Towards understanding and providing assurance about separation

Daniel Drozdzewski, Adrian Baldwin, Patrick Goldsack
Trusted Systems Laboratory
HP Laboratories Bristol
HPL-2007-3
January 26, 2007*

separation,
containment,
utility computing,
HPUX,
assurance

One of the key aspects of securing a system is to ensure separation and containment between different concerns. This could be between processes and communications within a single machine; through to different applications and network segments in an enterprise to customers in a shared data centre. Containment is generally achieved through a variety of often complex mechanisms making it hard to configure and even harder to assure users that the desired containment relationships are maintained.

In this paper we present an approach to assuring users about containment of systems by developing an abstract containment model suitable for many situations. This model then has detail added, through a series of refinements, to become closer to the implementing technologies. We present a refinement for compartments recently added to HPUX. We then show how we can provide assurance reports to users demonstrating that the containment properties in the model are being achieved.

* Internal Accession Date Only
Approved for External Publication

Towards understanding and providing assurance about separation

Daniel Drozdzewski
Hewlett Packard Labs
Bristol, UK
daniel.drozdzewski@hp.com

Adrian Baldwin
Hewlett Packard Labs
Bristol, UK
adrian.baldwin@hp.com

Patrick Goldsack
Hewlett Packard Labs
Bristol, UK
patrick.goldsack@hp.com

ABSTRACT

One of the key aspects of securing a system is to ensure separation and containment between different concerns. This could be between processes and communications within a single machine; through to different applications and network segments in an enterprise to customers in a shared data centre. Containment is generally achieved through a variety of often complex mechanisms making it hard to configure and even harder to assure users that the desired containment relationships are maintained.

In this paper we present an approach to assuring users about containment of systems by developing an abstract containment model suitable for many situations. This model then has detail added, through a series of refinements, to become closer to the implementing technologies. We present a refinement for compartments recently added to HPUX. We then show how we can provide assurance reports to users demonstrating that the containment properties in the model are being achieved.

1. INTRODUCTION

Providing barriers that enforce separations is fundamental to security whether it be physical security with walls and locks or computer security with logical separation mechanisms. Good system security relies on having a number of separation mechanisms at different levels within a system so that defence in depth can be achieved; if one barrier is broken all need not be lost. However, having strong separation mechanisms does not guarantee security; they must be used properly. Troy did not fall due to weak walls but due to weak processes in managing the import of unauthorised wooden horses.

Over the years many separation mechanisms have been developed to support computer security (and fault tolerance) and these mechanisms work over the full stack from hardware through to applications and management. For example:

Ring Architectures within processors ensure there is a strong separation between user and kernel level code limiting access to privileged instructions and memory segments. More recently ring architectures are being expanded to support multiple layers providing better segmentation with virtualisation.

VLANs [8] allow multiple separate LANs to be run over the same physical network by tagging packets at the network layer according to the logical LAN on which they reside.

VMM [2] allow different OS images to be run on the same physical hardware with the VMM acting as a separation mechanism between the systems.

Kernel imposed separation. As standard the operating system kernel provides separation between processes particularly protecting memory access and access control. Trusted operating systems [5] [11] add in a set of mandatory access control rules limiting the communications and accesses between different system compartments

Application sandboxing such as that provided in java provides protection from application code accessing resources to which they are not entitled.

These types of mechanisms are often used in combination to help secure solutions or even to produce shared flexible infrastructures such as SoftUDC [16]. The combination of these techniques can provide very strong security; for example, keeping customers sharing a data centre separate. However, these mechanisms can be hard to manage and configure; thus the strength of the solution is often only as good as the management. Management is often error prone demonstrated by studies [23] that have shown up to 80% of firewalls are incorrectly configured. Thus leads to the two problems addressed by this paper: firstly, are the desired separation goals being achieved and secondly how do we convince those people accountable for the solutions that this is the case.

Understanding a solution often requires the skills of multiple people working at different layers including application experts, a database administrator, operating system administrators, and networking experts. Equally responsibility for

security falls into these different domains and as such a clear vision of the security requirements can be lost. We start this paper with the development of an abstract model for specifying containment relationships that should exist within a solution. Having a simple abstraction for expressing containment and communications between containers allows solution architects to think about the required security properties as well as providing us a set of constraints from which we can measure and compare a working system. This allows us to produce assurance reports for customers demonstrating that the system is maintaining the desired containment and separation properties.

