Systematic Approach to Security Risk Investigation

Dušan P. Jokanovic
Furukawa Electric Co., Ltd. Sendai, Japan

Abstract: We propose a formal method for systematic investigation of the system security breaches when they occur. Our method is based on system specifications arranged in directed acyclic graph-like structures named constraint diagrams. Each vertex of a constraint diagram contains a different set of constraints imposed on a system and/or assumed on its environment. Enforcing the constraints should ideally prevent the occurrence of corresponding security breaches, where the more severe constraints, the higher the cost of observing them. Given a security breach, a designer can traverse constraint diagram to access accurately how much damage to the system was incurred and its cost of repair. Moreover, the method helps identifying unforeseen security holes in system implementations, which may benefit the future systems by designing ways to avoid them.

We make use of a property-oriented specification language, as a formalization vehicle, which combines both axiomatic and algebraic specifications into a two-tiered specification. This suits well to our purpose to describe both system's state-dependent and independent security properties. The risk investigation technique is explained on an example of a security breach such as acquiring unauthorized capabilities in the e-mail system.

1. INTRODUCTION

A designer of modern information systems faces a challenging task to reduce security risk and enforce information privacy. However, no matter how much foresight one has put into systems most systems do allow security breaches; people find ways to compromise software, tamper with hardware, or deceive personnel.

In order to facilitate designer's and administrator's job, we propose a formal method for systematic investigation of the system security breaches when they occur. In reality, it is impossible to formally model many environmental aspects including unpredictable or unanticipated events that may affect system security characteristics. Hazard analysis [1], as a complementary technique to formal methods, can identify a system's safety and security-critical components. Formal methods can then be used to describe and reason about these components, where reasoning holds only for those system input parameters that are made explicit. To the same end we have developed a formal method that enables risk estimation for certain kind of security breaches. Particularly, our approach allows for specifying some security properties (e.g., secrecy and integrity) and assessing the expense incurred to ensure

them. It incorporates constraints into specifications where, the weaker the constraints, the greater the risk of hazardous events in the system. Furthermore, the constraints are grouped into sets and arranged into diagrams. Our specification-based model has the following advantages that make it distinct compared to already proposed techniques for modelling security [2], [3] :

- 1 - It is abstract and diagrammatic in that the user is not swamped by superfluous implementation details;
- 2 - It is concerned only with functional behavior, yet it provides natural interface to the probabilistic models;
- 3 - It shows explicitly how changes in the environment correspond to changes in observable behavior.

The third characteristics listed above is often neglected in systems' security specifications. A system does not run in isolation; its behavior is affected by input from the external world, which in turn consumes the system's output. A system's behavior as captured in its specification is conditional on the environment's behavior. Whereas a designer places constraints on the system's behavior, he or she cannot place constraints on the environment but can only make assumptions about its behavior. When specifying a large, complex system, the designer should take special care to make explicit as many assumptions about the environment as possible. Our method helps in stating explicitly the circumstances under which the system is expected to behave properly and have required level of security.

The balance of the paper is as follows. In Section 2, we introduce notation and describe our model formally. In Section 3, the specification language [4] used as a formalization vehicle is discussed. In Section 4, our risk investigation technique is explained on an example of a security breach such as acquiring unauthorized capabilities, namely others' passwords [5], [6] in the e-mail system. Finally, Section 5 concludes with the brief overview of the technique.

2. MODEL

We assume that a system consists of a collection of sequential threads of control called *processes* that access data structures called *objects*. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate objects of that type.

We model a system computation as a *history*, which has a finite sequence of interleaved executions of operations on objects by processes. Each process can execute operations on more than one object and each object can be accessed by more than one process. To denote an operation of object x executed by process P , we write:

$$x :: op(args^*)/term(res^*) :: P$$

where op is an operation name, $args^*$ is a sequence of argument values, $term$ is a termination condition name, and res^* is a sequence of result values. The operation name and argument values constitute the invocation, and the termination condition and result values constitute the response. We use "Ok" for normal termination and write " $inv(op)$ " for the invocation of op .

