Protected Data Paths: Delivering Sensitive Data via Untrusted Proxies

Jiantao Kong and Karsten Schwan
College of Computing
Georgia Institute of Technology
{jiantao,schwan}@cc.gatech.edu

Patrick Widener
Department of Computer Science
University of New Mexico
widener@cs.unm.edu

Abstract

The ability to share sensitive information is a key necessity for today's distributed enterprise applications. This paper presents a kernel-level mechanism for controlling the exchanges of sensitive data, termed Protected Data Paths. The mechanism permits only machines with suitable credentials to cache and manipulate protected data, and it gives protection domains access to such data only as per their rights specified in the capabilities they possess. Our implementation of Protected Data Paths in Linux operates by creating protected communication channels between participating machines. Path establishment requires such machines' kernel domains to have suitable credentials. Data transferred via such paths is made available to application-level domains only as per their current data access capabilities, guaranteed by kernel-level supervision of such data accesses.

1. Introduction

With today's highly connected computing systems and their web service-based programming infrastructures, it has become common for applications to be constructed as sets of components developed by different companies and run on machines owned and operated by multiple organizations and/or application participants. For instance, for efficient content delivery to large numbers of web clients, web servers routinely interact with backend machines via intermediate proxy caches, and additional data caching occurs in edge servers, to better deal with the heterogeneous nature of the Internet [1,2,3,4]. Further, internally, web service infrastructures like JBOSS [5] or Websphere [6] make extensive use of results and parameter caching on intermediate machines, to offload backends that often constitute the bottlenecks in multi-tier web service applications.

An issue with the extensive data caching and data sharing ongoing in distributed infrastructures is that these systems provide few or no guarantees to end users about the privacy of the data being exchanged. That is, when an application-level proxy caches data on its machine, end users are not protected from potential violations in data access by such third parties. The unfortunate outcomes are (1) a lack of control over data movement and thus, potential violations of data privacy and/or (2) the resulting inability for corporations or entities to use efficient web-based infrastructures for secure data exchanges. Consider, for example, a health information system that consists of applications owned by different departments. Such a system usually deploys a HL7 hub to manage the exchanging of patient information in HL7 message format [7]. Here, patient data must be shared, privacy must be maintained, and in addition, each department should only have access to certain portions of the data.

Unfortunately, for distributed applications with data privacy concerns, there now neither exists a way to maintain privacy if intermediate proxies are used to operate on such data like relaying or caching, nor is there a way to differentially protect certain data portions. Instead, applications that are involved in sensitive information exchanges are assumed to be fully trustworthy and thus have full access rights to data. In the health information system above, patient information is fully exposed to the HL7 hub applications. However, a malfunctioning HL7 hub may pass additional information to the target users or even disclose information to unauthorized parties. Such behavior clearly violates the principle of least privilege [8], which states that data should be accessible only to those parties that must have access to it.

There are many other examples of applications and systems for which stricter controls on data access would be desirable. Firewall or proxy applications, for instance, should not be able to access actual data content beyond what is needed for their tasks, but unfortunately, to attain desired high performance, their current implementations often violate this principle. Particularly obvious examples are the content delivery infrastructures mentioned earlier, which routinely use multiple server mirrors and multiple

layers of proxy caches to attain desired levels of performance.

This paper describes a novel set of operating system mechanisms that permit sensitive data to be exchanged across different machines so as to follow the principle of least privilege. Specifically, the *Protected Data Paths* (PDPs) mechanisms dynamically construct protected data delivery paths across the multiple machines used by a distributed application. Sensitive data is protected from inappropriate access by using different protection domains, the kernel domains and application domains, to transfer data vs. managing such data exchanges:

- By providing efficient interfaces between the protection domains that transfer data and the ones that manage it, data privacy is guaranteed without compromising the high performance attained by the rich management methods present in current distributed systems [2]. In a Linux-based implementation of a distributed data cache, the operating system kernel (i.e., the kernel domain) transfers and store sensitive data, and application processes are the domains that manage data caching.
- By controlling how and which portions of data a protection domain can access, data privacy concerns are met while also enabling diverse ways in which data is manipulated. In the Linux implementation of PDP-based caching, the kernel maintains credentials (i.e., access right information) for application-level processes, and all accesses to sensitive cached data by application processes engender kernel-level access checks. Once moved into the application domain, data may be manipulated in any way desired.
- By permitting only trusted machines and their OS kernels to participate in a PDP, the integrity protected data delivery is guaranteed.

As indicated above, the Linux implementation of PDPs uses a straightforward credential-based implementation of capabilities to restrict the data access rights of participating machines and also, to delimit the data access rights of all protection domains that manipulate the data being exchanged. The list of capabilities owned by a protection domain completely defines its rights to transfer and/or to manipulate sensitive data. Our future implementation of the concept will use multiple virtual machines running on virtualized execution platforms, thereby further isolating path-level data from the parties not permitted to directly access or manipulate it.

Protected Data Paths are designed to deal with both the threats of compromised credentials and data delivery paths. Ignoring how users are authorized, which is not the focus of this paper, the current PDP implementation addresses capability forgery by using digitally signed credentials. It controls capability reuse by associating a capability with a specific end user application (i.e., protection domain) in a specific location (i.e., on some machine). Finally, capability propagation is prevented by never handing out actual capabilities but instead, providing only capability references to end user applications.

The Linux implementation of PDPs uses a trusted kernel module to performs credential management and enforce capability-based data access control. Application domains that require access to protected data do so via intercepted calls to the kernel domain. No application module can access or use data other than what is permitted by its capabilities, even if it only acts as a forwarding agent. For example, an untrusted proxy cache application will receive only reference tokens to the actual data objects cached in the kernel domain. The application can use these tokens to control caching, including using them to service repeated requests for the same object, but it will not be permitted to access the object's contents or to change or transform it. Finally, our current implementation does not encrypt data, so that it is open to attacks like eavesdropping, message alteration, TCP hijacking etc. This is easily corrected, by inserting additional encryption and decryption actions into data exchanges across machines. The intercepted I/O channels explained in Section 4 provide a simple way to implement encryption.

