Architecture and Concepts of the ARGuE Guard

Jeremy Epstein
jepstein@nai.com
NAI Labs
McLean Virginia

Abstract

ARGuE (Advanced Research Guard for Experimentation) is a prototype guard being developed as a basis for experimentation. ARGuE is based on Network Associates' Gauntlet firewall. By integrating capabilities developed under several government programs, we were able to create a system which is easier to extend than other guards, provides significant new features (such as integration with an intrusion detection system), and yet has a reasonable degree of assurance.

1. Introduction

Standalone system high networks are a thing of the past. Historically, networks of different classifications were usually not connected. When connection was imperative, a person in the middle was required to review all data flowing between the networks (a slow and error-prone process). Today, even users of highly classified networks need instantaneous access to resources on the Internet. For example, planning a military operation requires access to public "open source" news and weather information (e.g., CNN). Additionally, organizations need to allow limited subsets of users on the outside to access resources inside the classified networks, especially in coalition environments.

Connecting such networks together historically required a guard, a special purpose device designed to prevent information flowing from the inside (the more highly classified side) to the outside (the less highly classified side). Guards differ from firewalls in their primary intent: a firewall is mostly concerned with keeping unauthorized users out, while a guard has the additional goal of preventing information on the inside from being sent to the outside.

Existing guards suffer from several key problems:

- They were either built on special purpose operating systems to maximize their resistance to attack (which made them both expensive to obtain and manage), or they were built on weak COTS operating systems (which made them vulnerable to attack). Examples of the former class include the C2 Guard [Fiorino], which is built on the XTS-300 B3-rated operating system [XTS-300]. Examples of the latter class include the ISSE guard [ISSE], which is built on an ordinary Solaris operating system.[1]
- The guards were also built for particular applications, and were generally hard to extend to other uses. For example, a DISA-sponsored study [SGS] found approximately 50 different guards built by the US Department of Defense. None of these guards have the capability to deal with modern middleware protocols such as IIOP (used by CORBA).
- In many cases, guards require a human to "certify" each piece of data (e.g., E-mail message) to be released from the inside to the outside, which is difficult to do accurately. In general, the certification occurs inside the enclave, using trusted software which puts a digital signature on the data to be released. The signature is then verified by the guard before release. This technique relies on the correct operation of the user's approval software (i.e., the correct functioning of the user's workstation). For example, Secure Computing's Standard Mail Guard [Smith] requires that the user invoke a Fortezza card to perform signing of each message to be released, without any assurance that the Fortezza card is signing what the user intended. The SMG can verify that the signature was applied correctly, but cannot determine whether the signed data is in fact appropriate for release, or even if it is what the user intended to release. Even aside from assurance issues, this scheme is inappropriate for connections involving lower level protocols (e.g., IIOP), since users cannot realistically approve each object invocation.
- As special purpose devices, guards lack integration with other security devices, such as working with intrusion detection systems. They require a separate set of management capabilities, and cannot be managed along with the rest of the network.

Our goal in designing ARGuE was to use a modern firewall as a base, thus providing a strong platform that has already withstood concerted attack. We then

[1] Earlier versions of the ISSE were built on the Harris Nighthawk, a B1 UNIX system.

extended the firewall in ways to provide modular functionality that provides a reasonable degree of assurance. Our goal was not to build an accredited (or even accreditable) guard; as such we have not developed any of the formal documentation required to field a product. Rather, our goal was to explore how we could use existing commercial and research technologies to provide a prototype of a next generation guard.

The remainder of this paper is organized as follows: Section 2 describes the ARGuE capabilities and architecture. Section 3 describes the current status of our work, limitations, and our future research directions. Section 4 concludes the paper.

2. ARGuE Capabilities

This section describes the Gauntlet capabilities, how we extended it to create the prototype guard (ARGuE), and the ARGuE architecture.

As described above, our goal was to build a guard with reasonable assurance, yet based on COTS products for low cost, flexibility, etc. Our method of achieving assurance was to build on a strong firewall platform, adding multi-part proxies (described below) that reduce the risks inherent in all firewall proxies, and strengthen the foundation by using operating system level wrappers to constrain the behavior of the proxy. Collectively, we believe this layered defense approach results in a guard that provides both functionality and assurance.

2.1 Gauntlet Capabilities

By building ARGuE (Advanced Research Guard for Experimentation) on the Gauntlet firewall, we gained several key capabilities.

First, Gauntlet has a strong pedigree, having been installed in thousands of sites. Although it has never been evaluated for use in Multi-Level Secure (MLS) environments, its ability to withstand attack is understood.

Second, as a COTS software product Gauntlet is a low-cost solution, running on COTS hardware architectures such as Intel PCs, Sun SPARC, and Hewlett-Packard PA-RISC.

Third, existing Gauntlet facilities provide many of the necessary features for a guard, including Virtual Private Networks, and the ability to block based on IP addresses.

Fourth, Gauntlet is readily extensible by including a Proxy Development Toolkit (PDK). The availability of existing protocol proxies allows incremental development of the guard. Initially, the guard can use existing proxies (with the limited filtering capabilities they provide), replacing them as more sophisticated filtering proxies become available.

Fifth, Gauntlet includes sophisticated management capabilities, including integration with Network Associates' Cybercop intrusion detection products.

