ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

# Importance of Network Threat Visibility and Control against Program Threat

**Roopha Shree Kollolu Srinivasa**[1]

B. E Student, Department of Computer Science Engineering, East West Institute of Technology, India[1]

**ABSTRACT:** The Internet has changed crime in a huge way. No longer does a bank robber even need to be in the same country to rob a bank or financial institution – they can crack an unprotected web site from the comfort of their own home. No gun or physical presence is needed to rob a store – simply monitoring a poorly equipped store's WLAN can provide many credit card numbers. It is hard to safeguard your computer or prosecute criminals, when the criminal is in another country, possibly attacking through botnets.

**KEYWORDS: s**ecurity, network threats, viruses

## I. INTRODUCTION

Three case study exercises are useful in providing students a foundation in network security. All three each include a PowerPoint lecture and active-learning exercise, which serves as the case study. Three case studies related to networking include:

- Designing Information Security: Classifies information by confidentiality and criticality.
- Planning for Network Security: Determines services, connection establishment directions, security classifications, and access control and builds a colorful network diagram for security
- Using a Protocol Analyzer: Analyzes a protocol sequence generated upon laptop power-up, to determine which services, connections, and ports are used then.

Our case study exercises also help to prepare students for security planning and security evaluation. Two security planning exercises help students to learn the perspective of business (in this case, a doctor's office), in addition to the technical perspective. A protocol analysis helps students to exercise deep technical skills, when they evaluate a protocol analyzer dump, for a security scenario. The Health First Case Study provides the conversations (or information) for students to complete the exercises. A Small Business Security Workbook guides students through the security planning process, by using introductory text, guiding directions, and tables for students to complete. We next review the case study exercises in detail.

## II. THE IMPORTANCE OF NETWORK THREAT VISIBILITY

According to the Ponemon Institute's "2014 Cost of Cyber Crime: United States," the most costly cyber crimes are those caused by denial of service attacks, malicious insiders and malicious code, leading to 55% of all costs associated with cyber attacks. Not surprisingly, costs escalate when attacks are not resolved quickly. Participants in Ponemon's study reported the average time to resolve a cyber attack in 2014 was 45 days, at an average cost of $1,593,627 -- a 33% increase over 2013 cost and 32-day resolution. Worse, study participants reported that malicious insider attacks took on average more than 65 days to contain.

The increasing frequency, diversity and complexity of network-borne attacks is impeding threat resolution. Cisco's 2015 Annual Security Report found that criminals are getting better at using security gaps to conceal malicious activity; for example, moving beyond recently fixed Java bugs to use new Flash malware and Snowshoe IP distribution

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

techniques (increasing spam by 250%) and exploiting the 56% of Open SSL installations still vulnerable to Heartbleed, and others, or enlisting end users as cybercrime accomplices.
In this era of BYOD, BYOC, IoT and more, achieving real-world security for business-essential connectivity requires more visibility into network traffic, assets and patterns.

## III. TRUTHS AND MISCONCEPTIONS ABOUT VIRUSES

Because viruses often have a dramatic impact on the computer-using community, they are often highlighted in the press, particularly in the business section. However, there is much misinformation in circulation about viruses. Let us examine some of the popular claims about them.

- Viruses can infect only Microsoft Windows systems. **False**. Among students and office workers, PCs are popular computers, and there may be more people writing software (and viruses) for them than for any other kind of processor. Thus, the PC is most frequently the target when someone decides to write a virus. However, the principles of virus attachment and infection apply equally to other processors, including Macintosh computers, Unix workstations, and mainframe computers. In fact, no writeable stored- program computer is immune to possible virus attack. As we noted in Chapter 1, this situation means that all devices containing computer code, including automobiles, airplanes, microwave ovens, radios, televisions, and radiation therapy machines have the potential for being infected by a virus.

- Viruses can modify "hidden" or "read only" files. **True**. We may try to protect files by using two operating system mechanisms. First, we can make a file a hidden file so that a user or program listing all files on a storage device will not see the file's name. Second, we can apply a read-only protection to the file so that the user cannot change the file's contents. However, each of these protections is applied by software, and virus software can override the native software's protection. Moreover, software protection is layered, with the operating system providing the most elementary protection. If a secure operating system obtains control before a virus contaminator has executed, the operating system can prevent contamination as long as it blocks the attacks the virus will make.