The principal application benefiting from the use of Protected Data Paths evaluated in this paper performs static content serving and caching, using the Apache HTTP Server [3] along with the standard Squid [2] Web Proxy Cache. Data may be cached at the application level if no privacy concerns exist. Protected data is cached in the operating system kernel, using a kernel-level data cache also constructed in our research. Squid manages such kernel-level data through capabilities (i.e., protected data references), which, as stated earlier, are maintained by the kernel. Many other applications can benefit from the data privacy support offered by PDPs. One application currently being developed by our group is the aforementioned manipulation of patient data, by HL7 hub applications.

Experimental evaluations of the PDP concept and implementation demonstrate its key property: by using PDPs, the rich methods offered by web-based data transport and manipulation may be exploited while at the same time, gaining the ability to enforce application-specific data privacy constraints. For Apache, no effects are observed for request response times and throughput for files larger than 256K. For Squid, the PDP mechanisms results in small improvement in response time due to its use of the in-kernel data path. For smaller sized files, the additional credential-related operation in

the kernel result in small increases in response time and decreases in throughput. However, these costs can be amortized if the same PDP is used for multiple requests.

The next section compares Protected Data Paths with alternative methods. The detailed design of PDPs is presented in Section 3. Section 4 describes the implementation of Linux-based PDPs, and their usage with an Apache HTTP Server and a Squid Web Proxy Cache is elaborated in Section 5. Section 6 reports measurements of experimental results with micro-benchmarks, with Apache, and with Squid. Section 7 concludes the paper and reports on future directions.

2. Related work

Common ways protecting information in a distributed system are authentication [9,10,18,26] and encryption [20,21]. Here, a sender and a recipient authenticate each other using some handshake protocol, thereby establishing a secure communication channel. A session key is used to encrypt all future messages for security purpose. Examples include the HTTPS protocol on top of SSL [19] and SFTP on top of SSH [29] etc.

Methods like those described above are point-to-point because of their mutual-trust requirement. Even some secure group communication middleware or secure publish/subscribe systems [27,31] are built on top of point-to-point connections, where each connection is secured by the above mechanisms. In order to extend the data path to multiple nodes, for scalability, additional trusted nodes must be inserted into the data path, but finding such nodes may not be trivial. SSL tunneling [24] addresses this issue, but it only allows the Proxy to handle an SSL connection in transparent forwarding mode, thereby unable to perform other offloading tasks, such as data caching. In comparison to such work, PDPs provide a direct way of inserting into and then using additional trusted nodes with the distributed platforms used by large-scale applications. Furthermore, PDPs can deal with both the secure applications reviewed above and the large set of untrusted applications commonly used in the large-scale enterprise systems.

A fundamental point about PDPs is that essentially, this concept generalizes upon the extensive prior practice of delegating certain data transfer tasks to the operating system kernel, as with TCP Splicing [14,22], the sendfile system call, or kernel-level http servers. While prior work has focused on the performance advantages of such delegation, PDPs combine such performance goals with capability-based methods for controlling application access to ongoing data transfers.

While PDPs apply the capability model [8] to distributed systems, we do not innovate in the domains of authentication and authorization, like [16]. Instead, we simply adopt a framework that is sufficient to support the PDP idea. Further, while the PDP approach to protection is somewhat similar to the restricted delegation concept in Grid Computing [15,17], for restricted delegation, the crucial issue is to determine what access rights the end users should grant to certain intermediate nodes. This is necessary because the implementations of applications like Apache/Squid still retain full access rights to the data being transferred. In contrast, the PDP model separates each participating node into two parts: a trusted domain (e.g., the OS kernel) and the application domain. Users can give full access to the trusted domain (if desired), and at the same time, they can give the application domain only the limited access rights needed for data management or manipulation.

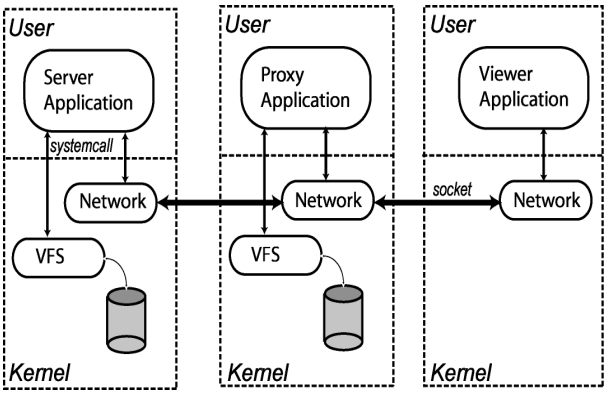

Figure 1. Data delivery path

3. Design of the data delivery model

This section presents the capability-based data delivery model used to realize Protected Data Paths. The model addresses potentially compromised capabilities and compromised data delivery paths. The model is designed to isolate the management of data exchanges from the exchanges themselves. It also carries out security checks via trusted modules in the system kernel.

