A Security Model for Multi-Agent Systems

Ben Mankin
Department of Computer Science
University of Bath
Bath, BA2 7AY, United Kingdom
ben.mankin@anarres.org

Julian Padget
Department of Computer Science
University of Bath
Bath, BA2 7AY, United Kingdom
jap@cs.bath.ac.uk

Abstract

Multi-agent systems are set to become one of the largest classes of distributed dynamic computations – but only if adequate security can be provided. Without a correct security mechanism, systems suffer from fundamental insecurities – reflected in limited host privileges, a lack of user trust and severe administrative constraints limiting utility.

By considering both the historical contributions of computer security and the needs of agent-based systems, we present requirements for distributed security which emphasize the importance and the separation of identification of responsibility from transmission of rights. Existing security models are evaluated against these requirements and generally found wanting. Consequently we propose a surprisingly simple but demonstrably powerful security model for multi-agent systems in which fine-grained trust contexts (frames) are generated automatically; which supports the transmission of rights, and hence delegation of responsibility; and which supports accountability and auditing – but which is also provably correct in polynomial time. We illustrate this scheme by means of several typical agent security scenarios, highlighting how current models fall short.

1. Background and Related Work

The purpose of a security model is to control access to objects. Intercepting access requests using a reference monitor is a solved problem. However, this reference monitor must implement a mechanism for identifying which principal made the request, and deciding whether that principal may have (a particular type of) access to the protected object. Neither of these problems is trivial, as evidenced by the current lack of correct implementations.

In addition to reference monitoring, we also assume the existence of certain cryptographic primitives. We introduce the narrative with some definitions in figure 1.

- **Principal**: The entity in a computer system to which privileges are granted.
- **Authentication**: The process of identifying a principal.
- **Authorisation**: The process of granting or denying access to a protected object.
- **Privilege**: A structure (s, o, r) where s and o are principals and $r \in R$, used for authorisation.

Figure 1. Glossary of key terms

1.1. A Brief History of Security

In 1975, Saltzer and Schroeder said in [1] "*Our definition of protection, [. . .] excludes those features usable only for mistake prevention [. . .]. These may provide just enough control that any breach of control is likely to be the result of a deliberate act rather than an accident. [. . .] It would be a mistake to claim that such systems provide any security.*"

The history of protection mechanisms is inseparable from that of resource sharing, from the first time sharing systems to modern virtualisation systems. M.I.T.'s Compatible Time Sharing System ([2]) and Digital Equipment Corporation's DECSystem/10 ([3]) had what [1] described as "*conceptually straightforward*" but "*surprisingly intricate*" mechanisms for controlling the sharing of information between users. However, mechanisms for information sharing in modern systems have not developed significantly beyond these extremely limited models. Many of the current generation of security systems are still based on these aftermarket additions to time sharing systems; they attempt to compensate for poor design with excessive restriction on the user, and they are barely capable of providing security in new arenas.

1.2. The Coming of Agents

Today, richer content and more complex content delivery systems are being developed. Such technologies allow the web to be used as a platform for delivering up-to-the-minute applications from the developer to the user. These technologies include Netscape's JavaScript ([4], [5]) Sun Microsystems' Java ([6]) Macromedia's Flash ([7]) and Microsoft's ActiveX ([8]). Such systems require the sharing of code as well as static data, and sharing must be possible not only within a single computer system or administrative unit but across network and administrative boundaries. With the introduction of mobile code, the concept of the 'isolated' system has gone out of the window, and we need a new model, based on the fundamental requirements of security, not just timesharing on a single host. We identify some of those requirements here.

Information and code must be shared across networks of systems without any common administrative control. Most computer systems are trusted environments, with one owner or administrative body overseeing all operations within the system. Processes trust system services under the assumption that all such services were provided by that administrator and are not malicious. With the advent of mobile code, agents and especially worms, this assumption becomes false, and it is computationally impossible ([9]) to prove that no code within a system has malicious intent. Therefore, any new design for a security system must account for possible malice in every element of a computation.

Control of the protection system must be delegated. Current protection systems are paranoid,[1] and tend to disallow almost entirely the reassignment of permissions by regular users. However, an increasing use of computers creates increasing administrative load, and it rapidly becomes impractical for any one administrator, or even group of administrators, to perform every reassignment of permissions required in day-to-day operations. It is important that any new security model must permit the modification of privileges by all users *if and only if* this modification would not lead to a violation of security policy in the future. Delegation, and hence reassignment of permission, is a common trait of agent solutions, so transmission of rights is an essential operation, but it must also be correct.