- Viruses can appear only in data files, or only in Word documents, or only in programs. **False**. What are data? What is an executable file? The distinction between these two concepts is not always clear, because a data file can control how a program executes and even cause a program to execute. Sometimes a data file lists steps to be taken by the program that reads the data, and these steps can include executing a program. For example, some applications contain a configuration file whose data are exactly such steps. Similarly, word processing document files may contain startup commands to execute when the document is opened; these startup commands can contain malicious code. Although, strictly speaking, a virus can activate and spread only when a program executes, in fact, data files are acted upon by programs. Clever virus writers have been able to make data control files that cause programs to do many things, including pass along copies of the virus to other data files.
- Viruses spread only on disks or only in e-mail**. False**. File-sharing is often done as one user provides a copy of a file to another user by writing the file on a transportable disk. However, any means of electronic file transfer will work. A file can be placed in a network's library or posted on a bulletin board. It can be attached to an electronic mail message or made available for download from a web site. Any mechanism for sharing files—of programs, data, documents, and so forth—can be used to transfer a virus.

- Viruses cannot remain in memory after a complete power off/power on reboot. **True**. If a virus is resident in memory, the virus is lost when the memory loses power. That is, computer memory (RAM) is volatile, so that all contents are deleted when power is lost.[2] However, viruses written to disk certainly can remain through a reboot cycle and reappear after the reboot. Thus, you can receive a virus infection, the virus can be written to disk (or to network storage), you can turn the machine off and back on, and the virus can be reactivated during the reboot.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

Boot sector viruses gain control when a machine reboots (whether it is a hardware or software reboot), so a boot sector virus may remain through a reboot cycle because it activates immediately when a reboot has completed.

- Viruses cannot infect hardware**. True**. Viruses can infect only things they can modify; memory, executable files, and data are the primary targets. If hardware contains writeable storage (so-called firmware) that can be accessed under program control, that storage is subject to virus attack. There have been a few

- Viruses can be malevolent, benign, or benevolent. **True.** Not all viruses are bad. For example, a virus might locate uninfected programs, compress them so that they occupy less memory, and insert a copy of a routine that decompresses the program when its execution begins. At the same time, the virus is spreading the compression function to other programs. This virus could substantially reduce the amount of storage required for stored programs, possibly by up to 50 percent. However, the compression would be done at the request of the virus, not at the request, or even knowledge, of the program owner.

## IV. TARGETED MALICIOUS PROGRAM

So far, we have looked at anonymous code written to affect users and machines indiscriminately. Another class of malicious code is written for a particular system, for a particular application, and for a particular purpose. Many of the virus writers' techniques apply, but there are also some new ones.

### Trapdoors

A **trapdoor** is an undocumented entry point to a module. The trapdoor is inserted during code development, perhaps to test the module, to provide "hooks" by which to connect future modifications or enhancements or to allow access if the module should fail in the future. In addition to these legitimate uses, trapdoors can allow a programmer access to a program once it is placed in production.

### Salami Attack

An attack known as a **salami attack**. This approach gets its name from the way odd bits of meat and fat are fused together in a sausage or salami. In the same way, a salami attack merges bits of seemingly inconsequential data to yield powerful results. For example, programs often disregard small amounts of money in their computations, as when there are fractional pennies as interest or tax is calculated.

Such programs may be subject to a salami attack, because the small amounts are shaved from each computation and accumulated elsewhere—such as the programmer's bank account! The shaved amount is so small that an individual case is unlikely to be noticed, and the accumulation can be done so that the books still balance overall. However, accumulated amounts can add up to a tidy sum, supporting a programmer's early retirement or new car. It is often the resulting expenditure, not the shaved amounts, that gets the attention of the authorities.

### Covert Channels: Programs That Leak Information

So far, we have looked at malicious code that performs unwelcome actions. Next, we turn to programs that communicate information to people who should not receive it. The communication travels unnoticed, accompanying other, perfectly proper, communications. The general name for these extraordinary paths of communication is **covert channels**.

Suppose a group of students is preparing for an exam for which each question has four choices (a, b, c, d); one student in the group, Sophie, understands the material perfectly and she agrees to help the others. She says she will reveal the answers to the questions, in order, by coughing once for answer "a," sighing for answer "b," and so forth. Sophie uses a

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

communications channel that outsiders may not notice; her communications are hidden in an open channel. This communication is a human example of a covert channel.

