RheoStat : Real-time Risk Management

Ashish Gehani [*] and Gershon Kedem

Department of Computer Science, Duke University

Abstract. As the frequency of attacks faced by the average host connected to the Internet increases, reliance on manual intervention for response is decreasingly tenable. Operating system and application based mechanisms for automated response are increasingly needed. Existing solutions have either been customized to specific attacks, such as disabling an account after a number of authentication failures, or utilize harsh measures, such as shutting the system down. In contrast, we present a framework for systematic fine grained response that is achieved by dynamically controlling the host's exposure to perceived threats.

This paper introduces a formal model to characterize the risk faced by a host. It also describes how the risk can be managed in real-time by adapting the exposure. This is achieved by modifying the access control subsystem to let the choice of whether to grant a permission be delegated to code that is customized to the specific right. The code can then use the runtime context to make a more informed choice, thereby tightening access to a resource when a threat is detected. The running time can be constrained to provide performance guarantees.

The framework was implemented by modifying the Java Runtime. A suite of vulnerable Jigsaw servlets and corresponding attacks was created. The following were manually added: code for dynamic permission checks; estimates of the reduction in exposure associated with each check; the frequencies with which individual permissions occurred in a typical workload; a global risk tolerance. The resulting platform disrupted the attacks by denying the permissions needed for their completion.

1 Introduction

This paper presents a new method of intrusion prevention. We introduce a mechanism to dynamically alter the exposure of a host to contain an intrusion when it occurs. A host's exposure comprises the set exposures of all its resources. If access to a resource is to be controlled, then a permission check will be present to safeguard it. The set of permissions that are utilized in the process of an intrusion occurring can thus be viewed as the system's exposure to that particular threat.

By performing auxiliary checks prior to granting a permission, the chance of it being granted in the presence of a threat can be reduced. By tightening the access control configuration, the system's exposure can be reduced. By relaxing the configuration, the exposure can be allowed to increase. The use of auxiliary checks will introduce runtime overhead. In addition, when permissions are denied, applications may be prevented

[*] Supported by a USENIX Association Research Grant and a North Carolina Networking Initiative Graduate Fellowship.

from functioning correctly. These two factors require that the use of the auxiliary checks must be minimized.

We first investigate how the auxiliary checks can be performed by modifying the access control subsystem. After that we introduce a model for measuring and managing the risk by dynamically altering the host's exposure. Finally, we demonstrate how the approach can be used to contain attacks in real-time.

2 Predicated Permissions

One approach is to use a subset of the security policy that can be framed intuitively. While this method suffers from the fact that the resulting specification will not be complete, it has the benefit that it is likely to be deployed. The specific subset we consider is that which constitutes the authorization policy. These consist of statements of the form $(\sigma \Rightarrow p)$. Here p is a permission and σ can be any legal statement in the policy, L. If σ holds true, then the permission p can be granted. The reference monitor maintains an access control matrix, M, which represents the space of all combinations of the set of subjects, S, set of objects, O, and the set of authorization types, A.

$$M = S \times O \times A, \quad \text{where} \quad p(i,j,k) \in M \tag{1}$$

Traditionally, the space M is populated with elements of the form:

$$p(i,j,k) = 1 \tag{2}$$

if the subject $S[i]$ should be granted permission $A[k]$ to access object $O[j]$, and otherwise with:

$$p(i,j,k) = 0 \tag{3}$$

In our new paradigm, we can replace the elements of M with ones of the form:

$$p(i,j,k) = \sigma, \quad \text{where} \quad \sigma \in L \tag{4}$$

Thus, a permission check can be the evaluation of a predicate framed in a suitable language, L, which will be required to evaluate to either true or false, corresponding to 1 or 0, instead of being a lookup of a binary value in a static configuration.

3 Active Monitoring

To realize our model of evaluating predicates prior to granting permissions, we augment a conventional access control subsystem by interceding on all permission checks and transferring control to our **ActiveMonitor** as shown in Figure 1. [1] If an appropriate binding exists, it delegates the decision to code customized to the specific right. Such bindings can be dynamically added and removed to the running **ActiveMonitor** through

[1] The impact of interceding alone (without counting the effect of evaluating predicates) does not impact the running time of SPECjvm98 [SPECjvm98] with any statistical significance.

a programming interface. This allows the restrictiveness of the system's access control configuration to be continuously varied in response to changes in the threat level.

Our prototype was created by modifying the runtime environment of Sun's Java Development Kit (JDK 1.4), which runs on the included stack-based virtual machine. The runtime includes a reference monitor, called the **AccessController**, which we altered as described below.

3.1 Interposition

When an application is executed, each method that is invoked causes a new frame to be pushed onto the stack. Each frame has its own access control context that encapsulates the permissions granted to it. When access to a controlled resource is made, the call through which it is made invokes the **AccessController**'s *checkPermission()* method. This inspects the stack and checks if any of the frames' access control contexts contain permissions that would allow the access to be made. If it finds an appropriate permission it returns silently. Otherwise it throws an exception of type **AccessControlException**. See [Koved98] for details.

We altered the *checkPermission()* method so it first calls the active **ActiveMonitor**'s *checkPermission()* method. If it returns with a `null` value, the **AccessController**'s *checkPermission()* logic executes and completes as it would have without modification. Otherwise, the return value is used to throw a customized subclass of **AccessControlException** which includes information about the reason why the permission was denied. Thus, the addition of the **ActiveMonitor** functionality can restrict the permissions, but it can not cause new permissions to be granted. Note that it is necessary to invoke the **ActiveMonitor**'s *checkPermission()* first since the side-effect of invoking this method may be the initiation of an exposure-reducing response. If it was invoked after the **AccessController**'s *checkPermission()*, then in the cases that an **AccessControlException** was thrown, control would not flow to the Active Monitor's *checkPermission()* leaving any side-effect responses uninitiated.

Code that is invoked by the **ActiveMonitor** should not itself cause new **ActiveMonitor** calls, since this could result in a recursive loop. To avoid this, before the **ActiveMonitor**'s *checkPermission()* method is invoked, the stack is traversed to ensure that none of the frames is an **ActiveMonitor** frame, since that would imply that the current thread belonged to code invoked by the **ActiveMonitor**. If an **ActiveMonitor** frame is found, the **AccessController**'s *checkPermission()* returns silently, that is it grants the permission with no further checks.

3.2 Invocation

When the system initializes, the **ActiveMonitor** first creates a hash table which maps permissions to predicates. It populates this by loading the relevant classes, using Java *Reflection* to obtain appropriate constructors and storing them for subsequent invocation. At this point it is ready to accept delegations from the **AccessController**.

