Mutating Network Models to Generate Network Security Test Cases

Ronald W. Ritchey
National Security Team
Booz•Allen & Hamilton
Falls Church, Virginia

Abstract

Security testing is normally limited to the scanning of individual hosts with the goal of locating vulnerabilities that can be exploited to gain some improper level of access on the target network. Scanning is a successful approach for discovering security problems, but it suffers from two major problems. First, it ignores security issues that can arise due to interactions of systems on a network. Second, it does not provide any concept of test coverage other than the obvious criteria of attempting all known exploitation techniques on every system on the network.

In this paper, I present a new method for generating security test cases for a network. This method extends my previous work in model checking network security by defining mutant operators to apply to my previously defined network security model. The resulting mutant models are fed into a model checker to produce counterexamples. These counterexamples represent attack scenarios (test cases) that can be run against the network. I also define a new coverage criterion for network security that requires a much smaller set of exploits to be run against the network to verify the network's security.

1. Introduction

Network security installations are frequently implemented using the fortress model. Attackers from the outside need to surmount formidable external obstacles before they can reach the internal protected systems. Most of the security effort is focused on attempting to create this barrier between the outside and the internal resources. Very little effort is focused on internal security. This model has the advantage of allowing the security team to focus on a much smaller set of hosts and configurations when implementing the security policy but it is fundamentally flawed.

Unless the barrier that is set up does not allow any connectivity there is some possibility that an attacker may circumvent the border defenses. This is due to the following issue. For a network to be useful it must offer services. These services are implemented in software and it is difficult to guarantee that any complex piece of software does not contain some flaws [Beizer]. These flaws frequently translate into security vulnerabilities. If exploitable flaws exist in a service, even if the flaws have not been discovered, they still represent a potential for network intrusion. New security bugs are frequently discovered in server software.

Given that it is impossible to state with final authority that there are no possible ways to bypass the border security, it follows that some effort must be expended upon internal systems if the network is truly to be considered secure. This leaves open the problem of how to best protect these internal systems, without overwhelming the security team.

Some sites rely on automated tools to perform vulnerability scanning of each host on the network. Programs such as Computer Oracle and Password System (COPS) [COPS], System Scanner by ISS [ISS], and CyberCop by Network Associates [NAI] are examples of tools that can scan hosts to attempt to discover vulnerabilities in the host's configuration. These tools typically perform a decent job of discovering host vulnerabilities, but they do require significant time and effort to execute and evaluate their results. In addition they do not attempt to identify how combinations of configurations on the same host or between hosts on the same network can contribute to the network's vulnerability.

In my previous work [Ritchey] I demonstrated the value of extending beyond a host-only vulnerability assessment. I created a modeling-based approach to network security that can be used to analyze the overall security of a network based on the interactions of vulnerabilities within a single host and within a network of hosts. This approach relies on model checking technology to analyze the resulting model to determine whether the network's security requirements are met or if there is a method that could be used to invalidate any of the requirements. Security requirements are encoded as assertions in the model checker. If the model checker can invalidate an assertion, it demonstrates this by showing the set of steps it followed to prove the assertion false. This sequence represents a potential path an attacker could use to gain access to the network.

In this paper I apply mutation operators to my network security model to produce new test cases. Mutant models that do not meet the defined security requirements (i.e. the change caused a security requirement to be invalidated) represent single configuration changes to the network that would result in a compromise of the network's security. By identifying which individual configuration changes result in network compromise, we significantly reduce the total number of system features that must be verified to assure the network's security.

2. An Overview of Model Checking

A model checking specification consists of two parts. One part is the model: a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value. The other part is temporal logic constraints over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic properties are satisfied over each possible path, that is, the model checker determines if the state machine is a model for the temporal logic formula. Model checkers exploit clever ways of avoiding brute force exploration of the state space [Birch]. If a property is not satisfied, the model checker attempts to generate a counterexample in the form of a trace or sequence of states.

The model checking approach to formal methods has received considerable attention in the literature. Tools such as SMV, SPIN, and Murø are capable of handling the state spaces associated with realistic problems [Clark]. I use the [SMV] model checker, which is freely available from Carnegie Mellon University and elsewhere. Although model checking began as a method for verifying hardware designs, there is growing evidence that model checking can be applied with considerable automation to specifications for relatively large software systems, such as TCAS II [Chan]. Model checking has been successfully applied to a wide variety of practical problems. These include hardware design, protocol analysis, operating systems, reactive system analysis, fault tolerance, and security [Holzmann].

