T1B2 1555
Proceedings of the 2002 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 17–19 June 2002

Secure and automated software updates across organizational boundaries

Lawrence Teo, Yuliang Zheng

Abstract— **The number of systems being compromised and broken into is increasing everyday. One major reason why this happens is because software components in these systems are not updated regularly, or not fast enough after a security alert is issued. A major contribution of this work is to propose a virtual network that can be used to update systems with potentially vulnerable software in a high-speed and secure manner across organizational boundaries. This will significantly reduce the risk of system compromise. Two technical challenges that we address are the security of the network, and the diversity of different platforms in an inter-organizational environment. The network makes heavy use of public key infrastructure (PKI) to insure the security of the network and XML technologies to distribute updates. We have developed an experimental prototype, and our current tests show favorable and promising results.**

I. Introduction

As organizations around the world become more interconnected with one another, the amount of critical and sensitive information being stored on their networks and sytems is increasing rapidly. It is very common for organizations to depend on this information for their day-to-day operations. For example, a customer database ensures the daily operations of an e-commerce site. Up-to-date information about the status of nuclear plants can help the energy company to predict and prevent any mishaps. This information is not confined to just large company servers and mainframes. The proliferation of mobile and wireless devices such as handhelds and notebook computers have also encouraged their users to store critical information on them for convenience.

It is therefore extremely vital that systems, networks, and devices that store these information are configured to be as secure as possible to reduce the likelihood of them being compromised. The loss or theft of the information may lead to huge financial losses, damage to critical infrastructure, and even loss of lives.

However, in recent years, there has been an alarming increase in the amount of intrusions into such computer systems and networks. Despite the availability of advanced security technologies to systems administrators, systems and networks are still being compromised. We believe there are two main reasons for this. The first reason is the rapid pace in which the attacker community can spread attacking tools and information about system vulnerabilities. For example, Northcutt [1] mentions that the attacker community has very effective online mentoring programs! Attackers frequently probe the Internet for vulnerable systems. This leads us to the second reason. The software components in these systems and networks that host critical information are frequently not updated on time, rendering them vulnerable to such attackers. There are many reasons for this. Systems administrators may be too busy to update the systems or keep themselves informed with the latest security alerts. They may also be inexperienced with the systems and not know how to patch them. It is also possible that such systems are being taken care of by average users who are not technically skilled or aware of security issues. Many of these intrusions could have been prevented, if only systems were updated on time.

Although we have discussed two reasons why systems are still being compromised, we wish to stress that there are many other reasons as well. For example, poorly written software often contain security holes. In contrast, it is also possible that well-written Commercial Off The Shelf (COTS) software may be improperly installed or misconfigured. Both cases lead to vulnerable systems.

A very visible evidence of these trends can be seen in the everyday problem of website defacements. Although these defacements are usually performed by intruders who are just looking to "vandalize" websites, and not steal any information, it does show that there are many systems on the Internet that are vulnerable and not patched on time. In fact, intruders sometimes offer to help the website administrator to patch their compromised systems! There may also be many cases where the intruder is not necessarily a human being, but instead may be a program, like a distributed denial of service (DDoS) tool, an email borne virus, or worm. For example, the recent Code Red and Nimda viruses targeted unpatched Microsoft IIS servers.

We believe that it is of paramount importance that systems be updated as frequently as possible, whether they are mainframes, servers, workstations, handheld devices, or public kiosk terminals. In this paper, we propose a secure and automated software update distribution network that can be used to update any software on any device and platform, in order to minimize the risk of critical information being compromised. Instead of relying on the systems administrator, our network automatically delivers

L. Teo and Y. Zheng: Laboratory of Information Integration, Security, and Privacy (LIISP), University of North Carolina at Charlotte, Charlotte, NC. {lcteo,yzheng}@uncc.edu

ISBN 0-7803-9814-9/$10.00

the updates to the systems whenever they are available and patches them automatically. We view this network as a complementary technology to other security solutions such as firewalls and intrusion detection. This network is part of a larger ongoing research project, which involves the development of an Internet-scale intrusion detection infrastructure. More details of the network and this infrastructure will be presented in the full version of this paper.

