DTE Firewalls
Phase Two Measurement and Evaluation Report

Timothy J. Fraser
Michael J. Petkac
Wayne G. Morrison
M. Lee Badger
Ben Uecker
Eve L. Cohen
John Grillo
Karen A. Oostendorp
Kelly C. Djahandari
Thomas P. Horvath
Calvin Ko
David L. Sherman
Christopher D. Vance

Trusted Information Systems
3060 Washington Road (Rt. 97)
Glenwood, MD 21738

TIS Report #0682

July 22, 1997

Executive Summary

This document is the second of three progress reports concerning the DARPA contract DABT63-95-C-0018 "Internet Safety and Security Task: Internet Safety Through Type-Enforcing Firewalls." The goals of this project are to assess the security and practicality of DTE firewalls - an advanced firewall technology based on Domain and Type Enforcement (DTE), and to construct a DTE firewall prototype. The first phase of the project demonstrated how DTE firewalls enabled secure enclaves to extend limited trust relationships to entities outside their perimeters, allowing organizations to safely import and export a greater variety of services than would be practical with traditional firewalls. The second phase takes this concept a step further by providing the infrastructure needed to create a secure virtualization of enclaves which we refer to as *enterprise zones.*

The enterprise zone concept is a tool which allows carefully-controlled collaboration between organizations. Enterprise zones are distributed computing environments which may span two or more DTE firewall-protected enclaves. An enterprise zone provides user processes with carefully-controlled access to a well-defined subset of the resources belonging to each organization that sponsors the enterprise zone. It also allows user processes which are distributed among several firewall-protected enclaves to communicate as securely as if they were physically located in the same enclave.

Contents

1 Introduction

This document is the second of three measurement and evaluation reports concerning the DARPA contract DABT63-95-C-0018 "Internet Safety and Security Task: Internet Safety Through Type-Enforcing Firewalls." The goals of this project are to assess the security and practicality of Domain and Type Enforcement (DTE) firewalls - an advanced firewall technology based on Domain and Type Enforcement[8], and to construct a DTE firewall prototype. The first phase of the project demonstrated how DTE firewalls enabled DTE-protected enclaves to extend limited-trust relationships to entities outside of their perimeters, allowing organizations to safely import and export a greater variety of services than would be practical with traditional firewalls[4]. The second phase of the DTE firewalls project takes this concept a step further by providing the infrastructure needed to create *enterprise zones*. An enterprise zone is a shared, possibly distributed, DTE execution environment that enforces a limited trust relationship between two or more enterprises.

The remainder of this section explains the enterprise zone concept, and describes how the phase two DTE firewall prototype uses its new dynamic policy configuration and cryptographic features to support it. Detailed information on the new dynamic policy configuration features may be found in section 2. Section 3 discusses the new cryptographic features, and section 4 describes the phase two prototype testbed. Finally, section 5 presents measurement and evaluation results for phase two DTE firewalls.

1.1 Enterprise Zones

An enterprise zone may be supported by a collection of hosts from two or more DTE firewall-protected enclaves, which are bound together in a limited trust relationship by their DTE firewalls. This relationship allows them to safely share a subset of their resources; in general, DTE firewalls and hosts may support multiple concurrent and independent enterprise zones. As described in section 3, the firewalls cryptographically protect messages passing between physical enclaves. Within each enclave, DTE provides access control over local resources. The combination of these techniques effectively allows all of the programs within an enterprise zone to communicate as safely as if they were behind the same firewall, regardless of their physical location. Figure 1 contains a diagram of two DTE firewall-protected enclaves supporting a single enterprise zone. In the diagram, rectangles represent hosts, solid lines represent network communication, circles represent actual enclave boundaries, and the dashed line represents the enterprise zone boundary.

In addition to providing protection from outsiders, enterprise zones support controlled collaboration between organizations which are sometimes partners and sometimes competitors. For example, two organizations cooperating on a joint project might create an enterprise zone to support the project's

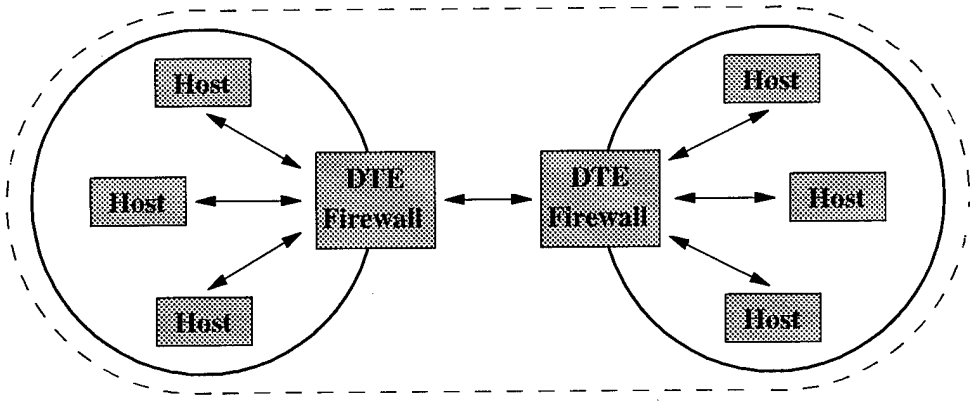

Figure 1: Two DTE Firewall-protected Enclaves Supporting an Enterprise Zone

activities. Each organization would specify which of its resources it would share with its partner in the enterprise zone. They would also specify how their partner would be allowed to use those resources. The DTE firewalls which support the enterprise zone would ensure that this resource-sharing policy was enforced at all times. Enterprise zones are an economically-attractive tool, since they allow organizations to make some of their existing resources available for collaboration, instead of requiring them to dedicate separate resources for the sake of security.

Enterprise zones are apt to be transient entities. In typical collaborative situations, the organizations involved want to share their resources for the duration of the project, and then cease sharing once it is done. In order to support rapid policy changes of this kind, the phase two DTE firewall prototype provides the ability to create and destroy secure enterprise zones dynamically.

1.2 DTE Firewall Technology Overview

As described by the DTE firewall architecture[12] and the phase one report[4], DTE firewalls are a combination of Domain and Type Enforcement (DTE) and application gateway firewall technology. DTE is an enhanced form of type enforcement, a table-oriented access control mechanism[8]. As with type enforcement, the DTE kernel views its system as a collection of active entities (e.g. processes), called subjects, and a collection of passive entities (e.g. files), called objects. Subjects are described as active entities because they may access objects (by reading or writing) and interact with other subjects (e.g., through signals.) Every DTE kernel has a security policy which defines a set of *domains* and a set of *types*. The kernel associates every subject with exactly one domain, and

every object with exactly one type. Each domain describes how the policy constrains the behavior of the subjects that are associated with that domain. For example, a domain might list what kinds of accesses its subjects are allowed for objects of each type. It might also list which signals its subjects may send to subjects associated with other domains. The kernel will only allow a subject to behave in a manner that is consistent with the policy described by its domain.

Domains describe secure environments which the kernel presents to user processes. Each environment gives processes access to a particular well-defined set of resources. The DTE kernel's ability to provide separate secure environments forms the basis of the separation provided by enterprise zones. In reality, enterprise zones are distributed secure environments constructed from a collection of equivalent secure local environments supported by DTE firewalls. When processes running in an enterprise zone interact with servers located outside of their physical enclave's perimeter, their requests are mediated by application proxies on the intervening DTE firewalls. The local secure environments on the DTE firewalls specify which resources are available to the proxies, which in turn limits the resources available to the processes they serve. A more detailed description of DTE and the secure user environments it provides may be found in [7].

The security policies enforced by DTE kernels are configurable. At boot time, a DTE kernel reads a description of the policy it will enforce from a file. This description is written in the DTE Language (DTEL)[9]. Administrators can use DTEL to describe a policy that provides the secure environments they need by specifying the appropriate domains and types, and the relationships between them. This configurability also allows administrators to create enterprise zones which provide secure environments that are tailor-made for a particular task or mission.

DTEL's structure resembles the structure of a programming language. It provides modules to break large policy descriptions into manageable parts, similar to the way some programming languages provide procedures to organize code. The phase two prototype uses this modularity as the basis for its dynamic policy configuration support. As described in section 2, the phase two DTE kernel allows administrators to change its security policy incrementally as it runs by loading and unloading dynamic policy modules. Dynamic policy modules are syntactically similar to phase one DTEL modules. Each module can describe the policy for a particular task or activity, such as support for a specific enterprise zone. When an administrator loads a particular module, the policy it describes becomes part of the kernel's security policy. An administrator may reverse the changes to the kernel's policy by unloading the module. Administrators may create and destroy enterprise zones as the kernel runs by loading and unloading the dynamic policy modules that define them.

The first phase of the DTE firewall project developed the infrastructure required to allow an organization to extend limited trust relationships to entities outside of its DTE firewall-protected enclave. This support for trans-firewall trust allows organizations to safely import and export a greater variety of services than would be possible with traditional firewall technology. This ability

is an essential part of the phase two DTE firewall prototype's support for enterprise zones.

For example, a group of organizations may wish to support an enterprise zone which provides a consistent set of shared resources to all of its processes regardless of their location. In order to make the shared resources available throughout the enterprise zone, each organization must export services to the other organizations, and the other organizations must import them. A DTE firewall's ability to extend limited trust relationships outside of its perimeter allows it to support these kinds of import and export relationships while enforcing each organization's resource-sharing policy.

1.3 Phase One Scenario

The phase one measurement and evaluation report presented a scenario which involved two fictitious organizations seeking to collaborate on a project by sharing some of their resources. The first organization was Strider Sprockets. Strider operated a DTE-aware host inside of a DTE firewall-protected enclave. The second organization was Strider's long-time competitor, Donalds Cogs, which operated a DTE-aware host outside of the Strider enclave. The two companies wished to pool a small portion of their private resources in order to co-develop a new product called a Gizmo, while keeping the balance of their private resources secret from each other.

The scenario demonstrated the phase one DTE firewall's ability to extend limited trust relationships to hosts outside of its security perimeter. This ability allowed the Strider firewall to export more services to the Donalds host, and import more services from it, than would have been prudent with a normal firewall. By importing and exporting each other's services, Strider and Donalds were able to conveniently share their resources.

In order to accomplish this limited sharing of resources, the two companies divided their data into two parts - one to share with their partner and one to keep private. Both companies created security policies which would enforce this separation on their hosts. Each host loaded a description of this policy at boot time. The Strider host policy, for example, associated the *gizmo_t* DTE type with its shared Gizmo project data, and the *strid_t* DTE type with its private data. The policy also provided a *gizmo_d* DTE domain for the Strider and Donalds employees working on the Gizmo project, and a *strid_d* domain strictly for Strider employees. In the phase one scenario, the two companies essentially implemented a statically specified enterprise zone for the joint project. A fragment of Strider's host policy (leaving out some system details) resembles:

```
policy strider_host_p;  /* this policy is loaded when the kernel boots */
   type strid_t;        /* the type for all Strider proprietary data   */
   type gizmo_t;        /* the type for all shared Gizmo project data  */

   domain strid_d = (crwxd->strid_t),  /* strid_d users access strid_t      */
                    (rd->gizmo_t);    /*          and can read gizmo_t      */
   domain gizmo_d = (crwxd->gizmo_t); /* gizmo_d users access only gizmo_t */
```

As indicated by the policy fragment, the *gizmo_d* domain gave the Gizmo workers access to the shared resources they needed, but denied them access to private Strider resources. The *strid_d* domain gave Strider employees access to private Strider resources and allowed observation of the *gizmo_t* data, without allowing them to make these private resources available to users in the *gizmo_d* domain. The Donalds hosts used a similar policy, with a *don_t* and *don_d* substituted for *strid_t* and *strid_d*. Both companies' policies also included an *anon_t* type and an *anon_d* domain intended for anonymous users, to allow them to retrieve publicly-available information from their hosts.

Figure 2 contains a diagram of the phase one scenario testbed. In the diagram, squares represent hosts, and the circle represents the Strider DTE firewall-protected enclave boundary. Each host square is labeled with the DTE domains it supports. Dotted lines represent network communication between hosts. These lines are drawn between pairs of domains, since the Donalds and Strider policies allowed processes on different hosts to communicate only with peers in the same domains (however, some information flow is allowed between domains since the more sensitive domains may observe, but not modify or execute, data from less sensitive domains.

In the operational testbed, the DTE kernels on the hosts and firewalls ensured that users were confined to their domains according to their security policy. Strider's DTE firewall policy allowed it to pass mail, NFS, FTP, HTTP, TELNET, and rlogin traffic between the Strider and Donalds DTE hosts, provided the transported data was of the alliance shared *gizmo_t* or the public *anon_t* type. The phase one prototype operated under the assumption that the secrecy and integrity of the data passing over its networks was assured. As described in section 3, the phase two prototype uses cryptography to provide the integrity and confidentiality that was only assumed in phase one.

The Strider DTE host policy ensured that exporting services to the Donald's corporation for the shared *gizmo_t* type would not compromise the secrecy or integrity of the private *strid_t* data. The exported services were isolated in the *gizmo_d* domain on the DTE host, which had no access to the *strid_t* data type.

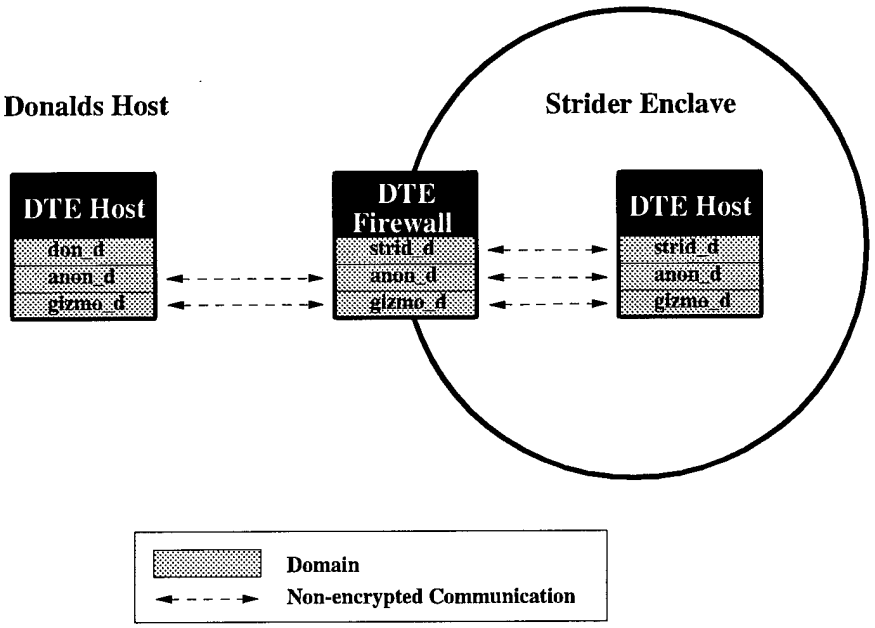

Figure 2: Phase One Scenario Testbed

1.4 Phase Two Scenario

The phase two scenario extends the phase one scenario to demonstrate new features provided by the phase two prototype. In the phase two scenario, Donalds and Strider form an enterprise zone to support their collaborative Gizmo project. This enterprise zone allows the two organizations to safely share their resources, as before, and also provides cryptography to protect the secrecy and integrity of Gizmo project data as it passes over the networks between their hosts. In addition, the phase two prototype's support for dynamic policy modules provides a convenient and scalable means to create and destroy enterprise zones without disrupting the availability of system services.

