Using Specification-Based Intrusion Detection for Automated Response

Ivan Balepin [1], Sergei Maltsev [2], Jeff Rowe [1], and Karl Levitt [1]

[1] Computer Security Laboratory, University of California, Davis
Davis, CA, 95616, USA
{Balepin, Rowe, Levitt }@cs.ucdavis.edu
[2] IU8, Bauman Moscow State Technical University,
Moscow, 105005, Russia
SVMaltsev@iu8.bmstu.ru

Abstract. One of the most controversial issues in intrusion detection is automating responses to intrusions, which can provide a more efficient, quicker, and precise way to react to an attack in progress than a human. However, it comes with several disadvantages that can lead to a waste of resources, which has so far prevented wide acceptance of automated response-enabled systems. We feel that a structured approach to the problem is needed that will account for the above mentioned disadvantages. In this work, we briefly describe what has been done in the area before. Then we start addressing the problem by coupling automated response with specification-based, host-based intrusion detection. We describe the system map, and the map-based action cost model that give us the basis for deciding on response strategy. We also show the process of suspending the attack, and designing the optimal response strategy, even in the presence of uncertainty. Finally, we discuss the implementation issues, our experience with the early automated response agent prototype, the Automated Response Broker (ARB), and suggest topics for further research.

1 Introduction

Automated response to intrusions is an exciting area of research in intrusion detection. Development of a system that resists attacks carried out or programmed by another human being can be approached in many ways, including the one in which we teach the machine to beat an attacker in the game of intrusion and response.

Let us begin by formulating the objectives of our work.

1.1 Objectives

With the growing speed and intensity of computer attacks [13] comes the need for quick and well-planned responses. Currently, some of the most intense intrusions are automated. A reliable automated response system, with the right approach, could certainly provide an efficient protection, or a degree of tolerance for all kinds of attacks. However, automated response remains mostly an area of research due to the following issues:

- Primitive response systems that ignore the cost of intrusion and response apply response actions that cause more harm than the intrusion itself
- A large part of commercially available Intrusion Detection Systems (IDS) produces an extensive number of false positive alerts, potentially causing numerous, unnecessary, and costly response actions [12]

Both cases lead to a denial of service to legitimate users of the system.

The objective of this work is to develop a consistent, organized, cost-based approach to automated response that would address these issues. An optimal response would stop the progressing intrusion at early stages, and clean up after it as much as feasible. The scheme described in this work is geared to produce such responses.

We start addressing the problem by considering host-based automated responses. The key parts of our approach are the basis for response decisions (the system map and the cost model), and the process (response selection even in the presence of uncertainty).

Let us briefly summarize the work previously done in the area.

1.2 Related Work

Primitive automated response actions are implemented in some Intrusion Detection Systems (IDS) commercially available today (i.e., re-setting suspicious network connections or “shunning” a certain network address – not accepting any traffic to or from it). [12] However, these actions are rather simple and reflexive by their nature. Even with a limited response arsenal, many practitioners report that they disable the systems’ intrusion prevention/response capabilities due to a high number of false positives from IDS’s which give an incorrect basis for response, and also a denial of service caused by non-sophisticated response strategies.

An interesting research work on Survivable Autonomic Response Architecture (SARA) [8] uses the term *autonomic response* by drawing an analogy with the autonomic nervous system, which automatically controls certain functions of an organism without any conscious input. The authors propose having two separate “loops” of response: a local autonomic response and a global response carried out by the hosts in a system in co-operation. The primary focus of the work is a network with multiple hosts.

Alphatech’s Light Autonomic Defense System (αLADS) relies on control theory when selecting a response [1]. The authors describe it as a part of Autonomic Computing, which, according to them, is an emerging area of study of design and construction of self-managed computing systems with a minimum of human interference.

Alphatech’s work is not applicable to general-purpose computer systems. The work is focused on developing a full-scale solution that has its own profile-based intrusion detection components and is intended to defend a very specific range of systems. The issue of compatibility with existing intrusion detection systems has not received much attention in published descriptions of αLADS. However, Alphatech’s work is of interest to further automated response research, since it is one of the early organized approaches to the problem of quick automated responses.

Another study of network-oriented automated response that relies on Control Theory is currently done at UC Davis [15].

A study by Toth, et.al., [16] proposes yet another promising model for automating intrusion response. The authors suggest approaching the problem of response to network intrusions by constructing dependency trees that model configuration of the network, and they give an outline of a cost model for estimating the effect of a response.

Other significant response works include a thorough consideration of some intrusion detection and response cost modeling aspects by Lee, et.al. [7], a response taxonomy by C. Carver and U. Pooch [3], and Fred Cohen's work on deception [4], which is another interesting perspective on countering malicious activity.

The analysis of related work leads us to the conclusion that the primary area of interest so far has been a computer network that consists of multiple hosts. The idea of responding at a level of a single host has received relatively little attention. Also, we note that despite the efforts to produce a working cost model for a set of protected resources, no well-developed and well-tested model currently exists that guarantees a consistent and fair representation of protected resources, and their true value.

1.3 This Work

This paper has the following remaining sections: Section 2, in which we describe the basis for constructing a response chain; Section 3, in which we discuss an implementation of our model; Section 4, which lists possible directions for future work; and, finally, conclusions in Section 5. The reason for separating the basis for response decisions from implementation notes on our prototype is to attempt to describe a model for host-based response in Section 2 that would not be tied to any particular operating system, and, potentially, could be used for applications other than host-based response.