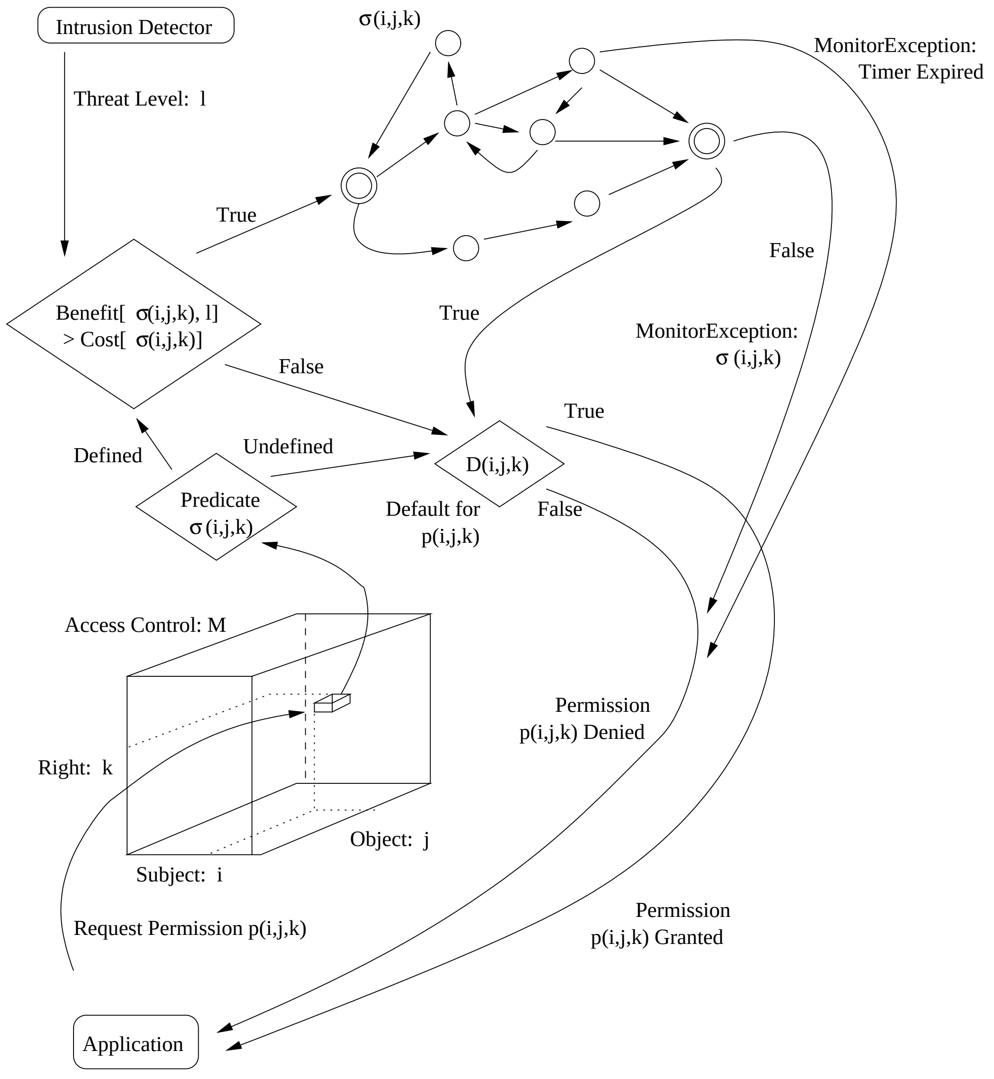

Fig. 1. Static permission lookups are augmented using an **ActiveMonitor** which facilitates the use of runtime context in deciding whether to grant a permission. **ActiveMonitor** predicated permissions have 3 distinguishing features: (i) Constant running time, (ii) Dynamic activation if expected benefit exceeds cost, (iii) Interrogatable for cause of denial.

```
public abstract class PredicateThread extends Thread{

    protected PredicateThread(Permission permission,
                                                Object lock);
    public void run(){

        if(condition) result=true;

        synchronized(lock){
            lock.notify();
        }
    }

    public boolean getResult();
}
```

Fig. 2. Skeletal version of **PredicateThread**

When the **ActiveMonitor**'s *checkPermsission()* method is invoked, it uses the permission passed as a parameter to perform a lookup and extract any code associated with the permission. If code is found, it is invoked in a new thread and a timer is started. Otherwise, the method returns `null`, indicating the **AccessController** use the static configuration decide if the permission should be granted. The code must be a subclass of the abstract class **PredicateThread**. A skeletal version is presented in Figure 2. This ensures that it will store the result in a shared location when the thread completes and notify the **ActiveMonitor** of its completion via a shared synchronization lock.

The shared location is inspected when the timer expires. If the code that was run evaluated to `true`, then a `null` is returned by the **ActiveMonitor**'s *checkPermission()* method. Otherwise a string describing the cause of the permission denial is returned. If the code had not finished executing when the timer expired, a string denoting this is returned. As described above, when a string is returned, it is used by the modified **AccessController** to throw an **ActiveMonitorException**, our customized subclass of **AccessControlException**, which includes information about the predicate that failed. The thread forked to evaluate code can be destroyed once its timer expires. Care must be taken when designing predicates so that their destruction midway through an evaluation does not affect subsequent evaluations.

Finally, the **ActiveMonitor**'s own configuration can be dynamically altered. It exposes *enableSafeguard()* and *disableSafeguard()* methods for this. These can be used to activate and deactivate the utilization of the auxiliary checks for a specific permission. If a piece of code is being evaluated prior to granting a particular permission and there is no longer any need for this to occur, it can be deactivated with the *disableSafeguard()* method. Subsequently that permission will be granted using only the **AccessController**'s static configuration using a lookup of a binary value. Similarly, if it is deemed necessary to perform extra checks prior to granting a permission, this may be enabled by invoking the *enableSafeguard()* method.

4 Risk

Given the ability to predicate permissions the successful verification of auxiliary conditions, we now consider the problem of how to choose when to use such safeguards.

The primary goal of an intrusion response system is to guard against attacks. However, invoking responses arbitrarily may safeguard part of the system but leave other weaker areas exposed. Thus, to effect a rational response, it is necessary to weigh all the possible alternatives. A course of action must then be chosen which will result in the least damage, while simultaneously assuring that cost constraints are respected. Risk management addresses this problem.