In this paper, we will first discuss the objectives and goals of our proposed network. We then provide a background overview of other work that are related to this project. Next, we present the architecture of our network, and a section on security and privacy issues. This is followed by a description of our experimental prototype and its current state. We then discuss the tests that we have done and our results. Following this, we suggest some possible future work and give our final comments in the conclusion.

II. Objectives and Design Goals

In this section, we describe our objectives, design goals, and assumptions regarding our proposed network. Traditionally, updating systems followed a passive paradigm, from the viewpoint of the vendor. Systems administrators would download updates from the vendor, and apply them manually. What we are proposing follows an active paradigm. The network actively pushes these new updates to registered systems, which are immediately updated.

Our objective is to build a secure virtual network for distributing software updates to any device on any platform in a *cross-organizational* manner. "Cross-organizational" means that the network is not be dependent on any one organization or vendor. The network is capable of delivering software updates from any vendor who follows our pre-defined standards.

The design goals of our network are as follows:

1. **Security.** Since the network involves automatically updating systems with little or no user intervention, it is extremely important that the updates are not tampered with, or modified in transit. Users should be confident that their systems are being updated with trusted and authenticated updates that have been audited and verified.
2. **Platform independence.** The diversity of platforms and operating systems being used by organizations, companies, academia, governments and individuals warrants the need for platform independence. A main goal of this project is to promote openness and maintain a uniform method of updating systems, regardless of the platform.
3. **Scalability.** The network should be highly scalable. In fact, it is intended for this network to be able to support a large number of users, say, in the order of hundreds, thousands, or even millions of users in the future.
4. **High-speed response.** As stated earlier, a major reason why intrusions occur is because software is not updated on time. To minimize the risk of systems being compromised or damaged, the network has to deliver software updates to registered systems as soon as the updates are available. In other words, the network has to propagate its updates to all affected system in as high speed as possible.
5. **Privacy.** Since our network works with delivering software updates, the systems registered with the network will have to send their system configuration to the supernodes. In such situations, it is natural for certain organizations to be concerned about the nature of the information being shared, and what it is being used for. One of the design goals is to address such privacy concerns. Another issue about privacy is that companies may not want to disclose their participation in this network. Therefore, our network should provide the option for them to choose whether they want to disclose their participation or not.
6. **Usability.** The network has to be usable and practical. Therefore, we have adopted a practical approach in building this network, as opposed to a theoretical, formal approach. Our goal is to see this network be publicly available and usable in the short term.
7. **Resilience.** Since the network supports automatic updates with minimal user intervention, it is vital that it is resilient, robust, and tamper-resistant.

A. Assumptions

Our assumptions that govern this network are as follows. A network of this nature relies on the existence of a trust management framework, such as public key infrastructure (PKI). When PKI is employed, it has to be reliable. Reliability here means that the identities of the various entities in the network can be certified with total accuracy. This implies that the root Certificate Authority (CA) and all CAs under it will enforce stringent procedures to verify the identities of entities who issue certificate requests, before deciding to issue a signed certificate to them.

In order for the PKI and our proposed network to operate, there should be an underlying physical network in place. In this project, it will be a TCP/IP network that is capable of delivering traffic reliably. Our current prototype uses Secure Sockets Layer (SSL) [2] and X.509 certificates to implement the PKI for the project. SSL will be used for authentication and encryption functions.

III. Related Work

Our network uses ideas from software update technologies, distributed intrusion detection systems, and peer-to-peer networks. This section discusses the relevant technologies that are related to our work.

A. Software update technologies

Traditionally, software is updated by manually finding and downloading them from the Internet or through mail, and applying them by hand. As networks become

more interconnected, it became feasible to download sizable updates from the Internet. For example, the Debian GNU/Linux operating system [3] contains a powerful tool called `apt-get` that automatically searches, downloads, and installs any package the user specifies by name.

A novel approach to software updates can be found in the open source BSD UNIX variants. These systems contain a "ports tree", a directory tree of Makefiles and system-specific source code patches. The ports tree works by downloading a package in source code form from the Internet. The integrity of the downloaded package is verified using the SHA1 checksum in the ports tree. The source code package is extracted and patched before being installed.