2 Basis for Automated Response

Several pieces of information are necessary in order to plan a sequence of response actions. For the system we are protecting, we need a clear representation of the most valuable resources and also the underlying resources that provide the basic functionality. The true value of some resources (for example, the TCP/IP network service) is heavily influenced by other resources that depend on them (network is needed by *httpd*, etc.), and we need a clear way to reflect these dependencies before we can decide how to deal with a compromised entity. We also need an organized way to store information about malicious and compromised entities, and to decide how they relate to our key resources. Part of this representation will be highly dynamic, since some entities reflected (processes, etc.) are dynamic; however, a large part of it, such as file structure, program configuration (dependence on files, sockets, etc.), and system configuration, can be determined statically.

We narrow the scope of the problem by noting that transferring an entire computer system to a safe state is a challenging task, and limiting the scope of the problem to returning a set of critical system resources to a reasonably safe and working state.

Resources we will model are anything of value in our system — system subjects and objects, files and running processes, sockets, file systems, etc. We arrange them in two different ways – the resource type hierarchy and the system map.

2.1 Resource Type Hierarchy

It is convenient to group resources by their type, since every such group most likely will have common response actions associated with it. Also, resource types can be arranged in a hierarchy similar to the one on Fig. 2.

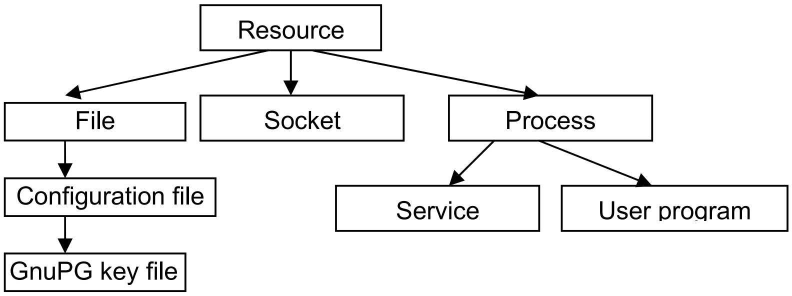

Fig. 2. An example of a response type hierarchy

On a Linux system, for instance, we can subdivide resources into files, sockets, processes, etc., in advance. However, in certain situations a more specific category would be appropriate. Consider a file that contains keys for automatic encryption/decryption of emails using the GNU Privacy Guard software (GPG, [5]). In addition to response action that applies to a more general node in the hierarchy (configuration files, “restore the configuration file from backup and restart the corresponding service”), we define a response action specific to this sub-category (“also revoke and re-issue the keys”).

Since these more specific categories, and their corresponding response actions, depend highly on the system configuration, we cannot define all of them in advance for all systems. We should provide a way for the users of the response system to define new custom categories with response actions tailored to specific resources on the target systems.

2.2 System Map

Although the response type hierarchy is useful for storing general response actions that might apply to a resource, it carries no information about the specific instances of resources on a system, merely their types. Therefore, we need an additional data structure to satisfy the requirements for a decision basis as mentioned above.

We suggest representing the necessary information as a directed graph, which we will refer to as the system map. The vertices of the graph (which we will refer to as map nodes) represent the resources in our system. Besides nodes, our map has node templates and edges.

2.2.1 Map Nodes

The system map contains important nodes of all types — the system's priorities. By "important nodes" we mean "all nodes with a non-zero cost," with cost assigned according to our cost model described in the corresponding sub-section. In addition to the priorities, our map also reflects the underlying basic resources that these priorities need for proper operation. For example, most applications need a working mounted file system with read/write access right in order to operate properly. Therefore, if we have applications A, B, and C that are our priorities, we place them on the map along with the node that represents the file system. We also note that the file system, as an underlying basic resource, does not need to be explicitly specified as a priority itself, since in this simple example it does not have any value of its own. It costs only as much as the priority nodes that depend on it.

Each node holds information about the resource it represents. Namely, we need to know the type of the resource according to our type hierarchy. We also need to know some type-specific information such as path and filename for types and sub-types of "file"; PID, name and owner for type and sub-types of "process", etc. A node also has a cost value associated with it.

Some static nodes might have several node templates associated with them in order to later construct dynamic dependent nodes. Finally, every node has a list of applicable response actions associated with it.

2.2.2 Node Responses

Every node has a list of *basic response actions* that restore its functionality. Currently, we require that this list contain only the actions that completely restore the node to a working state.

The node's list of responses is constructed from response actions that are listed for this type of node and its parent types in the type hierarchy. Each such response action has an *activation condition* associated with it. Referring to the example we have used before, type "configuration file" would inherit a response "restore from backup" from the parent type "file". The activation condition would be, "the node of this type was a target of an illegal *write* system call" or "the node of this type was a target of an illegal *unlink* system call."

Another important property of a simple response action is *what nodes it affects*. Currently, we assume that an action either damages several resources, or does not. If the chosen intrusion detection technique relies on system calls, activation conditions for each response action will be also expressed in terms of system calls. The number of system calls is finite (approx. 200 in a Linux system), and the number of node types is finite (6 in the prototype). Furthermore, there are only a few system calls that are applicable to one type of a resource. Thus, it is feasible to pre-define response lists for every valid combination.

We also complement the node's response list with a response "take no action". That is an essential response alternative that has a certain cost, just like other responses, and by including it, we will ensure that any response action we take is not more expensive than the intrusion itself.

Therefore, an entry in a node's response list has three fields:

- the action itself (a Linux command, etc.)
- the activation criteria
- the list of nodes the action damages

2.2.3 Map Edges

Edges on our map represent dependencies between the resources. If an edge is directed from node A to node B, it means that A provides some service to B, B depends on A, and, most likely, A produces information that B consumes.

However, it does not seem feasible to attempt to trace information flow through our map, since it contains nodes that are often times not comparable (for example, file systems and sockets), and also nodes that obscure information flow (if node A reads from node "file system", node B writes to node "file system", there is not necessarily an information flow from B to A). Therefore, we do not use our map for intrusion detection. For all information about the intrusion we rely on some detection technique.