It is assumed that operations on objects are executed atomically; that is, an operation either takes place completely or not at all, and operations appear to take place instantaneously with respect to one another.

2.1 Simple Object Automata

We model an object by a *simple object automaton*, an automaton that accepts certain histories. A simple object automaton is a four-tuple $\langle \text{STATE}, s_0, \text{OP}, \delta \rangle$, where STATE is the object's set of states, $s_0 \in \text{STATE}$ is its initial state, OP is a set of operations (the automaton's input alphabet), and $\delta: \text{STATE} \times \text{OP} \rightarrow 2^{\text{STATE}}$ is a partial *transition function*. We use 2^X to denote powerset of the set X.

The domain of the transition function can be extended to histories, $\delta^*: \text{STATE} \times \text{OP}^* \rightarrow 2^{\text{STATE}}$:

$$\begin{aligned}\delta^*(s, \Lambda) &= s \\ \delta^*(s, H \cdot p) &= \bigcup_{s' \in \delta^*(s, H)} \delta(s', p)\end{aligned}$$

where: "." denotes concatenation, and Λ denotes the empty history. We use $\delta^*(H)$ as shorthand for $\delta^*(s_0, H)$. A history H is accepted by an automaton if $\delta^*(H) \neq \emptyset$. We call $L(A)$, the language accepted by automaton A , the *behavior* of A .

2.2 Constraint Diagrams

Let A be a set of simple object automata having the same set of states, the same initial state, and the same operations, but possible different transition functions. We say that A is a *diagram of automata* if the set $\{L(A) \mid A \in A\}$ is a diagram under reverse inclusion (i.e., the smallest language is at the top). We call the language of the automaton at the top of the diagram the preferred behavior of the diagram.

A *constraint diagram* is given by a set of constraints C , a diagram of automata A , and a diagram homomorphism as follows:

$$\phi: 2^C \rightarrow A$$

It suffices to think of each constraint as an assertion to be satisfied by the environment. We orient the constraint diagram 2^C , i.e., the domain of ϕ , so that the largest (intuitively the strongest) set of constraints lies at the top, and $\phi(C)$ is the preferred behavior of A .

A constraint diagram is thus a directed acyclic graph of simple object automata parameterized by a set of constraints, where the stronger the set of constraints, the smaller the language accepted. Informally, a constraint diagram describes an object's conditional behavior. If the environment is such that the object satisfies constraints $C \subseteq C$, then the object will behave like the simple object $\phi(C)$ accepting the language $L(\phi(C))$. While an object is able to satisfy its strongest set of constraints, it will only accept histories from its preferred behavior. If changes to the environment, e.g., security violations, force the object to satisfy a weaker set, then it will accept additional "weakly consistent" histories, which are undesired but may be tolerated. Similarly, if changes to the environment, e.g., repairs of compensating actions, later occur, then the object will resume a more desired behavior.

The method is appropriate for modelling the behavior of objects for which there is a meaningful cost associated with moving up the constraint diagram. The higher one goes in the diagram the higher the price paid for the more preferred behavior. In Section 4 we use the constraint diagrams to model the cost of tolerating security breaches.

3. FORMAL SPECIFICATIONS

In this Section we briefly overview the most relevant characteristics of the Larch specification language [4] used to specify the states and transition functions for automata. We show the advantage of using a property-oriented specification language that combines both axiomatic and algebraic specifications into a two-tiered specification. An automaton's possible states are represented as a set of values, as specified by a Larch trait. In a trait, the set of operators and their signatures, shown following the keyword **introduces**, defines a vocabulary of terms to denote values. We give complete formal specifications for the simple data structure called Bag, as an example (Figure 1). From the Bag trait, *emp* and *ins(emp,5)* denote two different bag (multiset) values. The set of equational axioms following the **asserts** clause defines a meaning for the terms, more precisely, an equivalence relation on the terms, and hence on the values they denote. The **generated by** clause of Bag asserts that *emp* and *ins* are sufficient operators to generate all values of bags. Formally, it introduces an inductive rule of inference that allows one to prove properties of all terms of sort *B*.