The increasing usefulness of model checkers for software systems makes model checkers attractive tools for use in aspects of software development other than pure analysis, which is their primary role today. Model checkers are desirable tools to incorporate because they are explicitly designed to handle large state spaces and they generate counterexamples efficiently. Thus they provide a mechanism to avoid custom building these same capabilities into special purpose tools. For these reasons, I encode the security of a computer network in a finite state description and then write assertions in the temporal logic to the effect that "An attacker can never acquire certain rights on a given host." I then use the model checker to verify that the claim holds in the model or to generate an attack scenario against the network that shows how the attacker penetrates the system.

3. Discussion of Network Exploitation Methods

This section presents the network intrusion methodology that was used to develop the techniques presented in this paper.

3.1 Vulnerability

Breaking into a computer network requires that vulnerabilities exist in the network and that exploits for the vulnerabilities are known. Any network that an attacker has connectivity with will have some level of vulnerability. The goal of network security is to try to limit this vulnerability to the minimum required to accomplish the purpose of the network.

Network vulnerability is impossible to entirely eliminate. This is due to several factors. For a network to be useful it must offer services. These services are implemented in software and it is difficult to guarantee that any complex piece of software does not contain some flaws [Beizer]. These flaws frequently translate into security vulnerabilities. Sometimes, even when a security flaw is known, the operational need to offer a service with the vulnerability supercedes the need for the network to be totally secure. Network vulnerability may also be created by poor configuration. Given the large number of hosts on some networks, it is not surprising that some of them may not be set up to maximize their defenses. Many hosts are administered by the primary user of the system, who may lack the proper training to configure a secure computer system.

3.2 Exploitation

Before an attacker can attempt to break into a computer system several conditions must be met. An attacker must know a technique (e.g. exploit) that can be used to attempt the attack. However, knowing the exploit is not enough. Before an exploit can be used its preconditions must be met. These preconditions include the set of vulnerabilities that the exploit relies on, sufficient user rights on the target, sufficient user rights on the attacking host, and basic connectivity. The result of a successful exploit is not necessarily a compromised system; most exploits simply cause an increase in the vulnerability of the network. Results of a successful exploit could include discovering valuable information about the network, elevating user rights, defeating filters, and adding trust relationships among other possible effects. Most successful attacks consist of a series of exploits that gradually increase the vulnerability of the network until the prerequisites of the final exploit are met.

Network attackers normally start their work by searching for vulnerabilities on the hosts they can communicate with on the target's network. When a vulnerability is discovered they use it to increase the vulnerability level of the host. Once a host is compromised to the point that the attacker has some remote control of it, the host can then be used to launch attacks further into the network. This will more than likely include hosts that the attacker can not reach directly.

The attacker can use this new point of view to extend the number of hosts that can be searched for vulnerabilities; perhaps discovering new hosts that can eventually be taken over. This process can be continued until the network is fully compromised, the attacker can no longer find additional vulnerabilities to exploit or the attacker's goals are met.

4. Example Network

The purpose of this example is to provide a simple network structure to use for demonstrating the value of this analysis technique. It is composed of a small organization's network that includes a web server that they use to provide information to their customers. Due to budget constraints, public domain software is used throughout the network to reduce costs. The web server they have chosen to use is the widely used Apache web server [Apache]. They have installed the web server using the copy that was included on an

Figure 1

old RedHat Linux [RedHat] distribution. Because they are a small company they only maintain one network segment so the web server gets placed on the same segment as their file server. This network structure is shown in figure 1.