The true value of the map edges is that they allow us to properly carry out single response actions that involve several nodes ("restart the service that corresponds to this configuration file", i.e., the service that consumes information from the file). Also, the edges allow us to collect information about the nodes that depend on a certain node, therefore allowing us to calculate the dynamic cost of the node in our system.

Relationships between the nodes can be specified with greater detail, such as "node A writes to node B that often", or "node A writes to node B with probability N." However, for our purposes, it is sufficient to only reflect the fact that one node provides services to another node, and therefore, the latter depends on the former. Also, some authors model dependency alternatives (node A depends on node B *or* node C) [16]. From a standpoint of resources of a single host, this is a relatively rare situation, so we will not consider it here.

2.2.4 Constructing the Map

As we have mentioned before, the map will have a static part, which will consist of nodes that can be produced by static analysis of our priority resources when no processes are running. The static part of the map will have information about objects of the system, but not subjects. Operation begins just with the static part of the map. As the system runs, dynamic nodes are added.

In our design, there are five ways we can add a node to the map. Static nodes are added to the map upon upgrades/reconfigurations of the protected system.

For the dynamic part, we propose to add new nodes for every subject or object mentioned in the incident alert from the IDS that was not previously on the map. Such nodes would be assigned cost 0, since they were not included in the list of priorities, and they get assigned the most specific type from the type hierarchy that we are able to automatically determine. Consequently, the node will have a response list that corresponds to its assigned type.

Also, as we will describe in later sections, sometimes we will be able to classify a whole group of subjects as malicious, whereas only a few of them might have been explicitly mentioned in alerts. Such situations can occur, for example, when a malicious process caused an alert, and immediately produced a number of child processes that have not yet done anything illegal themselves. We will put the whole related group of subjects on the map and treat them just like the nodes mentioned in the alerts, despite the fact that only a few were mentioned in the alert. Then the whole group gets marked as suspicious, or "contaminated".

Finally, we will have some dynamic nodes that will represent our priority resources. Often times, a resource in general can be mapped to several nodes. For example, a "web server" resource encompasses the executable file, a number of running processes, and dependent resources (configuration files, sockets, etc.). At the time of static analysis, we will not have a running instance of a web server; however, we can get most of information about the web server process node at that time. Therefore, with every important executable we create a set of templates that will characterize the subjects and objects later to be produced by running the executable file. A node template is a prototype for building nodes that has all the information in place except for the type-specific information (like PID or filename) that gets filled in upon use of the template.

2.2.5 Properties and Benefits of the Map

The map has only a few static and dynamic nodes that are critical to the system's operation. They are not updated periodically; rather, we update them only when significant events happen (alerts for dynamic nodes and system re-configuration for static ones). Therefore, if our system runs for a long time without getting attacked, the map will not be updated in order to minimize the overhead.

The nodes on the map can be of very different nature, so they cannot always be compared directly (for example, file systems and processes).

Let us illustrate some properties of the map with a small example. Suppose, we have a Linux system equipped with System Health and Intrusion Monitoring IDS (SHIM, [6]) that has been compromised, and now has an active malicious process A that was produced by a program B which is not supposed to make system calls from the *exec* family. Process A has its parent's specifications imposed on it by SHIM. Suppose then that process A produces process C, and process C writes to a file. Since SHIM would promptly alert our response system about A, B and C being involved in several illegal *exec* system calls, the whole family would appear on the map, and would be marked as malicious. As far as the file that C has written to, if specifications for A allowed such behavior, then we would not get an alert about the file write, and, therefore, would not reflect that fact on the map. However, if it was not legal according to A's specifications and the system policy, we would get an alert about a possibly contaminated file, place the corresponding node on the map, and plan our response strategy with that alert in mind.

As we have shown above, our map contains all necessary information about our priorities, and resources they need to operate. The map also will reflect information about malicious entities, and their relation to our priorities. The map, as we have described it, gives us a solid basis for designing an intrusion response strategy.

2.3 Cost Model

In most cases, when deciding on response to a malicious action, there will be several response actions with activation criteria matching the current situation. We solve the problem of comparing these alternatives and selecting the optimal one by introducing an action cost model. The cost model helps us pick the best response and also ensure that we don't cause denial of service to ourselves by performing responses that are more harmful (i.e., more costly to us) than the intrusion itself.

Our cost model is based on numerical cost values associated with every map node. Designing a cost model that allows us to quickly associate a number with a resource and to precisely reflect the value of that resource is a difficult task. Most of the attempts to produce such a model left it up to the system administrators to determine cost values for their resources. Although it is true that only the system's owner, familiar with its configuration and primary functions, can point out the true value of the resources, it is very hard to assign the cost values in a consistent manner that would always guarantee optimal response without exhaustive testing of the system. In our implementation, we rely on ordering the resources by their importance to help produce a cost configuration that would yield an optimal response.

There are only a few priority nodes that have an actual cost value in our model. For example, let us consider a system with only one such priority – the web server. In the static part of the map it is represented with the executable file of the web server. There will be a static node for the file itself, and it will have a cost of 0. The static node for the executable file will, however, have a template for web server processes to be created, and that template will have a cost value associated with it. In our model, all process nodes that get created according to that template, will share an equal fraction of the template's cost with existing processes. For example, suppose the system's owner has estimated that the web server has cost x. When there is no web server running, the executable file will have no cost value. If one instance of *httpd* gets started, its node will get assigned a cost value x. If y nodes of *httpd* get created, each will get a cost value of x/y.

A static node can also get an explicit cost value assigned to itself, and not to its templates; or it might not even have any templates. For example, some files might be indicated as a priority, even though they are not used by any subjects of that system.