4.1 Risk Factors

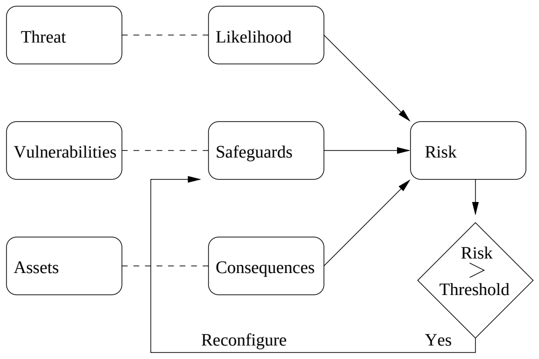

Fig. 3. Risk can be analyzed as a function of the threats, their likelihood, the vulnerabilities, the safeguards, the assets and the consequences. Risk can be managed by using safeguards to control the exposure of vulnerable resources.

Analyzing the risk of a system requires knowledge of a number of factors. Below we describe each of these factors along with its associated semantics. We define these in the context of the operating system paradigm since our goal is host-based response.

The paradigm assumes the existence of an operating system augmented with an access control subsystem that mediates access by subjects to objects in the system using predicated permissions. In addition, a host-based intrusion detection system is assumed to be present and operational.

Threats A *threat* is an agent that can cause harm to an asset in the system. We define a threat to be a specific attack against any of the application or system software that is running on the host. It is characterized by an intrusion detection signature. The set of threats is denoted by $T = \{t_1, t_2, \ldots\}$, where $t_\alpha \in T$ is an intrusion detection signature. Since t_α is a host-based signature, it is comprised of an *ordered set* of events $S(t_\alpha) = \{s_1, s_2, \ldots\}$. If this set occurs in the order recognized by the rules of the intrusion detector, it signifies the presence of an attack.

Likelihood The *likelihood* of a threat is the hypothetical probability of it occurring. If a signature has been partially matched, the extent of the match serves as a predictor of the chance that it will subsequently be completely matched. A function μ is used to compute the likelihood of threat t_α. μ can be threat specific and will depend on the history of system events that are relevant to the intrusion signature. Thus, if $E = \{e_1, e_2, \ldots\}$ denotes the ordered set of all events that have occurred, then:

$$\mathcal{T}(t_\alpha) = \mu(t_\alpha, E \overset{\prec}{\cap} S(t_\alpha)) \tag{5}$$

where $\overset{\prec}{\cap}$ yields the set of all events that occur *in the same order* in each input set. Our implementation of μ is described in Section 7.1.

Assets An *asset* is an item that has value. We define the assets to be the data stored in the system. In particular, each file is considered a separate object $o_\beta \in O$, where $O = \{o_1, o_2, \ldots\}$ is the set of assets. A set of objects $A(t_\alpha) \subseteq O$ is associated with each threat t_α. Only objects $o_\beta \in A(t_\alpha)$ can be harmed if the attack that is characterized by t_α succeeds.

Consequences A *consequence* is a type of harm that an asset may suffer. Three types of consequences can impact the data. These are the loss of confidentiality, integrity and availability. If an object $o_\beta \in A(t_\alpha)$ is affected by the threat t_α, then the resulting costs due to the loss of confidentiality, integrity and availability are denoted by $c(o_\beta)$, $i(o_\beta)$, and $a(o_\beta)$ respectively. Any of these values may be 0 if the attack can not effect the relevant consequence. However, all three values associated with a single object can not be 0 since in that case $o_\beta \in A(t_\alpha)$ would not hold. Thus, the consequence of a threat t_α is:

$$\mathcal{C}(t_\alpha) = \sum_{o_\beta \in A(t_\alpha)} c(o_\beta) + i(o_\beta) + a(o_\beta) \tag{6}$$

Vulnerabilities A *vulnerability* is a weakness in the system. It results from an error in the design, implementation or configuration of either the operating system or application software. The set of vulnerabilities present in the system is denoted by $W = \{w_1, w_2, \ldots\}$. $W(t_\alpha) \subseteq W$ is the set of weaknesses exploited by the threat t_α to subvert the security policy.

Safeguards A *safeguard* is a mechanism that controls the exposure of the system's assets. The reference monitor's set of permission checks $P = \{p_1, p_2, \ldots\}$ serve as safeguards in an operating system. Since the reference monitor mediates access to all objects, a vulnerability's exposure can be limited by denying the relevant permissions. The set $P(w_\gamma) \subseteq P$ contains all the permissions that are requested in the process of exploiting vulnerability w_γ. The static configuration of a conventional reference monitor either grants or denies access to a permission p_λ. This *exposure* is denoted by $v(p_\lambda)$, with the value being either 0 or 1. The active reference monitor can reduce the exposure of a statically granted permission to $v'(p_\lambda)$, a value in the range $[0, 1]$. This reflects the nuance that results from evaluating predicates as *auxiliary safeguards*.)

Thus, if all auxiliary safeguards are utilized, the total exposure to a threat t_α is:

$$\mathcal{V}(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|} \tag{7}$$

where:

$$\hat{P}(t_\alpha) = \bigcup_{w_\gamma \in W(t_\alpha)} P(w_\gamma) \tag{8}$$

5 Runtime Risk Management

The risk to the host is the sum of the risks that result from each of the threats that it faces. The risk from a single threat is the product of the chance that the attack will occur, the exposure of the system to the attack, and the cost of the consequences of the attack succeeding [NIST800-12]. Thus, the cumulative risk faced by the system is:

$$\mathcal{R} = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}(t_\alpha) \times \mathcal{C}(t_\alpha) \tag{9}$$

If the risk posed to the system is to be managed, the current level must be continuously monitored. When the risk rises past the threshold that the host can tolerate, the system's security must be tightened. Similarly, when the risk decreases, the restrictions can be relaxed to improve performance and usability. This process is elucidated below.

The system's risk can be reduced by reducing the exposure of vulnerabilities. This is is effected through the use of auxiliary safeguards prior granting a permission. Similarly, if the threat recedes, the restrictive permission checks can be relaxed.

5.1 Managed Risk

The set of permissions P is kept partitioned into two disjoint sets, $\Psi(P)$ and $\Omega(P)$, that is $\Psi(P) \cap \Omega(P) = \phi$ and $\Psi(P) \cup \Omega(P) = P$. The set $\Psi(P) \subseteq P$ contains the permissions for which auxiliary safeguards are currently active. The remaining permissions $\Omega(P) \subseteq P$ are handled conventionally by the reference monitor, using only static lookups rather than evaluating associated predicates prior to granting these permissions.

