A Framework for Protocol Vulnerability Condition Detection

Yuxin Meng
Computer Science department
City University of Hong Kong
Hong Kong, China
ymeng8@student.cityu.edu.hk

Lam-for Kwok
Computer Science department
City University of Hong Kong
Hong Kong, China
cslfkwok@cityu.edu.hk

***Abstract*—Intrusion detection system (IDS) detects an intrusion by comparing with its attack signatures. The generation of IDS signatures is based on the analysis of attack traffic, which is a result of exploiting vulnerabilities in a network protocol. Thus, the protocol analysis becomes an effective method to find out protocol vulnerabilities with regard to IDS. But the problem of protocol analysis in IDS is that how to detect all protocol vulnerability conditions in protocols. In this paper, we propose a novel framework to identify protocol vulnerability conditions by utilizing existing protocol analysis techniques. In particular, there are three major analysis steps in our framework: protocol semantic analysis, protocol implementation analysis and protocol state transition sub-condition analysis. In the final step of our framework, we illustrate the use of deletion, addition and modification operations with the purpose of generating all potential protocol vulnerability conditions from the normal protocol transition conditions. Experimental results show that this framework is encouraging and feasible.**

Keywords-intrusion detection; vulnerability analysis

I. INTRODUCTION

Rule-based intrusion detection and prevention systems (RIDS/RIPS) [1, 3] are mainly based on attack signatures to detect an attack. The attack signatures are stored in a rule database and updated to the latest version periodically. What is more, the generation of these attack signatures heavily depends on an exploit of vulnerability in a protocol. Take Snort [2] as an example, this lightweight RIDS monitors and analyzes the protocol packets (e.g., UDP, TCP, IP) according to its rules to alert and prevent intrusions. The common rule format of Snort is as blow:

Action-type protocol-type Source-ip Source-port -> Destination-ip Destination-port (content:"|attack signature|"; msg:"attack msg";)

For an ICMP DDoS attack by using tfn2k tool, the attack rule or signature can be produced in terms of the detected characteristic as below:

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS tfn2k icmp possible communication"; icmp_id:0; itype:0; content:"AAAAAAAAAA"; rev:5;)

The content “AAAAAAAAAA” in this rule is the attack signature (also called *characteristic*) for this exploit.

What is more, the vulnerabilities in a protocol can appear in different forms, e.g., the change of bit values or the change of packet sequence. In common cases, an attacker usually utilizes these forms of protocol vulnerabilities to do harm to the network security.

To identify protocol vulnerabilities, protocol analysis is a prevalent and effective method used in intrusion detection. The advantages of protocol analysis are listed below:

- Strong capability of vulnerability detection: protocol analysis does not only assist IDS to analyze network traffic in terms of protocol specification, but also has the ability to identify vulnerabilities during protocol implementation. For example, the input length and special characters checking and filtering.
- Target detection space reduction: protocol analysis lightens the analysis workload by cutting down the target number of protocol fields, e.g., searching for specific parts of packet rather than entire payload.

Problems. The coverage of signatures is the key problem for rule-based IDS in reducing the detection accuracy. The IDS signatures usually are easy for an attacker to evade by making some small modifications of the original message in a packet. For example, changing the size of variable-length fields or changing the field values in a packet with the purpose of mismatching the IDS signatures.

We argue that the coverage problem of IDS signatures stems primarily from the variants of vulnerabilities in a protocol. In particular, different forms of a vulnerability in a network protocol usually are caused by some minor changes of respective protocol vulnerability conditions. As a result, the ideal solution to this problem is finding out all potential protocol vulnerability conditions that lead the protocol state from a normal to an abnormal state.

Related work. The concept of modifying the software testing paths has been implemented in detecting software vulnerability conditions [17, 6, 8] and then help identifying software threats. Saxena et al. [18] introduced loop-extended symbolic execution that broadens the coverage of symbolic results with loops to find out the vulnerability conditions in programs. Our work attempts to make use of this concept of detecting software vulnerability conditions through creating various software testing paths into the detection of protocol vulnerability conditions by analyzing a protocol specification. We aim to make progress towards systematic detection of possible vulnerability conditions in network protocols by applying three operations to normal protocol state conditions.

