An Architecture for an Adaptive Intrusion-Tolerant Server [*]

Alfonso Valdes, Magnus Almgren, Steven Cheung, Yves Deswarte[**], Bruno Dutertre, Joshua Levy, Hassen Saïdi, Victoria Stavridou, and Tomás E. Uribe

System Design Laboratory,
SRI International,
333 Ravenswood Ave., Menlo Park, CA 94025
`valdes@sdl.sri.com`

Abstract. We describe a general architecture for intrusion-tolerant enterprise systems and the implementation of an intrusion-tolerant Web server as a specific instance. The architecture comprises functionally redundant COTS servers running on diverse operating systems and platforms, hardened intrusion-tolerance proxies that mediate client requests and verify the behavior of servers and other proxies, and monitoring and alert management components based on the EMERALD intrusion-detection framework. Integrity and availability are maintained by dynamically adapting the system configuration in response to intrusions or other faults. The dynamic configuration specifies the servers assigned to each client request, the agreement protocol used to validate server replies, and the resources spent on monitoring and detection. Alerts trigger increasingly strict regimes to ensure continued service, with graceful degradation of performance, even if some servers or proxies are compromised or faulty. The system returns to less stringent regimes as threats diminish. Servers and proxies can be isolated, repaired, and reinserted without interrupting service.

1 Introduction

The deployment of intrusion-detection technology on mission-critical and commercial systems shows that perfect detection and immediate mitigation of attacks remain elusive goals. Even systems developed at great cost contain residual faults and vulnerabilities. In practice, emphasis must shift from unattainable "bulletproof" systems to real-world systems that can tolerate intrusions. An *intrusion-tolerant* system is capable of self-diagnosis, repair, and reconstitution, while continuing to provide service

[*] This research is sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency. Approved for Public Release—Distribution Unlimited.

[**] LAAS-CNRS, Toulouse, France

to legitimate clients (with possible degradation) in the presence of intrusions [6]. This paper describes the conceptual architecture of such a system, and our experience with its initial implementation. Our design integrates concepts from distributed intrusion detection, fault tolerance, and formal verification.

Our primary concerns are availability and integrity: the system must remain capable of correctly servicing requests from honest clients even if some components are compromised. The architecture is also intended to be scalable: servers and proxies can be added to improve the system's performance and reliability, depending on the resources available.

1.1 Architecture Outline

The architecture, shown in Figure 1, consists of a redundant *tolerance proxy* bank that mediates requests to a redundant *application server* bank, with the entire configuration monitored by a variety of mechanisms to ensure content integrity, including intrusion-detection systems (IDSs). The proxies and application servers are redundant in capability but diverse in implementation, so that they are unlikely to be simultaneously vulnerable to the same attack. The application servers provide the application-specific functionality using COTS products, thus reducing the cost of specialized custom-built components. The remainder of the configuration provides intrusion tolerance by detecting, masking, and recovering from attacks or nonmalicious faults.

The system is designed to provide continued reliable and correct service to external clients, even if a fraction of the application servers and proxies are faulty. The faults we consider include transient and permanent hardware faults, software bugs, and intrusions into application servers and tolerance proxies. The architecture can support a variety of client-server systems, such as database applications or Web content distribution, under the assumptions outlined below.

An important aspect of our system is a distributed management function that takes actions in response to adverse conditions and diagnoses. All platforms and network interfaces within the system are instrumented with a variety of monitors based on signature engines, probabilistic inference, and symptom detection. Given the reports from this monitoring subsystem, the management function undertakes a variety of tolerance policy responses. Responses include enforcing more stringent agreement protocols for application content but reducing system bandwidth, filtering out requests from suspicious clients, and restarting platforms or services that appear corrupt.

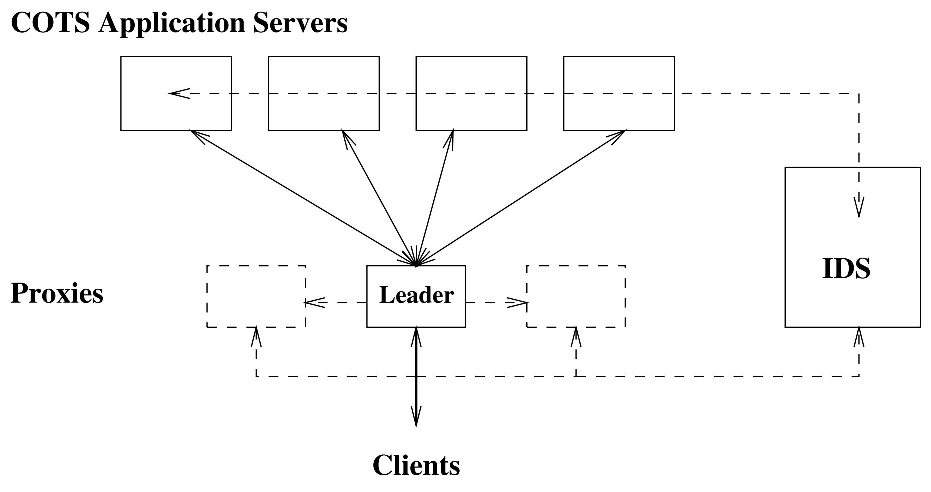

Fig. 1. Schematic view of the intrusion-tolerant server architecture

1.2 Assumptions

We assume that attackers do not have physical access to the configuration. We assume that no more than a critical number of servers are in an undetected compromised state at any given time.