At any given point, when the set of safeguards $\Psi(P)$ is in use, the current risk $\mathcal{R}'$ is calculated with:

$$\mathcal{R}' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}'(t_\alpha) \times \mathcal{C}(t_\alpha) \tag{10}$$

where:

$$\begin{aligned} \mathcal{V}'(t_\alpha) = & \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Omega(P)} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} \\ + & \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Psi(P)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|} \end{aligned} \tag{11}$$

5.2 Risk Tolerance

While the risk must be monitored continuously, there is a computational cost incurred each time it is recalculated. Therefore, the frequency with which the risk is estimated must be minimized to the extent possible. Instead of calculating the risk synchronously at fixed intervals in time, we exploit the fact that the risk level only changes when the threat to the system is altered.

An intrusion detector is assumed to be monitoring the system's activity. Each time it detects an event that changes the extent to which a signature has been matched, it passes the event e to the intrusion response subsystem. The level of risk $\mathcal{R}_b$ before e occurred is noted, and then the level of risk $\mathcal{R}_a$ after e occurred is calculated. Thus,

$\mathcal{R}_a = \mathcal{R}_b + \epsilon$, where ϵ denotes the change in the risk. Since the risk is recalculated only when it actually changes, the computational cost of monitoring it is minimized.

Each time an event e occurs, either the risk decreases, stays the same or increases. Each host is configured to tolerate risk upto a threshold, denoted by $\mathcal{R}_0$. After each event e, the system's response guarantees that the risk will return to a level below this threshold. As a result, $\mathcal{R}_b < \mathcal{R}_0$ always holds. If $\epsilon = 0$, then no further risk management steps are required.

If $\epsilon < 0$, then $\mathcal{R}_a < \mathcal{R}_0$ since $\mathcal{R}_a = \mathcal{R}_b + \epsilon < \mathcal{R}_b < \mathcal{R}_0$. At this point, the system's security configuration is more restrictive than it needs to be. To improve system usability and performance, the response system must deactivate appropriate safeguards, while ensuring that the risk level does not rise past the threshold $\mathcal{R}_0$.

If $\epsilon > 0$ and $\mathcal{R}_a \leq \mathcal{R}_0$, then no action needs to be taken. Even though the risk has increased, it is below the threshold that the system can tolerate, so no further safeguards need to be introduced. In addition, the system will not be able to find any set of unused safeguards whose removal will increase the risk by less than $\mathcal{R}_0 - \mathcal{R}_b - \epsilon$, since the presence of such a combination would also mean that the set existed before e occurred. It is not possible that such a combination of safeguards existed before e occurred since they would also have satisfied the condition of being less than $\mathcal{R}_0 - \mathcal{R}_b$ and would have been utilized before e occurred in the process of minimizing the impact on performance in the previous step.

If $\epsilon > 0$ and $\mathcal{R}_a > \mathcal{R}_0$, then action is required to reduce the risk to a level below the threshold of tolerance. The response system must search for and implement a set of safeguards to this end. Since the severity of the response is dependent on the current risk level, the risk recalculation can not be delayed despite the additional overhead it imposes at a point when the system is already stressed.

5.3 Recalculating Risk

When the risk is calculated the first time, Equation 9 is used. Therefore, the cost is $O(|T| \times |P| \times |O|)$. Since the change in the risk must be repeatedly evaluated during real-time reconfiguration of the runtime environment, it is imperative the cost is minimized. This is achieved by caching all the values $\mathcal{V}'(t_\alpha) \times \mathcal{C}(t_\alpha)$ associated with threats $t_\alpha \in T$ during the evaluation of Equation 9. Subsequently, when an event e occurs, the change in the risk $\epsilon = \delta(\mathcal{R}', e)$ can be calculated with cost $O(|T|)$ as described below.

The ordered set E refers to all the events that have occurred in the system prior to the event e. The change in the likelihood of a threat t_α due to e is:

$$\delta(\mathcal{T}(t_\alpha), e) = \mu(t_\alpha, (E \cup e) \overset{\prec}{\cap} S(t_\alpha)) - \mu(t_\alpha, E \overset{\prec}{\cap} S(t_\alpha)) \quad (12)$$

The set of threats affected by e is denoted by $\Delta(T, e)$. A threat $t_\alpha \in \Delta(T, e)$ is considered to be affected by e if $\delta(\mathcal{T}(t_\alpha), e) \neq 0$, that is its likelihood changed due to the event e. The resultant change in the risk level is:

$$\delta(\mathcal{R}', e) = \sum_{t_\alpha \in \Delta(T,e)} \delta(\mathcal{T}(t_\alpha), e) \times \mathcal{V}'(t_\alpha) \times \mathcal{C}(t_\alpha) \quad (13)$$

6 Cost / Benefit Analysis

After an event e occurs, if the risk level $\mathcal{R}_a$ increases past the threshold of risk tolerance $\mathcal{R}_0$, the goal of the response engine is to reduce the risk by $\delta_g \geq \mathcal{R}_a - \mathcal{R}_0$ to a level below the threshold. To do this, it must select a subset of permissions $\rho(\Omega(P)) \subseteq \Omega(P)$, such that adding the safeguards will reduce the risk to the desired level. By ensuring that the permissions in $\rho(\Omega(P))$ are granted only after relevant predicates are verified, the resulting risk level is reduced to:

$$\mathcal{R}'' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}''(t_\alpha) \times \mathcal{C}(t_\alpha) \tag{14}$$

where the new vulnerability measure, based on Equation 7, is:

$$\begin{aligned}\mathcal{V}''(t_\alpha) = & \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Omega(P) - \rho(\Omega(P)))} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} \\ & + \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Psi(P) \cup \rho(\Omega(P)))} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|}\end{aligned} \tag{15}$$

Instead, after an event e occurs, if the risk level $\mathcal{R}_a$ decreases, the goal of the response engine is to allow the risk to rise by $\delta_g \leq \mathcal{R}_0 - \mathcal{R}_a$ to a level below the threshold of risk tolerance $\mathcal{R}_0$. To do this, it must select a subset of permissions $\rho(\Psi(P)) \subseteq \Psi(P)$, such that removing the safeguards currently in use for the set will yield the maximum improvement to runtime performance. After the safeguards are relaxed, the risk level will rise to:

$$\mathcal{R}'' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}''(t_\alpha) \times \mathcal{C}(t_\alpha) \tag{16}$$

where the new vulnerability measure, based on Equation 7, is:

$$\begin{aligned}\mathcal{V}''(t_\alpha) = & \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Omega(P) \cup \rho(\Psi(P))} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} \\ & + \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Psi(P) - \rho(\Psi(P))} \frac{v(p_\lambda \times v'(p_\lambda))}{|\hat{P}(t_\alpha)|}\end{aligned} \tag{17}$$

There are $O(2^{|P|})$ ways of choosing subsets $\rho(\Omega(P)) \subseteq \Omega(P)$ for risk reduction or subsets $\rho(\Psi(P)) \subseteq \Psi(P)$ for risk relaxation. When selecting from the possibilities, the primary objective is the maintenance of the bound $\mathcal{R}'' < \mathcal{R}_0$, where $\mathcal{R}'' = \mathcal{R}_a - \delta_g$ in the case of risk reduction, and $\mathcal{R}'' = \mathcal{R}_a + \delta_g$ in the case of risk relaxation.