Larch provides two ways of reusing traits: a trait *T* can **include** or **assume** another trait *T1*. If *T1* is included, then *T* extends the theory denoted by *T1* by adding more operators and equations explicitly in *T*. With either kind of reuse, a **with** clause allows renaming of operator and sort identifiers. We use Larch interfaces to describe transition functions for simple object automata. The object identifier, e.g., *q* is an implicit argument and return formal (parameter) of each operation; the process's identifier, e.g., *P*, is an implicit argument, useful for applications for which the identity of an operation's caller is needed. A **requires** clause states the precondition that must hold when an operation is invoked. An omitted **requires** clause is interpreted as equivalent to "**requires true**." An **ensures** clause states the postcondition that the operation must establish upon termination. An unprimed argument formal, e.g., *q*, in a predicate stands for the value of the object when the operation begins. A return formal or a primed argument formal, e.g., *q*, in a predicate stands for the value of the object at the end of the operation. For an object *x*, the absence of the assertion *x' = x* in the postcondition states that the object's value may change. We use the vocabulary of traits to write the assertions in the pre- and postconditions of an object's operations; we use the meaning of equality to reason about its values.

For an operation *e* of a simple object automation *A* we write *e.preA* and *e.postA* for the pre- and postconditions of *e*. The transition function δ for *A* is defined such that:

$$(\forall s, s' \in \text{STATE}) s' \in \delta(s, e) \text{ iff } e.pre_A(s) \wedge e.post_A(s, s').$$

For each automaton in a diagram *A*, the sets of states (values) are the same, but their transition functions differ. Thus, their specifications will all use the same trait, but will have different interfaces. We will be reusing the Bag specification in the next section.

```

Bag: trait
  introduces
    emp:  $\rightarrow B$ 
    ins:  $B, E \rightarrow B$ 
    del:  $B, E \rightarrow B$ 
    isEmp:  $B \rightarrow \text{Bool}$ 
    isIn:  $B, E \rightarrow \text{Bool}$ 
  asserts
    B generated by [emp, ins]
    for all [b: B, e, e1: E]
      del(emp, e) = emp
      del(ins(b,e), e1) = if e = e1 then b else ins(del(b,e1), e)
      isEmp(emp) = true
      isEmp(ins(b,e)) = false
      isIn(emp, e) = false
      isIn(ins(b,e), e1) = (e = e1)  $\vee$  isIn(b, e1)

b:: Enq(e)/Ok() :: P
  ensures b' = ins(b,e)

b:: Deq()/Ok(e) :: P
  requires  $\neg$  isEmp(b)
  ensures isIn(b, e)  $\wedge$  b' = del(b,e)

```

Fig. 1 Bag trait and interfaces

4. SECURITY OBJECTS

Constraint diagrams can be used to specify certain kinds of security properties and to characterize the expense incurred to ensure security. The cost of preserving security is the cost in maintaining password, implementing a secure encryption algorithm, or hiring personnel to guard a locked room. Here, we consider the two security properties, *secrecy* and *integrity*. To preserve secrecy, we must ensure that unauthorized users are prevented from executing operations that return information about the object's state, and to preserve integrity, we must ensure that unauthorized users are prevented from executing operations that modify the object's state.

4.1. Secure Object Automata

Let S and O be the sets of subjects and objects. Intuitively, S consists of all system users and programs, i.e., processes in our general model; O consists of all the resources to be protected e.g., files, directories, and laserwriters. Let M be an *access-rights matrix* [3] where the (i, j) th entry in M is a set of rights that subject $i \in S$ has for object $j \in O$.

Unlike standard models of security (e.g., Bell's [2] or Lampson's [3]) in our model, a right of a subject i is not just a name of an access mode (e.g., read, modify, execute) or operation (e.g., *Enq*, *Deq*), but is a pair of predicates (i.e., pre and postconditions) on the name of each operation of object j . For example, an entry for a subject P on a file f might contain the following pair of predicates for a write operation:

```
f:: Write(v : value) :: P
    requires id(P) = owner(f)
    ensures val(f') = v
```

where "*id*", "*owner*," and "*val*" would be defined in the appropriate traits. This element of the (P, f) entry in M restricts the process P invoking the write operation to be the owner of file f .