3.1. Data delivery path

Information flows in distributed systems typically travel through multiple application components before reaching their destinations. Figure 1 illustrates a typical data delivery path involving a server application that provides the data, a proxy application running at a gateway, and a viewer application displaying data to an end user. This delivery path may be viewed as connecting multiple protection domains via inter-domain channels. For instance, in Figure 1, the server, proxy, and viewer

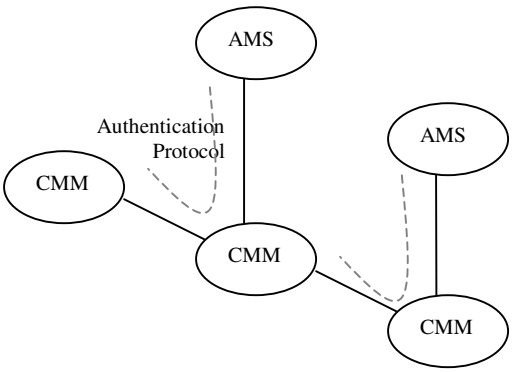

Figure 2a. CMM authentication

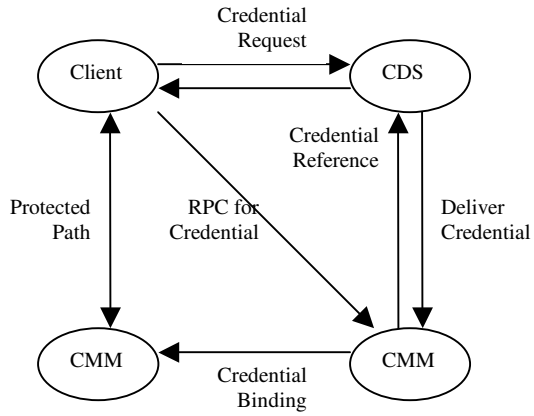

Figure 2b. Acquire credentials

applications are all different protection domains. Additional independent domains are those used by OS subsystems like the virtual file system and the network subsystem. All such domains exchange data with each other via system calls and socket connections.

We base data protection on the capability model originally proposed in Hydra [8]. Each protection domain owns a list of capabilities that describe its access rights to protected information. Data can be transferred from one domain to another only when the receiving side presents proper capabilities. Data access control is enforced on the inter-domain communication channel.

Given this model, four steps are involved in setting up a Protected Data Path. First, the client side application interacts with a Capability Distribution Server (CDS) on behalf of its end users. Using standard authorization and authentication procedures, the CDS generates proper credentials in response to such client requests. Second, the CDS stores the resulting credentials in the Credential Manager Module (CMM) resident in a trusted kernel domain while the client receives only credential references. Third, when the client side application establishes a connection to some server, it associates with this connection its credential references. Finally, for all data items exchanged across this client-server connection, credential checks are enforced automatically.

3.2. Credential manager authentication

The CMM is an essential part of all nodes participating in a Protected Data Path, except for the end point at the client side. Two CMMs authenticate each other through an Authentication Manager Server (AMS) [9], as illustrated in Figure 2a. A long-term secure connection is established for two CMMs after the authentication. We assume that the CMM will not be compromised after authentication because it resides on a trusted kernel. Further, the CMM remains trustworthy after authentication as long as the secure connection is not broken. Since the trust properties based on this authentication are transitive, all CMMs in a distributed system can be linked via a trusted overlay network, and all messages exchanged between two CMMs via such an overlay are ‘safe’, that is, their relay via intermediate overlay nodes uses only trusted entities. The purpose of establishing such longer-term trusted connections is to avoid the costly authentication steps for every communication between each two CMMs along a trusted path. The purpose of establishing some well-defined overlay across which all trusted data exchanges take place is to limit the number of connections of which each CMM must be aware. We note that the same techniques may also be used to establish trusted links across multiple CDSes.

3.3. Credential acquisition

To limit direct user access to credentials, protection domains other than CMMs cannot access credentials. Instead, they use references to credentials. Given this fact, credential acquisition (i.e., the acquisition of references to credentials) proceeds as illustrated in Figure 2b. First, the client application locates a CMM implementing the capability API. This CMM might reside on the same node as the client or on a remote node, depending on the client application’s execution environment. In the latter case, the client must first establish a reliable connection to the CMM. For the identified CMM, the client next sends the request for credentials to the CDS, along with the CMM’s location.

The CDS generates a credential upon receiving the client’s request. This credential is first delivered to the CMM specified in the client request, again along with the location of the client. Next, the CMM verifies the integrity

of the credential, stores it in a Credential Table, and generates a locally unique credential reference number. The credential table entry is associated with the client process if the client process is on the same node; or with the connection between the CMM and the client process if both are on different nodes. The lifetime of the credential entry is limited by the lifetime of the client process or the connection. Moreover, the client can only use the PDP interface to access the actual credential, and only the associated client can use this reference number. This prevents the improper propagation of credentials. Finally, the CMM sends back the credential reference number to the CDS, which then forwards it to the client.

As stated earlier, whenever some domain wishes to access data being exchanged along a Protected Data Path, it presents its credential reference via a system call to the local CMM or via a RPC on the previously mentioned connection to the remote CMM. This reference is then translated by the CMM to an actual credential. Several advantages of using credential references rather than actual credentials are that (1) by limiting access to credentials, we avoid the risk of improper credential propagation, and (2) because a client only has a reference, it must use the API provided by the CMM to access the actual data being exchanged across the path. (3) The CDS can directly interact with CMMs to revoke a credential, if necessary, without client involvement. A subsequent use of a revoked credential by a client would result in the receipt of an appropriate error message.

3.4. Categories of protected data

For management purposes, all sensitive data objects are categorized by types, and all credentials are associated with types rather than with individual data items [31]. Thus, a credential can be viewed as the user's access rights to some special type associated with the data being exchanged along a protected path. The implementation of credentials reflects this fact, where each credential contains both a simple type descriptor identifying the type of data to which it refers and a collection of access rights to this type of data.

Since there are many choices in how some data element may be accessed (i.e., which fields, which combinations of fields, which sums of which element entries, etc.), we represent access rights as operations (i.e., as code) that when applied to the data element, provide only the information permitted by this capability. This operational encoding of access rights ‘inside’ each credential affords us with considerable flexibility. Stated more explicitly, the following choices exist for how this may be implemented: a credential may directly specify some piece of code to be applied to the data, or instead, it may name some kernel-level service implementing this operation. Such code may be written in some high-level language that is compiled dynamically [13], or it may be some executable binary code [12]. The code may reside ‘in’ the credential, or it may be acquired later through a code repository [11]. For complex operations, the credential may just indicate some service name, and the capability check module will then automatically link this named service into the protected data delivery path.

