RC23205 (W0405-045) May 4, 2004
Computer Science

IBM Research Report

Attestation-based Policy Enforcement for Remote Access

Reiner Sailer, Trent Jaeger, Xiaolan Zhang, Leendert Van Doorn
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Attestation-based Policy Enforcement for Remote Access

Reiner Sailer, Trent Jaeger, Xiaolan Zhang, Leendert van Doorn
IBM T.J. Watson Research Center
Hawthorne, New York, 10532
{sailer, jaegert, cxzhang, leendert}@us.ibm.com

Abstract

Intranet access has become an essential function for corporate users. At the same time, corporation's security administrators have little ability to control access to corporate data once it is released to remote clients. At present, no confidentiality or integrity guarantees about the remote access clients are made, so it is possible that an attacker may have compromised such a system and is now downloading or modifying corporate data. Even if the remote user must know a password to establish a remote access tunnel to the corporate Intranet, it is possible that a malicious process can hijack an existing connection and access corporate services and documents by masquerading the authorized remote user. Thus, even though we have corporate-wide access control over remote users, the access control approach is currently insufficient to stop these malicious processes. We have designed and implemented a novel system that empowers corporations to verify client integrity properties and establish trust upon the client policy enforcement before allowing clients (remote) access to corporate Intranet services. Client integrity is measured using a Trusted Platform Module (TPM), a new security technology that is becoming broadly available on client systems, and our system uses these measurements for access policy decisions enforced upon the client's processes. We have implemented a Linux 2.6 prototype system that utilizes the TPM measurement and attestation, existing Linux network control (Netfilter), and existing corporate policy management tools in the Tivoli Access Manager to control remote client access to corporate data. This prototype illustrates that our solution integrates seamlessly into scalable corporate policy management and introduces only a minor performance overhead.

1 Introduction

Remote access to corporate Intranets is now an essential aspect of the corporate work environment. Client systems are often used by employees both inside the corporate Intranet and remotely from home. Since the remote client systems that access the corporate Intranet are largely administered by their users, this presents corporations with the problem of controlling access to their information. Such remote client systems can download confidential corporate information, so leakage of this information is a problem. Further, remote client systems can modify sensitive corporate data, so the integrity of the modifications are also an issue. In this paper, we present a novel system that enables corporate data servers to manage confidentiality and integrity requirements on accesses by remote client systems.

Figure 1a shows the generic systems model: AliceCorp is the corporation that offers remote access to its employees. Bob is one such employee who accesses the Intranet with his client system. The remote access client program on Bob's client enables access to the corporate Intranet.

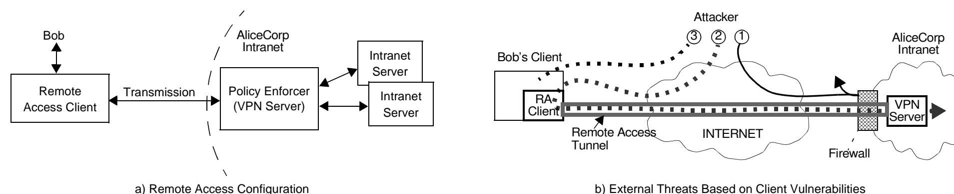

Figure 1: Remote Access System

AliceCorp requires that the corporate data remains protected when offering the remote access service, i.e., that remote requests originating from the client do not compromise the integrity of the corporate Intranet and that confidential contents of responses to the client do not leak through Bob's client. However, a client's ability to ensure enforcement of the corporate policy is rudimentary at best today.

Today, most clients are mainly under control of their users and the client configuration and software can differ considerably from what the corporation would prefer. A client machine that was originally configured by the corporate

administrators may no longer meet the original security requirements, such that several security vulnerabilities may be present. Consequently, clients have been conveniently considered insecure and untrusted because protecting them was either too expensive or too restrictive. As a result, companies used appearance factors, such as operating system type and patch level, the version of the anti-virus data base, or active/non-active system passwords to reason about the client's trustworthiness.

Specifically, this paper examines two kinds of attacks (illustrated in Figure 1b) that cannot be countered by existing remote access control mechanisms because of their lack of reliable client security guarantees:

Firewall-Bypassing: External attackers can gain unauthorized access to the corporate Intranet. Most corporations today, having invested into security for firewall and intrusion detection systems, have raised the bar for external attackers considerably. Therefore, (Figure 1b, attack 1) attackers find it easier to compromise the weaker client systems to hop indirectly into the Intranet. By using the external interfaces of Bob's client, attackers can gain access to Bob's remote access tunnel and thus bypass the corporate firewall (see Figure 1b, attack 2). This is a real-time attack that is very difficult to discover from within the corporation because the attacker's actions are covered by the user's actions. With many operating systems offering remote sessions, it cannot be determined whether a service request originating from a client is actually initiated by the client's local user or an external attacker that has gained unauthorized access to the client remotely or locally through a Trojan Horse.

Information-Leaking: Confidential data can also leak indirectly through the client (Figure 1b, attack 3). When data is sent to a remote client, control over these data items is transferred to the user's client and to the user. Users may allow other users, knowingly or by mistake, to download such data in peer-to-peer sessions or to copy data onto removable storage devices and distribute them afterwards; manipulated programs (Trojan Horses) on the client can gather and leak information as well.

Current research addresses some of the problems inherent in preventing these attacks, but significant issues remain. First, efforts are underway to leverage trusted hardware, such as the TCG trusted platform module (TPM) [1], to measure system integrity, but determining practical integrity levels and using them to make access control decisions remain open questions. Second, classical integrity policies, such as Biba and Low Water Mark (LOMAC) [2], exist, but their application to remote clients may be impractical. For example, some degree of dependence on lower integrity data may be permissible. Lastly, while the basic notions of reference monitors are well-understood [3] and distributed policy enforcement systems exist [4], preventing these attacks requires greater dependence on the abilities of the remote client than usual. For example, control of information leakage requires that the remote client be able to control itself network interfaces according to a corporate policy. Issues include determining which components can be trusted to enforce policies, determining the kinds of policies that can be enforced, and identifying practical limitations that can be made to enable more effective enforcement.

This paper presents an approach that can be used in the construction of VPN connections to control access to corporate data access and use by remote clients. First, we leverage the TCG/TPM to measure the integrity of remote client systems. We build this on an existing *integrity measurement system* that runs on Linux and measures all executable code, including libraries and kernel modules, that are loaded onto a Linux system [5]. Next, we define an *integrity policy model* that associates semantics with the integrity measurements of remote clients and enables expression of policies that leverage subject identity and integrity in access control decisions. We utilize the Tivoli Access Manager (AM) system [6] to represent and distribute our integrity policies to remote corporate clients. Lastly, we implement a *personal firewall* on remote clients running Linux that enforces confidentiality policies. Because we can measure the integrity of the remote system reliably using the TPM, the integrity of the remote client can be accurately determined and unauthorized access via the firewall can be prevented. Further, we can verify the integrity of our enforcement mechanism and manage possible leakage paths, so enforcement of confidentiality requirements on corporate information is possible. We have implemented a working prototype on Linux 2.6 and our measurements show that the performance impact of this approach is minor and accounts to about 4% on TCP traffic.

In Section 2, we detail the individual problems that underlie enforcement of access control on remote clients and outline the goals of this research. After describing related work in Section 3, we outline the approach that we take to measure and control remote client's access to corporate data in Section 4. Section 5 details the implementation. Section 6 contains the evaluation of the approach and its implementation. We conclude the paper and discuss future directions in Section 7.