2.2 ARGuE Extensions to Gauntlet

ARGuE extends the Gauntlet product in several ways: by adding "safer" multi-part proxies for critical protocols;

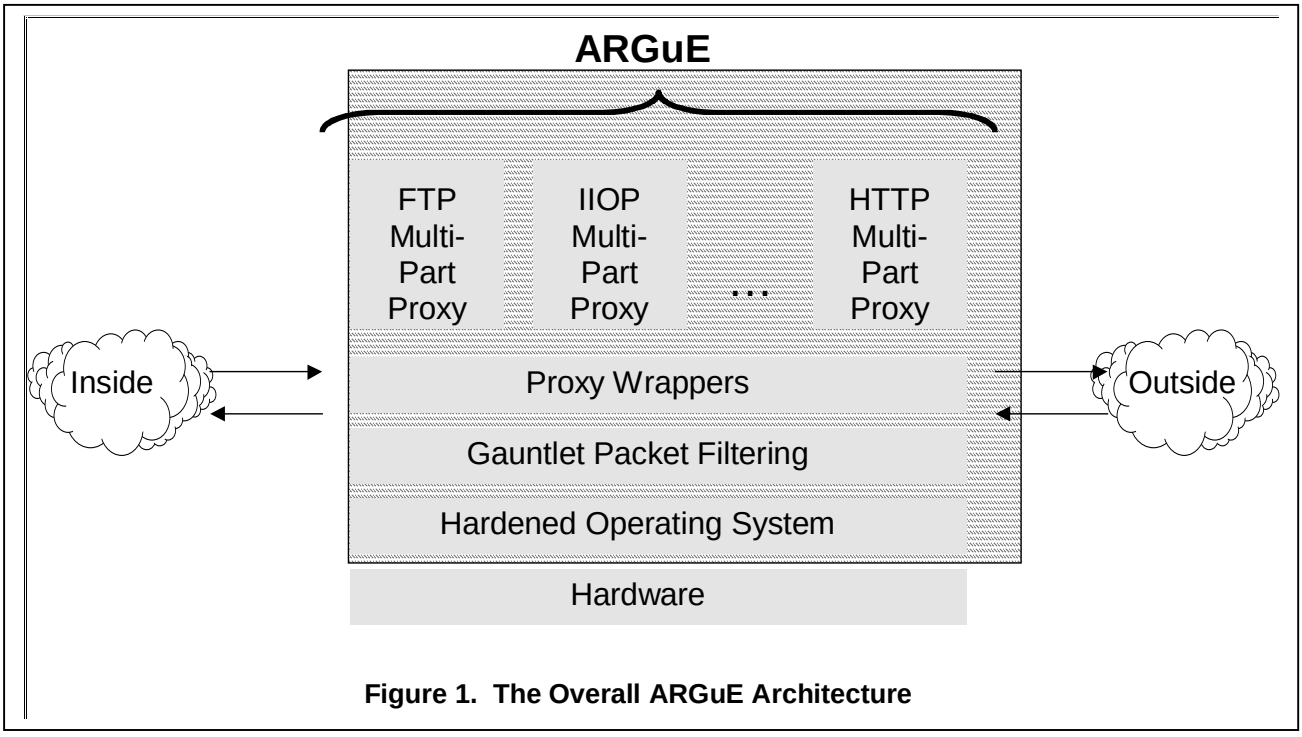

Figure 1. The Overall ARGuE Architecture

by providing "data sealing" capabilities; by integrating wrappers technology for constraining incorrect proxy behavior; and by providing data and application-specific intrusion detection information.

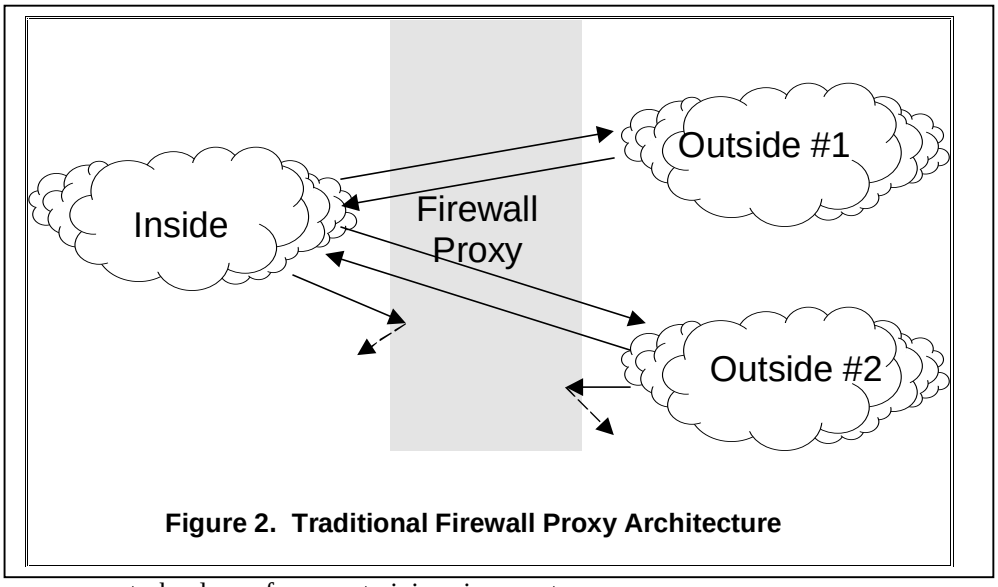

Figure 2. Traditional Firewall Proxy Architecture

Figure 1 shows that an ARGuE system is made up of a Gauntlet with one or more multi-part proxies, where each proxy is constrained using wrapper technologies.

2.2.1 The Multi-Part Proxy Architecture

Proxies in most firewalls (including Gauntlet) are trusted software that communicate between the "inside" and "outside" networks. Figure 2 shows a traditional proxy, which provides communications between one inside and two outside networks. In general, such proxies allow unlimited communication from inside to outside, and limited communication from outside to inside. Any flaw in the proxy (including subversion), can cause the proxy to provide direct communication from the outside to the inside.

By contrast, the ARGuE multi-part proxy, shown in Figure 3, divides the work of the proxy into several programs: one that communicates with (each) outside network, one that communicates with (each) inside network, and one in the middle that provides for filtering between each combination of inside and outside networks.

The inside listener/sender performs two functions: it listens for protocol operations (e.g., IIOP requests or replies) coming from the inside network and translates (externalizes) them into files, and it listens for the results of the content-based filter and translates the externalized files into protocol operations (e.g., IIOP requests or replies). Each outside listener/sender performs the analogous function for its attached outside network. The content-based filters review the file created by the inside or outside listener/sender, performing any necessary content-based decision-making depending on the direction of the transfer, and forwards the request to the opposite side. The content-based filter can also modify the file, thus performing sanitation (e.g., excising dirty words or "fuzzing" data values).

