Generating Policies for Defense in Depth *

Paul Rubel
BBN Technologies
Cambridge, MA
prubel@bbn.com

Michael Ihde
University of Illinois at Urbana-Champaign
Urbana, IL
ihde@crhc.uiuc.edu

Steven Harp, Charles Payne
Adventium Labs
Minneapolis, MN
steven.harp@adventiumlabs.org
charles.payne@adventiumlabs.org

Abstract

Coordinating multiple overlapping defense mechanisms, at differing levels of abstraction, is fraught with the potential for misconfiguration, so there is strong motivation to generate policies for those mechanisms from a single specification *in order to avoid that risk. This paper presents our experience and the lessons learned as we developed, validated and* coordinated *network communication security policies for a defense-in-depth enabled system that withstood sustained red team attack. Network communication was mediated by host-based firewalls, process domain mechanisms and application-level security policies enforced by the Java Virtual Machine. We coordinated the policies across the layers using a variety of tools, but we discovered that, at least for defense-in-depth enabled systems, constructing a* single specification *from which to derive* all *policies is probably neither practical nor even desirable.*

1. Introduction

Defense in Depth (DiD) [18], or loosely the ability of security defenses to compensate for each other's failures, is rarely achieved in real systems. Redundant security enforcement is expensive to implement, configure and maintain, and there is little guidance for doing so effectively. The correctly functioning system requires consistent security policies across all defense layers; however, the varying semantics of the underlying defense mechanisms make it difficult to measure consistency between their disparate policies.

Thus, there is a strong motivation to develop a single specification from which all policies will be derived. This topic has been the focus of significant research (c.f., [8, 20, 13, 9]) that has demonstrated that the master specification can eliminate unnecessary duplication and be analyzed effectively for desired properties. However, those research efforts focused on coordinating policies for identical or similar defenses within a *single* defense layer. What about coordinating policies across *multiple* defense layers? The variety of enforcement targets, and range of abstractions (from IP addresses and gateways to network services and processes), means that any *useful* master specification will need to contain many details at discordant levels of abstraction. That is, not all details are required at all defense layers, so they tend to get in the way when reasoning about a layer where they are not required. A master specification also raises concerns about hidden assumptions that might yield exploitable vulnerabilities and circumvent any gains promised by DiD.

This paper documents our experience defining and coordinating the network communication policies for a DiD enabled system. Defense technologies from the network layer to the application layer were deployed to address potential threats from a sophisticated attacker. Each layer, to the greatest extent, repeated the logical network communication rules of layers below it.

We initially pursued the master specification approach for selected layers. For example, we began by generating the host layer policy automatically from the application layer policy. However, we soon realized that a hybrid approach that created policies in a coordinated *but largely independent* fashion would yield the best balance of flexibility, autonomy (an important quality for DiD) and assurance of correctness. Our hybrid approach avoided simple mis-configurations by coordinating static policy elements shared by all policies, such as host names and port numbers, from a single source. Then we minimized the risk of hidden assumptions by (a) specifying each policy separately using a different author, (b) structuring each policy to deny

*This work was supported by DARPA under contract number: F30602-02-C-0134

everything that was not explicitly allowed and then defining the policy according to observed failures in order to achieve a policy that was *minimally sufficient*, and (c) supporting each policy with validation tools. The policy validation tools enabled software developers to review policies for correctness even if they did not understand the syntax of the policy enforcement mechanism. Using the validation tools, they discovered policy misconfigurations and fed this information back to the policy authors.

The next section discusses our DiD problem and the complexity of its network communication requirements. Then we describe for each defense layer the policy construction and validation process. We conclude with lessons learned and some thoughts for future work.

2. The DARPA Challenge

In 2002, DARPA challenged the research community to design and demonstrate an unprecedented level of survivability for an existing DoD information system using DARPA-developed and COTS technologies. In particular, DARPA required that the defended system must survive 12 hours of sustained attack from a class A red team. DARPA chose as its target a large, distributed, publish/subscribe/query (PSQ) system implemented using the Joint Battlespace Infosphere (JBI)[1], which relies on a public transport network to support its objective to deliver "the right information to the right people at the right time". The notional JBI system is illustrated on the left in Figure 1.

A candidate solution architecture, shown on the right in Figure 1, called DPASA[2] [6, 10], was designed and developed by our team, led by BBN Technologies. The DPASA team recognized that the survivable JBI must **deflect** or **prevent** attacks, **detect and report** attack related events, and **change** the system's configuration or behavior in response to damages caused by an attack. Key features of the DPASA solution included communication isolation zones, redundant but diverse host enclaves (each containing a complete set of management and PSQ functionality), Byzantine agreement for attack response, and prevention, detection and monitoring technologies strategically placed throughout every defense layer, and on every host, throughout the system.