Security must be correct. Systems builders have wanted mathematically correct security for as long as there have been crackers. However, while there has been much evolution of security systems from the original mistake-prevention systems described in section 1.1 to the systems of the current day, provable correctness has never yet been a *fundamental requirement of the design*. A new generation of protection systems should be able to prove correctness of the system with respect to policy at *any time*. The model described here responds to the needs of agent systems for anytime/anywhere authorisation through the parallel maintenance of security metadata, which enables rapid checking.

2. Identification of the Current Principal

The designer of a protection system will frequently intend that a principal p be allowed to perform an operation but that some other principal, say p', be prevented from performing the same operation. The principal "*performing*" the operation is the "*current principal*", and authorisation decisions must depend on this current principal, otherwise all decisions will either "*allow access to everybody*" or "*deny access to everybody*". A protection system is therefore required to identify the current principal and perform authorisation with respect to the permissions of that current principal ([11]).

Throughout the security literature, it is assumed that we have some method of identifying the current principal. Yet the ability to answer this question correctly is frequently under-emphasized in matters of security: [12] and following work in [13, 14, 15] went so far as to ignore it entirely. It has been researched in significant detail only in recent times, as in [16, 17, 18, 19, 11], and the relationship between the identification of the current principal and the dynamics of rights is only now becoming understood ([11]).

2.1. Principals and Responsibility

The defining characteristic of agent systems is that of cooperation: two or more agents or principals cooperating to achieve a task. However, our traditional, isolationist view of security has permitted us to take an atomic view of principals, and we consider the identification of the current principal to be trivial.

In a simple world, the current principal is that acting directly upon a protected object: the '*immediate principal*'. It is tacitly assumed that no other principal is involved in this operation; this atomic principal is acting only on its own behalf and is therefore entirely responsible for the outcome of its actions. This is not usually the case: it is far more common that principals will act jointly, and the current and immediate principals will be distinct.

A more complex representation of a principal is now required. A principal holds a set of permissions, and is optionally identified by a name. From a theoretical perspective, the permissions are all that matters, and a principal may be defined to be a set of permissions.

Cooperation is not necessarily benevolent. The user constructing a buffer overrun in a `root`-owned process is cooperating, in the sense of "*working together*", with `root`, but they are very likely not working for the greater good of `root`. A cooperation is what is formed when any two principals interact in a computation: when data from one process or system is transmitted or returned to another; when one process invokes another; when one process modifies the code or data used by another; these are all examples of cooperation and joint responsibility.

Consider the permissions of a cooperation C between the principals `Alice` and `Bob`. If there is a permission p such that $p \in C$ but $p \notin$ `Alice`, then `Alice` may be able to influence the computation such that a operation requiring the permission p is performed. In other words, `Alice` has gained access to p by cooperating with `Bob`. This is a violation of the "*Principle of Attenuation of Privileges*" ([20]), a fundamental tenet of computer security, which states "*Privileges should not be allowed to grow when they are transported from one place in the system to another.*" Therefore $p \notin C$ and the permissions of a cooperation are those in

1 Paranoid, *adj*: Relating to or affected by paranoia: a strong, usually irrational, feeling [. . .], resulting in a tendency to be suspicious and distrustful, and to become increasingly isolated ([10]).

Basic Ctx Rator	$\dfrac{e_1 \quad \to^S \quad e_1'}{e_1 e_2 \quad \to^S \quad e_1' e_2}$
Basic Fail Rator	$\text{fail}\, e \to^S \text{fail}$
Data Ctx Rand	$\dfrac{e_2 \quad \to^S \quad e_2'}{w_1 e_2 \quad \to^S \quad w_1 e_2'}$
Data Ctx Frame	$\dfrac{e \quad \to^R \quad e'}{R[e] \quad \to^S \quad R[e']}$
Data Ctx Untaint	$\dfrac{e \quad \to^S \quad e'}{\text{untaint } R \text{ in } e \quad \to^S \quad \text{untaint } R \text{ in } e'}$
Data Red Untaint Frame	$\text{untaint } R \text{ in } P[w] \to^S (P \cup (R \cap S))[\text{untaint } R \text{ in } w]$
Data Red Untaint Value	$\text{untaint } R \text{ in } v \to^S v$
Data Red Frame Rator	$R[w_1]w_2 \to^S R[w_1 S[w_2]]$
Data Red Frame Rand	$(\lambda_R x.e)P[w] \to^S \begin{cases} (\lambda_R x.e[x := P[x]])w & \text{if} \quad R \subseteq P = \text{true} \\ \text{fail} & \text{if} \quad R \subseteq P = \text{false} \\ \text{(not reducible)} & \text{if} \quad R \subseteq P = \mho \end{cases}$
Data Red Appl	$(\lambda_R x.e)v \to^S \begin{cases} e[x := S[v]] & \text{if} \quad R \subseteq S = \text{true} \\ \text{fail} & \text{if} \quad R \subseteq S = \text{false} \\ \text{(not reducible)} & \text{if} \quad R \subseteq S = \mho \end{cases}$

