

Simulation Analysis of a Notional Intrusion Tolerant System

Abstract

Present Intrusion Detection Systems (IDSs), and systems likely to be fielded in the foreseeable future, suffer from false alarms, missed detections, inaccurate reports, and delays between attack and detection. Emerging heterogeneous sensor correlation technology has some potential to reduce false alarms and improve accuracy and sensitivity. Automated response components must then consider the imperfect picture inferred from such systems, and select response actions accordingly. We present a simulation model for analyzing costs and benefits of response actions in an intrusion tolerant system. The simulated system consists of application and proxy servers, redundant in capability but diverse in implementation so as not to be vulnerable to the same attack. We model the efficacy of intrusion detection systems (IDSs) and emerging IDS correlation technology as providing a probabilistic assessment of the state of system assets. Actions are selected in response to diagnoses of adverse states, which may be inaccurate or false alarms. The system state evolves as a Markov process conditioned upon the present state, imperfectly observed, and the action selected. Actions have an associated present cost but may impart some future benefit in terms of effecting system evolution to a higher value state. The objective is to choose actions so as to maintain high system value in the presence of attacks.

Keywords: Intrusion tolerance, automated response, simulation analysis

Acknowledgments

This research is sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the author(s) and do not necessarily reflect the views of the supporting agency.

Introduction

Presently fielded Intrusion Detection Systems (IDSs) suffer from high false alarm rates, missed detection, inaccurate reports (alerts related to real attacks, but incorrectly classifying the attack), and time delays between attack and detection. Emerging technologies for correlating heterogeneous IDSs have some promise of improving sensitivity, accuracy, and false alarm rates. Nonetheless, as we consider the efficacy of such systems, and response mechanisms based on these systems, we must accept that response actions will be based on an uncertain and possibly stale picture of the monitored system. Realistically, we cannot definitively assert that a server is or is not compromised, for example. Rather, we can at best provide some estimate of the probability that this is the case. Particularly in the case of heterogeneous sensors, we may have the situation that sensors give conflicting alarms and some sensors give no alarms at all. For this reason, we consider a system based not on perfect detection and prevention of attacks, but toleration of attacks when the system state is probabilistically described. In this paper, we consider a system with redundant servers (redundant in capability, but diverse in operating system) to achieve intrusion tolerance. This system is instrumented with intrusion detection and correlation components, including asset distress monitors, and automated response. The selection of response actions will be based on procedures for decision under uncertainty.

When we consider coupling IDSs with automated response and recovery, therefore, we do so at some risk because the best picture of the world has some chance of being inaccurate or incorrect. Furthermore, it is important to recognize that actions typically have an associated cost, but provide some benefit. For example, restarting a suspect server incurs the cost that the server is unavailable for some time interval, but provides the future benefit that the server will provide maximum value when it is restarted. Here, we present a framework based on Markov Decision Processes (MDPs) to select response actions so as to maximize system value over time. The system state is probabilistically described, and its evolution depends on the state and the action taken. The action incurs an immediate cost but provides some expected future benefit. Of course, actions taken in response to false alarms incur a cost but provide no benefit. The purpose of our

simulation system is to explore these tradeoffs and dynamics in an intrusion tolerant system.

The cost functions we consider are the fraction of requests that are dropped or receive an invalid reply. Because there are finite delays between detection of and response to adverse conditions, it is impossible to reduce the fraction of dropped requests and invalid replies to 0. It is reasonable to add these with some suitable scaling (for example, assume invalid replies are 10 times as costly as dropped requests). Actions taken impact this cost directly; for example, a reboot action takes a host offline for an interval of time during which arriving requests are potentially dropped. Agreement protocols compete with client requests for server capacity, leading to a greater likelihood of queue overflow. At the same time, choosing no action can be costly as well; for example, neglecting or delaying a reboot request on a compromised host can result in invalid replies from this host.

Notional Architecture of an Intrusion Tolerant System

The system we are modeling consists of a number of proxy servers behind a firewall mediating client requests to a bank of application servers. The proxy and application servers are redundant in capability, but are diverse with respect to platform, operating system, and implementation. Each host in the server bank is in one of three states: UP, DOWN, or COMPROMISED. Hosts in the UP or COMPROMISED state are able to respond to client requests. The distinction is that in the COMPROMISED state, the host delivers invalid replies to client requests. In our typical system value function, an invalid reply is more costly than a dropped request. At present, we do not consider attacks against the tolerance proxy bank, but abstract this as a system management function. Hosts respond according to their true state, but the manager knows this state only by inference from the detection components. Four types of detection assist the manager in its diagnosis. The first is direct detection of the attack. It potentially detects all classes of attacks and is usually the fastest, but may miss the attack or generate a false alarm. The second is symptomatic asset distress (such as the blue sensor of [Valdes-Skinner]), where the manager detects a sudden request queue length explosion at a host. Note that this will

not detect a successful compromise. The third is the agreement protocol, where the manager issues a request of all application servers and compares the replies. This can detect a compromise attack, but possibly after a significant time interval. The fourth is the class of high-accuracy, high-overhead monitors that are deployed and removed by the manager; these provide greater accuracy, but degrade server capacity.

