A Network-Based Response Framework and Implementation*

Marcus Tylutki and Karl Levitt

University of California, Davis, CA 95616, USA
{tylutki,levitt}@cs.ucdavis.edu

Abstract. As the number of network-based attacks increase, and system administrators become overwhelmed with Intrusion Detection System (IDS) alerts, systems that respond to these attacks are rapidly becoming a key area of research. Current response solutions are either localized to individual hosts, or focus on a refined set of possible attacks or resources, which emulate many features of low level IDS sensors.

In this paper, we describe a modular network-based response framework that can incorporate existing response solutions and IDS sensors. This framework combines these components by uniting models that represent: events that affect the state of the system, the detection capabilities of sensors, the response capabilities of response agents, and the conditions that represent system policy. Linking these models provides a foundation for generating responses that can best satisfy policy, given the perceived system state and the capabilities of sensors and response agents.

Keywords: Autonomic response, response modeling, response framework.

1 Introduction

The first intrusion detection systems were developed as low level sensors that detected attacks by using attack signatures on low level event logs [1,2,3]. Since these sensors lacked the context of a high level system policy, correlation-based intrusion detection systems were developed, which allowed for a broader context for interpreting the higher-level effect of an observed event.

Similarly, current response systems operate on a relatively small scope of possible responses and state assessment with respect to policy. As an example, the Intrusion Detection and Isolation Protocol [7] uses a simple cost model for each link and modifies firewall rules to isolate infected hosts from the rest of the network. The Light Autonomic Defense System (LADS) [8] is an effective host-based solution, but does not incorporate a system-wide policy or system-wide responses. CIRCADIA [9] is a network-wide solution that uses a simple cost model with a table lookup for determining appropriate responses. Toth and Kruegel [10] present a useful dependency-based response model that can create and modify firewall rules, kill and restart processes on individual hosts, and

* This work was sponsored by NSF grant ITR-0313411.

D. Hutchison et al. (Eds.): IWAN 2005, LNCS 4388, pp. 65–82, 2009.
© IFIP International Federation for Information Processing 2009

reset a user profile to a predetermined template. However, their attack domain is mainly limited to resource management problems and they do not present a model capable of incorporating other response systems. In addition, these response systems assume the sensors that provide their alerts are infallible.

Some response agents, such as Honeynet [14,13] and the Deception Toolkit [11,12] are extremely useful response agents for a larger response model. These response agents can be refined with information from higher-level events. If a high-level IDS predicts an attacker's goals or future targets, this information could be used to reconfigure these agents to better deceive the attacker. Other response agents, such as DDoS mitigation systems [4,5,6], can also be used and configured based on their requirements and expected performance.

This paper presents the Autonomic Response Model (ARM), an expressive model that unites sensor capabilities, response agent capabilities, attack and state related events, and system policy. This allows for the model to generate a policy-based optimal response. An implementation and framework based on this model is discussed in detail, as well as some experimental results.

2 Autonomic Response Model

The decision agent for ARM receives alerts from intrusion detection systems (i.e., sensors), in addition to agents that report policy changes and model-based changes. Any resulting response set is submitted to the corresponding response agents. The decision agent also recalculates the optimal sensor configuration with respect to policy. If this differs from the previous global sensor configuration, sensor configuration updates are submitted to the appropriate sensors.

2.1 Basic Components

This model uses several components as building blocks. *Event classes* contain attribute/value based pairs that are used to describe attack events, policy events, and state events. Each event class also has a predetermined set of policy-derived detection constraints that must be satisfied by current sensor configurations.

Event instances are instances of event classes and describe an aspect of the current perceived state of the system. Alerts from sensors are translated into event instances. Each event instance has an associated false positive probability (FPP) that represents the probability that the corresponding event that the event instance represents does not exist. This probability is directly determined by the sensor configuration that reported it or the highest FPP of all prerequisite event instances of the event instance.

Rules describe the relationship between event classes. If event instances of the prerequisite event classes exist to satisfy all of the preconditions of a rule, postrequisite event instances are generated. These postrequisite event instances are initialized from postrequisite conditions associated with the rule, which refer to attributes of the prerequisite event instances or rule-specified constant values. In addition, prerequisite event instances may be modified by a postcondition.

Sensor configurations are represented by the detection thresholds for each event class it detects. A *detection threshold* is represented as a threshold for FPP, false negative probability (FNP), and timeliness (T). FPP represents the probability that an alert produced by the sensor configuration is based on an event that does not exist. FNP represents the probability that given an event exists that should have been reported as an alert, it was not. T represents the estimated time the sensor configuration takes to report the alert from the moment the event takes place. This information can be obtained from receiver operating characteristic (ROC) curves for each event class and sensor pair[1].

Many attack modeling languages can be used for these components [15,16,17,18]. In addition, to make the translation from sensor alerts to event instances, a common report language adapted from CIDF [19] or IDMEF [20] can be used, despite the difficulties they have encountered in gaining widespread acceptance.

2.2 Prevention and Recovery Response Model

Alerts are translated into event instances and are processed one at a time by the decision engine. System policy is represented using event classes, rules, and event instances that represent key aspects of system state. If an event instance is created that belongs to a policy violation labeled event class, then the decision engine searches for an optimal response set to handle the problem. This response set recovers from the effects of the policy violation and prevents attempts from an identical attack vector from resulting in another policy violation. Each policy violation event class also has a field that determines the acceptable FPP threshold that the corresponding event instance's FPP must be below to be considered as a valid policy violation. If a policy violation event instance ($PVEI$) does not exist, or existing $PVEI$ FPPs surpass their corresponding thresholds, then the decision engine waits for more alerts.