Our agreement protocols assume that all nonfaulty and noncompromised servers give the same answer to the same request. Thus, the architecture is meant to provide content that is static from the end user's point of view. The reply to a request can be the result of substantial computation, on the condition that the same result be obtained by the different application servers. Target applications include plan, catalog, and news distribution sites.

The system content can be updated periodically, by suspending and then resuming the proxy bank, but we do not address specific mechanisms for doing so (noting only that online content updates can introduce new vulnerabilities). Survivable storage techniques [16,23] can be used in the future to build a separate subsystem to handle write operations, while retaining the current architecture for read requests.

The architecture focuses on availability and integrity, and does not address confidentiality. We do not defend against insider threat or network-flooding denial-of-service attacks.

1.3 Paper Outline

Section 2 describes the basic building blocks of our architecture. Section 3 describes the monitoring mechanisms, which aim to provide a picture of the current system state, including suspected intrusions and faults. Section 4 describes how this information is used to respond to threats, adapting the system configuration to the perceived system state. Section 5 describes the details of our implementation, and Section 6 presents our conclusions, including related and future work.

2 Architecture Components

2.1 Application Servers

In our architecture, the domain-specific functionality visible to the end user (the *client*) is provided by a number of *application servers*. These provide equivalent services, but on diverse application software, operating systems, and platforms, so that they are unlikely to be vulnerable to common attacks and failure modes. They include IDS monitors but are otherwise ordinary platforms running diverse COTS software. In our instantiation (see Section 5), these servers provide Web content.

For our content agreement protocols to be practical, we assume there are at least three different application servers; a typical number would be five or seven. However, the enterprise can add as many of these as desired, increasing the overall performance and intrusion-tolerance capabilities.[1]

2.2 Tolerance Proxies

The central components of our architecture are one or more *tolerance proxies*. Proxies mediate client requests, monitor the state of the application servers and other proxies, and dynamically adapt the system operation according to the reports from the monitoring subsystem (described in Section 3).

One of the proxies is designated as the *leader*. It is responsible for filtering, sanitizing, and forwarding client requests to one or more application servers, implementing a *content agreement protocol* that depends on the current *regime*, while balancing the load. In the presence of perceived intrusions, increasingly stringent regimes are used to validate server replies. The regime is selected according to a chosen *policy*, depending on

[1] The architecture should be modified for large numbers of application servers, as discussed in Section 6.

reports from the monitoring subsystem and on the outcome of the agreement protocol currently in use.

Optional *auxiliary proxies* monitor all communication between the proxy leader and the application servers, and are themselves monitored by the other proxies and the sensor subsystem. If a majority of proxies detect a leader failure, a protocol is executed to elect a new leader. Our current design and implementation focuses on the case of a single proxy. However, we discuss inter-proxy agreement protocols further in Section 4.4.

The tolerance proxies run only a relatively small amount of custom software, so they are much more amenable to security hardening than the more complex application servers, whose security properties are also more difficult to verify. We believe there is greater return, both in terms of security and utility, in expending effort developing a secure proxy bank than in developing hardened application servers. Since the proxies are mostly application-independent, their development cost can be amortized by re-using them in different domains.

2.3 Intrusion-Detection System

The third main component of our architecture is an *intrusion-detection system* (IDS), which analyzes network traffic and the state of the servers and proxies to report suspected intrusions. Some IDS modules execute on one or more dedicated hardware platforms, while others reside in the proxies and application servers. We describe our IDS capabilities in more detail in Sections 3.1 and 5.1.

2.4 Functional Overview

When a client request arrives, the following steps are performed:

1. The proxy leader accepts the request and checks it, filtering out malformed requests.
2. The leader forwards the request, if valid, to a number of application servers, depending on the current agreement regime.
3. The application servers process the request and return the results to the proxy leader. If sufficient agreement is reached, the proxy forwards the content to the client.
4. The regime is adjusted according to the result of the content agreement and reports from the monitoring subsystem.
5. The auxiliary proxies, if present, monitor the transaction to ensure correct proxy leader behavior.

3 Monitoring Subsystem

The *monitoring subsystem* includes a diversified set of complementary mechanisms, including the IDS. The information collected by the monitoring subsystem is aggregated into a global system view, used to adapt the system configuration to respond to suspected or detected threats and malfunctions, as described in Section 4. Diversity helps make the monitoring subsystem itself intrusion-tolerant, since it may still be effective if some of its components fail.

3.1 Intrusion Detection

Our intrusion-detection systems feature diverse event sources, inference techniques, and detection paradigms. They include EMERALD host, network, and protocol monitors [19,15,20,29], as well as embedded application monitors [1].

Different sensors cover different portions of the detection space, and have different detection rates, false alarm ratios, and operational conditions (e.g., the maximum rate of incoming events that can be handled). Their combination allows detecting more known attacks, as well as anomalies arising from unknown ones. The advantages of heterogeneous sensors come at the cost of an increased number of alerts. To effectively manage them, they must be aggregated and correlated [30,18]. Alert correlation can also detect attacks consisting of multiple steps.

3.2 Content Agreement

The proxy leader compares query results from different application servers, according to the current agreement regime, as described in Section 4.1. If two or more results fail to match, this is viewed as a suspicious event, and suspect servers are reported.

3.3 Challenge-Response Protocol

Each proxy periodically launches a *challenge-response protocol* to check the servers and other proxies. This protocol serves two main purposes:

- It provides a heartbeat that checks the liveness of the servers and other proxies. If a proxy does not receive a response within some delay after emitting a challenge, it raises an alarm.
- The protocol checks the integrity of files and directories located on remote servers and proxies.