Figure 2. A Calculus of Data Inspection

the *intersection* of the cooperating principals. The cooperative unit may perform only those operations which would be permitted to each and every cooperating principal. Thus we may synthesize non-atomic principals.

It may not be necessary to compute the entire current principal in order to make an authorisation decision. It is usually possible to identify that the current principal must have a given privilege without totally evaluating the principal, and thus save time and effort in the protection system. We will achieve this using '*partial principals*': principals about which only partial information is available. If a (full) principal is a map from the set of all permissions $\mathbb{P}$ to $\{\text{true}, \text{false}\}$, then a partial principal is a map from $\mathbb{P}$ to $\{\text{true}, \mho, \text{false}\}$ where the special value $\mho$ represents lack of knowledge, and is an indication that we must appeal to a wider context to collect more information about the current principal before proceeding. With this extension to [17], described in detail in [11] and assumed here, it is possible to accurately model the implementations of stack security in the JVM, as described in [21].

2.2. A Formal Introduction

Variants of the lambda calculus have been developed to maintain information about responsibility at every point of a computation. The most notable recent contributions have been the λ_{sec}-calculi presented by [17] and [16]. These were designed to represent the stack inspection security model as used by Java ([21]) and .NET ([22]).

These calculi introduce a new operator to the λ-calculus: the *framing* operator. A framed expression $R[e]$ is an expression e for which a principal R is responsible, where $R \subseteq \mathbb{P}$ is a full principal. The traditional λ-calculus operations may be redefined within the dynamic context of a frame given rules for the manipulation of frames alongside the computation. Where the traditional λ-calculus might express the rule $e \to e'$, a λ_{sec}-calculus will express a rule $R[e] \to R[e']$, or more generally, $e \to^R e'$, where the annotation on the arrow indicates that the reduction $e \to e'$ is being performed in the immediate context of a frame R, the immediate principal.

The context of reduction becomes important when a security check is to be performed. For example, in the stack inspection calculus of [17] a security check is denoted '*check R in e*', where the dynamic context surrounding this expression must allow all the permissions in R for the reduction of e to continue. If the permissions are not present in the context, then reduction fails; this failure may be modeled by reduction to a distinguished *fail* token, or by throwing an exception.

2.3. A New Calculus for Security

The λ_{sec}-calculus is a powerful tool for augmenting the λ-calculus, but it in no way mandates the stack inspection mechanism described in [17]. In fact, we note several weaknesses: (i) It contains a rule "*Red Frame*": $R[o] \to^S o$ to discard frames from a '*fully reduced*' outcome. This rule explicitly discards information about responsibility for a value in the computation. If we are to consider in a security check every party responsible for influencing the outcome of a computation, then we must be able to evaluate principals passing arguments – that is, values arising from other computations – as well as those in the dynamic context. (ii) [17] also describes an '*outcome*' as the normal form of a lambda term, yet this normal form is not described in the small step reduction system. (iii) Further, the system implicitly uses applicative order reduction, and will fail in normal order reduction for a term like

$$S[P[\lambda x.x](\text{check } S \text{ in } 1)]$$

We have designed a new λ_{sec}-calculus which correctly implements our concept of responsibility. It is described in small step reduction rules in figure 2. It implements a data inspection calculus which is related ([11]) to the stack inspection calculus of [17] and [16], but unlike these earlier works, our calculus produces a unique outcome independent of the order of evaluation. We use the *framing* operator, which we describe as a denotation of responsibility for the enclosed expression. However, instead of introducing a new *check* operator, we modify the λ operator to contain a guard: $(\lambda_R x.e)P[v]$ may be reduced to $e[x := P[v]]$ only if $R \subseteq P$. Thus we may make security checks on all values as well as the dynamic context. In [11], we show that this modification allows us to unify the stack models described in [17] and [16] with the data inspection models of systems such as Perl ([23]).