Recently, Red Hat announced a service which will update customers' software with minimal user intervention. The Red Hat Network [4] allows subscribers to maintain a single, secure connection to Red Hat's servers, and packages can be automatically downloaded and updated on the clients' systems. This is desirable since security holes and bugs can be patched once updates are available. Our work differs from Red Hat's where we emphasize on the security, authenticity and integrity of the updates. This different emphasis is needed because we are working across organizational boundaries and not within a single organization.

B. Distributed intrusion detection systems

While our work is not a distributed intrusion detection system (DIDS), it borrows concepts from DIDSs. The first DIDS, aptly called Distributed Intrusion Detection System [5], consisted of components which collect data and forward them to a central monitor for analysis. While it was very radical compared to other intrusion detection systems (IDSs) of its day, it suffers from a single point of failure.

On the other extreme, there is an IDS called Cooperating Security Managers (CSM) [6] which has components that share equal capabilities. These components forward collected data to one another and maintain algorithms for identifying intrusions using a decentralized approach.

Autonomous Agents for Intrusion Detection (AAFID) [7] is an agent-based DIDS with three types of components. Agents perform data collection and low-level analysis. The results are sent to transceivers, which analyze and pass its results to monitors, the highest-level entity in the hierarchy.

C. Peer-to-peer networks

In the course of this project, we studied peer-to-peer networks like Gnutella and Freenet [8]. The Freenet project is an attempt to create a totally decentralized and distributed network to provide total user privacy, support freedom of speech, and curb censorship. Users can run a Freenet node in the Freenet network, which will use the resources of the node to store other users' files in an encrypted form. Unlike Freenet, our network has a central authority that oversees the updates process and ensures the authenticity of updates. This prevents illegitimate or malicious updates from being introduced into the network.

D. Standardization initiatives

The Common Vulnerabilities and Exposures (CVE) [9] project provides a common dictionary of vulnerabilities which vendors and software developers can use as a uniform base to describe vulnerabilities. When a new vulnerability is discovered, a new CVE entry will be created for it. When a vendor issues an update for a vulnerability, the update may refer to this CVE entry, thus there is a link between the update and the vulnerability. The software updates designed in this project aims to be CVE-compatible.

The Linux Standard Base (LSB) [10] aims to promote a set of standards to increase compatibility between Linux distributions and product vendors. As Linux becomes more popular, LSB will have greater implications in our work.

The National Institute of Standards and Technology (NIST) is working on a PKI interoperability project [11] which aims to bridge the different internal PKI standards among different organizations. Their research focus is to bridge different PKIs, which is potentially useful to address the problem of cross-organizational software updates.

IV. Architecture

In this section, we describe the architecture of our proposed software updates network. The term "network" that is being used here does not refer to a physical network, but rather to a virtual, logical network that operates on top of existing networks. We will be concentrating on explaining the experimental techniques and methods we are using to achieve our objectives and goals, although we do intend to implement the actual network in the future. Currently, we have developed an experimental prototype that demonstrates the core part of the architecture.

We considered different approaches in the process of designing the architecture of this network. With our design goals in mind, our first design decision was to decide whether to use a centralized or decentralized approach to distribute our software updates. A centralized location to distribute software updates would always keep the updates consistent, but it suffers from the single point of failure attack. A denial of sevice (DoS) attack on the central distribution location would render it incapable of sending out updates, and may result in all other systems not receiving the updates on time. Another approach would be a totally decentralized approach, where each node is an equal peer to another. This would solve the single point of failure problem but it has serious drawbacks as well. Since the security, and more specifically authenticity of updates is one of our design goals, nodes which are equal in peer will not know if any new updates can be trusted.

