NOOSE – Networked Object-Oriented Security Examiner

Bruce Barnett – General Electric Corporate Research & Development

ABSTRACT

NOOSE (Networked Object-Oriented Security Examiner) is a distributed vulnerability analysis system based on object modeling. It merges the functionality of host-based and network-based scanners, storing the results into several object classes. The remote agents are implemented as dynamically extended PERL agents. NOOSE is able to collect vulnerabilities from a variety of sources, including outputs from other vulnerability analysis programs (e.g., Muffet's CRACK), collecting information from systems that may or may not have cooperative agents on them. Communication is based on a secure, reliable datagram protocol implemented as a set of PERL object classes. Unlike some vulnerability systems, NOOSE presents the vulnerability information as an integrated database, showing how vulnerabilities may be combined into chains across multiple accounts and systems. It understands unconditional vulnerabilities (i.e., stack-overflow, password guessing) along with conditional (Trojan horse, rlogin, and NFS access). Conditional vulnerabilities gain limited or privileges if conditions exist, such as access to specific accounts. The information is presented as an object-oriented "spreadsheet" format, allowing the security manager to explore vulnerabilities at whim. Once complete, the vulnerability analysis can move both forwards and backwards interactively, showing both what a selected account can attack, as well as showing who can attack a selected account. Besides vulnerability analysis, the system can intelligently verify the installation of security patches, dynamically installing missing patches. NOOSE is therefore a flexible prototype, able to provide a subset of the functionality of COPS, SATAN and TRIPWIRE, yet because of the object model, be used for developing new paradigms, such as reacting to intrusions, information warfare, and survivability management systems.

Problem Statement

This paper discusses limitations in *Vulnerability Analysis systems* such as COPS, SATAN Tiger, RSS and ISS. For convenience, these systems will be referred to as *VA systems*. In this paper, a vulnerability is a potential path to break into someone's account to elevate their privilege. This paper also discusses *vulnerability chains*, which is defined to be two or more vulnerabilities, that can be executed in sequence. An example of a chain is using NFS to insert a Trojan horse into a directory, which can be executed by a system administrator, to gain root access to a file server. Once this key account has been breached, the group of related clients become vulnerable because of the relationship between servers and clients. This collection of systems will be called a *workgroup*. In a large facility, there may be dozens or hundreds of workgroups, with separate administrators.

The goal of this paper is to find a way to identify and eliminate these vulnerability chains among and between workgroups. The author believes current VA systems have difficulty in doing this because of the following reasons:

- Some potential vulnerabilities are accepted as a matter of policy, and the convenience value overrides the potential risk. These are considered conditional vulnerabilities, because other conditions determine how risky it is.
- Vulnerability chains composed of one or more conditional vulnerabilities across multiple systems aren't evaluated.
- The results of host-based scanning is not integrated with the results of network-based scanning.
- Some systems, such as name servers or file servers, need stronger protection than clients. Likewise, some accounts, such as system administrators, are key elements in protecting the overall security. However, vulnerability analysis systems ignore these different threat potentials.
- Many security systems characterize potential failures with a simplistic red/yellow/green indicator or a numbered scale with 5 values. This is too coarse a measurement to be useful. The simple statement "NFS is insecure" is often ignored, while a report that identifies the precise directory that can be used to compromise the root account is more likely to be fixed.

The author feels that many current VA systems don't properly analyze the consequences of an intruder who breaches a firewall, or an insider attempting to gain access. Systems have complex relationships between clients and servers, and the compromise of a single server can allow hundreds of clients to be compromised. Servers can also be clients of other servers, and clients can be used to compromise other servers.

Likewise, accounts have complex inter-relationships, and system administrator accounts require special protection.

The vulnerability chain that was mentioned earlier, for instance, makes it trivial to break into an account without detection. Using low-level NFS calls (e.g., Leendert van Doorn NFSSHELL) typically requires no special privileges and allows access to any file on an NFS-exported file system not owned by root, such as those owned by a system administrator. If that account is not directly accessible, a Trojan horse can often be used. Since these attack mechanisms use ordinary file access mechanisms, they are rarely detected. Therefore it is essential that the potential danger be reported and repaired. Identifying this type of danger is one of the primary goals of NOOSE.

There have been attempts to integrate information, but these have been limited to a common output format [12], or a common user interface [11]. Kuang [1] identified vulnerability chains, but was limited to a single system. NetKuang [5] is one attempt to correct this. NOOSE uses a second approach.