The choice of safeguards also impacts the performance of the system. Evaluating predicates prior to granting permissions introduces latency in system calls. A single interrogation of the runtime, such as checking how much swap space is free, takes about 1ms. When file permission checks were protected with safeguard code that ran for 150ms, 5 of 7 applications in [SPECjvm98] took less than 2% longer to run on average, while the other 2 applications took 37% longer. Hence, the choice of subsets

$\rho(\Omega(P))$ or $\rho(\Psi(P))$ is subject to the secondary goal of minimizing the overhead introduced. (In practice, the cumulative effect is likely to be acceptable since useful predicate functionality can be created with code that runs in just a few milliseconds.)

The adverse impact of a safeguard is proportional to the frequency with which it is utilized in the system's workload. Given a typical workload, we can count the frequency $f(p_\lambda)$ with which permission p_λ is requested in the workload. This can be done for all permissions. The cost of utilizing subset $\rho(\Omega(P))$ for risk reduction can then be calculated with:

$$\zeta(\rho(\Omega(P))) = \sum_{p_\lambda \in \rho(\Omega(P))} f(p_\lambda) \tag{18}$$

Similarly, if the safeguards of subset $\rho(\Psi(P))$ are relaxed, the resulting reduction in runtime cost can be calculated with:

$$\zeta(\rho(\Psi(P))) = \sum_{p_\lambda \in \rho(\Psi(P))} f(p_\lambda) \tag{19}$$

The ideal choice of safeguards will minimize the impact on performance, while simultaneously ensuring that the risk remains below the threshold of tolerance. Thus, for risk reduction we wish to find:

$$\min \zeta(\rho(\Omega(P))), \quad \mathcal{R}'' \leq \mathcal{R}_0 \tag{20}$$

In the context of risk relaxation, we wish to find:

$$\max \zeta(\rho(\Psi(P))), \quad \mathcal{R}'' \leq \mathcal{R}_0 \tag{21}$$

Both these problems are equivalent to the NP-complete *0-1 Knapsack Problem*. Although approximation algorithms exist [Kellerer98], they are not suitable for our use since we need to make a choice in real-time. Instead, we will use a heuristic which guarantees that the risk is maintained below the threshold. The heuristic is based on the greedy algorithm for the *0-1 Knapsack Problem* which picks the item with the highest benefit-to-cost ratio repeatedly till the knapsack's capacity is reached. This yields a solution that is always within a factor of 2 of the optimal choice [Garey79].

6.1 Response Heuristic

When the risk needs to be reduced, the heuristic uses the greedy strategy of picking the response primitive with the highest benefit-to-cost ratio repeatedly till the constraint is satisfied. By maintaining the choices in a *heap* data structure keyed on the benefit-to-cost ratio, each primitive in the response set can be chosen in $O(1)$ time. This is significant since implementing a single response primitive is often sufficient for disrupting an attack in progress. When the risk needs to be relaxed, the active safeguards with the highest cost-to-benefit ratios can be selected since these will be yield the best improvement to system performance. A separate *heap* is utilized to maintain these.

Risk Reduction We outline the algorithm for the case where the risk needs to be reduced. The first two steps constitute pre-processing and therefore only occur during system initialization. Risk relaxation is analogous and therefore not described explicitly.

Step 1 The benefit-to-cost ratio of each candidate safeguard permission $p_\lambda \in \Omega(P)$ can be calculated by:

$$\kappa(p_\lambda) = \frac{\sum_{t_\alpha : p_\lambda \in (\hat{P}(t_\alpha) \cap \Omega(P))} \{\mathcal{T}(t_\alpha) \times \frac{v(p_\lambda) \times (1 - v'(p_\lambda))}{|\hat{P}(t_\alpha)|} \times \mathcal{C}(t_\alpha)\}}{f(p_\lambda)} \tag{22}$$

Step 2 The response set is defined as empty, that is $\rho(\Omega(P)) = \phi$.
Step 3 The single risk reducing measure with the highest benefit-to-cost can be selected, that is:

$$p_{max} = \max\ \kappa(p_\lambda), \quad p_\lambda \in \Omega(P) \tag{23}$$

The permission is added to $\rho(\Omega(P))$.
Step 4 The risk before the candidate responses were utilized is $\mathcal{R}_a$. If the responses were activated the resulting risk $\mathcal{R}''$ is given by:

$$\mathcal{R}'' = \mathcal{R}_a \quad - \sum_{p_\lambda \in \rho(\Omega(P))} \kappa(p_\lambda) \times f(p_\lambda) \tag{24}$$

This is equivalent to using Equations 14 and 15. While the worst case complexity is the same, when few protective measures are added the cost of the above calculation is significantly lower.
Step 5 If $\mathcal{R}'' > \mathcal{R}_0$ then the system repeats the above from Step 3 onwards. If $\mathcal{R}'' \leq \mathcal{R}_0$ then proceed to the next step.
Step 6 The set of safeguards $\rho(\Omega(P))$ must be activated and $\rho(\Omega(P))$ should be transferred from $\Omega(P)$ to $\Psi(P)$.

The time complexity is $O(|\rho(\Omega(P))|)$. In the worst case, this is $O(|\Omega(P)|) \leq O(|P|)$. Unless a large variety of attacks are simultaneously launched against the target, the response set will be small.

Risk Relaxation In the case of risk relaxation, the algorithm becomes:

Step 1 For $p_\lambda \in \Psi(P)$ calculate:

$$\kappa(p_\lambda) = \frac{\sum_{t_\alpha : p_\lambda \in (\hat{P}(t_\alpha) \cap \Psi(P))} \{\mathcal{T}(t_\alpha) \times \frac{v(p_\lambda) \times (1 - v'(p_\lambda))}{|\hat{P}(t_\alpha)|} \times \mathcal{C}(t_\alpha)\}}{f(p_\lambda)} \tag{25}$$

Step 2 Set $\rho(\Psi(P)) = \phi$.
Step 3 Find the safeguard which yields the least risk reduction per instance of use:

$$p_{min} = \min\ \kappa(p_\lambda), \quad p_\lambda \in \Psi(P) \tag{26}$$

Add it to $\rho(\Psi(P))$.
Step 4 Calculate $\mathcal{R}''$:

$$\mathcal{R}'' = \mathcal{R}_a \quad + \sum_{p_\lambda \in \rho(\Psi(P))} \kappa(p_\lambda) \times f(p_\lambda) \tag{27}$$

Step 5 If $\mathcal{R}'' < \mathcal{R}_0$, repeat from Step 3. If $\mathcal{R}'' = \mathcal{R}_0$, proceed to the next step. If $\mathcal{R}'' > \mathcal{R}_0$, undo the last iteration of Step 3.
Step 6 Relax all measures in $\rho(\Psi(P))$ and transfer them to $\Omega(P)$.