We decided to adopt a few useful peer-to-peer ideas but still maintain a hierarchical architecture in our final archi-

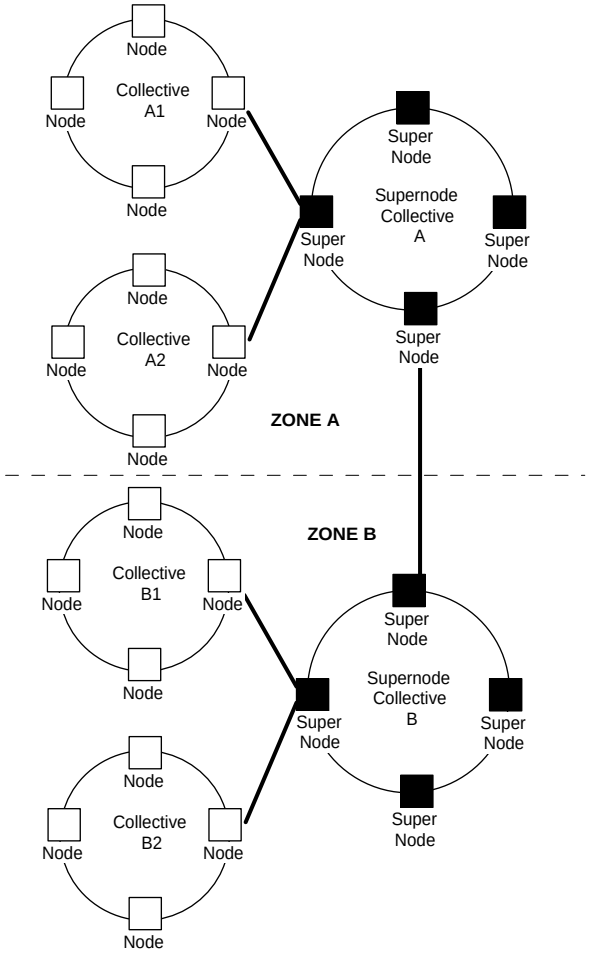

Fig. 1. The architecture of the network.

tecture. Our architecture consists of many entities, called agents, that are equal in rank. A host that runs an agent is called a node. Although the agents are equal in rank, they have the ability to upgrade the nodes into supernodes. Supernodes are in charge of distributing software updates to normal nodes, thus normal nodes will have an authoritative entity which they can trust. The following section defines the terminologies of these entities and a few new ones.

A. Terminologies

To fully understand the architecture, several terms have to be defined first. The following list defines the terms that will be used to describe the network architecture.

Host. A host is the end system which uses the services provided by the network. A host may be a workstation, a mainframe, a server, a handheld, or a kiosk terminal.

Agent. An agent is a background daemon that runs continually on a host. Agents are distributed by supernodes.

Node. A node is a host which is running an agent.

Collective. A collective is a collection of nodes. Nodes in a collective forward information to each other in order to achieve resiliency. If a node goes down, the other nodes will still have its information and operate normally.

Supernode. A supernode is a special node that provides higher-level services to collectives. The supernode is primarily responsible for delivering updates to the nodes.

Supernode collective. A supernode collective is a collection of supernodes, acting in the same way as the collective to achieve resiliency.

Zone. A zone is an area of the network or Internet under the authority of a supernode collective.

Update. In general, an update is a patch or a fix that can be used to correct a hardware or software bug, whether it is a security hole or not.

Certificate Authority (CA). A CA issues signed certificates to agents. An agent with a signed certificate is considered a trusted agent.

Repository. A repository is the storage area hosting the updates. There may be several repositories with the updates on them mirrored with one another.

The architecture of our network is shown in Figure 1. One possible application of this hierarchy would be for an organization to set up its own zone with a central root CA, and have second-tier CAs and supernode collectives in various departments. With this arrangement, organizations can set up their own independent software updates network. Certain organizations, such those in the military and defense, would benefit from this setup, due to their nature and need to have a separate independent network.

B. Agents and supernodes

An agent is a background daemon that is both a client and a server. An agent can transform a host into either a normal node or a supernode.

Supernodes are manually set up to communicate to each other to form a zone. Supernodes are certified by the CA in charge of the zone. There will be one CA per zone. All CAs in a zone are certified by root CA. Agents verify updates using a CA certificate chain list.

When a CA or agent is compromised, its certificate will be added to a public Certification Revocation List (CRL). This will inform the agents that the certificates used by those comprmomised hosts must not be trusted.

C. System Configuration

We view a system's configuration as having five categories: meta information, hardware, operating system, software, and services.

Meta information states what exactly the host is. For example, meta information may indicate that the host is a workstation or a handheld device.