Background

This implementation was based on our experience with an earlier Expert Fault Manager system [13], where the importance of relationships was emphasized. The author decided to apply what was learned while measuring security risks. However, several custom agents were on each system, and they needed to be upgraded manually. The communication system suffered from deadlocks occasionally. The core components were enhanced and applied to a vulnerability analysis prototype [14]. The work was influenced by the author's program for analyzing Trojan horses on UNIX systems, and various tools from Purdue [2, 3, 4]. This project was started in January of 1996, with funding was provided by Lockheed Martin and L-3 Communications. This paper describes the lessons learned from this prototype.

The author choose the name NOOSE because of the concept of drawing a circle around a set of arbitrary systems, and identifying vulnerabilities within this set, using OO techniques.

Description of System

The NOOSE system is written using PERL, chosen because of the power of the language, string parsing, and the ease of prototyping, as well as being able to potentially migrate code from COPS and SATAN/SAINT variants. NOOSE is implemented with 26 PERL-based object classes. A PERL agent exists on all systems cooperating with NOOSE. These agents talk to NOOSE using a centralized dispatcher, which in turn talks to an Information Warfare (IW) module. A Graphic User Interface (GUI) is written in PERL and TK, and provides a spreadsheet-like interface to the information contained in the IW module. The overall architecture can be seen in Figure 1.

Auxiliary files exist that contain information about patches, operating systems, and PERL modules to be uploaded to the agent.

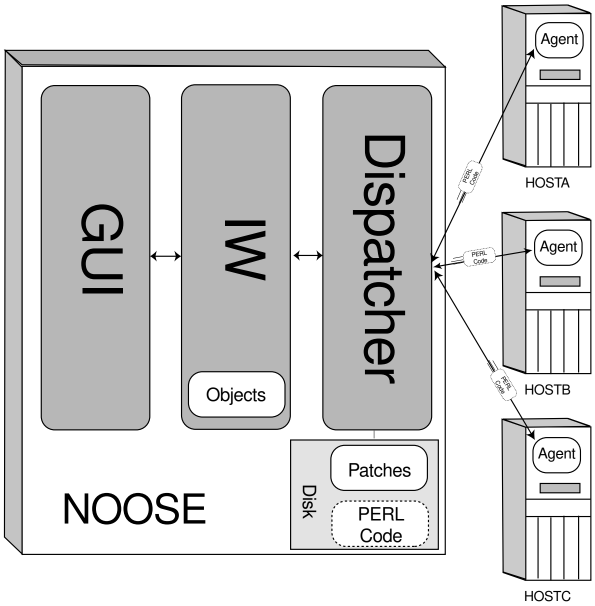

Figure 1: NOOSE architecture.

The agents just contains enough information to listen for new commands. New subroutines can be uploaded, and the revision tracked. If modules require other modules that are undefined, this is reported as an error. Therefore the agents are small, easy to install, and never need to be updated manually, which reduces maintenance costs. A simple dependency system was put in place, to identify required modules. However, the implementation used a single file for each operating system variant, and uploaded the entire file to the agent on demand. Therefore the dependency feature was rarely used.

The agent will run on a Windows NT box, but no vulnerabilities are currently gathered.

Object Classes

The primary goal of object-oriented programming is the ability to reuse object classes, lowering cost of development. Therefore our goals was to develop an object model that can support multiple algorithms. A data structure that can be used with two diverse algorithms has a better chance of being used by future algorithms. Object modeling provides a concise mechanism to document the data structures used in a system, often on a single page.

Object Model

The implementation contains of the following object classes:

- Communication (9 classes)
- System-related (1 Primary base class, 1 secondary base class, 13 sub-classes)
- GUI (2 classes)

System-related object classes

The NOOSE system uses the following object model, as shown in Figure 2. The 13 different object classes maintain information about remote systems, corresponding to their state:

- Patch
- Host
- Signature
- Operating System
- Account
- UID
- Vulnerability
- UNIX Resource (which has four sub classes)
  - File
  - Link
  - Directory
  - Missing

FileState

- PatchState

All system-related objects have a unique name. NOOSE uses a simple ASCII string composed of the following parts:

```
Object Class
Host responsible for object
Object-specific information
```

Examples are

```
account/pluto/smith
uid/pluto/214
host/pluto
dir/pluto//etc/mail/sendmail.cf
```

This provides a simple, extensible way to create new object types, as well as a simple method of locating the authority of the object (in this case, the host that must be queried to get information about that object.) In our communication paradigm, *instances* of objects resided on different hosts.

