Global Information Assurance Certification Paper

Exploration of Computer Immune Systems

GIAC Security Essentials Certification (GSEC)
Practical Assignment - Option 1
Version 1.4b

Robert Berbeco

August 10, 2004

Abstract

This paper explores the theory and implementation of computer immune systems, and the related field of Distributed Autonomous Agents (DAAs). In theory, a computer immune system is based on a biological immune system model. I will show how the principles of a biological immune system can apply to a computer immune system, and explain how a computer system could utilize self-awareness to identify and protect itself from intrusion, virus, or other anomalies. With self-awareness, a system could increase its basic knowledge set, and adapt to ward off future threats it currently does not have knowledge of. The theory and algorithms that exist to how a system can detect local system changes will be explored, and Distributed Autonomous Agents are reviewed as a method of sharing the information one system has with other nearby systems through the network. To compliment the theory, I will introduce an example of an experimental computer immune system, Cfengine. Cfengine actually has it roots as administrative tool, but with modifications it can serve as the basis for a rudimentary distributed computer immune system. In the concluding remarks I will summarize the paper and its relevance to the security field.

Computer Immune Systems

Computer immune systems incorporate the fundamentals of various Computer Science theories and implementations. Of the applicable theories, the first and foremost is pattern matching. With pattern matching, a computer system is able to differentiate between itself, normal activity, and a potential anomaly by tracking the patterns generated from normal operating processes, user habits, and system or access logs. Next, adaptation or system learning is necessary for the system to function autonomously. Adaptation can be accomplished through artificial intelligence, where much study has been done with fuzzy logic and adaptive rule sets. Once a system is able to adapt as a self-aware entity, the next logical step is for this system to share its knowledge with other neighboring systems. This communication and coordination can be accomplished with Distributed Autonomous Agents which could pass information from one system to another based upon a pre-defined response to a situation or an activated rule result. Once each individual system is able to effectively communicate with one another and actively fight as a combined unit to attack anomalies, the entire network of systems become an integrated unit acting as one computer immune system warding off attacks (whether from hacking, viruses, etc.). Initially this computer immune system will require human interaction for installation and configuration, but once configuration is complete and an initial rule set has been put in place it could autonomously handle threats.

The biological immune system is an extremely efficient protocol driven system that is autonomous, and can adapt either immediately or over time to deal with an external threat. Due to its effectiveness, fault tolerance, and efficiency, a biological immune system would serve as an excellent theoretical model for a computer immune system. Some questions would arise in attempting

to apply the biological model to a computer immune system. What could be adapted from biological immune systems in order to build a computer immune system that is fault tolerant and fault correcting? Can the mechanisms of natural selection and defensive counter attack be useful in computer immune systems? Some potential answers to these questions will be explored in the next section.

Principles of Biological Immune Systems Applied to Computer Immune Systems

Principles of the biological immune system that should be applied to a computer immune system include: it must be specific and should actively eliminate anomalies; it should be tolerable, distributable, adaptable, and able to dynamically cover the entire system(s); it should be able to identify anomalies systematically through their behavior; and should be mostly autonomous [1]. The first step involves establishing a baseline of identifying normal processes or habits on a system, so that detection of anomalies can be made possible. Once the computer immune system is able to recognize itself, it would be better tuned to recognizing and actively eliminating pathogens. Second, the computer immune system must be tolerable to normally executed processes, but still able to detect a normally executed process that has run amok. An example of this in a biological organism is cancer. In an organism with cancer, cancerous cells mutate from otherwise normal cells. The T cells, which serve as anomaly detectors and destroyers in a biological organism, are ineffective against this mutation since these mutated cells actually belong to the biological organism. The T cells will not actively bind and destroy the cancerous cells, and with the biological immune system ignoring the cancerous cells they are able to grow out of control. Third, the individual systems in a computer immune system should be able to distribute their detector sets throughout the system. Each computer system could have its own set of detectors, but the systems should be allowed to communicate with one another. In this way they can collectively ward off anomalous agents and malicious attacks. Fourth, each computer immune detector set should be dynamic and adaptable. Since it would be virtually impossible to initialize a completely thorough detector set manually, each computer system should be able to actively add to its repertoire when new threats are detected. The true autonomy of a computer immune system will be reached when it can proactively adapt to previously unknown threats without human intervention (like adding a security patch, which is currently done with most anti-virus programs).