A request is served by a particular host with probability equal to the probability that the host is up or compromised (in the latter case, the reply will be invalid). A request is dropped if it is not served by any of the available hosts. This will be the case if all hosts are down, but may happen randomly if hosts are in a degraded state. Requests are also dropped if the overall number of active requests exceeds a queue limit. It is important to note that system response depends on the true system state, but this state is known imperfectly and with delay to the manager. The manager's objective is to choose response actions so as to maintain system value (there is a cost associated with dropped requests, and a higher cost associated with an invalid reply) given its imperfect view of the system state.

System Evolution as Poisson Processes

Our simulation model is continuous in time and discrete in state, with multiple event streams altering the system state. We simulate events as Poisson processes [Karlin], in which inter-event times have an exponential distribution with a parameter interpreted as the rate of events per unit time. The distribution is "memoryless" in the sense that the residual time to the next event is exponential with the same rate parameter. In other words, if buses arrive according to a Poisson process, and we arrive at a bus stop, the time to the arrival of the next bus is exponential with a given rate parameter, regardless of the time of the arrival of the last bus.

For our purposes, this leads to a tractable implementation in that we do not usually need to track the evolution of individual objects in our simulation, but the count of such objects and in each state of interest and the rates of transitions to other states. If we have active requests and the next event is a reply, the system's evolution is modeled by simply

decreasing the count of active requests by one. If an attack transitions from one state to a (usually more critical) state, this is modeled by decreasing the count of attacks in the initial state by one and increasing the count in the subsequent state.

Modeling the separate event streams as Poisson processes is particularly attractive from a simulation standpoint due to a fundamental mathematical property of such processes [Gross-Harris].

Let λ_i be the (state dependent) rate of process $i, i = 1 \dots N$.

The time to the next event from any of the processes is distributed $\exp(-\lambda_i t)$.

The event is of class k with probability $p_k = \frac{\lambda_k}{\sum_i \lambda_i}$.

The class and time of the next event are independent.

Simulating the evolution of the system is then at least conceptually simple: sum the rates of all possible event classes, generate the next event time and class, and change the system state accordingly. A change of system state may change which events are possible (for example, if an attack is mitigated, the attack success event is canceled) or change the rates of possible events. This is achieved by maintaining a list of all event classes that are possible at any time. Class members include the event type, the base rate, and the rate multiplier. The base rate is fixed for the duration of the simulation, but the rate multiplier is state dependent. The rate of any given event process in the simulation at any point in time is given by the product of the base rate and the state-dependent multiplier, that is,

$$\lambda_i^{process} = Multiplier_i \times \lambda_i$$

The system state consists of the number of active requests, the state of each host (UP, DOWN, or COMPROMISED), and the state of active attacks against each host (simplified for our purposes to probe, crash, or root compromise). If an event class is impossible in the present system state (for example, if no active requests are awaiting a reply) the corresponding rate multiplier is set to 0. Also, components working in a degraded fashion (for example, if the manager has decided to deploy a high-overhead monitor) are modeled as having a rate multiplier less than unity. Other actions such as

invoking an agreement protocol require that multiple server processes respond to each request, impacting the response rate of the redundant server bank as a whole. Actions from the system manager are modeled as decisions that may immediately affect the system state and then usually invoke a corresponding Poisson process to model time to effect. For example, if the manager invokes a REBOOT action on a host, the host is immediately considered down and remains down for an exponentially distributed time interval (that is, when active, the REBOOT process for each host is Poisson).

If request and reply rates were constant, and no other events occurred, a straightforward queuing model would describe the situation [Gross-Harris]. In this case, it is possible to derive an analytic formalism without resorting to simulation, but our system evolution introduces issues of nonstationarity and other complexities that make this difficult if not intractable. For these reasons, we have chosen to model the system via simulation rather than attempting to derive an analytical formalism.

For convenience, we define the “total up capacity” as the sum of the “up” and “compromised” probabilities of all active servers, that is,

$$C_{total_up} = \sum_{i=1}^{Nhosts} [P_{up}(i) + P_{compromised}(i)]$$

Note that a compromised server responds to client requests, but the replies are invalid.

The model comprehends the following event classes:

Requests: The mission of the system is to receive and reply to valid requests from clients, which arrive at a rate $\lambda_{request}$. The request process is always active (rate multiplier is always unity).

