsets of *CLS3*. The state of an instance is represented by its membership of these sets. The state-machine transitions are linked to the same events as the methods of *CLS3*. Hence the state-machine constrains the invocation of class methods for a particular instance of the class. The contextual instance is modelled as a parameter *this_CLS3* which can be used in additional guards and actions in both the class diagram and the state-machine.

The transition `c`, from the initial state to *S1* also enters parent state *CLS3* and therefore represents a constructor for the class *CLS3*. The class method `c` is

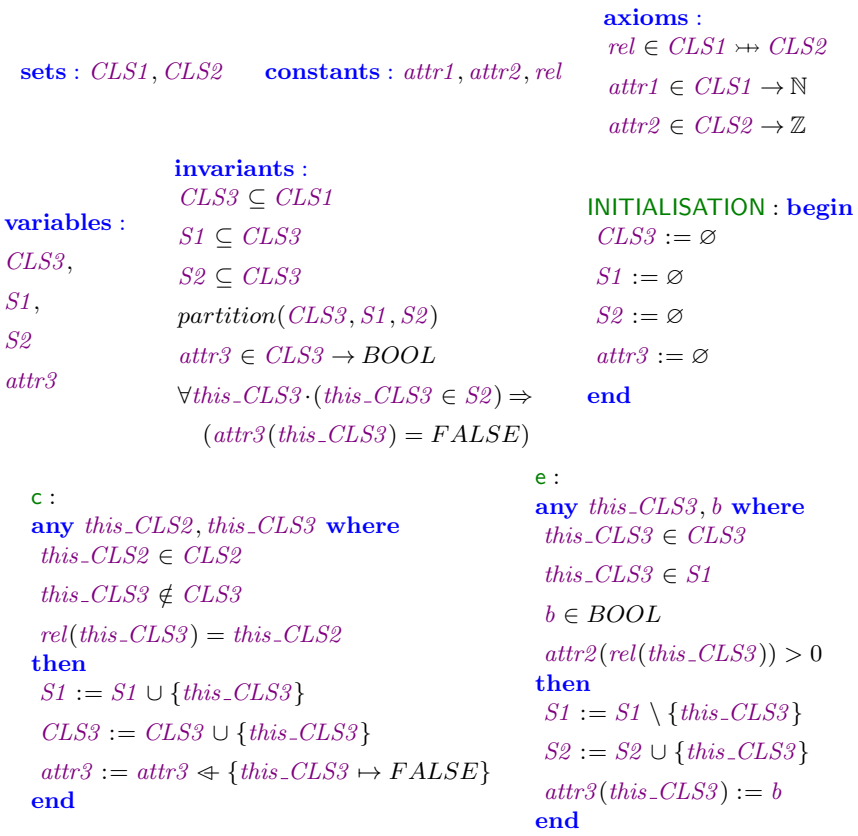


Fig. 2: Event-B translation of the iUML-B example

also defined as a constructor and automatically generates an action to initialise the instance of *attr3* with its defined initial value. The same event c is also given as a method of class *CLS2* in order to generate a contextual instance *this_CLS2* which is used in an additional (manually entered) guard to define a value for the association *rel* of the super-class. The transition and method e is a normal method of class *CLS3*, which is available when the contextual instance exists in *CLS3* and *S1*, and changes state by moving the instance from *S1* to *S2*. The other guards and actions shown in this event concerning parameter *b* and attribute *attr2*, have been added as additional guards and actions of the transition or method. These are not shown in the diagram as they are entered using the diagram's properties view. The state invariant shown in state *S2* applies to any instance while it is in that state. The Event-B version of the invariant is quantified over all instances and an antecedent added to represent the membership of *S2*. In the rest of this paper we do not explain the translation to Event-B.

2.4 Validation and Verification

Consistency of Event-B models is provided via means of proof obligations, e.g., invariant preservation by all events. Proof obligations can be discharged automatically or manually using the theorem provers of Rodin. Another important tool for validation and verification of our model is ProB [7]. ProB provides model checking facility to complement the theorem proving technique for verifying Event-B models. Features of the ProB model checker include finding invariant violations and deadlock for multiple refinement levels simultaneously. Furthermore, ProB also offers an animator enabling users to validate the behaviour of the models by exploring execution traces. The traces can be constructed interactively by manual selection of events or automatically as counter-examples from the model checker. Here, an animation trace is a sequence of event execution with parameters' value. The animator shows the state of the model after each event execution in the trace.