Relationship Superclass

All of the 13 system-related classes are derived from a base class that provides object lookup, creation, deletion, as well as relationship creation, querying, and navigation. This relationship or association is critical to the implementation. It creates one-to-one, one-to-many, and many-to-many relationships between

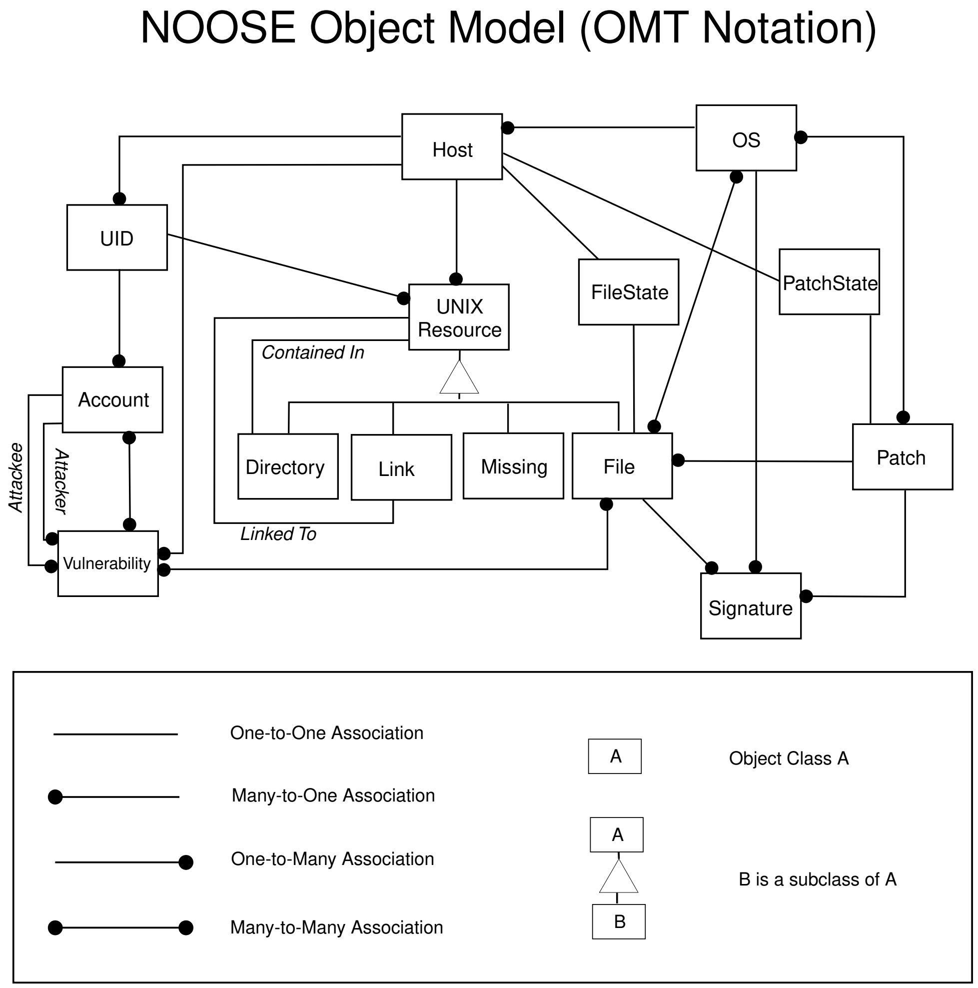

Figure 2: NOOSE object model.

any two system-related classes. Creating a one-to-many relationship between two instances (e.g., there are several files related to a patch) merely requires associating two object references with the method call:

```
$patchObject->one_to_many($fileObject);
```

No additional declarations are necessary. This simplified modifications of the object model. Other methods are *one_to_one*, *many_to_one* and *many_to_many*. If a one-to-one relationship is created,and a second relationship to the same class is added, an error is generated, suggesting a one-to-many relationship be used instead. The base class also provides ways to obtain, find, test, search and integrate related objects. It can be used as a collector object, and simplifies algorithm development significantly. The code fragment below demonstrates the methods as it finds and tests all of the required patched files corresponding to a revision of an operating system; see Listing 1.

These system-related classes, besides used to store information about the objects, allow algorithms to be easily constructed based on the relationship (or association) between objects. Often the relationship between two classes need not have a specific name, and the relationship is obvious from the context. Some objects have very few attributes, as the primary purpose is collection and navigation.

Two key classes are Accounts and Vulnerabilities. An account object is a username/hostname pair. The vulnerability object always has relationships to two accounts (except when the account has been removed, in which case it refers to the UID object class, which corresponds to the user ID number. In our system, a vulnerability is a potential mechanism to allow someone to go from one account to a different account.

The first relationship is to the account that can be compromised (the *attackee*), while the second shows the account that can compromise the first account (the *attacher*). Vulnerability objects may have a optional relationship to a file or directory, indicating the cause of the vulnerability.