Contribution. In this paper, we propose a framework to detect protocol vulnerability conditions by utilizing existing

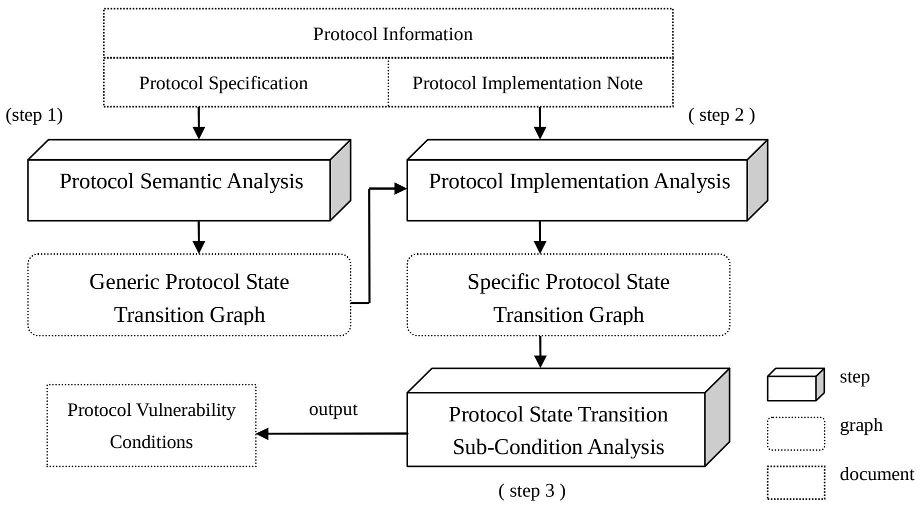
Protocol Information
Protocol Specification
Protocol Implementation Note
(step 1)
(step 2)
Protocol Semantic Analysis
Protocol Implementation Analysis
Generic Protocol State Transition Graph
Specific Protocol State Transition Graph
Protocol Vulnerability Conditions
output
Protocol State Transition Sub-Condition Analysis
(step 3)
step
graph
document

Figure 1. The framework for identification of protocol vulnerability conditions.

protocol analysis techniques with the goal of identifying all vulnerability conditions in network protocols. In particular, our framework has the ability to detect protocol vulnerability conditions without a real attack by applying the operations (deletion, addition and modification) onto normal transition conditions of protocol states.

To demonstrate the feasibility of our framework, we developed and evaluated the framework in an experimental environment which is constructed by Snort [2], Wireshark [8] and a packet generator [9]. Furthermore, we verified our framework by comparing our identified ICMP protocol vulnerability conditions with a set of ICMP Snort rules.

The rest of this paper is organized as follows: in Section 2, we introduce the steps in our framework that how to detect potential protocol vulnerability conditions and then give an in-depth description of operations in protocol state transition sub-condition analysis; Section 3 presents our experimental methodology and an experimental result; Section 4 states the future work; at last, Section 5 gives our conclusion.

II. Our Framework

In this section, we first give the definition of protocol vulnerability condition in our framework, and then introduce the representation format of protocol vulnerability condition.

Protocol vulnerability: a point that causes the execution of a protocol to be out of normal function and make errors.

Protocol vulnerability condition: A specific and certain condition that leads the protocol state to reach the protocol vulnerability which results in causing the protocol execution to an abnormal state.

In addition, the specific forms of protocol vulnerability conditions could consist of particular triggers (e.g., network parameters change, packet flag values reset) that cause a protocol vulnerability exploited, and abnormal points (e.g., implementation errors, coding ignorance etc.) that possibly compromise the function of a protocol.

What is more, we use the *IF/THEN* format to represent these protocol vulnerability conditions as the output in our framework. For example, as for Destination fragmentation vulnerability in ICMP (this vulnerability in ICMP packet due to the packet size is larger than 65536 octets, but the DF/Don’t fragmentation bit is set to 1), the representation of this protocol vulnerability condition could be (according to captured attack traffic):

IF {Datagram greater than 65536 octets and DF=1}
THEN {alert msg: ping of death attack}

This representation is compact and at protocol level. The merits of this representation (*IF/THEN*) are:

- Easy for understanding, the *IF* part describes the details of vulnerability conditions in protocols, the *THEN* part gives the description and information of this exploit.
- In favor of signature generation, it is comfortable for rule-based intrusion detection/prevention systems to produce attack rules and signatures according to the *IF/THEN* representation. In general, attack signature can be extracted from the *IF* part, and alert message is corresponding to the *THEN* part.