We now discuss some of the rules from our calculus:

- **Basic Ctx Rator** and **Basic Fail Rator** are immediate from the traditional λ-calculus.
- **Data Ctx Rand** is an extension to the traditional lambda calculus allowing the reduction of the operand of a framed value.
- **Data Ctx Frame** shows that a change of frame changes the context of execution by changing the immediate principal.
- **Data Red Untaint Frame** is an indemnification by the immediate principal S of a value w, which has presumably been verified '*trustworthy*' in some sense by the immediate principal. This rule is the step of an induction over the number of frames around the tainted value. **Data Red Untaint Value** is the base case of this induction.
- **Data Red Frame Rator** is perhaps the least intuitive of our rules. It must maintain a frame from each principal which has manipulated any value. Alternatives to this rule are discussed in [11].
- **Data Red Frame Rand** checks frames on the operand before performing a guarded substitution. This rule is the step of another induction over frames, of which **Data Red Appl** is the base case.
- **Data Red Appl** holds the immediate principal responsible for any unframed expression which may be substituted into a guarded expression.

Rules *Data Red Frame Rator* and *Data Red Appl* between them ensure that any term retains a frame from the immediate principal responsible for it when it is exported from the original frame. These frames are preserved by *Red Frame Rand*. In this way, correctness of the frames on any expression is guaranteed since any expression will at all times have a frame around it representing each principal by which it has been framed, and no principal may manipulate any expression which it does not frame.

As we outline later in section 4, the principles embodied in these reduction rules can readily be incorporated into languages or systems to provide dynamic authorisation of the agent carrying out an action (the current principal).

3. Manipulation of Rights

Many systems may reasonably use a protection mechanism within which the assignments of permissions do not change. However, such a static protection system is not practical for dynamic agent environments. Requirements change. New agents are introduced and old agents removed. Whole new networks may be formed. Any change in the structure of the system may require a change in the configuration of the protection mechanism in order that it may continue to discriminate between desirable and undesirable access.

In order to make the system dynamic, we must extend security policy to govern the creation, reassignment and destruction of privileges, and we must then develop a mechanism for implementing this policy.

3.1. The Safety Problem

Harrison, Ruzzo and Ullman showed in [12] that the general problem of deciding whether a dynamic protection system conforms to a security policy is undecidable. This problem is called the "*General Safety Problem*". The basic problem is described informally as follows:

A dynamic protection system consists of a set of rules for the modification of the protection system and an initial configuration for that system. The system computes by successive application of rules which modify the configuration. In parallel with the halting problem for Turing machines, there are configurations of the system which are reachable and configurations which are not reachable. Again in parallel with the halting problem, it is undecidable whether a particular configuration is reachable ([12]). More generally, it is undecidable whether any particular privilege may ever be created within the system.

A security policy is defined to be a set of predicates which must be shown to hold for all reachable configurations of a protection system. By the described result, it is impossible to state whether a given protection system complies with a given security policy.

Much work followed [12], including [13, 14, 15]; papers containing positive results include [24, 25]; but all the systems studied were either undecidable or oversimplified, and hence unrelated to any real world systems.

In [11], we renew the study of compliance with a security policy, following the approach of [12, 13] by making restrictions of the formal generalised definition of a protection system to identify a class of protection systems for which compliance with policy is always decidable, and discovered many such classes. However, in order to avoid the trap of studying only trivial systems or systems unrelated to the real world, we also developed new metrics of protection systems such that they could be compared and described quantitatively. One result of this work is a set of '*ideal protection systems*', of which we describe one here.

3.2. A New Model

Decidability of the safety problem is a laudable theoretical goal, and any practical system must be decidable in low order polynomial time. However, the decidability of safety is not the only important goal for any truly practical protection system. In order to be practical, a protection system must satisfy many other requirements, which include:

- Low computational cost for the system.
- Low computational cost for the system safety problem.
- Expressiveness of reachable configurations.
- Administrative flexibility and convenience.
- Secure delegation of security system administration.
- Human-oriented justification of all access decisions.
- Availability of accounting and auditing information.

We have discovered several '*ideal systems*' matching all of these requirements, and we describe one of those, the '*Level 2 Transitive Model*' ([11]) here.