3.5. Capability-bound I/O channel

Protected Data Paths require the association of capabilities with data paths, i.e., with I/O channels. Toward this end, we leverage the fact that operating system kernels already maintain I/O channels like file descriptors and socket connections on behalf of applications. We use a socket connection to demonstrate how this association and the implied capability checks are carried out. Other I/O operations, like those for disks, follow a similar pattern.

To establish a capability-bound socket channel, two applications first create a normal socket connection. Next, the request side application contacts the CMM storing the actual credential. The CMM translates the provided reference number into the actual credential and delivers it to the remote CMM on the target node through a trusted path. The CMM then determines where to apply the credential check on the protected data object, dependent upon whether there are CMMs on both sides and/or the network environment. Once determined, it intercepts all proper read/write system calls to the socket to replace them with credential-based socket operations, by inserting a Protected Data Layer (PDL) protocol header. After this I/O interception is complete, applications communicate with each other using what appear to them as original I/O system-calls. The only requirement for the sending side application is to specify the meta-information about the protected data object prior to the write operations. More details appear in the implementation section.

4. Additional implementation detail

The PDP prototype is based on the Linux 2.4 kernel. The Capability Distribution Server and the Authentication Manager Server run on an independent host as regular applications. The Credential Manager Module (CMM) is a kernel module in each of the node participating in the Protected Data Path except that it is optional on the client machine. The CMM provides APIs for applications to bind credentials to the I/O channels established by applications, thereby turning unprotected channels into Protected Data Paths.

```
Struct OwnerInfo {
   union {
        Process Identifier;
        TCP Connection {source address/port,
          destination address/port};
   };
};
Struct Operation {
    CHAR[32]    Operation Name;
    INT         Operation Type;
    CHAR[]      Operation Payload {
                    ECode or
                    Binary Code or
                    Service Name/Location
    };
};
Struct Credential {
    INT             Size;
    CHAR[32]        Credential Issuer;
    CHAR[32]        Credential Type
                        Descriptor;
    CHAR[64]        Type Dependent
                        Information;
    OwnerInfo       Owner;
    INT             N: Number of Operations;
    Operation*[N]   Pointers to Operation;
    Operation*      Default Operation;
    Operation*      Must Apply Operation;
    CHAR[]          Data of All Operations;
    CHAR[32]        Digital Signature;
};
```

Figure 3. Credential structure

4.1. Credential structure

A credential encodes the user's access rights on a specific protected data type. It has four parts: the data type, the owner, the list of permitted operations, and the digital signature. As illustrated in Figure 3, jointly, the credential issuer and the credential type descriptor uniquely specify the type of the protected data object. In addition to the type descriptor, we include some type-dependent information inside the credential. For example, for the type 'camera captured data' such information may include the dimension and format of the image. Such information may be critical to the operations specified in the credential.

The owner of the credential is defined by a process identifier if the credential resides in the same node as the application, or by a TCP connection descriptor if a remote CMM is involved. For security reasons, every time the application domain presents the credential reference number, the CMM will use the owner information to match the caller.

The other part of the credential is a collection of operations that define how the application can view this data type. An application may acquire a list of all operations inside a credential and then choose the proper operations when making a request for a protected data object, or such operations may be synthesized from declarative specifications. Among the list of operations, there will be one default operation defined for the case when a user has no preferences. Also, for some credentials, there will be some operations that 'must' be applied to the object. Such operations may be used, for example, to filter out certain information the user should not see.

The digital signature is the mechanism used to prevent credential forgery. While highly secure cryptography is expensive, we only do a one time check when the CMM first receives a credential. The reference to the credential by an application does not incur this integrity check, thus removing this expensive task from the data delivery path. Also, the credential exchange between CMMs on a trusted path does not require credential verification, either. In our prototype, we calculate the MD5 hash of the credential, and then encrypt it with the private key of the credential issuer. Assuming that the CMM knows the public key of the credential issuer, it can decrypt the signature to check the integrity of the credential.

4.2. Credential examples

We introduce two types of credentials, one restriction-based, the other reference-based. The purpose of the former is to limit the content an end user can see, whereas the latter is for proxy-type applications. Restriction-based credentials are simple. The main components of a restricted-based credential are a group of operations that will tailor the data for end users, by removing sensitive parts. Toward that end, a special Full-Access credential is assigned to every trusted kernel domain in which the CMM is installed.

Reference-based credentials permit an application to 'peek' into the data object to retrieve relevant meta-information, but the application cannot touch the actual data. Instead, the data object is stored in some inaccessible place, such as the *KCache* introduced later. An application with reference-only credentials still uses regular 'read' system calls, but such reads only return some reference token to the data object. The token behaves just like a regular data object. The application can pass the referenced data object to another protection domain using this token (if the target domain owns the proper credentials). This involves a translation of the token to an actual data object in the trusted OS kernel domain. Or, the application may cache the token for future access to the data object it references. The web proxy cache application is an example for such usage of reference-based credentials.

```
Struct Header-Modifier {
    INT         Type;
    INT         Location, Size, Format;
};
Struct Protected-Data-Layer {
    INT         Size;
    CHAR[32]    Credential Issuer;
    CHAR[32]    Data Type Descriptor;
    INT         Size of Message Header;
    INT         Size of Message Body;
    INT         Size of Message Trailer;
    INT         N: Number of Header/Trailer
                    Modifiers;
    Header-Modifier[N]  Header/Trailer
                            Modifiers;
};
Struct Data-On-Transfer {
    Protected-Data-Layer     Message
                                 Descriptor;
    CHAR[]      Message {
                    Protocol Header;
                    Real Data Object;
                    Protocol Trailer;
                };
};
```

Figure 4. Protected data on transfer

4.3. I/O interception

The PDP mechanism enforces the capability check on I/O channels between protection domains by intercepting all I/O system calls. This is done by mapping such calls to function pointers inside *sock* or *file* structures. By replacing those pointers, we can redirect all read or write calls to a special PDP handler.