3 Development

In this section, we discuss the development of the model. The model consists of three refinement levels. The abstract level captures the essence of the security property which is proven for the abstract representation of events that make new packets and move them around the network. The first refinement introduces some further detail of the network system and is proven to be a valid refinement of the first model. That is, it maintains the security property. Both of these first levels are un-implementable because they refer directly to a conceptual property of a packet which is the VLAN that the packet was intended for. In reality it is not possible to tell from a raw packet, which VLAN it was originally created for. The second refinement introduces tagging as a means to implement a record of this conceptual property. The refinement models nested tagging and the behaviour of a typical switch which, apart from tagging packets depending on their source, also removes tags for the native LAN. The automatic provers are unable to prove that removing tags satisfies the gluing invariant. This is the well-known security vulnerability to double tagging attacks. Adding a constraint to, effectively, disallow the native LAN from being configured as a VLAN, allows the provers to discharge this proof obligation. This corresponds to the usual protective measure against double tagging attacks.

3.1 M0 : An Abstract Model of VLAN Security

We aim to make the first model minimally simple while describing the essential security property. We use a class diagram (Figure 3) to introduce some 'given' sets for data packets (class PKT) and VLANs (class $VLAN$). The constant association PV describes the $VLAN$ that each packet is intended for. (Note that this is a conceptual relationship representing an intention and hence the implementation cannot access it). We abstract away from switches and devices and

introduce a set of nodes, class *NODE*, to represent both. The communications topology is given by the constant association, *Route*, which maps nodes to nodes in a many to many relationship.

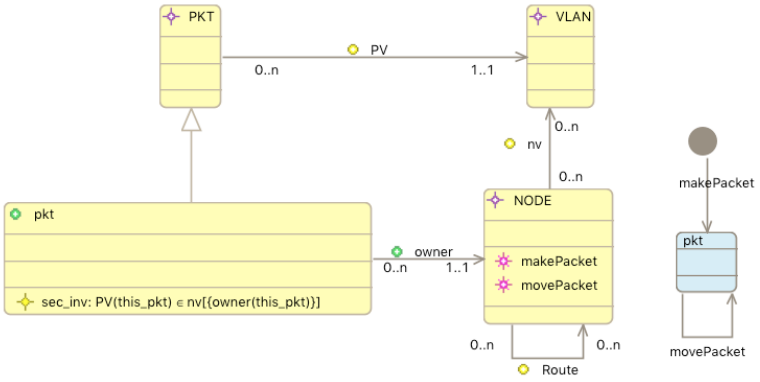


Fig. 3: Abstract model of VLAN security requirement

The set of VLANs that a particular node is allowed to see, is given by the constant association *nv*. For now this is a many to many relationship but in later refinements we will find that, while switches are allowed to see all VLANs, devices may only access the packets of one VLAN.

The class *pkt* represents the subset of packets that currently exist (whereas, *PKT* represented all possible packets that might exist currently or in the past or future). A packet that exists, always has exactly one owner node. The method makePacket takes a non-existing packet from *PKT* and adds it to *pkt* and initialises the new packet's *owner* to the contextual node instance. The method movePacket changes the *owner* of an existing packet to a new node that is non-deterministically selected from the nodes that the current owner node is directly linked to via *Route*.

The class invariant, sec_inv, in class *pkt* describes the security property[2]:

$$\forall this\_pkt \cdot this\_pkt \in pkt \Rightarrow PV(this\_pkt) \in nv[\{owner(this\_pkt)\}] \ , \qquad (2)$$

i.e., the VLAN for which this packet is intended, belongs to the VLANs that its owner is allowed to see. For this invariant to hold we need to restrict the method movePacket so that it only moves packets to a new owner that is allowed to see the VLAN of the packet. For now we do this with a guard, $PV(p) \in nv[\{n\}]$, where p is the packet and n is the destination node. However, this guard must be replaced in later refinements because it refers directly to the conceptual property

[2] A concise summary of the Event-B mathematical notation can be found at `http://wiki.event-b.org/images/EventB-Summary.pdf`.