The fielded DPASA design resulted in a Core made of four redundant quads (groups of management hosts) interacting with clients through an Access Proxy (AP) on the edge of each quad, bridging the core and client networks. The physical network topology provided some protection in this design, as clients could not directly interact with core components other than through the Access Proxy. Host, process and application layer defenses further restricted the client's communication options so that they could not interact with one another but only with the established services on the Access Proxies. The Access Proxies were also configured to only accept traffic from the known client hosts, stopping rogue clients from being introduced into the system.

Access within the Core was similarly restricted. The high-level management functionality resided on the System Manager (SM) host, but there was no direct route to the SM from the AP, in the event the AP was compromised. Host-resident defenses enforced separation and least privilege on each host. Each core and client process was granted access to only the files and directories required for its job and then with limited permissions on those files. File systems that contained non-changing data were mounted read-only to bolster the policy-provided protection. Dynamic code loading was also disabled for Java processes. At a host-level each client and management process ran as a unique user, which allowed further restrictions to be placed on file access.

For the remainder of this paper, however, we focus only on defining and deploying network communication policies. From a prevention perspective, DPASA's goal was to block the attacker using the DiD strategy illustrated in Figure 2 (the defense layer is shown in boldface while the prevention technology(s) used at that layer appears in italics). At the system layer, redundant hosts were deployed so that the failure of a single host would not stop the entire system. At the network layer, authorized enclave-to-enclave communication was enforced and encrypted by a VPN firewall/router. At the host layer, authorized host-to-host communication was enforced by the Autonomic Distributed Firewall (ADF)[16]. ADF is a host-based, embedded, distributed firewall that is implemented on the host's network interface card (NIC) and performs ingress and egress packet filtering. It protects the host from the network, and it protects the network from the host. In addition, all host-to-host communication was encrypted using ADF's Virtual Private Groups (VPG)[15], which provided a unique encryption key for each collection of hosts. At the process layer, authorized process behavior was enforced either by NSA's SELinux[3] or by Cisco Corp's Cisco Security Agent[4]

[1] *http://www.infosperics.org*

[2] Designing Protection and Adaptation into a Survivability Architecture

[3] *http://www.nsa.gov/selinux*

[4] *http://www.cisco.com*

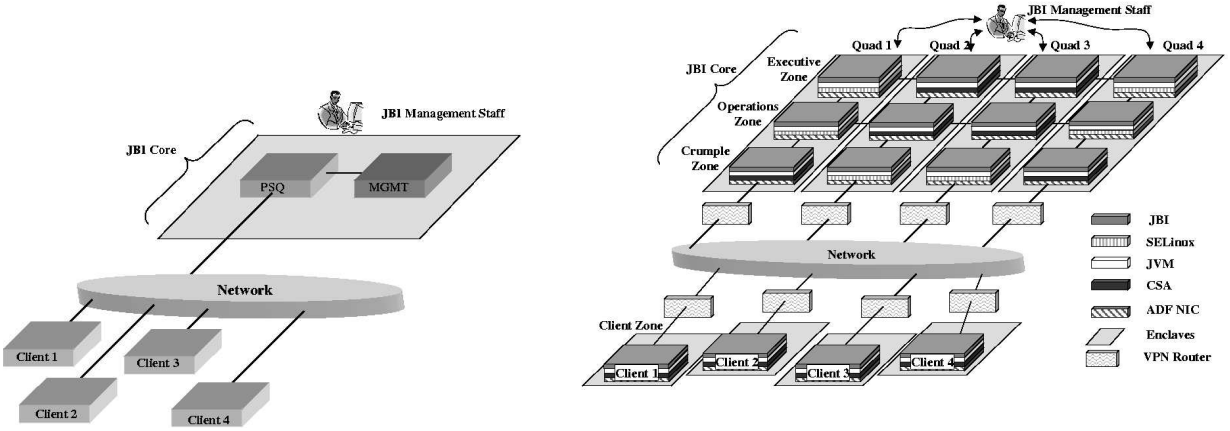

Figure 1. Baseline JBI (left) and Survivable JBI

(CSA) for non-Linux hosts. At the application layer, authorized JBI application behavior was enforced by the Java Virtual Machine (JVM).

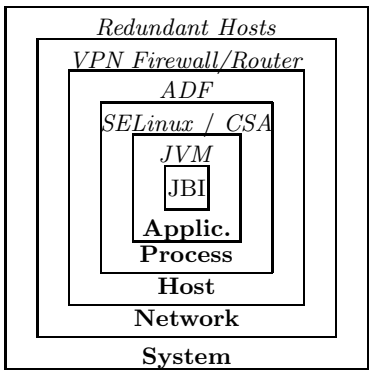

Figure 2. An attacker's perspective of DPASA Defense in Depth

Constructing network communication security policy proved challenging on several fronts. Clearly the richness of DPASA's DiD strategy meant significant *vertical duplication* of logical policy rules across the defense layers, but there was also significant *horizontal duplication* of those rules due to the redundant enclaves or *quads* in the DPASA JBI. For example, each quad contained a different mix of operating systems in order to minimize common mode failures, so the actual policies enforced by similar hosts in each quad differed significantly (e.g., between an SELinux host versus a CSA-enabled host) even though those hosts were performing identical logical functions.