In the next section we describe the abstract modelling approach for expressing containment that we have developed. This is followed by a description of the containment and associated role based privilege mechanisms recently introduced into HPUX and on which we demonstrate the approach. We then show how we refined the abstract model to more closely fit the technology whilst maintaining the abstract properties. Finally we show how we used the model as part of an assurance framework to help demonstrate that the system is secure and that the containment relationships are being maintained. This is followed by a discussion of the wider issues surrounding containment modelling and assurance.

2. CONTAINMENT MODEL

At the abstract level we took an approach to modelling containment defining things in terms of generic concepts of containment and communication channels. The modelling consisted of a way of representing the containment state, a set of invariants that must be true for all models and functions for modifying the state of the model. In this way the model can be changed as the solution changes whilst maintaining a fixed set of properties.

2.1 Model elements

The abstract model was formed from five main concepts that allowed us to express containment; communications between containers and ownership of the containers.

Elements. The concept of an element was used to describe things that are being contained. In a shared data centre scenario this would include physical or virtual machines. Where containment is being enforced at a finer gained level this can include processes.

Container A container is defined as a collection of elements whose communication within the container is not constrained but whose communications outside of the container are constrained to only those whose permission to communicate is defined within the model.

Communication channel A communication channel is the definition that entities in two different containers can in some (unspecified way) communicate. We purposely kept this at a very abstract level with the intent of refining it further once technology choices have been made (see section 5).

Owner The owner is the person or entity who is in someway responsible or accountable for the container. The concept of ownership can vary according to the level of abstraction. At the shared data centre model it is the company using the contained entities. Within an enterprise it may be the business (defined by the enterprising architecture) as owning the application.

Permission Permissions go along with the communication channels and are the definition that allows communication channels to be used. The approach we used to modelling did not explicitly create or define communication channels but inferred their existance from the appropriate set of permissions.

```
type Entity;
type Conatiner;
type Owner;
syntype Permission == Conatiner * Container;

syntype State==(Owner -m> set Permission) *
               (Owner-m> (container -m> set Entity));
```

Figure 1: The basic types for the containment model

2.2 Model Definition

The containment model was build using HPSL [3] [13] [10] (HP Specification Language) and started by defining the basic types of Entity, Container and Ownership. Permissions are defined as a relationship between two containers. The overall state is defined in terms of relationships. Firstly, that between the owner and the sets of permissions that they define and secondly between owners and a container to entity set relationship. The first element defines communication channels between containers and the second defines the entities within a container along with the owner of the container. Along with the state definition we created a set of basic functions allowing definition of the individual elements.

The basic definition of a state allows invalid states to be created and so a further type of a ValidState was created which includes a number of invariants that must always be true for the containment description to be acceptable. These included:

- All entities can only exist within a single container (see example)
- Each container is only owned by a single owner.
- Entities within containers can talk. That is, there is permission for the container to talk to itself.
- The owner of the permission must be the owner of the container at the lefthand side of the permission.

From the valid state definition we then have a series of functions for changing the state and ensuring that the state maintains the invariants. These functions (See Figure 3 for an example) take a valid state type and define how it is changed based on the new state change information. Each function is defined with pre and post conditions defining the

```
syntype ValidState == State inv s as (p,o) .

    //permissions and owners are
mappings from the same domain
     dom p = dom o

     and

     //each entity exists in exactly one container
     //which is owned by exactly one owner
     (forall e IN allEntities s .
          (exists1 oe:Owner, ce:Container .
               e in ((o oe) ce)
          )
     )
     and
...
```

Figure 2: Example invariant in the valid state definition

```
fn      addPermission:
ValidState * Owner * Permission
-> ValidState is
        addPermission (s as (per, own), oa, pa)

        pre     pa not in allPermissions s and
                oa in allOwners s


        return  s' as (per', own')
        post    per' oa = per oa union {pa}     and
                own' = own                      and
                (forall o in dom per. per o = per' o)
```

Figure 3: Functions for adapting the model

```
fn  canCommunicate: ValidState X Container X Container -> B
    canCommunicate (s as (per, own), ca, cb)

    pre ca in allContainers s    and
        cb in allContainers s

    == (ca * cb) in allPermissions s and (cb * ca) in allPe
```

Figure 4: The function defining the communication channel from the permissions

states and being a valid state type the invariants must also be preserved.