The integrity of application servers is also verified indirectly by content agreement, as mentioned above. However, a resolute attacker could take control of several servers and modify only rarely used files. The challenge-response protocol counters this by reducing file error latency, detecting file modifications before they return incorrect content to the users.

As an integrity check, the challenge-response protocol is similar to Tripwire$^{\text{TM}}$ [28]. For each data item whose integrity is to be checked, a checksum is computed from a secret value (the challenge) and the content of the item. To resist possible guesses by an attacker, the checksum is computed by applying a one-way hash function to the concatenation of the challenge and the content to be checked. The resulting checksum is then compared with a precomputed one. This is sufficient to check if static data items have been modified.

However, we must also consider the possibility that an attacker with complete control of a server or proxy modifies the response program to return a correct challenge response for a file incorrectly modified. In particular, if the challenge is always the same, it is easy for the attacker to precompute all responses, store the results in a hidden part of the memory or disk, and then modify the data. The attacker would then be able to return correct responses for incorrect files.

To guarantee the freshness of the response computation, a different challenge is sent with each request. The proxy could check that each response corresponds to the specified challenge by keeping a local copy of all sensitive files and running the same computation as the server, but this imposes an extra administrative and computational load on the proxy.

Instead, we can exploit the fact that servers and proxies are periodically rebooted (e.g., once a day), as a measure for software rejuvenation [11]. In this case, a finite number C of challenges can be generated offline for each file to be checked, and the corresponding responses computed offline, too. The results are then stored on the proxy. At each integrity-check period, the proxy sends one of the C challenges and compares the response with the precomputed result. The challenges are not repeated, provided

$$C > (\text{time between reboots}) \times (\text{frequency of challenge-response protocol}) \ .$$

An attacker could circumvent this mechanism by keeping copies of both the original and the modified files. But such a large-footprint attack should be detected, either by checking the integrity of the concerned directories, or by the host monitor IDS.

3.4 Online Verifiers

As part of the design process, we express the high-level behavior of the proxy as a *reactive system* that can be formally verified. An abstraction of the system is described using a finite-state omega-automaton and the properties of interest are specified in temporal logic. The high-level specifications can be formally verified using model checking [10]. However, this does not guarantee that the implementation (the *concrete* system) meets the corresponding requirements.

To fill this gap, we introduce *online verifiers*, which check that the abstract properties hold while the concrete system is running, by matching concrete and abstract states. If an unexpected state is reached, an alarm is raised. Only *temporal safety* properties can be checked in this way [27]; however, the challenge response heartbeat described in Section 3.3 provides a complementary liveness check.

The online verifiers are generated by annotating the proxy C program source. Since type-safeness is not guaranteed, and only high-level properties are checked, this does not detect lower-level faults, such as buffer overflows. We will apply complementary mechanisms at this level, such as compilation using the StackGuard tool [2].

4 Adaptive Response

We now describe how the system responds to state changes reported by the monitoring subsystem described in the previous section.

4.1 Agreement Regimes and Policies

A main role played by the proxy leader is to manage the redundant application servers. The proxy decides which application servers should be used to answer each query, and compares the results. The number of servers used trades off system performance against confidence in the integrity of the results.

Figure 2 presents the main steps in the *content agreement protocol* executed by the proxy leader. This protocol is parameterized by an *agreement regime*, which must specify, at each point in time:

1. Which application servers to forward the request to
2. What constitutes sufficient agreement among the replies
3. Which servers, if any, to report as suspicious to the monitoring subsystem

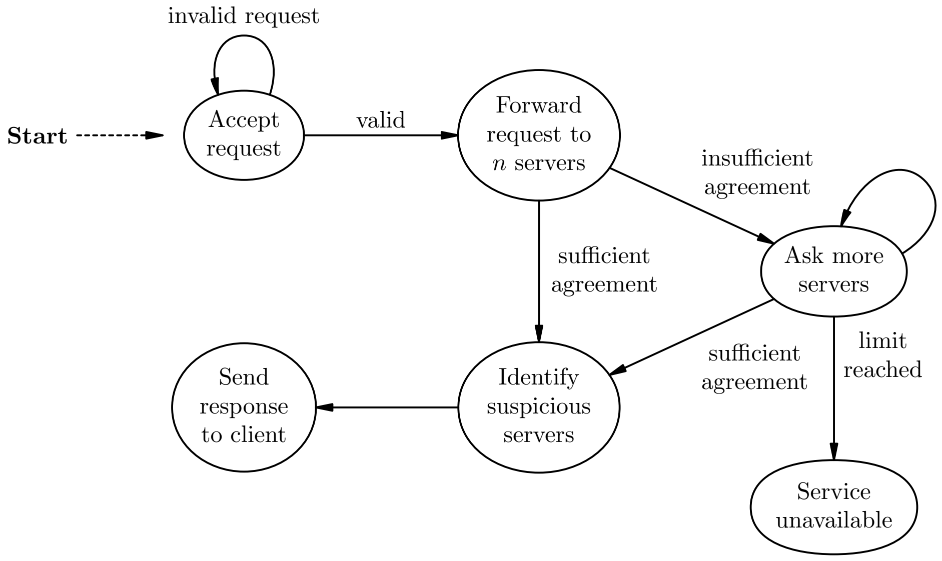

Fig. 2. Generic content agreement protocol

A good agreement regime should perform load balancing and choose the application servers as randomly as possible, so attackers cannot always predict which server will service a particular request.

Most important, the regime should be dynamically adjusted in response to alerts: a *policy* specifies the action to take next if no agreement is achieved, and which regime to use in response to various events. A policy must also specify how to respond if intrusions or other adverse conditions are detected, and when to return to a less stringent agreement regime. The transition to a stricter regime can also occur as a result of administrative action (e.g., if the administrator is alerted by external sources).