Replies: A necessary condition to ensure queue stability is that the service rate (the sum of the rates of the entire application server bank) strictly exceed the arrival rate. To ensure ample host capacity to allow the system to function with agreement protocols and crashed hosts, the base rate λ_{reply} for the reply process is chosen so that the sum of the rates for all hosts (the rate at which they would operate if they were not in a degraded mode) is about four times the rate for the request process, that is,

$$Nhosts \times \lambda_{reply} \quad 4\lambda_{request}$$

The reply process is active if there is at least one active request, in which case the multiplier is C_{total_up} . All server reply rates are the same for implementation simplicity, but this is not a theoretical requirement.

Host Crash: The system includes a low-rate process of nonmalicious failure for each host. The rate λ_{crash} is typically set to about $10^{-6} \times \lambda_{request}$. The multiplier is always unity.

Host REBOOT: The manager selects a REBOOT action as a response to adverse host states. The host in question is immediately considered DOWN, and remains in this state for a time interval that is exponentially distributed with parameter λ_{reboot} . After this interval, the host is considered UP, so that REBOOT events clear DOWN and COMPROMISED host states. The multiplier is unity if the manager selects a REBOOT action. The manager also invokes a REBOOT event in the case of an agreement failure.

Agreement Protocol: The agreement protocol occurs at a rate λ_{agree} or in response to manager action. A synthetic request is sent to all the hosts, and agreement is checked. The internal value $n_agree_requests$ tracks the number of hosts that have yet to reply to the agreement protocol. This protocol stops when all hosts reply with a valid response, or when any host replies with an invalid response. It also terminates if a polled host is down, but this takes longer to infer. In the latter two termination conditions, the manager posts reboot requests for the affected host (if only one is affected) or all the hosts. The response time is subject to the same response process as normal requests. If any hosts are compromised, this check will fail.

It is possible that a host compromise can sneak in during an instance of the agreement protocol, if the attack compromises a host that has already issued a valid reply for this instance, while some hosts have yet to reply. In this case, the manager may erroneously conclude that all hosts are correct, and will not detect the compromise until the next instance of the agreement protocol. This is exacerbated if the total request rate (client and agreement) is high relative to server capacity, in which case the delays to hear from all hosts will be longer.

This protocol is self-blocking in the sense that if there is an active agreement protocol (for example, if some hosts have not replied yet to the previous agreement request) no new agreement protocol can start.

Attacks: PROBE attacks arrive at rate λ_{probe} with a rate multiplier of unity. PROBE attacks proceed to states ROOT (which, if successful, compromises the victim host) or CRASH (which crashes the victim host) with rates $\lambda_{probe_to_root}$ and $\lambda_{probe_to_crash}$, respectively. Transitions from ROOT to CRASH occur with rate $\lambda_{root_to_crash}$. CRASH is considered a terminal state; transitions are also possible from PROBE and ROOT to the terminal state TERM with rates $\lambda_{probe_to_term}$ and $\lambda_{root_to_term}$, respectively. Attacks that reach a terminal state are no longer directly detectable, but may still be detected by asset distress monitors or failed agreement protocols. The effect of an attack that reaches a terminal state persists until remedied by manager action. Transitions between attack states have a rate multiplier equal to the number of active events in the predecessor state. The purpose of a PROBE attack is to discover a vulnerability to exploit in a subsequent attack stage.

Detections: Attacks in any state are subject to detection, with rate λ_{detect} and rate multiplier equal to the total number of active attacks. Symptomatic detection potentially diagnoses host distress modeling the blue sensor of [Valdes-Skinner]. This is not a Poisson process, but actually inferred from the number of dropped requests over a time interval. Finally, a compromised host can be detected by failure of the agreement protocol.

Preliminary Results

The following table summarizes rates, rate multipliers, and nominal values for the processes in the simulation. Here, we analyze the impact of varying rates from their nominal values. For each experiment, rates for all but the experimentally varied quantity are held at their nominal value. In all experiments, we simulate 100,000 events from an initial system state with all hosts up. The cost function is based on the percent of requests that are dropped or receive an invalid reply, with the latter more costly.

Event Class	Rate	Rate Multiplier	Nominal Value
Requests	λ_{req}	1	1000
Replies	λ_{rep}	$0, N_{active_requests} = 0$ $C_{total_up}, otherwise$	$4000/N_{hosts}$
Agree Requests	$\lambda_{agree_request}$	1	500
Agree Reply	λ_{agree_reply}	$n_agree_requests$	λ_{reply}
Crash	λ_{crash}	1	0.001
Reboot	λ_{reboot}	number of active reboot requests	10
Probe	λ_{probe}	1	10
Probe_to_root	$\lambda_{probe_to_root}$	number of active probes	5
Probe_to_crash	$\lambda_{probe_to_crash}$	number of active probes	5
Probe_to_term	$\lambda_{probe_to_term}$	number of active probes	5
Root_to_crash	$\lambda_{root_to_crash}$	number of active	5

		compromises	
Root_to_term	$\lambda_{root_to_term}$	number of active compromises	5

Value of Redundancy

We vary the number of hosts in the server bank from 1 to 4, holding the total server reply rate constant at four times the request rate. Almost 14% of the requests are either dropped or receive an invalid response in the single server case.