Definition 1: A history H is secure if for each operation " $A :: e :: P$ " in H there exists some (pre, post) pair of predicates for operation e in the (P, A) th entry of M such that $e.pre(s) \wedge e.post(s, s')$, where $s, s' \in \text{STATE}_A$ and s is the state of A upon invoking e and s' is the state upon return.

A *secure object automation* $\text{Secure}(A)$ is an object automation that accepts histories of the simple object automation A such that each history in $L(\text{Secure}(A))$ is secure.

```
Class: trait
    C enumeration of [operator, faculty, mailer, student]

Users: trait
    includes Class, Set[USet, User]
    User record of [id: Id, class: C]
    introduces
        ops:   USet → USet
        fac:   USet → USet
        mir:   USet → USet
        stu:   USet → USet
    asserts for all [u: U, us: US]
        u ∈ ops(us) = [class(u) = operator]
        u ∈ fac(us)  = [class(u) = faculty]
        u ∈ mir(us)  = [class(u) = mailer]
        u ∈ stu(us)  = [class(u) = student]
```

Fig. 2 Traits for users of the secure mail queue

Since an access-rights matrix M can be viewed as a set of permissions, we identify an environment's set of constraints (its "state") to be the complement of the set of permissions. Intuitively, constraints are prohibitions of the form "User X does not have the capability for operation Y on object Z ," that are formally derivable from the sets of pairs of predicates of M .

4.2. Secure Mail Queue

As an example of security object consider a *mail queue* used as a temporary repository for mail intended for other sites. The mail queue provides four operations: a user can enqueue a message, dequeue the oldest message from the queue, cancel an unsent message, and list unsent messages. Clearly, not everyone should be allowed to execute every operation. For this example, we recognize four disjoint classes of users (see Fig. 2): 1) *operators* may execute any operation, 2) *faculty* members may enqueue messages, 3) *mailer* processes may dequeue messages for transmission, and 4) *students*, naturally, have no privileges at all. The specification of the operations is given formally in Fig. 3. In this specification, we assume that each user U invoking an operation on the queue q has an unforgeable name ($id(U)$), and that any attempt to execute an unauthorized operation signals an *Unauthorized* exception.

We can use constraint diagrams to formulate a variety of alternative "less secure" mail queue specifications. Under ideal circumstances, each user has the set of capabilities appropriate to his or her class. The cost of preserving these constraints is the cost of keeping passwords secret, using secure encryption protocols, etc. An event that affects the environment occurs when a user acquires capabilities to which he is not entitled, increasing the set of possible behaviors. Since a user who has discovered a new security breach is always free to refrain from exploiting it, each such breach introduces the possibility of new behaviors without excluding the possibility of older behaviors, thus the resulting set of behaviors clearly forms a constraint diagram. The constraint diagram characterizes the extent to which the resulting insecure behavior is close to the preferred secure behavior.

Suppose Joan is a faculty member and Susan a student. Ideally, the following constraints hold (among many others):

- C1 Joan cannot dequeue messages;
- C2 Susan cannot list Joan's message.

If Joan discovers a way to relax constraint $C1$, then the specification for *Deq* would change as shown in Figure 4. Note that because Joan is always free to refrain from exploiting her knowledge, the precondition for the *Unauthorized* signal remains unchanged, and the corresponding automaton accepts a strictly larger language. Moreover, the specification is independent of how Joan manages to circumvent the prohibition against dequeuing messages.

Similarly, Susan might relax constraint $C2$ if he learns Joan's password. The resulting specification for *List* is shown in Figure 5. Here too, since Susan can always refrain from using Joan's password, the precondition for the *Unauthorized* execution is unchanged, and the set of possible behaviors is strictly larger. Finally, an even larger set of behaviors is possible if both constraints are relaxed.