To better understand how I/O channels are intercepted, let us make clear what kind of data is transferred between protection domains. In distributed applications, a message usually contains three parts. The header/trailer provide meta-information, and the data portion carries the actual data object. For example, the response to a *HTTP Get* request may consist of a HTTP header and the content of the requested file.

Assume that both the sending and receiving sides have CMMs installed. We insert a Protected Data Layer (PDL) to describe the data object being transferred by the Protected Data Path. The PDL header contains type information about the data object, the protocol information of the message, and some data-dependent fields in the protocol header and trailer. We request the sending side application to provide the above information as described in Figure 4 prior to writing any message containing a protected object to the socket.

The PDL header is used by the CMM in order to split the protocol header/trailer from the actual data object contained in a message, so that the credential operation can be applied to the appropriate data. Also, some protocols include data-dependent information in the header or trailer. For example, the HTTP header has a *Content-Length* field defining the size of the actual data object. Because the credential operations can change the data object, we have to define a Header/Trailer Modifier to update the protocol header accordingly. This modifier defines the necessary information to update the field. The current PDP implementation for HTTP supports only the data size update.

The interception of I/O channels is enforced at the moment when the application presents credentials to bind the I/O channels. We provide an API for the application to specify what credential reference number it owns and what communication path (we will use a socket as an example here) it has set up for receiving the data. The CMM then translates the reference number to an actual credential and marks the corresponding socket. Next, it contacts the CMM located at the other end of the socket and passes the credential to it via a trusted connection. The two CMMs then negotiate 'where' to apply certain credentials operations. That decision may depend on the properties of these operations, such as the resulting network bandwidth consumption [14], the current trust in the OS kernels that carry out the operations [25], etc.

Once the I/O channel is bound with a credential, the client side sends requests to the data owner in the usual fashion. Upon receiving a request, the data owner application notifies the CMM that it will transfer a protected object through a specific socket, with all the information illustrated in Figure 4. The CMM checks whether the socket has a correct credential bound to it. The application then writes the data using a regular write system call, which is redirected to the PDP code. If this code establishes that access rights require the execution of a data object processing handler, then a new data object is produced. At the same time, the protocol header/trailer is updated based on the PDL header information. Finally, a new message including the new PDL header, the updated header/trailer, and a new data object is passed to the original I/O write code.

If the credential operation is applied at the receiving side, then we extract the data object based on PDL header and produce a new object based on the credential operation. Next, the protocol header/trailer is updated accordingly, and that result is passed along with the new object to the application through the original read call.

4.4. KObject and KCache

The PDP mechanism requires operations to be performed on data objects in the OS kernel domain. Toward this end, PDP provides an abstraction that 'organizes' the kernel data buffers that contain actual object data. Figure 5 describes the basic structure of a

```
Struct MemBolck {
    CHAR*             Buffer;
    MemBolck*         Next;
};
Struct WriteDescriptor{
    VOID*             Last Block for
                          Write;
    INT               Offset inside Block;
};
Struct ReadDescriptor{
    VOID*             Last Block for Read;
    INT               Offset inside Block;
};
Struct KObject {
    CHAR[32]          Name;
    RW_Lock           Lock;
    FLAG              Flags;
    TIME              Expiration Date;
    Union {
        MemBlock*     FirstBlock;
        Page*         FirstPage;
    }u;
    INT               Size, ExpectSize;
    WriteDescriptor   WritePosition;
    ……                Other Minor Members;
    int (*iovec_for_read)(iovec* vec,
           ReadDescriptor* pos, int size);
    int (*iovec_for_write)(iovec* vec,
            int size);
    void (*commit_write)(int written);
};
```

Figure 5. KObject structure

KObject and the APIs it provides. Stated simply, a KObject is a sequential read/write data object represented by a list of chained memory blocks. Every KObject has a unique name, and provides an interface for read and write operations via its embedded function pointers. All kernel level data processing is based on KObjects. A credential operation accepts one KObject as input and produces another one as output. Depending on whether this operation can operate on the memory block incrementally, the representation of the KObject can be either one block of memory or multiple chained blocks.

The lifetime of the KObject is determined at creation time. Typically, it is limited by the lifetime of the application that acquires it, but that is not the case for the proxy cache application. This application operates by requesting data objects from servers or peers. For protection purposes, such objects are stored in the kernel domains as KObjects, and the proxy application uses reference tokens. Since application may cache this reference token for future requests even after restart, the corresponding KObjects should not to be dismissed. This is achieved by storing such KObjects in the kernel's KCache module, developed for purposes like these. KCache provides basic interfaces for creating, retrieving, and destroying KObjects. Also, it can perform memory-to-disk swaps for long term storage or relieving the memory pressure.

4.5. Discussion

In our current implementation, all data exchanges through a Protected Data Path are in plain-text mode, which means that they are subject to eavesdropping, IP spoofing, TCP hijacking etc. Such threats can be addressed by inserting a transparent layer for data encryption into the intercepted I/O channels. An SSL-like mechanism on top of the intercepted socket connection can free PDPs from such threats.

Another issue is system integrity. PDPs rely heavily on the presence of uncompromised OS kernels, and all machines participating in Protected Data Paths are required to be trustworthy. Although this may be easy to achieve if all nodes are owned by the same organization, it may be difficult if multiple organizations are involved. One method for addressing this issue is to use the Trusted Platform Modules [32] specification. We are currently exploring this by experimenting with future virtualization-capable processor architectures that follow this specification.