2 Background

We aim to provide a remote access policy enforcement architecture that can protect against *hijacking of authorized remote access sessions* by unauthorized parties (Security Goal SG1) and *leaking of confidential data from the client* through remote attacks against the client system (SG2). For this purpose, we will not only describe the policy enforcement architecture, but also the proper policy that achieves these goals.

2.1 Attacker Model

Figure 2 illustrates the attacker model that we consider and motivates our enhancements to the corporate policy necessary to specify the security requirements for remote access more accurately. It identifies three possible means of attack: (1) compromising user authentication; (2) compromising transport security; and (3) compromising client system integrity.

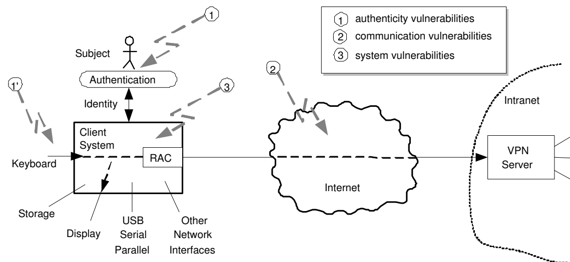

Figure 2: Remote Access Attacker Model

We focus on attack type 3 in this paper. Such attacks involve getting unauthorized code to run at higher levels of privilege. Such attacks can be implemented in a variety of ways, such as through viruses, Trojan horses, buffer overflows, or other code injection attacks. Using such techniques, the attacker can inject malicious code onto the client system after the corporate system administrators have configured the system. Such malicious code may be used to gain unauthorized access to the Intranet directly by using existing remote access connections, e.g., by remotely accessing the client system through any of the depicted interfaces (USB, Wireless, Bluetooth, Ethernet, etc.) and through it accessing the Intranet. Also, attackers can gain access to corporate data used in current remote access session or corporate data that has been stored on the client system. We are, however, not going to address attackers watching the display passively from remotely and in this way extracting information from the client (shoulder surfing attacks).

Attack type number 1 refers to the quality of user authentication. Corner and Noble propose a user authentication systems [7] that protects against such attacks. Attack type number 2 is usually countered by deploying a secure tunnel between the client and the VPN server [8].

2.2 Measurement

The first problem is to determine whether the remote client is running authorized code when it is connected to the corporate Intranet. Since any memory of a process can become executable, this is a difficult problem in general. However, integrity measurement approaches are emerging that can be leveraged to narrow the problem.

Recent advances in hardware enable better integrity measurement on client systems. Many clients are now sold with TCG Trusted Platform Module (TPM) [1] chips included, and the low cost of such chips indicates that broad application is possible. The TPM has a set of registers that it protects from the client and it provides two operations on each register content: *extend* and *quote*. The *extend* operation takes a value as input and computes the SHA-1 hash [9] of the current register content and that value. This enables the user of the TPM to build a hash chain. The envisioned use of such a hash chain is to measure a predefined sequence of code loads, such as for *authenticated boot* of an operating system from the system's BIOS and boot-loader. The *quote* operation results in the TPM generating a signed message, using a key protected by the TPM, of the target register's contents. This message can be used to send an authenticated hash chain to a remote party which in turn may validate the integrity of the code contained in the hash chain that belongs to the client. The TPM has other functions, such as random number generation and sealed storage, but these are not relevant to our discussion.

Recent work uses the TPM to measure the integrity of the application level on Linux systems [5]. Building on the integrity verification of the Linux operating system using the technique described above, Linux has been extended to use the TPM to measure the code loaded on it, including kernel modules, applications, and their shared libraries. The TPM stores a value representing a hash chain of loaded code while the extended Linux kernel stores the actual log of hashes. Remote parties can verify the integrity of a client system by retrieving the Linux hash log and the quoted (i.e., signed) hash chain value from the TPM.

With respect to our case, the corporate VPN server can validate the log for a client system before permitting the connection to be opened or permitting the client to download confidential corporate information. Further, the corporate VPN server can even use integrity measurement to determine if a client system is capable of enforcing policies that can control information leakage. The research question is whether such actions can be made practical for a remote access environment. We discuss the issues of using integrity measurement to estimate system integrity and ensure effective policy enforcement below.

2.3 Security Policy

Given the integrity measurements of the previous section, we want to be able to determine the likelihood that the employee, rather than an attacker, is accessing the corporate Intranet. Measurements identify the code that has been loaded by the client system at the time of measurement. Typically, this is used to identify vulnerable code. Since vulnerable code can be compromised by an attacker, the likelihood that an attacker will compromise such programs to masquerade as the employee is high (usually assumed to be 1).

However, the situation is not always so clear-cut, as code may be protected by limiting the interfaces to it and the inputs it receives. First, some code may have known vulnerabilities that may be leveraged only through network interfaces. If access to these interfaces is limited to trusted parties (i.e., trusted as much as the corporate Intranet), then the use of such code may be permissible. Second, some programs, such as Microsoft Word, may execute macros that can permit a document writer to masquerade as the user. In such cases, the source of the inputs may determine whether an attacker may be in control of the client's accesses to the corporate Intranet.

In traditional integrity models, such as Biba [10] and LOMAC [2], integrity of a subject is based on its dependency on other subjects. For example, a LOMAC policy requires that the integrity level of the subject be equal to the minimum integrity level of the objects that it has read or executed (where integrity level is inversely proportional to likelihood of compromise or vulnerability). Unfortunately, integrity measurement does not provide a complete picture of dependency. We measure the code that is loaded, not the information flows. However, a combination of knowledge about the code and the possible information flows allowed for this code may provide a sufficient model for reasoning about security decisions. We discuss the combination of policy enforcement with code measurement in the next section, but briefly discuss policy modeling below.

In order to reason about the likelihood that the subject making a request is the employee for whom the VPN connection was made, we must not only estimate this likelihood, but the policy model must also be able to express it. Most systems either associate policy with a user or a sensitivity level (e.g., for confidentiality or integrity). In this case, we have a combination of both the employee and an estimate of the integrity level of the client system. A research question is whether current policy models are capable of expressing such policies and can enforce them effectively.

2.4 Enforcement

Prevention of particular information flows may enable us to use code that has some potential for misuse in a high integrity manner. The problem is to determine what forms of control are useful and identify the necessary enforcement mechanisms. For example, we find that it is appropriate to use confidential data in processes that may not write the data to the client's file system (i.e., all persistent writes go to the corporate Intranet). The challenge is to identify such system restrictions that enable useful processing and determine how these can be measured, so that the corporate servers can verify that their policies can be correctly enforced.

As the example above indicates, we consider enforcement options that are enforced by the client system as well as those that may be implemented by the corporate server. Leveraging client controls is required to control confidentiality because once a document is downloaded to a client it is outside of the corporate Intranet controls. It is less obvious that we also find it useful to use the client controls to enable improvements in integrity. If we can prevent information flows that may lead to compromise as described above, then we can use software that has some vulnerable configurations for high integrity operations.

The types of enforcement options that we consider include access to network connections, file systems, and other objects managed by the operation system. Thus, it is necessary for the client operating system to provide the necessary controls over these objects. For example, Linux provides the NetFilter interface that enables fine-grained control of network communication within the operating system. The Linux Security Modules framework can be used to control access to file systems and other system objects. For example, an instance of a word processor could be started that can only communicate with the corporate Intranet or a RAM file system.

