International Journal of Electronics, Communication & Soft Computing Science and Engineering
ISSN: 2277-9477 Volume 2, Issue 2

Signature Free Buffer Overflow Attack Detection in Peer to Peer Network.

Mr.J.P.Mehare, Prof.V.S.Gulhane

***Abstract*— SigFree - online signature-free out-of-the-box application-layer method for blocking code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. SigFree is signature free, thus it can block new and unknown buffer overflow attacks. We show that the attack packets tested in our experiments with very few false positives.**

***Keywords*- Intrusion detection, Buffer overflow attacks, code injecting attacks**

I. INTRODUCTION

Throughout the history of cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly desired requirements: (R1) simplicity in maintenance; (R2) transparency to existing (legacy) server OS, application software, and hardware; (R3) resiliency to obfuscation; (R4) economical Internet-wide deployment. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

Existing defenses are limited in meeting these four requirements. Existing buffer overflow defenses are categorized into six classes. (1) Finding bugs in source code. (2) Compiler extensions. (3) OS modifications. (4) Hardware modifications. (5) Defense-side obfuscation.(6) Capturing code running symptoms of buffer overflow attacks.

To overcome the above limitations, in this paper, we propose SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code". In particular, as summarized 1) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434), which are used to monitor Microsoft SQL Databases, accept data only. 2) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306, and 5432 accept data only

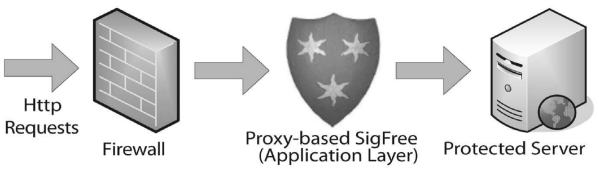

Fig. 1 SigFree is an application layer blocker between the protected server and the corresponding firewall

Accordingly, SigFree (Fig.1) works as follows: SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new OðNÞ algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded (if any). However, in this phase, some data bytes may be mistakenly decoded as instructions

In phase 2, SigFree uses a novel technique called code abstraction. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions or dependence degree to a threshold to determine if this instruction sequence contains code. Unlike the existing code detection algorithms that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade.

II. PROPOSED WORK

The proposed work consists concept of prevention and detection of buffer overflows. The work proposes SigFree, a real-time, signature-free, out-of the-box, application layer blocker for preventing buffer overflow attacks, one of the most serious cyber security threats. SigFree can filter out code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. SigFree

first blindly dissembles and extracts instruction sequences from a request. It then applies a novel technique called code abstraction, which uses data flow anomaly to prune useless instructions in an instruction sequence. Finally it compares the number of useful instructions to a threshold to determine if this instruction sequence contains code. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is transparent to the servers being protected, it is good for economical Internet wide deployment with very low deployment and maintenance cost.

Detection of Data Flow Anomalies: There are static or dynamic methods to detect data flow anomalies in the software reliability and testing field. Static methods are not suitable in our case due to its slow speed; dynamic methods are not suitable either due to the need for real execution of a program with some inputs.

Their scheme is rule-based, whereas SigFree is a generic approach which does not require any pre-known patterns. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to see if the packet really contains code Four rules (or cases) are discussed in their project: Case 1 not only assumes the occurrence of the call/jmp instructions, but also expects the push instruction appears before the branch; Case 2 relies on the interrupt instruction; Case 3 relies on instruction ret; Case 4 exploits hidden branch instructions. Besides, they used a special rule to detect polymorphic exploit code which contains a loop. Although they mentioned that the above rules are initial sets and may require updating with time, it is always possible for attackers to bypass those pre-known rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, SigFree exploits a different data flow analysis technique, which is much harder for exploit code to evade.

III. IMPLEMENTATION

There are various tools available for simulating different network models. Ns2/ns3, OPNET and NetSim are some of the tools that can be used for the simulation of the various network architectures and models. The ns-2 simulator is a discrete-event network simulator targeted primarily for research and educational use. The ns-2 is written in C++. ns-2 is open-source, and the project strives to maintain an open environment for researchers to contribute and share their software.

Ns-2 is scripted in OTcl and results of simulations can be visualized using the Network Animator nam. It is not possible to run a simulation in ns-2 purely from C++ (i.e., as a main() program without any OTcl). Moreover, some components of ns-2 are written in C++ and others in OTcl. Considering these features of ns-2 **ns-allinone-2.34** is used for the implementation of the proposed dissertation work. Initially **ns-allinone-2.34** is downloaded from [10]. NS-2 is designed to run from on most UNIX based operating systems. It is possible to run NS-2 on Windows machines using Cygwin. If you don't have a UNIX install, you can also use a virtual linux machine and run that under Windows. In the dissertation work the **Fedora core 13** operating system is used for installation and configuration of the **ns-2.34**. The **ns-2.34** is configured on the path /home/project/Desktop/project/. For configuring the ns the following commands are executed in the terminal. Before configuration we should make sure that we have standard development packages like 'make' and 'gcc'.

tar -xzf ns-allinone-2.34.tar.gz

cd ns-allinone-2.34

./install

After the execution of the above commands on terminal if everything is fine without any errors then we will get following messages on the terminal.