7 Implementation

Our prototype augments Sun's Java Runtime Environment (version 1.4.2) running on Redhat Linux 9 (with kernel 2.4.20). Security in the Java 2 model is handled by the **AccessController** class, which in turn invokes the legacy **SecurityManager**. We instrumented the latter to invoke an **Initialize** class which constructs and initializes an instance of the **RheoStat** class [Gehani03]. A shutdown hook is also registered with the **SecurityManager** so that the intrusion detector is terminated when the user application exits. The **ActiveMonitor** class initializes the first time the **AccessController**'s *checkPermission()* method is invoked.

RheoStat implements a limited state transition analysis intrusion detector, based on the methodology of [Ilgun95]. It adds a *pre-match timer* and *post-match timer* for each signature. The first handles false partial matches by reseting the signature if the match doesn't complete in a pre-determined interval. The second is used to reset the system after a signature match completes and sufficient time has elapsed to deem that the threat has passed.

7.1 Risk Manager

A **RiskManager** class uses events generated by **RheoStat** and invokes the **ActiveMonitor**'s *enableSafeguard()* and *disableSafeguard()* methods as required. The μ matching function, described in Section 4.1, is used for estimating the threat from a partial match. We use:

$$\mu(t_\alpha, E \overset{\prec}{\cap} S(t_\alpha)) = \frac{|E \overset{\prec}{\cap} S(t_\alpha)|}{|S(t_\alpha)|} \tag{28}$$

The new risk level is calculated with the updated threat levels. If the risk level increases above or decreases below the risk tolerance threshold, the following course of action occurs.

The **RiskManager** maintains two heap data structures as illustrated in Figure 4. The first one contains all the permissions for which the **ActiveMonitor** has predicates but

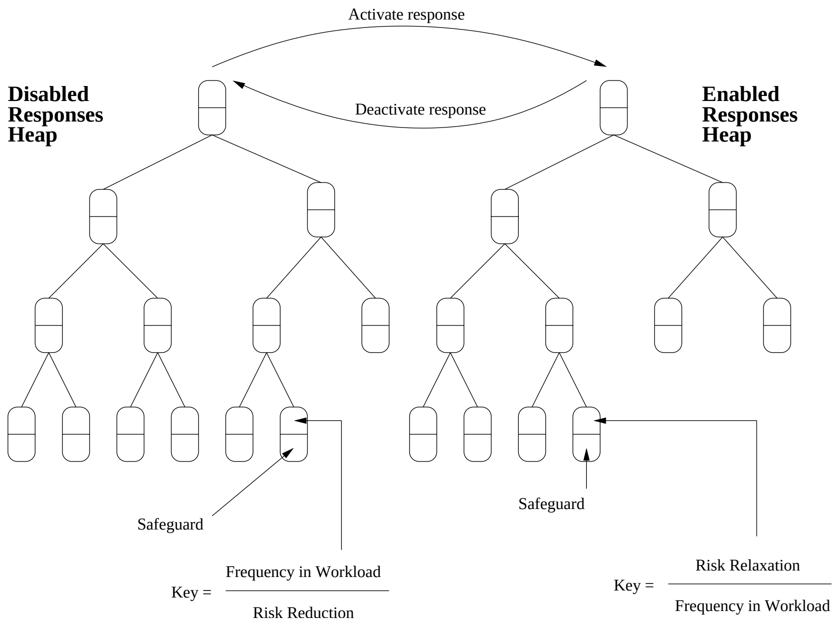

Fig. 4. When the Risk Manager needs to activate a response to effect risk reduction, it attempts to select the one which will minimize the runtime overhead while maximizing the risk reduction.

are currently unused. The objects are stored in the heap using the cost-to-benefit ratios as the keys. The second heap contains all the permissions for which the **ActiveMonitor** is currently evaluating predicates before it grants permissions. The objects in this heap are keyed by the benefit-to-cost ratios. When the risk level rises, the **RiskManager** extracts the minimum value element from the first heap, and inserts it into the second heap. The corresponding predicate evaluation is activated. The risk level is updated. If it remains above the risk tolerance threshold the process is repeated until the risk has reduced sufficiently. Similarly, when an event causes the risk to drop, the **RiskManager** extracts the minimum element repeatedly from the second heap, inserting it into the first heap, disabling predicate checks for the permission, while the risk remains below the threshold of tolerance. In this manner, the system is able to adapt its security posture continuously.

8 Evaluation

The NIST ICAT database [ICAT] contains information on over $6,200$ vulnerabilities in application and operating system software from a range of sources. These are primarily classified into seven categories. Based on the database, we have constructed a suite of attacks, with each attack illustrating the exploitation of a vulnerability from a different category. In each case, the system component which includes the vulnerability is a Java servlet that we have created and installed in the W3C's Jigsaw web server (version

2.2.2) [Jigsaw]. While our approach is general, we focus below only on 4 categories for brevity. We describe a scenario that corresponds to each attack, including a description of the vulnerability that it exploits, the intrusion signature used to detect it and the way the system responds. The global risk tolerance threshold is set at 20.

8.1 Configuration Error

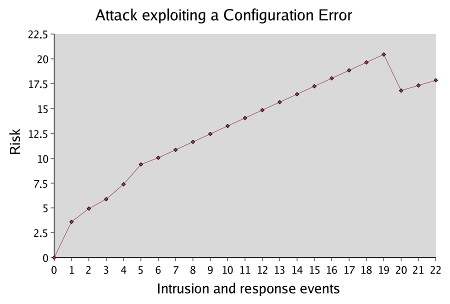

Fig. 5. Attack exploiting a configuration error.

A *configuration error* introduces a vulnerability into the system due to setting that are controlled by the user. Although the configuration is implemented faithfully by the system, it allows the security policy to be subverted. In our example, the servlet authenticates the user before granting access to certain documents. The password used is a permutation of the username. As a result, an attacker can guess the password after a small number of attempts. The flaw here is the weak configuration.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form which requests authentication information as well as the desired document. Third, the server receives another connection. Fourth, the servlet that verifies if the file can be served to the client, based on the authentication information provided, will execute. Fifth, the decision to deny the request is logged. If this sequence of events repeats twice again within the pre-match timeout of the signature, which is 1 minute, an intrusion attempt is deemed to have occurred.

