Learning Unknown Attacks – A Start♦

James E. Just[1], James C. Reynolds[2], Larry A. Clough[2], Melissa Danforth[3], Karl N. Levitt[3], Ryan Maglich[2], Jeff Rowe[3]

[1] Global InfoTek, Inc
jjust@globalinfotek.com

[2] Teknowledge Corporation
{ reynolds, lclough,
rmaglich}@teknowledge.com

[3] University of California
{danforth, levitt,
rowe}@cs.ucdavis.edu

Abstract

Since it is essentially impossible to write large-scale software without errors, any intrusion tolerant system must be able to tolerate rapid, repeated unknown attacks without exhausting its redundancy. Our system provides continued application services to critical users while under attack with a goal of less than 25% degradation of productivity. Initial experimental results are promising. It is not yet a general open solution. Specification-based behavior sensors (allowable actions, objects, and QoS) detect attacks. The system learns unknown attacks by relying on two characteristics of network-accessible software faults: attacks that exploit them must be repeatable (at least in a probabilistic sense) and, if known, attacks can be stopped at component boundaries. Random rejuvenation limits the scope of undetected errors. The current system learns and blocks single-stage unknown attacks against a protected web server by searching and testing service history logs in a Sandbox after a successful attack. We also have an initial class-based attack generalization technique that stops web-server buffer overflow attacks. We are working to extend both techniques.

1. Introduction

Designing secure systems to survive cyber-attacks is both hard and complex. Initially designers built stronger locks (e.g., put strong security mechanisms into and around the systems) to keep out the attackers. However, protection wasn't perfect and some attacks still got through. Then designers added alarms (e.g., intrusion detectors) to alert us to the attackers. Unfortunately, the detectors weren't perfect and some attacks still got through undetected. More recently, designers have shifted their focus to building systems that continue operating despite the attacks by leveraging the well-developed techniques of dependability and fault tolerance.

It is impossible to build an intrusion tolerant system that survives for any meaningful time without solving the problems of unknown attacks and finite failover resources. It is always useful to harden a system so the adversary's work factor to penetrate it is increased but there are limits to this approach. The time and effort required to identify a new vulnerability in your system and develop an exploit for it may be quite large. To a determined opponent, it's just time and money.

The real problem is that, once an attack is developed and put in place, the time required to execute it is very small and, in many cases, the time and effort required to create simple variants of the attack are quite small. If the threat environment for an intrusion tolerant system includes a well-resourced adversary (e.g., a state-sponsored cyber-terrorist group or organized crime), the system must be capable of dealing with many unknown attacks -- possibly repeated quickly (seconds or minutes). This attack scenario will rapidly exhaust any redundant components in the system and represents, in our opinion, the worst case design point.

Many fault tolerant mechanisms work because most faults are independent, low probability events and hence are easily masked. Common mode failure is a well-understood problem and one to be avoided often through the use of diversity. Attacks do not allow the first assumption of independence and the nature of the critical service to be protected may belie the use of diversity.

This project is the first attempt to build a prototype system that combines intrusion detection, responses that block attacks, failover to remove compromised elements, learning to create rules for blocking future occurrences of attack and generalization to block even significant variants.

1.1 Background

Commercial organizations, the Government and even the military have reduced their cost and, arguably, improved their reliability through the increased use of COTS software and hardware, even for critical applications. Unfortunately, they have also increased their vulnerability to well resourced adversaries who want to do serious damage to critical infrastructure, steal information, and disrupt services. Most researchers are saying that it is essentially impossible to build large scale software without faults and it is certainly impossible to prove such software contains no faults [1], [2], and [3]. Moreover, as two damaging recent attacks (Code Red 1 and Code Red 2, which exploited a known buffer

♦ This work was partially funded by Defense Advanced Research Project Agency under contract #N66001-00-C-8074. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Project Agency or the U.S. Government.

overflow vulnerability in Microsoft's web server, Internet Information Server) have amply demonstrated, faults are being exploited long after patches are available to fix the problems. This is not to say that software security cannot be improved but it is important to begin examining other approaches to security.

The US Defense Advanced Research Project Agency (DARPA) began a program in 2000 to apply fault tolerance techniques to building intrusion tolerance systems. As part of this effort, a number of organizations, including Teknowledge Corporation and University of California (Davis), are developing intrusion tolerant clusters.

The specific goal of our project (Hierarchical Adaptive Control of QoS for Intrusion Tolerance or HACQIT) is to provide continued COTS or GOTS-based application services in the face of multiple hours of aggressive cyber-attacks by a well-resourced adversary. This focus on COTS/GOTS applications means we do not have access to source code so the protections must be added around or to the binaries. We recognize that our defense cannot be perfect so two implied goals include (1) significantly increasing the adversary work factor for successful attacks and (2) significantly increasing the ratio of the attacker's work factor to generate successful attacks to the defender's work factor for responding to successful attacks. We also recognize that our system is expensive in terms of processing and overhead so we have modularized the components so that the amount of protection can be varied according to the need and budgets available.

1.2 Organization

The HACQIT project, its architecture, and basic intrusion tolerant design approach have been described in other articles [4, 5, 6]. The next section will provide enough information on HACQIT to enable the reader to understand the context, uses, and limitations of the learning and generalization as it exists today. The remaining sections will summarize the problem, the learning and generalization approach, its current implementation, test results, and conclusions / next steps.

2. HACQIT context

2.1 General problem and system model

Formal environment and attack assumptions have been made to specify the research problem as developing dependability in the face of network-based cyber attacks rather than dealing with denial of service attacks, insiders, Trojans and other lifecycle attacks. These assumptions include:

- Users and attackers interact with services via their computers and the LAN. There are no other hidden interactions.
- The LAN is reliable and cannot be flooded, i.e., denials of service (DoS) attacks against LAN bandwidth are beyond the scope of the research. The LAN is the only communication medium between users and services. DoS attacks directly against critical users or firewalls are also beyond the scope of the research.
- Critical users and the system administrators for the cluster are trusted. No hosts on the external LAN are trusted.
- The protected cluster hardware and software are free of Trojans and trapdoors at startup and have been patched against known vulnerabilities. Attackers do not have and have not had physical access to the cluster hardware or software. This prevents planting Trojan software/hardware and trapdoors through lifecycle attacks.
- Other unknown vulnerabilities exist throughout the system.

Figure 1 describes the "formal" system model of the problem and design environment that is being addressed by intrusion tolerant systems. The goal is to protect critical application(s) so that critical users can continue to access them while under attack.