**Timing Channels**

Other covert channels, called **timing channels**, pass information by using the speed at which things happen. Actually, timing channels are shared resource channels in which the shared resource is time.

A service program uses a timing channel to communicate by using or not using an assigned amount of computing time. In the simple case, a multiprogrammed system with two user processes divides time into blocks and allocates blocks of processing alternately to one process and the other. A process is offered processing time, but if the process is waiting for another event to occur and has no processing to do, it rejects the offer. The service process either uses its block (to signal a 1) or rejects its block (to signal a 0).

## V. CONTROL AGAINST PROGRAM THREAT

There are many ways a program can fail and many ways to turn the underlying faults into security failures. It is of course better to focus on prevention than cure; how do we use controls during software development—the specifying, designing, writing, and testing of the program—to find and eliminate the sorts of exposures we have discussed? The discipline of software engineering addresses this question more globally, devising approaches to ensure the quality of software. In this book, we provide an overview of several techniques that can prove useful in finding and fixing security flaws .

In this section we look at three types of controls: developmental, operating system, and administrative. We discuss each in turn.

**Developmental Controls**

Many controls can be applied during software development to ferret out and fix problems. So let us begin by looking at the nature of development itself, to see what tasks are involved in specifying, designing, building, and testing software. The Nature of Software Development Software development is often considered a solitary effort; a programmer sits with a specification or design and grinds out line after line of code. But in fact, software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product. Development requires people who can

- specify the system, by capturing the requirements and building a model of how the system should work from the users' point of view
- design the system, by proposing a solution to the problem described by the requirements and building a model of the solution
- implement the system, by using the design as a blueprint for building a working solution
- test the system, to ensure that it meets the requirements and implements the solution as called for in the design
- review the system at various stages, to make sure that the end products are consistent with the specification and design models
- document the system, so that users can be trained and supported
- manage the system, to estimate what resources will be needed for development and to track when the system will be done
- maintain the system, tracking problems found, changes needed, and changes made, and evaluating their effects on overall quality and functionality

One person could do all these things. But more often than not, a team of developers works together to perform these tasks. Sometimes a team member does more than one activity; a tester can take part in a requirements review, for

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

example, or an implementer can write documentation. Each team is different, and team dynamics play a large role in the team's success.

We can examine both product and process to see how each contributes to quality and in particular to security as an aspect of quality. Let us begin with the product, to get a sense of how we recognize highquality secure software.

**Modularity, Encapsulation, and Information Hiding**

Code usually has a long shelf-life, and it is enhanced over time as needs change and faults are found and fixed. For this reason, a key principle of software engineering is to create a design or code in small, self-contained units, called components or modules; when a system is written this way, we say that it is modular. Modularity offers advantages for program development in general and security in particular.

If a component is isolated from the effects of other components, then it is easier to trace a problem to the fault that caused it and to limit the damage the fault causes. It is also easier to maintain the system, since changes to an isolated component do not affect other components. And it is easier to see where vulnerabilities may lie if the component is isolated. We call this isolation encapsulation.

Information hiding is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components

## VI. TRUSTED O.S. DESIGN

Operating systems by themselves (regardless of their security constraints) are very difficult todesign. They handle many duties, are subject to interruptions and context switches, and mustminimize overhead so as not to slow user computations and interactions. Adding theresponsibility for security enforcement to the operating system substantially increases thedifficulty of designing an operating system.

Nevertheless, the need for effective security is becoming more pervasive, and good softwareengineering principles tell us that it is better to design the security in at the beginning than toshoehorn it in at the end. Thus, thissection focuses on the design of operating systems for a high degree of security. First, weexamine the basic design of a standard multipurpose operating system. Then, we considerisolation, through which an operating system supports both sharing and separating userdomains. We look in particular at the design of an operating system's kernel; how the kernel isdesigned suggests whether security will be provided effectively. We study two differentinterpretations of the kernel, and then we consider layered or ring-structured designs.

**Trusted System Design Elements**

That security considerations pervade the design and structure of operating systems implies twothings. First, an operating system controls the interaction between subjects and objects, sosecurity must be considered in every aspect of its design. That is, the operating system design must include definitions of which objects will be protected in what way, which subjects will have access and at what levels, and so on. There must be a clear mapping from the security requirements to the design, so that all developers can see how the two relate. Moreover, once a section of the operating system has been designed, it must be checked to see that the degree of security that it is supposed to enforce or provide has actually been designed correctly. This checking can be done in many ways, including formal reviews or simulations.