Our current Linux implementation of PDPs protects sensitive data from application access by storing it in kernel address space. Malicious users with root access rights to the machine can easily spy on the kernel level data. Our next step is to utilize virtualization techniques like Xen [33] to shift the PDPs into an isolated VM. By doing so, we can (1) fully isolate the protected data from the applications, (2) reduce the risk of kernel level operations of the credential check, and (3) provide the guarantee of system integrity at the hypervisor level.

5. Applications

Applications that can benefit from Protected Data Paths have several characteristics. First, sensitive information must be exchanged between different application components. Second, there must be application components that need not have detailed knowledge about the actual data being exchanged in order to handle such exchanges, an example being data relaying and caching. PDPs are important in this context when there are application components that are not fully trustworthy, or when there is reluctance or there are legal reasons for not investing such trust. For example, the HL7 hub in a health information system handles message exchanges between the IT subsystems in different departments. Such software only needs to know the basic rules of message routing. It need not see the actual patient transcript containing private information.

In this section, we examine a system that delivers digital content to end-users. It utilizes the Apache HTTP Servers and Squid Web Proxy Caches to form a content distribution network. Considering the complexity of the Apache and Squire software, we do not want to expose some of the digital content to them. Instead, we enhance them with PDPs. This only requires small modifications. For instance, we only added around 150 lines of code to the Squid Proxy Cache to use PDPs.

Here, all protected digital content is represented by regular URLs. The client side application acquires credentials from a Capability Distribution Server. It then sets up a Protected Data Path to the proxy or server. The Apache and Squid applications controlling content delivery on the path only have reference-based credentials to the data being exchanged. As a result, these potentially untrusted applications cannot access any of the sensitive data being exchanged.

5.1. End user application

There are standalone Capability Distribution Servers that authorize end users and issue credentials to the end user application. The client side application sets up a socket connection to either the Apache Server or the Squid Web Proxy Cache, and then binds credentials to the socket. It sends a request for digital content using the *HTTP Get* command on the socket connection. To ensure that the server side knows that this request is for a protected data object, a new HTTP header field '*Credential: Enable*' is inserted into the request.

5.2. Apache

We modify Apache slightly to be able to serve protected digital content, with the assumption that all data is stored securely, i.e., in a location not directly accessible to the server. The storage nodes present a VFS-like interface. For data movement, the Apache server does not need access to actual data; it need only move the data object from storage to some outgoing socket connection. As a result, the CDS assigns only reference-based credentials to the server. Apache opens the data object just like a regular file. It then binds the opened file with a credential. In this step, a KObject is created and the data is moved from storage to the memory buffers of the KObject.

There are two ways to move data from the KObject to the outgoing socket. One way is for Apache to make a read call, get a reference token to the KObject, and then treating the reference token as a regular data object and preparing a HTTP reply based on it. Next, Apache notifies the CMM about the format of the reply. This helps the CMM translate the reference token back to the KObject at write time. The other way is to utilize a sendfile-like call provided by KCache. With this call, a fast path is set up between the KObject and the socket to avoid the steps of KObject to/from reference token.

5.3. Squid

The Squid Web Proxy Cache provides more functionality than what is explained for the Apache Server above. Here, when Squid first receives a *HTTP Get* request from the client, it will set up a connection to the backend server or a peer proxy-cache. It then binds its reference-based credential to the connection. From Apache's point of view, Squid is no different from a regular client. However, since our purpose is to relay the server response to the client and cache the data object if possible, we apply the credential check and intercept the I/O call at the Squid side. The full data object is transferred to the kernel of the machine running Squid. Now the CMM moves the data into a KObject and generates a unique reference token. This reference token is passed to Squid when it tries to read the HTTP response. The reference token is then wrapped with a new HTTP header to form a normal HTTP response to the client. When writing out this response, the token is translated back to the actual data object.

Because Squid tries to cache the reference as a normal data object, the corresponding KObject must be maintained to have the same lifetime. We use a kernel-level cache, termed KCache, to organize all such KObjects. At creation time, a firm expiration date is specified for its longest possible lifetime. When a reference token expires in Squid's application-level cache, Squid should notify KCache to remove the corresponding KObject. Moreover, KCache swaps the KObject to/from disk under memory pressure, or at module load/unload time. Thus, the reference token from Squid and the KObject from KCache can be consistent at all times.

Another nontrivial issue is how to handle the reading and writing of a reference token. The CMM generates the reference token as long as it receives the header of the HTTP response. When Squid reads in this token, it thinks it has received all of the data. However, there may still be incoming data pending on the incoming socket. To deal with this, the read call will return a special error code EPENDING if there is still data pending on the incoming socket. The EPENDING return code provides an event for Squid to trigger data movement from the socket to the KObject. Similarly, when Squid sends out a HTTP response containing a reference token, it may get a 'false' successful return. The EPENDING return code will guarantee that Squid can move all data from the KObject to the outgoing socket.

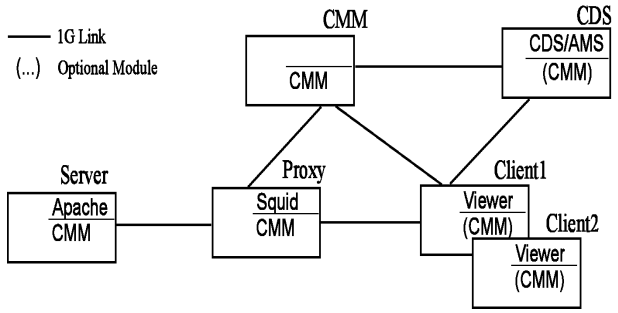

Figure 6. Test bed

6. Experiment

All experimental evaluations are performed on the Georgia Tech Emulab installation [30], termed netlab. Experiments use a simulated network with six nodes, as show in Figure 6. The network includes one node for the Capability Distribution Server and Authentication Manager. One node is used for running the Apache HTTP server, another node for the Squid Web Proxy Cache server, a node for storing client credentials, and two nodes for client applications. All nodes are Pentium IV 2.8GHz machine with Gigabit Ethernet links. There is no artificial network delay inserted between any two nodes.