The phase two testbed consists of two DTE firewall-protected enclaves, one for the Strider organization and one for the Donalds organization. Each enclave consists of a DTE application gateway firewall and a DTE-aware host running inside the firewall security perimeter. A diagram of the phase two testbed is shown in figure 3. In the diagram, squares represent hosts, and circles represent physical enclave boundaries. Each host square is labeled with the names of the DTE domains it supports. Solid lines represent encrypted network communication; dashed lines represent network

communication without encryption.[1] The permanent domains that are configured at boot time are shown as shaded regions. The transient domains originating in dynamic modules are not shaded.

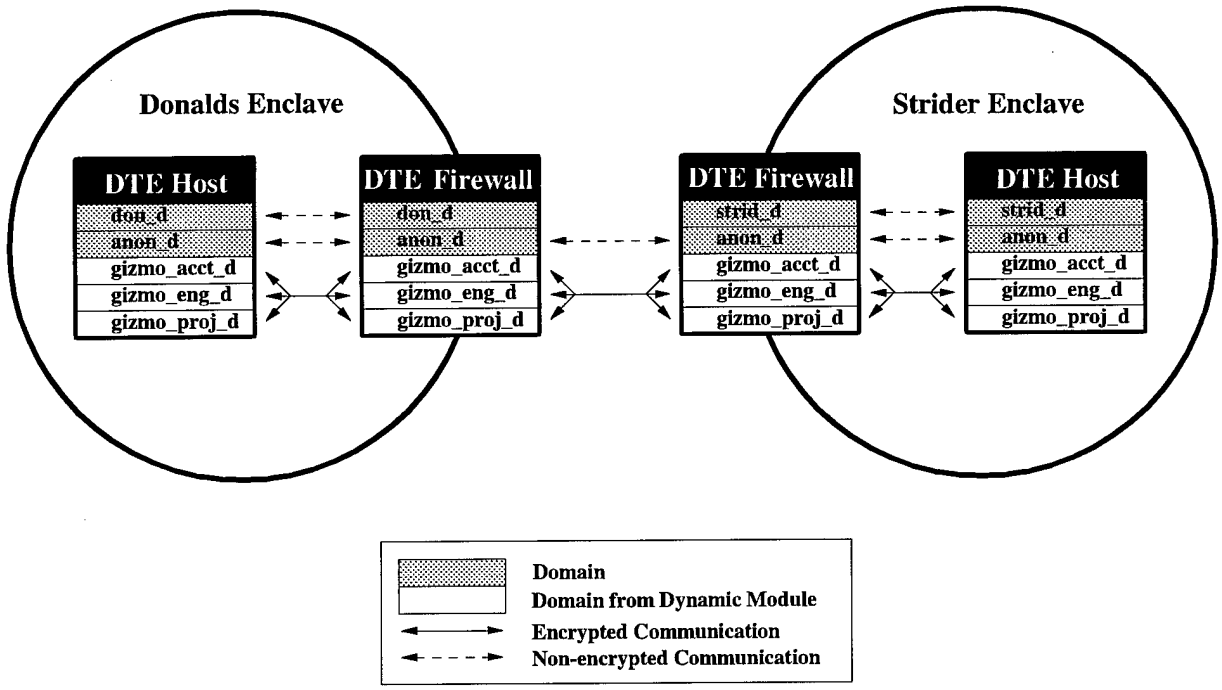

Figure 3: Phase Two Scenario Testbed

In the testbed scenario, Donalds and Strider create an enterprise zone which encompasses resources provided by the two hosts and the two firewalls. This enterprise zone is composed of three shared distributed environments, each of which corresponds to a domain shown in the diagram. There is a *gizmo_acct_d* domain for the accountants, a *gizmo_eng_d* domain for the engineers, and a *gizmo_proj_d* domain for the managers. Each environment gives its users access to the data they need from both physical enclaves, while protecting it from outsiders.

The gizmo project data is divided into three categories, each associated with its own DTE type. There is a *gizmo_rates_t* type for the accounting data, a *gizmo_eng_t* type for the engineering data, and a *gizmo_budget_t* type for the project management data. Whenever data of these three types is transferred over the network, it is encrypted with a shared key named *gizmo_k*. The diagram also

[1] Use of encryption within an enclave is optional.

shows a *don_d* domain and *don_t* type for Donalds proprietary operations and data, and a similar *strid_d* and *strid_t* for Strider. Finally, both enclave's policies provide an *anon_d* domain and an *anon_t* type for anonymous users and public information. A fragment of the policy for the Strider host is approximately:

```
policy strider_host_p;   /* this policy is loaded when the kernel boots */
   type strid_t;         /* the type for all Strider proprietary data   */
   type anon_t;          /* the type for all publicly available data    */

   domain strid_d = (crwxd->strid_t);  /* strid_d users access only strid_t */
   domain anon_d  = (crwxd->anon_t);   /* anon_d users access only anon_t   */

module gizmo_ve_m;              /* this policy is loaded as the kernel runs */
   type gizmo_rates_t;          /* the type for Gizmo accounting data       */
   type gizmo_eng_t;            /* the type for Gizmo engineering data      */
   type gizmo_budget_t;         /* the type for Gizmo management data       */

   domain gizmo_proj_d = (crwxd->gizmo_budget_t), /* control budget data     */
                         (rd->gizmo_rates_t),     /* observe rates data      */
                         (rd->gizmo_eng_t);       /* observe engineering data */

   domain gizmo_acct_d = (crwxd->gizmo_rates_t), /* control rates data       */
                         (rd->gizmo_budget_t);   /* observe budget data      */
   domain gizmo_eng_d  = (crwxd->gizmo_eng_t);   /* control engineering data */

   domain strid_d     += (rd->gizmo_eng_t);      /* augment strid_d to allow */
                                                 /* Strider staff to observe */
                                                 /* Gizmo technical data.    */

   /* This (56-bit) key is used to encrypt Gizmo data traveling between      */
   /* enclaves.                                                              */
   key esp gizmo_k     = (0x1234567890abcdef 0x1a2b3c4d5e6f0789)
                       -> gizmo_budget_t, gizmo_rates_t, gizmo_eng_t;
```

In the phase one prototype, the primary method for kernel security policy alteration was to manually edit the policy specification and reboot the kernel for the updated policy to take effect.[2] This

[2]Although the phase one DTE **dtmod** development tool still exists to modify a running DTE policy, the tool is intended for debugging security policies. It is not intended for use in a production system.

restriction is impractical for operational systems. The partnership described in the phase one and two scenarios focuses on the formation of a single alliance. In practice, however, organizations may take part in many concurrent alliances. These partnerships must be created and dissolved at varying rates within each organization. Some trust relationships may only exist for a short duration. Restructuring the policy and rebooting kernels for each change would result in an undesirable and impractical loss of service.

The phase two prototype addresses this deficiency with support for dynamic policy modules. The DTE policy presented in the phase one scenario was highly modular. It demonstrated how relatively independent modules could be designed and combined to produce custom DTE configurations to meet the needs of an organization. This beneficial use of modularity laid the foundation for phase two dynamic modularity development. The phase two scenario policy encapsulates its specification of the Gizmo enterprise zone in separate dynamic policy modules. This arrangement allows the Donalds and Strider administrators to create the secure virtual enclave by loading the appropriate dynamic policy modules. Similarly, when the Gizmo project ends, they may destroy the enterprise zone by unloading the modules. As shown in 3, the *gizmo_proj_d*, *gizmo_acct_d*, and *gizmo_eng_d* domains are transient whereas the other domains are permanent (i.e. would require a reboot to change).

The Gizmo enterprise zone provides its users with the Donalds and Strider resources they need to work. It also prevents private Strider and Donalds data from leaking to the opposite company, and it protects Gizmo project data from outsiders. Section 4 describes the operation of the phase two prototype testbed in more detail.

2 Dynamic Policy Modules

The phase two DTE kernel provides administrators with the ability to change a kernel's security policy in a controlled fashion by loading and unloading dynamic policy modules. These modules are syntactically similar to the DTEL modules provided by the phase one prototype to organize policy specifications into logical units.[3] The DTE kernel allows (only) a privileged administrator to load a dynamic policy module described by a DTEL module specification. Once loaded, the module's contents become part of the kernel's security policy. Similarly, the kernel allows a privileged administrator to unload a dynamic policy module. The act of unloading a module removes the policy elements it defines from the kernel's security policy.

Section 2.1 explains how administrators may use dynamic policy modules to modify a kernel's security policy. Section 2.2 describes the limits the DTE kernel places on dependency relationships between dynamic policy modules, and how these restrictions affect the prototype's functionality. Finally, section 2.3 discusses how the kernel restricts the kinds of policy statements dynamic policy modules may contain in order to protect the integrity of the static security policy it loads at boot time.

2.1 Policy Configuration with Dynamic Policy Modules

The phase two prototype provides administrators with two utility programs, **dtload** and **dtunload**, for dynamic security policy configuration. These programs allow administrators to keep pace with their organization's changing security policy requirements without kernel reboots and system downtime. Section 2.1.1 describes how the act of loading a dynamic module affects a kernel's security policy. Section 2.1.2 discusses how dynamic policy modules can be used to encapsulate the policy governing particular projects or activities (such as support for enterprise zones), and how the loading and unloading of a particular module can be used to turn a kernel's policy support for a particular project on and off. Finally, section 2.1.3 describes how a kernel unloads a dynamic policy module, and the effect this action has on the kernel's processes and file system.

2.1.1 Loading

The **dtload** command allows an administrator to load a specified dynamic policy module. Due to the security-critical nature of dynamic policy configuration, the kernel will service the **dtload** command's module load requests only if the command is run in a domain which has the **privload**

[3]The differences between dynamic policy modules and standard phase one modules are described in sections 2.2 and 2.3.

privilege. The act of loading a module adds all of the policy constructs (types, domains, domain attributes, DTE systems, IP address attribute assignments, key bindings, and key definitions) specified in the module to the kernel's security policy. Modules can add new domains to the kernel's policy, and they can add new attributes to existing domains. For example, the following DTEL *base_policy_p* fragment describes a policy that a kernel might load at boot time. It defines a type named *unix_t* and a domain named *unix_d*. The accompanying module fragment *dynamic_m* represents a dynamic policy module that an administrator might subsequently load to extend the policy described by *base_policy_p*. It defines a new domain named *reboot_d*, and augments the existing *unix_d* domain with a new *auto* right.

```
policy base_policy_p;
   type unix_t;

   domain unix_d = (/bin/csh),
      (crwxd->unix_t), privauth,
      privload, privswapon;
```

```
module dynamic_m;
   assumes type unix_t;
   assumes domain unix_d;

   domain reboot_d = (/sbin/reboot),
       (crwxd->unix_t), privreboot;
   domain unix_d  += (auto->reboot_d);
```

After an administrator uses the **dtload** command to load the *dynamic_m* module, the *reboot_d* domain and *unix_d*'s new *auto* right become part of the kernel's policy. The kernel's policy then exhibits identical policy enforcement behavior to a kernel which simply loads the following *combined_policy_p* at boot time:

```
policy combined_policy_p;
   type unix_t;

   domain unix_d   = (/bin/csh), (crwxd->unix_t), (auto->reboot_d),
                     privauth, privload, privswapon;
   domain reboot_d = (/sbin/reboot), (crwxd->unix_t), privreboot;
```

The module *dynamic_m* also demonstrates the concept of module "glue." Module "glue" consists of DTEL statements whose purpose is to connect a module's policy and the kernel's policy together into a single working whole. In addition to the statements which define its new *reboot_d* domain, *dynamic_m* contains a statement which augments the *unix_d* domain with the *auto* right to *reboot_d*. This statement is glue - it is necessary to make *reboot_d* a useful part of the kernel's policy. Without the *auto* right, it would be impossible for **reboot** processes to transition from *unix_d* to *reboot_d*. Since **reboot** processes would never be able to enter the *reboot_d* domain, the *dynamic_m* module

would not have any real effect on the kernel's policy enforcement behavior. Most dynamic modules contain statements like this one, which they use to glue themselves into the kernel's existing policy.

Dynamic policy modules provide needed flexibility. They also, however, complicate the task of recovering the phase two prototype's policy state on reboot. When a kernel boots, it reads its initial policy state from disk. However, any prior policy state which resulted from dynamic module loading before the boot is not automatically recovered in this fashion. The solution currently under development involves keeping a log of dynamic policy module load and unload operations along with copies of the modules themselves on disk. On reboot, the kernel can use this information to reload all of the modules required to rebuild its former policy state. Section 2.2 describes how the phase two prototype simplifies this process by ensuring that the order in which a group of dynamic policy modules are loaded has no effect on the resulting policy state.

2.1.2 Dynamic Policy Configuration

Dynamic policy modules are the atomic unit of policy change. Typically, when administrators need to extend a policy to govern a new activity, they will encapsulate the extension in a dynamic policy module. For example, the following module creates a secure environment for a small group of accountants charged with completing a semi-annual auditing task:

```
module audit_m;              /* policy governing temporary audit task */
   assumes domain user_c;    /* glue: holds basic rights needed to use system */
   assumes domain login_d;   /* glue: the login program runs in this domain */
   type audit_t;             /* new type for audit data */

   /* audit_d describes the user environment for auditors */
   domain audit_d = (/bin/csh), (crwxd->audit_t), user_c;

   /* glue: exec right allows login to put auditors into audit_d */
   domain login_d += (exec->audit_d);

   /* huey, luey, and duey are the systems used by auditors, and audit_k *
    * is the key used to encrypt the audit_t data flowing between them.  */
   dte_systems (huey, 11.22.33.1), (luey, 11.22.33.2), (duey, 11.22.33.3);
   key esp audit_k = (0x1234567890abcdef 0x1a2b3c4d5e6f0789) -> audit_t;
```

The module specifies a domain for the auditors' processes (*audit_d*), and a type for their data (*audit_t*.) It also specifies that the machines huey, luey, and duey are DTE systems at the auditors'

disposal, and that the *audit_t* data passing between them will be encrypted with the key *audit_k*. A small part of the module specification acts as glue between the module and the base policy. The module assumes a component called *user_c*, which is supposed to contain the minimum set of rights all users need to do work on the system (such as the ability to run the programs in **/bin** and **/usr/bin**.) By adding *user_c* to *audit_d*, the module makes it possible for auditors to run the programs they need to get work done in the *audit_d* domain. Also, the module augments the *login_d* domain with the right to *exec* programs into the *audit_d* domain, which allows the **login** program to place auditors into *audit_d* after they have logged in.[4]

When all of its parts are considered together, the module describes the complete policy for protecting the confidentiality and integrity of the auditors' data. Administrators can load the *audit_m* module on huey, luey, and duey, and these systems will enforce the specified policy. When the auditors finish their task, and they no longer require their secure environment, the administrators can destroy the environment by unloading the module.

2.1.3 Unloading

An administrator may unload a specified dynamic policy module using the **dtunload** command. As with **dtload**, the kernel will service the **dtunload** command's module unload requests only if the command is run in a domain which has the **privload** privilege. The act of unloading a module removes all of the policy constructs it introduced from the kernel's security policy. In terms of the *base_policy_p* and *dynamic_m* policy fragments found on page 11, after a kernel loads *base_policy_p* and *dynamic_m*, and then unloads *dynamic_m*, its policy enforcing behavior will be identical to a kernel which simply enforces *base_policy_p*.

When the kernel loads a dynamic policy module, it may introduce new domains and types into the kernel's security policy. Until the kernel unloads the module, it will enforce the policy concerning the processes and data associated with these new domains and types. However, when the kernel unloads the module, these domains and types will no longer be a part of its policy. This raises the question of what should be done with the associated processes and data; the kernel cannot allow them to exist if it no longer has a policy to protect them.