Cost-wise, another category of nodes on our map is the underlying service nodes. Most likely, these nodes will have a zero cost of their own. However, any harmful action on these nodes will also affect the costly resources that rely on them, and by reflecting these dependencies on the map we will take into account the true value of the underlying services.

Finally, we have all the resources that were not put on our map as a priority resource or its dependency. We assign all such resources cost 0; if they become malicious or get involved in an incident, they are put on the map, and a response action that affects these 0-cost nodes even in the most dramatic way will not be harmful for the system in general.

Once we determine the cost values for our map nodes based on these factors, we then can associate a cost value with any *action* that an intruder or the response system takes.

We define *the cost of an intrusion action* as the sum of costs of map nodes, previously in a safe state, that get negatively affected by the action. We define the *benefit of a response action* as the sum of costs of nodes, previously in the set of affected nodes, that this response action restores to a working state. Finally, we define the *cost of a response action* in terms of costs of the nodes that get negatively affected by the response action ("lost to the intruder," or not functioning properly). The goal of a response system is to carry out the response sequence that yields the maximum benefit at the minimum cost. We note that such an approach does not emphasize transferring the system to the ultimate safe state, or completely recovering from an intrusion, since there are situations in which these goals would be much more costly than the intrusion itself. With our approach, we are, however, guaranteed to come up with a response strategy that is optimal for the current situation.

2.4 Response Selection

Once we have the whole picture of the intrusion, our goal is then to "win" the resources on the "contaminated" side back. We start by listing all response alternatives at every contaminated node whose activation condition matches the intrusion. The goal of response selection is to build a response action sequence that will have one action out of a list of every contaminated node. That way, we ensure that every contaminated node is addressed. As mentioned before, an optimal response action is the one that yields the maximum benefit at the minimum cost. We then assume that a response sequence (response strategy) is optimal if it consists of response actions that are optimal for every node. Therefore, if we have the complete picture of the intrusion, we can build the response chain from optimal responses at every node, and then carry it out.

2.4.1 Managing Uncertainty

Sometimes we might encounter situations where we do not know for certain what the intruder has exactly done. For example, suppose the capabilities allowed the intruder to perform a *write* call on a file, which is illegal according to the current system policy. The file could have been overwritten, appended to, or erased completely (overwritten with an empty string). In certain situations, response actions, and their cost, may vary depending on what has really happened. Then we turn to decision theory, which provides well-defined ways to construct the response plan, for different requirements in presence of uncertainty.

The possible results of a *write* call would be over-written data in the file, data appended to the file, or data completely erased from the file (the latter being a special case of the first one). This allows us to list the possible system states. Every one of these states will have a potential damage value and a probability associated with it. Now, using the decision theory convention [10], we can describe the situation with the following "gain matrix":

	Π_1	Π_2	Π_3	Π_4	Π_5
A_1	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
...	...				
A_N	a_{N1}	a_{N2}	a_{N3}	a_{N4}	a_{N5}
Q	q_1	q_2	q_3	q_4	q_5

where Π_i are the possible states, q_i are the probabilities, and A_i represent the response alternatives. a_{ij} in this matrix, again, represents the usefulness, or benefit, of using the ith decision in case of a *j*th sub-state. This value can be estimated as:

$$a_{ij} = -c_i - \left(-\varepsilon_{ij}\right)^{\gamma} B_j . \tag{1}$$

where B_j is the potential damage of a sub-state,

c_i – response cost,

ε_{ij} – efficiency(benefit) of response *i* in sub-state *j*, and

γ is 0 if ε_{ij}=0, 1 otherwise.

Considering the above parameters, we observe that the greater the value of a_{ij}, the more useful the corresponding response alternative will be in the corresponding state.

We define the risk of losing in a particular game situation (r_{ij}) as the difference between the player's gain for strategy A_i for conditions of Π_j, and the player's gain for the strategy he would have chosen, had he known the conditions of Π_j. It is clear that had the player known the system state and its conditions in advance, he would have chosen the strategy that yields the maximum gain in its matrix column (m_j). According to our definition,

$$r_{ij} = m_j - a_{ij}, \text{ where } m_j = \max_i a_{ij} . \tag{2}$$

Defined in this way, the concept of risk also reflects how favorable a given state is to us. Consequently, a risk matrix constructed similarly to the gain matrix, gives us a more complete picture than the gain matrix.

Relying on probability significantly simplifies the decision making process, especially if we can produce relatively accurate probability estimates using the system history, general knowledge, anomaly analysis tools, etc.

A promising way to eliminate the uncertainty, or, at least estimate the values of probability of a certain intrusion sequence, is monitoring the system for a long period of time and building a profile for important resources. For that, machine learning techniques can be used; also, much can be drawn from the anomaly-based and misuse-based intrusion detection techniques [2]. We discuss these suggestions in more detail in Section 4.

Let us take mathematical expectation of probability-based gain $\overline{a_i}$ to be the effectiveness criterion *W* that we obviously would like to maximize.

$$\overline{a_i} = q_1 a_{i1} + q_2 a_{i2} + \ldots + q_n a_{in} . \tag{3}$$

The optimal strategy is the one that yields the maximum $\overline{a_i}$ in the gain matrix. It would also yield a minimum average risk based on the risk matrix.

Special care must be taken to accurately estimate probability. Pure probability, as a statistics-based value, might not always be available. In that case, it can be subjectively estimated. Certain events might be more likely than others according to the system logs. There are several techniques available that help us quantify these subjective estimates.

For cases in which we have no statistical information for the system states, we can assign equal probabilities to each possible state, i.e.:

$$q_1=q_2=\ldots=q_n=1/n. \tag{4}$$

This approach is called Laplace insufficient reason criterion ([10]).