In Figure 5, events $7-18$ and $20-22$ correspond to this signature. Events $1-6$ are of other signatures that cause the risk level to rise. Event 18 causes the risk threshold to be crossed. As a result, the **RiskManager** searches for and finds the risk reduction measure which has the lowest cost-benefit ratio. The system enables a predicate for the permission that controls whether the servlet can be executed. This is event 19 and reduces the risk. The predicate checks whether the current time is within the range of business operating hours. It allows the permission to be granted only if it evaluates to `true`. During operating hours, it is likely that the intrusion will be flagged and seen by an administrator and it is possible that the event sequence occurred accidentally, so the permission continues to be granted. Outside those hours, it is likely that this is an attack attempt and no administrator is present, so the permission is denied thereafter, till the post-match timer expires after one hour and the threat is reset.

8.2 Design Error

A *design error* is a flaw that introduces a weakness in the system despite a safe configuration and correct implementation. In our example, the servlet allows a remote node to upload data to the server. The configuration specifies the maximum size file that can be uploaded. The servlet implementation ensures that each file uploaded is limited to the size specified in the configuration. However, the design of the restriction did not account for the fact that repeated uploads can be performed by the same remote node. This effectively allows an attacker to launch the very denial-of-service (that results when the disk is filled) that was being guarded against when the upload file size was limited.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that allows uploads. Third, the server receives another connection. Fourth, it executes the servlet that accepts the upload and limits its size. Fifth, a file is written to the uploads directory. If this sequence of events repeats twice again within the pre-match timeout of the signature, which is 1 minute, an intrusion attempt is deemed to have occurred.

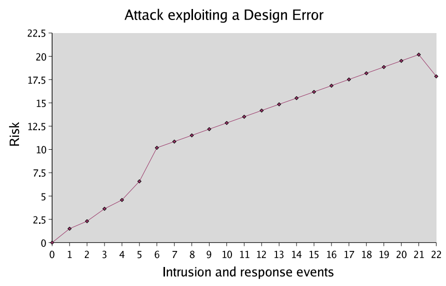

Fig. 6. Attack exploiting a design error.

In Figure 6, events $7 - 21$ correspond to this signature. Events $1 - 6$ are of other signatures that cause the risk level to rise. Event 21 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls whether files can be written to the uploads directory. This is event 22 and reduces the risk. The predicate checks whether the current time is within the range of business operating hours. It allows the permission to be granted only if it evaluates to `true`. During operating hours, it is likely that the denial-of-service attempt will be flagged and seen by an administrator, so the permission continues to be granted on the assumption that manual response will occur. Outside those hours, it is likely that no administrator is present, so the permission is denied thereafter, till the post-match timer expires after one hour and the threat is reset.

8.3 Environment Error

An *environment error* is one where an assumption is made about the runtime environment which does not hold. In our example, the servlet authenticates a user, then stores the user's directory in a cookie that is returned to the client. Subsequent responses utilize the cookie to determine where to serve files from. The flaw here is that the server assumes the environment of the cookie is safe, which it is not since it is exposed to manipulation by the client. An attacker can exploit this by altering the cookie's value to reflect a directory that they should not have access to.

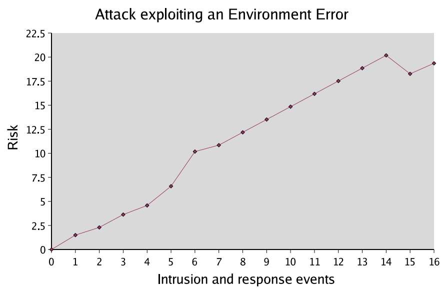

Fig. 7. Attack exploiting an environment error.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that authenticates a user. Third, the server receives another connection. Fourth, it executes the servlet that authenticates the user and maps users to the directories that they are allowed to access. It sets a cookie which includes the directory from which files will be retrieved for further requests. Fifth, the server receives another connection. Sixth, it serves the specific HTML document that includes the form which accepts the file request. Seventh, the server receives another connection. Eighth, the servlet that processes the request, based on the form input as well as the cookie data, is executed. Ninth, a file is served from a directory that was not supposed to be accessible to the user. The events must all occur within the pre-match timeout of the signature, which is 1 minute.

In Figure 7, event 3, events $8-14$ and event 16 correspond to this signature. Events $1-2$ and $4-7$ are of other signatures. Event 14 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls whether the file download servlet can be executed. This is event 15 and reduces the risk. The predicate simply denies the permission. As a result, the attack can not complete since no more files can be downloaded till the safeguard is removed when the risk reduces at a later point in time (when a threat's timer expires).

8.4 Input Validation Error

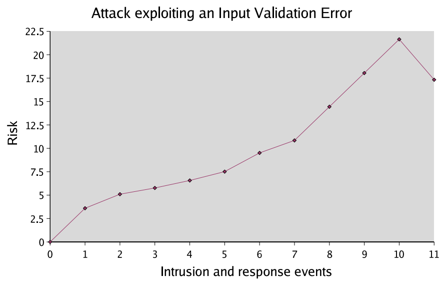

Fig. 8. Attack exploiting an input validation error.

An *input validation error* is one that results from the failure to conduct necessary checks on the data. A common example of this type of error is the failure to check that the data passed in is of length no greater than that off the buffer in which it is stored. The result is a buffer overflow which can be exploited in a variety of ways. In our example, the servlet allows a file on the server to be updated remotely. The path of the target file is parsed and a check is performed to verify that it is in a directory that can be updated. The file 'Password.cfg' is used in each directory to describe which users may access it. By uploading a file named 'Password.cfg', an attacker can overwrite and alter the access configuration of the directory. As a result, they can gain unlimited access to the other data in the directory.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that allows uploads to selected directories. Third, the server receives another connection. Fourth, it executes the servlet that checks that the uploaded file is going to a legal directory. Fifth, the 'Passwords.cfg' file in the uploads directory is written to. The events must all occur within the pre-match timeout of the signature, which is 1 minute.

In Figure 8, event 1 and events $7 - 10$ correspond to this signature. Events $2 - 6$ are of other signatures. Event 10 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls write access to the 'Passwords.cfg' file in the uploads directory. This is event 11 and reduces the risk. The predicate simply denies the permission. As a result, the attack can not complete since the last step requires this permission to upload and overwrite the 'Passwords.cfg' file. Enabling this safeguard does not affect legitimate uploads since they do not need to write to this file.