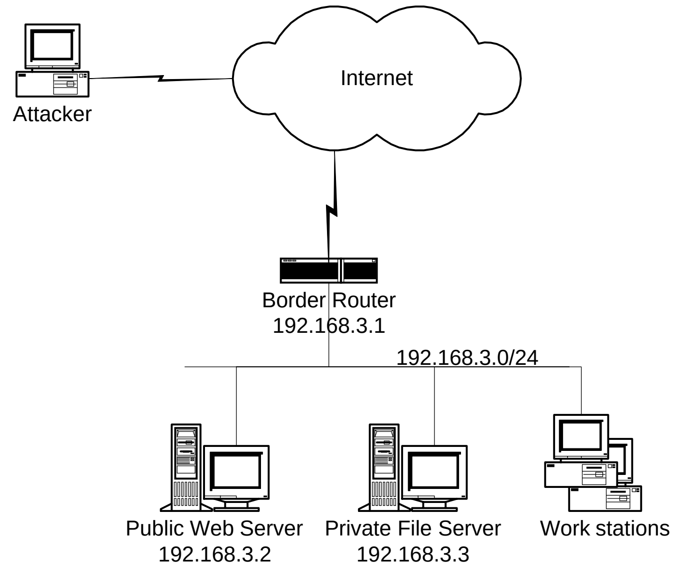

Figure 1

To protect this private server from the Internet they have installed packet filtering rules on their border router. These rules allow hosts on the Internet to connect to the web server, but not with the private server. Table 1 shows the filtering rules that are being enforced at the border router.

Source Address	Destination Address	Action
ANY	192.168.3.2	Allow
192.168.3.0/24	Not 192.168.3.0/24	Allow
ANY	ANY	Deny

Table 1. Border Filtering Rules

External users use web browsers to communicate with the public web server but they are not supposed to have any other access to the network. Private users rely on the private file server to hold their home directories that often contain company proprietary data. These directories are shared with the users of the network using Network File Service (NFS). They also occasionally use a custom database application located on the file server that they access by remotely logging in to the server using the rlogin command from their workstations.

5. An Overview of the Network Security Model

This section provides an overview of the network security model that was described in my previous network security analysis paper [Ritchey].

5.1 Composition of the Model

There are five major elements that make up our network security model.

- Hosts on the network including their vulnerabilities
- Connectivity of the hosts
- Current point of view of the attacker
- Exploit techniques that can be used to change the state of the model
- A list of security requirements the model should attempt to validate

Hosts represent potential targets for the attacker and have two major attributes, the attackers current access level on the host and the hosts set of vulnerabilities. A successful attack requires that vulnerabilities exist in the network and that exploits for the vulnerabilities are known. These vulnerabilities are broadly defined. In the model any fact about the host that could conceivably be required as a prerequisite for an exploit is a vulnerability. Taken from the example network, the web server would be described to the model as follows.

Public Web Server

Vulnerabilities		**Current Access Level**
Solaris version 2.5.1 Apache version 1.04 Telnetd Ftpd	Count.cgi Phf.cgi No shadow file Dtappgather	None

Table 2. Sample Host

This list includes vulnerability information such as the version of the operating system and web server, as well as the access level to the server that the attacker will begin with.

It is important that the model be able to model limited connectivity. Any network that an attacker has connectivity with will have some level of vulnerability. Because of this, a key security technique is network layer filtering. It is important for the model to be able to represent the connectivity between hosts that remains after all filters (firewalls) that exist between the hosts have been examined. To allow a simple example, the model represents connectivity between hosts as a matrix of boolean values. In the example network the router's filtering rules are represented by the following table.

	Attacker	**Border Router**	**Public Web Server**	**Private File Server**
Attacker	N/A	Yes	Yes	No
Border Router	Yes	N/A	Yes	Yes
Public Web Server	Yes	Yes	N/A	Yes
Private File Server	No	Yes	Yes	N/A

Table 3. Connectivity Matrix

Connectivity between different hosts will vary due to the different network filters. If a hacker can gain control of a host, the attacker may be able to launch attacks from the host. It is important to model the point of view of the attacker so that the set of hosts that are reachable by the attacker includes hosts reachable by hosts under the attacker's control. The model maintains a level of access field on each host. Any host that has a level of access higher than none may be used to launch some exploits. When determining which hosts the attacker can reach, the model checker looks for any hosts that are reachable from the union of all hosts with an access level greater than none.