2.2 HACQIT system architecture

HACQIT is not designed to be a general-purpose server connected to the Internet. Anonymous users are not allowed. All connections to the system are through authenticated Virtual Private Networks. We assume that the configuration of the system has been done correctly, which includes patching of all known vulnerabilities.

An attacker can be any agent other than the trusted users or HACQIT system administrators. Attackers do not have physical access to HACQIT cluster. An attacker may take over a trusted user's machine and launch attacks against HACQIT.

A failure occurs when observed behavior deviates from specified behavior. For HACQIT, we are concerned with software failures. Software failures are either repeatable or not. The causes of repeatable failures would include attacks (maliciously devised inputs) that exploit the some vulnerability (bug) in one of our software components. Non-repeatable failures may be caused by intermittent or transient faults. We cannot divine intent, so all inputs that cause repeated failures are treated the same. On the other hand, we recognize that the system may fail intermittently from certain inputs, in which case we allow retry.

To develop a system that meets these requirements, most designers would make the cluster very intrusion resistant, implement some type of specification-based monitoring of server and application behavior and use some set of fault tolerant mechanisms (e.g., redundancy and failover, process pairs, triple modular redundancy, n-version programming) for the servers to enable rapid failover and recovery. Our design employs these approaches and a few additional ones.

Our design is summarized in Figure 2. The HACQIT cluster consists of at least four computers: a gateway computer running a commercial firewall and additional infrastructure for failover and attack blocking; two or more servers of critical applications (one primary, one backup, and one of more on-line spares); and an Out-Of-Band (OOB) machine running the overall monitoring and control and fault diagnosis software. The machines in the cluster are connected by two separate LANs.

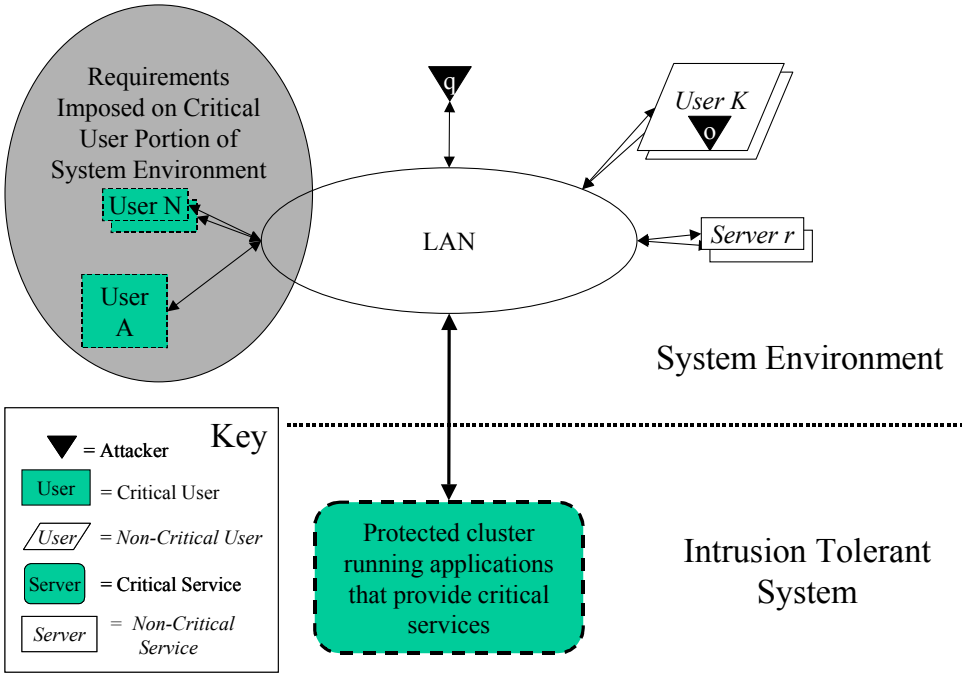

Fig. 1. Intrusion Tolerant System Environment

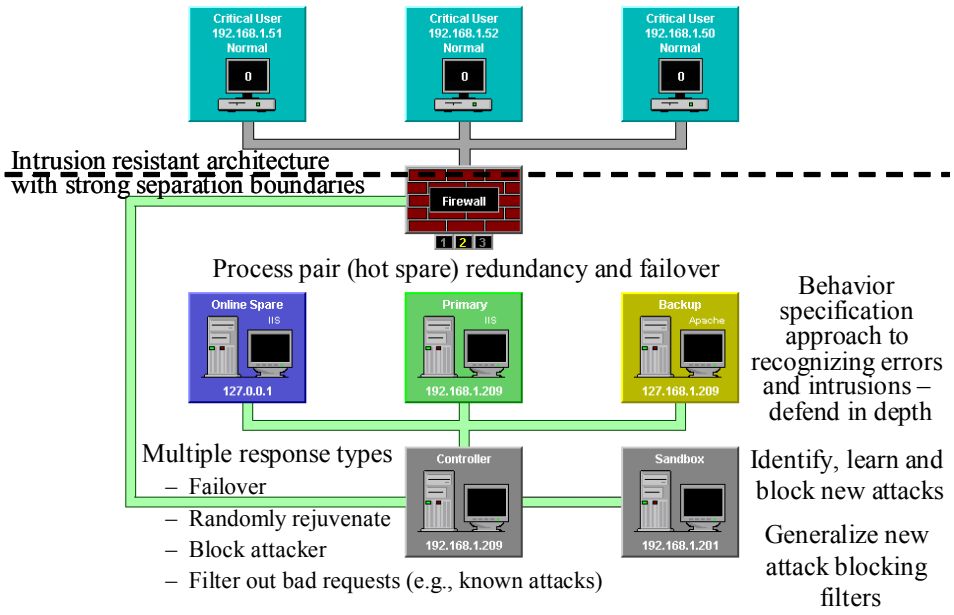

Fig. 2. Cluster design with learning components

HACQIT uses primary and backup servers running as a process pair, but they are unlike ordinary primary and backup servers for fault tolerance. Only the primary is connected to users. The virtual private network (VPN), firewall, gateway, and IP switch together ensure that users only talk to the critical application through the specified port on the primary server and vice versa. The primary and backup servers are not on the same LAN; they are isolated by the OOB computer, so no propagation of faults, for example by a automated worm or remote attacker, directly from the primary to the backup, is possible.