Before the kernel unloads a module, it executes a purging algorithm to deal with this problem. The purging algorithm saves what it can of the data associated with the module, and destroys the rest.

[4] In addition to a DTEL policy specification, the kernel also reads the **dt_member** and the **dt_role** file at boot time. The **dt_member** file introduces the notion of a *role*, which is a group of users with similar responsibilities. In addition to loading the *audit_m* module, the administrators might need to add an *audit_r* role to the **dt_member** file, along with a list of all the users who are auditors. In this situation, they would also have to associate the *audit_r* role with the *audit_d* domain by adding a mapping into the **dt_role** file. The DTE-aware **login** program uses the information provided in the **dt_member** and **dt_role** files to determine the domains that a user may log in to.

The details of the algorithm are provided below:

1. First, the kernel terminates any process which is running in a domain defined by the module. These processes cannot remain after the module is gone, because their domain would no longer exist in the kernel's policy.

2. The kernel unmounts any file systems corresponding to a *mount* statement in the module. When the module is removed, these file systems will no longer be legally mounted, so they cannot remain.

3. The kernel regrades any files associated with types specified by *regrade* statements in the module. Dynamic modules can use *regrade* statements to inform the kernel that files associated with certain of the module's types should have their type changed to a particular base policy type when the module is unloaded. Once a file is associated with a base policy type, it is under the protection of the base part of the policy, and can be allowed to outlive the module. Dynamic modules are not required to provide *regrade* statements for their types. The kernel deals with the types without regrade rules in the following step.

4. The kernel completes the purging algorithm by removing any files still associated with one of the module's types from the file system. When the kernel removes a directory in this way, it removes all of the directory's children, regardless of their type.

Once the purging algorithm is complete, the kernel unloads the module itself. The kernel accomplishes the purge and unload operation as one atomic unit. With the exception of the regraded data, no traces of the module's processes or data remain.

2.2 Inter-module Dependency Restrictions

Modules may assume types and domains which are defined in other modules. As mentioned in section 2.1.1, modules often assume domains in order to glue themselves into a kernel's existing policy. Whenever one module assumes a type or domain from another module, the first module becomes dependent on the second. Without the second, the first cannot exist in a kernel's policy. As described in section 2.2.1, these dependency relationships are a source of complexity for the kernel. The following sections describe how the phase two prototype limits the kinds of dependencies it supports among modules in order to control this complexity (section 2.2.2), and discusses how these restrictions impact the prototype's functionality (section 2.2.3).

2.2.1 Dependency Graphs

Conceptually, inter-module dependencies can be represented as graphs. The DTEL policy fragments shown below are related by a dependency graph.

```
policy base_p;
   type unix_t;

   domain unix_d = (/bin/csh), (crwxd->unix_t), (auto->reboot_d),
                   privauth, privload, privreboot, privswapon;
```

```
module mod1_m;
   assumes domain unix_d;
   type project_t;

   unix_d += (rxd->project_t);
```

```
module mod2_m;
   assumes domain unix_d;
   assumes type project_t;

   domain project_d = (/bin/csh),
      (crwxd->project_t), (rwxd->unix_t);
   unix_d += (exec->project_d);
```

The *mod1_m* module assumes the *unix_d* domain defined in *base_p*, so it depends on *base_p*. Similarly, the *mod2_m* module assumes *unix_d* and *project_t*, so it depends on both *base_p* and *mod1_m*. Figure 4A shows a graph of their dependency relationships. It represents each module with a rectangle, and each dependency relationship with an arrow.

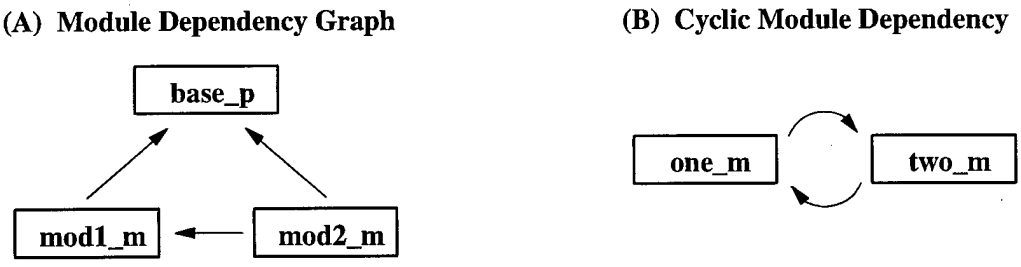

Figure 4: Two Module Dependency Graphs

The dependency relationships in figure 4A's graph add complication to the process of dynamic module loading and unloading. After *base_p* is loaded at boot time, the remaining two modules are loadable only in a certain order: *mod1_m* first and *mod2_m* second. Since *mod2_m* assumes

the *project_t* type from *mod1_m*, it can only be loaded when *project_t* already exists in the kernel's policy. Similarly, the two modules could be unloaded only in the reverse order: first *mod2_m* and then *mod1_m*. Since this example contains only two dynamic modules, the complexity is not overwhelming. With many dynamic modules, however, the ordering restrictions imposed by the dependence relationships may become considerable. Administrators might find themselves unable to unload a particular module which has outlived its usefulness because it defines types and domains needed by other modules, for example. This might make it difficult to remove an old trust relationship from a kernel's policy without rebooting.

Since figure 4A's graph contains no cycles, it is possible to find at least one ordering in which the modules may be dynamically loaded and unloaded. As shown by the following two modules, however, the DTEL syntax does not guarantee a lack of cycles.

```
module one_m;
   assumes type two_t;
   type one_t;
   domain one_d = (rwx->two_t);
```

```
module two_m;
   assumes type one_t;
   type two_t;
   domain two_d = (rwx->one_t);
```

Figure 4B shows a graph of these two modules' dependency relationships. Since the graph contains a cycle, there is no ordering in which these modules can be loaded or unloaded. Since module *one_m* assumes the existence of type *two_t*, type *two_t* must exist in the kernel's policy at the time module *one_m* is loaded. Similarly, module *two_m* depends on *one_m* for its definition of type *one_t*. In a situation involving both module *one_m* and *two_m*, an administrator would be unable to load either one, since each depends on the other being loaded first.

2.2.2 Restrictions on Dependence

The prototype avoids the complexity described above by restricting the kinds of dependencies which it allows among some modules. In order to accomplish this, the kernel divides its policy into two parts: the base part and the dynamic part.[5] The base part of the policy is made up of the policy loaded at boot time, in the manner of the phase one prototype. This part of the policy is permanent, and may not be removed using the **dtunload** command. Modules in the base policy are free to depend on any other module in the base policy in any way they choose. Since the base policy modules are all loaded at once (effectively at the same time) and are never unloaded, even cyclic dependencies do not cause any difficulty.

[5]The division is real only at a high level of abstraction. In the low-level implementation the base and dynamic parts are united in a single representation, for the sake of simplicity and efficiency.

The dynamic part of the policy is made up of the modules that administrators load after boot time - the so-called "dynamic modules" or "dynamic policy modules." The contents of the dynamic part of the policy are apt to change over time as administrators use the **dtload** and **dtunload** commands to add and remove modules. The kernel allows these dynamic modules to depend only on modules in the base policy. The kernel will refuse to load any dynamic module which depends on another module in the dynamic part of the policy.

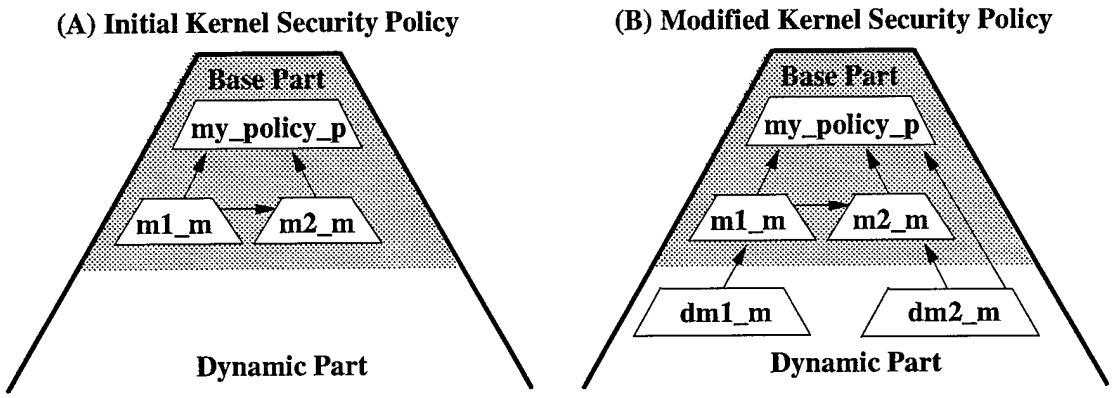

Figure 5: A Conceptual View of a DTE Kernel's Security Policy Before (A) and After (B) a Pair of Dynamic Modules are Loaded

Figure 5 shows two conceptual views of a kernel's security policy. Each view shows the modules that make up the policy, and the dependencies between them. Figure 5A shows the kernel's initial policy state immediately after booting. Its base policy contains three modules: *my_policy_p*, *m1_m*, and *m2_m*. Figure 5B shows the same policy with the addition of two dynamically-loaded modules: *dm1_m* and *dm2_m*. In accordance with the kernel's restrictions, the two dynamic modules depend on various parts of the base policy, but not on each other.

2.2.3 Impact on Prototype Functionality

The dependency restrictions described above limit dynamic policy modules to dependence only on other modules that will always be present in the kernel's policy. This arrangement avoids the troublesome situations where dynamic modules must be loaded or unloaded in a particular order. It also prevents the possibility of cyclic dependencies among dynamic modules. Since dynamic modules are not allowed to depend on each other, and base policy modules cannot depend on dynamic modules (because the dynamic modules are not loaded until after boot time), no

dependency cycles involving dynamic modules can arise.

Although this restriction makes the complexity introduced by dynamic modules manageable, it does limit the kinds of useful modules the kernel will accept. There are situations in which administrators might find it convenient to have dependencies among dynamic modules. Two dynamic modules might depend on some types defined in a third, for example. The administrators might desire the ability to leave the module with the types in the kernel's policy for a long period of time, while they load and unload the other two modules frequently. With the phase two prototype, however, they would either be forced to combine all three modules into one, or to put the types in the base part of the policy. Fortunately, either of the solutions would still result in the same policy enforcement behavior as the three original modules. Although it does decrease the ease with which policies can be modularized, the kernel's dependency restriction does not decrease the body of policies that are expressible.[6]

2.2.4 Indirect Module Dependencies

The dependency restrictions described in section 2.2.2 prevent dynamic modules from depending on each other directly. However, through the use of base policy domain augmentation, it is possible for dynamic modules to depend on each other *indirectly*. The following DTEL module fragments illustrate this point:

```
policy ipolicy_p;   /* base policy */
   type unix_t;
   component user_c = (/bin/csh), (crxd->unix_t);

module imod1_m;  /* dynamic module */
   assumes domain user_c;
   component user_c += (w->unix_t);
```

```
module imod2_m;  /* dynamic module */
   assumes domain user_c;
   domain foo_d = user_c;
```

Through the *user_c* base policy component, the dynamic policy module *imod1_m* grants the *foo_d* domain in the dynamic policy module *imod2_m* write access to *unix_t*. Even though module *imod2_m* does not assume any types or domains from module *imod1_m*, it depends on it indirectly for this right.

Unlike direct dependencies, indirect dependencies do not place ordering limitations on the dynamic module loading and unloading process. In the example above, administrators can load and unload

[6]The third phase of this project will investigate strategies for relaxing these restrictions on inter-module dependencies.

modules *imod1_m* and *imod2_m* in any order, although the policy enforcement behavior described by the *foo_d* domain is more restrictive without the presence of module *imod1_m*. As described in section 2.3, the phase two prototype restricts the ways in which dynamic policy modules may augment base policy domains. These restrictions can be specified to control or optionally eliminate opportunities for indirect dependencies at the cost of some increase in policy complexity.

2.3 Module Behavior Restrictions

In addition to their capacity for augmenting existing base policy domains, dynamic policy modules possess the ability to add new IP address attribute assignments, DTE systems, key definitions, and key bindings to the kernel's security policy. Like augmentation, these abilities exist primarily to allow dynamic modules to glue themselves into the kernel's existing policy by making sure the policy includes all of the elements they need to do their job.

These abilities allow dynamic modules to do much more than glue themselves into a kernel's policy, however. They also allow dynamic modules to change the relationships between base policy types and domains. Consider the following base policy and dynamic module:

```
policy base_policy_p;
   type oil_t, water_t;
   domain oil_d   = (rwxcd->oil_t);
   domain water_d = (rwxcd->water_t);
```

```
module dynamic_m;
   assumes type   oil_t, water_t;
   assumes domain oil_d, water_d;
   domain oil_d   += (rwxd->water_t);
   domain water_d += (rwxd->oil_t);
```

In this example, the intent of the base policy shown in the *base_policy* specification is to keep data of type *oil_t* and type *water_t* separate. But once the module *dynamic_m* is loaded, the separation created by the base policy is destroyed.

In some situations, this wide-ranging ability to modify the base part of a kernel's security policy may be desirable. Administrators may wish to use dynamic modules to manage a kernel's policy enforcement behavior by granting and revoking the rights of base policy domains. In other situations, administrators may wish to limit the kinds of statements dynamic modules may contain, in order to protect the integrity of the base policy. For example, some administrators might wish to use dynamic modules only to add policy governing new trust relationships. They might object to modules which change the policy governing existing trust relationships, like *dynamic_m* above.

The phase three prototype will support the notion of a "meta-policy" - a configurable policy which governs how dynamic policy modules may modify the base policy. The kernel will read a description of this meta-policy along with its base policy at boot time. Subsequently, the kernel will refuse to

load dynamic modules which would violate its meta-policy. The phase two prototype, however, does not support this configurable meta-policy mechanism. Instead, it provides a single non-configurable meta-policy which allows dynamic modules only the power they need to glue themselves into a kernel's policy, and no more. The phase two meta-policy places restrictions on several of the DTEL commands that can be used in dynamic modules, in order to ensure their good behavior. The restrictions are summarized in the following list:

assign statements: Unlike base policy modules, the kernel does not allow dynamic policy modules to contain the DTEL *assign* statement. This prevents them from changing the file system type bindings specified by the base policy.

domain augmentation: Domain augmentation allows dynamic policy modules to add limited kinds of subject and object rights to domains in the base policy. The kernel will allow a given dynamic policy module to grant rights to any new domains and types which it defines itself, but not to domains and types defined in the base policy or in other dynamic modules. This helps prevent dynamic modules from radically altering the relationships between base policy domains as described in the example above. The kernel further prohibits dynamic modules from granting any subject rights other than *exec* or *auto*. Dynamic modules may use domain augmentation only to grant subject and object rights; they cannot use it to add new entry points or privileges to base policy domains.

key bindings: The kernel allows dynamic modules to specify key bindings only for new types they define themselves. As described in section 3.2, key bindings are a new DTEL feature which allows a kernel's policy to specify that data of a particular type should be encrypted with a specified DES key. This restriction prevents dynamic modules from derailing network applications running in base policy domains by unexpectedly encrypting their data streams.

3 Encryption

In order to safely extend the security perimeter between two firewalls, network traffic between participating hosts must be protected. The traffic must be safe from modification, and in some cases safe from disclosure. Encryption and integrity at the IP-level provide this protection to the phase two DTE firewall. For this capability, we choose IPSec because it is becoming a standard among a variety of platforms. Also, IPSec is precisely defined in RFCs 1825[1], 1826[2], 1827[3], 1828[11]. Other protocols exist, but do not allow for inter-operability or are not well defined.