In the next two subsections, we first give details of the steps in our framework to account for the general procedure that how to detect protocol vulnerability conditions with high coverage by making use of current protocol analysis techniques. We then give an in-depth description on the use of operations (deletion, addition and modification) that how to identify potential protocol vulnerability conditions from known protocol state transition sub-conditions.

A. Framework Design

In Fig. 1, the framework illustrates that how to identify potential protocol vulnerability conditions by using protocol analysis techniques. The framework consists of three major steps: protocol semantic analysis, protocol implementation analysis and protocol state transition sub-condition analysis. The first step aims to produce a *generic protocol state transition graph* in terms of protocol specification (e.g., RFC [12], or use other intermediate language [10]); the second step considers the protocol implementation note to enrich and extend *generic protocol state transition graph* to a *specific protocol state transition graph*; and the goal of the last step is to produce potential protocol vulnerability conditions with operations (such as deletion, addition and modification) onto normal transition conditions of protocol states according to *specific protocol state transition graph* and then analyze the generated results to identify real protocol vulnerability conditions.

Protocol Information. According to the Fig. 1, this is the data source in our framework. The protocol information contains both essential details of Protocol Specification [4] and Protocol Implementation note [5].

- *Protocol Specification:* All semantic information of network protocols can be found in RFC (Request for Comments) [12], in which a series of documents that collect Internet information, UNIX and Software documents of Internet community. The extraction of protocol specification can be referred to previous work for details [4, 6, 7, 11]. In addition, it is useful and effective to find out lots of public specification in Vulnerability Database (e.g., NVD [13], CVE [14], OSVDB [16]).
- *Protocol Implementation Note*: This note contains the implementation details of network protocols (e.g., according to RFC documents, how to program the protocol). Moreover, we need to notice that the real implementation of a protocol may be changed a bit from the standard document due to the specific network environment and demands. We refer the reader to [5, 15, 17] for details of the extraction of protocol implementation note.

In practice, the network protocol implementations are usually distinct from the RFC documents due to practical environment. In this case, we could retrieve the essential protocol information with the method of protocol reverse engineering [6, 11, 19, 20], which is an effective method to find out the principles of a protocol by analyzing the relative structure, function, operation etc.

Step1: Protocol Semantic Analysis. The purpose of this step is to draw a generic protocol state transition graph by using protocol specification. State transition graph [20] (also called state transition diagram) is a graph that indicates the relationship between two states indicating that an object will take certain actions from the first state to the second state. Obviously, protocol state transition graph (PSTG) is a particular instance of the state transition graph in network protocols.

Generic protocol state transition graph (GPSTG). The term of *generic* means that this graph only contains indispensable protocol states and fewer transition conditions which shows the generic transition relationship among protocol states. This generic protocol state transition graph is easier to be drawn according to protocol specification, since there is no need to identify all specific state transition conditions in practical scenarios.

Step2: Protocol Implementation Analysis. This step aims for generating a specific protocol state transition graph by using protocol implementation note and relevant generic protocol state transition graph. Actually, the generation of the specific protocol state transition graph heavily depends on the information in protocol implementation note from which the specific protocol state transition conditions can be indicated.

Specific protocol state transition graph (SPSTG). The term of *specific* means that this graph contains all protocol states, as well all specific protocol state transition conditions. From another view, the SPSTG is an extended graph that emerges from GPSTG by utilizing much more information provided by the protocol implementation note.

Step3: Protocol State Transition Sub-Condition Analysis. The main purpose of this step is to analyze sub-conditions in the specific protocol state transition graph and generate potential protocol vulnerability conditions.

To facilitate the illustration of this analysis work, we give definitions of atom condition and compound condition.

Atom Condition: a certain kind of condition that cannot be decomposed further in semantic level (e.g., {DF=1}, {Datagram greater than 65535 octets}).