Measurement is not only the basis for integrity, but also the basis for enforcement. If we load a kernel on the client system that implements the enforcement properties that we desire, then the corporate server can use the measurement of this kernel to verify that the expected enforcement will be performed.

2.5 Experiment

The research problem is to find whether an acceptable solution exists for the problems outlined above. To recap, we envision that secure client access to a corporate Intranet requires that the following problems be solved: (1) determine the integrity level of the client system based on the code running on the client; (2) determine whether to trust this client to enforce information flow controls necessary to make such integrity assumptions about client; (3) determine whether additional security properties, such as confidentiality, need to be enforced by the client and whether the kernel supports these; (4) integrate the integrity of the client into the remote access control policies governing the client's access to the

corporate servers; (5) enforce this policy on remote access clients and VPN server; and (6) track changes in the remote client's security properties (i.e., sense relevant changes in the client's software stack) and implement resultant policy changes.

3 Related Work

We examine related work in the areas of integrity measurement, security policies for integrity management, and policy enforcement on remote clients.

System Integrity Measurement: Verifying the integrity of the software stack of client systems isn't a new problem; practical solutions have only appeared recently though. Arbaugh et. al. [11] describe an architecture to securely boot operating systems in such a way that only a trusted system will be booted in all cases. *Outgoing authentication* [12] enables attestation for the software stacks of cryptographic co-processors [13, 14]. Both approaches are too restrictive for client environments because they require only completely trusted configurations boot in the former case or are ultimately implemented as single application systems in the latter case.

Some subsequent research focused on using additional hardware to assess the software stack integrity. *Independent auditors* are explored by Hollingworth et. al. [15], Dyer et. al. [16] and Molina et. al. [17]. Hollingworth suggests using the second CPU on a dual-processor board to provide autonomous monitoring and control of the operating system. Dyer et. al. apply secure co-processors from which the client system and its network access can be both protected and monitored. Zhang et. al. [18] extended these ideas to monitor the integrity of a client kernel by examining kernel data structures from a secure co-processor. Molina et. al. explore a co-processor as an independent auditor that supervises the integrity of the host operating system independently.

More recently, research has focused on measuring the integrity of systems in a secure manner and enabling verification by remote parties, called *authenticated boot*. All these architectures envision leveraging the TCG Trusted Platform Module (TPM) [1] for securely storing measurements. The NGSCB approach [19, 20] also depends on special hardware to separate a trusted system partition from the standard operating system. Terra [21] is a trusted computing architecture that is built upon a trusted virtual machine monitor that –among other things– authenticates the software running in a VM for remote parties. However, the VMM must be trusted and deriving security properties from the footprint of VM partitions appears difficult. All of these solutions would be quite expensive and at the same time very restrictive if they were to be applied to remote access client systems.

Integrity Policies: Most access control policies aim to provide system integrity guarantees, although integrity is typically implicit. Access matrix style policies, such as role-based access control (RBAC) [22, 23], associate policies with subjects or roles that stand for a set of subjects. The integrity of a individual subject or object is not explicitly specified, but the permission assignments are intended to control access to provide sufficient integrity. Integrity is also implicit in secrecy policies, such as Bell-LaPadula [24].

Policies that explicitly reason about integrity include Biba [10], LOMAC [2], and Clark-Wilson [25]. In Biba and LOMAC, subjects and objects are given integrity levels, and subjects cannot retain their integrity level and depend upon lower integrity subjects or objects. In Biba, such information flows are prohibited, whereas LOMAC permits such flows by lowering the integrity of subjects dynamically based on the dependencies they use. For Clark-Wilson, low integrity dependencies are permitted, but only if the high integrity subject either discards or upgrades the integrity of the data. Of these, only LOMAC permits accesses to change based on the integrity level of a subject, but high integrity subjects often read low integrity data, so this model is often too restrictive.

Client Policy Enforcement: Steve Bellovin proposed in 1999 [26] that firewalls should be considered for client systems as well as their historic place at network boundaries in order to improve filtering effectiveness. Also, Ioannidis et. al. [8] proposed to distribute IPSEC credentials through trust management to such distributed firewalls. On the one hand, having the filter as close as possible to the client allows fine-grained access control and ensures that all packets received by and sent from the client are intercepted by the firewall. On the other hand, moving firewalls onto the client makes them more difficult to manage and exposes them to client vulnerabilities. None of the existing approaches considers validated client security properties in the access control policy.

4 Remote Access Security Architecture

We propose a remote client access control enforcement architecture (c.f. Subsection 4.2) that implements access control using the following steps:

(i) We use non-intrusive software-stack attestation (Integrity Measurement Architecture IMA [5]) and apply it to client systems to classify the integrity of the client. This measurement approach is described in Section 2.2. The corporate VPN server receives and verifies a list consisting of measurements of all executable content that has been loaded for execution (annotated SHA-1 values of files) into the remote client's run-time since reboot. We use this information to determine in Subsection 4.1 the integrity level of the client system (i.e., the likelihood that the client is

acting properly on behalf of the employee).

(ii) The integrity measurements are also used to determine if the client can enforce desired integrity and confidentiality goals. In this case, it is the presence of trusted enforcement programs rather than the absence of low integrity programs that is the issue. We describe this verification in Section 4.2.

(iii) Given the client's integrity class and the presence of the necessary enforcement software, the VPN server can delegate enforcement of the access control policy to the client system. The access control policy and decisions are described in Section 4.3.

In Section 4.4, we demonstrate the architecture's use.

4.1 Measurement

In this step, the VPN server uses the validated integrity measurements of the client system to classify its integrity and enforcement abilities. The former determines the identity for client accesses to corporate data and the latter determines the ability of the client to control data once it is received. That is, the former determines the integrity of the client and the latter determines its ability to enforce confidentiality requirements.

We describe the approach for classifying client integrity in this section and describe verification of enforcement ability in Section 4.2. Determining the integrity of the client is the first and most important step because this establishes trust (default is distrusted) of the VPN server into predictive operation of the client and thus the programs running on the client. Figure 3 illustrates the process.

First, we use a recently developed method [5] to determine the integrity of the software running on the client, described in Section 2.2. One measurement represents the hash chain of loads for the BIOS, boot-loader, and operating system. This should match a known value for this sequence of code loads. The second measurement covers the software loaded on the operating system, including programs loaded via *exec*, kernel modules, and shared libraries. The method ensures that a remote party can cryptographically verify the source, integrity, and freshness of the measurements. Further, the properties of the measurements ensure that no measurements may be removed or reorder once made.

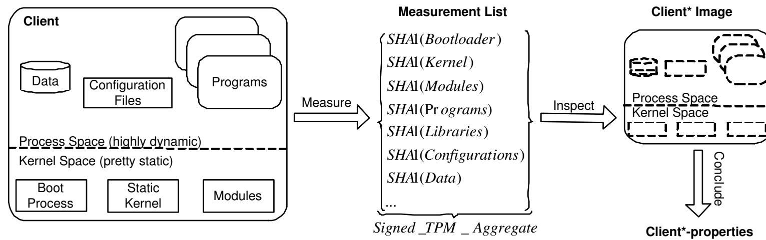

Figure 3: Attestation through Measurement Projection

Using this mechanism, the VPN server obtains and validates a list of measurements comprising all executable content that was loaded into the client's run-time since reboot. All executables corresponding to a measurement can potentially have been compromised and thus compromise the client system [1]. Consequently, compromised software might prevent the client from correctly measuring events that occur after the compromise. However, an important property of the measurement architecture is that it measures files and aggregates their measurement into the secure TPM register *before* they are loaded, so a compromised or even malicious program cannot remove itself from the measurement list [5].

