An Abstract Interface for Cyber-Defense Mechanisms

Franklin Webber
BBN Technologies
fwebber@bbn.com

Partha Pal
BBN Technologies
ppal@bbn.com

Paul Rubel
BBN Technologies
prubel@bbn.com

Michael Atighetchi
BBN Technologies
matighet@bbn.com

ABSTRACT

Defending a computer system against malicious attack depends on making many different defense mechanisms work together. In addition to protecting against intrusions, these mechanisms should provide intrusion detection and response. The semantics of input and output for these mechanisms – what the alert from an intrusion detector means, and the implications of issuing a command in response – can vary greatly from one mechanism to another. In this paper, we discuss the abstract interface we have developed for integrating various defense mechanisms to defend a distributed application. Our interface is more than an API: it defines not only the syntax of communication with defense mechanisms but also its meaning, thus allowing us to reason systematically about the state of attack and defense. We briefly describe our current work toward automating that reasoning and thus toward applications that defend themselves intelligently and automatically. We also argue that reasoning about attack and defense at an abstract level allows one to model and analyze whether the defense is effective.

1. BACKGROUND

Cyber-defense combines protection, detection and response. One *protects* a system by eliminating vulnerabilities that an attacker can exploit. While protection of a computer system must always be the first line of defense, experience suggests that it is never perfect and therefore one should not rely on it to the exclusion of other defense mechanisms [1]. One should always prepare to detect and respond to any attack that succeeds in circumventing a system's protection.

Intrusion *detectors* [3] observe and report symptoms of an attack, ideally while the attack is still in progress. Detectors take many forms, from generic tools such as Snort [5] to monitors tightly coupled to a particular application. A report sent by an intrusion detector is often called an *alert*. Tools exist to collect and classify alerts to help system administrators understand the progress of an attack [4].

A good defense *reacts* to intrusions by containing them and, if possible, undoing their damage and protecting against similar attacks in the future. Because an automated attack can propagate quickly throughout a vulnerable system, intrusion response depends on designing the system with enough redundancy that it can continue operating, perhaps in a degraded mode, while the defense reacts [8].

We have instrumented several systems with cyber-defenses of the kind described above. Our most recent work defended a distributed publish/subscribe application for the US military [2]. Our defenses were tested in numerous attacks by several professional Red Teams acting with few constraints; the defenses offered significant resistance to attack and are likely the current high-water mark in cyber-defense.

Our experience with trying to understand and reuse cyber-defenses has been the motivation for the abstract interface described in this paper.

2. ABSTRACT MODEL

To make a set of defense mechanisms work together well, one should begin with an abstract model in which the properties of each defense mechanism can be described. We begin with a model of cyber-defense as a control loop. As shown in Figure 1, the system being defended sends inputs to a controller, which then interprets the inputs to understand the progress an attacker is making and decides on a response, which results in outputs to the system to make changes or to collect more information.

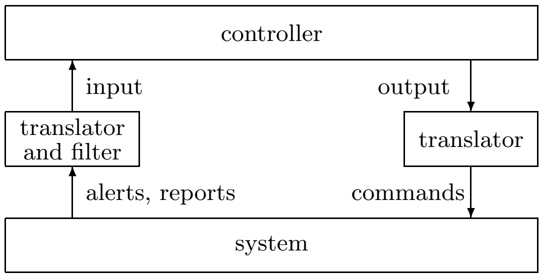

Figure 1: control loop for defense

The figure also shows a translation and filtering step at the interface to the controller: inputs are translated into a abstract form for the controller, redundant inputs are removed, and outputs from the controller in an abstract form are translated into commands to the system. This section describes the abstract form used by the controller.

Note that the distinction in the figure between the system and its controller is conceptual only: the implementation of the controller must in fact be part of the system being defended.

2.1 Controller Inputs

Inputs to the controller are of two kinds:

1. reports of anomalous events, each of which is an alert from some intrusion detector;
2. reports of expected system activity.