Servers	Rate	Percent Dropped	Percent Invalid
1	4000	10.05	3.36
2	2000	0.10	0.94
3	1333	0.51	0.24
4	1000	0	0.18

Not surprisingly, the system performance improves with redundancy, with the biggest improvement corresponding to the jump from one to two servers. For the remainder of this section, we model the server bank as consisting of 4 servers.

Rate of Agreement Protocol Requests

For this experiment, we varied the rate of the agreement protocol requests. Note the high fraction of invalid responses if no agreement protocol is in effect ($\lambda_{agree_request}=0$). Since this protocol is the primary mechanism for compromise detection, we believe the percent invalid will go higher if the simulation proceeds for a longer period. The only recovery mechanism is to reboot compromised hosts when they crash. A higher rate results in more timely detection of a compromised state. However, the agreement protocol competes with the client request and reply processes, thereby slowing down the detection

and repair of compromised states. This phenomenon of diminishing returns is apparent in the following table.

Since the agreement protocol can also detect DOWN hosts, omitting this protocol (the first line in the table) results in some number of dropped requests in addition to the large fraction of invalid replies.

The difference in the last two lines is probably not statistically significant, and the equivalence of the results may be due to the blocking effect of the agreement protocol (the realized rate reaches a limit regardless of the nominal rate).

$\lambda_{agree_request}$	Percent Dropped	Percent Invalid
0	0.88	18.50
100	0	0.35
500	0	0.07
1000	0	0.08

Value of System Capacity

If system capacity (maximum reply rate of the server bank) is low relative to the request rate, the queue can become unstable. Note that the total request rate is the sum of client and agreement requests, and is nominally 1500. Therefore, the first setting results in a potentially unstable system with a total response capacity of 1200. A large fraction of requests are dropped, and since the agreement protocol competes for the same scarce server capacity, compromised states evade detection for comparatively long intervals. There is a dramatic improvement when the server capacity increases to 500 (so the total server bank capacity is 2000).

λ_{rep}	Percent Dropped	Percent Invalid
300	65.07	14.25
500	3.83	0.80
1000	0	0.18
2000	0	0.16

Attacker Strategy

We assume that root compromises are more difficult and require more time than crash attacks. A root compromise that is not directly detected enables the attacker to control the host in question until the compromise is detected by the agreement protocol, or until the host crashes. Since invalid replies are more costly than dropped requests, it is in the attacker's interest to maintain a root compromise rather than to crash the host. In other words, the root-to-crash process is not in the attacker's interest.

Summary

We describe work in progress toward developing a simulation model to explore cost-benefit tradeoffs in response actions to adverse events in an enterprise network. The network consists of a set of redundant hosts serving client requests and subject to external attacks attempting to probe, crash, or compromise the system. The detection capability is abstracted from what we believe is an accurate picture of the state of the art of IDSs. Specifically, IDSs are capable of detecting attacks at various stages, but do so imperfectly and with delays during which the system may not respond or respond erroneously. Moreover, IDS reports may be false alarms and may prompt unnecessary and costly actions. An abstract manager selects actions given its imperfect view of the system. All actions have a cost, but there is a cost associated with dropped or invalid responses as well. In other words, choosing “no action” is itself costly in certain system states.

System processes (client requests, replies, attacks, and detections) are modeled as Poisson processes. We exploit certain properties of Poisson processes to provide fairly rich detail and fidelity with a tractable implementation. Our approach permits further refinement of any aspect (for example, higher-fidelity detection models). Manager actions typically cause an immediate response and then initiate a Poisson process to model the time to effect. In other words, the time to effect is not deterministic and not under control of the manager. For example, if the manager requests a reboot on a particular host, the immediate action is to take the host DOWN (if it was not so already), but the time until it comes back UP is exponentially distributed.

References

[Valdes-Skinner] Valdes, A. and Skinner, S. “Blue Sensors, Sensor Correlation, and Alert Fusion”, Recent Advances in Intrusion Detection (RAID 2000), Toulouse, France, October 2000. <http://www.raid-symposium.org/raid2000/program.html>

[Karlin] Karlin, S. and Taylor, H. “A First Course in Stochastic Processes”, Academic Press, New York, NY, 1981.

[Gross-Harris] Gross, D. and Harris, C. “Fundamentals of Queueing Theory”, Wiley and Sons, New York, NY, 1974.