Example 1 (Simple agreement regime). Given an architecture with N application servers, the *simple agreement regime* n, for $1 \leq n \leq N$, specifies that each client request be forwarded to n different application servers, randomly chosen, that are not considered compromised. Sufficient agreement is reached if a majority of servers (at least $\lfloor n/2 \rfloor + 1$) agree; machines in the minority are reported as suspicious. The absence of a reply is counted as a vote in the minority.

We refer to simple agreement regimes 1, 2 and 3 as *single*, *duplex* and *triplex modes*, respectively. The agreement regime where each client request is sent to all application servers is referred to as *full mode*. More complex regimes are possible:

Example 2 (Probabilistic agreement regime). A *probabilistic agreement regime* assigns to each server a real-valued *confidence*, as an estimated probability of noncompromise. Given a request, the leader chooses a set of servers such that the sum of their confidences is greater than a chosen lower bound. Results are ranked by the sum of the confidences of the machines that produced them. If the top-ranked result's confidence exceeds a given threshold, it is used. The confidence for servers in the weighted minority is reduced, and the monitoring subsystem is notified.

One possible policy uses the simple agreement regimes and queries one more server if no majority is obtained; a more conservative policy could move directly to full mode instead. In the probabilistic case, more servers can be queried if the top-ranked reply does not reach the desired confidence level. Suspicious application servers are taken offline for further diagnosis and repair (see below).

Note that the agreement protocol is just one mechanism of many to guard against corruption of content. To cause the system to provide invalid content, an attacker must corrupt multiple servers simultaneously so that they provide the same incorrect content, while evading the IDS and challenge-response protocol. Any content corruption detected by these stronger mechanisms is immediately repaired. The agreement protocol is intended to make delivery of corrupt content very unlikely, as long as assumptions about the maximal number of simultaneously compromised servers hold.

4.2 Risk vs. Performance

The choice of policy poses a tradeoff between what is considered the acceptable risk of bad content delivery versus the system performance requirements. The reliability of any particular regime depends on the quality of the IDS alerts and on the fault assumptions made.

Given assumptions about the number of failed servers, we can estimate the probability that the content provided to a client under a given policy is not correct. As an illustration, consider the simple agreement regime of Example 1. Let N be the number of application servers, F be the total number of compromised servers, and n be the starting simple regime (number of queried servers). The following focuses on integrity: to cause delivery of corrupt content, an attacker must modify content identically

F	initial simple regime n				
	1	2	3	4	5
0	0	0	0	0	0
1	0.2	0	0	0	0
2	0.4	0.3	0.3	0	0
3	0.6	0.7	0.7	1	1
4	0.8	1	1	1	1
5	1	1	1	1	1

Table 1. Probability of sending corrupted content, querying one more server in case of a tie

on all compromised servers. This is the worst-case assumption from the integrity point of view.[2]

Table 1 shows the probability of sending an incorrect response for $N = 5$, using a policy that starts with the simple regime n and queries one extra server in case of a tie, assuming that all F compromised servers agree in their incorrect response. The higher regimes reduce the probability of incorrect response and increase the probability of detecting compromised servers, provided only a minority of servers are faulty ($F \leq 2$). In duplex mode, service integrity is guaranteed if at most one server is compromised, but some requests may receive no response, unless the policy increases the regime. In triplex mode, continued and correct service is guaranteed if one server is compromised, but the system offers a reduced capacity to clients. The full regime provides correct service (with performance degradation) if several, but not a majority, of the servers are compromised (up to 2 for $N = 5$).

4.3 Responses to Alerts

Alerts are indications of possible attacks or compromises. Experience shows that all systems connected to the Internet are regularly attacked, or at least probed. IDS components generate a large number of valid alerts that do not always indicate system compromise, as well as numerous false alarms. Other parts of the monitoring subsystem generate alerts as well: the content agreement and challenge-response protocols provide alerts that identify likely compromises.

Although alerts can arise from direct detection of an attack, many report symptoms of a compromise that has already occurred. While attack detection usually indicates only the possibility of a compromise, symptom detection can reliably recognize a compromise after it has occurred. For

[2] If the attacker wishes to deny availability, content corruption can be arbitrary; no reply will be sent if the protocol fails.

example, unexpected server behavior such as an outbound connection unambiguously indicates some compromise.

The system can invoke a variety of actions in response to alerts, including:

- Temporarily blocking the address from which an attack appears to originate
- Increasing the agreement regime
- Increasing the coverage and frequency of the challenge-response protocol
- Disconnecting and rebooting a server
- Refusing service and alerting the system administrator

The choice of action depends on the current system state and the nature of the alert. Alerts can be evaluated according to the *severity* of the compromise they indicate, the *detail* they provide, and the *reliability* of the alert itself. We consider three classes of alerts: warnings of probes and clearly unsuccessful attacks (detailed, reliable, but not severe), credible indications that a specific asset is compromised (detailed, reliable, and severe), and evidence of significant attacks with unknown outcome (not detailed, somewhat reliable, and potentially severe).

The response to probes and unsuccessful attacks is to temporarily block the apparent source address. We are careful not to block for too long, since this can lead to denial of service if an attacker launches a nuisance probe with a spoofed source address.

If an application server is detected as compromised or otherwise failed, the system administrator is alerted, and the server is rebooted from a read-only copy of its operating system and data. The proxies detect that the server is back online by running a version of the challenge-response protocol. The proxy recovery protocol is similar, with the difference that the operational data for the proxy (e.g., current policy and regime, configuration) must be recovered from the other operational tolerance proxies, if present.