The potential for propagation from the primary to the Controller is limited by sharply constraining and monitoring the services and protocols by which the Controller communicates with the primary. When a failure is detected on the primary or backup server (possibly caused by an attack), it is taken off line. Continued service to the end user is provided by the remaining server of the process pair. A new process pair is formed with the on-line spare (if available), and both attack diagnosis and recovery of the failed server begins. Depending on policy, the Controller can also block future requests from the machine suspected of launching the attack.

The current critical application is a web-enabled message board that is duplicated on both the Microsoft IIS web server and the Apache web server machines. It contains dynamic data so HACQIT must maintain consistent state across the hosts and resynchronize data (checkpoint and restore) when failover and new process pair formation occurs. The spare server does not have current state when it is promoted into the process pair so a restore process is necessary to synchronize it.

2.3 HACQIT software architecture

The simplified software architecture is shown in Figure 3. The software implements a specification-based approach [7, 8] to monitoring and control for intrusion detection as well as defense in depth. It uses software wrappers [9], application and host monitors, response monitors, etc. to ensure that component behavior does not deviate from allowed. It does this in a protect-detect-respond cycle. Strong protections (and isolation) are melded with host and application-based fault, error, and failure detection mechanisms in a defense in depth design. Deviation from specified behavior may be indicative of an attack and thus, when such an alert is received, it triggers failover, integrity testing and other responses.

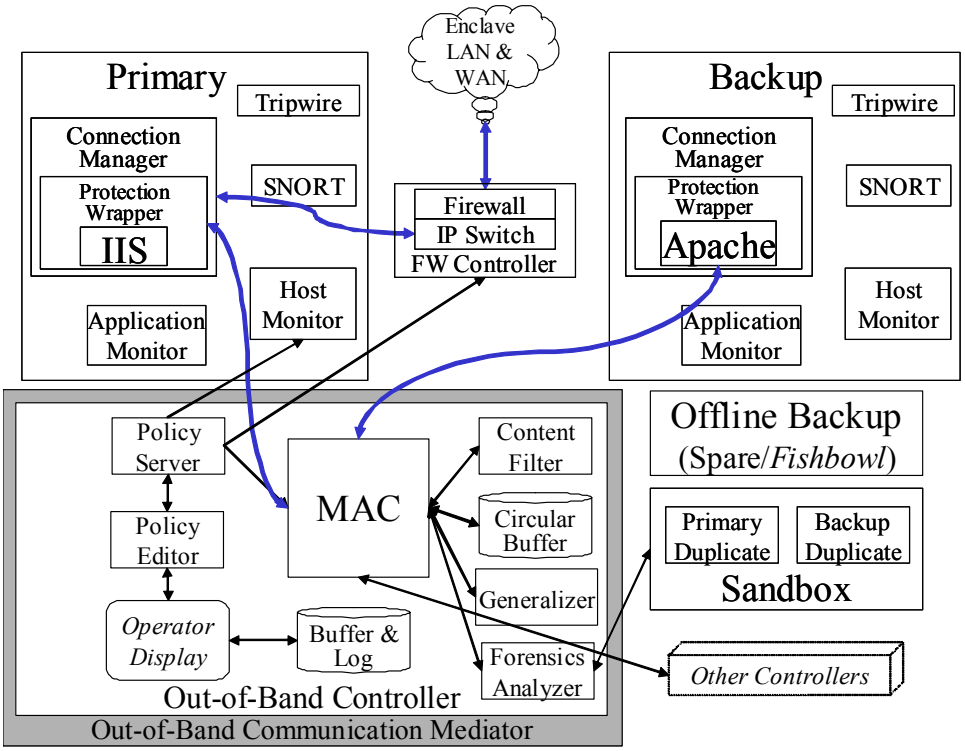

Fig. 3. Simplified software architecture

Wrappers are used to monitor and strictly control an application's ability to read and write files, execute or kill other processes, modify registry entries, change memory protection, etc. Any violations trigger an alert and failover response. The host monitor allows only specified processes to run and ensures that allowed processes do not consume more resources than allowed. It also monitors application liveness and generates a heartbeat

for the Controller. The out-of-band communication mediator ensures that only authorized processes use the out-of-band communications network to attack the mediator-adapter-controller (MAC) or any other software on the controller machine. Integrity of control messages is enhanced with digital signatures. The MAC monitors responses from both servers and periodically orders a "self test" through the Host Monitor (HM). This entails issuing a known request to the servers and checking its response against the known response.

The Out-of-Band Communication Mediator shown in Figure 3 is implemented by host firewalls (specifically the firewall by Tiny Software Inc.) on each machine. They only allow access to the server from the OOB machine by authorized programs like the MAC through authorized ports and only allow access to/from the gateway machine through port 80. Similarly, they only allow access to the OOB machine from the server by authorized programs like the wrappers and the HM through authorized ports. To change this configuration, a user must enter a password at the server's keyboard. An attacker would have to gain system privileges and by-pass or shutdown the host firewall to have access to the OOB machine. Creating an unauthorized process to remove a file would trigger an "Unhealthy" state by the HM and cause a failover and clean-up of the "Unhealthy" machine.

One of our innovations is that the Controller also performs random rejuvenation of each server on an average interval that is set by policy. During random rejuvenation, the Controller takes the selected machine off-line, performs the appropriate failover, starts up a new process pair with the spare machine, shuts down and restarts all applications on the rejuvenating machine, runs an integrity check on its files, and sets it up as an on-line spare when it is done. This techniques was originally intended to preserve the performance of the hosts and applications but it turns out to be a very powerful to avert latent successful attacks that have not been detected. It also limits the required size of the history buffer and limits the ability of any attack to go undetected.

3. The Problem of Unknown Attacks

Since we cannot ensure that software has no vulnerabilities, there will be unknown attacks that can succeed against the cluster. Our behavior specifications are tight but bad behavior must be observed by the wrappers or other sensors. An unknown attack can sneak past the defenses and compromise the protected web server. If the attack is stealthy enough to not execute any unauthorized processes, write any unauthorized files or use too much cpu or memory, it can remain undetected and active until the next random rejuvenation cycle when the system is purged.

While the emphasis of HACQIT is on availability of critical services, we need to say a few words about what an attacker can do in the above circumstances and what the countermeasures would be. First it would be easy to corrupt the critical application data since the web enabled application has permission to write to that file. The solution to this is a secure storage system such as that developed under the Perpetually Available and Secure Information Storage (PASIS) [10, 11]. Such a storage system captures all changes to files in a host-independent and secure manner. This enables recovery from such incidents.