Compound condition: a kind of condition that consists of more than one atom condition (e.g., {Datagram greater than 65536 octets and DF=1}).

In this step, the analysis work falls into three types. The first type is to analyze the specific protocol state transition graph and divide the protocol state transition conditions into sub-conditions until all atom conditions are identified within each protocol state transition.

The second type is to pick up overlap atom conditions since these conditions affects more than one state transition. As a result, these overlap conditions are more likely to be utilized to create protocol vulnerability conditions.

The last type of analysis work is to operate (delete, add and modify) on these atom conditions (not only the overlap atom conditions but also the other atom conditions) to form potential protocol vulnerability conditions. To delete, add or modify an atom condition has the possibility to change the contents of relevant compound condition and cause a flaw in the state transition. In this case, these changed conditions (which may cause a flaw in protocol state transitions) are protocol vulnerability conditions that an attacker can utilize.

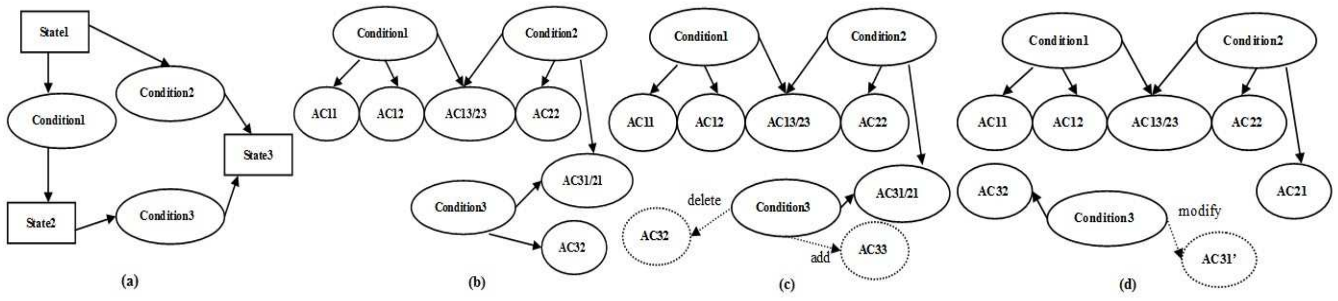

Figure 2. Operations of deletion, addition and modification.

B. *Operations in Protocol State Transition Sub-Condition Analysis*

In Fig. 2, we assume a specific protocol state transition graph and then illustrate the operations (deletion, addition and modification). In Fig. 2 (a), a specific protocol state transition graph is assumed that State1 can change to State2 and State3 if Condition1 and Condition2 are satisfied respectively. State2 converts to State3 as long as Condition3 is fulfilled. In Fig. 2 (b), we assume that Condition1 could be divided into three atom conditions {AC11, AC12, AC13}. Similarly, Condition2 and Condition3 have atom conditions {AC21, AC22, AC23} and {AC31, AC32} respectively.

Look for overlap atom conditions. As shown in Fig. 2 (b), AC13 is the same as AC23, and AC31 is equal to AC21. Thus, AC13/AC23 and AC31/AC21 are the overlap atom conditions. The advantage of finding out these overlap atom conditions is that these overlap atom conditions have more chances to affect the state transitions among State1, State2 and State3. Moreover, the overlap atom conditions are used in the addition operation in our framework. We then define the three operations of deletion, addition and modification.

Deletion of atom conditions. According to a specific protocol state transition graph, deleting or omitting an atom condition in a relevant compound condition could change the original contents of this compound condition and result in a fault or error during the state transition. As shown in Fig. 2 (c), if an atom condition AC32 is deleted, some errors may be occurred in the protocol state transition between State2 and State3.

Addition of atom conditions. Similar to deletion, adding an atom condition to a state transition condition may cause the state to an abnormal state as well. In Fig. 2 (c), adding an atom condition AC33, the State2 could do not know how to deal with the additional condition and thus produces a flaw during the protocol state transitions. In our framework, we use the overlap atom conditions for addition operation.