Ns-allinone package has been installed successfully.
Here are the installation places：
tcl8.4.11:/home/jayant/Desktop/project/ns-allinone-2.34/{bin,include ,lib}
tk8.4.11:/home/jayant/Desktop/project/ns-allinone-2.34/{bin, include, lib}
otcl: /home/jayant/Desktop/project/ns-allinone-2.34/otcl-1.11
tclcl: /home/jayant/Desktop/project/ns-allinone-2.34/tclcl-1.17
ns: /home/jayant/Desktop/project/ns-allinone-2.34/ns-2.29/ns
nam: /home/jayant/Desktop/project/ns-allinone-2.34/nam-1.11/nam
xgraph: /home/project/Desktop/project /ns-allinone-2.34/xgraph-12.1
gt-itm: /home/jayant/Desktop/project/ns-allinone-2.34/itm, edriver, sgb2alt, sgb2ns, sgb2comns, sgb2hierns

---

Please put /home/jayant/Desktop/project/ns-allinone-2.34/bin:/home/jayant/Desktop/project/ns-allinone-2.34/tcl8.4.18 /unix:/home/jayant/Desktop/project/ns-allinone-2.34/tk8.4.18/unix

into your PATH environment; so that you'll be able to run itm/tclsh/wish/xgraph.

IMPORTANT NOTICES:

(1) You MUST put /home/jayant/Desktop/project/ns-allinone-2.34/otcl-1.13:/home/jayant/Desktop/project/ns-allinone-2.34/lib,

into your LD_LIBRARY_PATH environment variable.
If it complains about X libraries, add path to your X libraries
into LD_LIBRARY_PATH.
If you are using csh, you can set it like:
setenv LD_LIBRARY_PATH <paths>
If you are using sh, you can set it like:
export LD_LIBRARY_PATH=<paths>

(2) You MUST put /home/jayant/Desktop/project/ns-allinone-2.34/tcl8.4.18/library into your TCL_LIBRARY environmental
variable. Otherwise ns/nam will complain during startup.

(3) [OPTIONAL] To save disk space, you can now delete directories tcl8.4.11
and tk8.4.11. They are now installed under /home/jayant/Desktop/project/ns-allinone-2.34/{bin,include,lib}

After these steps, you can now run the ns validation suite with
cd ns-2.34; ./validate

For trouble shooting, please first read ns problems page http://www.isi.edu/nsnam/ns/ns-problems.html. Also search the ns mailing list archive
for related posts.

For the validation of the installation we can run the next command on terminal as

cd ns-2.34
./validate

After successful installation we need to modify some of the environment variables namely PATH, LD_LIBRARY_PATH and TCL_LIBRARY. To accomplish this the following commands are executed on the terminal.

export set
PATH=$PATH:/home/project/Desktop/project/ns-allinone-2.34/bin:/home/project/Desktop/project/ns-allinone-2.34/tcl8.4.18/unix :/home/project/Desktop/project/ns-allinone-2.34/tk8.4.18/unix

export set
LD_LIBRARY_PATH=/home/project/Desktop/project/ns-allinone-2.34/otcl-1.13:/home/project/Desktop/project/ns-allinone-2.34/lib

export set
TCL_LIBRARY=/home/project/Desktop/project/ns-allinone-2.34/tcl8.4.18/library

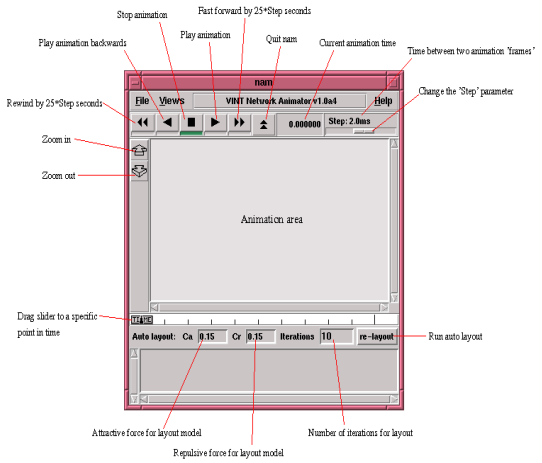

Fig 2: Network Animator used for viewing the simulation results

Whenever we want to run any simulation we need to set these variables. We can start ns with the command 'ns <tclscript>' (assuming that we are in the directory with the ns executable, or that our path points to that directory), where '<tclscript>' is the name of a Tcl script file which defines the simulation scenario (i.e. the topology and the events). We could also just start ns without any arguments and enter the Tcl commands in the Tcl shell, but that is definitely less comfortable
We can either start nam with the command 'nam <nam-file>' where '<nam-file>' is the name of a nam trace file that was generated by ns, or execute it directly out of the Tcl simulation script for the simulation which we want to visualize. Below we can see a screenshot of a nam window where the most important functions are being explained.