The concept of a communication channel is important within the containment model. It is not directly represented instead it is implied by permissions. Communication channels can now be defined by a function (see figure 4) that given a valid state and two containers returns true or false depending on whether there is a direct communication path. We define the existence of such a path as meaning there are a matching pair of permissions (one from each owner).

Further functions can be defined to explore paths between the various containers. As a more practical way to animate the model we produced a PROLOG [22] translation that allows the model to be animated and queried. This model includes predicates that find all paths between two given containers (or all pairs of containers). This PROLOG model also forms the basis of state validation work described in section 6.3.

3. HPUX COMPARTMENTS

Before discussing refinements of the model and how it is used to check and report on the state of containment we describe the main features. This section starts with a background on compartmentalisation within unix which contrasts having fixed but inflexible containment models with having more flexible managable models but which need to be properly managed. We then provide more details of the HPUX model [14]. As well as building on containment, ownership is an important aspect and as such we describe the role based privilige system that we use as a basis for ownership.

3.1 Background

Compartmentalised UNIX systems [7] were developed to meet the needs of the military security and are based on the Bell-LaPadula [5] [4] multi-level security systems with fixed write up and read down rules. These rigid policies form Mandatory Access Control rules that are enforced within the OS kernel. Having such a fixed model supported within the OS kernel means that assurance that the correct containment model is being achieved is not an issue. The model is effectively built into the mechanism and cannot be changed.

Whilst the mandatory access control model met the needs of the military they proved too inflexible for many commercial applications. New solutions such as a trusted Linux system [11] provided more flexible approach to containment where compartments can be defined along with communication channels between them. These rules or compartment

```
Compartment Name: webserver : sealed
 Disallowed Privileges: POLICY

 Network Communication Rules:
 ----------------------------
 ACCESS        PROTOCOLSRCPORT DESPORT DESCMPT
 grant client  tcp     0       3306       dbserver
 grant client  udp     0       0       dbserver
```

Figure 5: Communiction rules within HPUX compartments

policies can be configured according to the needs of a given solution and are then enforced by the kernel. With flexibility comes the problem of ensuring that the correct containment model is being maintained and assuring those reliant on the system that separations are being maintained.

The use of compartments within UNIX systems provides mechanisms that ensure that if one application or network service is compromised then the attackers access to the underlying system is limited. For example, if a web server were to be subverted giving an attacker root access to that compartment then they would only be able to gain read access to a file system and very limited communications with other applications based on the given rule set.

3.2 HPUX

An application can be configured to run in a given compartment where it has limited access to resources such as other processes, files and communication channels outside of its compartment. Policies are specified within a configuration file defining the resource accesses that are allowed. The compartments are configured and reconfigured using a priviliged command.

Processes can be labelled so that they always run within a gvien compartment or they can be started from a given compartment. Rules can be set so that processes forked within one compartment start in another compartment. The definition of a compartment also includes a set of rules specifying the communication channels available to processes. These may be IPC channels within the machine or network rules limiting how processes within the compartment can talk via specified network cards. Examples of such rules are shown in figure 5 where the initiation direction of the rule is specified along with the ports over which it can run; ports can be tightly specified or wild cards can be used for example specifying that incoming connections can be received on port 3306 (MySql default port) from any source port.

3.3 Privilige Management

Clark and Wilson [9] worked on an integrity model to help overcome some of the short comings of the Bell-LaPadula model. This has an access control model based on an access triple $user; op; data$ to state that a user has the rights to perform a given operation on a given set of data. HPUX uses this type of triple model to manage access to privileges and the systems over which they can apply. However the management of them can become complex and as such an RBAC privilege management system has been implemented.

Role based access control (RBAC) was initially proposed by Sandhu [20] as a way of simplifying the administration of access control rights. In the RBAC model permissions are formed from the ability to perform a given operation on a resource or data set. Roles are created to correspond to job functions or tasks that require a set of privileges. Users are then allocated rights to perform a role hence simplifying the association between users and permissions. RBAC usually includes the concept of a role hierarchy so that a role gains all the permissions of its child roles; although this can aid administration, care should be taken that it doesnt create powerful individuals; and break separation of duty policies [12].