3.1 Description of IPSec

IPSec is an Internet Engineering Task Force (IETF) specification for providing Internet Protocol (IP) authentication, integrity, and confidentiality. It is an intermediate step in the conversion between IPv4, the current standard, and IPv6. IPv6 will have the security mechanisms built into the protocol, so extra measures will not be necessary; the IPSec protocol provides IPv6 protection mechanisms to IPv4 packets.

The IPSec protocol specifies two mechanisms (transforms) to secure network transmissions: the Authentication Header (AH) to ensure data authenticity,[7] and the Encapsulating Security Payload (ESP) to provide data confidentiality.[8] Both AH and ESP provide data integrity. AH and ESP are described in RFCs 1826 and 1827, respectively. The IPSec protocol is independent of the types of AH and ESP transforms implemented.

When forming packets for transmission, IPv4 packets are encrypted and placed into the payload section of one or more IPSec packets. IPSec packets begin with a standard IPv4 header and a special ESP header which contains some of the information necessary to decrypt the payload. Figure 6 illustrates how IPv4 packets are converted into IPSec packets.

3.2 Description of DTE/IPSec

The NRL IPv6/IPSec Software Distribution is a reference implementation of IPv6 and IPSec based on the 4.4BSD-Lite networking software. In this implementation, ESP is provided through the US Data Encryption Standard in Cipher Block Chaining (DES-CBC) mode. With NRL's reference

[7] In this phase of the DTE firewalls project, data authentication is assumed. In the final phase of the project, authentication will be provided by the Domain Type Authority, but it will not be provided by IPSec's AH mechanism. For this reason, this section will only discuss the ESP aspect of IPSec.

[8] The DES algorithm used for ESP in NRL's Alpha 4 release of IPv6/IPsec was written by Phil Karn, and is the most efficient DES algorithm for the i386 architecture.

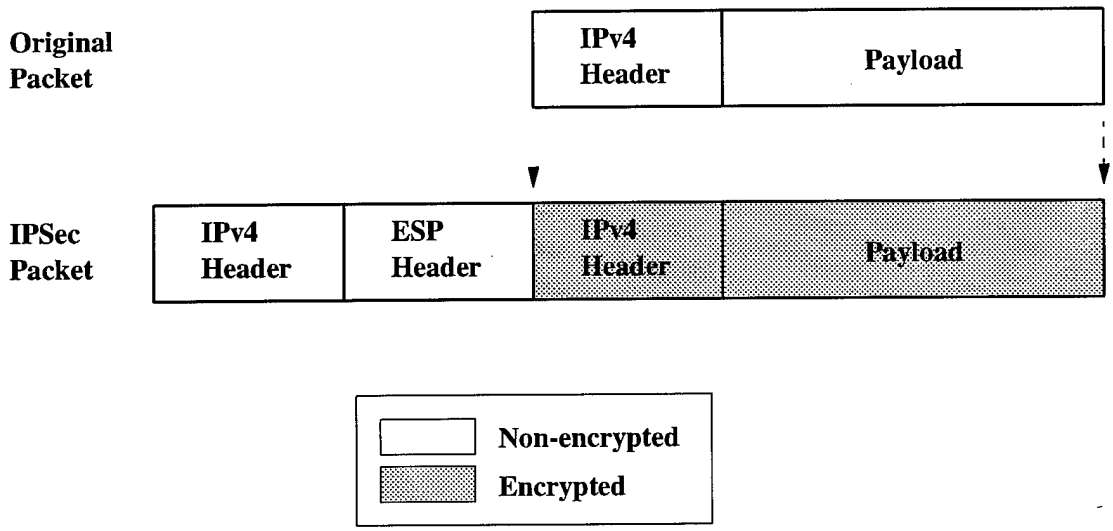

Figure 6: Converting an IPv4 Packet to an IPSec Packet

implementation of IPv6/IPSec as a starting point, we separated the IPSec from the IPv6 code and integrated it into the DTE kernel. The DTE prototype required two significant modifications for IPSec.

1. First, encryption key management was added to the kernel's DTEL library. The DTEL library was modified to associate ESP key information with DTE types. An association may be created that binds a type to processing for ESP. However, the default action is that types do not require ESP processing. These associations and the key information are stored in the DTE policy, and kernel-accessible library routines provide the ability to retrieve key information and bind key information to types. Also, DTEL has been extended to provide a language statement that associates IPSec key information with types.

2. Second, IPSec processing was added to the networking code. Immediately prior to transmission, packets are passed to the IPSec subsystem for ESP processing. If IPSec key information has been associated with a particular DTE type, any network packets sent with that type are automatically passed to the ESP portion of the IPSec code. After the IPSec processing has been completed, the packet is then transmitted to the network hardware. When an IPSec packet is received, it is decrypted and checked for integrity as needed, and then passed to the appropriate higher-level protocol code.

Figure 7 illustrates the basic DTE/IPSec algorithm for outbound IP packets:

1. A high-layer protocol (TCP or UDP in Figure 7 on the left system) requests that an IP packet carrying a data payload be sent. The IP layer builds the packet; the packet contains the data payload, is prefixed by the standard IP header, and contains the packet's DTE type identifier in the IP option space of the IP header. When the packet is complete and ready for transmission, the IP layer passes the packet to the DTE/IPSec portion of the kernel for processing.

2. The DTE/IPSec code compares the packet's DTE type to the list of types requiring ESP. (The packet's type is taken from it's IP option space and the list of types is found in the DTE policy data.) If ESP is specified for the type, the ESP processing code retrieves the type's ESP key from the DTE policy data and uses the DES-CBC transform to encrypt the IP packet.

3. The ESP processing code builds a new, unencrypted IP header for the actual packet that will be sent. This header has appended a new ESP Header and then the encrypted IP packet (see Figure 6). The ESP Header consists of two unencrypted 32-bit fields: the first field, the Security Parameters Index (SPI), contains the hashed type value of the packet and the second field holds the Initialization Vector (IV) for the DES-CBC transform.

4. The IP packet is then transmitted across the physical network as normal.

Figure 7 illustrates the basic DTE/IPSec algorithm for inbound IP packets:

1. An incoming packet is received (in Figure 7, at the lowest layer on the right side). It is queued by the hardware device driver.

2. The packet is taken off the queue, header validation is performed, and an attempt is made to defragment the packet. If defragmentation succeeds, the protocol number is used as an index into the protocol dispatch table and the packet is passed up the protocol stack. In the case of a properly-formed IPSec packet, DTE/IPSec kernel processing will result in a reversal of output processing.

3. If the ESP processing code finds that the packet is an ESP packet, then the SPI contains the hashed type value of the packet. The SPI is used to retrieve the DES-CBC key from the DTE policy data. This key is used to decrypt the encrypted portion of the packet.

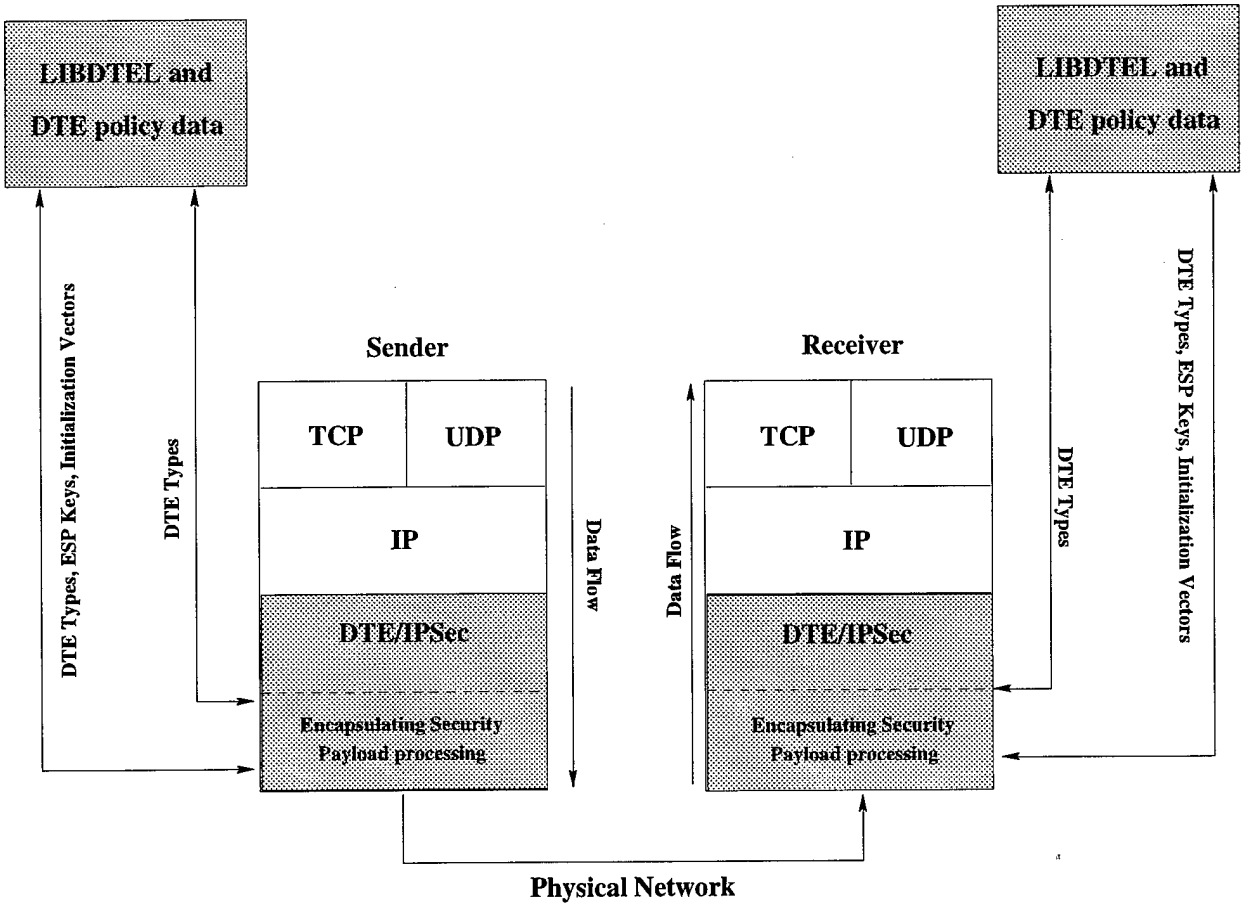

Figure 7: DTE IPSec in the Protocol Stack

4. The DTE/IPSec code then compares several of the fields in the inner and outer IP headers for consistency. These fields are the IP version, the source address, and the destination address. If an inconsistency is found, or if no key exists for the packet's type, the packet is considered invalid. When an invalid packet is received, an error is logged and the packet is discarded.

5. When the DTE/IPSec processing is complete, DTE options in the IP header are extracted. The packet's type stored in the packet's DTE options will indicate if the packet is of a type that requires encryption. If this is true but no ESP headers were included, then the packet is considered invalid, an error is logged and the packet is discarded.

6. If the packet is not found to be invalid, it is passed up the protocol stack to the IP code.

3.3 IPSec Key Management in DTE

General IPSec terminology defines a *security association* as a one-way abstraction of security information between two endpoints. A security association consists of the type (ESP), SPI, source address, destination address, transform (e.g. MD5, DES-CBC), key, and IV. Since DTE commonly speaks of *security attributes*, we will refer to a *key association* rather than a security association in the context of the DTE prototype. As mentioned in section 3.2, the cryptographic protection of network streams is bound to DTE types. A single type may require ESP processing, or be allowed clear-text transmission. Thus, a type may have zero or one key associations. These associations may be unique, or multiple types may share the same key association. Key associations may be contained in a dynamic module, though only for types introduced in that module. A dynamic module may not alter the key association of a base policy type.

Information in a DTE key association consists of the class (ESP), key name, key, IV,[9] and a list of types to which the association is bound. The general DTEL format of a DTE key association is:

```
key <class> <key-name> = (<key> [IV])->type1, type2, ... ;
```

As currently implemented, several fields of the original IPSec security association are not specified in the DTE key association. The source address and destination address of the IPSec security association are not necessary, since keys are bound to types. Additionally, the SPI need not be specified in the DTE key association since its value is given by the hashed type. Finally, the DTE/IPSec implementation exclusively uses the DES-CBC transform for ESP, so the transform need not be specified.

[9] Initialization Vectors are optional in key associations. Their use depends on the requirements of the transform.

4 Phase Two Scenario (Expanded)

For phase two, we reiterate and slightly expand the phase one measurement and evaluation scenario. This consisted of two fictitious companies, Strider Sprockets and Donald Cogs, that produce competing products, sprockets and cogs. Each organization utilizes a DTE-enhanced network, consisting of a DTE firewall and DTE hosts behind the firewall, to safeguard their corporate information. For a limited time, the two companies wish to develop a trust relationship to produce the Gizmo, a combination of sprocket and cog technology. The pre-alliance DTE base policies of the Strider and Donalds corporations classify user data into public and private portions. The public data, for such things as advertising of product information, is accessible to everyone, but the private data, pertaining to sprockets and cogs information, is accessible only to Strider and Donalds employees, respectively. Prior to the alliance, Strider and Donalds view each other as any other untrusted entity and can only access and exchange the unencrypted public data type available via the HTTP and MAIL services.

The Strider/Donalds alliance wish to employ a full suite of advanced applications and protocols to make their work efficient (e.g. the most recent web browsers, MAIL, FTP, RLOGIN, TELNET, and file sharing via NFS). The alliance must protect their joint information from others designing products to compete with the Gizmo and still protect trade secrets relating to sprockets and cogs from each other. To accomplish this task, the alliance wishes to form an enterprise zone consisting of the Strider and Donalds DTE firewalls and one DTE host behind each firewall. After formulating shared data types, roles and services for the alliance, the DTE policies of the alliance machines require modification to form the enterprise zone. DTE policy modification can be accomplished through the use of dynamic modules.

Dynamic modules can be as simple or as complex as necessary to attain their end goal, in this case to define an enterprise zone expressing a trust relationship. Properly developed dynamic modules also maintain the intent of DTE base policies so that loading of dynamic modules does not introduce unintended data disclosure or corruption. Since dynamic modules can be loaded through the **dtload** command, there is no need to interrupt system services by modifying DTE base policies and rebooting machines for our scenario's enterprise zone to take effect. Once Strider and Donalds have loaded the dynamic modules, each organization can access and exchange alliance data without release of private data types defined in the base policies. Data encryption keys, specified in dynamic modules, assure the confidentiality and integrity of Strider/Donalds data exchanged over unprotected networks. When the trust relationship is dissolved, the **dtunload** command unloads the dynamic modules thus returning the DTE policies to their pre-alliance specification without the need for base policy modification or system reboots.

Section 4.1 provides an overview of the Strider and Donalds base policies. This is followed by section 4.2, which describes the dynamic policy modules for the Gizmo enterprise zone.