To illustrate the policy author's challenge, Table 1 lists the policies affected and rules required for authorizing a simple network communication c from a JBI Client A, in enclave E_A, to the JBI core (B). The hosts receiving the rules appear in square brackets. Since there are four, redundant entry points into the JBI core (called Access Proxies), we can denote them collectively as B or individually as B_1, B_2, B_3 and B_4. They are each in a separate enclave, denoted E_{B_1} through E_{B_4}, respectively. Assume that B_1 is implemented on a Windows host, that B_2 and B_3 and A are implemented on SELinux hosts, and that B_4 is implemented on a Solaris host. CSA is enforced on all non-SELinux hosts. Further assume that all Access Proxy applications are written in Java and that while there are six different JVM executables (Sun's JVM on Solaris, Windows and SELinux, BEA's JVM on Windows and SELinux, and IBM's JVM on SELinux), all can enforce the same policy. The table illustrates that even a simple permission can affect almost a dozen policies. In addition, all of the required policy rules, except for the network layer (VPN), are specific to c and cannot be reused.

This simple example highlights the challenge of enforcing even a simple network communication rule across the various layers, but DPASA's network communication needs were far more complex. While DPASA relied on only 25 or so distinct network services, there were more than 570 network communication requirements, across more than 40 hosts, naming these services, and each requirement was subject to the analysis described in Table 1. Figure 3 illustrates the logical communication requirements for DPASA clients

Policy	Policy Rules Required
VPN	VPN to E_B $[E_A]$ VPN from E_A $[E_{B_i}]$
ADF	VPG from A to B for c $[A, B_i]$
CSA	Allow access from A for c $[B_1, B_4]$
SELinux	Allow access from A for c $[B_2, B_3]$ Allow access to B_i for c $[A]$
JVM	Allow access from A for c $[B_i]$ Allow access to B_i for c $[A]$

Table 1. Policies affected by one logical network communication rule to allow communication c from client A to core B

and *only one* quad of the DPASA core. In this case, CLIENT actually represents a dozen client hosts. Of the other nodes, each node starting with 'q1' is a single host, but nodes (other than CLIENT) without this prefix represent collectively the corresponding hosts in the three other quads. For example, the ADF Policy Server in quad 1, or q1adfps, engages in policy server replication services (PSReplication) with the Policy Servers (collectively named PS) in each other quad.

The next section describes our first step for managing this complexity: defining properties for shared policy elements.

3. Properties

Practical experience suggests that managing multiple policies, each containing redundant information, increases the risk of overlooked updates that will be discovered later by a developer, who is unaware of those updates, and will waste time trying to debug the policy. An example is when a key value, such as the port number for a required network service, is changed in some policies but not in others. To minimize this risk, we separated functional roles from actual identities when specifying policies. For example, in each quad a different host(identity) fills the role of an Access Proxy. The identities are calculated as the policy for each host or application is generated. In addition, this allows us to easily change the allocation of roles per host when moving DPASA software to new network addresses and environments.

Identities and application information were placed in a master property file that contained two sections: a mapping of a filename to a file identification number in the first section and in the second section a collection of role=identity pairs followed by three hyphens, a file identification number or numbers, and optionally a hyphen and quad specifier. An example is shown below.

```
_file_1=../../sm/conf/sm.prp
_file_2=../../proxies/conf/psqproxy.prp

PSQ_server0=192.168.4.34:8296 --- 1,2 - q4
rmi_host=192.168.4.14 --- 2
```

In this case, `PSQ_server0=192.168.4.34:8296` was placed in sm.prp and psqproxy.prp, files 1 and 2. This guaranteed that the specified role had a consistent identity in the specified files. The *q4* suffix declared that this binding was good only for quad 4. The other declaration was not specific to a quad and the suffix was omitted.

4. Application Layer Policy

Once the identities were specified in this structured way, they were associated with their roles in the JVM policies using policy templates. These templates were essentially JVM policies containing variables that were filled in using the master property file to create a single JVM policy for each application. Whenever we needed an IP address or port from the configuration file we would place a variable of the following form in the policy: *name_fileNumber*.