Finding Solutions. Each $PVEI$ has at least one prerequisite event instance that matched a rule to create it, even if it was generated from a one-to-one rule matching. Each prerequisite may have additional prerequisites that contributed to the generation of the $PVEI$. This generates a tree of event instances that resulted in the generation of the $PVEI$. A satisfying response solution set is able to recover all event instances on a path within this tree, as well as preventing at least one of these event instances from reverting back into the state that resulted in the generation of the $PVEI$. Recovery responses are therefore highly relative to the context of the event instances they are recovering within the policy violation tree. Recovery responses are designed to break the conditions of the rule in which the corresponding event instance was used to generate

[1] It is acknowledged that most current ROC curves for sensors describe their average overall detection capabilities, rather than being associated with specific alert types or categories. It is also acknowledged that these probabilities and values are highly dependent upon the environment in which they are recorded.

the $PVEI$. Prevention responses are designed to ensure that newly changed values do not easily revert to their previous values. Prevention responses on event instances at leaf nodes of the policy violation tree, which are not translated from an attack alert[2], are resistant to attacker influence from previous attack vectors.

If a prevention response is unavailable due to the lack of information on the origin of an event instance, backchaining can be applied. Suppose a Tripwire [21] sensor reports that a filesystem has been compromised, but no other alerts can identify the source of the service that resulted in the compromised filesystem. Backchaining obtains all services with access to the compromised filesystem that may be at fault. If prevention responses are initiated for all of these services, the effect is the same as initiating a prevention response for the compromised filesystem event instance.

Similarly, an anomaly-based sensor may not be able to pinpoint the origin of an event instance as well as a signature-based sensor. Alerts from a signature-based sensor typically directly determine the specific vulnerability that the attack attempted to exploit. By comparison, an anomaly-based sensor may only be able to report generic attack behavior from a particular source to a particular host and/or service. Backchaining can be used to cover all possible attack behaviors against the targetted service.

Evaluating Solutions. The response set for a particular path represents the response event classes that are associated with rules or other event classes. Each generic response event class can be initialized into a response event instance based on the values of the corresponding event instance to which it is responding, or based on the prerequisite event instances and prerequisite conditions of the corresponding rule matching within the policy violation tree.

Within a specific path in the policy violation tree, different response sets are tested by temporarily adding the corresponding response event instances. After the testing of all combinations of a path are complete, new paths in the tree are tested. The response set that produces the best state assessment when tested is the response set that is initiated.

A simple metric for assessing the state is the sum of all state assessment values (SAVs) of current event instances. Rather than have a SAV for all possible states, this approach has a SAV for each event instance, which corresponds to that event instance's influence on the assessment of the overall state of the system. Event instances that are critical with respect to policy, such as critical services, have positive SAVs. Event instances that represent penalties to the system policy and impact the availability or integrity of the system, have negative SAVs. Each event instance has a SAV associated with the event class from which is it a member. Rules and policies can have exceptions to this default allocation can modify or override these values.

This assessment method can be enhanced through the addition of assessment rules. These rules modify the overall SAV of a state based on the presence or

[2] Alerts from system state sensors, such as sensors that scan current versions for available services, are acceptable.

absence of particular event instances, and can be synergistic (i.e., two event instances result in a net SAV greater than their sum) or dyssynergistic (i.e., two event instances result in a net SAV less than their sum). Cost models based on risk analysis [22] can also be adapted to determine these values.

Once a global optimal response set is found, these response event instances are added to the decision agent's current state, and each response event instance is submitted to the appropriate response agent. Each response agent that receives the response makes the appropriate changes to its local configuration.

2.3 Sensor Retargeting

Sensors can become reconfigured based on policy changes. As critical tasks for a system change, so should its policy. Some of these critical tasks may be time dependent, short term tasks, while others may be more long term tasks. These tasks are represented in the policy model and correspond to individual event instances with corresponding SAVs. In addition to their use for assessing the state of the system, SAVs can be used to prioritize the detection of particular event classes. SAVs of an event class can be used to directly determine its allowable detection thresholds.

In addition, sensors have resource costs. If costs were ignored for integrity scanners, such as Tripwire, then the constant operation of these scanners are likely to impact the performance of the scanned hosts. Instead, the traditional tradeoff between performance and stability (or security in this case), is acknowledged, allowing these scanners to only run periodically. The balance of this tradeoff can be shifted depending on the SAV of the event class that the sensor attempts to detect. A higher SAV event class is more critical, and therefore results in lower detection thresholds.

$$S_i = \prod_{r=1}^{n} (RL_r - R_k(i)) \tag{1}$$

The overall cost for a set of sensor configurations can also be assessed with a more complex load balancing metric, as shown in Equation 1, where the solution set with the highest value of S_i is considered the most efficient with respect to policy. A similar metric can be adapted that measures the distance between current detection capabilities of sensor configurations and event class detection thresholds. This alternative metric prefers better detecting sensor configurations over resource conserving sensor configurations.