For another approach, we assume that we can order possible system states by their likeliness. In order to represent the probabilities in this case, we can use a converging arithmetic series:

$$q_1:q_2:\ldots:q_n=n:(n-1):\ldots:1. \tag{5}$$

where:

$$q_i=\frac{2(n-i+1)}{n(n+1)} \tag{6}$$

We can also rely on expert estimates.

If we manage to completely eliminate uncertainty in some situations, the probability values for the determined system state becomes 1, probabilities of all other states become 0, the matrix turns into a single column, and decision making becomes trivial.

2.4.2 The Optimal Decision Criteria

There are several methods for selecting the decision criteria in the decision theory ([10]). In the Minimax risk criterion (Savage criterion) we select the strategy from the risk matrix that provides us with the minimal risk value under the most unfavorable conditions. The efficiency *W* is then estimated as $W=\min_i \max_j r_{ij}$. The Minimax risk approach allows us to avoid making the high-risk decisions. The Maximin criterion favors strategies with the largest minimal gain (with *W* defined differently, see [10]). The Hurwicz criterion is neither pessimistic nor optimistic. Risk-based criterion is analogous to Hurwicz

Selection of criterion and its parameters is subjective. It is useful to analyze the situation using various approaches. If majority of criterions indicate that a certain strategy is optimal, it should certainly be selected. Should several different criterions suggest different strategies, it is up to the system owner to select (or pre-select) the right strategy based on the fact that some criterion might be preferred over the others.

3 Implementation

We have implemented several concepts mentioned in the previous section in a prototype response system, the Automated Response Broker (ARB). ARB is developed for Linux, and it relies on SHIM for detection. Let us briefly mention why we chose SHIM for that role.

3.1 Intrusion Detection: SHIM

SHIM is specification-based. It relies on the Generic Software Wrapper Toolkit (GSWTK,[9]) for all information about the system calls. SHIM does not try to recognize an attack as a whole. Instead, it relies on a set of specifications (for programs, or protocols, etc.) that reflect the system policy.

SHIM addresses a large part of intrusions by enforcing specifications for privileged Linux programs. System calls of interest are reported by the GSWTK, and then classified as legal or illegal according to the specifications, with an alert being issued for the latter.

SHIM is a great vehicle for testing our automated response scheme. Such a fine event granularity allows us to catch the exact system call that started the intrusion. Also, the fact that SHIM does not need the whole intrusion to recognize its signature, allows it to catch unknown intrusions, and intrusions that are still in progress. The last feature also gives us a chance to stop an intrusion in progress by responding to the first few steps of it that have been detected.

The underlying assumption about SHIM that we make is that it always promptly detects and reports all intrusions. Also, SHIM and GSWTK give us a capability to check if a system call is legal before it is executed. However, such a mode of operation causes a large overhead for every system call, and does not seem feasible.

3.2 Map Implementation

We build the map starting with a set of nodes we want to protect. It is the set of all programs that are constrained by SHIM (regardless of whether they are among our priorities; the cost will reflect that fact), and several nodes for resources that might not be constrained by SHIM, but the system owner wants to protect as well.

The type hierarchy is constructed upon installation of a system. It does not have a dynamic part and it does not change, since it simply contains information about the types of nodes, not the nodes themselves.

In ARB, the type hierarchy is constructed in C++. While it might be sufficient for experiments and testing, obviously a more convenient interface for configuring the type hierarchy is needed. Currently, we experiment with XML for type hierarchy definitions. XML so far has proved to be powerful enough to express all the information necessary, and there is an abundance of tools for parsing the type hierarchy defined in XML into our program.

Below is an example of a response list of a configuration file node. Event name and target constitute the activation condition for the action. The victim tag marks the damaged nodes.

```
<actions>
    <event name=”chown” target=”self”>
          <action>restore_attributes;</action>
          <action>kill_offender;</action>
          <victims>offender</victims>
    </event>
    <event name=”chown” target=”self”>
          <action>delete_self;</action>
          <action>kill_offender;</action>
          <victims>self</victims>
          <victims>dependents</victims>
          <victims>offender</victims>
    </event>
    <event name=”write” target=”self”>
          <action>
                restore_from_backup;
                fire_event(“httpd”,”restart”,true);
          </action>
    </event>
    …
</actions>
```

Should XML fail to be descriptive enough for the task, a new domain-specific language (DSL) will be designed for describing the type hierarchy.

Similarly to the type hierarchy, the map itself is constructed manually as a collection of C++ data structures. We are currently experimenting with more flexible ways to define a map, such as, again, XML or a new DSL.

As mentioned above, all components of a system in our prototype are determined manually. However, some of them can be pre-defined for most systems; some can be determined by automated analysis upon installation or re-configuration. The ultimate goal is to let the ARB user specify just the custom types, responses to custom types, and the system’s priorities. The remainder of the map (such as the basic types, underlying service nodes, all dependencies and templates) can be determined automatically. We list the requirements for automating the map construction and problems associated with it in the future work section.

Node response lists are constructed from the type hierarchy. The set of response actions that are implemented, or will be implemented in the prototype include: *delete a file, restore a file from backup, restart a service, change permissions, kill process(es), reboot the system, block a connection, re-configure a firewall rule, unmount a file system, change the owner of the process(es), start checkpointing, slow down the process(es), roll back to a checkpoint, return a random result, perform a random action, operate on a fake file, tunnel the process(es) to a sandbox, operate on a fake socket*.