We first examine the cost of PDPs related operations using a micro-benchmark. We then evaluate the PDP-enhanced versions of Apache and Squid.

Table 1. Preparation cost for a protected data path

Steps \ Credential Size	Time Cost (microsecond)				
	256B	1K	4K	16K	32K
Authentication between CMMs	11681.55				
Authorization from CDS	1347.23				
Acquire Credential from CDS	609.65				
Credential from CDS to CMM	344.1	375.09	388.4	799.7	970.9
Credential Verification	38.25	40.82	49.58	84.56	131.43
Bind Credential to Socket	502.70				

6.1. Micro-benchmark

We first measure the basic costs of setting up a protected data path. Table 1 shows the necessary steps in sequence. Authentications between CMMs are the essential to constructing the secure framework. Although this relies heavily on cryptological methods, the per-node-pair cost is small. The Capability Distribution Server authorizes end users and grants them the virtual reference number of the credentials. The CDS must deliver the actual credential to a trusted CMM. Delivery and credential verification costs may vary depending on the size of the credential. All these steps are per-credential and can be amortized over the lifetime of the application. Finally, to use the credential, the application must submit a request to the CMM for binding the credential to an I/O channel like a socket. This is a per-connection cost, but we expect that the user will use the same connection to request multiple sets of data.

6.2. Apache

We measure both request response time and throughput for the Apache HTTP Server using protected Data Paths. Figure 7 compares the response times of the original vs. modified Apache on different file sizes. As evident from the figure, the two lines are almost identical. The additional cost of binding credentials to sockets for the Protected Data Path is significant for small files but negligible for large files. Moreover, this cost can be amortized on HTTP persistent connections if users request multiple sets of data of the same type on one connection.

Figure 8 compares the throughput of the protected vs. unprotected Apache HTTP servers. Here, a 64-thread client program sends requests continuously to the server.

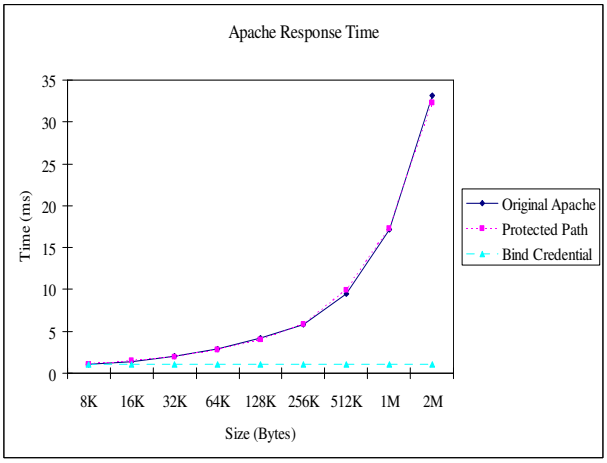

Figure 7. Apache request response time

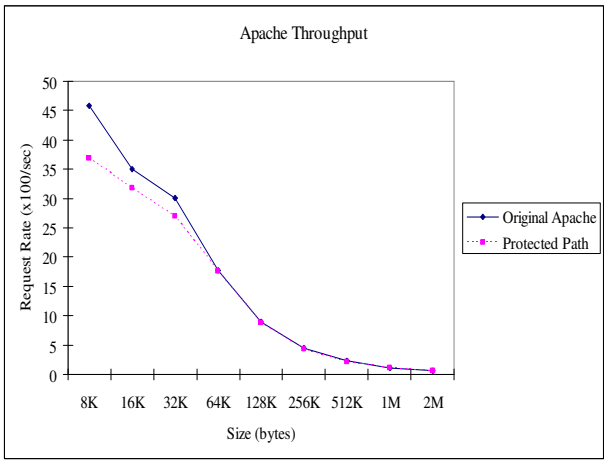

Figure 8. Apache throughput on different file Size

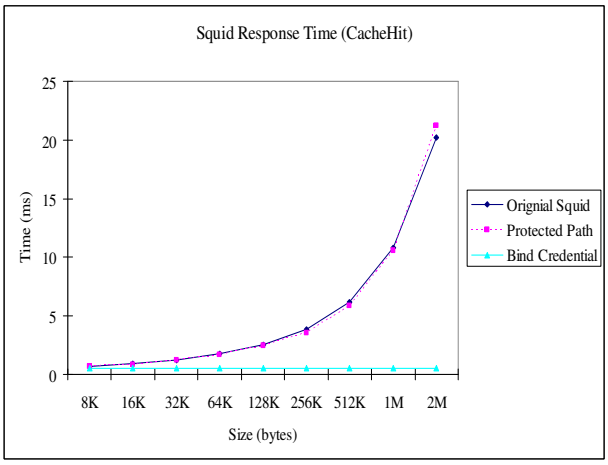

Figure 9. Squid request response time on cache hit

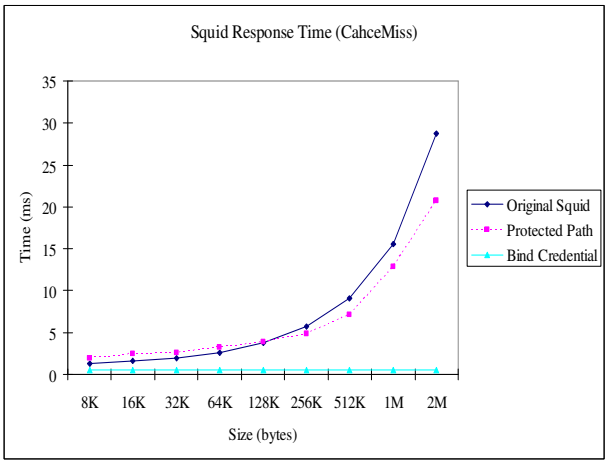

Figure 10. Squid request response time on cache miss

Each thread sends a new request immediately after the previous is complete. We can see that for large files, the use of PDPs has almost no effects on throughput. For small files, due to the cost of credential binding and the in-kernel credential operation, throughput is reduced by 10~20 percent.