Again, a mapping is necessary, this time from the requirements to design to tests so that developers can affirm that each aspect of operating system security has been tested and shown to work correctly.

Second, because security appears in every part of an operating system, its design and implementation cannot be left fuzzy or vague until the rest of the system is working and being tested. It is extremely hard to retrofit security features to an operating system designed with inadequate security. Leaving an operating system's security to the last minute is

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

much like trying to install plumbing or wiring in a house whose foundation is set, structure defined, and walls already up and painted; not only must you destroy most of what you have built, but you may also find that the general structure can no longer accommodate all that is needed (and so some has to be left out or compromised). Thus, security must be an essential part of the initial design of a trusted operating system. Indeed, the security considerations may shape many of the other design decisions, especially for a system with complex and constraining security requirements. For the same reasons, the security and other design principles must be carried throughout implementation, testing, and maintenance.

Good design principles are always good for security, as we have noted above. But several important design principles are quite particular to security and essential for building a solid, trusted operating system. These principles have been articulated well by Saltzerand Saltzer and Schroeder :

- Least privilege.Each user and each program should operate by using the fewest privileges possible. In this way, the damage from an inadvertent or malicious attack isminimized.
- Economy of mechanism.The design of the protection system should be small, simple, and straightforward. Such a protection system can be carefully analyzed, exhaustively tested, perhaps verified, and relied on.
- Open design.The protection mechanism must not depend on the ignorance of potential attackers; the mechanism should be public, depending on secrecy of relatively few key items, such as a password table. An open design is also available for extensive public scrutiny, thereby providing independent confirmation of the design security.
- Complete mediation. Every access attempt must be checked. Both direct access attempts (requests) and attempts to circumvent the access checking mechanism should be considered, and the mechanism should be positioned so that it cannot be circumvented.
- Permission based. The default condition should be denial of access. A conservative designer identifies the items that should be accessible, rather than those that should not.
- Separation of privilege.Ideally, access to objects should depend on more than one condition, such as user authentication plus a cryptographic key. In this way, someone who defeats one protection system will not have complete access.
- Least common mechanism. Shared objects provide potential channels for information flow. Systems employing physical or logical separation reduce the risk from sharing.
- Ease of use. If a protection mechanism is easy to use, it is unlikely to be avoided.

Although these design principles were suggested several decades ago, they are as accurate now as they were when originally written. The principles have been used repeatedly and

successfully in the design and implementation of numerous trusted systems. More importantly, when security problems have been found in operating systems in the past, they almost always derive from failure to abide by one or more of these principles.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

VII. **ASSURANCE IN TRUSTED O.S.**

**Typical Operating System Flaws**

Periodically throughout our analysis of operating system security features, we have used the phrase "exploit a vulnerability." Throughout the years,many vulnerabilities have been uncovered in many operating systems. They have gradually been corrected, and the body of knowledge about likely weak spots has grown.

**Known Vulnerabilities**

In this section, we discuss typical vulnerabilities that have been uncovered in operating systems. Our goal is not to provide a "how-to" guide for potential penetrators of operating systems. Rather, we study these flaws to understand the careful analysis necessary in designing and testing operating systems. User interaction is the largest single source of operating system vulnerabilities, for several reasons:

- The user interface is performed by independent, intelligent hardware subsystems. The humancomputer interface often falls outside the security kernel or security restrictions implemented by an operating system.
- Code to interact with users is often much more complex and much more dependent on the specific device hardware than code for any other component of the computing system. For these reasons, it is harder to review this code for correctness, let alone to verify it formally.
- User interactions are often character oriented. Again, in the interest of fast data transfer, the operating systems designers may have tried to take shortcuts by limiting the number of instructions executed by the operating system during actual data transfer. Sometimes the instructions eliminated are those that enforce security policiesas each character is transferred. A second prominent weakness in operating system security reflects an ambiguity in access policy. On one hand, we want to separate users and protect their individual resources. On the other hand, users depend on shared access to libraries, utility programs, common data, and system tables. The distinction between isolation and sharing is not always clear at the policy level, so the distinction cannot be sharply drawn at implementation.