4.1 Base Modules

The DTE firewall and host base policies of the Strider corporation prior to the alliance can be found in Appendix A. For simplicity, we assume the Donalds' base policies differ from Strider's base policies in only one type and domain name: Strider has domain *strid_d* and type *strid_t* while Donalds has corresponding domain *don_d* and type *don_t*. These domains define the primary work role of each organization. The remaining policies' types, domains, and access rights are identical and are not presented.[10] Although the development of the Domain and Type Authority (DTA) in phase three will further address non-homogeneous policies, the current prototype allows base policies of cooperative DTE machines to be non-homogeneous as long as the domain and type names exchanged between machines are consistent for access mediation. Thus, for example, the domains necessary for NFS communication, (the domains that **nfsd**, **nfsiod**, and NFS users run in), must be identically named on DTE client and DTE server machines and the machines can only exchange identically named data types as specified by the client and server DTE policies. In our scenario, the Donalds' corporation DTE policy does not understand Strider's *strid_d* domain and would deny any access attempts from that domain via NFS. However, the *strid_d* domain can still be present in the Strider policies for local NFS communication or via the DTE firewall for NFS exchange with a remote Strider site without hindering alliance communication.

The organization's base policies of the phase two scenario are, basically, subsets of the policies defined in the phase one Measurement and Evaluation report. Thus, rather than reiterate the detailed discussion of the base policies presented in the phase one report, we discuss the differences evident from a line-by-line comparison of the phase one and phase two base policies. The differences can be placed in two categories: phase two prototype functionality changes and revisions made to the phase one policy resulting from further validation testing after document publication.

The policy modifications due to evolving prototype functionality are straight-forward. First, because of dynamic module capabilities the phase two Strider firewall and host policies do not contain any references to the Cogs alliance: that is, the *gizmo_d* domain, the *gizmo_t* type and the subject-to-*gizmo_d* subject as well as the subject-to-*gizmo_t* object accesses contained in their phase one counterparts. Additionally, the phase one mandatory *tcp_d* domain and *tcp_t* type used to label protocol-specific data are no longer necessary and have been removed: all protocol-specific generated data is now internally typed and mediated by the DTE kernel. Finally, the phase two Strider policies reflect the use of associated domain privileges. The phase one prototype contained the **setauth** privilege giving a domain the authority to modify the DTE User ID (DUD). In phase two, this privilege has been renamed to **privauth** and six new privileges have been added to specify domains permitted to perform special DTE or Unix operations[9][11]

[10] Some additional differences, such as mount points, machine names and IP address information would also likely be present. For this discussion, though, these differences are irrelevant.

[11] Privileges are encapsulated by domains; the ability of a program to use a privilege depends on the program's

Two corrections to the published phase one host policy have also been made. First, **/dev/null** should have been typed *term_t* rather than *tmp_t*. The second correction deals with modifications for NFS use. The *nfs_t* type was introduced replacing *unix_t* type for NFS message exchange. The use of *unix_t* required NFS user domains, such as *anon_d*, to have write access to the *unix_t* system type, which includes system binaries thereby granting unnecessary access. Additionally, the domain *fw_nfs_d* must have *srcd* access to the *nfsd_d* and *nfsio_d* domains for forwarding of NFS request and response messages.

4.2 Dynamic Module Specification

After announcing their joint venture, the Strider and Donalds staffs define the personnel working on the project along with a high level specification for the roles, data types, data accesses, and network services necessary to conduct their alliance. The primary roles are defined to be project leader, accounting, and development engineer. The development effort will be conducted via NFS with each corporation exporting a single file system, **/usr/home/gizmo**, of project data. The following specifications are agreed on for Strider/Donalds interaction:

Roles
: The Strider/Donalds Gizmo alliance will consist of the project leader (*gizmo_proj*), accounting (*gizmo_acct*), and engineering (*gizmo_eng*) roles. The roles allow authorized personnel in each organization to log into domains granting appropriate access to shared data (see below).

Data Types
: The Strider/Donalds Gizmo alliance shared data will be labeled with types *gizmo_eng_t*, *gizmo_rates_t*, and *gizmo_budget_t*.

Data Accesses
: The project leader role will have the responsibility of creating budget projections. The project leader role will also be able to view, though not modify, all other project data. The accounting role will have the responsibility of creating and adjusting labor rates. The accounting role will be able to view, though not modify, the budget projections, but will not have any access to engineering data. The engineering role will have full access to engineering data, but no access to any other project data.

Data Encryption
: All project data shall be encrypted when it is sent over the network between firewall-protected enclaves.

(DTEL expressed) ability to enter a privilege-carrying domain.

Network Services

The project leader role will utilize the MAIL, RLOGIN, TELNET, HTTP, and NFS services. The accounting role will utilize the MAIL, HTTP, and NFS services. The engineering role will utilize the MAIL, HTTP, and NFS services.

Once the basic agreement for data sharing is formulated, the DTE administrator of each organization creates the dynamic policy modules permitting necessary data to be securely shared through common services and prohibiting unauthorized data disclosure.[12] As stated in the phase one report, a well-developed base policy minimizes the burden associated with this task. Additionally, the DTE **dti** command can be used as a helpful tool in dynamic policy development. The full firewall and host dynamic policies are presented in Appendix B. Complete descriptions of DTE firewall and host processing pertaining to the DTE policy can be found in the phase one report[4].

[12]Some additional pieces of information, such as the name and IP addresses of the other corporation's firewall and user identification information, also are needed for system configuration; see appendix B for more detail.

5 Measurement and Evaluation

The second phase of the DTE firewalls project took the phase one prototype, which was capable of supporting limited trust relationships with entities outside of its secure enclave, and extended it to provide the infrastructure necessary to support enterprise zones. This infrastructure makes the phase two prototype capable of providing *secure* distributed computing environments for collaborative relationships between organizations. It also provides the administrative tools required to dynamically create and destroy these environments in a controlled fashion. Using the phase two prototype, collaborating organizations may share a well-defined subset of their resources with each other through the enterprise zone abstraction.

This report measures the success of our prototype using the same four criteria used in the phase one report: safety, functionality, performance, and compatibility. The following list defines each criterion for network services:

safety: Safety refers to the degree of protection from attack the prototype provides to hosts, host applications, firewalls, and network applications, and the extent to which it protects important data from theft or corruption.

performance: This report is concerned with the difference in network throughput performance between groups of conventional (non-DTE) baseline systems and groups which contain DTE systems.

functionality: Functionality refers to the degree to which existing functionality has been preserved and any new functionality has been added. It also notes any new user procedures which are required for DTE firewalls.

compatibility: The compatibility issue examines the degree to which DTE firewalls inter-operate with non-DTE hosts and firewalls, and notes any required changes to system or application software.

We apply the same criteria to the phase two prototype's new dynamic modularity and encryption features, as appropriate. Section 5.1 summarizes the evaluations of the network services supported in the first phase of the project. These services include remote login (rlogin), TELNET, FTP, SMTP, NFS, and HTTP. Section 5.2 evaluates the phase two prototype's support for dynamic policy modules, and section 5.3 evaluates the prototype's new encryption features.

5.1 Network Services

The phase one report provided a detailed evaluation of the remote login, TELNET, FTP, SMTP, NFS, and HTTP services. With the exception of performance, the phase two prototype's new dynamic modularity and encryption features do not affect these evaluations. The following sections summarize the phase one report's evaluations for each network service[6]. The performance section (section 5.1.4) includes updated statistics and text which reflect the optimization of DTE networking code implemented during phase two. The original evaluations can be found in the Phase One Measurement and Evaluation Report[4].

5.1.1 Safety

The safety provided by DTE enhancements to firewalls can be grouped into two main categories: additional strength found in the firewall and defense in depth provided by the security policy coordination facilities of a DTE firewall. The additional strength in the firewall results from confining each proxy to a domain, in which it has access only to the files needed to perform its task. Therefore, for example, if the FTP proxy is compromised, it cannot alter or view system files. Although user data could be relabeled by a malicious proxy, this relabeling is restricted to the types the proxy can access: by restricting the types available to a proxy the DTE policy can specify differing levels of trust for different proxies. To promote defense in depth, DTE firewalls propagate (or associate) the DTE security attributes between clients and servers, allowing DTE mechanisms on the endpoints to enforce consistent security rules for communicating programs. The two cases of greatest interest are when the server is running DTE and when the client is running DTE.

For all services except NFS, an incoming connection causes the DTE server to spawn the server process into the domain associated with the client by the DTE firewall. Because the server process is confined by the client's domain, the security of the server host does not rely on the robustness or security of the server process and the client's access remains unchanged even though using a network service. With the additional security added by DTE mediation, the server is confined sufficiently to be located inside the firewall security perimeter, removing the sacrificial lamb aspect of conventional servers. DTE also provides control for NFS-driven file operations on the server based on the client's domain.[13] Untrustworthy clients can be granted limited access to NFS hierarchies while preserving full access for more trustworthy clients.

For some services (e.g., HTTP), the server requires access to certain system files (e.g., the password file) to function normally. Because of these requirements, DTE cannot prevent the export of all

[13]DTE security attributes carried in UDP datagrams provide system-to-system authentication even for connectionless services; because each packet contains reliable DTE security attributes, connectionless services can be authenticated.

system-sensitive data. However, it can prevent the overwriting of this data and the export of sensitive corporate data. In general, the client's domain prevents the client from using the server to corrupt data or system files or to access sensitive information. Additionally, the DTE policy on the server can prevent unintentional access to files accidentally copied into exported NFS hierarchies (by controlling access to the file types.) The client's domain labels all data created for the client by the server (e.g., mail messages;) subsequent access to the data (which may be sensitive or of low integrity) on the server system therefore is controlled.

DTE firewalls increase safety for clients primarily by forcing security agreement between client and server domains, and by preventing undesired "crosstalk" between clients in different domains. For non-DTE client systems, a DTE firewall associates a single DTE domain with an entire system. This domain establishes the security context for a system and therefore establishes the security context of the servers available to the client (but does not restrict which protocols can be used.) For non-DTE systems, this forces a choice for each system: will it process data of importance to the organization, or will it process data from unknown sources (e.g., Web sites, FTP archives?) Although a DTE firewall can be configured to allow both, doing so entrusts the security of the organization to the client's ability to resist possible attacks from outside.

In the case of DTE clients, the client system provides mechanisms strong enough to maintain separation, and therefore offers users greater flexibility with respect to security, and also with respect to what services can safely be consumed from outside. Using a DTE client, a user can choose different roles for different activities. For example, a user wishing to exchange electronic mail or surf unimpeded across the Internet can start a session in a domain that has access to the "unknown_type" data from the Internet but not to important corporate data or system-critical files. Browsers, Postscript viewers, and other programs that might be tricked by external entities are consequently unable to steal from or otherwise damage the organization. Similarly, a user can start a session that grants access to corporate data but not to the outside world. Typically, a user would employ a window system to run several environments simultaneously, and would use a DTE-constrained regrade facility to move data between the environments in a controlled manner. Due to DTE labeling of new data files, undesirable crosstalk between clients via intermediary files is also controlled. This feature can be used to prevent importation of executable programs, for example: a DTE policy can ensure that programs imported by one domain will not be executable by another.

5.1.2 Functionality

For importing services, functionality is rarely affected. In services such as rlogin and TELNET, when the client is a DTE system, user authentication can be automatically supplied by the client DTE system and the proxy can accept and use this authentication instead of requiring additional

authentication. Under some circumstances, this can increase usability. The DTE uids, however, must be set up in the firewall configuration; this adds a small amount of administrative overhead. Services such as HTTP and FTP can be made more widely available because the risk of malicious programs (e.g., applets) has been reduced via DTE. NFS requires slightly more administrative overhead, but becomes safely available – something not possible before. Outgoing mail service has the same functionality, unless the user attempts to send out a file inaccessible from the mailer's domain.

A greater increase in functionality can be found in the exporting of services. With the additional security of running a server in a domain restricted according to trust level of the client, the server no longer needs to be located outside the firewall. Instead it can be on a system behind the firewall, or even on the firewall itself. Furthermore, with the server starting in the domain of the client, it is feasible to grant access to different classes of information based on that domain – something very useful for HTTP and FTP. In this way, for example, an anonymous FTP server could regulate access to files without resorting to multiple servers or hidden files. Also, a single HTTP server could provide sensitive data to users in multiple domains without fear of having the data compromised or altered.

5.1.3 Compatibility

Each service can interoperate either with DTE or non-DTE systems; furthermore, the application-level proxies revert to standard FWTK behavior when run on a non-DTE kernel. The use of IP options to carry DTE information removes the need for changing any of the protocols. With the exception of the NFS server, which is kernel-resident in UNIX[14], few of the client or server applications have been changed to function with DTE firewalls: mail final delivery agents were modified to be cognizant of the different types associated with a user's mailboxes; the rlogin server was modified to take advantage of DTE authentication mechanisms.

Some of the services running under DTE require changes in the administrative configuration. For example, external NFS clients must explicitly name the firewall host as the server whose file systems they wish to mount, since they cannot know the name of the server behind the firewall.[15] As another example, the DTE-protected uids must be specified on the firewall for use with non-DTE systems.

[14]UNIX is a trademark of the X/OPEN companies Ltd.

[15]Note, however, that the firewall does not itself run the NFS server.

5.1.4 Performance

Initial data throughput performance tests included in the phase one measurement and evaluation report showed moderate DTE performance overheads for the remote login and TELNET services when used with DTE. For the NFS, HTTP, and FTP services, they showed a significant reduction in performance. This reduction was investigated and several DTE network performance optimizations were found. The optimizations consisted of modifications to DTE diagnostic logging in the low level IP output processing routine, and, more importantly, a modification to reduce the fragmentation of network packets caused by adding DTE options to IP header information. Although we assume more optimizations can be found to enhance DTE network performance, informal test results conducted using *netperf*, a public domain network performance benchmark,[16] showed these implemented optimizations considerably improved DTE network performance. As a result, the phase one performance tests were re-implemented shortly after the release of the phase one report, and the improved performance statistics were published in a revised report[5].

To re-evaluate the performance of DTE and DTE Firewalls, we constructed a testbed consisting of three Pentium[17] 166MHz machines on an isolated Ethernet running BSD/OS 2.0 and version "straw_19+" of the DTE prototype system.[18] We ran each test on a number of configurations where configuration is a triple (client, firewall, server) in which "y" indicates a system running DTE and "n" indicates a host not running DTE (so (n,y,n) is the configuration where only the firewall is running DTE). Performance of mail was not measured since it is not interactive.

For rlogin, TELNET, and FTP, we used an Expect script to repeatedly perform the following steps.

1. First, invoke the client application, specifying the firewall as the destination.
2. Next, authenticate the user on the firewall.[19]
3. Then, connect to the server, again authenticating the user.
4. Transfer a variable amount of data from the server through the firewall to the client, ranging from 0 bytes transferred to 5 MB (except FTP, for which we tested 32 MB transfers).
5. Finally, log off the server, also terminating the connection with the firewall.

[16] Netperf is copyrighted by the Hewlett-Packard company.

[17] Pentium is a trademark of the Intel corporation.

[18] This is the 19th internal version of the BSD/OS-based DTE prototype evaluation with some performance enhancements incorporated.

[19] When coming from a DTE client, rlogin and TELNET authentication is performed automatically, using the DTE-protected uid.

Performance numbers were calculated by averaging results from 20 iterations of each test.

For HTTP, we used *ZeusBench*,[20] a standard benchmark. *ZeusBench* connected to the server via the HTTP gateway, retrieved a specified web page, and closed the connection; the gateway required no authentication. We varied the concurrency, the document length, and the number of requests (1000 requests for 1K documents, and, to save time, 32 requests for 50K documents).