6.3. Squid

Evaluations of the Squid proxy have results similar to those attained for Apache. In these experiments, when the data object is located in the application cache of Squid (a cached reference token for the Protected Data Path case), Squid serves the request locally. Both implementations present similar results, as shown in Figure 9. When the data object is not located in the object cache of Squid, it contacts the server for the data. We see from Figure 10 that PDPs are worse for small files, but better for large files. This is because of reduced memory copying and reduced overheads for the KCache vs. application-level caching [14,23]. When dealing with large files, the in-kernel data path uses one less memory copy, thus improving response time by up to 25% in our measurements.

7. Conclusion

We have presented a mechanism to create a Protected Data Path for controlling the exchanges of sensitive data. It separates applications into multiple protection domains and assigns credentials to each domain based on their needs. Untrusted applications like third-party proxies with PDP enhancement can be inserted into the data path if running on a trusted system. In-kernel credential checks guarantee that such applications can only access data based on the credentials they own. We have also described an example in which PDPs are used by an untrusted proxy-cache to cache and deliver sensitive data without compromising the security policy.

Our future work will extend the notion of PDPs to provide differentiated data protection at kernel level using credential operations. In addition, by using modern virtualization techniques, path data will be further isolated from untrusted applications and/or from potentially compromised operating system kernels.

REFERENCE

[1] Akamai. www.akamai.com
[2] Squid Web Proxy Cache. www.squid-cache.org
[3] Apache HTTP Server. httpd.apache.org
[4] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object

Cache. In Proceedings of the USENIX Technical Conference, pp. 153-163, San Diego, California, January 1996.
[5] JBOSS Application Server. www.jboss.com
[6] IBM Websphere Application Server. www.ibm.com
[7] Health Level Seven http://www.hl7.org/
[8] E. Cohen and D. Jefferson. Protection in the Hydra Operating System. In Proceedings of the fifth ACM symposium on Operating systems principles. November, 1975.
[9] T. Y. C. Woo and S. S. Lam. Authentication for Distributed Systems. In ACM Computer, Volume 25, Issue 1. January 1992.
[10] M. Kaminsky, G. Savvides, D. Mazieres, M. F. Kaashoek. Decentralized User Authentication in a Global File System. In Proceedings of the 9th ACM Symposium on Operating Systems Principles. October 2003.
[11] P. Widener, K. Schwan, and F. Bustamante. Differential Data Protection in Dynamic Distributed Applications. In Procedings of the 2003 Annual Computer Security Applications Conference, Las Vegas, Nevada, December 2003.
[12] I. Ganev, K. Schwan, and G. Eisenhauer. Kernel Plugins: When A VM Is Too Much. In the 3rd Virtual Machine Research and Technology Symposium, May, 2004.
[13] G. Eisenhauer and K. Schwan. The ECho Event Delivery System. College of Computing Technical Reports GIT-CC-99-08
[14] J. Kong and K. Schwan. KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers. In Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '05). June 2005
[15] I. Foster, C. Kesselman, G. Tsudik and S. Tuecke. A security Architecture for Computational Grids. In Proceedings of the 5th ACM Conference on Computer and Communications Security. 1998.
[16] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson and A. Essiari. Certificate-based Access Control for Widely Distributed Resources. In Proceedings of the Eight Usenix Security Symposium, August, 1999
[17] G. Stoker, B. S. White, E. Stackpole, T. J. Highley and M. Humphrey. Toward Realizable Restricted Delegation in Computational Grids. In Proceedings of the International Conference on High Performance Computing and Networking Europe (HPCN Europe 2001) , Amsterdam, Netherlands, June 2001.
[18] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in Distributed Systems: Theory and Practice. In ACM Transactions on Computer Systems, Vol 10, Issue 4. November 1992.
[19] A. O. Freier, P. Karlton and P. C. Kocher. The SSL Protocol. Internet Draft, March 1996.
[20] Data Encryption Standard. Federal Information Processing Standards Publication 46-2. December 1993.
[21] Rivest, R., Shamir, A. and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, February 1978.
[22] D. Maltz and P. Bhagwat. TCP Splicing for Application Layer Proxy Performance. IBM Research Report RC 21139, March, 1998.
[23] M. Rosu and D. Rosu. An Evaluation of TCP Splice Benefits in Web Proxy Servers. In the 11th International World Wide Web Conference. May 2002.
[24] A. Luotonen. Tunneling TCP based protocols through Web proxy servers. Internet Draft. August 1998.
[25] R. Viswanath, M. Ahamad and K. Schwan. Harnessing Non-dedicated Wide-area Clusters for On-demand Computing. IEEE International Conference on Cluster Computing (Cluster 2005). September 2005.
[26] J. G. Steiner, B. Clifford Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In Proceedings of the Winter 1988 Usenix Conference. February, 1988.
[27] G. Eisenhauer, F. E. Bustamante, and K. Schwan. A Middleware Toolkit for Client-initiated Service Specialization. ACM SIGOPS Operating Systems Review. July 2001
[28] J. Galbraith and O. Saarenmaa. SSH File Transfer Protocol. Internet Draft, June 2005.
[29] T. Ylonen and C. Lonvick. SSH Protocol Architecture. Internet Draft, March 2005.
[30] Georgia Tech Netbed Based on Emulab. www.netlab.cc.gatech.edu
[31] Patrick Widener. Dynamic Differential Data Protection for High-Performance and Pervasive Applications. Ph.D. Thesis, Georgia Institute of Technology, July 2005.
[32] Trusted Computing Group. https://www.trustedcomputinggroup.org/home
[33] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Wareld. Xen and the Art of Virtualization. In Proceeding of the 19th ACM Symposium on Operating Systems Principles. October, 2003.