The following example lines shows a network permission in a JVM template:

```
permission java.net.SocketPermission \
 "<rmi_host__2>:<rmi_port__2>", "connect";
```

and the corresponding final product:

```
permission java.net.SocketPermission \
   "192.168.4.14:7183", "connect";
```

This scheme worked well when each policy template mapped to a policy on a single deployment host. However, some DPASA components were implemented on multiple hosts. Those applications each needed a policy to allow them to contact services bound to the IP address of that host. That would have required a different policy template for each host, differing in only the identity variable placed there. To avoid such duplication, we added meta-variables to the top of each template to change the variable values used when generating the policies. Each combination of meta-variables generated a unique policy file. An example is shown below where each client is allowed to communicate with a registry on its own host.

```
// metavar=CLIENT_IP ->c1_ip__13,c2_ip__13
  permission java.net.SocketPermission \
  "<--CLIENT_IP-->:<registry_port__3>", \
  "connect"; // ,Rmi
```

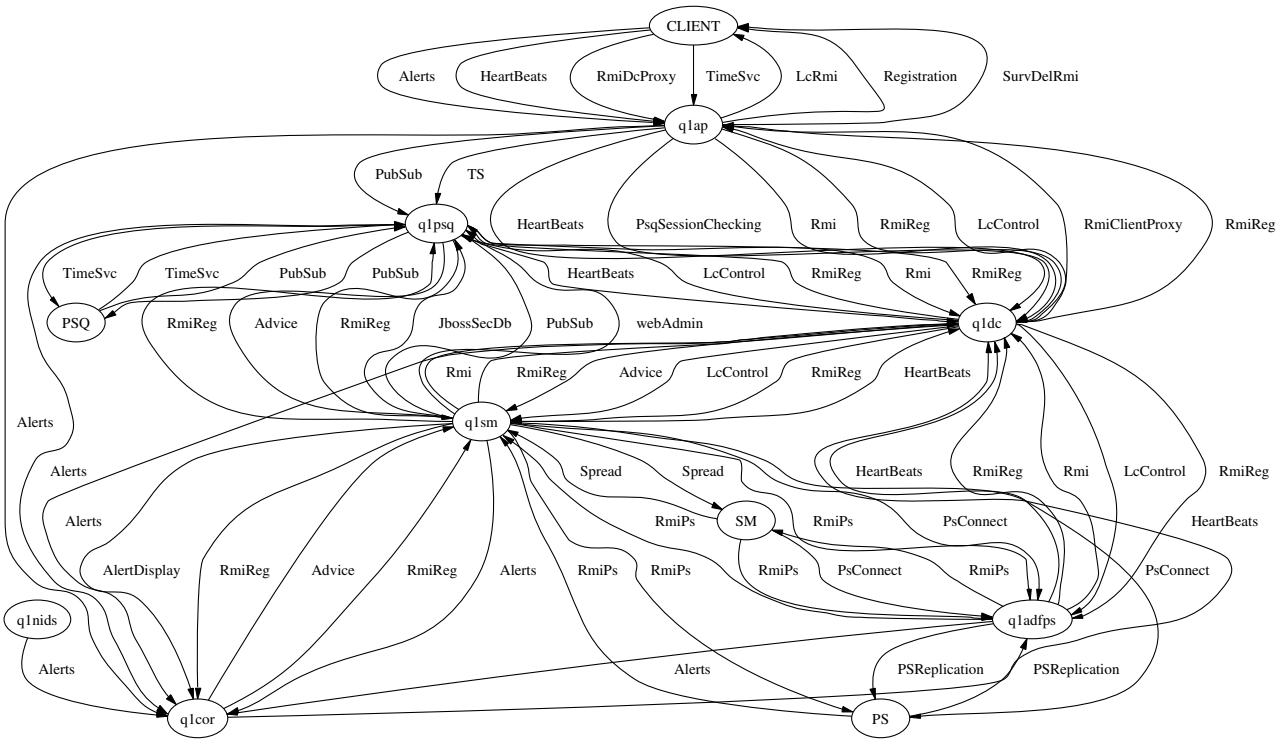

Figure 3. DPASA network communication requirements

In the above case two policy files are generated, one for each of the two client hosts. Wherever the meta-variable - -CLIENT_IP- - is found, it is replaced with one of the specified variables. The policy file names generated in this way are prefixed with the variable name, or names in the case of multiple meta-variables, used to create them so they are easily distinguished. Using the properties, meta-variables, and JVM templates, we were able to generate JVM policies for our components as the system was packaged for deployment.

To help validate the components protected by these JVM security policies, the generated JVM network permissions were used to automatically create network fuzzers, which would send random/malicious traffic between two end-points in the network in an attempt to find a vulnerability. Each sending or receiving permission generated a matching fuzzer. These fuzzers were collected together to represent a faulty application. This ensured that there was always a set of faulty clients or servers that could be used to test what would happen when a component behaved incorrectly. Using these fuzzers we could test the response of a component to authorized but incorrect or malicious traffic.

5. Process Layer Policy

Proper process behavior was enforced using SELinux on a majority of the DPASA hosts, and CSA on the Windows and Solaris hosts. While the two technologies served similar purposes, their policies and the approaches for constructing them were very different. For example, SELinux policies are very fine-grained and detailed, requiring significant effort to configure and maintain, especially in a multi-host environment like DPASA. To ease multi-host policy construction, we extended the SELinux policy construction tools to generate policies for multiple hosts simultaneously. CSA's policy management interface, on the other hand, worked very nicely in multi-host environments, but its browser-based interface made management of detailed policies very tedious. Also, CSA, unlike all of the other technologies we used, assumed that a privilege was authorized unless specifically denied, so we had to exercise caution when specifying rules. Finally, SELinux and CSA both included tools to generate policy rules directly from observed alerts, but we found the results were often opaque and difficult to maintain, so we created the deployed policies without using those tools.