Hardware information will be used when delivering updates that affect hardware, such as device drivers. We also note the processor that the host runs on, its speed, CPU

cache, memory size, and so on. Hardware information will also be important to identify hosts with hardware bugs.

Software describes the vendor of the software installed, their categories (for example, an email client will belong to the category "desktop/email"), and version number.

The *operating system*, being an extremely crucial part of the host, is treated separately from the software.

Services are the type of services that a host provides to external entities (such as HTTP or SMTP).

D. Updates

To deliver software updates efficiently, we have given a software update four different categories.

A *bugfix* is an update that corrects a general error in the program. In this context, the error is not a security hole.

A *security alert* update corrects a security-related bug that may lead to host compromise or a break-in. Examples of such bugs include buffer overflows, format string vulnerabilities, race conditions, and so on.

An *enhancement* adds new features to a software.

Sometimes, it is not possible to provide an update to a piece of software automatically. For example, there may be a bug in the operating system kernel, which would require a reboot. If the host is restarted on a busy server automatically, users may lose their work. In such cases, and all other similar cases, an *Announcement* update is used instead. The Announcement update just provides the systems administrator with information about a potential threat, and how to address it manually.

These four categories of software updates are not mutually exclusive. For example, an update may be both a security alert and an enhancement. Categorizing updates this way should reflect the real world more accurately.

The network aims to be compliant to the Common Vulnerabilities and Exposures (CVE) [9]. The update itself will include a list of entries specifying the CVE entries that it is associated with. This list shows the list of vulnerabilities in the CVE dictionary that the update corrects.

A typical update object has a name, package name, package version, URL, digital signature, a list of CVE entries, and an optional section on special instructions specifying how to run the update. The digital signature is used to verify the authenticity and integrity of the update.

E. Update Retrieval and Application

The architecture provides two methods that can be used by the agent to retrieve updates from the repository. The first method is to have the network deliver the update from the repository to the agents. In this method, the update is retrieved by the supernode, and then sent to all appropriate nodes that have matching system configurations. The advantage of this approach is that the update can be sent securely from the repository to the agent. However, this is not always possible. A commercial software vendor may place licensing restrictions on their updates, preventing them from being stored on third party repositories. A second update retrieval method is provided to address this issue. Since the repository cannot host the actual update, the supernode sends only the URL and a message digest of the update to the agent. The agent will retrieve the update from the URL. The update's authenticity is verified by running the message digest algorithm against the downloaded update, to see if it matches the one provided by the supernode. The update transfer from the vendor's site does not have to be secure, since the authenticity is verified by a positive match between the message digests.

F. Logging and Recovery

There may be times when a faulty update is issued by a vendor, and the update has been applied to various nodes. When such a scenario happens, it would be ideal to have an ability to "go back" to a previous state. This means that we will have to keep backups of old packages and logs of updates, which state which files have changed and at what time. This information can then be used to perform the recovery operation, which is similar to a database `ROLLBACK` operation. The format and content of these logs should be configurable by the systems administrator.

G. Response Mechanism

The administrator is given a choice on how to respond to the receipt of updates. The first method is to have the agent apply the update automatically. Alternatively, the agent can be set up to email the administrator with a notification without actually applying the update. The third option is to simply log the receipt of the update.

V. Security and Privacy

The critical nature of this network requires it to have strong and reliable security. This section describes the security issues associated with the network.

A. Possible Attacks

First, we provide a high level overview of possible attacks from both outsiders and insiders, and what security mechanisms can be used to address them. Some of the ideas discussed here are based on the concepts of survivability.

A possible attack from an outsider may be a DoS attack against the agents. One way of doing this might be to flood the agents or the nodes with traffic. Since DoS attacks are extremely difficult to prevent, one likely way to address this is to allow the collective to recover even though one of its nodes is unavailable. The remaining nodes of a collective may rebuild the ring by bypassing the affected node.

Insider attacks are arguably more dangerous than outsider attacks. For example, an inside attacker may attempt to impersonate another agent. The PKI mechanisms can be used to detect this. An insider may also attempt to

compromise the updates on a repository, and get agents to download modified updates. Additionally, the insider may attempt to compromise supernode agents to send the incorrect message digest to the agents. Agents can detect this behavior by requesting many message digests from different supernodes. If one of them differs from the rest, there is a possibility that that supernode has been compromised.