Thus, we determine the integrity of the client by evaluating each measurement against a known set of measurements. Each known measurement is associated with its integrity class. These classes partition the set of known software as follows.

- **Malicious**: That this program is known to be malicious (e.g., syslogd of the root toolkit, trojan versions of libraries). The presence of one of these programs in the measurements indicates a compromise.

- **Remote Vulnerabilities**: Such programs use network connections and have known vulnerabilities to remote attackers. Web and mail clients are potentially included in this category. Their use is prohibited when connected to the Internet and may be strictly limited for access to the corporate Intranet. For example, it may not be possible to read mail and access confidential information at the same time.

- **Local Vulnerabilities**: Such programs have vulnerabilities, but these vulnerabilities are limited to local data, such as files with embedded executable content. These programs should only be run using corporate data.

[1] As we measure software when it is loaded, we would not see if this software becomes compromised after loading. However, we can and will use the known vulnerabilities of measured software and decide whether we assume this case or not.

- **Uncontrolled**: Certain programs change the kernel state, but do not yet perform integrity measurements, such as insmod and modprobe. The execution of these programs could result in the load of malicious or vulnerable code. These programs should not be run at present, but could be modified to work with the system later.
- **Acceptable**: These programs contain no known vulnerabilities or malicious code and do not enable circumvention of the measurement system.

The known set includes fingerprints (SHA-1 hashes) of all known executables and other files that are expected to be found in measurement lists of remote access clients. In our example, it includes SHA-1 hash values of all Redhat 9.0 programs and libraries including updates, the fingerprints of our own extensions for client policy control, acceptable kernels, and boot configurations.

We use these sets of fingerprints to evaluate a client according to the rules 1- 5 shown in Figure 4.

$$[\forall e \in E(client) : (e \in Known) \wedge ((e \in Malicious) \vee (e \in Uncontrolled))] \rightarrow (e \in Distrusted) \quad (1)$$

$$[\forall e \in E(client) : (e \in Known) \wedge (e \in Internet) \wedge ((e \in Local) \vee (e \in Remote))] \rightarrow (e \in Distrusted) \quad (2)$$

$$[\forall e \in E(client) : (e \in Known) \wedge (e \in Intranet) \wedge (e \in Remote)] \rightarrow (e \in IntLow) \quad (3)$$

$$[\forall e \in E(client) : (e \in Known) \wedge (e \in Intranet) \wedge (e \in Local)] \rightarrow (e \in IntMedium) \quad (4)$$

$$[\forall e \in E(client) : (e \in Known) \wedge (e \in Acceptable)] \rightarrow client \in IntHigh \quad (5)$$

Figure 4: Rules for Determining the Client Integrity Level.

The above evaluation is valid until the client loads new executables that were not previously measured. To keep track of the integrity of clients connected to the Intranet, the client must update the VPN server with new integrity measurements. It is preferable that such updates be done at measurement time and prior to the actual load.

4.2 Policy Enforcement Architecture

This section introduces an access policy enforcement architecture (see Figure 5) that must be verified by the VPN server before delegating enforcement of corporate policies. First, we describe the architecture, then we describe its verification.

Policy enforcement consists of a *personalfirewall* that intercepts all IP packets the enter or leave the client. It decides whether or not to pass or drop a packet based on policy obtained from a *policyagent* residing in user space. We describe the specific policies enforced and its impact in Section 4.3. The policy is obtained from a remote corporate policy server based on the information in a configuration file *local.conf*. The firewall extracts the service type (e.g., SSH, Telnet, HTTPS, HTTP) and the service direction (incoming or outgoing service; this can be different from the packet flow direction) from the packet and submits this information to the policy agent. The policy agent uses a local replica of the corporate access policy, *authzn_persfw.db*, to retrieve the policy for this object (e.g., outgoing SecureSHell) and returns it to the firewall, where it is used for authorizing the packet and is stored for future reference in a kernel policy cache.

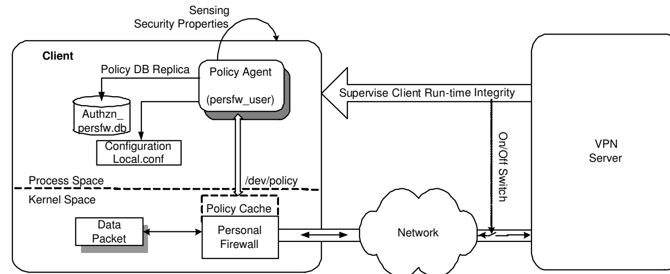

Figure 5: Policy Agent Architecture

Verification of the presence of a sufficient policy enforcement architecture is necessary before releasing confidential corporate data to the client system. The presence of the files listed above is required for policy enforcement. First, the measured kernel indicates whether it contains the personal firewall or not. Second, when the policy agent is loaded, it is measured. Third, the policy agent measures the configuration and policy database files that it actually uses. Lastly, the kernel must indicate which process it is using as the policy agent.

Thus, if the kernel and policy agent are not compromised, then the measurement of their input code and configurations should be sufficient to identify their integrity and prove to the VPN server that the client system is capable to enforce corporate policies.

Corporate policy enforcement entails protecting the integrity of corporate data from attackers and preventing the leakage of confidential corporate data to attackers. Essentially, both requirements involve control of information flows; integrity may be compromised by the flow of low integrity data into the corporation and secrecy may be compromised by the flow of corporate data to processes that can access the Internet. However, requiring complete, provable isolation may be too restrictive or complex. For example, some Internet data may be used to write a corporate document. With respect to confidentiality, the use of a program with some known vulnerabilities may be necessary to get the confidential document written. The policy enforcement architecture enables enforcement of such requirements. We describe the policy language for such requirements in Section 4.3.

In addition to running enforcement code, a kernel capable of enforcement will require other function to implement the necessary controls. For confidentiality, we may require that no data is saved in persistent storage on the client. Thus, access controls may prevent file system access or a RAM file system may be used locally. Also, emerging technologies, such as sealed storage, may be used to protect the confidentiality of data unless the proper code is run on the client. We do not specifically define these mechanisms, but enable the corporate and client system designers to leverage the mechanisms available at the time to meet the desired goals.

4.3 Security Policy

Given the client's integrity and the identity of the corporate employee, the personal firewall determines the access rights of the client to corporate data. Corporations might not be able to deliver access control decisions and enforcement on this fine-grained, per-packet-level for thousands of remote clients. However, our experiments with the prototype show that this architecture is sufficiently scalable if the policy enforcement is done locally on the client and only the general supervision of the client's computing integrity is supervised remotely by the VPN server. The actual decision function implemented by the personal firewall is shown in Figure 6.

```
Bool AccessDecision(packet(service,direction), cap(userid)) {

      [decision,client-constraints,packet-constraints] := azn(cap(userid),service,direction);

      return (decision AND hold(client-constraints)) AND (hold(packet-constraints));
}
```

Figure 6: Access Control Decision Function.

First, the personal firewall extracts service type and service direction from an access controlled packet. Then, the policy agent adds the user's credentials and retrieves the authorization decision from the policy data base. If the result is "permission denied", then this decision is communicated to the firewall, and this packet is discarded. If the result is "granted", then the policy agent evaluates extended attributes that refer to *client constraints*. Client constraints describe the integrity and confidentiality levels required of the client system (e.g., the client has mechanisms to implement high secrecy control). If they are fulfilled, then it communicates the *packet constraints* and a preliminary "granted" permission back to the personal firewall. The packet constraints describe the enforcement required by the client on the packet (e.g., protect the confidentiality of the data in a 'high' manner). The personal firewall will evaluate the constraints and determine whether they meet the *hold* predicate. This predicate implies that the client system's integrity and confidentiality enforcement meet or exceed the constraint level.