9 Related Work

We describe below the relationship of our work to previous research on intrusion detectors and risk management systems.

9.1 Intrusion Detection

Early systems developed limited ad-hoc responses, such as limiting access to a user's home directory or logging the user out [Bauer88], or terminating network connections [Pooch96]. This has also been the approach of recent commercial systems. For example, BlackICE [BlackICE] allows a network connection to be traced, Intruder Alert [IntruderAlert] allows an account to be locked, NetProwler [NetProwler] can update firewall rules, NetRanger [Cisco] can reset TCP connections and RealSecure [ISS] can terminate user processes.

Frameworks have been proposed for adding response capabilities. DCA [Fisch96] introduced a taxonomy for response and a tool to demonstrate the utility of the taxonomy. EMERALD's [Porras97] design allows customized responses to be invoked automatically, but does not define them by default. AAIR [Carver01] describes an expert system for response based on an extended taxonomy.

Our approach creates a framework for systematically choosing a response in real-time, based on the goal of reducing exposure by reconfiguring the access control subsystem. This allows an attack to be contained automatically instead of being limited to raising an alarm, and does not require a new response subsystem to be developed for each new class of attack discovered.

9.2 Risk Management

Risk analysis has been utilized to manage the security of systems for several decades [FIPS31]. However, its use has been limited to offline risk computation and manual response. [SooHoo02] proposes a general model using decision analysis to estimate computer security risk and automatically update input estimates. [Bilar03] uses reliability modeling to analyze the risk of a distributed system. Risk is calculated as a function of the probability of faults being present in the system's constituent components. Risk management is framed as an integer linear programming problem, aiming to find an alternate system configuration, subject to constraints such as acceptable risk level and maximum cost for reconfiguration.

In contrast to previous approaches, we use the risk computation to drive changes in the operating system's security mechanisms. This allows risk management to occur in real-time and reduces the window of exposure.

10 Future Directions

We utilized a simple μ function that assumed independent probabilities for successive events. However, μ functions can be defined even when pre-conditions are known. By measuring the frequencies of successive events occurring in typical and attacked workloads, conditional probabilities can be derived. A tool to automate the process could be constructed.

The exposure reduction values, workload frequencies, consequence costs and risk threshold were all manually calculated in our prototype. All such parameters will need to be automatically derived for our approach to be practical. The frequencies with which permissions are utilized can be estimated by instrumenting the system to measure these with a typical workload.

A similar approach could be used to determine the average inherent risk of a workload. An alternative would be the creation of a tool to visualize the effect of varying the risk threshold on (i) the performance of the system and (ii) the cost of intrusions that could successfully occur below the risk threshold. Policy would then dictate the trade-off point chosen.

The problem of labeling data with associated consequence values can be addressed with a suitable user interface augmentation - for example, it could utilize user input when new files are being created by application software. The issue could also be partially mitigated by using pre-configured values for all system files.

Finally, some attacks may utilize few or no permission checks. Such scenarios fall into two classes. In the first case, this points to a design shortcoming where new permissions need to be introduced to guard certain resources such as critical subroutines in system code. The other case is when the attack has a very small footprint, in which case our approach will fail (as it can't recognize the threat in advance).

11 Conclusion

We have introduced a formal framework for managing the risk posed to a host. The model calculates the risk based on the threats, exposure to the threats and consequences of the threats. The threat likelihoods are estimated in real-time using output from an intrusion detector. The risk is managed by altering the the exposure of the system. This is done by dynamically reconfiguring the modified access control subsystem. The utility of the approach is illustrated with a set of attack scenarios in which the risk is managed in real-time and results in the attacks being contained. Automated configuration of the system's parameters, either analytically or empirically, remains an open research area.

References

[Bauer88] D. S. Bauer and M. E. Koblentz, NIDX - A Real-Time Intrusion Detection Expert System, Proc. of USENIX Technical Conference, p261-273, 1988.

[Bilar03] Daniel Bilar, Quantitative Risk Analysis of Computer Networks, PhD thesis, Dartmouth College, 2003.

[BlackICE] http://documents.iss.net/literature/BlackICE/BISP-UG_36.pdf

[Carver01] Curtis Carver, Adaptive, Agent-based Intrusion Response, PhD thesis, Texas A and M University, 2001.

[Cisco] http://www.cisco.com/application/pdf/en/us/guest/products/ps2113/c1626/ ccmigration_09186a00800ee98e.pdf

[FIPS31] Guidelines for Automatic Data Processing Physical Security and Risk Management, National Bureau of Standards, 1974.

[Fisch96] Eric Fisch, Intrusive Damage Control and Assessment Techniques, PhD thesis, Texas A and M University, 1996.

[Garey79] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.

[Gehani03] Ashish Gehani, Support for Automated Passive Host-based Intrusion Response, PhD thesis, Duke University, 2003.

[ICAT] http://icat.nist.gov/icat.cfm

[Ilgun95] Koral Ilgun, Richard A. Kemmerer and Phillip A. Porras, State Transition Analysis: A Rule-Based Intrusion Detection Approach, IEEE Transactions on Software Engineering, 21(3), p181-199, March 1995.

[IntruderAlert] http://enterprisesecurity.symantec.com/content/ProductJump.cfm? Product=171

[ISS] http://documents.iss.net/literature/RealSecure/RSDP-UG_70.pdf

[Jigsaw] http://www.w3.org/Jigsaw

[Kellerer98] H. Kellerer and U. Pferschy, A new fully polynomial approximation scheme for the knapsack problem, Proceedings of the APPROX 98, Lecture Notes in Computer Science, v1444, p123-134, Springer, 1998.

[Koved98] Larry Koved, Anthony J. Nadalin, Don Neal, and Tim Lawson, The Evolution of Java Security, IBM Systems Journal 37(3), p349-364, 1998.

[NetProwler] http://symantec.com

[NIST800-12] Guidelines for Automatic Data Processing Physical Security and Risk Management, National Institute of Standards and Technology, 1996.

[Pooch96] U. Pooch and G. B. White, Cooperating Security Managers: Distributed Intrusion Detection System, Computer and Security, (15)5, p441-450, September/October 1996.

[Porras97] P.A. Porras and P.G. Neumann, EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances, Proceedings of the Nineteenth National Computer Security Conference, p353-365, Baltimore, MD, October 1997.

[SooHoo02] Kevin Soo Hoo, Guidelines for Automatic Data Processing Physical Security and Risk Management, PhD Thesis, Stanford University, 2002.

[SPECjvm98] http://www.specbench.org/osg/jvm98/