B. Confidentiality

Like any other system, our network is bound to be a potential target for abuse and attacks. For example, an attacker may observe the transmission of data between the supernode and the nodes. Following this, the attacker can learn whether the organization has updated any system, or how frequently this is done. This intelligence can then be used to launch a more targeted attack. One approach to prevent such traffic analysis is to anonymize the traffic between collectives and supernode collectives in a broadcast fashion. We plan to examine this further in the future.

C. Integrity and Authentication

In order for software updates to be trusted, they need to be verified that they are from a reliable source, and that they have not been modified in transit. A software update that is modified may have been illegally modified as a Trojan horse for malicious reasons. As such, it is very important for these updates to be verified.

There are two entities that need to be authenticated: the supernode agent that delivered the software update, and the update itself. There are several ways to achieve this. One way would be to use a tool like Pretty Good Privacy (PGP) ([12]) to discern whether the downloaded update is from the trusted source or not. However, PGP is not sufficient for hosts with critical information, because PGP leaves it up to its users to decide whom they trust. With critical information, it is not sufficient to rely on users alone to trust and verify identities of these entities.

Another way of addressing this is to hardcode the addresses and public keys of trusted supernodes onto the agents themselves. Thus, software updates downloaded from supernodes can be verified with the hardcoded public key. However, this approach is not scalable. Adding new supernodes to a collective would require every agent registered with that collective to be updated with the new supernodes' addresses and public keys. Since we are working with cross-organizational networks, scalability is an important design goal. Thus, hardcoding is not an option.

Our approach is to use a public key infrastructure for authenticating supernodes, updates, and agents. CAs can issue certificates to these entities, upon verifying their identities using a stringent set of procedures. Another advantage of using PKI is that it is highly scalable. We can hard code the public certificate of the root CA onto the agents. When there is a new supernode collective which is certified by another CA, the agent can look at the certificate list to verify whether the supernodes are trusted by the other CA, which in turn should be trusted by the root CA. The idea is to have a CA in every zone, all under the root CA.

To preserve the integrity of updates, it is important to protect the tools used to apply the updates. For example, on a Red Hat Linux system, these tools would be binaries like `rpm`, a package management tool, and `wget`, a network file retrieval tool. It is extremely important that these files are kept secure so that they are not compromised. If compromised, an attacker may replace them with malicious Trojan versions, thus giving the user a false sense of security. Therefore, these binaries should be kept in a separate secure area, such as a read-only medium. The paths to these binaries and their message digests can be hardcoded into the agent. Every time the agent runs them, the agent must first check their integrity by calculating their message digests and matching them with the hardcoded values.

D. Availability of Updates

Instead of storing updates on only one repository, updates may be mirrored on several repositories to limit the impact of DoS attacks. The contents on these repositories are synchronized periodically to ensure consistency.

E. Privacy

Since privacy is an important issue in this network, agents will be available to systems administrators in a way that the administrator will have full control over the agent. There are two methods we can use to do this. Assuming that there is total trust between the administrator and the agents, the agents may be distributed in binary-only form. Since total trust is unlikely, an alternative would be to distribute the agent in source code form. However, sometimes it may be undesirable to provide the entire package in source code form. For example, the public certificate of the root CA needs to be protected from modifications.

VI. Experimental Prototype

We have developed an experimental, proof-of-concept prototype of the virtual network, which implements the core subset of the features of the architecture. The prototype was developed primarily using C++ on the Linux platform. XML was used to represent the system configuration such as the system hardware, software, and operating system version. In addition, XML was also used as the common format to transfer data between agents.

Although a major design goal in this project was platform independence, we chose ANSI C++ over the more obvious choice of Java because in our opinion, Java is still not fast enough. Also, eventually we may need to use raw sockets in this system, which Java does not allow.

To achieve confidentiality in transferring of information between nodes, we incorporated SSL [2] using the OpenSSL

library. XML support was achieved with libxml++.

An agent is set up as a background daemon that listens for connections. Depending on the command-line parameters used to load the agent, it will make the host act as either a normal node or a supernode.