Note that, like integrity, confidentiality constraints are expressed in terms of high, medium, and low classes. High confidentiality aims to eliminate all risk of leakage. We describe scenarios that aim to achieve this in Section 4.4. Medium confidentiality may originate either from a less capable kernel configuration (e.g., lacking control of all information flows) or the use of some applications that may have local vulnerabilities. Low confidentiality may occur if information flows to processes with remote vulnerabilities are possible (e.g., through root compromises and storage of corporate data in persistent files).

4.4 Architecture Example

We demonstrate the principles of the architecture in this section. Figure 7 shows selected measurements that we will refer to in the course of this section.

First, we validate the policy enforcement functions on the client. The architecture consists of the 4 major components:

```
#000: BC68F266F8B75E558CD27470B70B53200B480FAB (bios and grub stages aggregate)
#001: A8A865C7203F2565DDEB511480B0A2289F7D035B grub.conf (boot configuration)

#002: 1238AD50C652C88D139EA2E9987D06A99A2A22D1 vmlinuz-2.6.5-bk2-lsmtcg (kernel image including personal firewall and IMA)

#003: 84ABD2960414CA4A448E0D2C9364B4E1725BDA4F init (first process)
#004: 9ECF02F90A2EE2080D4946005DE47968C8A1BE3D ld-2.3.2.so (dynamic linker)
#005: 336536B0E22FF762BB539D7FCB7CD283D4622342 libc-2.3.2.so

...
#439: 2300D59E0AB01A6B9D203CE2A7655177E6247882 persfw_user (client policy agent)
...

#440: BB18CB801C9D27E255C209CB56A47C9EA9CBDD12 libpdauthzn.so (policy client shared libraries)
#441: D12D96BAA8D148BC3C8DF0F3B75859B425A829EE libpdcore.so
#442: 99406B21398D0E2FD88943725E3CC3F9ECD72C49 libamaudutl.so

...
#453: DF541AEDFECB35116808306E89C05591E3ABE160 local.conf (policy agent induces measurement of its configuration file)
#454: 6AC585D072AC9F32AC9CDF8698CAA004AA6DC781 authzn_persfw.db (policy agent induces measurement of its database replica)
...
```

Figure 7: Exemplary set of client measurements $E(client)$.

(i) the personal firewall that is integrated into the Linux kernel ($vmlinuz - 2.6.5 - bk2 - lsmtcg$, c.f. #002 in Fig 7) and that controls the network traffic of the client, (ii) the policy agent ($persfw_user$, c.f. #439 in Fig 7) that retrieves authorization decisions from the policy database, (iii) the configuration file of the policy agent ($local.conf$, c.f. #453 in Fig 7), and (iv) the local copy of the policy database ($authzn_persfw.db$, c.f. #454 in Fig 7) that is replicated with the corporate master data base when the policy agent starts up on the client. We instrumented the policy agent to induce measurements on the configuration file $local.conf$ and the policy data base file $authzn_persfw.bd$ before loading and using them. The measurement architecture offers a simple user-space measurement macro for this. Every slight variation of these files (program version differences, policy changes in the data base, changes in the configuration file) will be reflected by differing measurement values. The VPN server compares now measurements #002,349,453,454 to a set of known and trusted fingerprints on the VPN server these programs.

At this point, we know whether these programs are authentic and configured according to the corporate policy. What we do not know is whether they are actually running or not (they could have run earlier and exited). However, we know that the kernel is running and from the kernel measurement, we can conclude whether the personal firewall is actually installed and thus filters all network traffic. The personal firewall by default (hard-coded) allows only traffic between the client and the VPN policy server that supplies replication of the policy database file.

Authenticity Evaluation of the Client The following client properties are sufficient to protect from external attackers trying to hijack the remote access connection (security goal SG1, c.f. Attack) and thus from masquerading an authorized user: (a) all network traffic must go through the personal firewall, (b) as long as the client is connected, the personal firewall must only allow traffic between the client and the Intranet, (c) we assume no attackers inside the Intranet, which could establish a control flow in the client through the remote access tunnel.

We choose that the kernel enforce properties (a) and (b) because the kernel can be configured (and this configuration validated) not to support hardware interfaces through which attackers could gain access to the client while bypassing the policy enforcement architecture. We use the kernel image measurement and compare it with a known set of SHA-1 values of kernels exhibiting images with different (meaningful) configurations. Table 1 shows kernels with different configurations. If our client kernel measurement equals one of the hashes in the table, then we assign to it the properties that we induce from its integrity (i.e., authenticity implies that kernel integrity ensures that the code running is as expected) shown in the rightmost column.

#	Kernel(SHA-1)	Configuration	Property
1	123AD...A22D1	IMA, Modem, Wireless, Onboard-Ethernet, Persfw Netfilter, IPSEC, no other communication, modules support	medium Authenticity
2	E1F98...34AA1	IMA, Onboard-Ethernet, Persfw, Netfilter, IPSEC, no other communications, no modules support, no serial port support	high Authenticity
3	AA424...4131B	IMA, Wireless, Bluetooth, USB-networking Infrared, Persfw, Netfilter, IPSEC	low Authenticity

Table 1: Properties Based on Kernel Configurations

Kernel #1 is classified medium because it supports Modem connections, which can use non-IP communication and thus bypass the IP netfilter-based firewall. Additionally, this kernel could load kernel modules that support other IP

networking protocols (which would then be detected by the integrity heartbeat). Kernel #2 receives high authenticity because the only external network communication is completely controlled by the IP netfilter-based personal firewall. Additionally, it does not support features over the serial line, such as a wireless keyboard, which could be abused by nearby attackers to issue unauthorized keyboard commands on behalf of the user. The remote access VPN server only accepts properly received IPSEC packets that are authenticated against the remote client and the the remote client only accepts properly authenticated packets from the VPN server. Thus, external attackers cannot control the client during its remote access sessions. Kernel #3 cannot provide enough isolation to prevent potential attackers from leveraging non-IP interfaces (which are not controlled by the netfilter and thus not policied by the personal firewall), so it cannot control information flows as necessary on the client. We can consider only kernels that support the Integrity Measurement Architecture (IMA [5]) because this is a preliminary to determine the client's integrity.

We now combine the integrity level of the remote client with the authenticity level derived from its kernel image and conclude that exemplary kernel #2 on a remote client with integrity validation of high (no unknown, no malicious, no vulnerable software) ensures that the remote client satisfies security goal SG1 and thus requests from this client can be trusted to have actually been initiated by the user logged into the remote client and authenticated against the VPN server.

Confidentiality evaluation of the client. Given the assurance of the client system integrity, we now continue to create a policy that protects the confidentiality of corporate data that is released to such a client system. We approach this goal by confining data that is retrieved from the Intranet to the client system's current boot cycle. Within this boot-cycle, we ensure that this data cannot leak through client interfaces (e.g., USB, WIFI, Serial Port) or be carried over to other (less restrictive) boot cycles via file systems or other persistent storage, c.f., Figure 8.