Another possibility would be for the attacker to simply monitor what was happening within the application (spy) and exfiltrate data. Since the most likely avenue of attack is by compromising a critical user machine, the attacker would effectively have access to the critical application and data anyway. This essentially becomes the insider problem. Exfiltration via other routes is difficult because of the firewall settings and isolation of the cluster.

For the purposes of critical application availability, the central concern is that an attacker has found an unknown attack that can be used to penetrate the cluster. Such an attack can be used to shutdown the vulnerable web server or the application behind it. The HACQIT goal is to maintain at least 75% availability in the face of on-going attacks. For the attacker to win, all that he/she must do is to find a small set of vulnerabilities in each of the diverse web servers or other critical applications. This essentially guarantees that the attacker will succeed in shutting down the cluster more than 25% of the time. As long as that vulnerability remains and the exploit succeeds, the attacker can just keep hitting the cluster with it and cause another failover. It does not matter how expensive these vulnerabilities are to find, once they are found and exploits developed for them, the time to launch successive attacks is minimal. The results of this will be devastating on the defenders.

Even if the IP address of the attacker is blocked or that user cut off in other ways, the attacker can always come back unless the cluster is cut off from users. Such an action amounts to a self-inflicted denial of service and is clearly unacceptable. Since the attacker has the ability to automate his attack, even physically capturing the attacker would not necessarily stop the attacks. Since it takes time to clean up a server after an attack before it can be put back into service with any confidence, unless the cluster has an indefinitely large number of backup servers for failover, it seems like a losing game for the defender. If the attacker has found a simple, inexpensive way to vary the attack signature, the problem becomes even more difficult for the defender

Can this problem be fixed? In principle there is no solution. But, as the reader will see, we are using classical machine learning methods (using observed instances of the attack to learn the most general description of an attack that has variants, followed by the most general blocking rules) combined with the use of sandbox to experiment offline with the observed instances

to create other instances. Short of analyzing source (or object) code, that's the best we can do, and it is likely to be very effective. Our experiences with Code red and its variants can attest to this.

4. Solution concept

Cyber attacks (network-based intrusions into a system) have several important differences from other natural or man-made faults: They are repeatable, they are not random (although certain types of attacks may depend on timing of events), and, if known, they can be filtered out at system or sub-system boundaries. These distinctions enabled us to develop a set of learning techniques to help deal with the unknown attack problem.

Given an observed failure on a cluster server, our goal is to identify an attack in the recorded cluster traffic. Repeatability of the attack against the critical application server is the key criteria of an attack, particularly given the difficulty of establishing malicious intent. We developed a set of components that learn an attack after it is first used, develop blocking filters against it, and generalize those filters to disallow simple variants of the attack that depend upon the same vulnerability. By preventing reuse of an unknown attack, would-be adversaries are forced to develop a large number of new attacks to defeat the cluster for any significant period. This raises the bar significantly on the amount of effort that an adversary must expend to achieve more than momentary success.

Clusters can communicate with one another so that the protective filters developed at one site can be propagated to clusters at other sites that have not yet experienced the same attack. This ability to do group learning is a very powerful feature of the design and implementation.

The information necessary for the forensics based learning system to work is provided by several key components including (1) logs of all network inputs to the cluster for, at least, the last N minutes, (2) logs of all system sensor readings and responses that indicate errors or failures in components, and (3) a "Sandbox" for testing attack patterns and filters. The Sandbox is an isolated duplicate of the critical application servers, i.e., the redundant process-pair software, sensors, and hardware. Note that it is most effective if the number of minutes of buffering (N) is equal to or slightly greater than the number of minutes between random rejuvenation. Search speed is obvious faster if N is a smaller number of minutes rather than larger.

Our approach to identifying, learning, and blocking unknown attacks begins when an error (i.e., a deviation from specified behavior)) is observed in the cluster, usually associated with the critical application. It proceeds in parallel with and independent from the failover process that guarantees continuity of service.

Since our goal is to prevent the success of future versions of this newly observed unknown attack, it is not necessary to understand the details of the attack after the initiating event that puts control into the attacker's code. What we want to do is to prevent the initiating event, which is often a buffer overflow, and we would like to do this as quickly as possible.

While it is useful to have a general process with guaranteed convergence to a solution, the practical aspects of the time required to test many hypotheses of attack sequences against a Sandboxed application are formidable. It can take several minutes to restart some applications after a failure and some applications cannot be run in multiple processes on the same computer. Our more practicable approach involves examining a variety of heuristics and specification / model-based protocol analyzers that can be used to shrink the search space of suspect connection requests to a very small number of candidates that must be verified in the Sandbox.

Table 1. Steps in Learning and Generalization of Unknown Attacks

No.	Step Description
1.	Determine if observed error is repeatable based on connection history file since last rejuvenation. If repeatable, declare attack and continue. If not, return.
2.	Determine which connection request (or requests) from history file caused the observed error.
3.	Develop filter rule to block this connection request(s) pattern, test it, and send to content filter. Also block the associated user ID and IP address.
4.	Characterize the observed attack (i.e., classify it according to meaningful types).
5.	Shorten the blocking filter, if possible.
a.	Determine if the observed attack sequence has an initiating event
b.	If the initiating event is smaller than the observed attack sequence, shorten the blocking filter to block just the specific initiating event and test it.
6.	Based on characterization and observed attack specifics, generalize the blocking filter to protect against simple variants of the attack and test it.
7.	Return.

Given the observer error in the cluster, the essential functional steps in our learning and generalization "algorithm" are shown in the Table above.

The first two steps rapidly produce an initial filter rule that blocks the previously unknown attack. The remaining steps then incrementally improve the rules by shortening and generalizing them if possible.

A caveat is required here. Since we are dealing with Turing complete languages and machines, Rice's theorem

implies that we cannot prove intrusion tolerance for the system. Nevertheless, within the assumptions imposed on the system model, we believe we can deliver very useful and usable results.

The fundamental metric in determining the success or failure of the HACQIT cluster is whether an attacker can generate an effective attack rate higher than the cluster's effective learning and generalization rate. Intrusion resistance and intrusion tolerance don't have to be perfect. They just have to be good enough to convince the attackers to try a different, less expensive approach.

There are also several responses that the cluster controller can take to thwart attacker or to make learning easier. For example, random rejuvenation can be used to force an attacker to start over again with a stealthy attack. It is also useful for limiting the size of the history file that must be analyzed after a successful attack. It is also possible to cut off the attacker or "excessive user" via blocking his IP address at cluster or enclave level firewall. Since all users come into the cluster over a VPN and spoofing is not possible, this is particularly effective if the address or user ID of the attacker can be learned from the captured attack sequence in the history log. All inputs can be stopped for short periods if the attacks are overwhelming the system.