PV and is therefore not implementable. We also ensure that makePacket only creates packets with a *PV* value that its maker node is allowed to see.

We use a state-machine (Figure 3) to constrain the sequence of events that can be performed on a packet. The state-machine is lifted to the set *PKT* of all packets, At this stage we only require that makePacket is the initial event that brings a packet into existence, and this can be followed by any number of movePacket events.

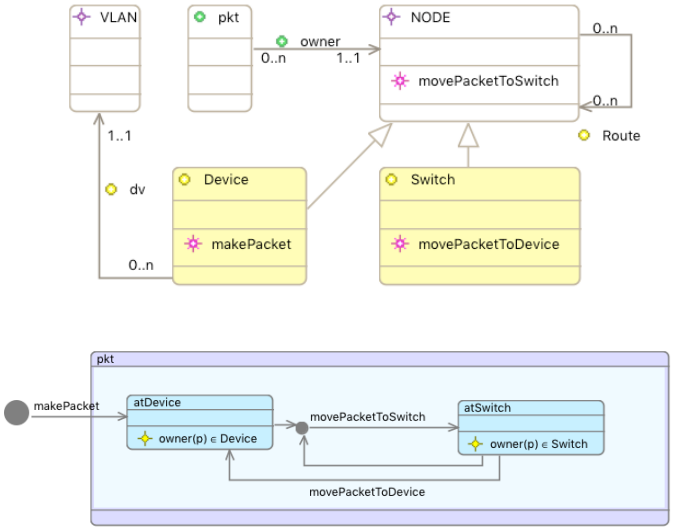


Fig. 4: First refinement of VLAN introducing Switches and Devices

3.2 M1: Introducing Switches and Devices

In M0, to keep things simple we did not distinguish between switches and devices. However, they have an important distinction since switches are allowed to see all VLAN packets. The design will utilise this distinction so we need to introduce it early on. In **M1** (Figure 4) we introduce two new classes, *Switch* and *Device*, as subtypes of *NODE*.

Since switches are implicitly associated with all VLANS (i.e. trusted), we do not need to model which VLANs they are allowed to access. Therefore, we replace *nv* with a functional association *dv* whose domain (source) is restricted to *Device*. It is a total function, rather than a relation, because a device has access to exactly one VLAN and again we model this as a constant function since we do not require it to vary.

Switches are not allowed to create new packets so we move makePacket to *Device*. Since, when moving a packet, the destination kind affects the security

checks, we split movePacket into two alternatives: movePacketToSwitch which does not need any guard concerning PV and movePacketToDevice where we replace the guard, $PV(p) \in nv[\{n\}]$, with $PV(p) = dv(n)$ to reflect the data refinement. Note however, that the new guard still refers to PV.

The refinement introduces the need for some further constraints on the sequence of events for a particular package. We introduce sub-states *atDevice* and *atSwitch* (Figure 4) to show that a packet can only be moved to a device from a switch. Note that these states could be derived from *owner* (hence the invariants in states *atDevice* and *atSwitch*) however, the state diagram helps visualise the process relative to a packet which will become more significant in the next refinement level.

3.3 M2: Introducing Tagging

We can now introduce the tagging mechanism that allows switches to know which VLAN a packet is intended for. Our aim is that, in this refined model, switches should not use the PV relationship other than for proving that the tag mechanism achieves an equivalent result. We introduce a new given set, TAG, (Figure 5) which has a total functional association TV with $VLAN$. This function represents the VLAN identifier within a tag, which is part of the implementation, i.e., guards that reference TV are implementable. We add a variable partial function association, *tag*, from *pkt* to TAG, which represents the tagging of a packet.