Establishing a Sense of Identity/Self How a Computer Can Identify Itself

Process level detection

In addition to the biological immune system model, applying pattern recognition to a computer immune system would be useful. Pattern recognition in a program code could be applied to individual binaries and be used to detect harmful processes. Some processes that could be detected would include user

initiated deletions of system files, or services which attempt to conceal themselves like malicious root kits. In order to determine processes to allow or reject, a rule set must be created to detect strings which can lead to dangerous behavior.

Program code stored on a disk inertly is very unlikely to cause damage to a system. When the code is executed, the potential for system damage comes from the system calls initiated by the running process. In applying this logic, attention would be restricted to system calls in running privileged processes. Since a heterogeneous environment typically exists in how individual systems are configured, patched, and used, one way to deal with process detection would be by using process databases on each system [2]. Once a stable database is constructed for a given process, the database can be used to monitor the process' ongoing behavior; then sequences of system calls would form the baseline of normal patterns for the database. After the baseline is established, abnormal sequences could indicate anomalies. This method would have two main steps: first, scan traces of normal behavior and build up a database of normal patterns; second, scan new traces that might contain abnormal behavior, looking for patterns not present in the normal database [2]. To build the database, a window will be slid of size i+1 across a trace of system calls to record which calls follow which within the sliding window. As an example, in the below table where i=3, the sequence of system calls are *open*, *read*, *nmap*, *nmap*. As the window is slid across the sequence of system calls, each call that follows it is recorded at position 1, 2,…,i. So for this example, the following table would exist:

call	**position 1**	**position 2**	**position 3**
open	*read*	*nmap*	*nmap*
read	*nmap*	*nmap*	
nmap	*nmap*		

When the database of normal patterns is completed, a check can be done by using the same i+1 method to test for the presence of legal sequences. Once the normal process database is created, new behavior can be determined normal or anomalous by counting the number of mismatches between a new trace and the database; and comparing that information against a predetermined threshold value. Below the threshold would be normal, and above the threshold would be considered anomalous or potentially malicious. The normal database should initially be created with a standard set of artificial messages. Then as new local user processes are scanned these can be added to the database dynamically. Each computer system would individually generate its own normal database, based on local software/hardware and usage patterns. Since configuration time is necessary for initializing the self database, the downside is that this would have to be mostly completed prior to a computer system being on-line. This method would be the start of a computer establishing its identity to itself.

System call level detection

There are many ways in which system call data can be used to determine normal behavior of programs, and all of these would involve building a model using traces of normal processes. There are three main methods of detecting intrusion detection through system calls: frequency-based methods, data mining, and finite state machine.

Frequency-based methods model the frequency distributions of various events. A frequency-based method proposed by Helman & Bhangoo [3] involves ranking each sequence by comparing how often the sequence is known to occur in normal traces with how often it is expected to occur in intrusions. Sequences occurring frequently in intrusions and/or infrequently in normal traces are considered to be more suspicious. Unfortunately, this method makes several assumptions that are problematic for the system-call application: it assumes that data is independent and stationary, and it does not characterize the frequencies of abnormal sequences accurately [2].

Data mining approaches are designed to determine what features are most important out of a large collection of data [2]. Sequences are characterized as occurring in normal data by a smaller set of rules that capture the common elements in those sequences. During the monitoring stage, any sequence that violates these rules is treated as an anomaly.

Finite state machines attempt to recognize the language of the program traces [2]. An example of a very powerful finite state machine is the hidden Markov model (HMM), used in speech recognition and also in DNA modeling [4, 5]. A HMM's states represent some unobservable condition of the system being modeled. In each state, there is a probability of producing other observable system outputs and a separate probability indicating the next states. HHMs have the highest amount of accuracy when compared with other system call intrusion methods, but at a huge computational expense since all the datasets must be configured extensively before the system can be run [2].