Wildcards in Account names

Accounts consisted of two pieces of information: the username, and the hostname. NOOSE uses a "*" to indicate a wildcard, which may be in either the host, username or both. Accounts with wildcards are used to describe specific vulnerability classes. If an account had no password, then anyone on any system could access that account. This fictitious user is indicated by the account "account/*/*." If Joe Smith's account has a "+" in the ".rhosts" file, then the vulnerability can be initiated from the account "account/*/smith". If Smith's account on host "pluto" has a "." first in the searchpath, then this account can be compromised by the "account/pluto/*" account, which means anyone on host *pluto*. This simple naming convention can be used to describe the starting attack point of any vulnerability. In the case of vulnerabilities within a group or netgroup privileges, multiple vulnerabilities are created, with the attacking accounts expanded to the complete list of individuals within this group.

Vulnerability Chains

A vulnerability chain occurs when multiple vulnerabilities can be used to achieve a particular goal (or in this case, an account). Figure 3 shows such a chain.

In this case, assume a hacker can break into the *lpd* account on host *hosta* because it was missing a security patch. Next the attacker uses *NFSSHELL* to access the home directory of account *hostb/smith*. This account has write privileges in */usr/local/bin* and a Trojan horse is created. The *backup* account has this directory in the searchpath, and executing the Trojan horse compromises the account. Once done, the user may gain access to the *root* account on a file server, which allows access to all of the clients.

```
#Specify host to check
$os = Os->fetch($host->get_os_type); # Find the OS type and revision
foreach $patch ($os->get_many("Patch") { # get the patches for the OS
    # Get the files included in each patch cluster
    foreach $file ($patch->get_many("File")) {
        # Files have more than one signature - depends on OS rev
        foreach $signature ($file->get_many("Signature")) {
        # Only look at those that match the OS and revision
            if ($signature->has($os)) {  found a match
                $signature->verify(
                    host=>$host,
                    file=>$file,
                    patch=>$patch);
            }
        } # Signatures
    } # files
} # Patches
```

Listing 1: Testing patched files.

Communication classes

All network communication is comprised of objects and methods, which are specified as ASCII strings. The dispatcher looks at the object, determines the host to send the message to, and sends the information to the clients using a protocol layered on top of User Datagram Protocols. The original Expert Fault Manager implementation limited communication to small packets, and sent information as simple ASCII. Packet loss and security wasn't addressed. However, as larger pieces of information was needed, and the system suffered from deadlocks, a redesign was needed. A decision was made to provide the old function as a base class, and creating sub-classes that add features.

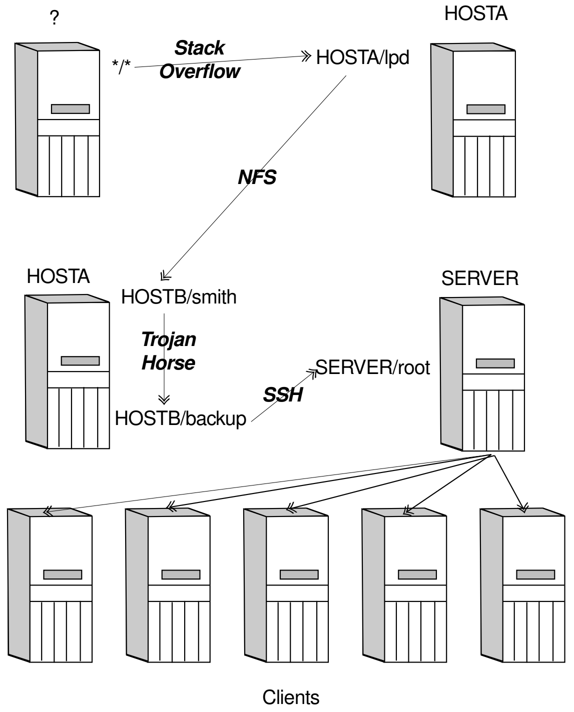

Figure 3: Vulnerability chain.

The original system used UDP because of the need for realtime predictable responses, and the desire to extend the protocol to use multicast and broadcast transports in the future. Three low-level UDP classes are broken into UDP Client, UDP Server, and UDP Common functions. All three are sub-classed to provide reliability and packet assembly and disassembly. These are again sub-classed to provide secure communication, using symmetric keys for encryption. The security used is weak, as keys are distributed manually, and reused for each session.

The communication object class provide a simple send and receive function. A system acting as a relay or filter, which is both a client and server, while avoiding deadlocks, merely needs the PERL fragment in Listing 2.

The wait method always waits on the socket associated with the object to the left of the arrow. Additional objects, corresponding to sockets, may be included. This single event loop, combined with timeouts and the test for pending data, was important in preventing deadlocks.