HPUX supports the creation of permissions over sets of resources by allowing a privilige to apply to a resource or set of resources as specified by a wild card. Roles are associated with these permissions in a way that allows role hierarchies to be created. Users can then be associated with a role. Each time a user attempts to execute a privileged command the kernel checks the permission set for the roles that the user holds (including child roles in the RBAC hierarchy) and the command is only executed if a match is found.

4. MAPPING HPUX TO THE MODEL

Our containment model has the concepts of containers, permissions, owner and entities and when applying it to the HPUX system we need to map these concepts to the underlying technology concepts described in the previous section. Clearly the containers map to HPUX compartments as these form the basic units of containment. Processes are the things that are being contained and hence they become the entities. The rules specified within the compartment communication channels map on to the permissions within our model although clearly the HPUX rules are much richer than our simple bidirectional communications.

Ownership, as a more abstract and less well defined concept, was a harder concept to map and here we chose to map roles to owners (assuming a flat role model). This was done as roles represent the grouping of permissions responsible for a given task. The mapping of a user to multiple roles could in some way break our ownership invariants which imply there should be additional checks within an assurance system. The continued existence of a root account also proves problematic for the ownership model in that it effectively holds all roles and if not removed or locked down provides a weak point in the containment model.

Mapping of processes to entities is again not a clean mapping as processes can be dynamically started and they do not have clear identities. Instead we mapped entities to applications (or commands) that are configured to start within a specified compartment. This information is readily available within configuration files and reflects how a server would be set up and managed. As with the user accounts an assurance report could be created that looks for other (unknown) processes running within a compartment hence demonstrating the risk of unknown entities is being mitigated.

5. REFINING THE MODEL

From the above description it is clear the initial approach to modelling has abstracted away too many details from the

```
type Machine;
type ContainerName;
syntype Container == Machine * ContainerName;

fn  allMachines:    State -> set Machine is
    allMachines (s as (per,own))
          == { left c | c in allContainers };

fn   containersOnMachine: State, Machine->
                              set Containers is
     containersOnMachine(s as (per, own), m)
         ==
          { right c| c in allContainers(s)
               and left s==m}
```

Figure 6: Functions for adapting the model

technology being modelled. Having said this, the highly abstract nature helps in producing a broad containment model which then needs further refinement to bring it closer to the particular technology being used. If the underlying technology were to be changed the first abstract model of containment would not change but this refinement may need to take account of different details.

Here we looked at two simple refinements to the original model that provide detail on the original model. Firstly we refined the notion of a container to better identify it as a compartment on a given machine and secondly we expanded the notion of communications to deal with different types of IP connections. In refining this latter element we concentrated on IP rules rather that IPC rules. The IP rules provided the richest refinements and the model could easily be further refined to deal with the simpler IPC communication channels.

5.1 Container Identity

Containers now take on a more concrete view in that they are compartments on given machines and this description provides a way of identifying any given container. We now define two new types of a ContainerName and a Machine and change the previous Container type to be a composite type formed from these two types (see figure 6).

The impact of this change on the model is then minimal; however, it does allow us to add further functions that allow us to look at containers that exist on particular machines. We considered adding an additional invariant to specify that all containers on the same machine must have the same owner; such an invariant would be necessary where the root privilege cannot be separated and shared out between the different container owners.

5.2 Communications

Communication channels within the original model were modelled by having duel permissions between two containers and in refining the model we retain this concept but introduce a further definition of the communication channel being authorised on each side. The communications function is no longer as simple as specifying that there are permissions on both sides; the permissions must now in someway match.

Permissions were modified from simply being a mapping between two containers to being a type defined by a mapping between the two containers and a connection type. The connection type is itself quite complex in that it describes an IP connection; hence it is constructed as a mapping between direction, protocol, source port and destination ports (see figure 7).

The previous model has a specification that a container must be allowed to talk to itself and this invariant had to be updated to take account of the connection within the permission type.

The canCommunicate function has an increase in complexity to deal with the additional complexity of the permission which involved introducing a new function to deal with matching rules. Here the rules have to be matched so that a client communicates with a server (or either being bidirectional) and source and destination ports need to be made to match.