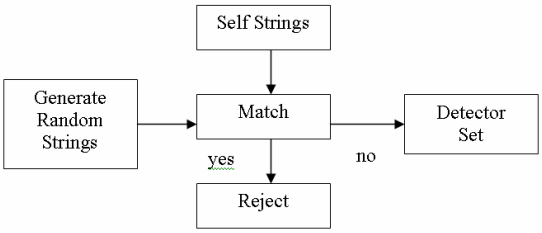
Algorithms to Detect Local System Changes

After a computer has been able to differentiate normal process execution and anomalies, it must be able to detect changes to itself. When considering change detection it should be non-specific since its generality could be applied in many settings to catch changes that might otherwise go undetected. Some algorithms that can be utilized for change detection include: the exhaustive detector generating algorithm (or T cell algorithm), linear time detector generating algorithm, and greedy detector generating algorithm [6].

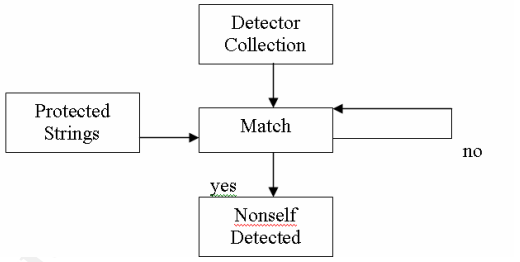
The exhaustive detector generating algorithm was inspired by the generation of T cells in the immune system and has two phases: first – generate a set of detectors; second – monitor the protected data by comparing them with the detectors and if a detector is activated a change is known to have occurred [6]. The algorithm only concerns itself with protected strings since they do not

change much over time and “self” is defined as being the protected string, and “non-self” to be any other string [6].

To generate valid detectors, the self string must be split into equal-size segments. For example, the following 32-bit string could be broken into eight substrings, each of length four: 0010 1000 1001 0000 0100 0010 1001 0011 which would produce the collection of self substrings to be protected. The next step would be to generate random strings to match against the self strings. Random strings that match the self strings are eliminated and strings that do not match any of the strings become members of the detector set [6]. The following diagram exemplifies this process:



Once a detector set of strings has been produced, the state of self can be continually monitored by matching strings in the self set with the strings in the detector set. If ever a match is found, it is concluded by the system that the self set has changed. The following diagram exemplifies this process:



The exhaustive detector generating algorithm has many advantages: it is tunable, the detector set does not grow with the number of strings being protected, protection is symmetric, and can be distributed among several systems using similar detector sets [6]. The algorithm is tunable since one can choose a desired probability of detection, and then estimate the number of detector strings required as a function of the size of the strings to be protected (probability = 1/strings to be protected). The size of the detector set does not grow with the number of strings being protected since the number of detector strings created is independent of the number of original self strings. Protection is symmetric since changes to the detector set are detected by the same matching process that notices change to self strings. The biggest disadvantage to the algorithm is the computational difficulty of generating the initial detector sets. This computational difficulty is increased even more by the fact that the number of initial detector strings grows exponentially with the number of self strings, and the

probability of detection increases exponentially with the number of independent detection algorithms running in parallel on multiple systems [6].

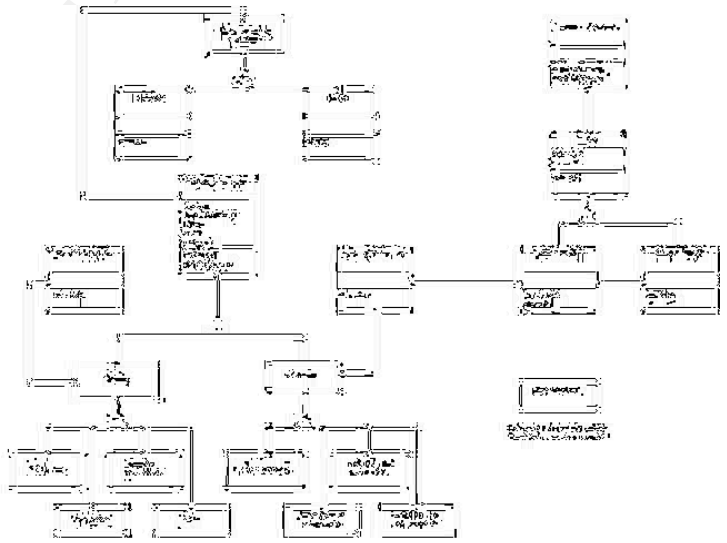
The linear time detector generating algorithm attempts to address the computational issue associate with the exhaustive detector generating algorithm; has two phases; and runs in linear time with the respect to the size of input [6]. In phase one, the algorithm counts the recurrence of a number of strings unmatched by initial strings; and in phase two, enumeration is used by counting the recurrences from picking detectors randomly in the set of candidate detectors. By this method, the linear time detector generating algorithm filters the detector set down to be more specific. When compared to the exhaustive detector, it is much more computationally efficient as it is running in linear time to size of the self set and the detector. The disadvantage of this algorithm is it needs much larger space requirements than the exhaustive detector algorithm for constructing the data sets; and if really long initial protected strings are used, the time to create the data sets starts to increase exponentially [6].