Empirically, the overlap atom conditions are much more vulnerable in the protocol state transitions. To ensure the feasibility and effectiveness of this operation, we thus utilize the overlap atom conditions to generate potential protocol vulnerability conditions for the operation of addition in our framework. For example, the AC13/23 and AC31/21 are overlap atom conditions according to Fig. 2 (b), we then can attempt to add AC13/23 to Condition3 instead of the AC33 (which is only a generic atom condition) to guarantee that our produced conditions are limited. Like this, we can add AC31/21 to Condition1 in evaluating the effects of these changes as well.

Modification of atom conditions. To change the original content of an atom condition is an effective way to cause some errors in the protocol state transitions. In Fig. 2 (d), if we modify AC31 to AC31' in Condition3, the errors may be caused between the protocol state transition between State2 and State3. The major modification skill is to set the packet bit values to an opposite value or another different value. For example, if DF=0 in original ICMP packet, the DF bit will be set to 1 (DF=1) during the modification.

If deleting, adding or modifying an atom condition can cause a normal protocol state to an abnormal state, this atom condition or the relevant compound condition is regarded to a real protocol vulnerability condition.

The merits of these operations are: 1) the effects and results of applying these operations onto atom conditions could be able to cover all possible forms of protocol vulnerability conditions (which may consist of an atom condition or more than one atom condition). Namely, these operations have the ability to generate all potential protocol vulnerability conditions; 2) it is more likely to identify the variants of known exploits and has the chance to discover unknown vulnerability conditions by applying these three operations and analyzing the produced conditions.

III. EVALUATION

In this section, we evaluated our framework in a constructed environment by using existing tools such as Snort [2], Wireshark [8] and a packet generator [9]. The environment is shown in Fig. 3.

In the following parts, we begin by discussing our experimental methodology to explain that how to launch our evaluation and achieve the results. Then we show the results

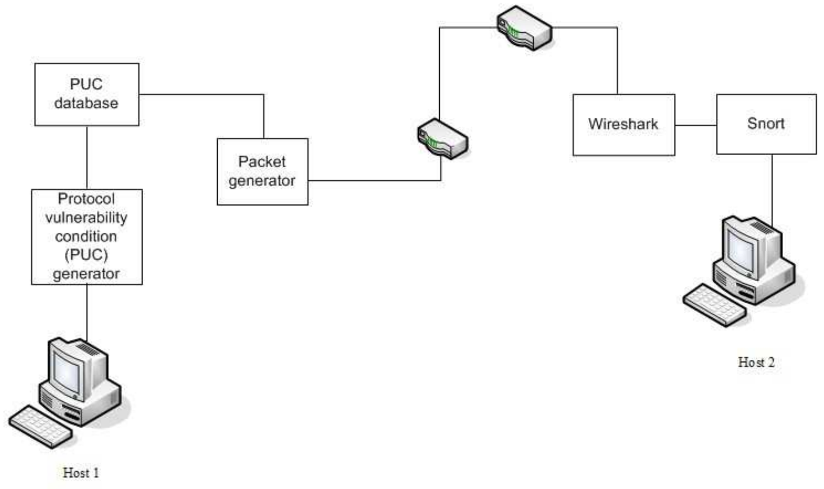

Figure 3. Experimental environment and deployment.

of our experiment on the protocol of ICMP to demonstrate the feasibility of our framework.

A. Experimental Methodology

In Fig. 3, we developed a protocol vulnerability condition generator (the input is protocol atom condition, thus we should draw SPSTG in advance) in Host 1 that generates a majority of potential protocol vulnerability conditions with deletion and addition operations. But as for the modification operation, it is more laborious to obtain the results. The PUC database provides a passive storage space for the generated potential protocol vulnerability conditions. Then, we used a packet generator to create packets according to the potential vulnerability conditions in the PUC database (e.g., setting the bit value in the packet to an opposite or different value, changing the sequence of bits). At last, the crafted packets are sent to the Host 2 through routers.

The Host 2 is the target for the experiment. Therefore, we deployed two open source tools Wireshark and Snort into this host with the purpose of monitoring network traffic and detecting abnormal packets that come from Host 1. The Wireshark is a powerful tool to capture network traffic and perform packet analysis while the objective of Snort is to detect abnormal packets according to its rules and signatures.