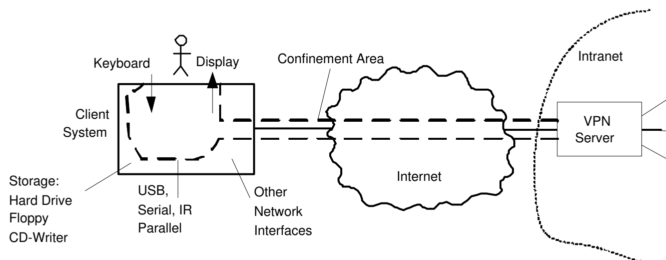

Figure 8: Confining Data on the Remote Access Client

To achieve confidentiality, we require the client to exhibit the high integrity property, which implies strong policy control on the client's communication interfaces. Consequently, a kernel entrusted to enforce confidentiality properties on data will have a configuration as shown for kernel #2 in Table 1 and cannot support any read/write file systems on persistent client storage. We configure the client to mount file systems over the protected remote access link from the Intranet. Additionally, we configure the client to use a RAM disk for its root file system, which supports just enough functionality to run a Gnome Desktop and a web browser and the remote access policy architecture. This web browser can then be used as the client interface to access confidential data inside the Intranet.

At the time the remote access client disconnects from the Intranet, it must reboot in order to clean the confidential information from the memory. The personal firewall is configured not to re-open the network interface after a client disconnects from the corporate VPN until the client reboots if it has accessed services requiring confidentiality properties. We define a list of hash values of kernels that are accordingly configured and compare the measured kernel hash of the client system to compare it against this list in the same way as it is shown for integrity in Table 1.

Some client interfaces however, such as the display and keyboard, must be allowed in order to render the client usable. Whether sound is to be allowed or not is an issue of corporate policy. Attacks through these interfaces are excluded from our attacker model as stated in Section 2.

We envision more flexible operation of the client if the kernel provides an encrypted file system, the key of which is only available from the policy agent after it has verified the system (the capability of *sealed storage* with the TPM security chip could be used as well). As we consider the best protection is not to store the data at all persistently outside the Intranet, we consider only kernels without any local persistent file system (ext3, ext2, etc.) as of high confidentiality. We consider kernels with local encrypted file systems as medium confidentiality (because many more things can go wrong trying to securely store the file-system encryption keys) and other kernels as low confidentiality.

5 Implementation

This section describes the implementation of the policy enforcement prototype. First, we describe how we integrated remote access service policies into Tivoli Access Manager. Then we describe our implementation of the remote access

policy enforcement. Finally, we sketch how remote attestation is implemented on our remote access client prototype.

5.1 Policy Integration

We build our corporate access control using the Tivoli Access Manager (TAM) [6]. TAM is a centralized server where employee identities and their access rights are stored. Access rights are specified as permissions for subjects to perform operations on objects.

We use the identity of the user logged into the remote client to find subject identity information (i.e., TAM user id, member group ids). In our prototype, remote users are allowed to access a service if they have execute permission ('x') on the object. Objects are identified by a type of service (HTTP, ssh, etc.) combined with the service direction (incoming, outgoing).

Permissions are stored in Access Control Lists (ACLs) attached to the service object and are extended by additional security property requirements using TAM extended attributes of the ACL (e.g., required client security properties, transport security on the remote access tunnel, or restrictions to specified server IP addresses). Access of a user id of a remote client (subject) to an Intranet Service (object) is permitted, if the ACL attached to the Intranet Service object allows the subject to execute this object and if additional security properties as specified in the extended attributes of the object hold (c.f. Figure 9 and Table 10).

Using this approach, we define policies to (i) allow service-specific policies that take into account the service direction (incoming/outgoing from the remote access client's perspective) and (ii) attach additional security requirements (e.g. client and packet constraints) to this service's specific policy. For this purpose, we define a new objectspace in Tivoli AM and call it "remoteAccessPolicy" and populate it with the supported services. An objectspace is basically a directory structure, where each node and leaf represents a virtual resource to which access control lists can be attached. Figure 9 shows the object space tree for our remote access policy. Figure 10 shows attribute names and values, as well as in the rightmost column the enforcement entity: packet-related constraints are enforced by the firewall, client-related authenticity and confidentiality constraints are enforced by the client policy agent, and general client-integrity constraints are enforced by the VPN server.

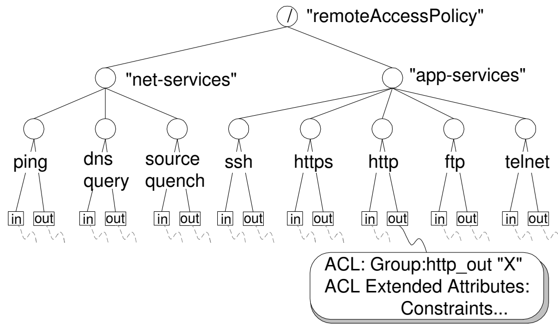

Figure 9: Remote Access Policy Object Space

Extended Attribute Name	Value	Constraint
ServerPort	80	packet (FW)
TransportProtocol	TCP	packet (FW)
ServerIP	10.9.*.*	packet (FW)
TransmissionSecurity	SSL	packet (FW)
TransmissionSecurity	IPSEC	packet (FW)
MinClientIntegrity	Medium	client (VPN)
MinClientAuthenticity	Medium	client (PA)
MinClientConfidentiality	NONE	client (PA)

Figure 10: Extended Attributes Example: http_out

The authorization request for a user Mycroft would look as follows: aznDecision(cap, '/remoteAccess Policy/app-services/http/out', 'x'). *Cap* comprises the capabilities of Mycroft (IDs of groups of user Mycroft), the next parameter denotes the service and type (virtual resource object), the third string specifically asks for "eXecute" permission. The capabilities of the user are acquired by the authorization agent once throughout its initial binding to the authorization replica and re-used as long as the client's user does not change. The aznDecision call returns either *accessdenied* or *accesspermitted*. If access is permitted, then it returns also the extended attributes connected to the ACL that was used to determine the authorization result.

In our example, if the user at the remote access client is not member of the 'http_out' group, then the policy client receives an 'access denied' response. Otherwise, it receives back the 'access permitted' including the additional constraint attributes, implemented as a list of String-pairs. These extended attributes are shown in Table 10 and they refine the access constraints for the outgoing HTTP service: we allow outgoing HTTP 'http_out' outgoing (from a client perspective) for members of the 'http_out' group to all servers in the subnet 10.9.*.* if the remote access tunnel is either SSL or IPSEC protected. These constraints are enforced by the personal firewall (FW). Additionally, the client must exhibit at least medium Integrity level, which is checked by the VPN server, and at least medium Authenticity level, which is checked by the policy agent (PA) locally on the client. This example shows, how the corporate remote access policy manages firewall rules and constraints for remote clients that are then enforced by the personal firewall as described in the next section.

As an example, to allow a user with ID Mycroft outgoing HTTP, PING, DNSLookup, and SSH, we merely need to put user Mycroft into the groups *http_out*, *ping_out*, *dnslookup_out*, and *ssh_out*. To require at least medium authenticity properties of the client (c.f., Section 4.3) for outgoing HTTP traffic, we add an extended attribute pair

"MinClientAuthenticity, Medium" to the ACL with the name *http_out*. We name the ACLs the same way as the service objects because the specified extended security constraints bind them to a specific service.

5.2 Personal Firewall and Policy Agent

The personal firewall and the policy agent are the major policy enforcement functions on the client (c.f., Figure 5). The client security constraints are evaluated in the policy agent, if applicable. The remaining packet-related constraints as well as the overall result of this access control decision or a denied result are communicated to the personal firewall and enforced there locally.