We need an abstract form for these inputs to solve the following problems:

- Intrusion detectors can produce a lot of data. Even a system that is not under attack can generate a huge volume of reports about its activity. Without reducing this volume, the controller may be overwhelmed.
- An attacker might easily flood the controller with fake data in an attempt to overwhelm and disable it.
- Intrusion detectors commonly produce false positives, i.e., inaccurate alerts that an attack is underway even when it isn't. In general, the information content of reports sent to the controller will be highly variable, with some reports being nearly certain and others being nearly worthless.
- An attacker might easily send fake data to the controller, i.e., intentionally inaccurate alerts about an attack being underway, in an attempt to cause the defense to make bad responses ("spoofing").

2.1.1 Accusations

We treat every anomalous event as an *accusation*: system component C claims that component A behaved badly when interacting with component B. In most cases B and C will be the same component, which is reporting on unexpected behavior by A. Figure 2 shows the general case.

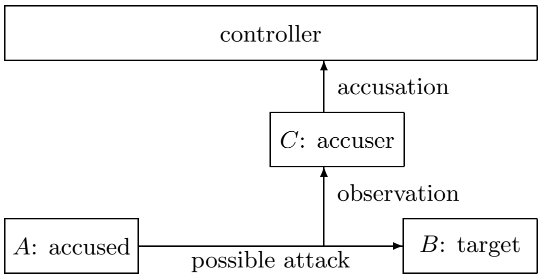

Figure 2: accusations

Accusations differ according to the kind of anomalous behavior observed. The two main categories are:

1. **commission failure**: the accused component sent data that does not conform to any expected communication protocol;
2. **omission failure**: data expected from the accused component did not arrive.

Special cases of (1) include: **timing**: in a real-time system, valid data arrived at the wrong time; **policy**: in a secure system, the accused attempted an unauthorized action; **flooding**: the accused is sending data much too fast.

The information content of an accusation determines how the controller should handle it. First, only accusations whose source is known with near-certainty should be kept. Usually, this means every accusation must have a recognized digital signature or be transmitted via a trustworthy VPN. An accusation whose source is not known could easily have been faked by an attacker and so carries little useful information; it means merely that an attack is going on somewhere. Second, an accusation that carries its own proof contains more information than one without proof. Proof would be an invalid message signed by the accused, which proves that the accused was corrupt at some time. Without proof, the accusation means that either the accusation is true or the accuser is corrupt and is possibly lying under the control of an attacker. Third, repeated accusations carry little new information other than a timestamp (which is of doubtful value if generated by the accuser, who might be corrupt).

2.1.2 Evidence

Although accusations are the primary data used by the controller to interpret the progress of an attack, more information may be needed. This additional information comes from reports of normal operation of the system, which we refer to as *evidence*.

So far, we have found need for two kinds of evidence:

- **communication**: a report by component X that an expected message arrived from Y. Communication evidence is needed to resolve uncertainty about which network components are still working. One possible interpretation of an omission accusation is that some network component is dead; communication evidence can rule out some of these possibilities.
- **mission status**: a report by component X that component Y has entered a new phase of the mission in which certain responses should or should not be applied to Y. For example, in some situations, Y should not be quarantined because its role in the mission is critical. Often $X = Y$, i.e., a component reports its own status.

Evidence is like accusations in that: it has little useful information if the source is not known; it is always possible that the source of the evidence is corrupt and is lying; and repeated evidence carries little new information. These properties of every input solve the problems listed at the beginning of this section: inputs arrive in an abstract form, thus reducing the volume of data; an attempt to flood the controller with valid inputs will be noticed immediately; because the meaning of the inputs is defined, reasoning about false positives and spoofing is made possible, and Section 3 will describe how that reasoning reduces false positives and spoofing.

2.2 Controller Outputs

Outputs from the controller should be at the same abstract level as the inputs. This means that if an (abstract) output has its intended effect on the system, then its effect on future (abstract) inputs can be predicted.

We have so far concentrated on the kinds of output on the following list:

- **reset**: return a component to its initial state;
- **refresh**: return a component to a checkpointed state;
- **quarantine**: block a component's interaction with the rest of the system;
- **reintegrate**: reverse a quarantine, unblocking a component;
- **ping**: test liveness of a component and intermediate network components.

The effect of reset and refresh of X is to undo corruption, assuming that X was not corrupt in the state it is returned to. The effect of quarantine of X is to prevent an intrusion from spreading from X but also to cause new accusations that X is dead. The effect of a ping of X is to cause a reply that will become new communication evidence.