3.3 Node Costs

The most difficult task of any implementation of a response system is performing a consistent cost assignment that reflects the true value of resources. This part of map construction cannot be completed in advance, or even automated, since it needs input from the owner of the system. Currently, we approach the problem by first manually ordering the key resources of the system, so that the resources (R_i) are listed in the following form:

$$R_0 < R_1 = R_2 = R_{3\,\ldots} < R_{i-1} < R_i \qquad \textbf{(7)}$$

The least important resource gets assigned priority 1, and the priorities of all other (more important) resources are approximated as *N* times the priority of the next less important one:

$$\text{Priority}(R_j) = N * \text{Priority}(R_{j-1}), \qquad \textbf{(8)}$$

where *N* is an approximate value and is determined experimentally. Finally, for convenience, we obtain a cost value C_i for a resource R_j from priority values according to the following formula:

$$C_i = 100*\text{Priority}(R_i)/\Sigma\text{Priority}(R_i), \qquad \textbf{(9)}$$

where $\Sigma Priority(R_i)$ is a sum of the priorities of all resources.

Currently, the process of assignment is completely manual. The cost assignment method described above is only an approximation of the real resource costs. Work is being done on improving the method to ensure consistency of the cost assignment.

3.4 Damage Assessment and Response Selection

Let us say ARB received an alert about some malicious actions involving several nodes on our map. Currently, ARB reads alerts from a socket that SHIM writes to. A closer form of integration with SHIM is being developed, since the current implementation is sufficient for evaluation of response, but is vulnerable to attacks.

First, we need to stop the intrusion, if it is still in progress. The map is partitioned into a set of nodes that are affected (or might have been affected) by the incident, and a set of nodes that are not dependent on any in the first set, and therefore, not affected by the incident.

Upon an alert that, say, mentions only one subject, the damage assessment procedure of ARB puts all ancestors and offspring this subject may have on the map independent of further alerts. That allows us to freeze the intrusion in progress before the children attempt to perform further malicious action, since having a suspicious process as a parent already gives us a right to mark a child process as suspicious as well, without waiting for further alerts. We freeze the intrusion by temporarily suspending the contaminated processes (by sending them a *kill -19* message).

ARB operates with a concept of an incident. Alerts are grouped to form a single incident if they report subjects from the same family as suspicious. ARB considers the damage assessment procedure completed when it constructed and froze the entire family of suspicious processes. All new alerts are treated as parts of a new incident. The testing of ARB that we have done so far indicates that such approach allows us to clearly separate individual incidents, freeze an incident, assess the damage, and carry out response actions.

Upon completion of the damage assessment procedure, we have the suspended intrusion, the frozen suspicious processes, and the complete picture of an intrusion in form of the partitioned map.

Finally, the response strategy is built and carried out, as described in the previous sections.

3.5 Example

Let us demonstrate how ARB carries out the entire process of response selection with an actual example. We will consider a classic vulnerability in the RedHat Linux 6.2 *dump* utility [11], which examines the files on a file system, and determines the ones that need to be backed up. These files are copied to a disk, tape, or other storage medium. The dump utility depends on the environment variables *TAPE* and *RSH*.

The goal of the dump exploit is to set the *RSH* environment variable to an executable file that will be executed with suid root privileges. File */bin/bash* is copied to */tmp/rootshell*, and the root shell is executed.

Specifications for the *dump* utility are provided by SHIM. According to them, *dump* is allowed to make few system calls: *open* and *read* certain files, *fork*, and *connect*. Consequently, when this intrusion happens on a system that runs SHIM, but is not protected with ARB, the system administrator will get several alerts. There will be an alert about *dump* copying the shell executable to the */tmp* directory. Another key alert is issued when *dump* executes file */tmp/rootshell*. The last alert will be issued when the attacker uses the obtained root shell to issue the *open* and *write* system calls to the target. The target in our example will be the file *secring.gpg*, which contains the keys the GPG software uses for encryption/decryption.

Let us first show the relevant parts of the map before the intrusion begins (Fig. 3).

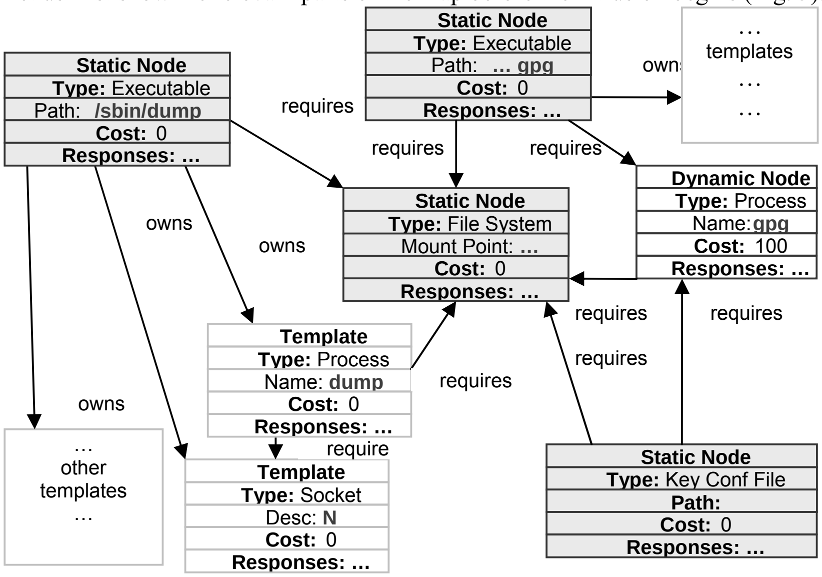

Fig. 3. The map of a part of a computer system before an intrusion. Only a few essential nodes are shown.

The map was built according to the type hierarchy on Fig. 2. According to the map, our only priority in the entire system is the *gpg* program that encrypts and decrypts email messages. However, we also put *dump* node on the map, since it has SHIM specifications.

Experiments with the ARB prototype showed that it takes a variable period of time for SHIM to issue an alert, and for ARB to receive it and process it. For certain test cases with favorable conditions, that period was short enough for ARB to freeze the entire attack right after the first alert. For test cases under the least favorable conditions, however, ARB completed the damage assessment procedure only after the attacker already had access to the root shell.