Adaptation is used to respond to the third class of alerts. The response to evidence of attacks with unknown outcome is to move to a more stringent agreement regime, which provides greater integrity assurance and helps identify compromised servers. The *alert manager* is a proxy module that accepts alerts from the monitoring subsystem. It assesses the state of each application server as up, down, or suspicious. This information is used to choose the appropriate agreement regime, as part of the implemented policy.

Finally, some alerts are both reliable and very severe, and indicate the compromise of a significant number of components. Such alerts arise, for example, if there is still insufficient agreement after all application servers are queried. In such cases, the only safe option is to shut down the system and alert the system administrator.

As an additional security mechanism, servers and proxies are regularly rebooted to prevent software aging and defeat incremental intrusions, as mentioned in Section 3.3. This does not fix any flaws present in the system, but gives an attacker only a limited window of time to exploit them.

4.4 Multiproxy Protocols

Our implementation to date has focused on detection and response protocols for multiple application servers mediated by a single leader proxy, and no auxiliary ones. However, we are developing the general case, where redundant proxies mitigate the weakness presented by the leader as a single point of failure (albeit a hardened and closely monitored one).

In a multiproxy configuration, auxiliary proxies monitor requests and replies from the application servers, and can thus quickly observe if the leader has failed. Like the leader, each auxiliary proxy includes an alert manager, which receives reports from the monitoring subsystem. A detected failure anywhere in the system is reported to all proxies, which must agree on the response. In particular, the proxies must maintain a consistent view of the current regime and of which proxy is the current leader.

Our multiproxy design includes three proxies and is intended to tolerate the compromise or failure of one of them. Proxies communicate with each other via a multicast channel implemented using a local ethernet link (Figure 1). This channel is assumed to ensure with very high probability that any message from one proxy is delivered to all other proxies, and that all messages are received in the same order by all proxies. In particular, this guarantees that *asymmetric* failures (a faulty proxy sending inconsistent messages to its peers) do not occur. Under this assumption, simple majority voting protocols are sufficient for maintaining consistency. (If needed, reliable atomic multicast protocols, such as [25], can be implemented to satisfy this assumption.)

When a proxy decides that a new agreement regime should be enforced, it multicasts a request to the other proxies, which reply by multicasting their agreement or disagreement. The regime is changed only if at least two of the three proxies agree. The disagreeing proxy, if any, is reported to the alert managers.

If an auxiliary proxy suspects that the leader is faulty, it multicasts a request for changing leader. If the other auxiliary proxy agrees, the leader is disconnected for repair and the administrator is alerted. The proxy that initiated the change of leader remains auxiliary and the other becomes the new leader. If the auxiliary proxies disagree on the leader status, the "accuser" is considered suspect by the other two.

A similar protocol is used if an auxiliary proxy is suspected of compromise. The decision to expel a proxy (leader or auxiliary) requires unanimity between the two others. If they do not agree, the accuser is suspect but not immediately shut down. After a delay, the accuser may persist and reinitiate the expel protocol. After a fixed number of "false accusations", the accuser is itself considered faulty and restarted. This reduces the risk of prematurely removing a noncompromised proxy that accused another by mistake.

5 Implementation

Our instantiation of the architecture provides intrusion-tolerant Web services. The first prototype has been completed, focusing on the content-agreement protocols, Web server deployment, and IDS. The Web servers used are Apache 1.3.20 running under Solaris 8 and FreeBSD 4.2, Microsoft IIS 5.0 running under MS Windows 2000, and Netscape FastTrack 4.1 (iPlanet) under RedHat 7.1.

Figure 3 shows the main components of our proxy implementation. The regime manager is responsible for executing the content agreement protocol (Figure 2). The alert manager takes input from the IDS subsystem and the challenge-reponse protocols, and notifies the regime manager when changes are warranted.

5.1 Monitoring Subsystem

The implemented monitoring subsystem includes a variety of intrusion detection sensors and alert correlation engines, as follows:

– Network-based sensors detect a variety of network attacks and probes in real time. These sensors run on a dedicated machine that monitors the traffic between the clients and the proxy, and the private subnet between the proxy and the application server bank.

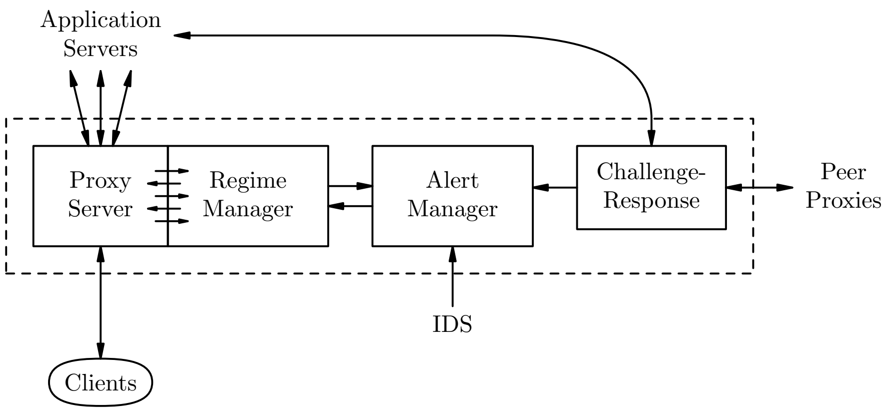

Fig. 3. High-level view of proxy implementation

eXpert-Net is a suite of network-based sensors, each of which focuses on a specific network protocol. It features an attack knowledge base and inference engine developed using a forward-chaining rule-based system generator, P-BEST [14]. eXpert-Net can detect complex attacks and variations of known attacks. For example, by performing session and transaction reconstruction, eXpert-HTTP, a sensor for monitoring Web traffic, detects attacks that would be missed if the analysis were performed on a per-packet basis.