Sensors can also be preemptively retargeted. Suppose an event instance (EI_n) is generated from a rule matching of other event instances. If EI_n belongs to an event class with very low detection thresholds due to its SAV, then these detection thresholds are passed down to the prerequisite event classes from the rule that generated EI_n. The first step for this detection threshold propagation is to obtain $\gamma(P, r)$, which represents how close prerequisite event instances in P are to creating a successful match with rule r and is defined in Equation 2. Each event class possesses an α value, which represents the event class' relative importance

compared to other event classes for rule matchings[3]. γ is initialized with the sum of all α values of prerequisite event classes, multiplied by a β factor[4], which is an attribute of the rule that is matched. If a prerequisite event instance matches all preconditions for the rule, the entire α value for that prerequisite's event class is subtracted from γ. Partial matches result in only subtracting $\frac{\alpha}{2}$. These values are also influenced by the false positive probabilities of each prerequisite event instances, as shown in Equation 2.

$$\gamma(P,r) = \beta_r \sum_{i \in P} (\alpha_i) - \sum_{j \in Full(P,r)} (\alpha_j \, (1.0 - FPP_j)) - \sum_{k \in Partial(P,r)} \left(\frac{\alpha_k}{2} \, (1.0 - FPP_k) \right) \tag{2}$$

New detection thresholds are propagated for prerequisite event classes from each detection threshold of each postrequisite event class. The k^{th} new propagated FPP detection threshold ($ND_{FPP_{i,k}}$) for prerequisite event class i is defined in Equation 3, where $D_{FPP_{j,n}}$ represents the n^{th} FPP detection threshold for postrequisite event class j. False negative probability thresholds and timeliness thresholds are propagated using the same equation, but for their respective threshold values.

$$ND_{FPP_{i,k}} = \left(1.0 + \frac{\gamma \, (P,r)}{\alpha_i} \right) D_{FPP_{j,n}} \tag{3}$$

2.4 Attacks and Countermeasures

Attacks against ARM exploit the expressiveness of event classes and the established relationships between them. Even with a perfect model of event classes and their relationships, attacks may exploit the time it takes for the model to respond to an event by flooding it with trivial decisions that take a significant amount of time for the decision agent to determine. In addition, poorly designed models may cause infinite loops for exhaustive decision agents. When control systems perpetually overestimate or underestimate the current or future state of the system, increasingly inaccurate responses can occur, resulting in an unsteady state. Control systems used in chemical and electrical engineering have adapted meta-control agents that observe the behavior and results of their control system to enhance their reliability and have been able to prevent unsteady states from occuring within limited domains. Through the application of a similar meta-control agent for this model, many of the effects of these attacks can be mitigated.

[3] As an alternative, α values could be tied to a specific prerequisite event class and rule pair which would be more precise, but would likely result in more α values to calibrate.

[4] β is considered to be greater than 0. This can result in propagating more strict detection thresholds if β is less than 1, since γ can be negative for this case. Additionally, if β is always greater than 1, the propagating detection thresholds are always less strict.

3 Implementation

Despite the previously mentioned response systems, and a response testbed [23], a modular response framework and testbed is not readily available. This modular response framework was developed and used in the Emulab [24] environment and is freely available upon request.

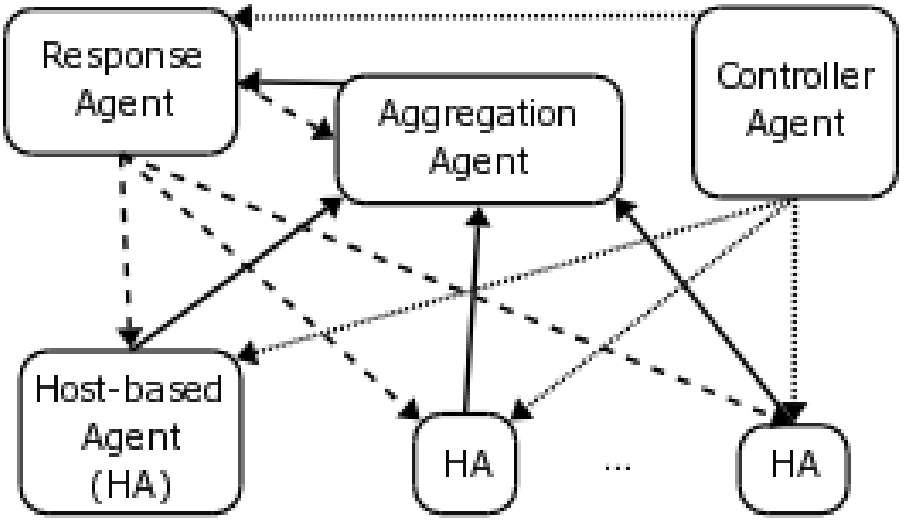

Fig. 1. Overall Response Framework

The implementation of this response framework that is presented in Figure 1 is comprised of four types of agents. The *host-based agent* is responsible for: simulating the effects of host and network based intrusion detection sensors, executing responses received from the response agent, and storing local host-based vulnerability profiles and sensor configurations. The *aggregation agent* is responsible for aggregating reports received from all host-based agents and submitting the relevant new reports to the response agent. Reports that have a higher false positive probability than a previously received report is not forwarded to the response agent. The *response agent* receives all sensor reports from the aggregation agent. Each report is processed individually. If a new policy violation occurs, the response agent searches for an optimal response set. If one is found, the corresponding responses are submitted to all applicable host-based agents. In addition, new sensor configurations are evaluated to determine if a new global sensor configuration better satisfies system policy. New sensor configurations are sent to the corresponding host-based agents. The *controller agent* is responsible for initializing all the other agents, and initiating any external attacks against the network.

3.1 Document Types

Messages passed between agents are in the form of XML documents. This subsection briefly describes the XML schemas used in the implementation. The `event class` schema contains the unique identifier for that event class, as well as all appropriate fields that are associated with the class. Fields can be of type integer, float, or string. This schema also supports detection thresholds for the event class, as well as the α value previously discussed in Subsection 2.3.

The `alert` schema contains a unique identifier of the host and attack event that triggered it, as well as a FPP value that is associated with the probability that the alert is a false positive. Alert schemas contain fields that represent specific values that differ from the defaults of the event class from which it is derived. In addition, `alert` schemas are used for passing response messages from the response agent to the host-based agents.