5.1. SELinux Policy

The SELinux policies operated at a much finer level of granularity (individual system calls) than the JVM policies, and we made no attempt to directly translate from one to the other. However, the properties that were used to fill in the JVM templates were very useful when generating SELinux policies. Each port and IP address had a unique SELinux *type* in the policies, and these types needed to be bound to the correct IP addresses and ports for each host. Since these changed periodically, manual maintenance was not an attractive option.

A useful level of automation was achieved through a modified policy construction process. The source files for SELinux policies are normally prepared with the `m4` macro preprocessor before being processed by the policy compiler. Three additional files were included in the preparation in order to help automate the binding of network details. First, the host IP address and port number definitions were converted from the master property file into m4 macro definition statements. Then two files were created to bind the DPASA IP and port symbols to the SELinux type symbols. The normal source file for network binding, `net_contexts`, was then modified to include the new files in correct places.

Additional macros were defined and used to perform modifications such as extracting the port or IP part of a combined IP and port specification. A "quad" macro was used to generate the correct symbol suffix for a policy being compiled for a host in one of the four quads. (The correct quad was computed from the hostname of the target host, but could be manually fixed for cross compilation). For example, the port binding for the heartbeat service of the downstream controller (DC), a DPASA component, was specified with:

```
defport( dc_heartbeat_port_t, udp, \
        Qx(dc_heartbeat_port__1) )
```

Finally, the makefile for the SELinux policy was modified to include a target to regenerate the m4 macros from the correct master configuration source as needed.

The DPASA SELinux policies employ 5 roles, 500-600 types and 32000-35000 rules, depending on the jobs performed by the host. The number of rules required to address a given policy goal is reduced by using SELinux attributes to label equivalence classes of objects such as hosts. For example, all of the hosts in the client network are labeled as `client_node_type` and policies that apply to any client node can refer to `client_node_type` instead of each individual client. For reference, a client node with only the baseline SELinux policy (stock policy from Gentoo with standard extensions to handle X) contains 425 types and 30107 rules.

The SELinux policies were empirically validated through the correct operation of the JBI. That is, our first priority was to ensure that the JBI worked as expected. SELinux alerts observed during functional testing were converted manually into additional policy rules, which granted the process the minimal permission required for correct operation. Since permissions were added only as required, we had confidence that DPASA reasonably followed the principle of least privilege. Unfortunately, the compressed development schedule for this project left no time to explore any of the analysis tools provided by the SELinux community; employing a formal-model analysis tool ([5, 22, 12]) would have provided additional assurance that the SELinux policies correctly protected the DPASA applications.

5.2. CSA Policy

Because they were complementary technologies at the same defense layer, CSA and SELinux policies should have been logically identical; however, the two tools differed vastly in their conceptual models. Also, CSA lacked any facility (even an implicit one such as SELinux's text-based configuration files) to integrate with other tools. As a result, we could not integrate CSA either with SELinux or with the DPASA properties infrastructure (see Section 3). So while the SELinux policies, which were integrated fully with the properties infrastructure, were able to satisfy the DPASA communication requirements (see Figure 3), CSA, for reasons of concern about divergence with SELinux, ADF and JVM (all of which were coordinated by the properties infrastructure), satisfied only a subset of those requirements.

To satisfy all requirements, we would have needed to duplicate the properties infrastructure within CSA. However, the system was still under development as CSA policies were being written, so there was a real risk of divergence. For each CSA-protected host, we opted instead to specify the other hosts with which it was permitted to communicate and to restrict that communication to authorized protocols (i.e., TCP and UDP). We did not specify the specific network services (e.g., Alerts or RmiReg). As the system development stabilized more, those services could be added; however, on-going maintenance remained a big concern.

In all, eleven policies were defined. There was one policy for each Solaris or Windows host, except that the ADF Policy Servers (Windows) shared a policy. The typical Unix policy contained approximately 13 allow rules, 8 deny rules and 8 monitoring rules (detection

only). The typical Windows policy contained approximately 24 allow rules, 18 deny rules and 7 monitoring rules.

Like SELinux, CSA policies were validated through the correct operation of the JBI. Unlike SELinux, CSA offered the ability to generate an easy-to-read summary of each host's policy. CSA's strategy of allowing what is not explicitly denied made careful review even more critical. We first denied everything (e.g., all network access), then we specified only authorized accesses (e.g., the remote hosts and protocols authorized for communication).