Before an attacker can attempt to break into a computer system the attacker must know an exploit that can be used to attempt the attack. However, knowing the exploit is not enough. Before an exploit can be used its preconditions must be met. These preconditions include the set of vulnerabilities that the exploit relies on, sufficient user rights on the target, sufficient user rights on the attacking host, and basic connectivity. The model defines exploits by the set of vulnerabilities, source access level, target access level, and connectivity they require, plus the results they have on the state of the model if they are successful. Exploits are used by the model to affect changes to the security of the hosts under analysis. The quality and quantity of exploits encoded in the model have a direct relationship with the quality of the analysis that can be performed with the model. An example exploit included in the demonstration is the PHF.cgi program. This program shipped with several versions of the Apache web server and allowed a remote attacker the ability to execute programs on the host. Table 4 shows how the exploit was represented in the model.

PHF Exploit

Prerequisites	Source Access Level	Target Access Level	Results
(Apache versions up to 1.0.4 OR NCSA versions up to 1.5a) AND phf program	ANY	ANY	Access level changed to httpd

Table 4. Sample Exploit

Security requirements are written as invariant statements in the model checker's temporal logic formula language. Each security requirement needs to be formulated as a temporal logic formula. The full expressivity of temporal logic is not normally used. Most assertions take the form of events that should never happen. For example, a typical security requirement might be "An attacker can never access the Private File Server". In SMV this would be formulated as AG (PrivateFileServer.Access = None). If the model checker can reach any state where an invariant statement is false then we know that it is possible for an external attacker to violate one of our security requirements.

5.2 Execution of the Model

After the network has been described to the model checker and initialized, the model checker begins to determine whether the security assertions made about the model are true. The model checker starts by determining the set of hosts the attacker has connectivity with and non-deterministically chooses one. The model checker then tries to locate an exploit that can be used against the host. All prerequisite vulnerabilities for the exploit must exist on the target host. In addition, the prerequisite access levels (source and destination) must be met. If successful, the results of the exploit are used to change the state of the target by adding additional vulnerabilities and/or changing the attacker's access level on the host.

If an exploit is successful it reduces the overall security of the network. This is true because it may be possible to run other exploits against the host if the additional vulnerabilities added match the prerequisite vulnerabilities for another exploit. The attacker may also be able to communicate with new hosts if the attacker's access level on the target host is increased (thereby allowing the attacker to use the target host to launch attacks).

The model checker will continue to locate hosts to attack and will continue to search for valid exploits to use until all possibilities have been explored or all of the security assertions have been proven false. The results are counter examples for each disproved assertion and a validation for any remaining assertions.

6. Mutating the Model

Mutation analysis was originally a code-based analysis technique for automating the development of test cases [Offutt]. Recent applications of this technique have been used to design test cases from specifications. Here mutation analysis is applied to define test cases for network security.

Mutation analysis works by defining mutation operators that are used to create many versions of the original program (in this case model). Mutation operators are defined so that when each is applied it causes some small but significant change to the program. As an example, one mutation operator for code-level analysis changes a less-than comparison to a greater-than comparison.

An individual mutant operator is used to create each mutant version. Each version represents a mutant of the original program that has been intentionally flawed. Test cases are created and run against the mutants with the goal of causing the mutant version to fail. Test cases that cause a mutant to fail are said to *kill* the mutant. The ultimate goal is to design a test set that causes all generated mutant programs to fail. It is important to note that some mutants are impossible to kill. This occurs when the mutant is functionally equivalent to the original program.

6.1 Defining Mutation Operators

For network compromise to be possible, an exploit's prerequisites need to be met. This makes the exploit prerequisites an excellent source for defining our mutant operators. Potential operators include adding vulnerabilities, increasing access levels, and adding connectivity. Each of these changes would have the effect of making the model less secure than the original. There is no point in introducing mutations that remove vulnerabilities, connectivity, or access level as this would have the effect of making the mutant model more secure than the original model.

Each of these potential mutant operators represents a different real-world change that could occur to the network. However, not all changes that are possible in the model are likely in the network. It is important that we recognize what each mutant type represents in the actual network so that we can intelligently filter out inappropriate mutants.

Add connectivity is the simplest of the mutant operators. An add connectivity mutant demonstrates the effect of a change to a firewall's ruleset that allows more traffic past. This type of configuration change is common (though frequently unwise) on production networks. By adding connectivity to the model, we demonstrate what level of access could be gained by an attacker if the firewall configuration was changed. In the example network, there is only one restriction placed upon communication, the attacker cannot talk directly with the private file server. In this case, the add connectivity mutant would allow us to see the ramifications of allowing the attacker direct access to the private file server.