Our ideal systems may most easily be described in the Access Matrix formalism of [12]. The Access Matrix is a matrix indexed by principals and objects, where each cell $[s, o]$ contains the rights held by the index principal s over the index object o. The existence of a privilege (s, o, r) is denoted $r \in [s, o]$. For example, the privilege $(\texttt{root}, \texttt{/etc/passwd}, \texttt{write})$, would be represented in the matrix by $\texttt{write} \in [\texttt{root}, \texttt{/etc/passwd}]$.

This particular ideal system contains two rights, an access right r and a '*grant*' right g. A principal is allowed to access an object if the privilege (s, o, r) exists; that is, if $r \in [s, o]$ (see Figure 3).

However, the access relation expressed by r is transitive. That is, if $r \in [s, x]$ and $r \in [x, o]$, then we are permitted to create a privilege $r \in [s, o]$ (see Figure 3).

The remaining operation in our system is the grant operation. While this operation is traditionally implemented as "*If* $g \in [s, o]$ *then grant* $r \in [t, o]$", this approach has considerable computational weaknesses which we do not have space to discuss here. We instead identify a '*grant role*' x for each object o, and identify this by granting $g \in [o, x]$ (see Figure 3).

The safety problem for this system is identical to the authorisation problem: a graph walk must be performed in either case. While our formalisation above requires that a transitive access relation be 1 or more arcs, in practice we make this relation reflexive, thus each principal also has access to itself. We will assume this henceforth.

Many further computational results regarding this system are given in [11], including those mentioned in the introduction to this subsection, but it suffices to mention here only the consequences affecting the implementation of our new agent security framework.

3.3. The Privilege Lattice

There is a surprising duality between the mechanism for the identification of the current principal, as described in section 2.3, and the mechanism for the manipulation of rights, as described in section 3.2. The main point of note is that in both cases, the principals form a lattice. In the case of the λ_{sec}-calculus, the principals are nodes of the subset lattice. In the case of the transitive model, principals have no structure, but lattice meet and join relations may easily be imposed upon the directed graph formed by the relation r. The operations performed on principals and privileges in either case are also unifiable: The subset or membership operator of the λ_{sec}-calculus is semantically identical to the transitive relation expressed by r in our dynamic model.

We unify these two models by constructing an artificial lattice of principals which is given semantics such that we may express both the λ_{sec}-calculus and the dynamic model simultaneously within it. Principals may be related by the rights r and g as in section 3.2, but the hierarchical nature of the lattice introduces an automatic extra r relation: any parent node has access to the sub-node. Additional privileges may be granted to any principal, extending the lattice into a general directed graph.

Principals within this lattice are given names using a hierarchical naming structure similar to the Domain Name System ([26]). Thus, an example principal might be `uk.ac.bath.cs.alis`, a principal which implicitly has all permissions within the @lis project, or `!uk.ac.bath`, the join of all principals below `uk.ac.bath`. See section 4 for an example. This artificial lattice representation of principals has many advantages, including:

- **Principals need not be countable**: There is never a need to enumerate principals.
- **Privileges need not be countable**: A privilege is a Cartesian product of principals.
- **Principals are easily serialisable as metadata**: A principal is a name by which associated metadata may be referenced.
- **Distribution is easy**: Portions of the lattice may be delegated and re-delegated like DNS.
- **The system is computationally simple**: Meet and join may be computed in constant time; subset, membership, access relation and safety may all be computed in linear time ([11]).

3.4. Implications and Costs

Any practical implementation of the λ_{sec}-calculus from section 2.3 must maintain a *frame* as metadata on every piece of code or data in a system. Such a frame would indicate responsibility of a particular principal for the framed code or data and would be manipulated by the mutator as described in our λ_{sec}-calculus.

The major failure of all existing models is that the metadata is too readily discarded. Currently, when data is written to a file and reloaded, or transmitted over a network, frame information is lost, and any later permissions check will not be able to take into account these principals.

It is likely that current systems discard metadata because the underlying systems or protocols cannot support

command $\mathrm{access}(O)$
    let $s = \mathbf{current\_principal}$
    if
        $r \in [s, O]$
    then
        allow_access
end

command $\mathrm{infer}(S, X, O)$
    if
        $r \in [S, X]$
        $r \in [X, O]$
    then
        enter r **into** $[S, O]$
end

command $\mathrm{grant}(T, X, O)$
    let $s = \mathbf{current\_principal}$
    if
        $r \in [s, X]$
        $g \in [O, X]$
    then
        enter r **into** $[T, O]$
end

Figure 3. The access, infer and grant commands

the richer semantics required. There are issues of compatibility with legacy applications, and there is scope for '*cheating*' or lying, especially in the case of transmissions over a network or between administrative units. The legacy can always be supported, but the possibility of cheating requires discussion.