In typical LAN protocols, already tagged packets can be tagged again to allow switches to use VLANs for internal system purposes. Although, for simplification, we omit this internal tagging, we allow tags to be nested so that we can model a double tagging attack by a device. Therefore we model nested tags with a variable partial functional association, *nestedTag* from TAG to itself. When a packet arrives at a switch from a device, the switch can tell which VLAN it belongs to from the port that it arrived on. However, for simplicity, we avoid introducing ports in this refinement. Instead we model this information via a variable functional association *from* from *pkt* to $NODE$. Hence a switch can determine which VLAN a packet, p, is for via $dv(from(p))$. Port configuration could easily be introduced in a subsequent refinement without altering the main points of this article. A significant behaviour of switches that relates to security is how they deal with packets for the native VLAN. Therefore, in the Event-B context for **M2**, we introduce a specific instance of $VLAN$ called $NativeVLAN$.

The behaviour (Figure 5) is refined to add procedures for handling tagged packets. State *atSwitch* is split into three sub-states, $RCVD$ for packets that have just been received from a device, $TAGD$ for packets from a device that have been successfully processed and $REJECTED$ for packets that are found to be invalid. A device may now send an untagged packet to a switch (transition movePacketToSwitch_untagged) and allow the switch to determine appropriate tagging, or it may tag the packet itself (transition movePacketToSwitch_tagged) in which case the switch will check the tag. In the latter case the tag may be valid or invalid and may have nested tags.

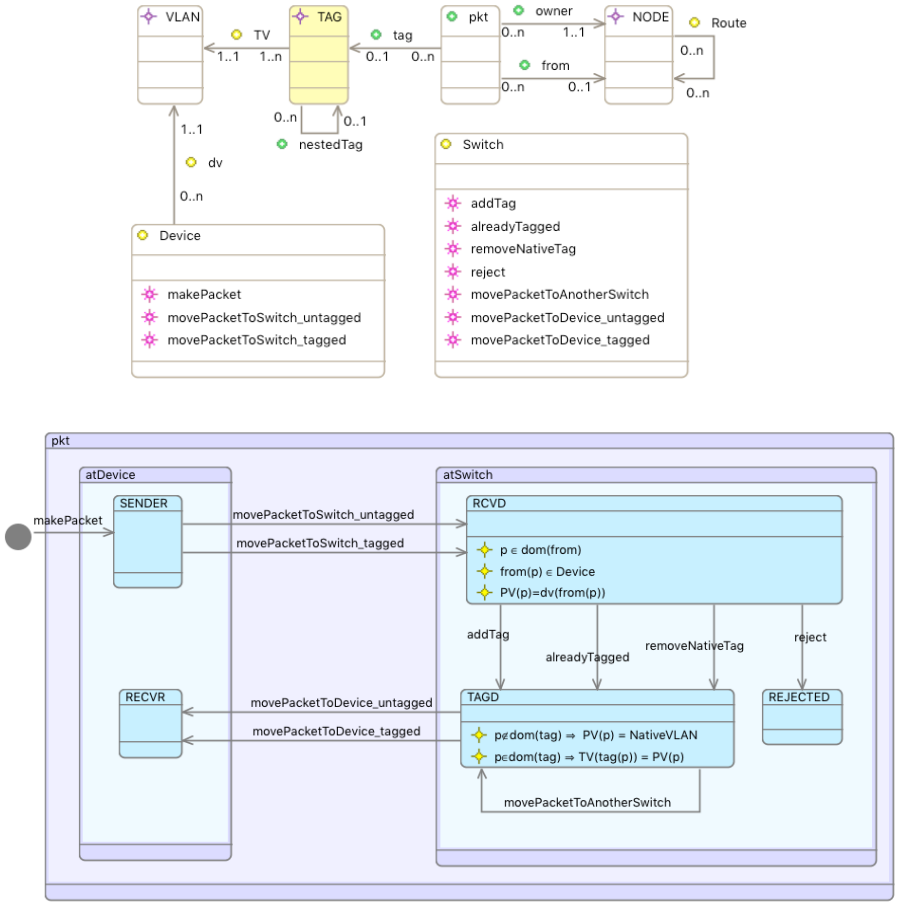


Fig. 5: Second refinement of VLAN introducing tagging

After receiving a packet, p, at the state $RCVD$, the new owner switch processes it by taking one of the following transitions:

- addTag: if p is not already tagged, a tag, tg, such that $TV(tg) = dv(from(p))$, is added and the packet is accepted by moving it to state $TAGD$.
- alreadyTagged: if p is already tagged correctly (i.e., $TV(tg) = dv(from(p))$) and not tagged as the native VLAN (i.e. $TV(tg) \neq NativeVLAN$) the packet is accepted as is.
- removeNativeTag: if p is correctly tagged for the native VLAN (i.e., $TV(tg) = dv(from(p)) = NativeVLAN$), the tag is removed and the packet is accepted. The tag is removed in such as way as to leave p tagged with a nested tag if any.
- reject: if p is incorrectly tagged (i.e., $TV(tg) \neq dv(from(p))$), it is rejected by moving it to state $REJECTED$ which has no outgoing transitions.

After processing packet, p, the switch can either pass it on to another switch or, if available, pass it to a device via one of the following transitions:

- movePacketToDevice_untagged: if p is not tagged, and the switch is connected to a device, n, on the native VLAN (i.e. $dv(n) = NativeVLAN$),
- movePacketToDevice_tagged: if p is tagged and the switch is connected to a device, n, which is on the VLAN indicated by the tag (i.e. $dv(n) = TV(tag(p))$).

It is these two transitions that refine movePacketToDevice, which need to establish the security invariant using tags rather than the unimplementable guard concerning PV. This has been done as indicated above by the conditions on $dv(n)$. It can been seen by simple substitution, that the state invariants of $TAGD$ enable the prover to establish that the new guards are at least as strong as the abstract one ($PV(p) = dv(n)$). We also need to prove that these state invariants are satisfied by the incoming transitions of state $TAGD$. A state invariant $PV(p) = dv(from(p))$ is added to $RCVD$ in order to allow the prover to establish this. Again, this can be checked using simple substitutions of the guards of addTag, alreadyTagged and removeNativeTag using this state invariant. The other two state invariants for $RCVD$ are merely to establish well-definedness of the function applications. The state invariants of state $RCVD$ are clearly established by the actions of incoming transitions movePacketToSwitch_untagged and movePacketToSwitch_tagged.

3.4 Analysis

We analyse the protocol using both theorem proving and model checking techniques. Given the model in Section 3.3, the automatic provers discharge all proof obligations except for one. The prover cannot establish that the transition removeNativeTag establishes the state-invariant

$$p \in dom(tag) \Rightarrow TV(tag(p)) = PV(p) \tag{3}$$

of state $TAGD$. In general, a failed invariant preservation proof identifies the property (the invariant) that may be at risk and the transition (event) that may violate it. We say 'may' because lack of proof does not necessarily indicate a problem. It can be a result of insufficient prover power. We therefore use the ProB model checker to confirm the problem.

As with any model checker, we instantiate the context of the system, in this case, the network topology. The network topology under consideration can be seen in Figure 6. The switches, i.e., $SWCH1$ and $SWCH2$ have access to all VLANs, namely, $VLAN1$, and $VLAN2$. The native VLAN $NativeVLAN$ is defined to be $VLAN1$. Devices $DVCE1$, $DVCE4$ belong to $VLAN1$ and devices $DVCE2$ and $DVCE3$ both belong to $VLAN2$. We define two packets $PK1$ and $PK2$ where $PK1$ is intended for $VLAN1$ and $PK2$ is for $VLAN2$, i.e.,

$$PV = \{PK1 \mapsto VLAN1, PK2 \mapsto VLAN2\} .$$

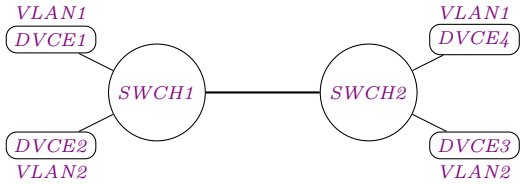


Fig. 6: Network topology for analysis

Finally, we define two tags *TAG1*, and *TAG2* corresponding to *VLAN1* and *VLAN2*, respectively. A tag with nested tag is numbered accordingly, for example, *TAG12* is for *VLAN1* and has an inner tag for *VLAN2*. Our subsequent analysis is based on this particular setting.