A. Normal node

Before a normal node agent can be started, there are two important tasks that have to be performed first. First, the node has to be certified by the CA first. This is done by sending a certificate request to the CA, who will then sign the request and issue an X.509 certificate to the node.

The second task is to create a secure directory to keep essential updating binaries. A database of the message digests for these binaries is then hardcoded into the agent.

Upon starting up, the normal node agent runs an automated check on the host to identify its system configuration: the operating system and its version, machine architecture, and so on. It then proceeds to register itself with its assigned supernode. When it connects to the supernode, it first requests for the supernode's certificate. All communications are secured using SSL. The agent then authenticates the supernode by checking if the supernode's certificate is signed by the root CA. If it is, the agent sends its system configuration (represented in XML) to the supernode agent. At this point, the node's system configuration will be checked by the supernode to see if it matches any known updates for that specific configuration. If there are any, the node will receive an update object from the supernode, which contains the name, URL, and a signature (which may be an MD5 message digest). The node then downloads the update from the repository using the provided URL and verifies its integrity using the MD5 message digest. If it matches, the update is installed.

After the registration phase, the agent enters into daemon mode, where it listens for any connections from supernodes or other nodes. When a supernode has any new updates, it will send them to the nodes.

B. Supernode

In this prototype, the supernode has three main roles. The first role is to keep an up-to-date database of system-specific updates, their signatures, and the repositories they are stored in, which is specified using a URL. The second role is to accept the registrations from normal node agents and keep them in a database. The third role is to deliver updates to the normal node agents upon matching their system configurations and their specific updates.

VII. Results and Discussion

The prototype was tested on a 10Mbps Ethernet network (Figure 2) with five hosts (Table I). Host A was set up as the CA and supernode collective. We used one supernode agent to simulate the entire supernode collective. Host E was set up as the repository, which was configured as a web server to deliver updates using the HTTP protocol.

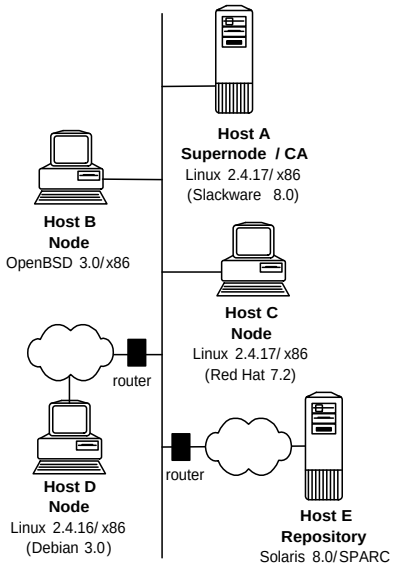

Fig. 2. The test network.

TABLE I
Hosts on the test network.

Host	Role	OS	Architecture
A	Supernode/CA	Linux 2.4.17 Slackware 8.0	x86
B	Normal node (Webserver)	OpenBSD 3.0	x86
C	Normal node (Desktop)	Linux 2.4.17 Red Hat 7.2	x86
D	Normal node (Laptop)	Linux 2.4.16 Debian 3.0	x86
E	Repository	Solaris 8.0	SPARC

First, each machine was manually arranged to have a valid, signed certificate. Then, the CA was set up on host A using OpenSSL tools. We then generated private and public keypairs for hosts B, C, and D. Certificate requests for these normal node hosts were sent to the CA on host A. The CA then issued signed certificates which were sent back to the various hosts emulating normal nodes.

We tested our prototype using three UNIX programs – sudo, pine, and wu-ftpd. Hosts B (OpenBSD), C (Red Hat Linux), and D (Debian Linux) were installed with old or vulnerable versions of sudo, pine, and wu-ftpd respectively. We then placed updated replacement versions of the programs in the repository on Host E. The supernode agent on Host A was then configured with a database that maps the operating system with its specific update. For example, Red Hat Linux was mapped to pine. This means that

ISBN 0-7803-9814-9/$10.00

the pine version on the Red Hat machine is vulnerable and should be replaced. The sudo and pine packages we used are actual packages from Red Hat and Debian respectively, while the sudo update was a custom-made update.