The communication classes contains options for verifying the reliability of the communication. Errors (i.e., dropped packets) can be purposely created by dropping a percentage of the packets pseudo-randomly. This was used to verify the reliability of the communication subsystem. Other options specify buffer sizes, timeouts, status, and counts of packets sent and received. The higher level routines reassemble packet fragments, and retransmit missing packets. The methods used for the reliability and secure subclasses are the same as the base class as far as the application is concerned. However, there were several dozen private used in the implementation.

Several protocols were attempted to provide reliable communication. To validate the protocol, a

```
my $client = udpsecclient->new( # act as a client to another server
                    port=>$port,
                    machine=>$machine,
                    security=>$level);

my $server = udpsecserver->new( # set up our own server
                    port=>$port,
                    security=>$level);
while (1) {
    $server->wait($client); # Wait for activity on either socket
    if ($server->pending) { # Our server gets data
        $client->send($server->receive); # send to other server
    }
    if ($client->pending) { # Our client gets results from server
        $server->send($client->receive); # send back to our client
    }
}
```

Listing 2: Simple client/server model.

Design of Experiment (DoE) methodology was used, varying buffer sizes, size of files, and number of simultaneous connections. Developing the reliable protocol was difficult. While the DoE did not clearly indicate the optimum parameters, a reliable protocol was selected that used a common (and symmetric) send/receive method for clients and servers. The sender breaks up the message into smaller pieces, and sends them out sequentially. The receiver responds with a positive acknowledgment, indicating all packets were received, or a negative acknowledgment, which requests the sender to re-transmit the missing packets. Once a positive acknowledgment from the receiver has been received, the sender transmits an acknowledgment of the acknowledgment and changes states. This same method is used by both sides to communicate. Therefore the client requests informations from a server, a minimum of 6 packets are transmitted, and more if packets are lost. The primary difference between the client and the server classes was the states: a server is finished after sending information, and the client is finished after receiving information. Caching was easily added, with answers based on the combination of the object and the method. The results are saved if there were no errors.

Functionality

The system can potentially accept vulnerabilities from any source, but a parser must be build to extract the vulnerability type and account information. The system, as currently implemented, performs the following vulnerability checks:

```
Trojan Horse
NFS
RLOGIN/SSH
Output of Alec Muffet's CRACK program
Missing security-related patches
```

The first three operate on account objects (once for each account), while the last two operate on a host basis (once for each host).

The Trojan horse program, uploaded to the agent, finds the shell of the user and examines the appropriate start-up files. It parses the files, and keeps track of files that are sources, as well as the value of variables. When searchpaths are specified or modified, the values of these variables are used to determine the potential searchpath. If branches are taken, the program assumes both paths are used, so the searchpath examined by the program is the superset of the actual searchpath. The program, for safety, doesn't evaluate commands within back-quotes. Instead, a predetermined set of commands are evaluated once on each host, as specified by the program, and if one of the users has this string in their searchpath, the pre-determined value is inserted into the string. If it is not known, it is ignored. Consider the following C shell fragment:

```
if ( -f /local $a)
    localpath = ( /local/`arch`/bin )
else
    localpath = ( /usr/`arch`/bin \
           /usr/local/`arch`/bin )
endif
set path = ( $localpath $path )
```

All three directories will be examined in addition to the default value of the searchpath, assuming the directories exist. The software also detects recursive loops, and handles them to a depth of two, and aborts if a loop is detected. Using this, it is possible to get a searchpath for each account. This, in turn, is used to measure the potential for Trojan Horse attacks on all accounts. Currently the system only examines the permissions of the directories, as well as the permission of the parent directories, and symbolic links. The permission of the files inside the directory are not examined, unlike the author's Trojan checking program.

The algorithm asks the agent to extract all of the user and group information from the host, if necessary. Then it creates a set of account objects matching each found. Originally, each group and account was asked individually, but a new dispatch method was created to retrieve all of the account information in a single query for efficiency.

When one account is selected for the vulnerability check, the IW module asks the agent to get the searchpath, which returns a list of directories by name. The IW module creates instances of these objects as needed in its own database, and asks the agent to list who has write permission for each of the directories (if this has not been asked before). Because each object has a name that is unique, and also specifies the host responsible for the information, instances of objects are externalized, and the object name and method is passed to the agent in plain text, and the results is also in plain text, fragmented and encrypted according to the communication object class. By examining each directory for user-, group- and world-write permission, the agent can return a list of accounts that have the ability to write to the directory specified. A special type of object is needed to correspond to missing directories. This checks for the potential of someone being able to create a directory in the future.

When a Trojan Horse is found, one or more vulnerabilities are created that specifies the victim and the potential attacker(s), as well as the file or directory that caused the problem. In the case of a vulnerability by group-write permission, multiple vulnerabilities are created, listing everyone in the group as a unique attacker. If a directory is world-writable, or the user allows the current directory to be in the searchpath, the attacker is the wildcard account on this machine.