Firstly, we want to identify whether the state-invariant (3) can indeed be violated. We model check the whole refinement-chain from **M0** to **M2**. ProB indeed identifies a counter-example trace which leads to the violation of the invariant as follow.

$$
\begin{aligned}
&\ldots \\
\longrightarrow\ & \mathsf{makePacket}(PK1, DVCE1) && (4)\\
\longrightarrow\ & \mathsf{movePacketToSwitch\_tagged}(PK1, SWCH1, TAG12, DVCE1) && (5)\\
\ldots\ & \ldots \\
\longrightarrow\ & \mathsf{removeNativeTag}(PK1, SWCH1) && (6)
\end{aligned}
$$

In the trace, *DVCE1* creates *PK1* (4) before moving it to *SWCH1* with tag *TAG12* (5). When *SWCH1* removes the native tag *TAG12* from *PK1* (6), resulting in *TAG2*, the state-invariant (3) becomes invalid since *PK1* is intended for *VLAN1*, but it is now tagged with *TAG2*, which is identified for *VLAN2*.

However, the violation could be caused by an unnecessarily strong gluing invariant. To verify whether the security invariant (2) is indeed violated in **M2**, we model check **M2** without **M0** and **M1** but with the security invariant copied from **M0** to **M2** in place of the gluing invariant. Once again, ProB returns a counter-example trace which is an extension of the previous trace, i.e.,

$$
\begin{aligned}
&\ldots \\
\longrightarrow\ & \mathsf{makePacket}(PK1, DVCE1) && (7)\\
\longrightarrow\ & \mathsf{movePacketToSwitch\_tagged}(PK1, SWCH1, TAG12, DVCE1) && (8)\\
\ldots\ & \ldots \\
\longrightarrow\ & \mathsf{removeNativeTag}(PK1, SWCH1) && (9)\\
\ldots\ & \ldots \\
\longrightarrow\ & \mathsf{moveUntaggedPacketToDevice\_tagged}(DVCE2, PK1, SWCH1) && (10)
\end{aligned}
$$

After removing the native tag of *PK1* (9), the packet is moved from *SWCH1* to *DVCE2* (10). At this time, *PK1* has arrived to a device (*DVCE2*) which does not have permission to receive any packet for *VLAN1*.

Note that there are three different points in the process leading to the security breach:

- the point where the security attack is initiated (8),
- the point where the design assumptions are violated and (9),
- the point where the security is breached (10).

Coming back to the original failed invariant preservation proof obligation, we can now confirm that it is indeed possible for the invariant to be violated[3]. Examination of the pending goal that the prover is attempting to prove reveals more detail about the problem.

$$TV((\{p\} \times nestedTag[tag[\{p\}]])(p)) = PV(p) \ ,$$

It shows that the prover has replaced the packet's tag with its nested tag in the design property, and is attempting to show that the VLAN of the nested tag is also for the correct VLAN for the packet. From the theorem prover, therefore, we know that

- the switch's procedure of removing the native tag causes a problem,
- the problem is that the nested tag becomes the packets main tag and does not necessarily indicate the correct VLAN

When a constraint, $NativeVLAN \notin ran(dv)$, i.e., no device can be configured to use the native VLAN, is added to the model the proof obligation is immediately discharged since the guard of the transition removeNativeTag can easily be shown to be false. This constraint corresponds to the recommended protective action to prevent double tagging attacks.

Overall, the theorem provers can identify the security flaw in a design or protocol. They do not need to find an example attack but can pinpoint the exact nature of the flaw directly. This is because proof obligations are generated from the actions of individual events. While the provers indicates the nature of the violation of the design assumption, they do not reveal the complete sequence from attack to security breach. The model-checker, while being restricted to example instantiations, is able to illustrate the process from initial attack through to security breach.