Protocol		Data Transferred	Baseline (n,n,n)	Percentage Change (%) (n,y,n)	(n,y,y)	(y,y,n)	(y,y,y)
rlogin		0K	1.98	0	0	-15	-10
		200K	3.43	6	8	-16	-23
		500K	5.33	<1	<1	-14	-16
		5MB	32.21	<1	<1	-2	-2
TELNET		0K	6.79	<1	<1	-15	-12
		200K	7.79	<1	2	-10	-10
		500K	9.89	1	1	-9	-7
		5MB	41.36	<1	1	-2	-1
FTP		0K	1.98	9	13	6	11
		200K	3.98	4	6	4	5
		500K	4.59	3	4	3	4
		5MB	14.23	3	6	<1	3
		32MB	70.33	2	2	-1	<1
HTTP	Concurrency Level 4	1K	355.81	8	13	12	15
		50K	64.71	71	89	92	94
	Concurrency Level 8	1K	199.20	22	24	26	25
		50K	60.23	75	94	97	114

Table 1: Raw Performance in Seconds and DTE Overheads

As shown in table 1, DTE overheads for rlogin, TELNET, and FTP are modest, with a maximal impact of 13% in the worst case. With additional performance optimization, these could probably be reduced. As is shown in the table, performance actually increases for rlogin and TELNET when the client is running DTE. This is because the DTE client passes a DTE uid which the firewall can accept instead of performing costly authentication. This performance increase does not manifest for FTP because the FTP daemon has its own (always invoked) built-in authentication which we did not disable.

[20] ZeusBench version 1.0 is copyright Zeus Technology Limited 1996.

Unlike rlogin, FTP, and TELNET, the HTTP service is approximately 50% slower in the worst case. After analysis, we believe this performance decline for HTTP is somewhat artificial, resulting from the low-performance implementation of the HTTP application gateway in the FWTK, which does a separate *read()* and *write()* system call for each byte transfered. This overstates DTE system call overheads because, in this test, the system spends most of its time dispatching system calls that each do very little work but incur the full burden of access control checking. More recent application gateways, such as Gauntlet's,[21] perform more efficient I/O; we expect that DTE performance for those gateways should approximate the DTE performance for rlogin, TELNET, and FTP.

For NFS, we used Iozone and NFSstones, two widely-used NFS benchmark packages. The Iozone package tests sequential file I/O by writing a 64 MB sequential file in 8,192 byte chunks, then rewinds it, and reads it back (i.e., it measures the number of bytes per second that a system can read or write to a file). The size of the file was big enough to prevent the cache from dominating the results. NFSstones creates and deletes many directories, then does a variety of file accesses, including writes, sequential reads, and non-sequential reads. Using these results in a formula, it generates a single numeric indicator of relative NFS performance.

		Baseline	Percentage Change (%)	
		(n,n,n)	(n,y,n)	(n,y,y)
Iozone	Bytes/Second Written	107,372.50	4	20
	Bytes/Second Read	409,430.50	28	38
NFSstones	NFSstones/Seconds	69.83	18	38

Table 2: NFS Test Results

As shown by the results in figure 2, performance of writes under NFS is moderately affected by the addition of DTE to the firewall; adding DTE to the server produces a higher impact on performance, with a 20% performance hit. Reads under NFS, however, dominate NFS performance, with a slowdown of 38% when both the firewall and the server are DTE hosts. However, neither the NFS application-level gateway nor the DTE-enhanced NFS server is optimized. Two possible locations for performance degradation in the NFS server are the double mediation necessary because of the primary/auxiliary domain combination and the manipulation of additional file handles needed for DTE mediation in NFS-mounted files.

[21] Gauntlet is a registered trademark of Trusted Information Systems, Inc.

5.2 Dynamic Policy Modules

This section applies the four evaluation criteria: functionality, safety, performance, and compatibility to the phase two prototype's support for dynamic policy modules. This support is described in section 2.

5.2.1 Functionality

The main contribution of dynamic policy module support to the phase two prototype is increased functionality. As described in section 2.1.2, dynamic policy modules provide administrators with an organized framework for managing policy change. Administrators can use dynamic policy modules to specify the policy governing new activities and trust relationships. They may add policy support for a new activity or trust relationship to a DTE kernel by loading the appropriate module. Similarly, they can remove the support by unloading the module. Administrators may load and unload modules as the kernel runs. The ability to dynamically reconfigure a kernel's policy as it runs allows administrators to add and remove policy support for trust relationships without requiring system down-time and the resulting disruption of service availability. This method of policy configuration is superior to the phase one method, which involved modifying a kernel's base policy description and then rebooting the kernel.

5.2.2 Safety

Dynamic policy module support increases the safety provided by the phase two prototype by making it easier for administrators to respond to changes in their organization's security policy. By loading and unloading modules, administrators can construct new trust relationships or terminate old ones as the kernel runs, without disrupting the policy governing existing trust relationships. The prototype's support for dynamic policy modules also includes safeguards to prevent dynamic policy modules from reducing the safety provided by the prototype. The prototype grants the ability to load and unload dynamic policy modules only to users with UNIX root privileges operating in domains which have the DTE **privload** privilege. In addition, the restrictions described in section 2.3 provide support for protecting base policy integrity from dynamic module misbehavior.

5.2.3 Performance

We anticipate that dynamic module loads and unloads will be relatively infrequent events. Still, a reasonable response time for these operations is a desirable goal. Each dynamic policy module load or unload modifies the DTE policy resident in kernel memory. Test results show that the

integration or deletion of a dynamic policy module's contents with a DTE kernel's running policy does not take significant time to process.

For our experiments, the dynamic policy modules used included a *null_m* module, consisting solely of the module's name, and modules *five_m*, *ten_m*, and *twenty_five_m*, consisting of five, ten, and twenty five domains and types, respectively. The *null_m* module was chosen to give a base time for comparison with other modules. Each of the other three modules' domain definitions included five entry points, access to five types, and *exec* access to five domains. Each non-null module also assumed a base type and a base domain and augmented the base domain with access to five types and *exec* access to five domains. The dynamic policy modules used in the testbed's firewalls (appendix B) are similar in size to the *twenty_five_m* module; the other testbed modules are smaller.

The experiment consisted of two trials. The first trial was conducted on a host which was enforcing the "minimal" base policy shown in appendix C. The second trial was conducted on a host which was enforcing the pre-alliance Strider host policy given in appendix A.2. The minimal policy only defined one domain and one type while the Strider host DTE policy included twenty-two domains and ten types. During both trials, each of the four test dynamic policy modules were subjected to one thousand timed load/unload cycles.

Table 3 gives the average number of seconds required to load and unload each dynamic policy module during both trials.[22] The tests were performed using a **csh** script (with standard scheduling priority) on a Pentium 100MHz/32MB machine. The dynamic policy module loading and unloading mechanisms have not been optimized for performance.

	Base Policy	
Dynamic Module	minimal	Strider host
null_m	.0433	.0501
five_m	.0957	.1054
ten_m	.1506	.1639
twenty_five_m	.3592	.3841

Table 3: Average Time to Load and Unload a Single Module in Seconds.

[22]The results reflect the time for modification of the DTE policy resident in kernel memory only. If processes were running in the domains defined by the unloaded module, or if files existed which were associated with the module's types, additional processing time would be required for the kernel to kill the processes and regrade or delete the files.

5.2.4 Compatibility

Since the phase two prototype's dynamic policy module support does not modify the protocols it uses to interact with other systems, its impact on compatibility is minimal. It does add a number of new system features and behaviors, however. The phase two prototype provides system administrators with two new utility programs, **dtload** and **dtunload**, for dynamic policy module management. As described in section 2.1.3, the kernel will execute a purging algorithm whenever it unloads a module. This algorithm may cause the kernel to kill processes and delete files which were subject to the policy described by the module.

Dynamic policy modules themselves are written in the same DTEL policy specification language as the phase one base policy description. Although some statements which are legal in base policy specifications are not legal in dynamic module specifications, it is possible to convert most base policy modules to dynamic policy modules without wholesale revision.

5.3 Encryption

This section applies the four evaluation criteria: safety, functionality, performance, and compatibility to the phase two prototype's new IP-level encryption features. These features are described in section 3.

5.3.1 Safety

The phase two prototype's new IP-level encryption features increase the safety it provides to network applications. As described in section 3, the phase two prototype cryptographically protects the confidentiality and integrity of data traveling between DTE firewall-protected enclaves over public networks. The quality of this protection depends upon the strength of the cryptographic algorithm and the strength and secrecy of the keys used[10]. (Currently, the phase two prototype uses the DES algorithm in CBC mode with 56-bit keys.) This data protection adds to the safety the phase two prototype provides to all of its network services, and removes the need for the phase one assumption of secure communications.

5.3.2 Functionality

The new IP-level cryptographic protection also adds to the functionality of the phase two prototype by making support for enterprise zones possible. Without its facility for IP-level encryption, the

phase two prototype would be unable to securely link the distributed parts of an enterprise zone across public networks.

5.3.3 Performance

The phase two prototype's IP-level encryption support reduces the throughput performance of network services. Table 4 shows the results of five TCP/IP throughput performance tests designed to compare the phase two prototype's performance to the performance of other kernels. The results are graphed in figure 8. Each test involved a pair of kernels, running on identical 100MHz/32MB Pentium machines, communicating over an isolated network. The actual kernels used in each test were as follows:

BSD to BSD (non-encrypted): The first test measured the throughput between two BSD 2.1 kernels. The results of this test provided a baseline against which the results of the other test could be compared.

BSD IPSec to BSD IPSec (non-encrypted): The second test measured the throughput between two modified BSD 2.1 kernels which incorporated NRL's IPv6/IPSec code. During this test, the IPSec encryption features were not used.

DTE to DTE (non-encrypted): The third test measured the throughput between two phase two DTE kernels. During this test, the network traffic between the two kernels was not encrypted.

BSD IPSec to BSD IPSec (encrypted): The fourth test measured the throughput between two modified BSD 2.1 kernels which incorporated NRL's IPv6/IPSec code. During this test, the kernels encrypted their communication.

DTE to DTE (encrypted): The fifth test measured the throughput between two phase two DTE kernels using encryption.

Each test was conducted using the netperf benchmarking tool, which measured TCP throughput performance in a series of 8 trials. Each trial used a different message size, ranging from 128 bytes to 32768 bytes. In all trials, the send and receive sockets were given 8192-byte buffer capacities. The test results are correct within $^{+}/_{-}$ 2.5%.

The results illustrate a number of interesting points. For instance, the unmodified BSD kernel performance is worse than both the encrypted and non-encrypted performance of the modified BSD IPSec kernel for messages of 512 bytes or less. This is probably due to the modified kernel's NRL

Message Size (bytes)	Throughput (10^6bits/second)				
	BSD to BSD (non-encrypted)	BSD IPSec to BSD IPSec (non-encrypted)	DTE to DTE (non-encrypted)	BSD IPSec to BSD IPSec (encrypted)	DTE to DTE (encrypted)
128	0.02	2.53	0.07	1.23	0.07
256	0.07	3.08	0.08	1.54	0.08
512	0.10	5.92	4.44	2.90	1.60
1024	7.73	7.13	7.60	3.04	2.38
2048	7.82	7.18	7.27	3.09	2.43
4096	7.84	7.12	7.25	3.11	2.45
8192	7.85	7.13	7.26	3.10	2.39
32768	7.84	7.15	7.26	3.08	2.45

Table 4: TCP/IP Throughput Performance Test Results

IPSec implementation, which is based on the 4.4BSD Lite-2 kernel's IP stack. This implementation appears to be more efficient for small messages than the unmodified BSD 2.1 IP stack.

Most importantly, the results show that the phase two DTE prototype's throughput performance with encryption lags behind the BSD IPSec kernel's by only a small margin. With optimization, the DTE prototype's throughput with encryption might become more competitive with the performance of the BSD IPSec kernel.

5.3.4 Compatibility

As validated by our experiments, the phase two DTE kernel is fully-compatible with non-DTE kernels which use IPSec with the DES-CBC transform. The phase two kernel also remains compatible with all IPv4-based non-DTE non-IPSec kernels, although their lack of IPSec support prevents DTE kernels from cryptographically protecting any communication with them. This limitation requires hosts with DTE kernels to avoid the use of cryptography when using services based on non-DTE non-IPSec hosts.

The Distributed Name Service (DNS) provides a good example of this limitation on the use of cryptography. In order for a DTE host to make use of a DNS server based on a non-DTE non-IPSec host, its policy must not bind a key to the default output type of any domain that contains processes which must query the DNS server. If the DTE host's policy binds a key to one of

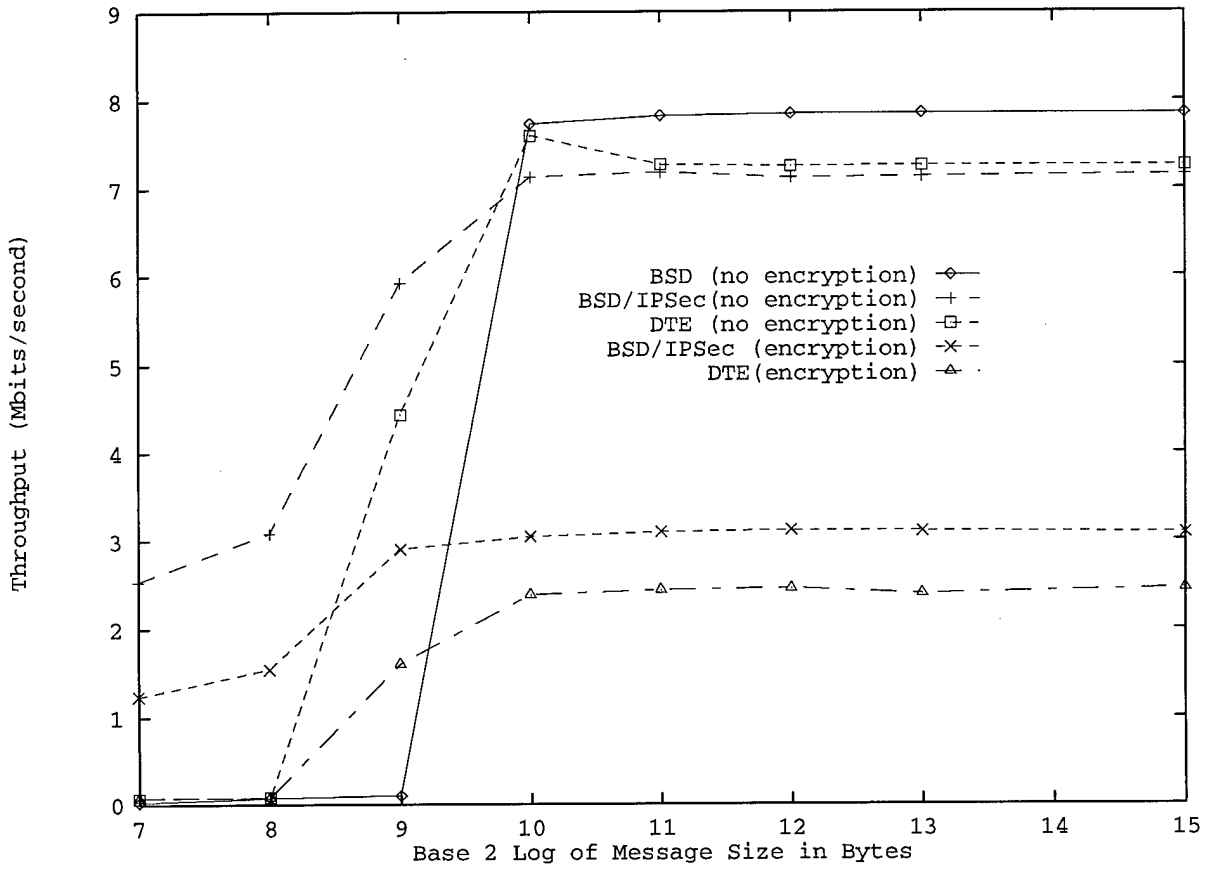

Figure 8: Throughput Performance

these default output types, its kernel will encrypt the DNS requests of that type and render them unintelligible to the server. The DTE host's policy must also avoid binding a key to the default output type of the domain it assigns to the DNS server's host. If the policy binds a key to this type, the DTE host will expect the DNS server's replies to be encrypted. Since the DNS server's host cannot encrypt these replies, the DTE host will discard them all.