6. ASSURANCE AND REPORTING

The purpose of the containment model is to be able to demonstrate to those relying on the systems that certain separation properties are being maintained. Providing assurance about containment is unlikely to be sufficient to assure those relying on the ICT systems that risks are properly managed and mitigated. As such we have taken the approach of using the containment model to perform tests within the context of an overall assurance framework which analyses and reports on risks and mitigations over many aspects of a system. The next subsection briefly describes this assurance framework with the remaining three subsections addressing how the containment model was used to report on the HPUX system.

6.1 Assurance Framework

An assurance framework [1] has been developed to support automation of audit and security testing along with reporting of risks to various stakeholders within an enterprise. At a high-level the framework allows elements within the enterprise architecture to be represented along with the enterprise control framework. This control framework consists of identified risks along with policies and controls that help mitigate these risks. At the lower level the framework supports the creation of tests to check that controls are properly enforced. A series of test mechanisms for comparing data in different ways are built into the framework and an API allows the test set to be easily extended.

The assurance framework collects data into an audit database and the assurance description is then used analyse the data to find cases where controls or polcies have not been properly followed. A hierarchical report is created with traffic light indicators at the top level showing the overall status of different areas of risk. Where these indicate a problem a user can then dig down through the report to identify which controls or policies are not being followed. Detailed tests then show the cases where they fail and allow for further investigation. The assurance reports are produced at regular weekly or monthly intervals and report on the overall

```
type Direction = client | serv | bidir;
type Protocol = udp | tcp | raw;
syntype Port = Nat0 inv n . n < 2**16;
syntype Connection==Direction*Protocol*Port*Port;

syntype ValidState ...
...
 and
    //entities within container are allowed
        to communicate anyway required
    (forall c IN allContainers s.
        (forall p1, p2: Port, dir:
                    Direction, proto: Protocol.
            (c , (dir, proto, p1, p2) , c)
                            in allPermisions s
        )
    )

 and
...

fn  canCommunicate: ValidState X Container
                        X Container -> Boolean is
    canCommunicate (s as (per, own), ca, cb)
    pre    ca in allContainers s    and
           cb in allContainers s

      ==(
        exists ab_connection, ba_connection:
     Connection .
            (ca, ab_connection, cb) in
                            allPermissions s
       and
            (cb, ba_connection, ca) in
                            allPermissions s
       and
            ruleMatch(ab_connection, ba_connection)
      )

fn  ruleMatch: Connection X Connection ->
                                        Boolean is
    ruleMatch(cA as (directionA, protocolA,
                                portAa, portBa),
              cB as (directionB, protocolB,
                                portBb, portAb) )
    ==
    {
        (
            {directionA, directionB} =
                                {client, serv} ) or
            bidir in {directionA, directionB}
        )                           and
        portAa = portBa         and
        portAb = portBb         and
        protocolA = protocolB
    }
```

Figure 7: Adding to the permissions within the refined model including refinements of the invariants and communciation functions

approach to operational risk in contrast to real time threat analysis.

As such the assurance framework provides an aid to auditors who would typically perform manual audits on samples of data as well as providing reports for various stakeholders within the enterprise demonstrating how well risks are being mitigated. Pilots carried out with auditors and security officers have demonstrated the value of the approach in demonstrating appliaction and IT infrastructure are under control.

6.2 Pulling data

Prior to producing plug in tests for the assurance framework it is necessary to be able to pull data from the unix systems into the audit database. This was done using java agents that ran the appropriate commands on the unix machine to pull the configuration state from the compartmentation system and the configuration of the RBAC privilige system. The output of these commands is parsed and stored in tables in the database that mirror the model structure as described in section 4.

We took the approach of pulling a consistent snapshot of the state of the unix system since this gives us a state that can be compared to the desired state. There are issues about how this snap shot should be triggered; this could be done via monitoring the syslog for changes but this may fail where logging is turned off. Alternatively regular snapshots could be taken and validated against the model.

6.3 Testing the model state

The containment models we developed are logical models that we have translated into PROLOG to explore the model whilst maintaining the logical relationships. In validating the state of the system against the logical model we decided to uses these PROLOG versions of the model. To achieve this we integrated JLOG [15] (a java based PROLOG) into the assurance framework allowing the containment model to be used directly.

The assurance framework manages getting data within a given reporting time period from the audit database and this data was converted into a state representation that follows the model. Using the model we performed two types of testing:

Invariants Here we took the state as derived from a HPUX system and validated the invariants.