The `host profile` schema contains information about a host, including its IP address and services available on that IP address. This schema also mentions which filesystems each service has access to, as well as the version number of the service and its dependencies with other services.

The `IDS profile` schema lists the identifiers of the event classes that the specific configuration detects and their corresponding detection values for FPP, FNP, and T. This sensor configuration schema also contains a generic resource cost for operating the sensor with this configuration. This schema also supports a NULL configuration for sensors that are disabled.

The `rule` schema includes prerequisite event class identifiers, postrequisite event class identifiers, preconditions, and postconditions. Preconditions are represented by referencing the local rule identifier of the prerequisite event instance along with the name of its field that is being compared. For example, position 3 and field "Port" would refer to the rule's third prerequisite event instance's port value. Comparisons can be made to a constant value, or to another prerequisite event instance's field value. The supported operators are equal, not equal, greater than or equal, less than or equal, greater than, and less than. Postconditions only support the equal operator for the initialization of postrequisite values. Postconditions can also be used to modify prerequisite event instance field values by referencing negative identifier values. In addition, `rule` schema include the β value of the rule that is used for preemptive sensor retargeting previously described in Subsection 2.3.

The `response map` schema maps event classes to recovery and prevention response event class sets. It references the event class identifiers for each event class involved in the mapping, as well as additional fields for the response event classes to specify additional initialization information, as well as additional fields for the source event class that limit the applicability of the mapping. For example, a response map for an infected filesystem event class could restrict the response map to a specific filesystem. In addition, the response map specifies if the corresponding response sets are prevention response sets, recovery response sets, or both.

The next two schemas were adapted from Joseph McAlerney's thesis [25], which presented a framework for simulating and recording worm behavior using

agents and XML documents. The `event profile` schema represents the vulnerability profile of the host-based agent with respect to an individual attack event class. This schema was modified to support a requirements field, which lists the required services or filesystem needed for the corresponding attack event to succeed. The `event properties` schema is used for documents that represent attack events. It specifies propagation details, including the rate and amount of attacks that are initiated, if the attack event is intended to propagate. This schema was modified from the worm simulation version by adding fields to represent the effects of the attack, which correspond to creating specific event instances. This schema was also modified to represent the filesystem (including memory) that the attack resides in, if it is persistent.

3.2 Host-Based Agent

The code used for the host-based agents were adapated from the worm simulation project [25] to support responses, sensor reconfigurations, sensor simulation for false positives, false negatives, and timeliness values. The code was also adapted to support the changes to event profiles and event properties discussed at the end of Subsection 3.1[5]. Host-based agents are the only agents in the implementation that are multi-threaded. When a new document is received, a new thread is created to parse and process the document. Mutexes are used to ensure shared data constructs are not shared while a thread is operating in a critical section. A host-based agent is initialized with: all event class definitions used in an upcoming experiment, current IDS configurations, a host profile document describing the simulated services running on the host, and an event profile document for each attack class used in the upcoming experiment.

Once an event properties document is received from either another host-based agent or the controller agent, the document is parsed into a local structure and current IDS configurations are checked to determine if any sensors successfully detect the attack. If the randomly generated probability is above the detecting sensor configuration's FNP value for the attack's corresponding event class, a new thread is spawned which generates the corresponding alert and submits it to the aggregation agent after sleeping for a period of time derived from the T value of the sensor configuration. If a sensor configuration fails to detect the attack, it is locally blacklisted from being able to report future occurrences with the same attack identifier[6].

If the host-based agent is not vulnerable to the received attack event, the thread terminates. Otherwise, the effects of the thread get added as local event

[5] All other agents, with the exception of the controller agent that sends arbitrary XML files to a designated host and port, were not associated with the worm simulation project.

[6] To clarify attack identifiers, suppose an experiment consisted of two worms. Each worm would have a different attack identifier. If a host-based agent received the worm from multiple hosts, each attack event would have the same attack identifier. However, if it received a different worm from the same host, it would have a different attack identifier.

instances, and sensor configurations that detect the corresponding event classes of these effects are examined. As above, if a sensor configuration detects an event instance, a new thread is spawned which submits the alert after sleeping.

False positives are represented by receiving a non-attack event property document that mirrors a specific attack event property in every other way. If the randomly generated probability is below the FPP[7] for a given sensor configuration, an alert is generated and submitted as mentioned above, representing a false positive that appears identical to a true positive.

Host-based agents are also responsible for updating local sensor configurations to those received from the response agent. In addition, host-based agents process responses from the response agent. Some responses, such as filesystem recovery responses, require a delay, which is specified in the appropriate delay-related fields of the response. Similar to alert reporting, a new thread is spawned which sleeps for the amount of time the response takes to complete. Other responses, such as firewall rule changes, are made instantaneously. In addition, since dependencies between filesystems and services are represented, responses that temporarily disable services or filesystems also disable services that require them. These filesystems and services are restored when the corresponding response is complete. Some responses require a service or filesystem to be available before it can become available again. Responses that have overlapping requirements are initiated sequentially on a first-come-first-serve basis. If a response recovers a filesystem that an attack event was residing in, the attack event ceases propagation. Prevention responses prevent future attack events from succeeding by making the host no longer satisfy the requirements for the attack event that are specified in the event profile.