5. Analysis of Approach, Implementation, and Results

Each step in the learning and generalization process can be implemented differently for efficiency. For example, the most general approach to identifying an unknown attack once it has occurred is a search process (essentially "generate and test") using the Sandbox to establish truth [12]. Thus, Step 1 (determine if the component error was the result of an attack) could be implemented by simply rerunning the recorded service requests from the history log through the Sandbox.

We have found that it is faster to run a model-based service request checker against the log entries to eliminate "obviously correct" requests while retaining potentially suspicious requests. The actual selection rules are discussed later in this section. This approach speeds up both the "Was it an attack" testing and the subsequent "Which request(s) is the attack" testing by significantly reducing the search space. Since it takes significant time to restart a failed application or server, this is important. Generally, the more requests we can "test" analytically, rather than empirically in the Sandbox, the faster the search process.

While the last steps (shorten and generalize the filter rules) can be viewed as a search process, it is much more efficient to implement it using a knowledge-based matching approach. That is the reason for the classification step just before it. Since it is impossible to work with "unknown" attacks as a whole, we are working on meaningful taxonomies of unknown attacks to give us traction on the problem of reducing the search space. These sub-classes should be more amenable to informed search approaches and, even if we cannot identify all sub-classes of unknown attacks, we still obtain very useful search speed-up in the majority of practical cases. This is discussed further in the next two sections.

1. Using the component names shown in Figure 4, fault and intrusion tolerance are provided by the following control flow:
2. When the primary receives a request, it is forwarded to the MAC before it goes to the web server.
3. The MAC calls the Content Filter with the new request as the parameter. It also assigns a serial number and time stamp to the request.
4. The Content Filter, which contains a list of all bad requests received to date, checks the bad request list for a match with the new request.
5. If there is a match, the Content Filter returns false to the MAC (meaning "reject the request"); otherwise, it returns true ("accept request").
6. The MAC returns the answer (reject or accept) to the primary where the request is either forwarded to the web server or dropped as appropriate.
7. If the MAC received a true response from the Content Filter, it also forwards the request (with serial number and timestamp) to the backup server so the process pairs can be kept in sync.
8. Assuming that the request is legitimate, the MAC also waits for each server to process the request and then compares the return codes or times out if the response takes too long. If there is a mismatch or time out, the MAC response is policy driven. Under most circumstances, it will remove the suspect server from the process pair and begin normal failover and recovery steps.

The unknown attack identification and blocking functions are performed by learning components highlighted in Figure 4. These consist of the Forensics Analyzer, the Sandbox, the Circular Buffer and Event Logs, and the Content Filter and Generalizer.

The Content Filter examines each service request to determine if it matches a known bad request on its list of bad requests before it is passed to the primary or backup web servers. The Circular Buffer captures the last N-minutes of complete service requests from users to the critical servers while the event log captures sensor and event (such as failure) data.

If MAC detects error, it initiates failover and asks the Forensics Analyzer to start forensics. The MAC passes to it the history buffer, the state of the sensors and servers that caused the failover, and any likely candidates for the bad request. The Forensics Analyzer tests if any repeatable attacks are present in Circular Buffer by testing suspicious requests in the Sandbox. If it determines that an attack has occured, it then identifies specific attack request(s), and develops a new filter rule to block it. The

Generalizer improves content filter rules to stop simple variants of some types of attacks. More details on this are provided below.