The use of files as the transfer mechanism is to reduce the binding between the different parts of the system. However, there is no fundamental architectural reason why the connection needs to be using a file. For example, a different implementation might use UNIX shared memory segments as the communication method, using appropriate permissions on the memory to control which

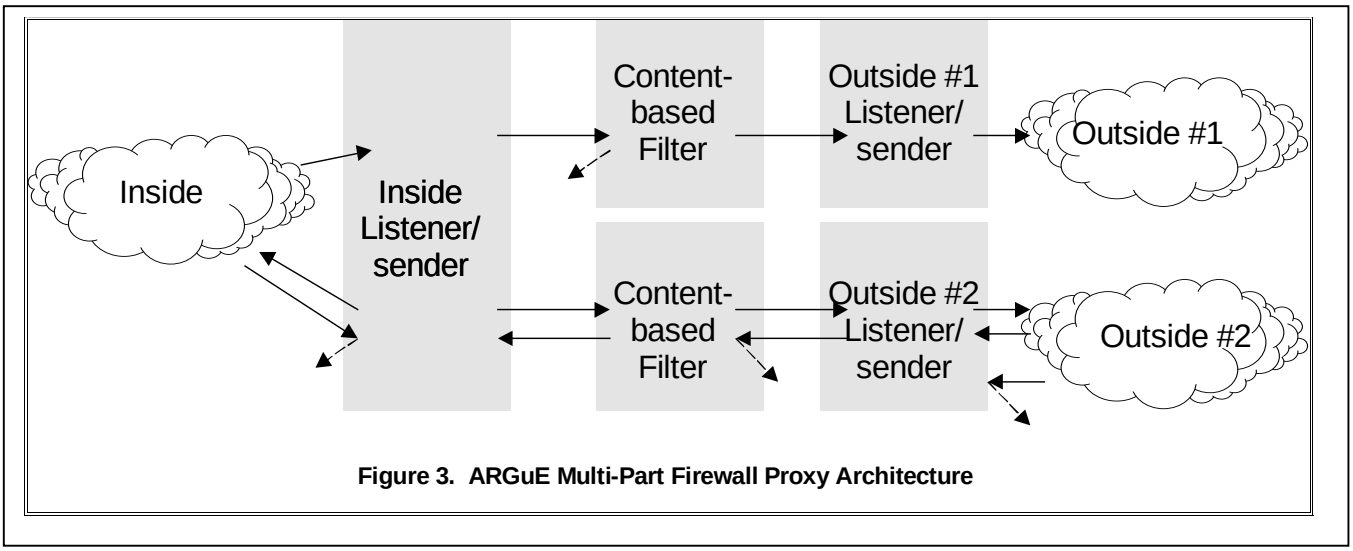

Figure 3. ARGuE Multi-Part Firewall Proxy Architecture

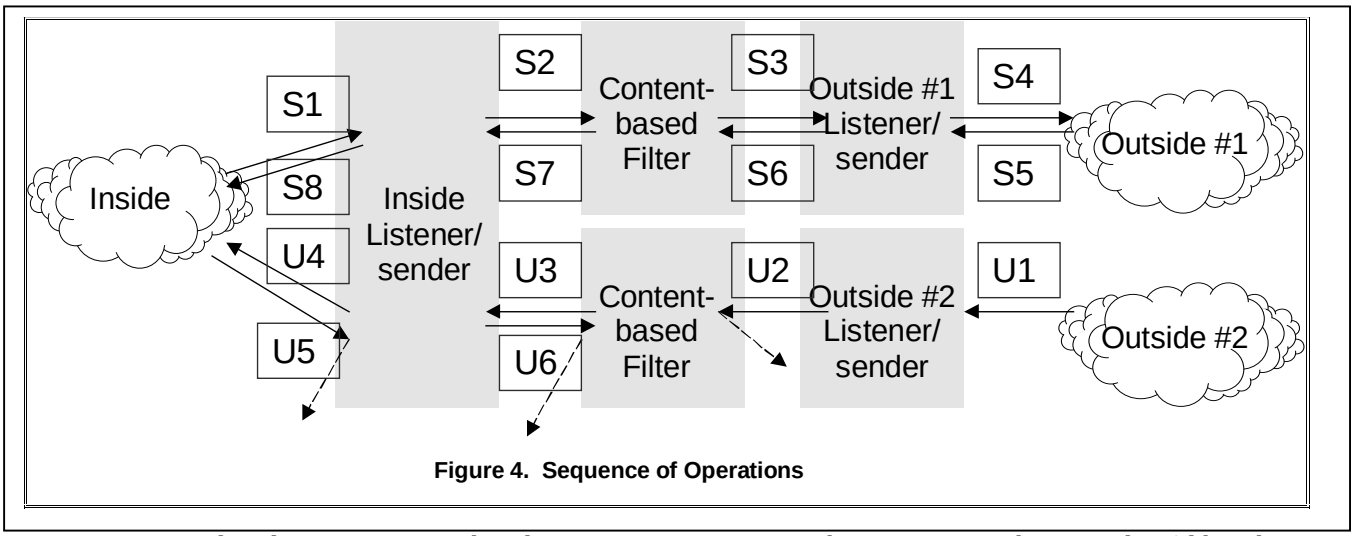

Figure 4. Sequence of Operations

processes can read and write. Using shared memory would probably be faster than the existing file-based mechanism.

Figure 4 shows the sequences for a typical successful or unsuccessful protocol operation.

Consider the sequence of operations shown in the upper portion of Figure 4. The operation starts on the inside, where the client application sends a request (S1) to the inside listener/sender. Access controls may be performed at this step before the request is externalized into a file and transferred (S2) to the content-based filter. The filter makes a decision based on the contents of the request, and forwards the file (S3) to the outside listener/sender. The file is converted from the file format back into the original protocol format, and sent to the server on the outside network. When the server responds (S5), the outside listener/sender may perform access controls, and then converts the response to a file, continuing back through the content-based filter (S6) to the inside listener/sender (S7), where it is converted back into a protocol stream and sent to the originating client.