6 Work in Progress

The main thrust of the third phase of the project is directed toward the development of the Domain Type Authority (DTA). DTA development will require the support of several related efforts. These efforts include the addition of automated configuration management features into the TIS Firewall Toolkit (section 6.2), the development of module parameterization techniques (section 6.3), and the introduction of configurable kernel meta-policy mechanisms (section 6.4). The DTA itself is briefly described in section 6.1. Section 6.5 examines the possibility of reducing the dependency restrictions on dynamic policy modules. Although it is not directly related to DTA development, this improvement has the potential to increase the functionality of the prototype.

6.1 Domain Type Authority

The DTA is intended to be a globally available, fault-tolerant, distributed, trusted service that distributes dynamic policy modules to DTE firewalls which need to establish distributed trust relationships. It would allow network application clients and servers to automatically establish safe expectations and limitations on their interactions. Figure 9 shows how the DTA is intended to operate. When a client application attempts to establish communication with a remote service for the first time, its DTE firewall queries the DTA and either receives a dynamic policy module specific to that service or employs a default. In the diagram, the dashed arrows represent both the requests made by each DTE firewall to the DTA for dynamic policy modules to govern their new trust relationship, and the path of the dynamic policy modules the DTA sends in reply. Once both enclaves have loaded the new dynamic policy modules, the client and the server may interact, as shown by the solid arrows.

6.2 TIS Firewall Toolkit Enhancement

The TIS Firewall Toolkit used by the DTE firewall prototype is designed only for manual reconfiguration. As a DTE firewall loads and unloads dynamic policy modules from a DTA, it will need to automatically reconfigure its application proxies. The third phase of the project will investigate the automatic reconfiguration problems posed by the various firewall and network service configuration files, including **/usr/local/etc/netperm-table** and **/etc/inet.conf**.

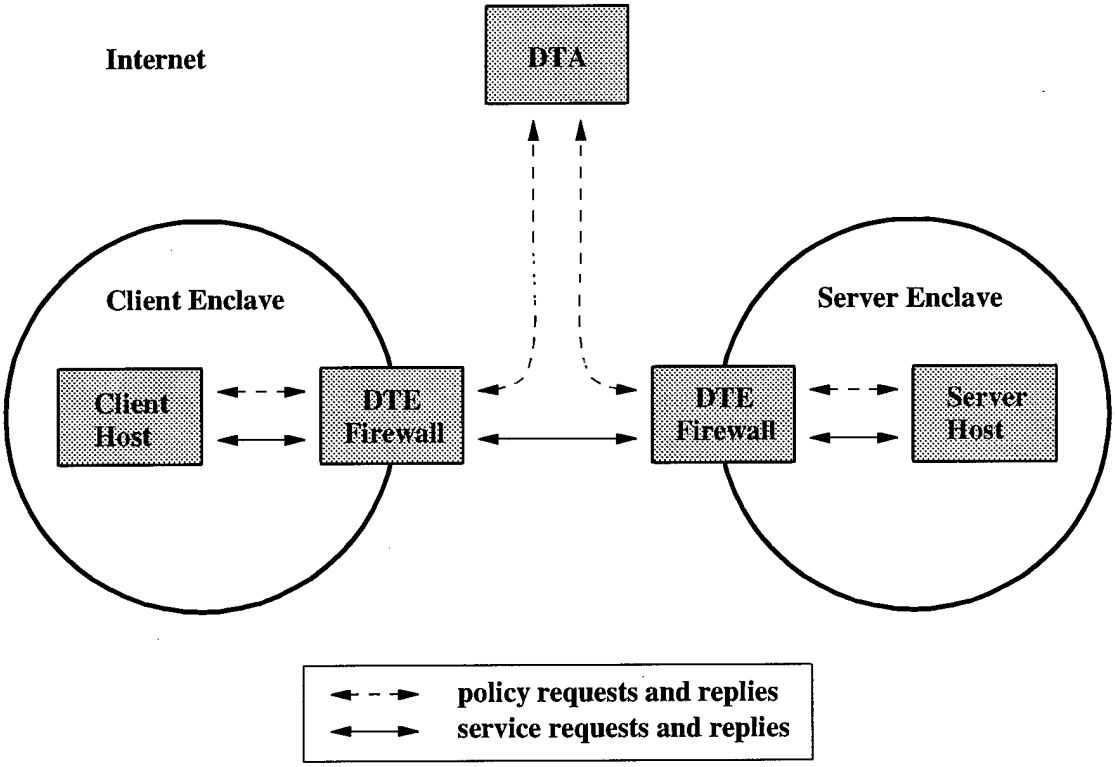

Figure 9: Distributed Domain Type Authority (DTA)

6.3 Dynamic Module Parameterization

As described in section 2, most dynamic policy modules require a certain amount of glue to integrate themselves with the base policy. The operation of the DTA must address this issue. The DTA is intended to serve dynamic policy modules to a wide variety of hosts. Since these hosts might have an equally wide variety of base policies, a given module must be capable of gluing itself into a variety of base policies. In order to achieve this goal, the third phase of the project will also investigate the possibility of adding parameterization to modules. Just as parameters to functions allow the same function to be called with different arguments, parameterization would allow policy modules to make use of different but operationally equivalent base policy domains and types.

6.4 Configurable Kernel Meta-policies

The third phase of the project will also investigate making the restrictions on dynamic module behavior described in section 2.3 configurable at kernel boot-time. Using a new DTEL "load point" mechanism we will investigate here, administrators might define one or more meta-policies which specify which restrictions apply in particular situations. For example, administrators might define a separate meta-policy for each DTA they use. Local trusted DTAs might be given generous meta-policies which allow their dynamic policy modules to modify base policies with few restrictions. Less-trusted remote DTAs might be given highly restrictive meta-policies which limit their dynamic modules to only the most harmless behaviors.

6.5 Relaxed Dependency Restrictions

A variety of additional development possibilities exist which are not directly related to the main DTA effort. The foremost among these is the possibility of relaxing the dependency restrictions described in section 2.2. As that section noted, these restrictions limit the kinds of useful modules a DTE kernel will accept. The scenario described in section 4 illustrates this point. The scenario defines the Strider/Cogs alliance in a single module that is loaded to begin the joint venture and not unloaded until it ends. As the project nears completion, a test engineer role might be desirable for validation testing. This role would almost certainly require some access to development engineer data. It would be convenient to add this role to the policy by loading a dynamic policy module which assumes the *gizmo_eng_t* type. Unfortunately, since this type is itself defined in a dynamic policy module, this dependency would not be allowed by the phase two prototype.

The phase three prototype may address this issue by adopting a more complex model of inter-module dependencies than the current base-part/dynamic-part arrangement. Dynamic policy modules might be allowed to assume types and domains from other dynamic policy modules. We will investigate whether the phase three prototype might arrange modules into directed acyclic graphs, where each node in the hierarchy depends on the types and/or domains of its parents. All of the nodes would then ultimately depend on the root of the graph, which would correspond to the base policy. Development in this direction must proceed cautiously, however, since the complexity introduced by this scheme might outweigh the benefits of its flexibility.

References

[1] R. Atkinson, "Security Architecture for the Internet Protocol", RFC 1825, August 1995.

[2] R. Atkinson, "IP Authentication Header", RFC 1826, August 1995.

[3] R. Atkinson, "IP Encapsulating Security Payload (ESP)", RFC 1827, August 1995.

[4] L. Badger, K. Oostendorp, W. Morrison, K. Walker, C. Vance, D. Sherman, D. Sterne, "DTE Firewalls Initial Measurement and Evaluation Report," Trusted Information Systems, Inc. TIS Report #0632, September 26, 1996.

[5] L. Badger, K. Oostendorp, W. Morrison, K. Walker, C. Vance, D. Sherman, D. Sterne, "DTE Firewalls Initial Measurement and Evaluation Report," Trusted Information Systems, Inc. TIS Report #0632R.

[6] K. Oostendorp, L. Badger, C. Vance, W. Morrison, D. Sherman, D. Sterne, "Domain and Type Enforcement Firewalls." Trusted Information Systems, Inc. Submitted to the 1997 Annual Computer Security Applications Conference (ACSAC).

[7] L. Badger, D. Sterne, D. Sherman, K. Walker, "A Domain and Type Enforcement UNIX Prototype," Usenix Computing Systems Volume 9, Cambridge, MA, 1996.

[8] W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, p. 18, 1985.

[9] K. Djahandari, W. Morrison, L. Badger, T. Fraser, T. Horvath, K. Oostendorp, M. Petkac, D. Sherman, C. Vance, "DTEL User's Guide," Trusted Information Systems, Inc. TIS Report #0659D.

[10] P. Karn, P. Metzger, W. Simpson, "The ESP DES-CBC Transform", RFC 1829, August 1995.

[11] P. Metzger, W. Simpson, "IP Authentication using Keyed MD5", RFC 1828, August 1995.

[12] W. Morrison, T. Fraser, L. Badger, K. Oostendorp, K. Djahandari, T. Horvath, M. Petkac, D. Sherman, C. Vance, "Distributed Type-Enforcing Firewall Architecture," Trusted Information Systems, Inc. TIS Report #0667. April 30, 1997.

[13] K. Walker, D. Sterne, L. Badger, M. Petkac, D. Sherman, K. Oostendorp, "Confining Root Programs with Domain and Type Enforcement.", Proceedings of the 6th Usenix Security Symposium, San Jose, CA, 1996.

A Strider Base Policies

A.1 Firewall Policy

The following policy was used as the base firewall policy for the Strider Corporation in Phase 2 development. As discussed in subsection 4.1, it is primarily a subset of the Phase 1 policy.

```
#define SHELLS  (/bin/{sh,csh,tcsh}, /usr/contrib/bin/tcsh)
/*-----------------------------------------------------------------------*
 * Common component for FW and host.  Provides initial domains and an
 * administrative login.  The comm_c component holds access to
 * communications types thay may be sent to us; it is extended below.
 */
policy base_p;
        type    unix_t, anon_t;
        component comm_c = (rd->unix_t);
        component user_c = comm_c;
        domain anon_d    = (crw->anon_t), user_c;
        domain daemon_d  = (/sbin/init), (crwxd->unix_t), (auto->login_d),
                           comm_c, privswapon;
        domain login_d   = (/usr/bin/login), (crwxd->unix_t), privauth,
                           (exec->admin_d);
        domain admin_d   = SHELLS, (crwxd->unix_t), user_c,
                           privlog, privdtmod, privswapon,
                           privencrypt, privreboot, privload,
                           (sigkill,sighup,sigtstp -> daemon_d);
        initial_domain   = daemon_d;
        mount  (/dev/sd0a, /), (/dev/sd0h, /usr), (/dev/sd0g, /usr/home);
        inet_assign anon_d  0.0.0.0;
        assign -r       unix_t          /;

/*-----------------------------------------------------------------------*
 * Common component for FW and host.  Defines domains/types required by
 * the DTE kernel and also specifies what systems are running DTE.
 */
module required_m;
        assumes type    unix_t, nfs_t;
```

```
        assumes domain  daemon_d, comm_c;
        domain  daemon_d += (auto->nfsd_d, nfsio_d);
        component nfs_c   = (cr->nfs_t), (rwd->unix_t), comm_c;
        domain nfsd_d     = (/sbin/nfsd), nfs_c;
        domain nfsio_d    = (/sbin/nfsiod), nfs_c;
        dte_systems (stridHost, 11.22.33.2), (stridFw, 11.22.33.1);

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines a domain to switch proxies into
 * separate domains.
 */
module switch_m;
        assumes type   unix_t;
        assumes domain daemon_d, admin_d, login_d, comm_c;
        type    fw_t, bin_t, fw_auth_t;
        domain  daemon_d += (rx->bin_t, fw_t),  (auto->fw_d);
        domain  admin_d  += (rwxd->fw_t, bin_t),
                            (sigkill,sighup, sigtstp->fw_d);
        domain  login_d  += (rx->bin_t),  (r->fw_t);
        domain  comm_c   += (r->fw_t);
        domain  fw_comm_c;
        domain  fw_d      = (/usr/local/etc/{netacl, authsrv}), (crwxd->fw_t),
                            (rd->unix_t), (rx->bin_t), (rw->fw_auth_t),
                            fw_comm_c;
        assign  -s      bin_t      /bin/cat;
        assign  -s      fw_auth_t  /usr/local/etc/{fw-authdb,fw-authdb.db};

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines user domain names and types to be
 * passed through the firewall proxies, and associates user domains with
 * non-DTE systems (which must have default output types).
 */
module talkto_hosts_m;
        assumes type   anon_t;
        assumes domain comm_c, fw_comm_c, anon_d, daemon_d;
        type    strid_t;
        component fw_comm_c += (rw->anon_t, strid_t), (srcd->anon_d, strid_d);
        domain  dtacl_d;
        domain  daemon_d    += (r->strid_t, anon_t);
```

```
        domain  strid_d      = (crw->strid_t), comm_c;
        inet_assign     strid_d    11.22.33.3;

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines the rlogin/TELNET proxy domain.
 */
module fw_rlogin_m;
        assumes type   fw_t, unix_t;
        assumes domain fw_comm_c, fw_d;
        domain  fw_d       += (auto->fw_rlogin_d);
        domain  fw_rlogin_d = (/usr/local/etc/{rlogin-gw, tn-gw}),
                              (crwd->fw_t), (rd->unix_t), privauth, fw_comm_c;

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines the ftp proxy domain.
 */
module fw_ftp_m;
        assumes type   fw_t, unix_t;
        assumes domain fw_comm_c, fw_d;
        domain  fw_d     += (auto->fw_ftp_d);
        domain  fw_ftp_d  = (/usr/local/etc/ftp-gw),
                            (crw->fw_t),  (rd->unix_t), privauth, fw_comm_c;

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines the NFS mount daemon proxy domain.
 * The portmap_d needs to reply back to any client in the type the client
 * sent.
 */
module fw_nfs_m;
        assumes type   fw_t, unix_t, nfs_t;
        assumes domain fw_comm_c, daemon_d, admin_d, nfsio_d, nfsd_d;
        domain  daemon_d  += (auto->fw_mount_d, fw_nfs_d, portmap_d),(r->nfs_t);
        domain portmap_d  = (/usr/sbin/portmap), fw_comm_c,
                              (rwd->fw_t), (crwd->unix_t), (srcd->admin_d);
        domain fw_mount_d = (/usr/local/etc/stirrup), (rwd->fw_t),
                              (crwd->unix_t), fw_comm_c, (srcd->admin_d);
        domain fw_nfs_d   = (/usr/local/etc/nfs-gw), (rwd->fw_t),
                              (cr->nfs_t), (rwd->unix_t), fw_comm_c,
                              (srcd->admin_d, nfsd_d, nfsio_d);
```

```
/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines the http proxy domain.
 */
module fw_http_m;
        assumes type   fw_t, unix_t;
        assumes domain fw_comm_c, fw_d;
        domain  fw_d     += (auto->fw_http_d);
        domain  fw_http_d = (/usr/local/etc/http-gw), (crwd->fw_t),
                            (rd->unix_t), privauth, fw_comm_c;

/*-----------------------------------------------------------------------*
 * Used only on the firewall.  Defines the mail domains.  Smapd execs
 * sendmail.
 */
module fw_mail_m;
        assumes type   fw_t, unix_t;
        assumes domain daemon_d, anon_d,strid_d, user_c;
        type    tmp_t,spool_t;
        component user_c += (d->unix_t); /* sendmail needs shared libs */
        domain  daemon_d += (rw->tmp_t), (auto->mail_d);
        domain  admin_d  += (rw->tmp_t);
        domain  anon_d   += (/usr/sbin/sendmail), (rw->tmp_t), (rdm->spool_t);
        domain  strid_d  += (/usr/sbin/sendmail), (rw->tmp_t), (rdm->spool_t);
        domain  mail_d    = (/usr/local/etc/{smapd,smap},/usr/sbin/sendmail,
                            /usr/libexec/mail.local),
                            (cr->fw_t), (rmd->spool_t),(rd->unix_t),
                            (r->anon_t, strid_t),
                            (exec->anon_d, strid_d);
        assign  -s      tmp_t      /dev/null;
        assign          spool_t    /usr/var/spool/{mqueue,smap};
```