In this experiment, we used Snort rule database (version 2.8) to verify our identified potential protocol vulnerability conditions. In practice, we evaluate our framework on ICMP by comparing our identified protocol vulnerability conditions with the ICMP Snort rules. Based on these conditions, we can create specific ICMP packets to challenge the Snort. In this case, if the number and contents of our detected potential protocol vulnerability conditions can cover all of the ICMP Snort rules, we can show that our framework is feasible.

B. Analysis with an Example on the detection of protocol vulnerability conditions

Based on our experimental methodology, we give an example to illustrate the experiment on ICMP. According to the *description* part of page 5 in RFC 792 [12], we produce normal transition conditions (a compound condition) manually that trigger State1 (send packet) to State2 (wait for response). The conditions have then been divided into three sub-conditions: SubC1 {the distance to the network is finite}, SubC2 {indicated protocol module or process port is active} and SubC3 {datagram need fragmented and DF=0}.

Furthermore, we identify atom conditions as follows: {distance finite}, {protocol module is active}, {process port is active}, {datagram must be fragmented} and {DF=0}. To better understanding, all these atom conditions are presented at semantic level. There are no overlap atom conditions in this example since there are only two protocol states.

Subsequently, we apply operations (deletion, addition and modification) into these atom conditions to affect related transition conditions.

Deletion: delete any one or more above atom conditions. For instance, deleting {distance finite}, the remaining conditions will be {SubC2 and SubC3}. If deleting two atom conditions such as {distance finite} and {process port is active}, the result is {protocol module is active and SubC3}.

Addition: this operation is based on expert knowledge to some degrees. In our method, the atom condition for addition comes from the overlap atom conditions. Since there is no overlap condition in this example, we can skip this operation.

Modification: this action aims to change the contents of atom conditions. For example, atom condition {distance finite}, {DF=0} could be modified to {distance infinite}, {DF=1}. The purpose of these changes is to reverse the meaning of atom conditions or to set condition values to another different value.

We use the *IF/THEN* to represent the generated potential protocol vulnerability conditions. For instance, during the experiment, we can detect a protocol vulnerability condition through modifying the atom condition {DF=0} to {DF=1} in affecting Sub3. Thus, the representation of this protocol vulnerability condition in our framework is:

IF {datagram must be fragmented and DF=1},
THEN {alert msg: host crack down}.

In the experiment, these oversized packets can cause the host to be crashed or rebooted since the host does not know how to deal with these oversized packets. What is more, we find out the relevant rule in Snort rule database to verify this is a real protocol vulnerability condition. The snort rule is:

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Destination Unreachable Fragmentation Needed and DF bit was set"; icode:4; itype:3; reference:cve,2004-0790;reference:cve,2005-0068; classtype:misc-activity; sid:396; rev:7;)

The meaning of this rule is to alert that ICMP message needs fragmentation but the Don't Fragment flag is on, which is the same to our detected protocol vulnerability condition.

Performance Analysis. In the experiment, we evaluated our framework by the use of 115 ICMP Snort rules (in the *icmp.rules* and *icmp-info.rules* folders). In particular, we classified the ICMP Snort rules into 7 types such as ICMP fragmentation, ICMP ping, ICMP redirect, ICMP parameter problem, ICMP unreachable problem, ICMP TTL and ICMP conversion error.

The experimental results on the protocol of ICMP show that our detected ICMP vulnerability conditions cover 100% of the ICMP Snort rules except for those software oriented ICMP rules. We discovered that one protocol vulnerability condition at protocol level could cover more than one Snort rule since the Snort rules are very specific at byte-level. That is, our approach can generate some new exploits based on the variations of the protocol vulnerability conditions at byte level. By analyzing attack patterns of these new exploits, we might be able to discover more new attack signatures, and thus the Snort rules, before the new attacks actually arrive.