The lower portion of Figure 4 shows an unsuccessful operation, beginning on the outside (although an unsuccessful operation could begin either on the inside or outside). In this case, the request is sent to the outside listener/sender (U1), which may perform access controls before converting the request to a file (U2) and sending it to the content-based filter. After reviewing the contents of the request, the filter may reject the request (the dotted line), or it may forward the request on to the inside listener/sender (U3), which converts the file back to a protocol request, and sends it to the inside server (U4). The server's response (U5) is sent to the inside listener/sender which may perform access control and reject the response (shown as the dotted line), or accept it and forward it (U6) to the content-based filter, which may also reject the request (the dotted line).

One complicating factor in the unsuccessful case is that if the response is rejected (at any of the locations shown in Figure 4), the client is left waiting for a response. Depending on the application architecture, it may be necessary to generate a synthetic response to the client indicating that the request or response has failed. In some cases it may be sufficient to reject it without

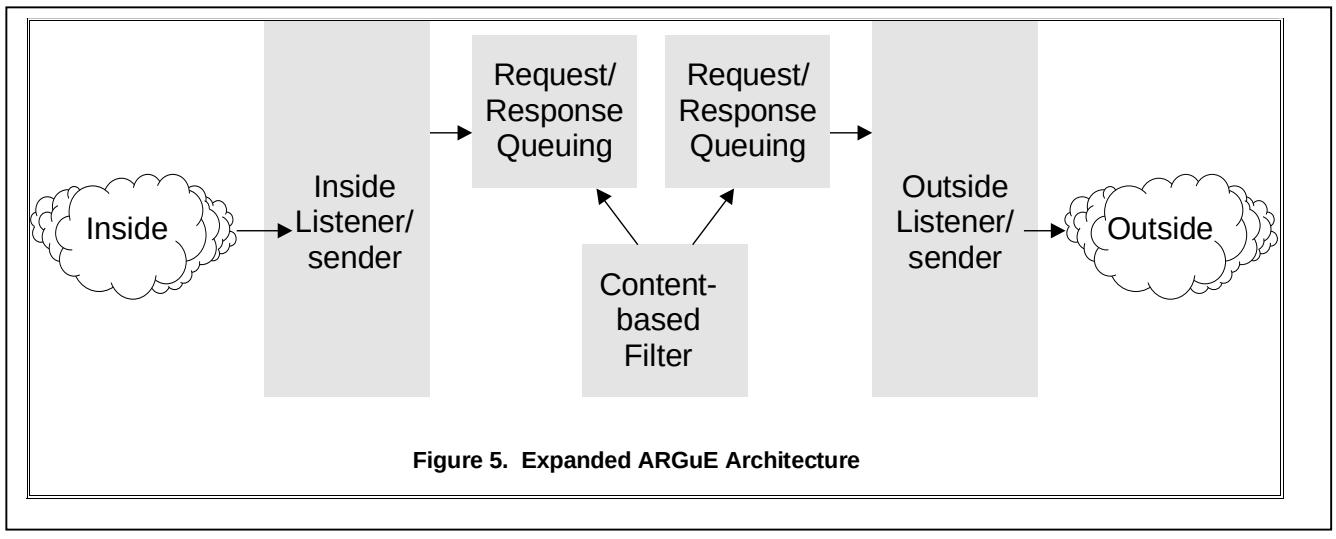

Figure 5. Expanded ARGuE Architecture

indicating whether it was the request or the response that was unsuccessful; in other cases it may be necessary to indicate what the cause of the failure was. In any case, some information leakage will occur as a result of the synthetic response, so careful definition of the generic response is necessary.

Figure 5 shows an expanded version of the architecture, adding the queuing and dequeuing components which are not shown in Figure 3. Each queuing component has an input and an output directory for each transfer direction. Only filters copy files from the input directories to the output directories.

The concept of a multi-part proxy was inspired by the C2Guard [Fiorino], which uses a similar sequence (shown in Figure 6) to split up the operation. The C2Guard consists of three computers: a Sun Solaris system that queues files from the inside and passes them over a serial line to the Wang XTS-300; the XTS-300 running the content-based filters; and a second Sun Solaris system that accepts the files over a serial line from the XTS-300 and transfers them to the outside. (The process is equivalent for files being transferred from the outside to the inside.) The queuing and dequeuing computers are required to be dedicated to that purpose; they accept (and send) files using NFS and FTP. In environments where protocols such as IIOP are required, another pair of computers (shown as the protocol/file translators) are required to translate from the native protocol to file format and back. The key difference is that ARGuE requires a single computer to provide all of the capabilities (protocol handling, queuing, and filtering) as opposed to three or five computers as in the case of the C2Guard.

2.2.2 Writing Content-based Filters

Each multi-part ARGuE proxy consists of the five parts enumerated above:

1. the inside listener/sender,
2. the inside request/response queuing,
3. the content-based filter,
4. outside request/response queuing, and
5. outside listener/sender.

Of these five parts, the first two and last two are generic for a given protocol. That is, all installations of ARGuE which use IIOP as the transfer mechanism will use identical software. The content-based filter, however, is highly application specific.

There are two parts to content-based filtering: figuring out what is to be filtered (i.e., the organizational security policy), and translating those rules into filtering code. Of these, the first is clearly more difficult, because organizations frequently do not know what makes data sensitive, especially at the protocol level where the higher level semantics are stripped away. It is further complicated with classified data (i.e., when connecting classified to unclassified networks) in that few individuals are willing to take the risk of identifying information as unclassified. It is far simpler to claim that all data is classified, thus leading to the traditional unconnected system-high systems. Determining what is to be filtered is a primarily social exercise, not a technical one, and as such is not further explored in this paper.