There are only three possible '*cheats*' which might modify frames as they are transmitted over a network. However, if we assume the existence of a cryptographic signature operation, two of them are shown to be impossible, and the third is required for the normal operation of the system:

- A principal does not frame data which it has modified. Any principal receiving this data knows the identify of the sender, and will notice the absence of the sender's frame.
- A principal removes a third principal's frame from data. This is allowed, and is the '*untaint*' operation. If the principal does this 'wrong', then other principals' policies will dictate that the principal is no longer trusted. This strategy represents a long term loss for the first principal, while never allowing it to raise its own privilege. However, this also allows for privacy to be created within a system which could otherwise expose too much metadata about a computation.
- A principal adds a third principal's frame, when that principal is not a part of the computation. This is impossible because frames will be cryptographically protected such that they may only be generated by the responsible principal.

Therefore the metadata is effectively inviolate.

4. Scenarios and Implementation

We assume the existence of participants `Alice`, `Bob`, `Carol` and `Trent`, with privileges A, B, C and T respectively.

Scenario 1: Computation on a Remote System. An agent `Alice` moves to a host `Trent` to perform computation. All code and data associated within `Alice` is framed by a principal `Alice`; thus the results of any computation performed on host `Trent` would be framed by principals `Alice`, `Trent` and those principals providing any additional data to the computation.

There is no concept of a 'trusted' platform; such trust is infeasible. If `Trent` is able to influence the computation of `Alice`, then he must be considered responsible for the result.

Usually, programs are only run on computers within the organisation which constructed the program. In this case, $A \subseteq T$, and $A \cap T = A$, thus the influence of T is nil. This is the case which we are accustomed to seeing in the isolationist world. Many mistakes are made attempting to emulate this nil behaviour with untrusted hosts.

Scenario 2: Transmission of Rights. $g \in [\texttt{Alice}, o]$**, thus `Alice` may grant the privilege** $r \in [\texttt{Bob}, o]$**.** Any computation performed by `Bob` will now have access to the privilege $(\texttt{Bob}, o, r)$ as long as this computation does not reference data or code which itself does *not* hold that privilege. If `Bob` requests access to o, o will ask `Bob` (and subsequently verify with o) whether access was granted to `Bob` before allowing access.

If `Bob` has access to `Carol`, and `Alice` grants $r \in [\texttt{Carol}, o]$, then `Bob` may still access o: o will search the hierarchy for all privileges accessible to `Bob`, identifying in this search `Carol` and thus o. `Carol` will verify that `Bob` indeed has access to `Carol`, and o can verify that `Carol` has access to o.

Scenario 3: Sharing Data. `Alice` wishes to grant access to an object `Alice.obj` to all sub-principals of `Alice`. `Alice` grants the privilege $r \in [!\texttt{Alice}, \texttt{Alice.obj}]$. Since `!Alice` is the join of all principals below `Alice`, all of these principals will immediately have access to `Alice.obj`.

'*Bottom*' privileges form the lower half of the lattice: every principal has a corresponding bottom privilege. They may be used for protecting shared data while leaving it accessible to all elements of a sub-hierarchy, and for accounting, as generic 'responsibility' frames for an entire sub-hierarchy.

5. Related Work

5.1. Role Based Access Control

Role Based Access Control ([27]) introduces a single layer of indirection into the basic Access Matrix model ([12]). It allows the grouping of multiple permissions into a role, and allows the role and associated permissions to be granted *en bloc* to a principal.

This system neither mandates nor suggests any mechanism for the identification of the current principal; it is only directed at solving the problem of the management of rights. While it does provide some administrative convenience in simple systems, it fails miserably to satisfy many of the fundamental requirements to even be considered for analysis: As described by [27], it is not a closed system: there are entities outside any role based access control mechanism which have permission to modify it.

Role based access control also suffers from the same problem of managing grants and revocations as does a trivial privilege management system: If a principal is granted a role for two different reasons and one of those reasons expires, it is not decidable within the system whether the role should be revoked or not; the information is simply not available. In our transitive system, `Alice` would have access to `Bob.r0` and `Bob.r1`, each of which would have access to `Bob.target`. We may now revoke $r \in [\texttt{Alice}, \texttt{Bob.r0}]$ without affecting `Alice`'s access to `Bob.target` via `Bob.r1`. This applies even in the case that `Bob.target` is itself a principal holding further privileges, to any depth of indirection.