Communication path checks Here we used the model to compare the actual state of the system to a desired state described within the model. Both these types of testing were performed using the PROLOG model along with the actual state from the system. Results are passed back into the assurance system where they are stored in a database ready for reporting.

6.3.1 Invarient Checking

The simplest form of invariant checking is to simply return a Boolean value resulting from testing all the invariants. Whilst this is useful, ideally the assurance system helps

isolate the problem. To do this each invariant was written as a separate predicate with an overall invariant being constructed as the conjunction of the individual predicates. This allows us to write an invariant checking predicate that checks each individual invariant predicate and lists those that fail.

6.3.2 Path Checking

The main aim of modelling is to ensure that a containment model is being achieved within a system. To do this we have a desired state of the containment and need to compare this to the actual state derived from the systems. The testing needs to highlight the differences between the desired state and actual state.

There are a number of different facets that we could explore within the differences and here we have concentrated on checking that additional communication channels that emerge in the actual state but that are not specified in the desired state.

Firstly we performed a simple test to find additional compartments that exist within the actual state but not within the desired state. This is done using an allContainers predicate within the model that returns a set of all containers and then performing a simple comparison finding additional elements not in the desired state. These additional compartments were returned to the assurance system.

More importantly we looked for additional communication channels that exist between compartments. We can list all direct communication channels between compartments using the canCommunicate predicate (see figure 7) and this can be done for both the actual and desired states. The two sets of communication channels can now be compared; this is not quite as simple as just looking for the difference between the two lists. Communication channels are described in terms of a source, destination, and a connection and we also check that connections match the specification reporting on either more permissive or less permissive connections.

Where there are additional containers within the actual state and they talk to containers within the desired state they may help form unintended indirect communication channels. When this is the case we use a findPaths predicate in the model to find any of these indirect channels between our known containers. These indirect paths are also returned to the assurance framework.

6.4 Containment reporting

An assurance description can now be given for the HPUX system that includes the invariant and the path checking tests. This allows the assurance system to construct a report showing that the HPUX system is in someway well run. Here we have concentrated purely on testing containment properties but the assurance report should include other information useful for those assessing the system. For example in section 4 we raise the need to look at root account usage; other elements in the report should look at how many users have priviliges; failed logins; changes to ip filter rules; new accounts added and so on.

Within this report we would include the results from the model testing. This would include a good/bad status for the invariants along with a list of any that have failed. The path testing checks list the three aspects tested:

1. A list of any additional compartments
2. A list of direct communication channels that do not match the desired state. Each entry in the list contains fields listing the source compartment, the destination compartment, the connunication rule for each and a flag stating the rule is more permissive, less permissive, an additional rule, or a missing rule.
3. A list of additional indirect communications due to additional compartments. This list includes the source and destination compartments along with the compartments forming the path.

The report for the path testing also includes counts of these different attributes allowing the results to be interpreted within the assurance description. For example a threshold may be set to report the state as being bad if there are additional communication channels but warnings would be given where there are additional compartmenst; such judgments would of course depend on the percieved risk from the different failures.

6.5 Deriving the model

One of the issues with the approach as described is that the containment model is in someway contained within the assurance description. This implies that it is thought about up-front and is only changed as part of a review of the control framework. This should not be the case, particularly in the context of a shared dynamic data centre which would add and remove compartments as required. From this perspective the desired state model should be contained within a database and maintained through a portal that allows the customer to describe the systems they require. Such a setup allows the containment model to be used to demonstrate that the service provider is meeting the customers separation requirements. From the testing perspective achieving this is simply manipulating the place where the model is picked up from. However, this leaves the harder problem of how the customer selects and manipulates their system and therefore the containment requirements through the portal. The modelling approach we have taken does support the model being manipulated and changed through a number of functions.

7. DISCUSSION

7.1 Meeting assurance goals

As outlined in the introduction there are many strong mechanisms to enforce separation in shared systems, the point of this work is not to describe or promote the compartment or RBAC mechanisms used by HP-UX. Rather it has been to focus on how stakeholders can gain assurance that separation mechanisms are managed and configured correctly. The abstract and refined models provide a basis for collecting, filtering and analysing the large amount of system information available. As such the questions for discussion are:

1. Who would use these reports for assurance and how useful are they?

2. Do the models or the modelling process bring more confidence to the resultant reports?

3. How representative are the HP-UX environment and mechanisms studied, e.g. how well will refinement work for other mechanisms?