Snort [26] is an open-source intrusion-detection sensor that has an extensive and updated knowledge base of attack signatures. Although both Snort and eXpert-Net are signature-based sensors, the diversity of their knowledge bases and implementation enables them to complement each other.

eBayes-TCP [29] uses a probabilistic model-based technique. This combines the generalization potential of statistical anomaly detection with the attack specificity of signature-based detection.

– Host-based operating-system-level sensors complement the network-based sensors by monitoring events at the operating system level. EMERALD eXpert-BSM [15] analyzes the audit records generated by the Sun Solaris Basic Security Module (BSM) to perform real-time signature-based detection. We are investigating deploying STAT host-based monitors [31] and Real Secure sensors [12] to monitor the other operating system platforms, including Windows. The generated alerts must be in a common format, such as IDMEF [5], accepted by the correlation engines.

– Host-based application-level sensors couple with an application to obtain high-level, semantically rich data for intrusion-detection analyses. We have developed an Apache module to collect HTTP transaction data, which is analyzed by the signature-based sensor eXpert-HTTP [1].
– A *blue sensor* [29] discovers hosts and services, and monitors their operational status, based on Bayesian inference. This sensor is coupled to the EMERALD eBayes-TCP session monitor. As a result, detection sensitivity is increased: failed accesses to invalid hosts and services are inherently more suspicious. Moreover, false alarms are reduced: accesses to unavailable services are expected to fail, so innocent traffic accessing these services will not trigger an alert.
– An EMERALD probabilistic alert correlator [30] performs data fusion from heterogeneous sensors.

5.2 Content Agreement using MD5 Checksums

A basic function performed by the proxy is checking that the pages returned by two or more application servers match. To improve the efficiency of this process, we use MD5 checksums [24], which the servers compute for each page served. These checksums are cryptographically strong: producing a fake page that matches a given MD5 checksum is an intractable problem given the current and foreseeable state of the art.

When comparing content from several servers, only the MD5 checksums need to be retrieved from all but one, which is queried for both checksum and content. The proxy verifies that the checksums match; if so, it also verifies that the received content matches the common MD5. This has the following advantages:

– Internal network bandwidth and proxy memory requirements are reduced
– When querying multiple servers, it is more efficient to compare the checksums than the full n pages; verifying a single MD5 is relatively inexpensive (linear-time in page size)
– The leader proxy can keep a cache of checksums, to be checked when lower agreement regimes are used. If cache hits occur, the proxy can operate at a higher assurance level despite using fewer application servers for content.

The servers have all been instrumented to include the MD5 checksum header, as specified by HTTP/1.1. The Apache Web server has experimental support for MD5 checksum headers, while the feature was manually added to both iPlanet and IIS.

The MD5s could be replaced by a more general cryptographic signature, where each application server signs its pages using a unique private key, and the proxy verifies each of the signatures. Such a signature can certify that the origin of each message is the corresponding application server, preventing spoofing on the private network (which can also be prevented using dedicated hardware). It does not, however, guarantee the integrity of the content itself. For instance, if source data files are compromised by an undetected attack, the application server will correctly sign the wrong data.

Similarly, public-key infrastructure can be used so that the end-users (clients) can certify that the origin of a served page is, indeed, the proxy server. Again, this mechanism by itself cannot guarantee the integrity of the content, but only the authenticity of the source.

5.3 Policy Implementation

Our implementation supports a variety of policies based on a generalization of the simple agreement regimes of Example 1. In general, each regime is specified by a pair (n, k), where n is the number of servers to query, and k is the minimum number of servers that yield sufficient agreement in that regime. Let N be the number of application servers, and $\mathcal{P}$ the set of pairs (i, j), where $N \geq i \geq j \geq 1$.

The policy is specified by two functions, $\mathit{start} : \{0, 1, 2, \ldots\} \rightarrow \mathcal{P}$ and $\delta : \mathcal{P} \rightarrow \mathcal{P} \cup \{\texttt{panic}\}$. If the alert manager assesses that F servers are suspicious, then $\mathit{start}(F) = (n, k)$ specifies the initial regime. The client request is forwarded to n servers, and a response is considered correct if there is agreement between k of them; otherwise, the function δ is used to identify a new pair (n', k'), which dictates the new regime, querying $n' - n$ extra servers. This is repeated until satisfactory agreement is obtained, or the `panic` state is reached, in which case the system is considered too corrupted to function. The alert manager is notified of the content agreement results.

Implemented policies can range from efficiency-conscious ones that initially ask only a few servers and query one additional server when needed, to integrity-conscious ones that query more servers initially and immediately query all servers when in doubt.