Increase access level is more difficult. Increase access level implies that an attacker is starting with more access than a typical external attacker would have. This could occur if the attacker was an insider or was given information by an insider. One problem in adding access to arbitrary hosts inside the network is the problem of connectivity. The attacker should only be able to use machines during an attack that can be communicated with by the attacker. Any host in the model that has an access level greater than none can be used by the model in the attack. Due to this, it is important that access levels on systems that the attacker cannot gain access to not be modified. As it is unclear at the beginning of the analysis which systems the attacker will eventually be able to communicate with, it is difficult to decide in advance which systems could be correctly mutated by increasing the attacker's access level. An alternative is to introduce a vulnerability that would guarantee the attacker increased access should the attacker gain connectivity to the host. This is easily accomplished by defining a new *super* vulnerability with its matching exploit. With this change, the add vulnerability mutant is sufficient to model the increase access mutant. From the example network, an increase access level mutant would grant the attacker user privileges on the web server, even though the attacker does not normally start with these privileges. This might be valuable if we assume that there is some method (yet unknown) that would allow the attacker to gain this access. This is actually a reasonable assumption as new exploitation methods are discovered daily.

The *add vulnerability* mutant models changes to the configuration of the target system. These could include adding software, changing permissions, modifying settings, etc. There are many changes that are likely to occur during the lifecycle of a system that would result in the addition of vulnerabilities. However, out of the pool of all vulnerabilities, there are even more vulnerabilities that would not be either likely or possible. A good example would be trying to add Unix specific vulnerabilities to a Windows NT system. These invalid vulnerabilities must be eliminated before the analysis is conducted.

In the example network the analysis to weed out invalid add vulnerability mutants was conducted manually. To be able to automate the elimination of unreasonable vulnerabilities requires that more be known about each vulnerability. A structure is required that would allow vulnerabilities to be categorized. One possible structure is shown in figure 2. In this structure, vulnerabilities are separated into four categories, devices, operating systems, software, and basic vulnerabilities. Vulnerabilities can only be applied if their parent vulnerability already exists. For each vulnerability (except devices), a parent vulnerability would need to exist before the vulnerability could be applied. For example, there is an elevation of privilege exploit that relies on a faulty version of a program called dtappgather. The vulnerable version of dtappgather was shipped with several versions of the Solaris operating system. Another way to say this is that the dtappgather vulnerability relies on the host running one of a set of particular versions of Solaris. So we model the dtappgather vulnerability as a software vulnerability that relies on Solaris 2.5, Solaris 2.5.1, or Solaris 2.6. If any host in the model is running any of these operating systems, it would be reasonable to add the dtappgather vulnerability.

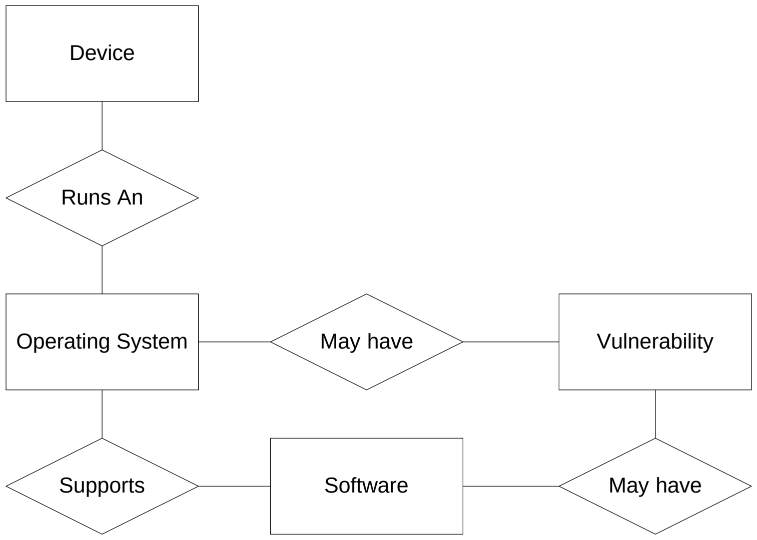

Figure 2

Another consideration is the likelihood of a particular configuration change. It is frequently uninteresting to consider a change in a device type, or an operating system. These types of changes occur very infrequently. In some circumstances, it may also not be interesting to see if adding any possible software package would invalidate the security. These categories of vulnerability should be automatically eliminated from consideration.