Regardless of the current conditions, our goal is to stop the intrusion and clean up after the actions that already happened. The system map for the worst case that has been observed is shown in Fig. 4.

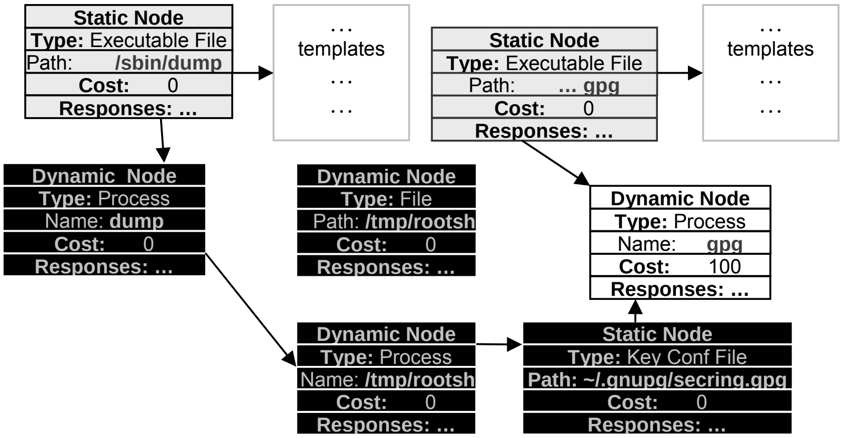

Fig. 4. The map of a part of a computer system after ARB has stopped an attack.

According to the new map, four nodes are contaminated as the result of the intrusion. A node for */tmp/rootsh* appeared on the map because the file was involved in an illegal file copy by *cp*. However, the *cp* process itself is gone by the time ARB completed damage assessment, so it is not reflected on the map.

ARB starts building the response sequence addressing node by node, in arbitrary order. The *dump* process node is an issuer of an illegal *exec* system call, so ARB chooses the most efficient response – killing the process – since the value of nodes affected by the response is 0. The */tmp/rootsh* node was created as a result of an illegal *creat* system call, and it does not have any cost or dependencies. The matching response would be to remove the file. Finally, the response for the secring.gpg file is selected as follows. Several response alternatives apply to the file, including deleting it or restoring it from the backup. Deleting the file would certainly damage it. By using the map, we detect that the *gpg* process depends on the file; therefore, deleting the key file would damage the file and the process, and the cost of such response would equal to the sum of affected nodes – namely, 100 points. Another alternative with an activation criterion that matches a *write* system call is restoring the file from backup, with a cost of 0. We select the second alternative as the least expensive one. Another matching response action is "restart the corresponding service(s)," and it was inherited by the custom type "key configuration file" from general type "configuration file." By using the map, we determine the corresponding service to be the *gpg* in this case, and we restart it with a restored key file.

In this case, SHIM has not indicated that the content of the file has been read. Therefore, the response alternative "reissue all keys" does not apply, and we do not re-issue the keys.

3.6 Experience with ARB

The ARB prototype was tested for several well-known attack scripts. Work is in progress to extend it and test it with the broadest range of other intrusions. ARB can only be run on Linux kernel 2.2.14, since the current version of GSWTK relies on that kernel version, and the current version of SHIM relies heavily on GSWTK. As we mentioned before, the map in ARB is built manually for only a subset of all resources that really should be on the map.

The current version of ARB does not handle uncertainty in intrusions. It does, however, successfully freeze the set of test attacks, stop them, and respond to them. The attacks we handle include the two examples from this paper.

The prototype so far has forced us to re-design our original approach to automated response greatly, and posed several new problems, which were not obvious before. One such problem was the fact that at first we did not define when the damage assessment procedure is complete, and we can actually start deciding and carrying out a response. In order to resolve the issue, the concept of an incident was introduced in the prototype.

We currently continue to work on the prototype, and we expect promising results from the future work with ARB.

4 Future Work

4.1 Automating the Map Construction

First, since with SHIM all malicious actions that involve a map node can be expressed as Linux system calls, and the number of Linux system calls is relatively small, we can partially automate the generation of nodes' response lists. For a new type of a node, we list all applicable system calls that this node can make as an activation criteria. Then, we either borrow the corresponding response actions from the type above in the hierarchy, or ask the system user to define a response action and the damaged nodes. Then we construct a list of applicable system calls that target this type of a node as activation criteria, and obtain the corresponding response and damaged node information in a similar manner. Therefore, we can simplify the task of constructing response lists by guiding the user through the process and producing the output in some convenient format like XML.

Also, construction of the map itself and analysis of node dependencies can be mostly done automatically. When constructing a map, we can rely, for example, on the program installation package (i.e., Linux RedHat Package Manager information); the program's source code (when available); documentation (*man* pages); etc., for dependency information about opened files, sockets, pipes, inter-process communication, etc. We are also currently working on learning program dependency data from execution traces. Designing a tool that would assist a system's user with map construction presents an interesting implementation task.

4.2 Learning the Configuration

The problem of assigning the true costs, determining the actual relationships between the nodes, and testing the efficiency of ARB can be determined experimentally. As the next step of this work, the following experiment will be carried out.

One Linux system (*defender*) will be equipped with SHIM, ARB and several protected valuable services. Another system (*attacker*) will continuously generate attacks targeting every node of the *defender*. A third system (*referee*) will record the outcomes and help restoring the *defender* after successful attacks.

At first, the attacks will be run under supervision. As a measure of efficiency, uptime and performance of a certain service under attack will be measured with ARB protection, and compared to its uptime and performance under the same attack without the protection.