6. Host Layer Policy

Initially, ADF policies were translated automatically from JVM policies. For the few non-Java components, either "fake" JVM policies were created or the component's ADF policy was specified directly. However, the JVM policy could not support the translation without annotation. For example, JVM policies do not distinguish between TCP and UDP protocols, and do not, by default, identify the local host or the ephemeral port (if any). This information was added as a comment to each connect authorization in the JVM policy listings and can be seen in the listing below which describes a UDP connection from 192.168.4.162 on port 5701 to 192.168.4.170 on port 9901.

```
permission java.net.SocketPermission \
    "192.168.4.170:9901", "connect"; \
    // 192.168.4.162,Heartbeats,UDP,5701
```

The JVM policies were then processed to create a single, intermediate specification containing entries of the form

```
source IP, source port, destination IP, \
destination port, protocol, service
```

where *source IP* and *destination IP* are standard numeric IPv4 host addresses, *source port* and *destination port* represent TCP or UDP ports or port ranges, *protocol* is any valid Internet protocol, and *service* is a character string by which to identify the network service implied by the other fields in this entry. These entries, when combined with the non-JVM entries, constituted the complete *connection specification* from which all ADF policies were generated.

Unfortunately, there were two critical shortcomings with translating ADF policy automatically from JVM policy. First, the "fake" JVM policies were not vetted adequately, since they were never actually used to enforce Java process behavior, and we encountered numerous errors when ADF enforced the incorrect rules resulting from these policies. Second, we eventually reached a point where the generated ADF policies violated design constraints for ADF. In particular, the generated policies exceeded the maximum ADF policy size, and they required more VPG keys to be assigned to a host than its NIC could support. We considered developing a tool to perform the required optimizations, we decided against doing so because its utility would have been restricted to DPASA.

However, while we determined that the connection specification should not be generated automatically from the JVM policies, there was still much value in creating it. The connection specification served many useful purposes: (a) it was a single source from which all ADF policies could be generated; (b) it was used to generate a graphical depiction of each policy, in a manner similar to Figure 3; and finally, (c) it was used to validate authorized communications against independent network scans. So we changed our strategy to maintain a "permanent" connection specification so that only valid ADF XML would be generated; however, we continued to perform an automatic translation of JVM policy into the temporary specifications for the purpose of policy discovery. The remainder of this section discusses each of the purposes described above.

ADF policy was generated per host. A connection statement such as

```
host A, 1024-65535, host B, 80, TCP, web
```

really implies two ADF policies: one for host A and one for host B. In this case, the ADF policy for host A would allow it to send TCP 80 packets to host B encrypted using the VPG key for B and receive replies to those requests encrypted with its own VPG key. The ADF policy for host B would allow it to receive TCP 80 packets from host A encrypted with its own VPG key and send responses to A encrypted with the VPG key for A. By generating both policies from a single statement, we ensured consistency between the two policies, which is particularly critical for successful VPG communication.

Once all VPG policy rules were generated, the rules for a given host were collected, ordered for optimal evaluation (ingress filtering then egress filtering for better performance against network attack), and then were translated into XML, imported into the ADF Policy Server, and distributed to the host. In all, 28 ADF policies were generated (some hosts were grouped under a common policy) with an average of 21 rules per policy, or just over 600 rules total. Because the translation routines were demonstrated to be trustworthy enough to generate correct ADF XML from a well-formed connection statement, policy debugging was done mainly by analyzing the connection specification itself, rather than by examining the ADF XML output. This simplified ADF VPG policy debugging for developers unfamiliar with ADF.

To further facilitate developer validation of the ADF VPG policies, we developed scripts to convert the high-level connection statements into *dot*[5] diagrams, such as the one illustrated in Figure 3. The dot diagrams provide no more information than the connection specification itself, but the data is presented in a form that is more visually pleasing.

Finally, we compared the permissions granted in ADF's connection specification against network scans in order to detect policy misconfigurations, which are, according to *The CERT Guide to System and Network Security Practices*[4], the most common cause for firewall breaches. Typical policy audits for border firewalls are performed with scanning hosts placed on each side of the firewall under test. However, no clear boundary exists for distributed firewalls. With them, the visible policy is the union of both the sender and receiver rulesets, thus the communications allowed to/from a host will be dependent on the network perspective. To complete a full, thorough audit the network scan must be initiated from each host to every other host. This captures the combined effect of the egress filtering on the sending host and the ingress filtering on the receiving host. If desired, extra hosts with no egress filtering could be added to the scan to find errors masked by egress filters.

The primary goal of our network scan tool was to 1) detect unauthorized communication paths and 2) detect unnecessary communication paths which exist in the system. The scans were coordinated via ssh on a separate DPASA control network (used for development purposes only), so no reconfiguration of ADF itself was required to support the scan. Each scan was performed by nmap, with the central controller receiving the results in standard nmap XML format. Each file contained all live hosts that were found during the scan and the state of the ports on those hosts. The state of the ports were classified as: open (responded with TCP SYN-ACK or UDP data), closed (responded with TCP RST or ICMP Port Unreachable), or filtered ports (No Response). If there were no misconfigurations, all ports should return filtered except those that were explicitly allowed in the connection specification.