4 Summary of Approach

To summarise, our approach is as follows:

1. Create an iUML-B Class diagram model of the entities and relationships that are essential concepts of the system. Add a state-machine to model the required behaviour of the system. Only model sufficient concepts to express the security property. Do not model the mechanism that implements the security.

[3] This is because removing the native tag may reveal an invalid nested tag (the known security flaw exploited by double tagging attacks).

2. Express the security property as an invariant over the entities in the model. Make sure that the model preserves the invariant.
3. Refine the iUML-B model (possibly over several iterations) to introduce the mechanism that will ensure the system is secure. Do not constrain the behaviour of elements unless the security system has control over this behaviour. That is, allow attacks to occur within the model.
4. Animate each refinement level to ensure that the model behaves in a useful way. This is important to *validate* that our formal model captures the behaviour of the real system.
5. If any POs are not proven check the type of PO and the goal to see whether there is a mistake in the model. Correct the model as necessary.
6. If unproven POs remain for the gluing invariant, this may mean that the security mechanism has a flaw. Analyse the problem as follows:
   - Examine the PO. Note the event that it relates to and examine the goal of the prover. This can often be used to interpret what is going wrong or whether a manual proof is possible.
   - Run the model-checker to establish that there really is a problem. If the model checker can not find a trace to the violation, a manual proof may be possible.
   - Remove the gluing invariant and copy the security property invariant from the abstract model and run the model checker (without previous refinement levels). If it does not find a trace that violates the security property, the gluing invariant may be too strong.
   - If a trace to the security property is found there is a flaw in the protocol. The trace can be examined to analyse the nature of the attack, the flaw in the security mechanism and how it leads to the security violation.

In the example presented in this paper, the abstract model (step 1) **M0** was developed in Section 3.1, and the security invariant (step 2) was introduced in the same section. The refinement process (step 3) involved an intermediate refinement **M1** in Section 3.2 and a final refinement **M2** in Section 3.3. At each refinement level, animation with ProB (step 4) and examination of unproven POs (step 5), helped us to arrive at a correct and useful model. A security flaw was detected and analysed (step 6) as described in Section 3.4.

5 Conclusion

Our investigation into a known example of a security vulnerability indicates that formal modelling with strong verification tools can be extremely beneficial in understanding security problems. The tools at our disposal include an automatic theorem prover as well as a model checker. In our previous work on safety-critical systems we have found that these tools exhibit great synergy and this is also the case when analysing security protocols.

We use iUML-B class diagrams and state-machines as a diagrammatic representation of the Event-B formalism. The diagrams help us create, visualise and communicate the models leading to a better understanding of the systems.

Although we use animation to informally validate system behaviour, we have not yet done any rigorous analysis of liveness properties. A future aim of our research is to incorporate liveness reasoning into our approach.

This refinement-based approach can be applied to any problem that involves sets of entities that are interacting in some way via a procedure or protocol. For example, an authentication protocol such as Needham-Schroder could be modelled abstractly as a class of *agents* sending *messages* and receiving them with property *perceived sender* based on an *actualSender*. This could then be refined to replace direct references to the *actual sender*, with encrypted *nonces*.

Finally, we envisage that without refinement, formulating the gluing invariant that links the specification to the implementation would, in general, be challenging. Here the role of the gluing invariant is essential as its violation helps the designer to identify the point where the design assumptions are offended, causing the actual security breach. A similar observation has been made in [3].

Acknowledgement

This work is funded by the Enable-S3 Project, www.enable-s3.eu.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.
3. Michael Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects of Computing*, 14(1):2–34, October 2002.
4. Enable-S3 consortium. *Enable-S3 project website*. `http://www.enable-s3.eu`. Accessed 04/12/2016.
5. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
6. IEEE. *802.1Q-2014 - Bridges and Bridged Networks*. `http://www.ieee802.org/1/pages/802.1Q-2014.html`. Accessed 02/12/2016.
7. Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.
8. Mar Yah Said, Michael Butler, and Colin Snook. A method of refinement in UML-B. *Softw. Syst. Model.*, 14(4):1557–1580, October 2015.
9. Colin Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, pages 29–30, Toulouse, France, 2014. `http://eprints.soton.ac.uk/365301/`.
10. Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.