The greedy detector generating algorithm tries to address the issues from both of the above algorithms, and this algorithm achieves a better coverage of the string space with the same number of detectors by not selecting the detectors at random [6]. This algorithm has two phases: phase one – generate two arrays, one for the self set and the second one for current state of detector set; phase two – generate strings unmatched by the self set. During its execution, a running count of the number of non-self strings can be retained that are still unmatched by any detector; and this algorithm results in a much more compact detector set and low failure probability than the above two algorithms, but it runs much slower in real-time than the others [6].

Some factors must be taken into consideration before a determination can be made of which of the above algorithms to use in detecting local system changes. First, what is the level of security you need to achieve and maintain? The number of strings that need to be checked against the detector set depends on the answer to this question, and a balance has to be met between the level of security and the degradation of performance of the system in achieving the desired security level. Second, what is the frequency of checking you wish to use and should it be intermittent or constant? Intermittent may not be acceptable if a single occurrence of change can be fatal, whereas constant may involve too many system resources to maintain. Third, do you want to implement a weighted, distributed, and/or distributed independent detection type? With weighted, the detectors are chosen more frequently based on previous performance or known expected changes. Distributed will have the detector set split over a number of autonomous agents with each completing anomaly checks in parallel. Distributed independent will have each agent with a detector set generated independently from all other agents. This is similar in concept to a population of bio-organisms with separate immune systems [6].

Share the Information with Other Systems
Distributed Autonomous Agents

A Distributed Autonomous-Agent (DAA) Network-Intrusion Detection System is a collection of autonomous agents running on the various hosts of a heterogeneous network, and it provides the foundation for a complete solution to the complexities of real-time detection [7]. These agents would monitor intrusive activity on their individual hosts, can be configured specifically for the operating system in which it runs, and would coordinate with one another to eliminate intrusions and anomalies [7]. An agent would be installed on each network host and each agent would be independent, but would communicate, cooperate, and coordinate with one another when incoming events are detected. A proposed DAA by Bassus et al uses alerts that can escalate depending on the level of danger to the system and level of transferability to other systems; and these escalating alerts can be passed to other agents so that those systems can take appropriate measures to counteract the potential malicious event [7]. Within this system, potential attacks can be classified as either misuse behavior or anomalous behavior. Misuse behavior consists of attacks that can be defined or ones that are known to exist. In order to determine whether an attack occurred, the system could compare the behavior with the behavior of defined attack patterns, and a match would indicate a possible attack [7]. Therefore misuse examples would include denial of service, hack attempts, illegal logon attempts, port scans, and other attacks where the pattern of attack can be defined. From Kumar's research [8], misuse can be further categorized as existence, regular expression, or sequence; and intrusions can be represented by an event or series of events. Anomalies consist of attacks involving a non-typical use of system resources and are recognized by methods that develop profiles of normal user's behavior [7]. If a particular behavior reaches a pre-set threshold, the behavior becomes anomalous. Kumar's research [8] suggest that these threshold values could be determined by the activity intensity rate, audit record distribution of activities, categorical distribution of activities, and ordinal measurable activities. The following diagram [7 – Figure 3 in reference] represents the object model of the agent design for this proposed DAA system:



In the above system an agent will be installed on each host in the network. This agent will be continually monitoring for potential misuse and anomalies, and will send messages to other agents if a malicious event is detected. The monitoring and notification process is achieved by the Communication Interface, which represents the superclass for the Listener and Sender objects. The Listener will listen for and receive all incoming messages from other agents, and it processes them depending on the attack. The Sender sends messages to other agents, which will be processed by that agent's Listener. While the agent is active on a particular system, the Misuse and Anomaly Detectors will be running. The Misuse Detector is responsible for monitoring system resources and audit files to find and identify potential security issues. If one is found, it will instantiate a Misuse object to deal with the threat [7]. The Anomaly Detector is responsible for monitoring system resources for out of the ordinary behavior. It will call the Profile of an account or multiple accounts and compare this profile with a stored Profile. If differences within the two profiles reach a pre-determined threshold, it will instantiate an Anomaly object to deal with the threat [7]. The Profile Generator is responsible for tracking and measuring system resources and audit files, and then creating profiles that can be used by the Anomaly Detector to detect potential malicious changes. The Current Profile is a profile for a pre-determined period up to the current time, and the intent of the Current Profile is to represent the most up-to-date profile of the system. The Stored Profile is used for historical comparisons, and is periodically checked and merged with the Current Profile to ensure that its data is fairly up-to-date. The Intrusion Attack object is responsible for maintaining a database of attributes and methods of intrusion types that have been detected by the agent. This data store can be configured for a base level of attributes so each agent has a basic frame of reference; then as new intrusions are detected they can be added. As this Intrusion Attack object is updated on the local host, the agent is also responsible for coordinating this information with other agents. This enables each agent in the system to the most recent intrusion types available for reference.

Overall this architecture would have some advantages as it is decentralized, distributed, and each agent in this system actively cooperates; yet function independently of one another. Since this architecture is decentralized and distributed it enables growth, there is no single point of failure in the system, and is very flexible. If there is a need to have centralized administration this could be added as a console which would communicate with agents through the existing channels. Real-time response is enabled by the channels of communication that are enabled between the agents. As new threats are detected on one system other systems' agents would become notified and would enable these other systems to become "immune" to the threat.

Cfengine

Cfengine is an administrative tool created by Mark Burgess, Faculty of Engineering, Oslo College, Norway, that can be used on BSD and System-5-like UNIX operating systems and can be utilized through a TCP/IP network [9].

Cfengine is meant to allow system administrators to create a single, central system configuration which will define how every host, that has Cfengine installed, on a network should be configured. A master configuration file will define how each host will be configured, since the configuration of each host is checked against this file, and then any deviations from the individual host configurations will be fixed automatically.

Cfengine's role in designing a basic level computer immune system is that it is useful for automation; and as a reactor to examined situations on a system or systems and performing the corrective action [10]. Cfengine also communicates within its environment to maintain the distributed network of systems, and this is an important method of converging individual behaviors of each Cfengine host into one defined behavior across the network [10]. As a reactor, Cfengine enables the ability to configure corrective actions for potential malicious actions. For example, Cfengine can examine the state of a host system and execute an administratively specified corrective algorithm. If Cfengine is configured to log the changes it makes to the system due to the corrective algorithm, it can reanalyze these changes in order to alter Cfengine's program next time [11]. Cfengine also allows for logging and storing the history of the system which can be used for statistical analysis of machine behavior in order to provide feedback, and this feedback can be used to determine threshold behavior for activating countermeasures [11]. This strategy can be applied to system resources, network collisions, and hacking attempts.

There are some advantages and disadvantages to using Cfengine as a rudimentary computer immune system. Cfengine, in concept, is very similar to Microsoft Windows 2000/2003 group policy implementation, and is very versatile. Once the cfengine.conf file is created and Cfengine is designated to run as a cron job; the policy on the system is self-maintaining and does not need to be touched, unless the system administrator wishes to add something to it. Cfengine also is good at performing repetitive administrative tasks like examining files, creating files, aliasing files, replacing files, renaming files, editing files, changing rights, and starting and stopping processes.