After all the host scans were performed, the results were combined to create the global network view. This process was straight-forward and achieved with a Python script. The final result was an XML file containing a list of each communication path that was found in the system. Using XSLT to transform this output, the results were compared automatically with the ADF connection specification to discover misconfigurations and unnecessary communication paths.

[5] *http://www.graphviz.org*

The scan did detect extra communication paths that were not authorized within the ADF connection specification, but further examination revealed that the ADF policies themselves had been manually edited to authorize these paths. This finding underscores the importance of independent testing, because the connection specification did not reflect the true protection profile.

7. Lessons Learned

Our hybrid policy construction approach worked well because policies could be developed independently and simultaneously by various authors, which was particularly important given DPASA's compressed development schedule. Since the policy authors were geographically dispersed, the coordination required for a master policy specification would have been difficult. Also, it was not necessary for all authors to develop expertise on all technologies. The approach of starting with a minimal policy and adding to it, as operational failures (due to policy) were observed, helped to minimize concerns about unnecessary privileges, such as was discussed at the end of Section 6. While it is possible that some functional behaviors could have been removed during system development, resulting in "orphaned" policies, many eyes, validation scanning, and daily coordination on development progress and concerns helped avoid that risk. Also, having a single-source for policies means that the orphans will be generated only if the functionality is completely orphaned and not on account of missing a change in some other area.

The validation support tools also worked well. The ADF connection specification and dot diagrams clarified what was being enforced better than either the JVM policies that produced it or the ADF XML policy that resulted from it. In particular, since the JVM policy author only filled in a template using variables (as discussed in Section 4), the actual values for some policy elements were not easily visible *until* the connection specification was generated.

We used a variety of management interfaces to configure policy for DPASA. JVM and SELinux were text-based and configured using an editor and common command line tools. CSA and ADF were GUI-based, but we avoided using ADF's GUI to develop policies. Instead, we used common command line tools like m4 and awk to construct and translate the connection specification into an XML format that could be imported and assigned to hosts using ADF's GUI. Unfortunately, CSA's GUI did not provide similar support, so while its web-based interface was probably the friendliest for a novice user, it was awkward to integrate into a larger

policy environment such as DPASA. In the end, some form of command line support proved invaluable for integrating these tools.

A valuable lesson was that policy construction should not begin until the system functions are reasonably stable. This can be accomplished by either setting an acceptable, but perhaps overly broad, policy early on and then implementing the system, being sure to fit within the specified policy, or by implementing the system, being mindful of security concerns, and then creating the policy to tightly fit the system requirements when the component's functionality is stable. Since we needed to incrementally develop the system while still learning about the abilities of the code we had been given to defend and had to adhere to a tight time schedule, we chose the second option. This was not such a concern for JVM policy, since it was maintained by the developers themselves and could be updated easily as new code was added, or for ADF, since it was generated nearly automatically from the JVM policy, but SELinux was especially sensitive to any changes in process behavior. Developers had to be instructed in how to relabel the file system after new files were added and how to start authorized processes in the proper SELinux role, or else functional tests to support policy debugging were rendered useless. SELinux policy refinement depended on observing the system in operation in a permissive mode while collecting denial audits. However, it was impossible in most cases to fully test applications in isolation: related applications needed to be running correctly as well. The constantly changing and challenging-to-test system impeded policy development to a surprising degree.

8. Related Work

Several efforts have demonstrated that policy generation for multiple enforcement points from a single source is practical, but they have focused mainly on creating policies for identical or similar defense at a *single layer of defense*, rather than on creating policies for multiple, diverse defense layers as described here. Nevertheless, these efforts suggest important goals for future work in DiD policy specification. For example Guttman's[13] policy language for filtering routers yields policies that can be verified formally against desired security properties. Were formal semantics available for even a subset of our defense mechanisms, refinement theory could be applied to ensure that DiD is actually achieved. Bartal *et al*[8] require less rigor in Firmato, a policy language for perimeter firewalls. Firmato relies on a graphical validation strategy similar to ours; however, their graphs are built directly from the generated rules, which gives more confidence than our strategy, which based the graphs on the specification that produced those rules. Our graphs could be misleading if the graph generator and the rule compiler do not share the same semantics. Other related work along these lines includes Bradley and Josang[9] (Mesmerize), who describe a framework for managing network layer defenses, and Uribe and Cheung[20], who propose an approach for coordinating firewall policy with network intrusion detection strategies. Finally, Service Grammars[17] provide a framework for simplifying configurations based on high-level special-purpose languages.

More general motivation for our work, and empirical evidence of the difficulty of writing even small security policies, was gathered by Wool[21], who found that complexity directly affects the number of errors in the firewall policy. A rough estimate of DPASA's complexity places it as a moderately complex system, likely to have errors. Since DPASA needed to be as error free as possible, we needed methods to manage the complexity of our policies.