Mail Q : trait

includes Bag with [Q for B]

$q:: \text{Enq}(m: \text{message}) / \text{Ok}() :: U$
requires $\text{class}(U) = \text{operator} \vee \text{class}(U) = \text{faculty}$
ensures $q' = \text{ins}(q, m)$
 $q:: \text{Enq}(m: \text{message}) / \text{Unauthorized}() :: U$
requires $\neg (\text{class}(U) = \text{operator} \vee \text{class}(U) = \text{faculty})$
ensures $q' = q$
 $q:: \text{Deq}() / \text{Ok}(m: \text{message}) :: U$
requires $\text{class}(U) = \text{mailer} \vee \text{class}(U) = \text{operator}$
ensures $q' = \text{del}(q, m)$
 $q:: \text{Deq}() / \text{Unauthorized}() :: U$
requires $\neg (\text{class}(U) = \text{mailer} \vee \text{class}(U) = \text{operator})$
ensures $q' = q$
 $q:: \text{Cancel}(m: \text{message}) / \text{Ok}() :: U$
requires $[\text{class}(U) = \text{operator} \wedge \text{isIn}(q, m)] \vee$
 $[\text{class}(U) = \text{faculty} \wedge \text{sender}(m) =$
 $\text{id}(U) \wedge \text{isIn}(q, m)]$
ensures $q' = \text{del}(q, m)$
 $q:: \text{Cancel}(m: \text{message}) / \text{Unauthorized}() :: U$
requires $[\text{class}(u) = \text{student}] \vee$
 $[\text{class}(u) = \text{faculty} \wedge \text{sender}(m) \neq \text{id}(u)]$
ensures $q' = q$
 $q:: \text{Cancel}(m: \text{message}) / \text{Absent}() :: U$
requires $[\text{class}(U) = \text{operator} \wedge \neg \text{isIn}(q, m)] \vee$
 $[\text{class}(U) = \text{faculty} \wedge \neg \text{isIn}(q, m)]$
ensures $q' = \text{del}(q, m)$
 $q:: \text{List}() / \text{Ok}(p: \text{queue}) :: U$
requires $\text{class}(U) = \text{operator} \vee \text{class}(U) = \text{faculty}$
ensures $q = q' \wedge$
 $\text{class}(U) = \text{operator} \Rightarrow \forall m. (\text{isIn}(p', m) \Leftrightarrow \text{isIn}(q, m)) \wedge$
 $\text{class}(U) = \text{faculty} \Rightarrow \forall m. (\text{isIn}(p', m) \Leftrightarrow$
 $\Leftrightarrow (\text{isIn}(q, m) \wedge \text{sender}(m) = \text{id}(U)))$
 $q:: \text{List}() / \text{Unauthorized}() :: U$
requires $\neg (\text{class}(U) = \text{operator} \vee \text{class}(U) = \text{faculty})$
ensures $q = q'$

Fig. 3. Secure mail queue trait and interfaces

The corresponding constraint diagram is shown in Fig. 6. Each of its nodes represents one of the alternative system specifications. In our example, the system specification given in Figure 3 corresponds to the top node of the diagram. If, somehow, system constraints get relaxed the specification has to be modified in the way shown in Figures 4 and 5. Changes shown in Figures 4 and 5 apply to the specifications represented by the left and right node in the second row of the constraint diagram, respectively. At last, the bottom node represents specification which contains both modifications.

For security, a system should ideally never stray from the preferred behavior; system designers build in clever encryption and file protection schemes to ensure that "at all costs" secrecy and integrity of data are preserved. Thus, the designer would not explicitly handle cases of unpreferred behaviors because they should never occur. In practice, however, no matter how much foresight designers have put into systems, most do have "holes". Some of these breaches are more severe than others in terms of the damage done and the cost of repair. For example, if Susan finds out Joan's password because Joan was careless, then that is not as bad as if he found out her password by having gained access to a password file, thereby gaining to everyone directories. At practically zero cost, Joan can simply change her password, whereas having everyone change their passwords would impose a nontrivial (and aggravating) cost. Hence, given some security breach, the designer can traverse a constraint diagram to identify how much damage to the system was incurred and its cost of repair.

```

q:: Deq() / Ok(m: message) :: U
requires class(U) = mailer  $\vee$  class(U) = operator  $\vee$  id(U) = Joan
ensures q' = del(q,m)
q:: Deq() / Unauthorized() :: U
requires  $\neg$  (class(U) = mailer  $\vee$  class(U) = operator)
ensures q' = q