3.3 Aggregation Agent

When the aggregation agent receives an alert, it compares the host and attack identifiers of the alert to previously seen alerts. If it does not find a match, it records the alert and forwards it to the response agent. If it finds a match, it compares the new alert's FPP to the recorded alert's FPP. If the new alert's FPP is lower than the previous matching alert's FPP, it passes this alert on to the response agent and overwrites the old alert with the new alert. Otherwise, the alert is dropped.

3.4 Response Agent

The response agent is initialized by receiving: event class documents defining all event classes to be used in the upcoming experiment, all IDS configuration profiles for all available sensors, the host profiles of all hosts, an event profile document for each host and attack event pair, rule documents including backchain

[7] Recall that the definition provided for FPP is the probability that a given alert is a false positive, rather than the probability that a false positive alert will be generated from non-attack events. As a result, each sensor configuration could also contain these alternative false positive probabilities for this purpose.

rules as described near the end of Subsection 2.2, and response map documents that map event classes to available responses.

When an alert document is received, it is first translated to a local event instance (EI_l). If EI_l's values are identical to a currently existing event instance (EI_p), and EI_l's FPP is lower than EI_p's FPP, then EI_p's FPP is updated to EI_l's FPP, detection propagation thresholds are recalculated, and the response agent skips attempting to match the new event instance with other existing event instances. Otherwise, rules that require EI_l's event class are then checked with EI_l and all current event instances. As event instance combinations are tested, preemptive detection threshold propagation occurs, as discussed previously at the end of Subsection 2.3. Newly generated event instances inherit a FPP equal to the highest FPP of their prerequisite event instances. New event instances and modified event instances are added to a queue. Once all rules are checked for EI_l, event instances from the queue are added one at a time, just as EI_l was added, until the queue is empty. Rules that modify currently existing event instances must be designed carefully. Incorrect versions of these rules may result in an infinite loop where an event instance is constantly changed back and forth or the queue never becomes empty. All newly added and modified event instances are appended to a rollback queue as transactions.

$$SAV_{overall} = \sum_{i \in EI_{Current}} (SAV_i \, (1 - FPP_i)) \tag{4}$$

If a policy violation event instance is generated, the response agent searches for a response solution set as described previously in Subsections 2.2 and 2.2. Before testing a response set, the rollback queue is cleared. After adding response event instances the system state and the resulting state is assessed, the system state is rolled back by rolling back each transaction on the rollback queue. A tested response set's resulting system state assessment ($SAV_{overall}$) is defined in Equation 4, where $EI_{Current}$ represents the set of currently existing event instances. If a response set is found to provide a state that is estimated to be better than the current state, the responses are translated to alerts that are then submitted to the corresponding host-based agents.

After the response phase, sensor configurations are analyzed with respect to all event class detection thresholds that must be upheld. If it is found that a detection threshold can not be satisfied by any existing sensor configuration, the response agent generates a local alert and the detection threshold is flagged as impossible. Some detection thresholds can also be virtual, representing the notion that they are intended to be inherited through preemptive detection threshold propagation rather than satisfied for the parent event class. All sensor configurations are then tested to determine the global sensor configuration that satisfies all detection thresholds but has the lowest impact on resources. For the purposes of this implementation, resource impact is assessed by the sum of all resource costs of sensor configurations into a single value, which could be enhanced with a more thorough cost model [22] or use of the more advanced metrics previously discussed in Subsection 2.3. If a new sensor configuration is found, the IDS profile representing the new configuration is sent to the affected host-based agents.

4 Experiments

A worm buffer overflow scenario was used for the experiments with this implementation. The majority of experiments were executed on a 7 node network on Emulab [24] where one node provided the aggregation agent and the response agent, and the remaining 6 nodes provided the host-based agents.

4.1 Setup

The experiments used a host-based anomaly IDS, a network-based signature IDS, and a host-based integrity IDS. The host-based anomaly IDS is similar to an anomaly-based IDS presented by Wenke Lee and Salvatore Stolfo [26]. In this case, a sliding window is used to observe anomalies in traffic patterns. The larger the traffic window, the lower the false negative and false positive probability, but results in a larger timeliness value strictly based on the size of the window, which includes window sizes of 5, 10, 30, 60, and 90 seconds. Since this sensor takes a traffic stream as input, if there exists a temporary cache of this traffic, the retargeted sensor could process old traffic with a new sensor configuration for an additional attempt to detect an attack or provide more evidence to a correlation-based sensor.The network-based signature IDS is loosely based upon Snort [27] or Bro [28] and only has default and NULL configurations. The timeliness values for this sensor is estimated to be approximately 80 milliseconds, based on a report presented in [29]. The host-based integrity IDS is Tripwire [21], which can scan a filesystem every 10, 15, 20, 30, 45, 60, 120, 240, or 720 minutes. The more frequent filesystem checks are intended for small but critical filesystems. Available responses included upgrading a service, disabling a service, and restoring a filesystem.

The worms tested were run at propagation speed of one scan per 5 microseconds (fast), one scan per 50,000 microseconds (medium), and one scan per 80,000 microseconds (slow). In most experiments the vulnerability density was set to 0.5, representing 3 vulnerable nodes and 3 invulnerable nodes in the 7 node experiments. Experiments that exhibit the retargeting capabilities of the implementation used a vulnerability density of 0.83, which resulted in 5 vulnerable nodes and one invulnerable node in the 7 node experiments.