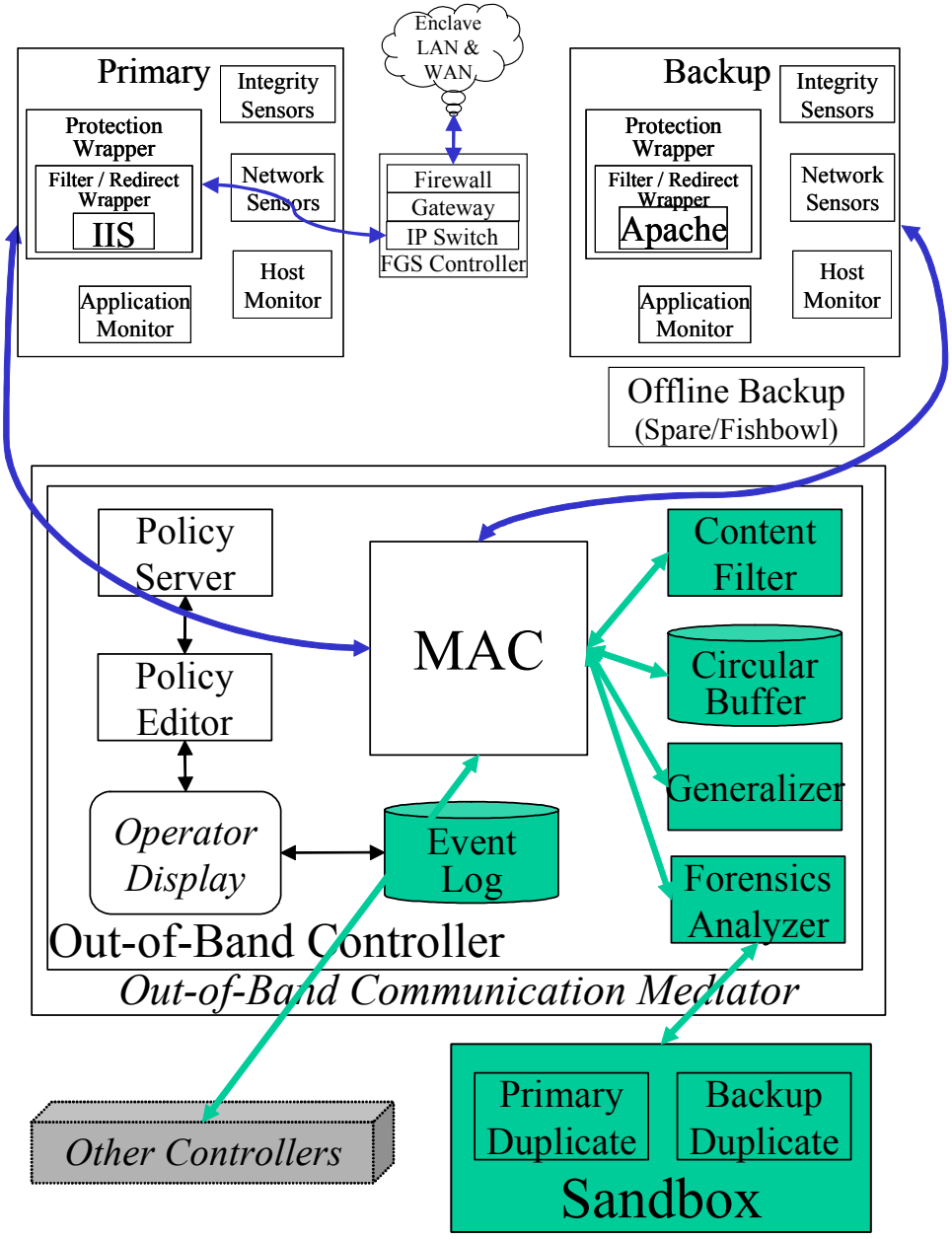

Fig. 4. Learning Components

There are also occasions when the MAC is able to determine relatively unequivocally that an intrusion has occurred. Examples include detecting unauthorized file writes or program executions. In many cases, the MAC itself can determine with high reliability which connection request is the likely attack. For example, if a particular request attempts to write an unauthorized file or start an unauthorized process, it is most likely an attack. In this case, the suspect request is forwarded to the Forensics Module as a prime candidate.

The Forensics Module looks in the circular buffer of past requests to identify suspicious requests. Illustrative rules for identifying suspicious web requests are shown in Table 2. Rule one is the result of the fact that many buffer overflow attacks use a repeated sequence of characters to move past the fixed length buffer. No valid HTTP transactions use methods other than GET or POST in our environment, thus rule two. This would obviously need to change when other methods are common. Attempts to access file types other than the standard set served are

classified as suspicious by rule four. Rule five classifies as suspicious those requests that use unusually long commands that are typically found in remote command execution attacks against server-side scripts. Unusual characters found in the request string are also a good indication of a suspect transaction, and are included in rule six. The `%` character is used for various encoding methods, such as hex encoding, and is very common in several classes of attacks. The `+` character is interpreted as a space. Many directory traversal attacks against Microsoft IIS servers include them. The `"..."` characters are also a sign of these types of attacks. The `<`, `>`, and `<` characters indicate cross site scripting attacks which attempt to inject Javascript into a webpage dynamically created by a script. The `//` characters can represent a subset of a long sequence of `/` characters which is an attempt to exploit an old Apache vulnerability or an attempt to proxy through the server.

Table 2. Illustrative Rules to Identify Suspicious Web Server Requests

Filtering rules to prioritize suspicious entries in web server transaction log
1) Repeated characters > 50
2) HTTP method not GET or POST
3) Protocol header other than HTTP/1.0 or HTTP/1.1
4) File extension other than `htm, html, txt, gif, jpg, jpeg, png,` or `css`
5) Command length > 20
6) Request string contains any of `%, ?, +, .., //, <, >, <, ;`

The Forensics Module then determines which suspicious request (or requests) was responsible for the observed symptoms of the attack by testing each in the Sandbox. If there is no repeatable error, the Forensics Module returns. If there is a repeatable error, it has determined what request should be blocked in the future. The forensics module then passes the known bad request to be blocked to the MAC, which calls the UpdateBadReq method of ContentFilterBridge (which implements the Content Filter) with the bad request as the parameter. UpdateBadReq adds the bad request to a static bad request list in memory and writes it to the bad request file. Currently, requests are truncated to the first two components of an HTTP request, namely, the method and URI.

Every time a request is received on the primary, it is forwarded to the MAC. The MAC calls the AllowRequest method of ContentFilterBridge with the new request as the parameter. The method checks the bad request list for an exact match with the new request. If there is a match, it returns false to the MAC, meaning block the request; otherwise, it returns true.

Thus far the learning is straight-forward and quite general. Unfortunately, the attack pattern that is being blocked is quite specific. If simple attack variants can be produced easily (e.g., by changing the padding characters in a buffer overflow attack or changing the order of commands in a cgi-script attack), then this specific learning approach is easily circumvented by an attacker. What is needed is a way to rapidly generalize the observed attack pattern so as to block all simple variants of an attack that are based on the same vulnerability initially exploited. This is a challenging area and is the subject of a continuing research effort.

As a proof-of-concept, we implemented generalization for a common but prevalent class of attacks: web server buffer overflows. Our initial approach was to enhance the AllowRequest method so that if an exact match is not found, it then analyzes the components of the requests (both new and bad) to determine if the new request is "similar" to a known bad request. If it is similar, AllowRequest returns false; otherwise, it returns true. In this way, learning is generalized from specific requests that have been identified as bad.

In principal, similarity is rule based and consists of two steps: classification and generalization. Classification categorizes bad requests into meaningful types such as buffer overflow or remote command execution and, as required, further into sub-types. Generalization develops a set of rules for determining similarity between an observed bad request and a new request based on the classification results. These rules can be implemented either as an active checking process or as comparison templates for use by another program.

For the proof-of-concept on web server buffer overflow attacks via http requests, we implemented one rule that acts as both a classifier and a generalizer. It is the following:

If (1) the query length of the bad request is greater than (256+X) [this part of the rule classifies the request as a buffer overflow type[1]] and (2) the methods of the new request and the bad request are the same and (3) the file extensions of the new and bad requests are the same and (4) the query length of the new request is greater than or equal to the query length of the bad request, then return false (i.e., block the request).

Even with X=0 in this rule, many variants of Code Red I and II are blocked. The initial or padding characters in the query are irrelevant to how Code Red works; the length is critical; so whether "XXX..." or "NNN..." or "XNXN..." are in the query of the attack, the attack is blocked. In addition, the name of the file (minus the extension) is also irrelevant to how Code Red works, because it is the file extension that identifies the resource (Index Server) that is vulnerable to a buffer overflow, and it is the query that causes the buffer overflow, not the

[1] X starts out equal to zero. Its role will be discussed later.

entire URI. (The URI contains the path identifying the resource and, optionally, the query.)

The reason for the first condition in the rule is to differentiate in a trivial way between bad requests that are buffer overflow attacks and bad requests that are some other type of attack, like remote command execution. Unfortunately, it introduces the possibility of false negatives, that is, a bad request that was a buffer overflow attack, but with the overflow occurring after less than 256 characters, would be ignored as an example to be generalized.