3. REASONING ABOUT ATTACK AND DEFENSE

The reasoning done by the controller, whether it is a human system administrator or a computer program, divides naturally into two phases:

1. *Input Interpretation*, in which the controller seeks a good explanation for its inputs by answering the questions: which components are corrupt? which are dead? which are flooded? We summarize below how we have automated the search for good explanations.
2. *Output Selection*, in which the controller uses its best explanation of the inputs to choose a response that is likely to counter the attack. We have automated this choice as a set of rules that "fire" when components are corrupt, dead, or flooded, and a set of rankings that prioritize the rules that "fire", using knowledge of responses that have previously been tried.

Choosing a good explanation involves coping with several kinds of uncertainty. First, the controller does not know how the attacker is likely to behave. We handle this uncertainty by making assumptions about what the attacker is more likely to *know*, assuming that his behavior is limited to attacks he knows how to carry out, and otherwise making no assumption about what attacks he is likely to *choose*. We identify a set of kinds of vulnerability that the attacker might exploit, e.g., attacks against every host operating system of a particular kind, or attacks against every component of a replication group. We assume that an explanation needing more kinds of exploits is less likely. Therefore, a good explanation is one that needs the fewest kinds of attacker exploits.

Second, the controller does not know which inputs are accurate and which are lies created by the attacker, sent from corrupt components. We handle this by proving theorems: a component is corrupt if it is necessarily corrupt in every possible attack consistent with the inputs and a given number of exploits.

Third, the set of inputs may be logically inconsistent. We handle this by discarding inputs until consistency is reached: an input contradicted by a later input from the same source will be discarded, and inputs that show the least *coherence* [6] with the rest may be discarded.

4. EFFECTIVENESS OF THE DEFENSE

If the controller is automated, a key question is: how effective is the defense it provides? Although there are many ways to measure effectiveness, one important measurement is of the worst case: given a set of vulnerabilities to exploit, what is the greatest damage the attacker can inflict? Because an automated controller is entrusted with disabling components that it interprets to be corrupt, it may be possible for a clever attacker to trick the controller into helping the attack. A worst-case analysis of the defense puts a limit on the attacker's ability to do that.

We have done rudimentary work on two kinds of analysis. First, we have run simulations of systems under attack, modeled at the level of abstraction described in this paper. It may be possible to estimate the worst case using many randomly chosen simulation runs.

Second, we have done a game-theoretic analysis of a special case of the controllers described in this paper [7]. It may be possible to generalize that analysis.

5. CONCLUSION

This paper summarizes some recent work at BBN toward automating the defense of computer systems against malicious attack. Our goal is to encode knowledge we have gained while building and using cyber-defenses manually into an algorithm that can control such defenses automatically. Formulating such an algorithm has required us to specify how a variety of defense mechanisms can be integrated, and their behavior reasoned about. This specification treats the interaction between the mechanisms and their controller at an abstract level. We believe that reasoning at this level of abstraction contributes to the intelligent use of information for cyber-security.

6. REFERENCES

[1] B. Blakely. The emperor's old armor. In *New Security Paradigms Workshop*, pages 2–16, Sept. 1996.

[2] J. Chong et al. Survivability architecture of a mission-critical system: The DPASA example. In *Comp. Security Applications Conf.*, Dec. 2005.

[3] S. Kent. On the trail of intrusions into information systems. *IEEE Spectrum*, Dec. 2000.

[4] P. G. Neumann and P. A. Porras. Experience with EMERALD to date. In *Proc. 1st Usenix Workshop on Intrusion Detection and Network Monitoring*, Apr. 1999.

[5] SourceFire. Snort: the de facto standard for intrusion detection. Internet URL http://snort.org, 2008.

[6] P. Thagard and K. Verbeurgt. Coherence as constraint satisfaction. *Cognitive Science*, 22:1–24, 1998.

[7] F. Webber et al. A model of quarantine in cyber-defense. Technical Report ITUA Validation Report, Chapter 5, F30602-00-C-0172, BBN Technologies, 2004.

[8] F. Webber, P. Pal, et al. Defense-enabled applications. In *DARPA Info. Survivability Conf. and Expo.*, May 2001.