4.2 General Results

Figure 2 presents the average of 10 experiments using a 7 node experiment with a vulnerability density of 0.5 with a propagation speed of one scan per 5 microseconds. The y-axis represents the number of hosts, and the x-axis represents time in seconds on a logarithmic scale. Each graph is comprised of three lines: one for the number of infected hosts, one for the number of clean hosts, and one for the number of contained hosts. Five of these experiments under these conditions resulted in one node that failed to detect the attack with an anomaly or signature-based sensor, but where the Tripwire sensor succeeded in detecting

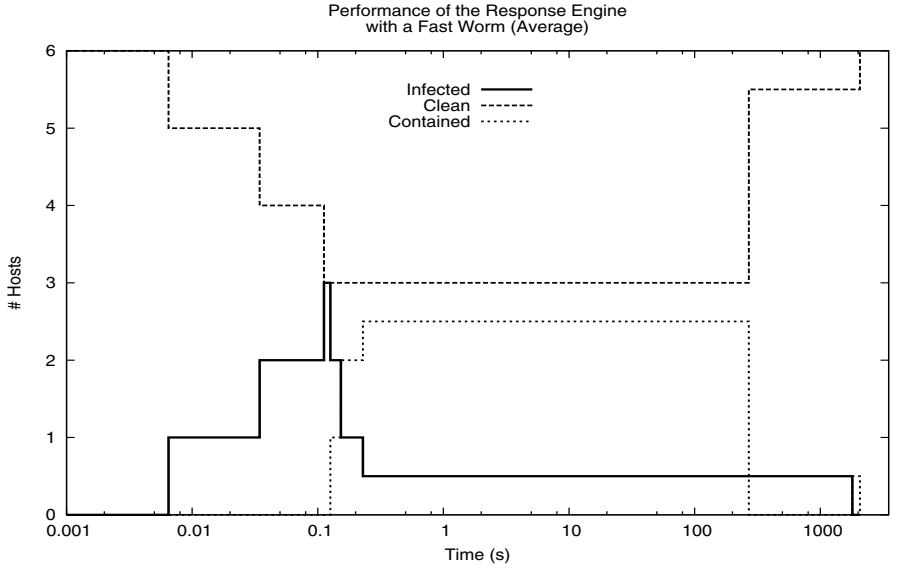

Fig. 2. Average Performance of Fast Experiments

the attack[8]. In this case, the node was only recovered after sending the Tripwire sensor alert, which took ˜1800 seconds or 30 minutes with the default sensor configuration compared to the ˜0.1 seconds or less for nodes that detected the attack with an anomaly or signature-based sensor.

Figure 3 presents the results of one experiment using the same experimental setup but with a propagation speed of one scan per 50,000 microseconds. In this case, the worm is caught before being able to spread to a vulnerable host, resulting in only one infection. About half of the remaining medium experiments exhibited this behavior, with the remainder representing the behavior shown for the fast worms. All but one of the slow experiments exhibited behavior similar to that of Figure 3.

One of the rules used represents the generation of an unknown worm event class. This rule requires three event instances that have the same compromised filesystem but reside on different hosts. In experiments with a vulnerability density of 0.83, as this rule received more partial matches, and eventually a full match, the preemptive detection threshold propagation discussed at the end of Subsection 2.3 resulted in new sensor configuration changes, which included

[8] Even though this allows for the case that a host-based anomaly IDS can miss the attack against one host, but catch the attack against another host, these discrepancies can be due to different background traffic observed on each host at the time of the attack. Similarly, this allows for the network-based signature IDS to catch an attack against some hosts, but miss them against others. This can be due to polymorphic worms where the available signature is able to detect some variants of the worm, but not all, and the worm changes form as it spreads.

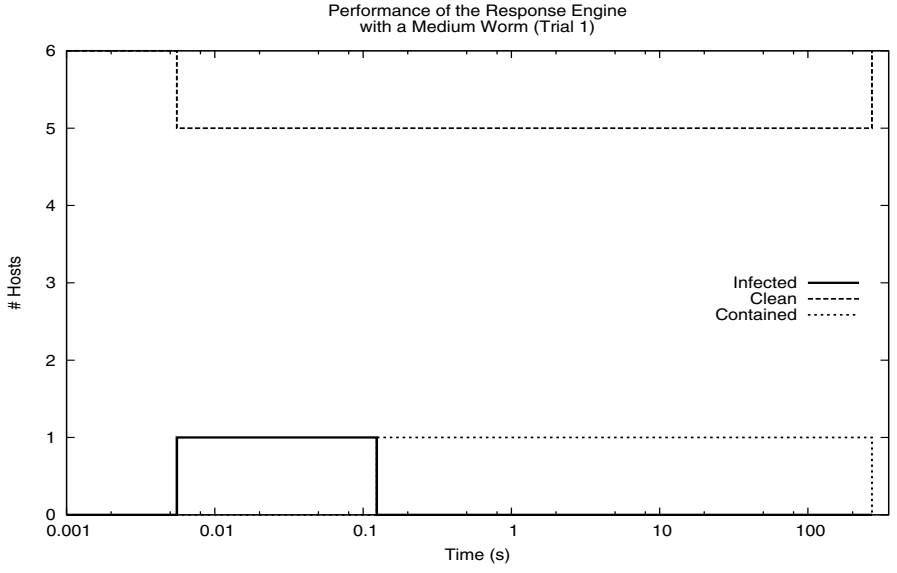

Fig. 3. Medium Experiment #1

lowering the Tripwire timeliness values from one scan per 30 minutes to one scan per 20 minutes. For some nodes that failed to detect the attack with a signature or anomaly-based sensor, this reduced the amount of time the alert was sent from the infected node to the response agent from ˜1800 seconds to ˜1200 seconds.