6.2 Coverage Criterion for Network Security

A well-designed and configured network should not produce any counter examples when analyzed by this system. However, this analysis is conducted at a particular place in time, and with a particular network configuration. It is unlikely that the network's configuration will remain static. Changes will be made periodically. These changes could introduce additional vulnerabilities that could invalidate the previous analysis. It would be useful to be able to extend beyond a spot check of the network's security and instead address specifically what changes should be avoided in the future to prevent undermining the network's security. This is the advantage of this mutation system.

By introducing a single vulnerability into the model and viewing any resulting counterexamples, the system can identify the individual configuration changes that could lead to network compromise. This would lead to a list of configuration changes that could be phrased as "if we change this parameter, then this security requirement will be invalidated". Extending beyond a single mutation per mutant gives us the ability to look for combinations of configuration changes that would undermine the network. For example, if we allow two mutant operators per mutant model, we could create lists of security issues that could be phrased as "if we change this parameter, and this parameter, then this security requirement will be invalidated".

This definition of network security mutation coverage is based upon the number of mutant operators that can be applied together to produce a counterexample free analysis of the network. If the coverage level is set to one, then the network should be able to have any single configuration value changed without undermining the security of the network. If the level is two, then the network can survive two configuration changes without fear of a compromise. The higher the number, the more difficult it would be for the network to be placed into an insecure state.

6.3 Running the Analysis

To perform the analysis, a network model is created that reflects the current configuration of the network. Security requirements are then encoded into the model in the form of the invariant statements. Next mutants of the model are generated based upon the different mutant operators. Depending upon the coverage level, different numbers of mutant operators may be applied to produce a single mutant model. If any mutants have been generated that are not relevant or reasonable in the context of our network, they are removed in the next step. Finally, the remaining mutant models are run through the model checker. If all mutants verify the security requirements (i.e. none produce counterexamples) the network meets the current coverage level. Otherwise, the network needs to be reconfigured to eliminate the source of the vulnerabilities and the analysis must be run again.

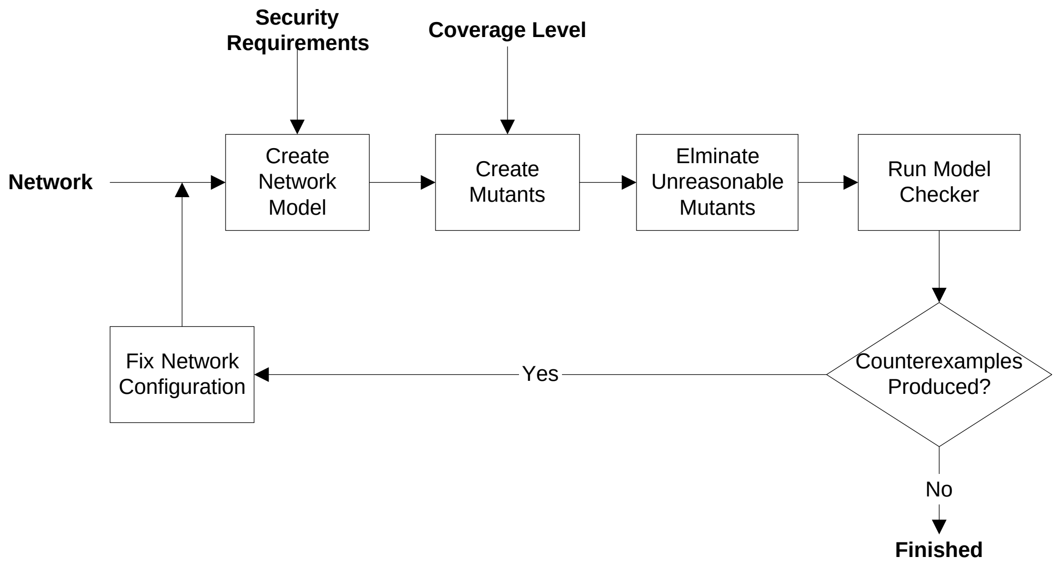

Figure 3

7. Results

A model of the example network was created and a level one analysis was conducted (only one mutation allowed per mutant). The result was a total of 30 different mutant models, one mutant created by an add connectivity mutation, and 29 created by add vulnerability mutations. Of these, 13 were judged to be unreasonable and were removed. This left 17 mutations that were analyzed by the model checker. Six of these models produced counter examples. By evaluating the first couple of steps in the counter examples produced, it is possible for a security analyst to create recommended configuration changes. These counter examples with sample security recommendations are shown in table 5.