The ARGuE design allows for use of any executable program as a filter. Thus, the filter could be written in a low level language like C, in a scripting language such as Perl, or in an interpreted language such as Java. Since the filtering is done directly on the boundary controller

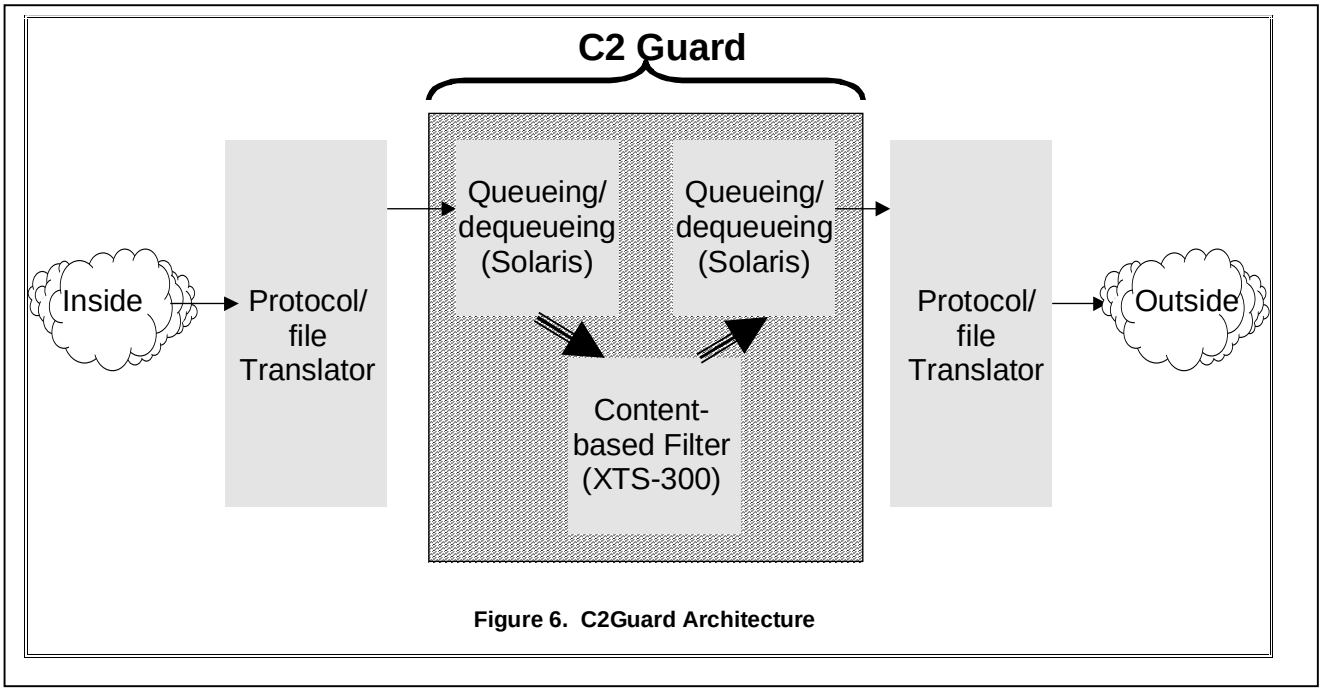

Figure 6. C2Guard Architecture

machine, any vulnerabilities in the filter may make it vulnerable to outside attacks. While we could use our Wrappers technology to limit the capabilities of the Perl or Java interpreters, there seemed to be little to be gained by that choice, and much to lose if there are other types of vulnerabilities in Java that we are not currently aware of. For these reasons, we rejected use of Perl or Java, primarily because of the complexity they introduce.

Instead, we decided to use Felt [Guttman], a language developed specifically for filter development. Felt provides constructs for parsing input files into fields and filtering based on content values. It can also be extended by writing C code, which can be embedded in the filter definition.

While it is non-procedural (with the exception of C extensions), Felt is still a programming language, and as such, requires significant expertise to write filters. Since filters are application specific, the effort involved in development is a significant cost in fielding a guard, as opposed to the generic parts of the system which can be amortized over multiple instances. As an area for further research, we are investigating whether a graphical user interface (GUI) could be used to present templates to a non-programmer who is knowledgeable about the security constraints, and have that person fill in the content limitations. The result of the GUI would be a Felt program, which could then be reviewed along with the GUI inputs to verify that it meets the organizational security requirements.

2.2.3 Data Sealing in ARGuE

In some cases, data comes into a system from the outside, is stored, and is then exported back either to its origin or to some other outside organization. In these cases, filtering can be avoided if the data can be recognized as having previously been imported, using the theory that if something came in then it must be acceptable to send out again.

In ARGuE, we implemented this concept by allowing filters to place digital signatures on data items as they transit from the outside to the inside. An inside-to-outside filter can then verify the digital signature as part of the filtering process, most likely in preference to performing content-based filtering.

The most difficult part of data sealing is determining what to do with the seal data (i.e., the data that makes up the seal itself). Our goal was to add seals to CORBA (IIOP) traffic, preferably without changing either the client or server application. We considered three approaches:

1. Figure 7 shows the initial approach where the seal would be embedded in the Interoperable Object Reference (IOR). This method was not feasible because it interfered with the operation of the CORBA application.
2. Figure 8 shows the second approach where the seal is calculated and embedded in a field designated as part of the application definition. This method worked acceptably, although it required that both the client and server applications be aware of the seal to reserve space for the seal storage. Additionally, it required that the server application store the seal with the data and retrieve it whenever retrieval is required. Thus, it was operable, but it did not meet our goals.
3. Figure 9 shows the third approach where the seal is calculated in the guard and stored there. Neither the client nor the server need be aware of the seal