4.3 Scalability Test Results

For scalability concerns, the experiment was also executed with 15 and 31 nodes with a 0.5 vulnerability density. On average, 7 node experiments resulted in the response agent calculating the last optimal response set within 0.019993 seconds. The 15 node experiment was able to calculate the last optimal response set within 0.280744 seconds, while the 31 node experiment took 2.49284 seconds resulting in about a 10 fold increase each time the number of nodes are doubled. However, it should be noted that these results were obtained with extensive debug log information, printing out timestamps and data for key points within the decision engine, including partial and full rule match information, and new detection threshold additions from propagation. With the trimming of just rule match notifications (but not information about newly generated event instances), the 31 node experiment was able to reduce the calculation time to 0.700421 seconds, which could likely be reduced further with less precise timestamps and further trimming. Note that decreasing the number of nodes or lowering the vulnerability density increases performance.

4.4 Advanced Scenario

The following scenario can be adapted into an experiment with this implementation with relatively minor adjustments. In this scenario, the attacker utilizes multiple attack vectors for the primary goal of obtaining access to a relatively secure workstation, *halfdome*. *Halfdome* runs a local firewall and does not provide any services that are externally visible. The network *halfdome* is on (N) does not utilize any firewalls and is externally accessible.

In the first step, the attacker initiates a worm similar to the worm described in the experiments within Subsections 4.2 and 4.3. Although *halfdome* is not compromised, *pinatubo*, which resides on network *N,* is vulnerable and becomes compromised. This attack goes undetected by quick intrusion detection sensors, but will be detected by an upcoming integrity scan. The attacker then attempts to sniff passwords from network *N*, but is unable to find unencrypted traffic involving *halfdome*. However, the attacker is able to discover that *halfdome* uses *hawkeye* for DNS requests, which happens to be a Windows 2000 DNS server.

The attacker then launches a Distributed Denial of Service (DDoS) attack against *hawkeye* using attacks from remote hosts, as well as attempting to steal *hawkeye*'s IP address using *pinatubo* and other local, compromised hosts by spoofing ARP requests and replies. The DDoS attack is easily detected by sensors and is reported to the response agent which initiates pushback on cooperating routers [4]. Pushback provides some mitigation of the attack, but is unable to provide complete protection from the attack due to the limited domain of routers that support pushback. After the response agent receives a status alert on the partial success of the pushback response, it activates a proportional-integral-derivative controller-based response [6]. This combined response towards the DDoS attacks sufficiently mitigates the attack to allow for critical services to satisfy availability levels determined by policy.

During the external attack, the response agent also receives an alert of spammed ARP spoofs from hosts trying to steal *hawkeye*'s IP address. This event is detected, but the response agent does not initiate a response since it is unsure which ARP replies are spoofs and which are genuine. A correlation sensor is able to suggest a common service the ARP spoofing hosts shared as a possible source of infection. The response agent also receives this report, but due to the high false positive probability, it does not issue a response.

During the DDoS attack, the attacker is able to poison the DNS cache of *halfdome* by spoofing DNS replies [30,31] from hosts that were able to temporarily successfully steal *hawkeye*'s IP address. Because the attacker attempts to spam the DNS replies to *halfdome* in an attempt that one will get through, an anomaly-based sensor detects the attack and forwards an alert to the response agent. A correlation sensor sees the anomaly-based sensor's alert and correlates this with the ARP spoofs and the possible common compromised service to produce a low false positive report on the compromised hosts. This results in a Tripwire integrity scan of all suspicious hosts while the traffic from *halfdome* to external sites is temporarily throttled at *halfdome*'s local firewall. All suspicious hosts are soon confirmed to be infected, which results in the restoration of in-

fected filesystems and the disabling of the previously correlated service on the infected hosts.

Alternatively, backups of the recovered systems are made with the disabled service. Once complete, an attempt is made to upgrade the vulnerable service to a more recent version. An automated testing procedure is executed to detect if dependent services are still able to function with the upgraded service. If successful, additional previously infected hosts attempt to upgrade their service as well and test for problems. If a problem occurs, the previous image is rolled back with the service disabled and the local administrator is notified to remedy the problem.

In order to use this scenario with the current developed framework, a few key changes would have to be made. First, response feedback would have to be added by preserving policy violations that were responded to, observing the results of previous responses by retargeting sensors to observe the corresponding event classes, and initiating alternative responses that are stored along with the previous policy violation. Second, the response agent must be able to correlate seperate alerts into an overall attack vector, which is a problem many correlation systems have attempted to solve. Third, the `event properties` schema must be modified to be able to encapsulate other `event properties` documents, allowing for any attack scenario.

5 Conclusions and Future Work

In this paper we presented a modular, extensible response framework along with an implementation of a response system that utilizes this framework. The framework allows for various simulated or real intrusion detection systems, response agents, and aggregation agents. The response model and implementation presented demonstrated the benefits of sensor retargeting and supporting an expressive model that encompasses a wide variety of attacks, sensors, response agents, and policies. The experimental results presented could be compared for different high level response systems for specific response scenarios for purposes of evaluation. Although the experimental scenarios were relatively simple, a detailed scenario is presented that can be executed with minor modifications to the current implementation. Although attacks such as infinite loops are possible through poor design as described in Subsection 3.4, they can be prevented or mitigated with loop timeouts, or with the integration of a professional expert system that is designed to catch such loops.

There are many other approaches for extending and enhancing this work, in addition to those proposed in Subsection 4.4, including the following:

- Bayesian inferencing can be used to more accurately calculate the false positive probabilities of an event instance by taking several additional conditional probabilities into account.
- This probabilistic model can be used to create a metric that assesses the detection or response capability of a system by comparing the probabilities

that a system can be recovered and prevent future attacks for a given general scenario within a specified timeframe.
- The integration of a professional expert system into the response agent, which would greatly increase the efficiency of the implementation, but would make the preemptive detection threshold propagation discussed in Subsection 2.3 more difficult.
- A model for making sensor configuration detection threshold values a function of state properties makes these values much more realistic and dynamic.
- By modifying rules to allow for any type of computation, rather than straight-forward expert system rules, entire sensors/response engines or their components can be included in the model.