The supernode agent was activated on host A. It then started listening for connections from agents. At this time, the agent on the other hosts were activated. Each agent automatically gathered information about its host's system configuration. The agent then initiated an SSL connection to the supernode agent on host A, and authenticated and registered itself with the supernode agent. The system configuration was then sent to the supernode agent. The supernode agent then looked through its updates database and matched them against the specific update packages in the repository. Once a match was found, the update was immediately delivered to the host, where the agent verified its integrity and applied the update. For example, in one of our tests, the pine email client on Host C was automatically updated from pine 4.33 to 4.44. To achieve high speed, the agents install the updates immediately after a verified integrity check. Table II shows the download speed ($S1$) and installation speed $S2$ (including integrity check time) for the updates, relative to the hosts' CPU speed, update method (M), and update size. The pine update application process took 19.76s because of its large size (2.63MB) and it was run on a slow CPU (166MHz).

TABLE II
UPDATES APPLIED.

Update	M	CPU	Size	$S1$	$S2$
sudo 1.6.5	custom	400MHz (Host B)	71.2KB	0.06s	0.06s
pine 4.44	rpm	166MHz (Host C)	2632KB	1.55s	19.76s
wu-ftpd 2.6.1	dpkg	700MHz (Host D)	250KB	0.24s	4.1s

VIII. Future Work

The experimental prototype only implements the core subset of our architecture. We are currently refining and extending the prototype as part of our ongoing large-scale intrusion detection infrastructure project. Future plans include porting and testing the network on more platforms in a larger test environment to examine performance issues with network latency, optimized protocol design, and topology issues. Emerging technologies, such as newer PKI implementations and IPv6, are being investigated.

Another interesting issue to look at is low-level updates. In this paper, software updates have been referred to as *application* updates. An interesting challenge would be to reconfigure the *OS kernel* dynamically without interfering with users' tasks. One solution is to notify all users of a reboot and issue a process standby (similar to a laptop's standby mode) before performing the reconfiguration.

One more important task is in the area of standardization. We need to define the stringent procedures that CAs have to enforce in order to verify the identity of an agent's user. We also need to define standards for organizations to tune their updates so that they can be used on our network. There are certain issues that we cannot address though. For example, commercial software developers may have restrictive licenses that do not allow their updates to be placed on an independent network. Also, if a commercial vendor takes a long time to release its updates following a security alert, we will have to wait until the update is released before it can be placed in the repository.

IX. Conclusion

One major reason why systems are compromised is because they are not updated on time. In this paper, we described a network that can be used to distribute software updates to systems in a secure and automated manner across organizations. We have developed an experimental prototype that shows favorable and promising results. We are currently refining and extending the prototype as part of our future large-scale intrusion detection infrastructure.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments and suggestions.

References

[1] S. Northcutt and J. Novak, *Network Intrusion Detection: An Analyst's Handbook*. New Riders Publishing, 2nd ed., 2000.

[2] A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL protocol version 3.0." http://www.netscape.com/eng/ssl3/, November 1996.

[3] "Debian GNU/Linux." http://www.debian.org/.

[4] "Red Hat Network." https://rhn.redhat.com/.

[5] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. L. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur, "DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype.," in *Proceedings of the 14th National Computer Security Conference*, pp. 167–176, October 1991.

[6] G. White, E. Fisch, and U. Pooch, "Cooperating security managers: A peer-based intrusion detection system," *IEEE Network*, vol. 10, pp. 20–23, January/February 1996.

[7] J. S. Balasubramaniyan, J. O. G. Fernandez, D. Isacoff, E. Spafford, and D. Zamboni, "An architecture for intrusion detection using autonomous agents," Tech. Rep. 98/05, COAST Laboratory, Purdue University, May 1998.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," *Designing Privacy Enhancing Technologies: Intl. Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009*, 2001.

[9] The MITRE Corporation, "Common Vulnerabilities and Exposures (CVE)." http://cve.mitre.org/.

[10] "Linux Standard Base." http://www.linuxbase.org/.

[11] W. T. Polk and N. E. Hastings, "Bridge certification authorities: Connecting B2B public key infrastructures." http://csrc.nist.gov/pki/documents/B2B-article.pdf.

[12] P. Zimmerman, *PGP User's Guide*. MIT Press, 1994.