5.2. Stack Inspection and Java

Java ([21]) uses a stack inspection model, described in [17], which also represents principals as privilege sets rather than as opaque objects. While this flexibility allows computation with principals, we must still be able to identify all principals involved in a computation, thus to combine their privileges into a current principal. The stack inspection security model is only an approximation of this, focusing entirely on the call chain and neglecting to consider the effect of returned values. Even with this omission, the Java implementation of stack inspection is still a powerful mechanism since the simplicity of the returned values reduces their potential as bearers of malicious intent.

Java is also not a dynamic system: It has no mechanism by which privileges may be granted to principals. The operation termed '*grant*' by [17] is an equivalent of our '*untaint*' operator, and is not related to the '*grant*' operation in a dynamic protection system.

5.3. FIPA Security

FIPA ([28]) is an organisation that defines standards and specifications for agent communication and interaction. Among their specifications are a number of security documents and definitions. [29] begins a description of a domain model which is compatible with some λ_{sec}-calculi.

While the documents output from FIPA acknowledge the existence of some of the problems addressed in this paper, at present no solutions appear to be proposed. Indeed, much of the FIPA documents focus on protocols, the structure of messages and how various fields are encrypted, but without having an underlying security model to provide requirements for the protocol it is extremely hard both to define a protocol and to decide whether it is useful or not.

5.4. Data Inspection and Capabilities

There are data inspection systems in the wild. Currently, the most successful is Perl ([23]), which implements an extremely simplistic data inspection mechanism with only one privilege and two principals. The objective of the Perl model is to identify whether untrusted data participates in a call to any trusted operation within a single Perl process. It is quite successful in this limited objective, but makes no claims to any form of privilege management system.

Capability systems ([30, 31, 32]) are very closely related to data inspection systems, but as of the present day, none are known to be in mainstream use.

6. Conclusions

We have outlined the design and construction of a correct, practical protection system which can be shown to conform to a security policy. A prototype of our privilege management system has been in successful operation in a Unix-like system for four years, where it supports the distributed and delegated management of a Multi User Dungeon system.

The key to the implementation of the calculus is the maintenance of security metadata within any computation. This metadata is external to the computation, just as the frames in the λ_{sec}-calculus above are external to the underlying λ-calculus. It must be maintained alongside the computation throughout the lifetime of any code or data.

The only requirement for correctness is that the implementation have knowledge of all participants in a computation and be permitted to combine metadata from the inputs to frame the output. The definition of a computation may be as low level as a single basic operation on a microprocessor, or may be as high as the the workings of an entire computer or organisation.

Two approaches to implementation are:

1. As an add-on library for any given target language. This is relatively non-intrusive (for client code) and provides the benefits of correct security for the agent network as a whole. This implementation has the lowest initial cost, and constitutes a nice proof of concept for a particular application field. In conventional form this approach imposes a burden on the programmer to ensure that all incoming and outgoing data is passed through the security library. However we look forward to incorporating this into the HARMON*IA* platform ([33, 34]), in which case the security checking can be transparently integrated into the communication framework and need not bother the agent programmer at all.
2. A more pervasive solution, in line with the Aroma virtual machine [35], is to implement security as low as the virtual machine level, in which case no modification of existing agent source code would be required: The virtual machine would have access to all incoming and outgoing data and would be able to manipulate frames at a very fine grained level within the mutator. Any program running in the modified virtual machine would benefit from the additional security arrangements.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[2] P. A. Crisman, ed., *The Compatible Time-Sharing System*. Cambridge, Mass.: M.I.T. Press, 1965.

[3] D. Stone, "Pdp-10 system concepts and capabilities," *PDP-10 Applications in Science*, vol. II, pp. 32–55, undated (ca. 1970). Digital Equipment Corporation's DECsystem/10.

[4] Netscape Communications Corporation, *Client Side JavaScript Guide for version 1.3*. 466 Ellis Street, Mountain View, CA 94043-4042: Netscape Communications Corporation, 1999. Available at `http://devedge.netscape.com/library/manuals/2000/javascript/1.3/guide/` Notes on security have mysteriously vanished from the documentation for later versions 1.4 and 1.5.

[5] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly and Associates, Inc, 1998.

[6] Sun Microsystems, "The source for Java Technology," 2003. `http://java.sun.com/`.

[7] Macromedia, "Flash MX 2004," 2003. `http://www.macromedia.com/software/flash/`.