We integrated the *personal firewall* into the Linux kernel. It first registers a queue handler for INET packets with the kernel netfilter [27]. It then registers to the *NF_IP_LOCAL_IN* hook (delivers all incoming IP packets) and *NF_IP_POST_ROUTING* hook (delivers all outgoing IP packets) a simple function that immediately re-inserts all intercepted packets with the QUEUE verdict. This way, we receive them immediately back into our registered queue handler and can be sure not to block the kernel or network traffic if we have to delay some packets to resolve access control decisions that involve policy lookups in user space. The queue handler then enforces the remote access policy on each data packet before re-inserting it with DROP or ACCEPT verdict according to the access control decision.

We implemented a stateful packet filter that keeps session information on TCP packets and thus requires access control only on the the TCP connection setup packets. For packets that belong to an existing TCP session, we only check whether the initial client security properties changed. For packets requiring access control decisions, the personal firewall will create a policy query request and send it to the policy agent through the /dev/policy interface (returning an earlier read request by a polling policy agent thread). It queues such packets in order and handles them as soon as the access control information is received from the policy agent, which writes the result and extended requirements if applies back into /dev/policy. The personal firewall keeps a cache of earlier access control decisions and tries first to resolve the query locally with the cache. Usually, the cache is very effective because there are only a few services used by typical clients. Then, this cache can be used until the policy changes, which is indicated by the policy agent.

We have implemented some global rules into the personal firewall, such as to drop all packets that are not initiated by the client or the VPN server end of the remote access tunnel. Consequently, the client can communicate over IP only through the tunnel. We also limit traffic to the remote access client and the corporate policy server for the protocol and port used by the policy agent to replicate the policy data base. To avoid service interactions, we switched off any IPTables kernel support that could compete with our registration of the personal firewall with netfilter hooks.

The *policyagent* uses the authorization interface [28] of Tivoli Access Manager to resolve authorization requests. This interface is initialized by binding to it with the userid and password of the remote access user against the Tivoli Authentication Server. It returns this user's credentials back. These credentials are then used for subsequent authorization requests as described in Section 5.1. We use the local mode of the authorization interface that creates a local replica (about 600 KBytes for a tree as outlaid in Figure 9) of the AM master policy data base on the client. Authorization requests are resolved locally on the client by the authorization library code, which improves the performance considerably (about 11,000 authorizations / second). Tivoli Access Manager allows either periodical polling of the master policy data base for updates or to receive a notification in case of master policy data base changes. We chose to implement the latter because we do not want thousands of remote access clients to poll the corporate policy server unnecessarily. This configuration is subject to notification deletion attacks; however, as the policy data base is measured and validated, the VPN server can detect out-of-date policy data bases on clients.

The user space policy agent polls the "/dev/policy" character device file for authorization queries from the personal firewall. Any received request is then translated into a format that is understood by the Policy Access Manager authorization service (see Section 5.1). The translated request is fulfilled by the local replica of the master data base that returns the authorization decision. The authorization agent process translates the decision into a data structure that is written back into the "/dev/policy" device file. An included transaction number (strong monotonically increasing) ensures that request and response are correctly related to each other. Waiting queues ensure smooth operation of the kernel. The policy cache size is set to 15 entries, thus equivalent authorization requests (same service, direction, and user) can usually be answered without user space interaction.

The policy agent determines the client security properties by reading the kernel measurement from the measurement list and comparing it to known SHA-1 values with known properties stored in the local.conf configuration file, which is measured and validated by the VPN server. Measuring this file at the client and validating it at the VPN server ensures that the policy agent and the VPN server agree about the corresponding assignment of kernel SHA-1 value and client security properties. The policy agent assumes the measurement list correct and expects the VPN server to disconnect the client if the client integrity becomes compromised. If it applies, the policy agent then validates the required client security properties with the ones derived from the current kernel measurement. If the requirements are satisfied (given properties dominate the required properties), then the general verdict (accept) and remaining packet-related constraints (e.g., server IP constraints, transport security constraints) are returned for enforcement to the personal firewall by writing it into the "/dev/policy" character device file.

We implemented the policy agent on the remote access client system running Redhat Linux 9.0 with a 2.6.5-bk2 kernel. The policy agent agent comprises about 1100 lines of code (Loc) not including the authorization library that is part of Tivoli AM. The kernel part comprises about 1800 Loc including cache handling, connection tracking, packet classification, and authorization retrieval from user space.

5.3 Remote Attestation

We used a Linux Security Modules version of the Integrity Measurement Architecture (IMA) as described in [5]. We wrote daemon process (1300 Loc) that allows local parties to easily obtain a new signed TPM aggregate (Quote) while submitting a 120bit random number with the request. It is based on a public TPM library [29]. The remote attestation prototype determining the client integrity property is implemented on the server system using Java. On the remote client side, we have a small attestation server that accepts remote attestation requests –including a 120bit random number– from dedicated machines (modeling the VPN server) and returns (i) the current kernel measurement list that is obtained through a local proc interface and (ii) the signed current TPM measurement aggregate including the random number from the request. On the VPN server side, we maintain a data base of SHA-1 values of known programs annotated with attributes: *trusted*, *malicious* (which currently includes the "uncontrolled" programs, see Section 4.1), *vulnerable*, *lowvulnerable*. This allows us to determine the client's integrity tests as described in figure 4. The data base has about 25,000 entries hashed on the SHA-1 value, which makes fast lookups possible. A client running Redhat Linux, the Gnome Desktop, Web Browser, and terminals accumulates about 400-550 measurement entries. As any executable is only represented once in the measurement list, this number does not depend much on the uptime of the client as long as the executables are not altered.

6 Analysis

Policy decisions and enforcement overhead. The remote access client runs Redhat Linux 9.0 on a 2GHz IBM Thinkpad T30 that includes a TPM security chip. We used a 2 GHz Netvista Linux workstation as the VPN server and a 2 GHz Windows 2000 Server for the Tivoli Access Manager run-time suite including LDAP and Authorization server. Table 2 shows the network round-trip delay through the personal firewall performing policy decisions on each data packet.

The personal firewall needs to lookup the policy for every TCP connection setup packet and for every UDP packet. Using the policy cache means that authorization decisions are stored; thus, successive policy lookups are resolved in the kernel cache and don't involve interaction with the user space policy agent. This holds until the policy changes or the client security properties change. The kernel cache and existing TCP connection tracking entries are marked dirty in this case. Dirty-flagging the kernel cache is necessary because the policy client makes the client-based access control decision when delivering the authorization request to the personal firewall. The personal firewall then makes the packet-based access decisions, assuming that –for cached authorization results– the user space client-based access decision did not change. The dirty-flagging of existing TCP connections is necessary because by default packets belonging to existing TCP connections inherit the access control policies of the TCP connection setup packets.