Number	Mutant	1st Exploit	Security Recommendation
1	Add Connectivity from Attacker to PrivateFileServer	Add BSD trust from Attacker to PrivateFileServer	Eliminate BSD daemons on PrivateFileServer
2	Add PHF program to PublicWebServer	Use PHF to gain user access to PublicWebServer	Verify PHF not on PublicWebServer
3	Password Hashes known on PublicWebServer	Brute Force Passwords on PublicWebServer	Use strong authentication on PublicWebServer
4	Root password known on PublicWebServer	Telnet to PublicWebServer	Use strong authentication on PublicWebServer
5	BSD trust between Attacker and PublicWebServer	rlogin to PublicWebServer	Eliminate BSD daemons on PublicWebServer
6	User passwords known on PublicWebServer	Telnet to PublicWebServer	Use strong authentication on PublicWebServer

Table 5

8. Conclusions and Future Work

The significant contribution of this work is its ability to provide direct advice to assist security analyst to create highly secure networks. The results of this analysis point out the exact areas of the network that need to have additional protection. This automates the discovery of the factors that could lead to network compromise. In addition, the coverage criterion provides a unique and quantitative way to rate a network's security.

For this technique to be realized as a fully functioning tool several tasks need to be accomplished. First, the basic network modeling analysis this mutation technique relies upon needs to be fleshed out. This requires that the model be populated with an significant set of exploits, that a tool be created that scans for the vulnerabilities these exploits require, and that the connectivity model is extended to provide a more complete representation of TCP/IP functionality. In addition, each vulnerability in the model needs to be categorized, and the system features that the vulnerabilities relied upon needs to be recorded. Finally, the mutation engine needs to be automated so that it accepts as input the network model and the coverage level and either produces a list of mutants that violate the model or verifies that the model meets the coverage criterion.

9. References

[Apache] Apache Web Server information and software on the web at www.apache.com.

[Beizer] B. Beizer, “Software Testing Techniques, 2nd edition,” Thomson Computer Press, 1990.

[Birch] J. Birch, E. Clark, K. McMillan, D. Dill, and L.J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, *Proceedings of the ACM/SIGDA International Workshop in Formal Methods in VLSI Design*, January, 1991.

[Chan] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin, Model Checking Large Software Specifications, *IEEE Transactions on Software Engineering*, Vol. 24, No. 7, July 1998.

[Clark] E. Clark, O. Grumberg, and D. Long, Verification Tools For Finite-State Concurrent Systems, *A Decade of Concurrency – Reflections and Perspectives*, Springer Verlag, 1994.

[COPS] Computer Oracle and Password System (COPS) information and software on the web at ftp.cert.org/pub/tools/cops.

[Holzmann] G. Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering*, Vol 23, No 5, May 1997.

[ISS] Internet Security Systems, System Scanner information on the web at www.iss.net.

[Mayer] A. Mayer, A. Wool and E. Ziskind, Fang: A Firewall Analysis Engine, *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2000.

[NAI] Network Associates, CyberCop Scanner information on the web at www.nai.com/asp_set/products/tns/ccscanner_intro.asp.

[Offutt] J. Offutt, Practical Mutation Testing, *Twelfth International Conference on Testing Computer Software*, pages 99--109, Washington, DC, June 1995.

[RedHat] RedHat Linux information and software on the web at www.redhat.com.

[Ritchey], R. Ritchey and P. Ammann, Using Model Checking To Analyze Network Security, *2000 IEEE Symposium on Security and Privacy*, May 2000.

[SMV] SMV information and software on the web at www.cs.cmu.edu/~modelcheck.

[Zerkle] D. Zerkle and K. Levitt, NetKuang – A Multi-Host Configuration Vulnerability Checker, In *Proceedings of the Sixth USENIX Unix Security Symposium*, San Jose, CA, 1996.