[8] Microsoft, "ActiveX Controls - Microsoft Papers, Presentations, Web Sites and Books," 2003. `http://www.microsoft.com/com/tech/ActiveX.asp`.

[9] M. Minsky, *Computation: Finite and infinite machines*. London: Prentice-Hall, 1972.

[10] Chambers Harrap, *The Chambers 21st Century Dictionary*. Chambers Harrap Publishers Ltd, 2003.

[11] B. Mankin, *The Formalisation of Protection Systems*. PhD thesis, Department of Computer Science, University of Bath, 2004. In preparation.

[12] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, pp. 461–471, Aug. 1976.

[13] M. Harrison and W. Ruzzo, "Monotonic protection systems," in *Foundations of Secure Computation* (R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, eds.), pp. 337–365, New York: Academic Press, 1978.

[14] R. J. Lipton and T. A. Budd, "On classes of protection systems," in *Foundations of Secure Computation* (R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, eds.), (New York), pp. 281–296, Academic Press, 1978.

[15] T. A. Budd, "Safety in grammatical protection systems," *International Journal of Computer and Information Sciences*, vol. 12, pp. 413–431, Dec. 1983.

[16] F. Pottier, C. Skalka, and S. Smith, "A systematic approach to static access control," in *Proceedings of the 10th European Symposium on Programming (ESOP'01)* (D. Sands, ed.), vol. 2028 of *Lecture Notes in Computer Science*, pp. 30–45, Springer Verlag, Apr. 2001.

[17] C. Fournet and A. D. Gordon, "Stack inspection: theory and variants," *ACM SIGPLAN Notices*, vol. 31, pp. 307–318, Jan. 2002.

[18] P. Ørbaek, "Can you trust your data?," in *TAPSOFT '95: Theory and Practice of Software Development* (P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds.), vol. 915 of *Lecture Notes in Computer Science*, pp. 575–589, Springer-Verlag, 1995.

[19] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, (New York, NY), pp. 147–160, ACM, Jan. 1999.

[20] N. Minsky, "The principle of attenuation of privileges and its ramifications," in *Foundations of Secure Computation* (R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, eds.), (New York), pp. 337–365, Academic Press, 1978.

[21] L. Gong, "The Java Security Architecture," 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.d%oc.html`.

[22] Microsoft, "Microsoft .NET," 2003. `http://www.microsoft.com/net/`.

[23] "Perl programmers reference guide: Perlsec," 2001.

[24] A. K. Jones, R. J. Lipton, and L. Snyder, "A linear time algorithm for deciding security," in *Proc. 17th Annual Symp. on Foundations of Computer Science*, 1976.

[25] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *Journal of the ACM*, vol. 24, pp. 455–464, July 1977.

[26] Network Working Group, "RFC 819: The Domain Naming Convention for Internet User Applications," Aug. 1982.

[27] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, pp. 38–47, Feb. 1996.

[28] The Foundation for Intelligent Physical Agents, "Welcome to the Foundation for Intelligent Physical Agents," Jan. 2004. `http://www.fipa.org/`.

[29] FIPA Architecture TC, "Fipa domains and policies specification," Aug. 2001. `http://www.fipa.org/specs/fipa00089/`.

[30] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pp. 170–185, Kiawah Island Resort, near Charleston, South Carolina, Dec. 1999.

[31] R. M. Needham and R. D. M. Walker, "Protection and process management in the CAP computer," in *Proc. IRIA Internat'l Workshop on Protection in Operating Systems*, (Rocquencourt, France), pp. 155–160, Institute de Recherche d'Informatique et d'Automatique, France, 1974.

[32] R. M. Needham and A. J. Herbert, *The Cambridge CAP Computer and Its Operating System*. Reading, MA: Addison-Wesley, 1982.

[33] D. J. Pastor, A. M. Turu, O. Cliffe, and J. Padget, "HARMON*IA:* from institutional specifications to deployment." submitted for publication.

[34] D. J. Pastor and J. Padget, "Towards HARMONIA: automatic generation of e-organisations from institution specifications," in *Proceedings of Ontologies in Agent Systems at AAMAS03*, 2003. Published electronically at `http://CEUR-WS.org/Vol-73/oas03-jimenez.pdf`.

[35] N. Suri, "State capture and resource control for Java: The design and implementation of the Aroma Virtual Machine," in *Proceedings of the Java Virtual Machine Research and Technology Sy[m]posium (JVM '01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA* (USENIX, ed.), (Berkeley, CA, USA), pp. ??–??, USENIX, 2001.