#	Configuration	RT	Overhead
1	Reference UDP	162 μs	0%
2	Reference TCP	200 μs	0%
3	No Policy Cache UDP	1087 μs	570%
4	No Policy Cache TCP	209 μs	5%
5	Policy Caching UDP	180 μs	11%
6	Policy Caching TCP	208 μs	4%

Table 2: Prototype Performance (PL 100 bytes)

The reported times in Table 2 are averages of 100,000 round trips times. Line 3 shows the overhead for UDP round-trips when the policy cache is disabled compared to the baseline value reported in Line 1. In the case of a policy cache miss, the user space authorization agent is asked for the authorization decision twice (for outgoing and incoming UDP packets). As the authorization decision is the main overhead, this shows that a single user space authorization decision induced by the personal firewall adds about $(1087 - 162)/2 = 463\mu s$, which translates to about 2000 authorizations per second. Line 4 measures only the TCP connection tracking overhead added by the policy agent because authorization decision is made only once at TCP connection setup time (which is not included in the measurement). Line 5 shows that using the policy cache, the overhead for UDP traffic is only about 11%. Most of the performance-critical traffic will be TPC traffic, which yields about 4% overhead. The inital overhead to resolve authorization for TCP setup packets is equivalent to the UDP overhead (5% in the case of a cache hit).

The initial binding of the authorization client to the authorization service consists of the local replication of the

master authorization data base (600 KBytes in our example) on the Access Manager server as well as acquiring the remote user's capability set. This one-time initialization at the startup of the policy agent takes about 2 to 3 seconds. Thus, even in cases where the cache is not yet loaded with the necessary policy decision, our prototype delays the packet for only a very short time, which should be invisible to the user. Even this could be eliminated by cache-preloading as the general remote access policy is expected to be fairly static.

Deciding the client security properties by the policy agent on the client involves comparing the local kernel measurement to a list of known kernel measurements and to read the properties of this kernel from the configuration file; the related configuration file is measured and loaded by the policy agent at startup and validated by the VPN server throughout the client integrity validation. As there are only a few kernel hashes to be considered, this operation does not add visible overhead as our set of known kernels is pretty small. Thus, the overhead of the policy client enforcing the client security constraints during the service access control are negligible. The overall overhead for service access control on the client is dominated by the personal firewall packet-related access control and by the time to lookup the policy in case it is not yet in the kernel cache or the kernel cache is marked dirty because of policy changes.

In summary, the maximum delay for UDP or TCP connection setup packets (e.g., delay when starting SSH) is about 463μs. The average overhead afterwards is about 4%. The initial delay can be mostly eliminated by pre-loading the cache with the most likely needed authorization decisions. Our experiences of the prototype system confirm that the policy agent does not impose significant overhead on client systems.

Overhead of measuring clients The general overhead of the integrity measuring architecture is very low [5]. We added the integrity-heart-beat, which requires signing of the current TPM aggregate and delivering it together with the client's measurement list to the VPN server. Full initial measurement validations incur about 3 seconds round-trip delay. This includes sending the request from the VPN server to the client, receiving back the signed aggregate and a list of 400-500 measurements, validating the measurement list against the signed aggregate, evaluating every individual measurement against the database that identifies measurements as known, malicious, vulnerable, or lowvulnerable, and finally inferring the client integrity properties as described in Figure 4. The related processes are not optimized and use many string operations in Java as well as XML transport encapsulation. A subsequent evaluation needs only to retrieve a new signed aggregate (1/5 second) and the newly added measurements plus the TCP round-trip between the VPN server and the remote access client. The database size (flat file) contains about 25,000 lines, which corresponds to 25 thousand entries (known hashes of libraries, binaries and bash command files) of a typical Redhat 9.0 system that includes a Web Browser, tools, and web server etc. These entries are sorted by the fingerprint value and thus can be easily searched.

7 Future work and Conclusion

We have designed and implemented a novel access control architecture that enables corporations to verify client integrity properties and establish trust into the client's policy enforcement before allowing clients remote access to corporate Intranet services. To this end, we have shown how to (1) determine the integrity level of the client system based on the code running on the client; (2) determine whether to trust this client to enforce information flow controls necessary to make such integrity assumptions about client; (3) determine whether additional security properties, such as confidentiality, need to be enforced by the client and whether the kernel supports these; (4) integrate the integrity of the client into the remote access control policies governing the client's access to the corporate servers; (5) enforce this policy on remote access clients and the VPN server. Finally, we have introduced an integrity-heart-beat that enables the VPN server to track changes in the remote client's security properties (i.e., sense relevant changes in the client's software stack) and implement resulting policy changes. We have implemented a Linux 2.6 prototype system that utilizes the TPM measurement and attestation framework, existing Linux network control (Netfilter), and existing corporate policy management tools in the Tivoli Access Manager to control remote client access to corporate data. This prototype illustrates that our solution integrates seamlessly into scalable corporate policy management and introduces only minor performance overhead.

References

[1] Trusted Computing Group. *Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands*, October 2003. Version 1.2, Revision 62, http://www.trustedcomputinggroup.org.

[2] T. Frazer. LOMAC: Low water-mark integrity protection for cots environments. In *IEEE Symposium on Security and Privacy*, May 2000.

[3] J. P. Anderson. Computer Security Technology Planning Study, 1972.

[4] G. Karjoth. Access Control with IBM Tivoli Access Manager. *ACM Transactions on Information and System Security*, 6(2):232–257, 2003.

[5] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Thirteenth Usenix Security Symposium*, August 2004 (Accepted for publication).

[6] IBM Tivoli. IBM Tivoli Access Manager for e-business. http://www-3.ibm.com/software/ tivoli/products/access-mgr-e-bus/.

[7] Mark Corner and Brian Noble. Zero-interaction authentication. In *ACM MOBICOM 2002*. MOBICOM, September 2002.

[8] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proceedings of the ACM Computer and Communications Security (CCS) 2000*, pages 190–199, November 2000.

[9] D. Eastlake and P. Jones. Secure Hash Algorithm 1 (SHA1), September 2001. Request for Comment 3174.

[10] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[11] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. *IEEE Computer Society Conference on Security and Privacy*, pages 65–71, 1997.

[12] S. W. Smith. Outgoing authentication for programmable secure coprocessors. In *ESORICS*, pages 72–89, 2002.

[13] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[14] IBM PCI-X Cryptographic Coprocessor, 2004. http://www-3.ibm.com/security/cryptocards/html/pcixcc.shtml.

[15] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, pages 1037–1041, 2000.

[16] J. Dyer, R. Perez, R. Sailer, and L. van Doorn. Personal Firewalls and Intrusion Detection Systems. In *2nd Australian Information Warfare & Security Conference (IWAR)*, November 2001.

[17] J. Molina A. Mishra and W. Arbaugh. The co-processor as an independent auditor. Available at http://www.missl.cs.umd.edu/komoku/documents/coauditor.ps.

[18] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. In *Tenth ACM SIGOPS European Workshop*, September 2002.

[19] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *ACISP 2002*, LNCS, pages 346–361. Springer-Verlag, July 2002.

[20] B. A. LaMacchia. Next-generation secure computing base (NGSCB), April 2003. RSA Conference 2003, San Francisco.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 193–206, 2003.

[22] D.F. Ferraiolo and D.R. Kuhn. Role based access control. In *15th National Computer Security Conference*, 1992.

[23] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. In *IEEE Computer*, volume 29(2), pages 38–47. IEEE Press, 1996.

[24] D. E. Bell and L. J. LaPadula. Securecomputer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

[25] D. R. Wilson D. D. Clark. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, 1987.

[26] S. M. Bellovin. Distributed Firewalls. login, November 1999.

[27] The netfilter/iptables project, 2004. http://www.netfilter.org.

[28] The Open Group. Authorization (AZN) API – Technical Standard. http://www.opengroup.org/products/publications/catalog/c908.htm.

[29] IBM Watson Research - Global Security Analysis Lab: TCPA Resources, 2003. http://www.research.ibm.com/gsal/tcpa/.