References

1. Snapp, S., Brentano, J., Dias, G., Goan, T., Heberlein, T., Ho, C., Levitt, K., Mukherjee, B., Smaha, S., Grance, T., Teal, D., Mansur, D.: DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an Early Prototype. In: Proc. 14th National Computer Security Conference (1991)
2. Heberlein, L., Dias, G., Levitt, K., Mukherjee, B., Wood, J., Wolber, D.: A Network Security Monitor. In: Proc. IEEE Symposium on Security and Privacy (1990)
3. Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA. SunSHIELD Basic Security Module Guide, Solaris 7, Part No. 805-2635-10 (October 1998)
4. Ionnidis, J., Bellovin, S.M.: Implementing Pushback: Router-based Defense against DDoS Attacks. In: Proc. The Network and Distributed System Security Symposium (2002)
5. Sterne, D., Djahandari, K., Wilson, B., Babson, B., Schnackenberg, D., Holliday, H., Reid, T.: Autonomic response to distributed denial of service attacks. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, p. 134. Springer, Heidelberg (2001)
6. Tylutki, M., Levitt, K.: Mitigating distributed denial of service attacks using a proportional-integral-derivative controller. In: Vigna, G., Krügel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 1–16. Springer, Heidelberg (2003)
7. Rowe, J.: Intrusion Detection and Isolation Protocol: Automated Response to Attacks. In: Recent Advances in Intrusion Detection (1999)
8. Kreidl, O., Frazier, T.: Feedback Control Applied to Survivability: A Host-Based Autonomic Defense System. IEEE Transactions of Reliability 52(3) (2003)
9. Musliner, D.: CIRCADIA Demonstration: Active Adaptive Defense. In: Proc. DISCEX 2003 (2003)
10. Toth, T., Kruegel, C.: Evaluating the Impact of Automated Intrusion Response Mechanisms. In: Proc. 18th Annual Computer Security Applications Conference (2002)
11. Cohen, F., Lambert, D., Preston, C., Berry, N., Stewart, C., Thomas, E.: A Framework for Deception (July 2005) (accessed July 2005), http://www.all.net/journal/deception/Framework/Framework.html
12. Cohen, F.: Leading Attackers through Attack Graphs with Deceptions. Computers and Security 22(5), 402–411 (2003)
13. The Honeynet Project (accessed June 2005), http://www.honeynet.org

14. Spitzner, L.: The Honeynet Project: Trapping the Hackers. In: Proc. IEEE Symposium on Security and Privacy (2005)
15. Templeton, S., Levitt, K.: A Requires/Provides Model for Computer Attacks. In: Proc. 2000 New Security Paradigms Workshop, pp. 31–38 (2000)
16. Cheung, S., Lindqvist, U., Fong, M.: Modeling Multistep Cyber Attacks for Scenario Recognition. In: Proc. DISCEX 2003 (2003)
17. Michel, C., Mé, L.: AdeLe: An Attack Description Language for Knowledge-Based Intrusion Detection. In: Trusted Information: The New Decade Challenge: IFIP TC11 16th International Conference on Information Security (IFIP/SEC 2001), pp. 353–368 (2001)
18. Cuppens, F., Ortalo, R.: LAMBDA: A language to model a database for detection of attacks. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 197–216. Springer, Heidelberg (2000)
19. Staniford-Chen, S., Tung, B., Schanckenberg, D.: The Common Intrusion Detection Framework (CIDF). In: Information Survivability Workshop (1998)
20. Debar, H., Curry, D., Feinstein, B.: The Intrusion Detection Message Exchange Format. Internet Draft (July 2004) (accessed July, 2005), http://xml.coverpages.org/draft-ietf-idwg-idmef-xml-12.txt
21. Kim, G., Spafford, E.: The Design and Implementation of Tripwire: A File System Integrity Checker. Technical Report CSD-TR-93-071, Purdue University, West Lafayette, IN 47907-1398
22. Lee, W., Fan, W., Miller, M., Stolfo, S., Zadok, E.: Toward Cost-Sensitive Modeling for Intrusion Detection and Response. Journal of Computer Security, 5–22 (2002)
23. Rossey, L., Cunningham, R., Fried, D., Rabek, J., Lippmann, R., Haines, J., Zissman, M.: LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In: Recent Advances in Intrusion Detection (2001)
24. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasadm, S., Newboldm, M., Hiber, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: Proc. 5th USENIX Operating systems Design and Implementation Symposium (2002)
25. McAlerney, J.M.: An Internet Worm Propagation Data Model". M.S. thesis, University of California, Davis (2004)
26. Lee, W., Stolfo, S.: Data Mining Approaches for Intrusion Detection. In: Proc. 7 th USENIX Security Symposium (1998)
27. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Proc. 13th Systems Administration Conference, USENIX (1999)
28. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23-24), 2435–2463 (1999)
29. Kruegel, C., Toth, T.: Flexible, Mobile Agent Based Intrusion Detection for Dynamic Networks. In: Proc. European Wireless (2002)
30. DNS Poisoning Summary (March 2005) (accessed July 2005), http://isc.sans.org/presentations/dnspoisoning.php
31. How to Prevent DNS Cache Pollution, Article ID 241352 (accessed July 2005), http://support.microsoft.com/default.aspx?scid=kb;en-us;241352