Taking these in turn, there are multiple assurance stakeholders involved in shared environments. The simplest scenario might be the need to control the communications (concerns) between an external facing website, and the back end database it is relying on. Looking more to the future, we envision shared data centres operated by a service provider. In this case the service provider will want the reports to convince them that the way the environment is being operated is keeping their customers appropriately separated. In both these cases there will be multiple assurance stakeholders including: system/application owners, the customers, administrators and even the customers' auditors who, prior to checking the controls are working, need to be assured that the isolation is effective. The resultant reports show graphically and intuitively whether the recently grabbed state of the system conforms to the abstract desired properties of the model, and where it does not, provides simple navigation to interpret the problem; a red light, or highlighted metric quickly leads to the user to the invariant, or part of the state with a problem. For example, a report may highlight that an additional communication path has been added between the backend container and the application server container allowing say web traffic (port 80) as well as the specified JDBC traffic (port 3306).

Intuitively this feels useful, especially if it is compared with current state of the art where multiple scripts are written and run as required along with the collection of vast amounts of log information. From a security perspective this information is sometimes fed into security event correlation mechanisms to detect immediate threats but sifting through the data to validate the way the system has been run and managed is usually a human process. Here the having a containment model helps document and provide a basis from which assurance can work effectively. That said, this work did not go as far as testing the usefulness or usability of these reports with auditors, system/application owners or other assurance stakeholders. This kind of validation has been done for similarly structured reports with models for testing standard IT process controls [1].

For the second question, the modelling process clearly brings more rigor to the analysis process, and it is easy to relate reports back to the orginal modelling done. As such if the model makes intuitive sense, the results should be comprehendable and thus increase confidence. A natural question to ask is on what basis can we trust the validity of the model and its refinements. The goal of this paper is to explore the model-reporting loop and make judgements about the usefulness of the approach. Interesting, but slightly orthogonal issues such as proving the validity of the model or that certain operations do not affect invariants have been left as further work.

For the third question, clearly the compartment mechanisms mapped nicely to the containment abstraction. Another well conforming example could have been to model the mechanisms for isolating virtual infrastructures sharing the same physical infrastructure such as that described in the SoftUDC paper [16]. In this case containers would be virtual subnets, entities would be virtual machines, and the separation (mechanism) would be enforced by the configuration of the virtual networking. So this shows the approach is not overly specific, however it remains as further work to look into the feasibility to use the model on very different mechansism such as multi level security systems described in section 3.1, and also to look into shared storage environments such as SANs.

The reason it is not enough to focus on the mechanism is that the IT environment is always changing. As such the link between the models and the reality of the environment may also be subject to change. IT departments are moving much more to standard processes for change management, i.e. ITIL [17], which amongst other things ensure good planning and scheduling of all changes. In such an environment all changes (i.e. deviations from the model defined state) should be explainable by planned changes, so it should be possible not only to report deviances, but also whether such changes were planned. If the changes really break the model then this should have been realised in the planning process, and either the model will need to be changed or there should be a wider review of the required change. As mentioned the modelling approach has been applied on process checks, such as change control so it should be possible to model and check that events and deviations correspond to planned changes. So whilst some of the above should be possible, further work is needed to understand how the model could be maintained as part of a wider change process, where ultimately changes are all expressed through the model, and this automates both the change and assurance reporting associated with it.

7.2 Policy based management

Clearly this work relates to ideas within policy based management [21] where polices are constructed and used to help manage the systems. These approaches often try to control the system via policies, for example, trying to use policy statements to configure or maintain the configuration of a system. This can lead to systems with a proliferation of low-level policies that lose the high level abstraction that made high level policies desirable in the first place. This can be contrasted by the use of a model to validate aspects of the system. This is very much a simpler and more tractable task now detailed configurations need not be considered just those related to the aspects being modelled and hence the model retains a degree of abstraction ensuring it clearly expresses useful concepts without getting lost in detail.

7.3 Data Security

Currently assurance is a manual process where data is periodically pulled from a system and samples are examined and analysed manually by auditors. Clearly bringing automation to these manual processes brings benifits in reducing errors both through removing the need to sample, the reduction in human error and by having more regular assurance reports. Currently there is little concern over the security of the actual audit data being analysed. However we believe in the

longer term and particularly with the emergance of dynamic and shared data centres there will be a need to demonstrate the integrity of the data. There are two main issues within this problem: the first is the trust in the logging mechanism or commands that produce the data and the second is in the integrity of the audit trail.