6 Discussion and Conclusions

Our intrusion-tolerant architecture combines a variety of traditional security mechanisms, as well as concepts from fault tolerance and formal

Security mechanism	*Intended function*
IDS	– Detects attacks by monitoring network – Detects unexpected traffic on internal network – Triggers adaptation mechanisms – Alerts system operator of serious conditions
Content agreement	– Corroborates content served to client – Detects erroneous application server (AS) behavior
Adaptive agreement policies	– Given evidence of suspicious events, reduces likelihood of serving incorrect content – Avoids querying suspicious and compromised AS
Challenge-response	– Verifies integrity of application servers and proxies – Triggers adaptation mechanisms – Provides heartbeat for liveness check
Proxy hardening	– Limits proxy vulnerabilities – Restricts communication between AS and outside world – Restricts many malformed requests forwarded to AS
Online verification	– Ensures that proxy software behaves according to specification
Regular reboot of proxies and AS	– Prevents unreliability due to software "aging" – Defeats long-term, incremental intrusions
Proxy peer monitoring	– Verifies that proxies are operating properly

Table 2. A summary of security mechanisms in the proposed architecture

verification. Table 2 summarizes these mechanisms, and the protective functions they play. The system is adaptive, responding to alerts and intrusions, trading off performance against confidence in the integrity of the results. Our architecture allows for a wide variety of response policies to be implemented, depending on the environmental assumptions and cost-benefit analysis made.

6.1 An Example—Code Red Scenario

The utility of the various mechanisms and levels of detection and recovery can be illustrated using the well-known Code Red worm [17] as an example.

Both the EMERALD eXpert-Net and Snort IDSs presently installed detect the Code Red attack. The proxies are notified of the detection, and the source address of the attack is temporarily blocked. Consider now an attack that, like Code Red, depends on a buffer overflow included as part of the request, but is sufficiently different to evade the diverse IDS.

Because of its length and unusual syntax, the attack request is likely to be blocked by the proxy. Note that a hardened proxy, compiled with tools such as StackGuard [2], should not be vulnerable to general buffer overflow attacks.

Suppose nonetheless that the request is forwarded to the application servers. Code Red will only affect the IIS servers, which will be a minority in a diverse system implementation. These servers will be diagnosed as failed and restarted when erroneous content is fetched from them, or as a result of running the challenge-response protocol. Meanwhile, the other servers remain available to provide valid content.

Worms such as Code Red propagate by hijacking the Web server and initiating outbound connections from the server. Such connections are detected by the IDS on the private net and blocked at the proxy, and the corresponding server is diagnosed as failed.

In summary, the attack would have to bypass a number of mechanisms to be successful, including direct detection on the external network, symptom detection on the private network, failed agreement, and challenge-response protocols. The means used by the attack to infect the system, corrupt content, and propagate are effectively thwarted.

More generally, we believe that a multiplatform attack that can corrupt content on a majority of sufficiently diverse application servers using a single query or action is very unlikely. An attack that exploits simultaneously present but different vulnerabilities requires more malicious traffic, increasing the likelihood of detection by at least one component.

6.2 Dangers and Tradeoffs

A carelessly implemented and overly reactive response policy can cause the system to operate continuously at degraded capacity. Indeed, with sufficient knowledge of the detection and response mechanisms, an attacker may launch attacks to trigger responses that result in self-denial of service.

As a simple example, it is possible to craft an HTTP request that will make different server brands respond differently, as discussed in [1]. Replacing a regular space (ASCII 20h) with the `tab` character (09h) in a `GET` request yields a valid reply from the Apache and the iPlanet servers, but an error code from the IIS Web server. In this case, there is nothing wrong with the IIS server and it should not be rebooted. Rather, the client behaves suspiciously and should be blocked.

As mentioned in Section 2.2, the proxy forwards a sanitized version of the original request. This includes changing all whitespaces to the regular space, so we avoid such attacks.

6.3 Related Work

A number of commercial products, such as Tripwire™ and Entercept™, concentrate on protecting stand-alone Web servers, with no redundancy management. Our architecture can use these techniques as additional security measures.

SITAR [32] is an intrusion-tolerant architecture for distributed services, and shares many of our goals and assumptions. It includes adaptive reconfiguration, heartbeat monitors, runtime checks, and COTS servers mediated by proxies. While we focus on the relationship between IDSs, content agreement, and adaptive policies, SITAR focuses on using fault-tolerance techniques, including Byzantine agreement protocols, to coordinate the proxies.

The ITUA project [3] relies on intrusion detection and randomized automated response. It provides middleware based on process replication and unpredictable adaptation to tolerate the faults that result from staged attacks. HACQIT [13] uses a primary/backup server architecture that does not perform content agreement, and focuses on learning by generalizing new attacks.

Adaptive Fault Tolerance: One main characteristic of our architecture is its ability to automatically adapt the server redundancy to the level of alert. Most fault-tolerant systems adapt their redundancy to component failures, rather than the level of perceived threat. When a persistent fault is diagnosed, the faulty unit is isolated, repaired and then reinserted. In most cases, this delay is made as short as possible to maintain a similar level of redundancy for the whole mission.

Architectures such as SIFT [33] and GUARDS [21], assign different levels of redundancy to concurrent tasks according to their criticality. In other fault-tolerant architectures such as Delta-4 [22] and FRIENDS [8], different redundancy levels and techniques are assigned according to fault assumptions rather than criticality. In these architectures, as well as in AQuA [4], the redundancy levels assigned to tasks can be modified by operator commands to adapt to environment changes.

In SATURNE [7], idle computers are activated to increase the redundancy of active tasks, proportionally to task criticality. Similarly, the

AFT architecture [9] dynamically adapts the task redundancy according to task criticality and real-time constraints.

6.4 Future Work

We are working towards a rational tradeoff analysis, which we see as essential to the development of automated response mechanisms and their eventual adoption. Choosing a response policy poses difficult tradeoffs between integrity, availability, latency, and assumptions about potential threats. We will study the application of Markov decision process theory to this problem. Our goal is an analysis framework that will let us evaluate different policies given a variety of assumptions about faults, attacks, and accuracy of the IDS and monitoring subsystem.