The biggest disadvantage that I could see to a potential use of Cfengine as the basis of a computer immune system is it appears to lack a robust way of pattern matching. Since pattern matching is needed for identification of self and anomaly in a computer immune system, I think that this is a major shortfall to using the system. The only pattern matching that seems to be possible currently with Cfengine is by file attributes (like strings, dates, etc.). For this system to be more useful as a potential computer immune system this shortfall will have to be overcome. Another potential disadvantage is that Cfengine, like other systems that require human configuration, is susceptible to human errors that may potentially render the system useless if these errors are not caught when the system is live.

The biggest advantage to using Cfengine is that it currently appears to be the only publicly and readily available basic level computer immune system model. The system has been used and researched for about 5-6 years so there is much known about the system and extensive amounts of documentation is available. This system appears to be a good first step towards creating a functioning computer immune system prototype.

Conclusion

The concept of a computer immune system is still a task that requires more research and experimentation. Much of the current research that has been completed is still fragmented amongst other related fields such as multi-agent systems, artificial intelligence, and distributed administrative systems. What is now required is to combine these individual concepts into an integrated and autonomous end product.

Hopefully, in the near future computer immune systems will become realized. As security becomes more of an issue and a priority for companies and individuals; the creation and implementation of a computer immune system will help to automate responses to malicious activities and with active learning enable “immunity” across a network. It could execute in the background without a central point of failure to detect and eliminate anomalies before they overrun a system or network. This will free up the human individuals that are currently spending much of their time to perform some of these repetitive actions to do other required tasks. The computer immune system would become self-sustaining over time. Since it appears that currently available basic computer immune systems, such as Cfengine, still require human intervention during much of their existence (initially and for updates), the realization of a true functioning autonomous computer immune system will require more devotion, thought, and research for the future.

References

[1] Forrest, S., Somayaji, A. Hofmeyr, S. “Principles of a Computer Immune System.” Department of Computer Science, University of New Mexico, 1997.

[2] Warrender, C., Forrest, S., Pearlmutter, B. “Detecting Intrusions Using System Calls: Alternative Data Models.” Department of Computer Science, University of New Mexico, 1999.

[3] P. Helman and J. Bhangoo. “A statistically based system for prioritizing information exploration under uncertainty.” IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans. July 1997 27(4): 449–466.

[4] L. R. Rabiner. “A tutorial on Hidden Markov Models and selected applications in speech recognition.” Proceedings of the IEEE. 1989 77(2): 257–286.

[5] L. R. Rabiner and B. H. Juang. “An introduction to Hidden Markov Models.” IEEE ASSP Magazine. January 1986 (1986): 4–16.

[6] M. Ayara, J. Timmis, R. de Lemos, L. de Castro and R. Duncan. “Negative Selection: How to Generate Detectors.” ICARIS 2002. 2002. URL: http://www.aber.ac.uk/icaris-2002/Proceedings/paper-35/ayara-etal.pdf (2002)

[7] Barrus, J., Rowe, N. “A Distributed Autonomous-Agent Network-Intrusion Detection and Response System.” Command and Control Research and Technology Symposium, Monterey CA. June-July 1998. URL: http://www.cs.nps.navy.mil/people/faculty/rowe/barruspap.html (1998)

[8] Kumar, Sandeep. “Classification and Detection of Computer Intrusions.” Department of Computer Sciences, Purdue University. Ph.D. Dissertation, 1995. URL: http://www.cs.plu.edu/pub/faculty/spillman/seniorprojarts/ids/classification.pdf (1995)

[9] Burgess, Mark. “cfengine - Conceptual Basis.” Centre of Science and Technology, Oslo College. URL: http://www.cfengine.org/cfdetails.html

[10] Burgess, Mark. “Computer Immune Systems.” Centre of Science and Technology, Oslo College. 28 May 1998. URL: http://www.iu.hio.no/~mark/research/immune/immune.html (28 May 1998)

[11] Burgess, Mark. “Computer Immunology.” Proceedings of the 12th Systems Administration Conference (LISA '98). 6-11 December 1998. URL: http://www.usenix.org/publications/library/proceedings/lisa98/full_papers/burgess/burgess_html/burgess.html (6-11 December 1998)