This rule has been constructed from extensive analysis of buffer overflows in general, buffer overflows in IIS and Apache web servers, and Code Red, in particular. Note that it only generalizes "learned" behavior. That is, if the HACQIT cluster has never been attacked by Code Red, it will not stop the first Code Red attack. It will also not stop the first case of a variant of Code Red that uses the .IDQ extension[2]. This variant would first have to be "experienced", learned as a bad request, and then generalized by the above rule. Most importantly, the rule does not prevent use of a resource like Index Server; it prevents a wider variety of attacks that exploit an identified vulnerability in it from reducing availability of the web server.

Although this rule appears Microsoft-oriented, as the concept of file extensions does not exist under Unix, it would work against attacks exploiting vulnerabilities in other software, such as php and perl, because these resources also use file extensions. It might be possible to generalize this to file types under Unix. The key distinction to be made is, does the path in the URI identify a document to be returned to the client or does it identify an executing resource such as a search engine, a DBMS, etc.?

Finding the minimum length of padding characters for a buffer overflow attack is not difficult. We have implemented an enhanced version of the forensics and generalization modules that iteratively tests attack variants in the Sandbox with different padding character lengths. Specifically it successively tests padding character lengths between 256 and (Y-256) where Y is the length of the observed buffer overflow padding size. From this testing, it determines the value of X (which appears in the first condition of the generalization rule above) and passes it to the ContentFilterBridge for inclusion in the revised generalization rule. The observed padding size is currently determined by the number of characters before the first non-printing character (i.e., not ACSII character coder 32 through 126) in the query. While this is only an approximation that depends on certain assumptions being true, it proved to be a very useful approach for the proof-of-concept implementation. Our investigation with Code Red II shows the padding in the query that causes the buffer overflow is no more than one byte over the minimum required; that is, if you remove two characters from the query, a buffer overflow will not occur, and IIS will respond to the request correctly and continue to function according to specification.

It is worth comparing this automatic generalization with Snort's hand-coded rules for preventing Code Red attacks. Snort is widely used, open source, lightweight Intrusion Detection System. Immediately after the flurry of initial Code Red attacks, Snort aficionados began crafting rules to block these attacks. It took at least two days before rules were posted on the Snort site. These were not generalized and did not work against trivial variants. Some three months later, the rules block on ".ida" and ".idq" in the URI and "payload size" greater than 239 [13]. The use of the file extensions shows some generalization but the use of 239 as a limit on legitimate requests intended for Index Server in fact cause false positives because the payload can be much greater than 239 (at least 373) without causing the web server to fail.

Other improvements to generalization would use analysis based on HTTP headers and body content. These and other improvements are the central focus of the next phase of research.

One additional aspect of the design of the ContentFilterBridge software is worth discussing. It first calls AllowRequest with the bad request received from the MAC. If AllowRequest returns true, that means the bad request is not on the bad request list, so it is added. If AllowRequest returns false, this means it is on the bad request list, so it is not added to the list. This prevents duplication.

With the addition of generalization, not only will duplicates be prevented, but also trivial variants will not extend the bad request list to a performance-crippling length. As there are over 2^{1792} (or more than the number of atoms in the universe) variants of Code Red, this is an important and effective aspect of the design.

6. Next steps

6.1 Software Improvements

In its initial implementation, the Forensics module truncates bad requests to the first two components of the HTTP request, namely, the method and URI. This makes sense in the case of the buffer overflows on web servers but it needs to be enhanced so there is a more robust way to identify the initiating event of an attack. In addition,

[2] Index Server uses file types indicated by the extensions, ".IDA" and ".IDQ". These two extensions are used by IIS to identify the Index Server resource, which is then passed either the whole URI or the query component of the URI. The "path" component of the URI does not affect the behavior of the Index Server, except for the file extension identifying it as the resource target. Any file name other than "default" in "default.ida" works as well.

there is much work to do to enhance the Forensics module's process for finding initial attack sequences efficiently, especially for multi-request attacks.

Similarly, the initial generalization rule base will be moved into a separate Generalization module that reflects the architecture. This module will attempt to generalize all requests or patterns returned by the forensics module to the content filter and insert specific new rules into the content filter. More broadly, we want the Generalizer to be able to task the Forensics Module to run Sandbox tests on any proposed set of filter rules and generalization parameters to what works, e.g., which contain the essential initiating event. In this way, we can refine the generalization while providing continued protection at the Content Filter level.

There is a great deal of work to be done in developing rules for generalizing attack patterns so that simple attack variants won't work. We would like to do this by focusing on meaningful attack classes. The literature contains many works on classifying various aspects of computer security including fault tolerance, replay attacks in cryptographic protocols, inference detection approaches, COTS equipment security risks, and computer attacks. Essentially all of these authors have emphasized that the utility of a taxonomy depends upon how well it accomplishes its purpose and that there is no such thing as a universal taxonomy.

Another module that we will likely need is one that allows us to simulate vulnerabilities in applications and generate resulting sensor reading. It is difficult to rely on real world attacks on our specific applications. There are simply not enough of them in circulation to give us the breadth of attack types that we need for the research.

6.2 Theory Improvements

As Krsul [14] states, "Making sense of apparent chaos by finding regularities is an essential characteristic of human beings." He laid out the essential characteristics of successful taxonomies: (1) They should have explanatory and predictive value. (2) Computer vulnerability taxonomies should classify the features or attributes of the vulnerabilities, not the vulnerabilities themselves. (3) Their classes should be mutually exclusive and collectively exhaustive. (4) Each level or division should have a *fundamentum divisionis* or basis for distinction so that an entity can be unequivocally placed in one category or the other. (5) The classification characteristics should be objective, deterministic, repeatable, and specific. Note that item (3) above is very difficult to achieve in practice outside the realm of mathematics and should be probably be replaced by extensibility as a goal.

Krsul developed a very extensive list of classes particularly focused on erroneous environmental assumptions. Unfortunately, his and most of the previous efforts (see review by Lough [15]) on developing taxonomies have focused on identifying and characterizing vulnerabilities in source code so that programmers could identify and eliminate them before the software was deployed. At one level this are fine in that they can give us insights into types of vulnerabilities. For example, the classic study by Landwehr et al. [16] lists the following types of inadvertent software vulnerabilities:

1. Validation error (incomplete/inconsistent)
2. Domain error (including object re-use, residuals, and exposed representation errors)
3. Serialization/aliasing (including TOCTTOU errors)
4. Identification/authentication errors
5. Boundary condition violation (including resource exhaustion and violable constraint errors)
6. Other exploitable logic errors

While these are important efforts and give us insights, we really need a taxonomy of remote access attacks, particularly one that characterizes the initiating events that can be exploited via network-based attacks on COTS or GOTS software.