We also want to provide more guarantees concerning the ability of the system to avoid self-denial of service as a result of a low-level attack, as well as the ability to tolerate and recover from adverse conditions in a timely fashion.

Our architecture should be modified if a large number of application servers are needed. Diversity of COTS implementation is not practical in this case. Instead, the architecture should be based on clusters, each with a diverse COTS base and proxy bank, replicating the single-cluster model that we have described. A load managing component based on our hardened proxy can forward each query to one or more clusters, depending on the current load.

Acknowledgments: We thank Dan Andersson and Phil Porras for their feedback and comments, and Ayda Saidane and Vincent Nicomette for their contribution to the challenge-response protocol.

References

1. M. Almgren and U. Lindqvist. Application-integrated data collection for security monitoring. In *Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 22–36. Springer-Verlag, Oct. 2001.
2. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan. 1998.
3. M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett. Intrusion tolerance approaches in ITUA. In *Fast Abstract Supplement of the 2001 Intl. Conf. on Dependable Systems and Networks*, pages B–64, B–65, July 2001.

4. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *17th IEEE Symposium on Reliable Distributed Systems (SDRS-17)*, pages 245–253. IEEE Computer Society Press, Oct. 1998.
5. D. Curry and H. Debar. Intrusion detection message exchange format: Data model and extensible markup language (XML) document type definition, Nov. 2001. Work in progress.
6. Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion tolerance in distributed computing systems. In *Proc. Intl. Symposium on Security and Privacy*, pages 110–121. IEEE press, May 1991.
7. J.-C. Fabre, Y. Deswarte, J.-C. Laprie, and D. Powell. Saturation: Reduced idleness for improved fault-tolerance. In *18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 200–205. IEEE Computer Society Press, 1988.
8. J.-C. Fabre and T. Prennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47:78–95, Jan. 1998.
9. O. Gonzalez, H. Shrikumar, J. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *18th IEEE Real-Time Systems Symposium (RTSS '97)*. IEEE Computer Society Press, Dec. 1997.
10. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Engelwood Cliffs, NJ, 1991.
11. Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *25th Symposium on Fault Tolerant Computing*, pages 381–390. IEEE Computer Society Press, June 1995.
12. Real Secure server sensor policy guide version 6.0, May 2001. `http://www.iss.net`.
13. J. E. Just and J. C. Reynolds. HACQIT (Hierarchical Adaptive Control of QoS for Intrusion Tolerance). In *17th Annual Computer Security Applications Conference*, 2001.
14. U. Lindqvist and P. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161. IEEE press, May 1999.
15. U. Lindqvist and P. Porras. eXpert-BSM: A host-based intrusion detection solution for Sun Solaris. In *Proc. of the* 17^{th} *Annual Computer Security Applications Conference*, Dec. 2001.
16. P. Liu and S. Jajodia. Multi-phase damage confinement in database systems for intrusion tolerance. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 191–205, June 2001.
17. R. Permeh and M. Maiffret. `.ida` "Code Red" worm. Security Advisory AL20010717, eEye Digital Security, July 2001. `http://www.eeye.com/html/Research/Advisories/AL20010717.html`.
18. P. Porras. Mission-based correlation. Personal communication, SRI International, 2001. `http://www.sdl.sri.com/projects/M-correlation`.
19. P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Security Conference*, Oct. 1997.
20. P. Porras and A. Valdes. Live traffic analysis of TCP/IP gateways. In *Proc. Symposium on Network and Distributed System Security*. Internet Society, Mar. 1998.

21. D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabjac, and A. Wellings. GUARDS: A generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems*, 10:580–599, June 1999.
22. D. Powell, G. Bonn, D. Seaton, P. Verssimo, and F. Waeselynck. The Delta-4 approach to dependability in open distributed computing systems. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18)*, pages 246–251. IEEE Computer Society Press, June 1988.
23. G. R. Ranger, P. K. Khosla, M. Bakkaloglu, M. W. Bigrigg, G. R. Goodson, S. Oguz, V. Pandurangan, C. A. N. Soules, J. D. Strunk, and J. J. Wylie. Survivable storage systems. In *DARPA Information Survivability Conference and Exposition II*, pages 184–195. IEEE Computer Society, June 2001.
24. R. Rivest. The MD5 message digest algorithm. Internet Engineering Task Force, RFC 1321, Apr. 1992.
25. L. Rodrigues and P. Verissimo. xAMp: a multi-primitive group communications service. In *11th Symposium on Reliable Distributed Systems*, pages 112–121, Oct. 1992.
26. M. Roesch. Snort: Lightweight intrusion detection for networks. In *USENIX LISA'99*, Nov. 1999. www.snort.org.
27. F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.
28. Tripwire white papers, 2001. http://www.tripwire.com.
29. A. Valdes and K. Skinner. Adaptive, model-based monitoring for cyber attack detection. In *Recent Advances in Intrusion Detection (RAID 2000)*, pages 80–92, Oct. 2000.
30. A. Valdes and K. Skinner. Probabilistic alert correlation. In *Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 54–68. Springer-Verlag, Oct. 2001.
31. G. Vigna, S. Eckmann, and R. Kemmerer. The STAT tool suite. In *DISCEX 2000*. IEEE press, Jan. 2000.
32. F. Wang, F. Gong, C. Sargor, K. Goseva-Popstojanova, K. Trivedi, and F. Jou. SITAR: a scalable intrusion tolerance architecture for distributed server. In *Second IEEE SMC Information Assurance Workshop*, 2001.
33. J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostack, and C. Weinstock. SIFT: the design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE*, 66:1240–1255, Oct. 1978.