Once such a setup is implemented, it can be used to analyze the flaws in ARB response strategies to determine the "blame" for successful attacks. Furthermore, node costs and degrees/probabilities of relationships can be represented as weights in a neural network, and some machine learning algorithm (backpropagation, other gradient descent methods, etc.) can be applied to continuously improve the ARB setup, possibly with much of the supervision done by the *referee* system.

4.3 Other Directions

Introducing nodes of type CPU, or memory, or user may allow us to model and respond to denial-of-service attacks. We did not consider the topic in this work, but it seems promising; especially when the intrusion detection technology will provide us with ways to clearly identify denial-of-service attacks.

Storing information about past intrusions and incorporating that knowledge in response is also promising. For example, a large number of attacks in a small period of time might cause the system to take extra response measures targeted at preventing future intrusions rather than responding to ones already in effect. Also, we might design a set of more strict specifications for the privileged programs that would reflect a stricter system policy in response to a large number of intrusions. Another option would be to implement a "pre-emptive" mode as a wrapper in GSWTK: all system calls would be checked in advance, and not carried out if illegal. This mode of operation would cause a large overhead for every single system call; however, it might be useful when trying to counter particularly severe types of intrusions. Again, the cost of switching to such mode has to be carefully weighed.

Currently, our response model does not consider actions that partially restore a node, and it assumes that an action either damages resources, or does not. Considering actions that only partially restore resources and introducing a degree of damage also deserve consideration for further work.

Another interesting research direction would be to attempt to combine our host-based approach and network-oriented response mentioned in Section 1, to design a network-wide response system that possibly might be based on single host components, such as ARB, cooperating with each other to protect the entire network.

Finally, in our opinion, the most exciting future work option is combining a specification-based IDS, features of anomaly- and misuse-based IDS's and the requires/provides model of intrusions [14] to form basis for response decisions. With SHIM being a "low-level", system-call oriented IDS that ignores the intrusion as a whole, and focuses on individual constraint violations instead, it is able to catch violations that have never been seen before, and cannot be detected with signature-based detection systems; whereas signature-based systems can see farther ahead than SHIM, since they have a signature of the entire intrusion.

In a situation where we receive several SHIM alerts (which represent the first few steps of an intrusion), we can use our system map to calculate the capabilities of the attacker, describing them in JIGSAW [14], and also browse the signature database for all signatures that, at least partially, match the current intrusion. By using some historical data from an anomaly-based system we can determine probability of each intrusion path (signature), and initiate a game with the intruder. By winning such a game, we will be able to prevent complex intrusions instead of responding to the ones that are already in full progress.

5 Conclusion

In this work, we stated the problems associated with automated intrusion response, and began addressing them.

The system map and the resource hierarchy provide a basis for response. The damage assessment procedure and response selection that accounts for uncertainty produce the optimal response strategy.

The current implementation of these ideas – ARB – successfully responded to several host attacks. Work is being done to improve ARB and measure its efficiency and performance.

References

1. Alphatech: ALPHATECH Light Autonomic Defense System, http://www.alphatech.com/secondary/techpro/alads.html (last accessed June 30, 2003)
2. Amoroso, E: Intrusion Detection: an introduction to Internet surveillance, correlation, trace back, traps, and response, Intrusion.net Books, New Jersey. (1999)

3. Carver, C.A, Jr. and Pooch, U.W.: An Intrusion Response Taxonomy and its Role in Automatic Intrusion Response, Proceedings of the 2000 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY. (6-7 June, 2000)
4. Fred Cohen & Associates, Deception for Protection, http://all.net/journal/deception/index.html (last accessed June 30, 2003)
5. Free Software Foundation, Inc., The GNU Privacy Guard, http://www.gnupg.org (last accessed June 30, 2003)
6. Ko, C.C.W.: Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, Ph.D. Thesis, Davis, CA. (August 1996)
7. Lee, W., Fan, W., Miller, M., Stolfo, S., Zadok, E.:Toward Cost-Sensitive Modeling for Intrusion Detection and Response, Journal of Computer Security, Vol. 10, Numbers 1,2 (2002)
8. Lewandowski, S., Van Hook, D., O'Leary, G., Haines, J., Rosse, L., SARA: Survivable Autonomic Response Architecture, DISCEX II'01, Anaheim, CA. (June 2001)
9. Network Associates Laboratories: Secure Execution Environments/Generic Software Wrappers for Security and Reliability, http://www.networkassociates.com/us/nailabs/research_projects/secure_execution/wrappers.asp (last accessed June 30, 2003)
10. Raiffa, H.: Decision Analysis: Introductory Lectures on Choices under Uncertainty, Addison-Wesley, Reading, MA. (1968)
11. RedHat, Inc.: Red Hat Security Advisory RHSA-2000:100-02, http://rhn.redhat.com/errata/RHSA-2000-100.html (last accessed June 30, 2003)
12. SecurityFocus, Mailing List: FOCUS-IDS, http://www.securityfocus.com/archive/96/310579/2003-02-03/2003-02-09/1 (last accessed June 30, 2003)
13. Staniford, S., Paxson, V., Weaver, N.: How to 0wn the Internet in Your Spare Time, Proceedings of the 11th USENIX Security Symposium (2002)
14. Templeton, S., Levitt, K.: A requires/provides model for computer attacks. In Proceedings of the New Security Paradigms Workshop, Cork, Ireland. (September 2000)
15. Tylutki, M.: Optimal Intrusion Recovery and Response Through Resource and Attack Modeling, Ph.D. Thesis, Davis, CA. (September 2003)
16. Toth, T., Kruegel, C.: Evaluating the impact of automated intrusion response mechanisms, 18th Annual Computer Security Applications Conference, Las Vegas, Nevada. (December 9-13, 2002)