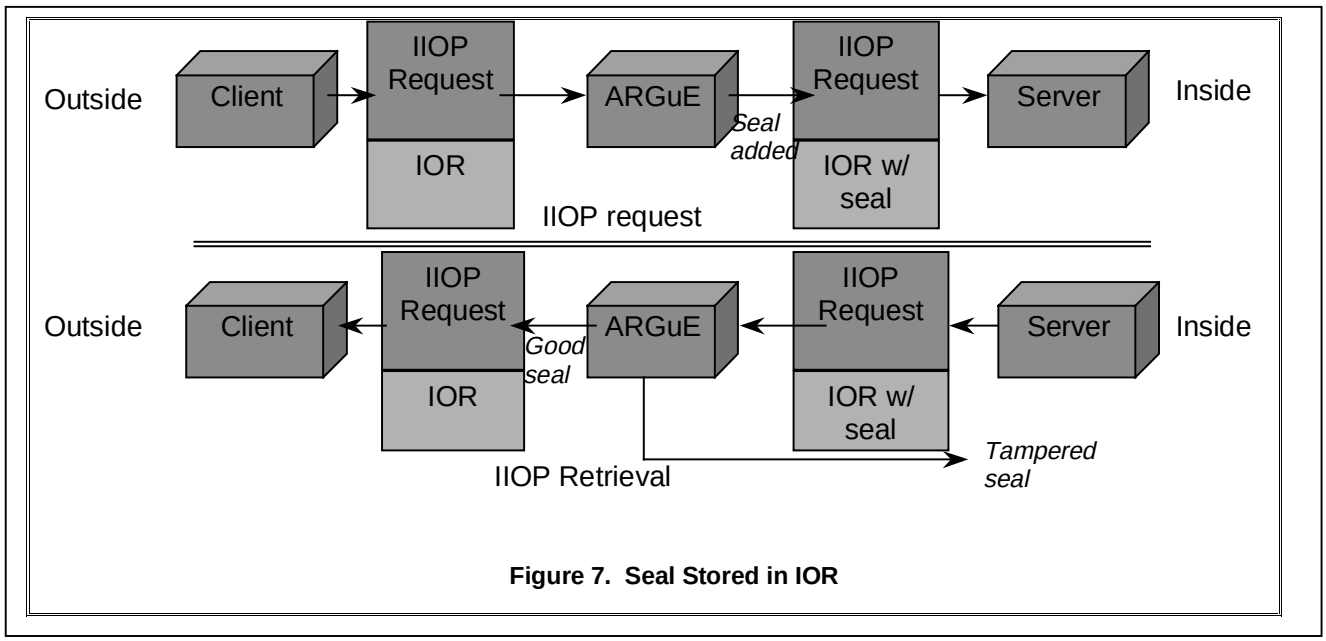

Figure 7. Seal Stored in IOR

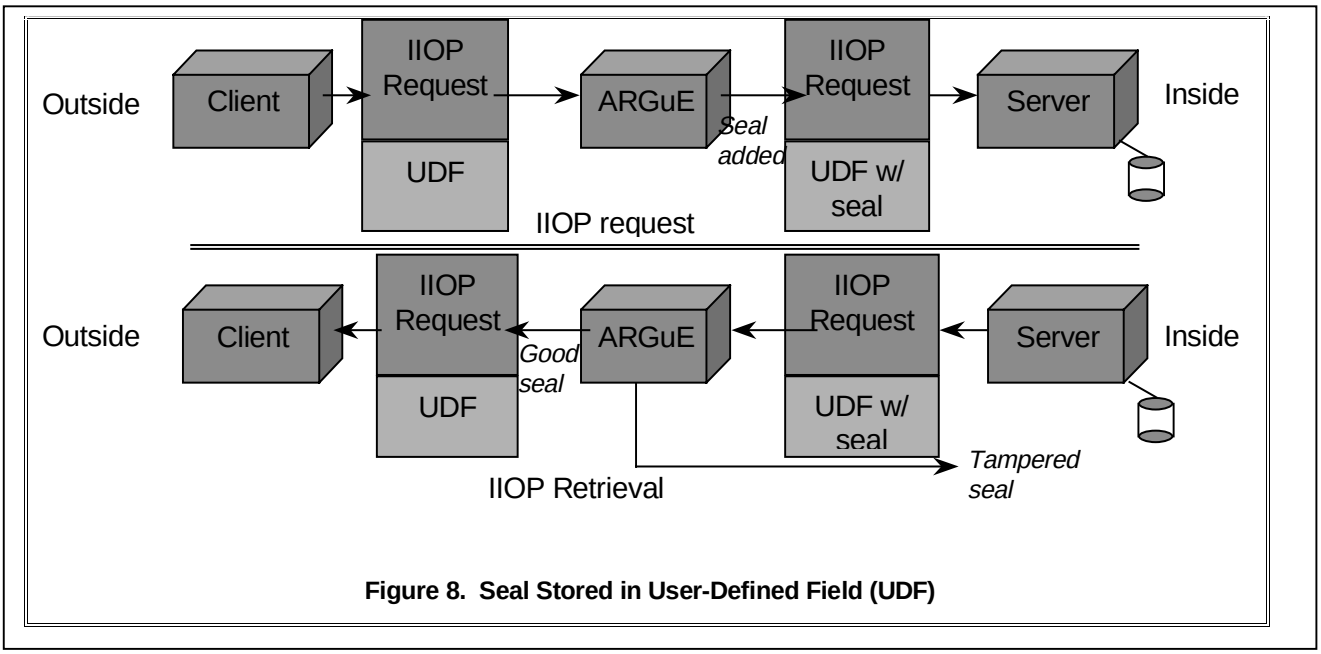

Figure 8. Seal Stored in User-Defined Field (UDF)

calculation, as it is kept exclusively within the guard. However, there are several problems with this approach. The guard must have a mechanism to determine whether a particular piece of data being presented for release has a corresponding seal in the database. The simplest method is for the guard to recalculate the seal, and then search its database for a previous instance of the seal value. Presuming that the digital signature algorithm is sufficiently strong that collisions are acceptably unlikely, this method will allow release of previously signed data, but will not allow release of unsigned data. A concern is that the guard cannot know when it is safe to discard any of the seals it has kept in its database, since it does not receive any notifications that the server has discarded data (the server being unaware of the presence of the database in the guard).