UID's are used when a directory is owned by an account that no longer exists (i.e., has no name associated to it.)

NFS

When an account is examined for NFS vulnerability, the algorithm determines if the file is on a NFS client or a NFS server. Examined on a client, the "attacker" is the root account on the server where the directory is exported, as well as the root account on the client. However, on a server, the system examines the export list, and the netgroup information, and specifies all of the accounts that have the ability to modify the files. In other words, when examining "smith" on "pluto", if "smith" on "neptune" has write permission, then "root" on "neptune" does as well. The system also examines the NFS server if the client port numbers must be less than 1024, indicating a privileged account. If not, then the attacking account is the wildcard account "anyone" on "neptune". If the directory is exported to the world, the attacker is identified as "anyone" on "anyhost," indicating that anyone on any machine that can access the system can break into the account.

RLOGIN

The agent examines the system configuration for /etc/hosts.equiv and $HOME/.rhosts as well as the SSH equivalent files to determine which counts have access to the selected account. NIS netgroups and "+" in the .rhosts file are understood, and appropriate vulnerabilities are created.

CRACK

The system parses a file generated by CRACK, and looks for a matching hostname. When found, it creates a vulnerability between the "anybody" on this host to the account whose password was guessed.

Checking System Patches

The IW module first reads a series of files that contain a one-line summary per file associated with a patch. The OS type, revision, architecture, file path, patch ID, size, and MD5 value of each file is specified. This file is created automatically for Solaris systems using a shell script that weekly retrieving the current patch status. The IW module, while reading this file, creates Patch, File, OS and Signature objects. These must be separate objects because a file may have multiple signatures depending on the OS, revision, and patch. Also created is a FileState and PatchState object which corresponds to the actual state of a file and patch on a particular host, instead of the generalized information, valid for all systems of the same revision. In other words, PatchState and FileState have an association with a specific host.

The system gets the list of files for each patch and examines the signature of each file if it has permission. Five results are possible: (1) correct revision, (2) wrong revision, (3) file does not exist, and (4) insufficient privileges to read file and (5) system cannot execute the external MD5 program to check signature. The system can form a conclusion based on partial information. If it cannot read one file of a patch, and a second is the wrong revision, it concludes the patch has not been applied. If it is uncertain because it cannot read all of the files, but the rest are correct, it indicates the patch might be applied. Therefore determining if patches have been applied is accurate even if someone replaced a file after a patch has been applied, or modified any of the utilities (i.e., showrev). Missing patches create different vulnerabilities, depending on permissions of the un-patched file. This is simplistic, and uses the *set-uid* permissions and owner information to identify the attacked account. If the file is a library, the attacked account is considered to be the *root* account. A future version should use an internal MD5 checksum utility to prevent tampering with the executable.

Using the GUI, it is possible for the security officer to see the state of the patch and associated files. If desired, new patches can be uploaded to the system, and applied.

User Interface

The GUI (Figure 4) presents the information in three sections, General operations, host-specific operations, and account-specific operations. The general commands are a series of buttons that print out summaries, load the patch database, dump the database, and interface to other applications for advanced analysis.

The host specific operations include the following actions:

- Uploading the PERL modules.
- Querying the revision of the current modules
- Fetching the user and group information
- Listing all of the UID's on a host
- Listing all of the accounts on a host.
- Scanning the output of CRACK, merging new vulnerabilities
- Scanning the system for missing patches
- Displaying the results of the patch analysis

Displaying the UID or accounts presents the information in a scrolling list below, which can be used for the account-specific operations. Typically accounts are displayed in the scrolling list, and one or more accounts can be selected, and the account-specific methods can be used. Operations iterate over each item selected. If no account is selected, the account in the Argument: location is used. Typically the account that is of primary interest is copied and pasted into this for convenience.

The account-specific methods include:

- Trojan – Perform a Trojan Horse scan on the selected account
- NFS – Perform a NFS vulnerability check
- RLOGIN – Perform checks on rlogin/rsh/rcp/ssh
- Attacker – shows everyone who can attack the selected account

- Attackee – shows everyone the selected account can attack
- Show Vulnerabilities – shows account-specific vulnerabilities

Dispatcher and objects

The dispatch system was reused from an earlier project, using a version of PERL that did not support OO. The earlier project used "pseudo-object-oriented" techniques. That is, the dispatcher used strings to identify objects, methods and parameters. Based on its table, it would select a protocol, host and port an communicate with the remote system using the communication object classes. This included methods for query/response, uploading files and patches, and asking the remote agent to evaluate and execute PERL code. There was a weak correlation between these pseudo-objects and the object class used by the IW module.