```

Fig. 4 Changes to interfaces if Joan can dequeue messages

```

q:: List() / Ok(p: queue) :: U
requires class(U) = operator  $\vee$  class(U) = faculty  $\vee$  id(U) = Susan
ensures q = q'  $\wedge$ 
   class(U) = operator  $\Rightarrow \forall m. (isIn(p', m) \Leftrightarrow isIn(q, m) \wedge$ 
   class(U) = faculty  $\Rightarrow \forall m. (isIn(p', m) \Leftrightarrow isIn(q, m) \wedge sender(m) = id(U))) \wedge$ 
   id(U) = Susan  $\Rightarrow \forall m. (isIn(p', m) \Leftrightarrow (isIn(q, m) \wedge sender(m) = Joan))$ 
q:: List() / Unauthorized() :: U
requires  $\neg$  (class(U) = operator  $\vee$  class(U) = faculty)
ensures q = q'

```

Fig. 5 Changes to interfaces if Susan learns Joan's password

The constraint table shown in Figure 6 includes main items of designer concern arranged systematically. The correctness condition for the e-mail example pertains to preserving system's security and integrity. The preferred behavior column states the name of data structure used to describe the intended system's behavior (in our case e-mail queue). The system's constraints considered here enforce access rights. The incurred cost relates to the cost of protecting passwords. Finally, the set of events which may cause a change to system's behavior taken into account here comprises variations of obtaining passwords action.

5. CONCLUSION

In this paper we have given only a very simple example of system's security analysis. This should not be misleading. On the contrary, the constraint diagram method is able to capture security properties of large complex systems by offering abstract and graphical interface to users. We have been developing a tool, based on the method, intended to assist:

- 1 - Designers as an after-the-fact investigative tool to identify unforeseen security holes in system implementations;
- 2 - System administrators to access damage incurred by a security breach.

At present, we are experimenting with specifying various system's security properties for which we have derived constraint diagrams. We found them useful in identifying those classes of security holes that are either not explicitly or inappropriately handled. Constraint diagrams can also help to identify the costs associated with patching an existing system to handle previously undetected fault classes or making a new system more secure in the anticipation of security breaches.

An important aspect of our method is to make explicit the role of the environment in its effect on the observable behavior of the rest of the system. In particular, we hold the environment responsible for security breaches. Our method provides a way to explain the behavior of the rest of the system when such failures occur. In addition, it may provide a unified and general framework for specifying system behaviors, and characterizing the essential tradeoffs between the costs of preserving correctness properties and the costs of relaxing them not only concerning security properties but also the other aspects of the information system.

REFERENCES

1. N.G. Leveson, " Software Safety: What, Why, and How," ACM Computing Surveys," June 1986, pp. 125-164.
2. D.E. Bell, " Secure computer systems: Unified exposition and multics interpretation," Tech. Rep. ESD-TR-75-306, The MITRE Corp., Bedford, MA, March 1976.
3. B.W. Lampson, " Protection," ACM Oper. Syst. Rev., Vol. 19, No.5, Dec. 1985, pp. 13-24.
4. J.V. Guttag et al., " The Larch Family of Specification Languages," IEEE Software, Vol.2, No. 5, September 1985, pp. 24-36.
5. R. Morris and K.L. Thompson, " Password security: A case history," Commun. ACM Vol. 22, No. 11, November 1979, pp. 594-597.
6. L. Gong, et al., " Protecting poorly chosen secrets from guessing attacks," IEEE J. Selected Areas in Communication Vol. 11, No. 5, June 1993, pp. 648-656.