Since our focus is on unknown network-based attacks, recent work by Richardson [17] is of interest. He developed a taxonomy for DoS attacks that identifies the following attack mechanisms:

1. Buffer overflows
2. IP fragmentation attacks
3. Other incorrect data attacks
4. Overwhelm with service requests
5. Overwhelm with data
6. Poor authentication or access control
7. Poor authentication scheme
8. IP spoofing
9. Data poisoning

Other miscellaneous protection shortcomings)

These categories will be informed by other studies of taxonomies [e.g., 18, 19]. The results will form the initial basis for our categorization of initiating events of unknown attacks. Priorities will be given to those attacks that are known not to have adequate protection measures built into the cluster currently and for which there are not easy fixes to the design that would prevent them. For example, IP fragmentation attacks against the primary can be prevented with a proxy on the firewall or gateway and IP spoofing is prevented by the VPN.

7. Conclusions and recommendations,

Our design for an intrusion tolerant server cluster uses a behavior specification-based approach to identify errors and failover to the hot spare. It then uses fault diagnosis to recognize the attack that caused failover (or violated QoS) and block it so repeated attacks won't defeat us again. We learn exact attacks by testing entries from complete log files in a "Sandbox" until we duplicate the observed failure. Single stage attacks can be recognized in seconds, automatically.

We have demonstrated that it is possible to generalize web server buffer overflow attack signatures after the initial identified attack so that simple variants that exploit the same vulnerability will be blocked also. We do this using a similarity measure for the class of attack. We have implemented rules that generalize a large subset of buffer overflow attacks aimed at web servers and have tested it using the Internet Information Server (IIS) by Microsoft, and believe that it will also work for Apache and other web servers also. For buffer overflow attacks, which have become the most common type of attack, we can also learn the minimum length of the request that causes the buffer overflow. This is important to minimize the probability of blocking legitimate transactions, i.e., the false positive rate.

We believe this knowledge-based learning is broadly applicable to many classes of remote access attacks and has significant uses outside of intrusion tolerance. We also believe that the generalization approach can be significantly extended to other classes of attack. The key, we believe, is generalizing an attack pattern to protect against all variants that exploit the same vulnerability rather than trying to generalize a specific attack to protect against all such attacks in the class. The ease of generalizing an attack pattern should be proportional to the ease of creating simple attack variants that work against the same vulnerability.

In summary, we have developed an approach to dynamic learning of unknown attacks that shows great promise. We have also implemented a proof of concept for generalization that works for a significant class of buffer overflow attacks against web servers on Microsoft NT/2000. Our results so far indicate that the generalization algorithms will be specific to particular types of attacks (such as buffer overflow), to particular protocols (such as http) and to particular application classes. More work is needed to determine whether they must be specific to particular applications but that is a likely outcome if the application class is not dominated by standard protocols.

We recommend that other researchers examine this knowledge-based approach to identifying unknown attacks. We hope they find it useful enough to apply it to other areas.

8. References

1. Schneier, B: *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., 2000 206, 210

2. Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, 1993 107

3. Lampson, B.: "Computer Security in the Real World". Invited essay at 16th Annual Computer Security Applications Conference, 11-15 December, New Orleans, LA, available at http://www.acsac.org/2000/papers/lampson.pdf

4. Just, J.E., et al.: "Intelligent Control for Intrusion Tolerance of Critical Application Services". *Supplement of the 2001 International Conference on Dependable Systems and Networks*, 1-4 July 2001, Gothenburg, SW

5. Reynolds, J., et al.: "Intrusion Tolerance for Mission-Critical Services". *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 13-16, 2001, Oakland, CA

6. Reynolds, J., et al.: "The Design and Implementation of an Intrusion Tolerant System". *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 23-26 June 2002, Washington, DC, pending

7. Ko, Calvin: "Logic Induction of Valid Behavior Specifications for Intrusion Detection". *IEEE Symposium on Security and Privacy 2000*: 142-153

8. Ko, Calvin, Brutch, Paul, et al.: "System Health and Intrusion Monitoring Using a Hierarchy of Constraints". *Recent Advances in Intrusion Detection 2001*: 190-204

9. Balzer, R., and Goldman, N.: "Mediating Connectors". *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 31-June 4, 1999, IEEE Computer Society Press 73-77

10. Strunk, J.D., et al.: "Self-securing storage: Protecting data in compromised system". *Operating Systems Design and Implementation*, San Diego, CA, 23-25 October 2000, USENIX Association, 2000 165-180

11. Ganger, G.R., et al.: "Survivable Storage Systems". *DARPA Information Survivability Conference and Exposition* (Anaheim, CA, 12-14 June 2001), pages 184-195 vol 2. IEEE, 2001

12. Russell, S., Norvig, P,: *Artificial Intelligence: A Modern Approach*. Prentice Hall, New York, 1995

13. Roesch, M.: *Snort Users Manual*, Snort Release: 1.8.3. November 6, 2001, available at http://www.snort.org/docs/writing_rules/

14. Krsul, I.V.: *Software Vulnerability Analysis*. PhD thesis, Purdue University, West Lafayette, IN, May, 1998, p. 17, available at https://www.cerias.purdue.edu/techreports-ssl/public/97-05.pdf

15. Lough, D.L.: *A Taxonomy of Computer Attacks with Applications to Wireless Networks*. PhD Thesis, Virginia Polytechnic and State University, Blackburg, VA, available at http://scholar.lib.vt.edu/theses/available/etd-04252001-234145/

16. Landwehr, C. E., Bull, A. R., McDermott, J. P., Choi, W. S.: "A Taxonomy of Computer Program Security Flaws". *ACM Computing Surveys*, Volume 26, Number 3, September 1994

17. Richardson, T.W.: *The Development of a Database Taxonomy of Vulnerabilities to Support the Study of Denial of Service Attacks*. PhD thesis, Iowa State University, 2001

18. Aslam, T.: *A Taxonomy of Security Faults in the Unix Operating System*. Master's Thesis, Purdue University, Department of Computer Sciences, August 1995. http://citeseer.nj.nec.com/aslam95taxonomy.html
19. Du, W. and Mathur, A.: *Categorization of Software Error that Led to Security Breaches*. Technical Report 97-09, Purdue University, Department of Computer Science, 1997