A very sophisticated attacker will try to cover their tracks and install or change the system so that it fails to report their presence. In our HPUX example this would involve changing or patching the underlying OS so that commands do not show the presence to their communication channels or the kernel does not block the channels. The trusted computing group [19] provides mechanisms for measuring the OS systems as they boot and based on these measurements further auditing agents can be measured hence reducing the possibility of an attacker going unnoticed.

On the data integrity side an attacker could try to attack system log files or audit files before they are relayed to the central database. In the case of a shared service utility there needs to be trust in the audit trail itself demonstrating that the service provider has not changed it to support their case in any dispute. There are a number of secure audit mechanisms for both securing audit data whilst in a hostile environment using forward integrity mechanisms [6] and then securing the overall audit trail as a whole even from interference from the service provider [18].

8. CONCLUSION

This paper starts by discussing the problems with relying on security based on flexible but strong security separation mechanisms when the management of these mechanisms is not properly checked and validated. We have presented an approach of capturing the separation requirements within an abstract model that can be further refined to ensure it meets the needs of the given seperation technologies. We have shown how such a model can be used to check that compartment based separations are being correctly maintained in an HPUX system. We have also shown how such containment testing can be used as part of an overall assurance report demonstrating that risks are being mitigated appropriately.

9. REFERENCES

[1] A. Baldwin, Y. Beres, and S. Shiu. Using assurance models to aid the risk and governance lifecycle. *BT Technical Journal*, in press.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[3] S. Bear. *An Overview of HP-SL*. HP Labolatories, Bristol, March 1991.

[4] D. E. Bell. *Looking Back at the Bell-La Padula Model*. Proceedings of the 21st Annual Computer Security Applications Conference. Reston, VA, 2005.

[5] D. E. Bell and L. J. L. Padula. *Secure Computer Systems: Mathematical Foundations*, volume 1 of *MTR-2547*. The MITRE Corporation, Bedford, MA, 1973.

[6] M. Bellare and B. Yee. Forward integrity for audit logs. Technical report, UCSD tech report, 1997.

[7] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. *Compartmented Mode Workstation: Prototype Highlights*, volume 16 of *IEEE Transactions on Software Engineering*. June 1990.

[8] N. G. Chan Wai Kok, M. Salim Beg. *Inter Bridge VLAN Registration Protocol for IP Subnet VLAN*. Proceedings of the 25th Annual IEEE Conference on Local Computer Networks. November 2000.

[9] D. Clark and D. Wilson. *Comparision of comercial and military security policies*. IEEE Symposium on Security and Privacy. Oakland, April 1987.

[10] J. L. Cyrus, D. Bledsoe, and P. Harry. Formal specification and structured design in software development. *Hewlett Packard Journal*, 1991.

[11] C. Dalton and T. H. Choo. *An operating system approach to securing e-services*, volume 44 of *Communications of the ACM*. 2001.

[12] C. Goh and A. Baldwin. Towards a more complete model of role, 1998.

[13] P. Goldsack and T. Rush. Specifying an electronic mail system with hp-sl. *Hewlett Packard Journal*, 1991.

[14] HP. *HP-UX 11i Security Containment Administrator's Guide*. Hewlett-Packard Development Company L.P., ¡http://docs.hp.com/en/5991-1821/index.html¿, 2005. [Online; accessed August-2006].

[15] "JLog". Jlog - prolog in java, 2006.

[16] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. Softudc: A software-based data center for utility computing. *Computer*, 37(11):38–46, 2004.

[17] V. LLoyd. *Planning to implement service management (IT Infrastructure library)*. HM Stationary Office, 2000.

[18] N. Murison and A. Baldwin. Secure distributed audit for shared customer environments. Technical report, Hewlett Packard Labs, 2006.

[19] S. Pearson, editor. *Trusted Computing Platforms: TCPA technology in context*. HP Books, Prentice Hall, 2002.

[20] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[21] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994.

[22] L. Sterling and E. Shapiro. *The Art of Prolog, Advanced Programming Techniques*. The MIT Press, Cambridge, Massachusetts, 1986.

[23] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37:62–67, 2004.