A considerable body of research has also been performed on rule set anomaly detection (also called conflict detection) [14, 7, 3, 1, 2, 11]. An anomaly-free rule set will be consistent (rules are ordered correctly), complete (every packet matched at least one rule), and compact (no redundant rules) [11]. Our methods do not perform conflict analysis (although it could be added), but we make a best-effort to create anomaly-free rule sets.

Complimenting anomaly detection and policy construction, Stang *et al* [19] presents Archipelago as a security tool for estimating system security. Using Archipelago, the "important" nodes (those most central in the connection graph) can be identified and brought to a higher level of secureness. Whether centrality is a good measure of the "importance" of a node is unclear without further empirical studies. Providing policy analysis, in addition to our policy generation, using methods similar to that of Archipelago represents an interesting area of future work.

9. Conclusions

In consideration of DiD enabled systems, constructing each policy in isolation is labor-intensive and can lead to configuration errors. On the other hand, generating all policies from a single specification — an approach advocated for policies *within* a particular defense layer, such as the network layer [8] — is perhaps even more labor-intensive and error-prone for DiD solutions, because too many details are required in that

specification that will apply only to specific layers, making the specification unwieldy. Instead, we advocate a hybrid approach that (a) encourages selective sharing of policy elements while maintaining policy autonomy, (b) encourages independence between policy authors, (c) builds policies from observed failures to be minimally sufficient, and (d) integrates validation support for other policy stakeholders. Such an approach minimizes the risk of exploitable vulnerabilities that could circumvent the benefits of DiD.

A critical measure of success, of course, is how well the resulting policies and their underlying defenses perform against a determined adversary. At this writing, red team assessment of DPASA is still underway; however, preliminary results confirm that a carefully crafted DiD solution is a formidable defense. We believe that the approach described here also makes such a defense practical.

Acknowledgments The authors wish to acknowledge the significant contributions of our "Blue" team colleagues at BBN Technologies (specifically Michael Atighetchi and Lyle Sudin), SRI International, Adventium Labs (specifically Richard O'Brien), and the University of Illinois at Urbana-Champaign, as well as the "White" and "Red" teams.

References

[1] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly detection and rule editing. In *IEEE/IFIP Integrated Management IM'2003*, 2003.

[2] E. Al-Shaer and H. Hamed. Management and translation of filtering security policies. In *IEEE Internation Conference on Communications*, 2003.

[3] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM'04*, 2004.

[4] J. H. Allen. *The CERT Guide To System and Network Security Practices*. Addison Wesley Professional, 2001.

[5] M. Archer, E. Leonard, and M. Pradella. Analyzing security-enhanced linux policy specifications. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 158– 169, June 2003.

[6] M. Atighetchi, P. Rubel, P. Pal, J. Chong, and L. Sudin. Networking aspects in the 'dpasa' survivability architecture: An experience report. In *The 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05)*, 2005.

[7] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. *Computer Networks*, 42(6):717–735, 2003.

[8] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[9] D. Bradley and A. Josang. Mesmerize: an open framework for enterprise security management. In *CRPIT '32: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*. Australian Computer Society, Inc., 2004.

[10] J. Chong, P. Pal, M. Atigetchi, P. Rubel, and F. Webber. Survivability architecture of a mission critical system: The dpasa example. In *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE, 2005.

[11] M. G. Gouda and A. X. Liu. Firewall design: Consistency, completeness, and compactness. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004.

[12] J. Guttman, A. Herzog, J. Ramsdell, and C. Skorupka. Verifying information flow goals in security-ehanced linux. *Journal of Computer Security*, 13(1):115–134, June 2005.

[13] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 1997. IEEE.

[14] A. Hari, S. Suri, and G. M. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE INFOCOM*, 2000.

[15] T. Markham, L. Meredith, and C. Payne. Distributed embedded firewalls with virtual private groups. In *DARPA Information Survivability Conference and Exposition, 2003*, volume 2, pages 81–83, April 2003.

[16] C. Payne and T. Markham. Architecture and applications for a distributed embedded firewall. In *17th Annual Computer Security Applications Conference*, December 2001.

[17] X. Qie and S. Narain. Using service grammar to diagnose bgp configuration errors. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 237–246, Berkeley, CA, USA, 2003. USENIX Association.

[18] D. Ryder, D. Levin, and J. Lowry. Defense in depth: A focus on protecting the endpoint clients from network attack. In *Proceedings of the IEEE SMC Information Assurance Workshop*, June 2002.

[19] T. Stang, F. Pourbayat, M. Burgess, G. Canright, K. Engo, and A. Weltzien. Archipelago: A network security analysis tool. In *Proceedings of the 17th Large Installation Systems Administration Conference*, 2003.

[20] T. E. Uribe and S. Cheung. Automatic analysis of firewall and network intrusion detection system configurations. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 66–74, New York, NY, USA, 2004. ACM Press.

[21] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, June 2004.

[22] G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145, New York, NY, USA, 2004. ACM Press.