IV. FUTURE WORK

While the evaluation demonstrates the analysis steps in our framework, it does reflect some limitations which we could work in the future. First of all, the number of protocol vulnerability conditions increases exponentially by using the three operations. The benefits are high vulnerability coverage and unknown protocol vulnerability condition detection, but it does need much more storage space and is hard to avoid redundant conditions that have the same effects as well. To overcome this issue, we plan to develop a reference engine to correlate these potential protocol vulnerability conditions with the goal of reducing the unreasonable conditions. A second area for future work is the design of a system to generate byte-level IDS rules from the detected protocol vulnerability conditions automatically. In addition, we plan on applying our framework into other network protocols to further evaluate its feasibility.

V. CONCLUSION

In this paper, we proposed a framework aiming to detect all protocol vulnerability conditions in a network protocol. There are three steps in our framework: protocol semantic analysis, protocol implementation analysis and protocol state transition sub-condition analysis. In particular, we describe the relationship among these steps of the framework and define three operations of deletion, addition and modification in the final step that are used to generate potential protocol vulnerability conditions from normal transition conditions of protocol states. In the experiment, we develop and evaluate our framework on the protocol of ICMP in a constructed network environment by using Snort, Wirewark and packet generator. The experimental results show that our framework is feasible and encouraging.

REFERENCES

[1] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23-24), pp. 2435–2463 (1999).

[2] Snort, http://www.snort.org/. (access on 1 Apr. 2011)

[3] Scarfone, K. and Mell, P.: Guide to Intrusion Detection and Prevention Systems (IDPS), NIST Special Publication 800-94, pp. 1-127, (Feb 2007).

[4] Comparetti, P.M., Wondracek, G., Kruegel, C., and Kirda, E.: Prospex: Protocol Specification Extraction. In: IEEE Symposium on Security and Privacy, pp. 110-125, Oakland, CA (2009).

[5] Yating H., Guoqiang S., and Lee, D.: A model-based approach to security flaw detection of network protocol implementations. In: IEEE International Conference on Network Protocols, pp. 114-123, Orlando, Florida, USA (2008).

[6] Cui, W., Peinado, M., Chen, K., Wang, H.J., and Irun-Briz, L.: Tupni: Automatic reverse engineering of input formats. In: ACM Conference on Computer and Communications Security, pp. 1-12, VA (2008).

[7] Wondracek, G., Comparetti, P.M., Kruegel, C., and Kirda, E.: Automatic network protocol analysis. In: Network and Distributed System Security Symposium, pp. 1-14, San Diego, CA (2008).

[8] Wireshark, http://www.wireshark.org. (access on 21 Mar. 2011)

[9] Colasoft Packet Builder, http://www.colasoft.com/packet_builder/. (access on 21 Mar. 2011)

[10] Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., and Guo, C.: Generic application-level protocol analyzer and its language. In: Network and Distributed System Security Symposium, pp. 1-15, San Diego, CA (2007).

[11] Lin, Z., Jiang, X., Xu, D., and Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Network and Distributed System Security Symposium, San Diego, CA (2008).

[12] Request for Comments (RFC), http://www.ietf.org/rfc.html. (access on 1 Apr. 2011)

[13] National Vulnerability Database (nvd): http://nvd.nist.gov. (access on 1 Apr. 2011)

[14] Common vulnerabilities and exposures (cve): http://cve.mitre.org. (access on 1 Apr. 2011)

[15] Shu, G. and Lee, D.: Testing Security Properties of protocol implementations: a machine learning based approach. In: Proceedings of IEEE ICDCS, pp. 25-32, Toronto, Canada (2007).

[16] Open Source Vulnerability Database (osvdb), http://osvdb.org/. (access on 1 Apr. 2011)

[17] Peled, D., Vardi, M. Y., and Yannakakis, M.: Black-box checking. In: Proceedings of IFIP FORTE/PSTV, pp. 225-240 (1999).

[18] Saxena, P., Poosankam, P., McCamant, S., and Song, D.: Loop-extended symbolic execution on binary programs. In: International Symposium on Software Testing and Analysis, pp. 225-236, Chicago, IL (2009)

[19] Cui, W., Kannan, J., and Wang, H.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In: 16th Usenix Security Symposium, pp. 199-212, Boston, MA (2007).

[20] Guha, B. and Mukherjee, B.: Network security via reverse engineering of TCP code: vulnerability analysis and proposed solutions. IEEE Network, pp. 40-48 (Jul/Aug 1997).