A third potential problem area is incomplete mediation. Recall that Saltzerrecommended an operating system design in which every requested access was checked for proper authorization. However, some systems check access only once per user interface operation, process execution, or machine interval. The mechanism is available to implement full protection, but the policy decision on when to invoke the mechanism is not complete.

Therefore, in the absence of any explicit requirement, system designers adopt the "most efficient" enforcement; that is, the one that will lead to the least use of machine resources.

Generality is a fourth protection weakness, especially among commercial operating systems for large computing systems. Implementers try to provide a means for users to customize their operating system installation and to allow installation of software packages written by other companies. Some of these packages, which themselves operate as part of the operating system, must execute with the same access privileges as the operating system. For example, there are programs that provide stricter access control than the standard control available from the operating system. The "hooks" by which these packages are installed are also trapdoors for any user to penetrate the operating system. Thus, several well-known points of security weakness are common to many commercial operating systems. Let us consider several examples of actual vulnerabilities that have been exploited to penetrate operating systems.

**Formal Verification**

The most rigorous method of analyzing security is through formal verification, which was introduced in Chapter 3. Formal verification uses rules of mathematical logic to demonstrate that a system has certain security properties. In formal verification, the operating system is modeled and the operating system principles are described as assertions. The collection of models and assertions is viewed as a theorem, which is then proven. The theorem asserts that the

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

operating system is correct. That is, formal verification confirms that the operating system provides the security features it should and nothing else.
Proving correctness of an entire operating system is a formidable task, often requiring months or even years of effort by several people. Computer programs called theorem provers can assist in this effort, although much human activity is still needed.

**Validation**

Formal verification is a particular instance of the more general approach to assuring correctness: verification. Validation is the counterpart to verification, assuring that the system developers have implemented all requirements. Thus, validation makes sure that the developer is building the right product (according to the specification), and verification checks the quality of the implementation . There are several different ways to validate an operating system.

- Requirements checking. One technique is to cross-check each operating system requirement with the system's source code or execution-time behavior. The goal is to demonstrate that the system does each thing listed in the functional requirements. This process is a narrow one, in the sense that it demonstrates only that the system does everything it should do. In security, we are equally concerned about prevention: making sure the system does not do the things it is not supposed to do. Requirements checking seldom addresses this aspect of requirements compliance.

- Design and code reviews. Design and code reviews usually address system correctness (that is, verification). But a review can also address requirements implementation. To support validation, the reviewers scrutinize the design or the code to ensure traceability from each requirement to design and code components, noting problems along the way (including faults, incorrect assumptions, incomplete or inconsistent behavior, or faulty logic). The success of this process depends on the rigor of the review.

- System testing. The programmers or an independent test team select data to check the system. These test data can be organized much like acceptance testing, so behaviors and data expected from reading the requirements document can be confirmed in the actual running of the system. The checking is done in a methodical manner to ensure completeness.

## VIII. DIGITAL SIGNATURE

A **digital signature** is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, such that the sender cannot deny having sent the message (authentication and non-repudiation) and that the message was not altered in transit (integrity). Digital signatures are commonly used for software distribution, financial transactions, and in other cases where it is important to detect forgery or tampering.

Digital signatures are often used to implement electronic signatures, a broader term that refers to any electronic data that carries the intent of a signature, but not all electronic signatures use digital signatures. In some countries, including the United States, India, Brazil, and members of the European Union, electronic signatures have legal significance.

Digital signatures employ a type of asymmetric cryptography. For messages sent through a nonsecure channel, a properly implemented digital signature gives the receiver reason to believe the message was sent by the claimed sender. In many instances, common with Engineering companies for example, digital seals are also required for another layer of validation and security. Digital seals and signatures are equivalent to handwritten signatures and stamped seals. Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes, in the sense used here, are cryptographically based, and must be implemented properly to be effective. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not signa message, while also claiming their private key remains secret; further, some non- repudiation schemes offer a time stamp for the digital signature, so that even if the private key is exposed, the signature is valid. Digitally signed messages may be anything representable as a bitstring: examples include electronic mail, contracts, or a message sent via some other cryptographic protocol.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

We are all familiar with the concept of a signature. We sign a document to show that it originated from us or was approved by us. The signature is proof to the recipient that the document comes from the correct entity. When a customer signs a check to himself, the bank needs to be sure that the check is issued by that customer and nobody else. In other words, a signature on a document, when verified, is a sign of authentication; the document is authentic. Consider a painting signed by an artist. The signature on the art, if authentic, means that the painting is probably authentic.