Problems Encountered

There were four significant problems with the implementation:

- This prototype evolved from an earlier system, and the object classes were weakly integrated with the remote execution of commands. Agents and their methods could have multiple states. Methods could be undefined or out of date. Queries could be pending, obsolete, or never asked. Answers could be cached on the agent, or in the dispatch, or integrated into the database. Timeouts could occur anywhere. A redesign with a unified view of the remote information is needed.
- Secondly, retrieving the information was often piecemeal. Sometimes objects were created merely as collection objects and navigation objects. Accounts were created when vulnerabilities were found, and attributes and associations about these objects would often be left undefined. Therefore a large part of the code is testing for the existence of information, and performing queries to fill in missing information if needed.
- Any distributed system is unreliable and asynchronous. Therefore any remote query could fail, and failures had to be handled. A better design would have asynchronous gathering of information, and well-defined algorithms driven by an expert system, reporting on problems.
- The design of the GUI allowed gathering and traversal of the information in any order desired. There was no pre-determined order in the information gathering process. This provided flexibility, but added to the complexity.

These problems caused the code to be bulky and inelegant. Everything worked, but similar code had to be inserted in multiple places.

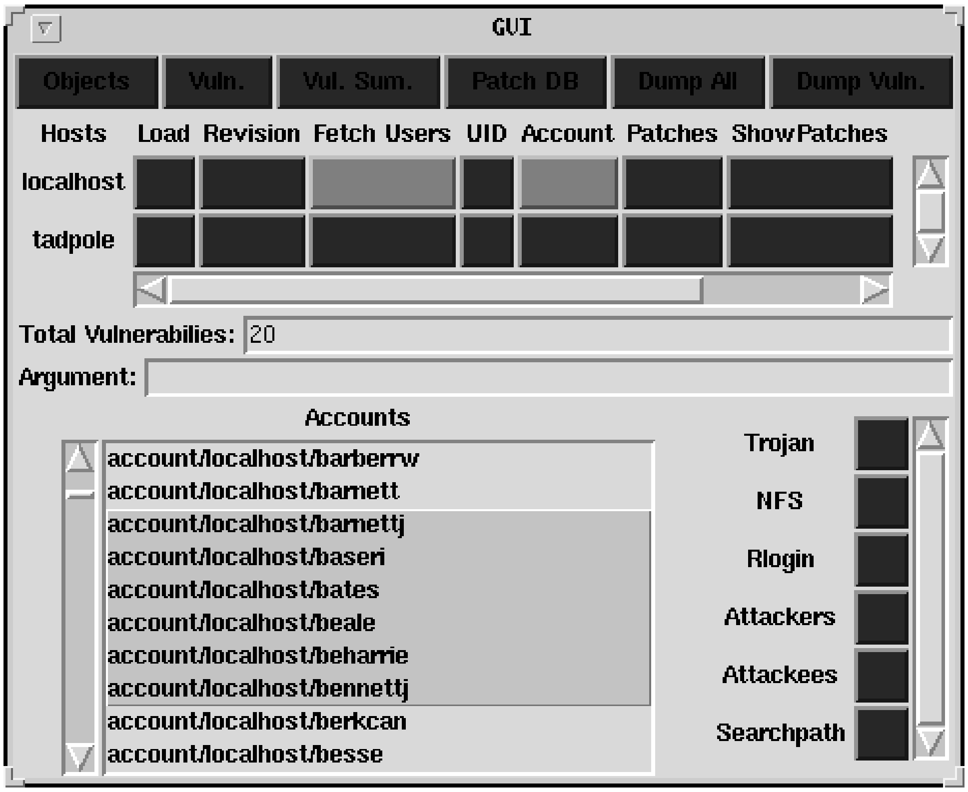

Figure 4: NOOSE GUI.

The second biggest difficulty was the development of the secure, reliable datagram-based protocol. Developing distinct classes between the server and client was essential, but finding the common methods that supported three types of communication (raw, reliable, secure) was difficult. A minor change in the algorithm could easily cause a deadlock.

Conclusions

As in our earlier project [13], an important element to these algorithms is the ability to traverse data structures using associations between objects. This allows algorithms to use the context surrounding objects while making decisions about single objects. This also stimulates new algorithm development, while reusing existing code. In some cases, the objects needed few attributes, as it was used primarily as a navigation point in the data structure.

The system has reasonable performance. Large servers with 2000 accounts could have each account examined in about 30 minutes. A large proportion of this time is believed to be associated with auto mounting the directories found in users search paths.

Because much of the software required handling missing data, and undefined values, a multi-threaded, asynchronous programming model is needed, combined with an object model that unifies the data structures of the agent and system.