A.2 Host Policy

The following policy was used as the base host policy for the Strider Corporation in Phase 2 development. As discussed in subsection 4.1, it is primarily a subset of the Phase 1 policy.

```
#define SHELLS  (/bin/{sh,csh,tcsh}, /usr/contrib/bin/tcsh)
#define PROGS   SHELLS, (/usr/libexec/{rlogind, telnetd, ftpd}, \
                         /usr/contrib/bin/httpd)
/*--------------------------------------------------------------------*
 * Common component for FW and host.  Provides initial domains and an
 * administrative login.
 */
policy  base_p;
        type   unix_t, anon_t;
        component comm_c =  (r->unix_t);
        component user_c = comm_c;
        domain anon_d    = (crw->anon_t), user_c;
        domain daemon_d  = (/sbin/init), (crwxd->unix_t), (auto->login_d),
                           comm_c, privswapon;
        domain login_d   = (/usr/bin/login), (crwxd->unix_t), privauth,
                           (exec->admin_d);
        domain admin_d   = SHELLS, (crwxd->unix_t), user_c,
                           privlog, privdtmod, privswapon,
                           privencrypt, privreboot, privload,
                           (sigkill,sighup,sigtstp -> daemon_d);
        initial_domain   = daemon_d;
        mount (/dev/sd0a, /), (/dev/sd0h, /usr), (/dev/sd0g, /usr/home);
        inet_assign    anon_d    0.0.0.0;
        assign -r      unix_t    /;

/*--------------------------------------------------------------------*
 * Common component for FW and host.  Defines domains/types required by
 * the DTE kernel and also specifies what systems are running DTE.
 */
module required_m;
        assumes type   unix_t;
        assumes domain admin_d, daemon_d, comm_c;
```

```
        type    nfs_t;
        domain  admin_d  += (rw->nfs_t);
        domain  daemon_d += (auto->nfsd_d, nfsio_d), (r->nfs_t);
        component nfs_c   = (cw->nfs_t), (rwd->unix_t), comm_c;
        domain  nfsd_d    = (/sbin/nfsd),(cw->nfs_t), (rwd->unix_t), comm_c;
        domain  nfsio_d   = (/sbin/nfsiod), (cw->nfs_t), (rwd->unix_t),
                            comm_c;
        dte_systems (stridHost, 11.22.33.2), (stridFw, 11.22.33.1);

/*-------------------------------------------------------------------*
 * Used only on hosts.  Defines types and domain names needed to communicate
 * with the firewall.
 */
module talkto_fw_m;
        assumes domain comm_c;
        type    fw_t, bin_t;
        component comm_c += (r->fw_t);
        domain fw_d, fw_ftp_d, fw_http_d, fw_rlogin_d,
                mail_d, fw_mount_d, fw_nfs_d;

/*-------------------------------------------------------------------*
 * Used only on hosts.  Define shared /tmp and pty access for user domains.
 * Defines a restricted regrade domain.  More root confinement would happen
 * here.
 */
module host_m;
        assumes type   unix_t;
        assumes domain admin_d, login_d, daemon_d, user_c;
        type    tmp_t, term_t, http_t;
        domain  login_d  += (exec->regrade_d);
        domain  admin_d  += (rwd->http_t);
        domain  daemon_d += (rdm->tmp_t),(rw->term_t),(auto->dtacl_d,portmap_d);
        component user_c += (rdx->unix_t), (rdm->tmp_t), (rw->term_t),
                            (wd->http_t);
        domain portmap_d = (/usr/sbin/portmap,/sbin/mountd), (crwd->unix_t);
        domain dtacl_d   = (/usr/bin/dtacl), login_d, (crwxd->unix_t), privauth;
        domain regrade_d = SHELLS, (/bin/{ls,cp,mkdir,rm}, /usr/bin/dti),
                           (crwd->unix_t);
        assign -r      tmp_t       /tmp;
```

```
        assign -r       tmp_t       /usr/var/tmp;
        assign -s       term_t      /dev/null;
        assign -u       http_t      /usr/var/log/httpd;
        assign          term_t      /dev/{ttyp0,ttyp1,ttyp2,ttyp3,ttyp4,
                                         ptyp0,ptyp1,ptyp2,ptyp3,ptyp4};

/*-----------------------------------------------------------------------*
 * Used only on hosts.  Define user domains to confine network applications.
 * For client-mode, run in a user session; for server-mode, activate from
 * dtacl.
 */
module user_domains_m;
        assumes type   anon_t, nfs_t;
        assumes domain admin_d, regrade_d, dtacl_d,login_d, user_c,
                       anon_d, portmap_d;
        type    strid_t;
        /* plug into rest of policy */
        domain  admin_d   += (rwd->strid_t, anon_t),
                             (sigkill,sighup,sigtstp->strid_d,anon_d);
        domain  dtacl_d   += (r->strid_t, anon_t), (exec->strid_d, anon_d);
        domain  regrade_d += (rwd->strid_t, anon_t);
        domain  login_d   += (exec->strid_d, anon_d);
        domain  portmap_d += (rw->strid_t, anon_t, nfs_t),
                             (srcd->strid_d, anon_d);

        /* user execution environments */
        domain anon_d    += PROGS, (dx->anon_t);
        domain strid_d    = PROGS, (crwdx->strid_t), user_c, privauth;

        assign  -r      anon_t          /usr/home/anon, /www/docs/public;
        assign  -r      strid_t         /usr/home/strid, /www/docs/private;

module  dns_m;
        assumes type unix_t, anon_t, strid_t;
        domain  dns_d = (r->anon_t, strid_t, nfs_t), (crw->unix_t);
        inet_assign dns_d     11.22.33.99;

module mail_m;
        assumes type   anon_t, strid_t, tmp_t;
```

```
assumes domain anon_d, strid_d, admin_d;
type    spool_t;
domain  admin_d  += (rdm->spool_t);
domain  anon_d   += (/usr/sbin/sendmail), (rdm->spool_t);
domain  strid_d  += (/usr/sbin/sendmail), (rdm->spool_t);
domain  daemon_d += (/usr/sbin/sendmail), (rdm->spool_t);
assign          spool_t         /usr/var/spool/mqueue;
assign          spool_t         /usr/var/mail;
```

B Strider Dynamic Modules

A common need for all dynamic modules is to reference the DTE base policy types and domains. Dynamic modules may reference base types and domains by using the DTEL *assumes type* and *assumes domain* statements. Both host and firewall dynamic policies contain the *assume* statements. Each also declares the new data types, *gizmo_budget_t*, *gizmo_rates_t*, and *gizmo_eng_t* and an associated key for encryption of all Gizmo generated IP messages.[23] The encryption of IP messages is a vital requirement in our notion of an enterprise zone.

B.1 Firewall Dynamic Module

The base policy of the firewall includes a base module, *talkto_host_m*, that defines user elements to be passed through the firewall by proxy applications, a base module, *switch_m*, that creates an execution environment (*fw_d*) to start the proxies in their own domain, and base modules for each firewall proxy. These base modules require augmentation by the firewall dynamic module to support the enterprise zone.

The firewall policy must identify the Donald's firewall in a DTEL *dte_systems* statement to ensure all proper DTE header information is included in IP message exchange. The *gizmo_proj_d*, *gizmo_acct_d*, and *gizmo_eng_d* domain names must be defined for the firewall to recognize Gizmo client domains and properly process requests and responses. The firewall's **inetd** daemon automatically transitions to the *fw_d* domain to activate the firewall toolkit's **netacl** program. The **netacl** program reads data from the client socket requiring read access to each of the Gizmo data types. The **netacl** program automatically transitions the domain of the requested proxy to start the firewall proxy program. In order to pass the data type through the firewall, the firewall proxies for the FTP, RLOGIN, TELNET, NFS, and HTTP services must be able to read and write the data type. Additionally, in order to relay the source domain of received messages, each of these proxies must have *srcd* access to the source domains they receive. The HTTP proxy *fw_http_d* domain and the NFS proxy *fw_nfs_d* domain, which may be accessed by each of the three Gizmo domains, have read and write access to all three Gizmo data types as well as *srcd* access to each Gizmo domain. However, the proxy domains *fw_rlogin_d* for processing of RLOGIN and TELNET and *fw_ftp_d* for FTP request processing, only have read and write access to the *gizmo_budget_t* and *srcd* access to *gizmo_proj_d* since only the project leader role has **FTP**, **RLOGIN**, and **TELNET** access. The firewall's MAIL proxy is unique from the other proxies and requires read access to each of the Gizmo types and exec access to transition to each of the Gizmo domains. The **/usr/sbin/sendmail** entry point and accesses for the *gizmo_proj_d*, *gizmo_acct_d*, and *gizmo_eng_d* domains are also necessary

[23] If desired, each Gizmo data type could be assigned a unique key.

for **MAIL** processing. Some of the Gizmo user domain accesses would also be necessary if a DTEL *inet_assign* statement specifying a Gizmo domain was included in the dynamic policy.

The following module was used as the dynamic firewall module for the Strider Corporation in Phase 2 development.

```
module fw_gizmo_m;

        assumes type   fw_t, spool_t, tmp_t, unix_t;
        assumes domain daemon_d, fw_d, fw_ftp_d, fw_http_d, fw_nfs_d,
                       fw_rlogin_d, mail_d, comm_c;

        type gizmo_budget_t, gizmo_rates_t, gizmo_eng_t;

        component mail_c    = (/usr/sbin/sendmail), (rw->tmp_t), (rdm->spool_t),
                              rd->unix_t), (r->fw_t);

        /*
         * Domains corresponding to the base policy talkto_host_m module
         */
        domain gizmo_proj_d = (crw->gizmo_budget_t), mail_c;

        domain gizmo_acct_d = (crw->gizmo_rates_t), mail_c;

        domain gizmo_eng_d  = (crw->gizmo_eng_t), mail_c;

        /*
         * Domains corresponding to the base policy switch_m module
         */
        domain fw_d        += (r->gizmo_budget_t, gizmo_rates_t, gizmo_eng_t);

        /*
         * Domains corresponding to the base policy proxy modules
         */
        domain fw_ftp_d    += (rw->gizmo_budget_t), (srcd->gizmo_proj_d);
        domain fw_http_d   += (rw->gizmo_budget_t, gizmo_rates_t, gizmo_eng_t),
                               (srcd->gizmo_proj_d, gizmo_acct_d, gizmo_eng_d);
        domain fw_nfs_d    += (rw->gizmo_budget_t, gizmo_rates_t, gizmo_eng_t),
                               (srcd->gizmo_proj_d, gizmo_acct_d, gizmo_eng_d);
```

```
domain fw_rlogin_d += (rw->gizmo_budget_t), (srcd->gizmo_proj_d);

domain mail_d      += (r->gizmo_budget_t, gizmo_rates_t, gizmo_eng_t),
                      (exec->gizmo_proj_d, gizmo_acct_d, gizmo_eng_d);

dte_systems (donaldsFw, 22.33.44.1);

key esp gizmo_k = (0x0123456789abcdef 0x111122223333)->gizmo_budget_t,
                                                gizmo_rates_t, gizmo_eng_t;
```

B.2 Host Dynamic Module

The host dynamic module description closely follows the dynamic module specification described in section 4.2. The base policy's *user_domains_m* module is the primary module requiring augmentation. The *gizmo_proj_d*, *gizmo_acct_d*, and *gizmo_eng_d* domains define the user roles of the alliance. The *login_d* domain is augmented to allow transitions to each user domain for local host logins. Each of the Gizmo user domains contain the *gizmo_user_c* component giving the user logging into the local host machine necessary access to base policy data to accomplish normal tasks. The *gizmo_user_c* component is comprised of the base policy *comm_c* component and access to the *spool_t* type, used in mail processing. Each domain has full access to the data type associated with its domain (e.g. *gizmo_eng_t* for the *gizmo_eng_d* domain) and additional access to other Gizmo data types given in the specifications. Most of the alliance work is to be done via NFS and each organization's host is both an NFS server and an NFS client. The DTEL *mount* statement specifies the file system to be mounted. The NFS domains, *nfsd_d* and *nfsio_d*, are given read, write, and directory traversal access to all three Gizmo data types to perform NFS operations. The *dns_d* domain is augmented to read DNS requests for each Gizmo user domain. The augmentation of the *dtacl_d* domain, and the non-shell entry points of the Gizmo user domains ensure that exported services are started in the proper user domain. Regrade rules are also included for each Gizmo data type.

The following module was used as the dynamic host module for the Strider Corporation in Phase 2 development.

```
/*
 * The common entry point programs for all dynamic module defined domains
 */
#define SHELLS          (/bin/{sh,csh,tcsh}, /usr/contrib/bin/tcsh)
#define COMMON_PROGS    (/usr/contrib/bin/httpd, /usr/sbin/sendmail)

module host_gizmo_m;

        assumes type    fw_t, unix_t, tmp_t, term_t, http_t, spool_t, nfs_t;
        assumes domain  dns_d, dtacl_d, login_d, nfsd_d, nfsio_d, strid_d;

        type gizmo_rates_t, gizmo_budget_t, gizmo_eng_t;

        regrade (gizmo_rates_t->unix_t);
        regrade (gizmo_budget_t->unix_t);
        regrade (gizmo_eng_t->unix_t);
```

```
component gizmo_user_c = (rw->nfs_t, term_t), (rdx->unix_t),
                         (rdm->tmp_t, spool_t), (wd->http_t), (r->fw_t);

domain  gizmo_proj_d = SHELLS, COMMON_PROGS,
                       (/usr/libexec/{rlogind, telnetd, ftpd}),
                       (crwxd->gizmo_budget_t),
                       (rd->gizmo_rates_t, gizmo_eng_t),
                       gizmo_user_c;

domain  gizmo_acct_d = SHELLS, COMMON_PROGS,
                       (crwxd->gizmo_rates_t), (rd->gizmo_budget_t),
                       gizmo_user_c;

domain  gizmo_eng_d = SHELLS, COMMON_PROGS,
                      (crwxd->gizmo_eng_t),
                      gizmo_user_c;

domain  dns_d     += (r->gizmo_budget_t, gizmo_rates_t, gizmo_eng_t);
```

A9

New Text Document.txt

22 JANUARY 1998

POC: ROME LAB
COMPUTER SYSTEMS BRANCH
ROME, NY 13441-3625