When Alice sends a message to Bob, Bob needs to check the authenticity of the sender; he needs to be sure that the message comes from Alice and not Eve. Bob can ask Alice to sign the message electronically. In other words, an electronic signature can prove the authenticity of Alice as the sender of the message. We refer to this type of signature as a digital signature.

Comparison

Before we continue any further, let us discuss the differences between two types of signatures:

conventional and digital.

***Inclusion***

A conventional signature is included in the document; it is part of the document. When we write a check, the signature is on the check; it is not a separate document. On the other hand, when we sign a document digitally, we send the signature as a separate document. The sender sends two documents: the message and the signature. The recipient receives both documents and verifies that the signature belongs to the supposed sender. If this is proved, the message is kept; otherwise, it is rejected.

**Verification Method**

The second difference between the two types of documents is the method of verifying the signature. In conventional signature, when the recipient receives a document, she compares the signature on the document with the signature on file. If they are the same, the document is authentic. The recipient needs to have a copy of this signature on file for comparison. In digital signature, the recipient receives the message and the signature. A copy of the signature is not stored anywhere. The recipient needs to apply a verification technique to the combination of the message and the signature to verify the authenticity.

**Relationship**

In conventional signature, there is normally a one-to-many relationship between a signatureand documents. A person, for example, has a signature that is used to sign

manychecks, many documents, etc. In digital signature, there is a one-to-one relationshipbetween a signature and a message. Each message has its own signature. The signatureof one message cannot be used in another message. If Bob receives two messages, oneafter another, from Alice, he cannot use the signature of the first message to verify thesecond. Each message needs a new signature.

**Duplicity**

Another difference between the two types of signatures is a quality called duplicity. In conventional signature, a copy of the signed document can be distinguished from the original one on file. In digital signature, there is no such distinction unless there is a factor of time (such as a timestamp) on the document. For example, suppose Alice sends a document instructing Bob to pay Eve. If Eve intercepts the document and the signature, she can resend it later to get money again from Bob.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

**Vol. 2, Issue 2, February 2014**

**Need for Keys**

In conventional signature a signature is like a private "key" belonging to the signer of the document. The signer uses it to sign a document; no one else has this signature. The copy of the signature is on file like a public key; anyone can use it to verify a document, to compare it to the original signature.

In digital signature, the signer uses her private key, applied to a signing algorithm, to sign the document. The verifier, on the other hand, uses the public key of the signer, applied to the verifying algorithm, to verify the document. Can we use a secret (symmetric) key to both sign and verify a signature? The answer is no for several reasons. First, a secret key is known only between two entities (Alice and Bob, for example). So if Alice needs to sign another document and send it to Ted, she needs to use another secret key. Second, as we will see, creating a secret key for a session involves authentication, which normally uses digital signature. We have a vicious cycle. Third, Bob could use the secret key between himself and Alice, sign a document, send it to Ted, and pretend that it came from Alice.

**Process**

Digital signature can be achieved in two ways: signing the document or signing a digest of the document.

## IX. CONCLUSION

Secret sharing schemes are ideal for storing information that is highly sensitive and highly important. Examples include: encryption keys, missile launch codes, and numbered bank accounts. Each of these pieces of information must be kept highly confidential, as their exposure could be disastrous, however, it is also critical that they should not be lost. Traditional methods for encryption are ill-suited for simultaneously achieving high levels of confidentiality and reliability. This is because when storing the encryption key, one must choose between keeping a single copy of the key in one location for maximum secrecy, or keeping multiple copies of the key in different locations for greater reliability.

## REFERENCES

1. Lansford, J., (2000). HomeRFTM/SWAP: A Wireless Voice and Data System for the Home. Intel Communications Architecture Labs, Hillsboro, Oregon, 2000
2. O'Hara, B. & Petrick, A., (1999). IEEE 802.11 Handbook: A Designer's Companion, Standards Information Network, IEEE Press, New York, New York, 1999.
3. The Wireless LAN Standard. Cisco Systems, 2000.
4. 802.11a: A Very-High-Speed, Highly Scalable Wireless LAN Standard., White Paper, 2002, www.proxim.com