While any of these methods can be implemented, ARGuE currently uses the second and third of these methods (with the second being preferred).

One of the more difficult aspects of data sealing is determining what can realistically be sealed, and still have the seal be meaningful. For instance, if a seal were applied to a one byte value, then there would be too few realistic seals to be meaningful. There is no "magic" correct size for sealed data, but it is a consideration in

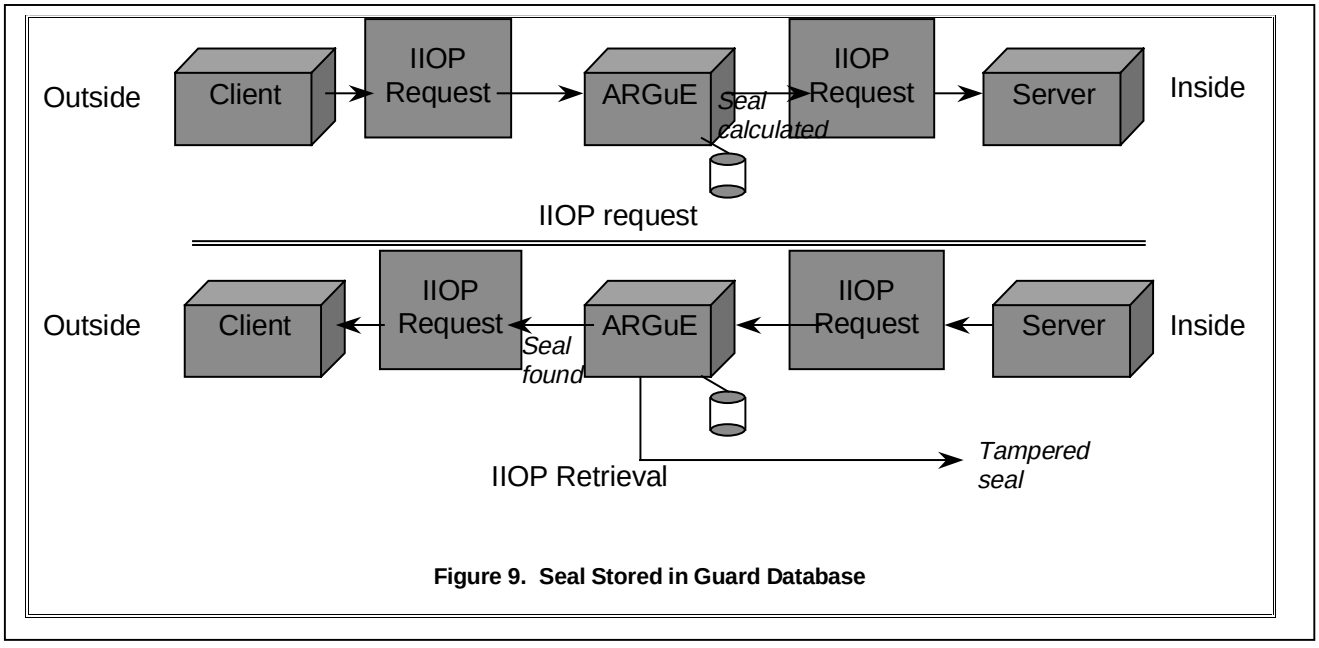

Figure 9. Seal Stored in Guard Database

developing the application-specific filtering software that adds and verifies seals.

2.2.4 Wrappers to Constrain Proxies

One of the concerns associated with having all of the guard processes on a single platform is that a flaw in any of the processes could lead to undesired results. For example, a flaw in the content-based filter or in any of the listener/sender processes could allow traffic to flow directly from the inside to the outside, bypassing the content-based filtering.

At first blush it might seem that existing operating system controls could be used to constrain behavior of the proxy. However, our base is the UNIX version of Gauntlet, and UNIX, like most operating systems, does not provide a thorough access control system. For example, while it is possible to control access to individual files within the Gauntlet (which ARGuE uses through use of unique UIDs and GIDs to represent each part of the multi-part proxy), there is no means to control which TCP/IP ports may be accessed by an application, other than the restriction that unprivileged processes cannot bind to ports below 1024. Our goal was to provide finer grained controls on the operation of the proxy parts.

Our means of controlling the proxies is to develop wrappers, using the Wrapper Definition Language (WDL) [Fraser]. There are three wrappers associated with each ARGuE multi-part proxy:

- A wrapper for the listener/sender programs, parameterized to allow it to bind to specified TCP/IP ports on the appropriate network interface (i.e., so the inside listener/sender can access the inside network, and the outside listener/sender can access the outside network, but not vice versa), and to allow it to access files in its corresponding directories, but not the directories belonging to the opposite side. This wrapper will vary depending on the protocol being processed by the proxy, especially insofar as the port numbers that can be bound.
- A wrapper for the queuing/dequeuing programs, parameterized to allow it to access the correct directories only, but not allowing them to access any TCP/IP ports.
- A wrapper for the filtering program, allowing it to operate on the queuing directories, but not allowing it to access any TCP/IP ports.

The wrappers are designed to allow the corresponding program to do little as possible, to minimize the risk of erroneous or malicious code. For example, the wrapper for the listener/sender program allows the following:

- Fork a child process
- Use semaphores used for synchronizing the different parts of ARGuE
- Make open() calls on files in specified directories (with limitations on what directories can be opened for reading and what directories can be opened for writing)
- Make read() and write() calls to access already opened files
- Make unlink() calls to remove files from specified directories
- Exit

2.2.5 Integration with Intrusion Detection

The final part of ARGuE is integration with intrusion detection technology. The Intruder Detection and Isolation Protocol (IDIP) [IDIP] can be used both to identify potential attacks, and to cause real-time changes in configurations to respond to those attacks.