The three-way handshake for reliable datagram communication is important for deadlock protection and obvious in hindsight. The symmetric nature allowed code reuse, and therefore simplified the object classes. Using this, it was possible to transmit patch clusters of 70 megabytes. Occasionally timeouts occurred where both sides were waiting, but the system typically recovered. Creating communication object classes designed for sub-classing allows future extensions while retaining the same API.

Future Directions

The communication classes can be extended is several ways. A better key distribution system can be inserted, as well as alternate encryption algorithms. Also, a subclass can be created that combines multiple lightweight messages in a single authenticated/encrypted packet. A mechanism for proxy agents can be added, as well as a means to locate agents by broadcast. Threaded asynchronous communication would simplify code development, as would uniform methods for testing and retrieving remote attributes.

Objects are currently stored in memory only, or dumped to an external file for analysis. An ASCII representation of the database is about 1 Megabyte in size for 2000 vulnerabilities. A persistent database is desirable, especially to a relational database.

Alternate mechanisms of viewing and analyzing vulnerabilities is desired. Several simple algorithms for categorizing vulnerabilities are suggested:

- Identify files with the largest number of associated vulnerabilities.
- Counting the number of paths between any two accounts.
- Identify all of the accounts that can potentially break into a selected account.
- Identify the worst case result of a single compromised account.

The author feels the concepts can be used for advanced policy management systems, as security can be measured more accurately, and relationships can be constructed between files and services.

Summary

The author strongly feels that object modeling is essential to writing next-generation security algorithms, allowing a single database to be used for diverse algorithms. This single database merges together the base functionality of host-based scanners, network-based scanners, file tampering and intelligent patch management systems. The author hopes the object model will be useful to others developing similar applications.

The author believes that this implementation shows several unique traits. The implementation simplifies prototyping new algorithms, and allows reusing data for multiple applications. Adding the patch management software was simpler than the core communication classes. The agent structure simplifies support. The object class for communication can be extended in many ways. The use of the vulnerability and account object class provides a simple and elegant, yet powerful way to integrate information from multiple sources, as we feel there is great Notential and flexibility to this mechanism.

Author Information

Bruce Barnett graduated from RPI in 1973. He is currently a Computer Scientist doing research at General Electric's Corporate Research and Development Center, PO Box 8, Schenectady, NY, 12309. His research covers traffic analysis, expert systems, realtime video multicast, and security systems. His electronic address is barnett@crd.ge.com.

References

[1] Robert W. Baldwin, *Kuang: Rule-based security checking,* Documentation in ftp://ftp.cert.org/pub/tools/cops/1.04/cops.tar .

[2] Dan Farmer & Eugene H. Spafford, "The Cops Security Checker System," *USENIX, Summer 1990* .

[3] D. Farmer and W. Venema, "Security administrator's tool for analyzing networks," http://www.fish.com/zen/satan/satan.hmtl .

[4] Gene Kim and E. H. Spafford, *The design of a system integrity monitor: Tripwire,* Technical Report CSD-TR-93-071, Department of

Computer Sciences, Purdue University, West Lafayette, Indiana, November 1993.

[5] Dan Zerkle and Karl Levitt, ‘‘NetKuang – A Multi-Host Configuration Vulnerability Checker,’’ *USENIX 1996*.

[6] Bruce Barnett, ftp://coast.cs.purdue.edu/pub/tools/unix/trojan/trojan.pl .

[7] Doug Schales, ‘‘Tiger,’’ ftp://coast.cs.purdue.edu/pub/tools/unix/tiger .

[8] Internet Security Systems, *Internet Scanner and System scanner*, http://www.iss.net/ .

[9] Diego Zamboni, *SAINT: A Security Analysis Integration Tool*, ftp://coast.cs.purdue.edu/pub/doc/tools/SAINT.ps.gz .

[10] Diego Zamboni, *New COPS Analysis and Report*, ftp://coast.cs.purdue.edu/pub/tools/unix/carp-ncarp .

[11] *Merlin*, http://ciac.llnl.gov/ciac/ToolsMerlin.html .

[12] *SPI – Security Profile Inspector*, http://ciac.llnl.gov/cstc/spi/spiwnt/spiv20.html .

[13] Bruce Barnett, Andrew Crapo, ‘‘An Expert Fault Manager using an Object Meta-Model’’, *Proceedings 20th Conference on Local Computing Networks*, Minneapolis, MN, Oct 1995.

[14] Dai Nha Wu, Bruce Barnett, ‘‘Vulnerability Assessment and Intrusion Detection with Dynamic Software Agents,’’ *Ninth Annual Software Technology Conference*, Salt Lake City, Utah, May 1997.