IDIP integration is currently limited to notification of potential attacks. The listener/sender processes can detect incorrectly formatted protocols (by comparing the data received to the expected protocol), and can send appropriate notifications. The filter is capable of generating application specific alerts, depending on values received. For example, if data representing the amount of a bank deposit is being passed through ARGuE, an intrusion detection system might be configured as follows:

- For deposits less than $1000, no alert is ever generated.
- For deposits less than $10,000 made during banking hours, no alert is generated, but if outside of normal hours then a low level alert is generated.
- For deposits less than $100,000, an alert is generated, but the transaction is allowed to go through.
- For deposits less than $1,000,000, an alert is generated and the transaction is blocked, but other transactions are permitted.
- For deposits greater than $1,000,000, an alert is generated, and future transactions are refused (perhaps because it indicates a significant security breach).

The most interesting alerts are application specific, and hence require knowledge and planning on the part of the filter developer.

One of the critical issues in interacting with an intrusion detection system is where to report problems. We decided to report problems detected by the inside listener/sender and by the filter to devices on the inside, and problems detected by the outside listener/sender (but not the filter), to devices on the outside. It is unclear whether outsiders should also be notified of potential attacks found by the filter. We believe that attacks on the filter are most likely to be attempts to release data which fails the filtering criteria, and is more likely to indicate an error in the program sending the data (or the filter itself)

than a concerted effort by an inside user to leak information.

3. Current Status, Limitations, and Future Work

ARGuE has been demonstrated in several government testbeds to connect together networks of the same classification (but where we pretended that the data was of different classifications). Protocols transferred in these demonstrations included IIOP (CORBA) and FTP. In the latter case, the data was stored within the file being transferred using XML. Use of XML meant that the filter had to parse the XML to find the relevant data to be filtered; an additional step.

ARGuE has several significant limitations:

- It is not, and is not planned to be accredited. It is only a testbed.
- There are inherent delays introduced by the architecture. Since filtering is not possible until an entire operation is available for review, the listener/sender processes accumulate an entire request. If the protocol being processed is FTP, this means that the entire file is collected into the ARGuE device before filtering begins and subsequent transfer. Thus, the transfer time is at least twice what it would be if ARGuE were not reviewing the data. Use of shared memory instead of files (as described earlier) would reduce this latency somewhat, but the requirement for assembling the entire operation would remain. The doubling of transfer time is therefore fixed, while the interim processing may be speeded up.
- Addition of each new protocol requires different listener/sender pairs. While it may be possible to adapt these from existing firewall proxies, they have different requirements, and hence will always be less flexible than corresponding firewalls. Additionally, the wrappers will differ somewhat for each listener/sender pair, thus increasing development effort.
- Developing filters is still hard, especially for protocols where the level of detail available in each request is small, and hence difficult to determine whether a particular message should be allowed through. For organizations accustomed to "out of the box" firewalls, the software development effort is a significant issue (although no worse than other guards).
- Because the filters are dynamically invoked for each message, they are inherently stateless. This has been adequate for our current purposes, but may not be in the future.
- CORBA implementations do not have fixed TCP ports, as do other network services such as SMTP or HTTP. Most CORBA implementations provide a mechanism to specify a particular TCP port to be used. Since the proxies must listen on particular ports, this requires application effort. However, this is no worse than the effort required to use CORBA applications through a firewall.

Our future directions include an in depth look at the assurance granted by the multi-part proxies (as compared to the traditional single part proxy), and the value of wrappers on proxies. As noted above, we also plan to build a graphical user interface to make it easier for non-programmers to specify filtering rules, and to compare the quality of filtering rules generated by non-programmers with those developed by programmers. Additionally, we plan to integrate ARGuE with central network management capabilities, so the set of filtering rules can be dynamically changed under the control of an administrator (perhaps in response to an operational need to transfer data, or to restrict traffic when a leak is suspected).

4. Conclusion

ARGuE provides significant features normally found in guards. The concept of a implementing a multi-part proxy within a single computer is unique to ARGuE, and provides assurances not typically found in firewalls, especially when married with the wrapper technology. The data sealing capability, while certainly not new, provides a useful capability in environments where data is imported and subsequently exported.

5. Acknowledgements

We are indebted to Sami Saydjari at DARPA who encouraged this effort, to Don Faatz at MITRE who contributed many of the ideas toward this architecture, to Debi Robertson for her tireless efforts to help us gain access to the Felt compiler used for data filtering, and to all our colleagues at NAI Labs who provided the inspiration, technologies, and work environment that made the CORBA Guard possible. In particular, James Croall developed the first version of the ARGuE software, Brian Schechter made significant enhancements and developed the data sealing capabilities, Matt Woods wrote many of the initial filters, and Chris Marcellin wrote the proxy wrappers. Finally, we appreciate the many constructive suggestions from the referees.

6. References

[ISSE] "Imagery Support Server Environment (ISSE) Guard System Description", http://www.itd.sterling.com/rome/projects/products/isse/ISSE_SD.html

[Smith] Richard Smith, "Constructing a High Assurance Mail Guard," *Proceedings of the*

17th National Computer Security Conference, Baltimore MD, October 1994.

[Fiorino] Thomas Fiorino *et al*, "Lessons Learned During the Life Cycle of an MLS Guard Deployed at Multiple Sites", *Proceedings of the Eleventh Annual Computer Security Applications Conference*, New Orleans LA, December 1995.

[Fraser] Timothy Fraser, Lee Badger, and Mark Feldman, "Hardening COTS Software with Generic Software Wrappers", *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland CA, May 1999.

[Guttman] Joshua Guttman, John Ramsdell, and Vipin Swarup, *Felt, a Security Filter Compiler*, Personal Communication, November 1998.

[IDIP] "Dynamic, Cooperating Boundary Controllers Final Technical Report", Boeing report D658-10822-1, August 1998.

[SGS] "Security Guard Study", Defense Information Systems Agency, August 1995.

[XTS-300] Final Evaluation Report, Wang Government Services, Inc., XTS-300 (Report CSC-EPL-92/003.C), National Computer Security Center, 1992.