After the successful configuration of ns environment for the implementation of the proposed dissertation work initially we have created **bod** (Buffer Overflow Detector) package inside **ns-2.34** which defines various properties of the data in terms of packet which are transferred to and from

various node which can be identified as packets which are using implemented **bod** protocol for buffer overflow attack detection in the network. The implemented algorithm which is written inside **bod.cc** is situated inside the router through which packets are transferred and filtered. It also includes different parameters which are defined for the implementation of the bod system which includes different types of packet which is transferred between different nodes. We have generated large number of attack packets and attacks which are categorized as DoS attack and some of the packets as normal packets. Inside bod we have defined four files **bod.cc, bodPacket.cc, bod.h and bodPacket.h**. After adding bod into **ns-2.34** we need to modify some of the files of ns environment which are **ns-2.34/common/packet.h**, **ns-2.34/tcl/lib/ns-packet.tcl** and **ns-2.34/Makefile.** After these changes we need to again execute make command on the terminal to reflect the changes in the ns environment.

IV. RESULT

The result of implemented work is carried out by different simulations which are implemented to demonstrate the different ways of buffer overflow attack detection in the different types of network architecture. Initially simnids.tcl is implemented to demonstrate network buffer overflow detection system where there are two attacker nodes, one sender node and one receiving node. For the execution of this tcl script initially all the environment variables are set and the following command is executed on the terminal.

ns /home/jayant/simbod.tcl

After the execution out.nam file is created inside the current working directory and we get the following output on the terminal.

Sending L...Sent in 5.000000 Seconds.

Sending K...Sent in 8.000000 Seconds.

Sending B...Sent in 11.000000 Seconds.

Sending B...Sent in 14.000000 Seconds.

Sending E...Sent in 17.000000 Seconds.

Sending L...Sent in 20.000000 Seconds.

Sending K...Sent in 23.000000 Seconds.

Packet received

Transaction (set D) ==> asdfg

Buffer Overflow Attack Detected

Detected in 4.000000 Seconds.

Packet received

Transaction (set D) ==> jkhgasdfg

DoS attack

Detected in 3.000000 Seconds.

Packet received

Transaction (set D) ==> locnnmasdfghf

Buffer Overflow Attack Detected

Detected in 4.000000 Seconds.

Packet received

Transaction (set D) ==> oqwdjhkcmd

SET L1={ (a,3) (s,3) (d,5) (f,4) (g,4) (j,2) (k,2) (h,3) (o,2) (c,2) (n,2) (m,2) }

Ln={ (sda,0) (sds,0) (sdd,0) (sdf,0) (sdg,0) (sdj,0) (sdk,0) (sdh,0) (sdo,0) (sdc,0) (sdn,0) (sdm,0) (asd,0) (ssd,0) (dsd,0) (fsd,0) (gsd,0) (jsd,0) (ksd,0) (hsd,0) (osd,0) (csd,0) (nsd,0) (msd,0) }

Cn={ (sdf,3) (asd,3) }

Cn={ (sdfg,3) (asdf,3) }

Cn={ (asdfg,3) }

It can be observed from the output buffer overflow and DoS attacks are detected in the network. The DoS attack is detected when there is packet loss that means the packet doesn't receive to the destination node. We can run the simulation by executing the following command on the terminal.

nam out.nam

After execution the output is generated inside the network animator. Figure shows the network with two attackers' one source node and one destination node which receive the packets from these attacker nodes. The intermediate node is the router which detects whether a packet is attack packet or simple packet.

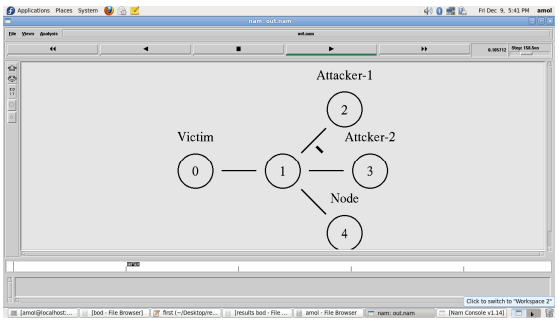

Fig 3 Network with One Source Node, One Destination Node and Three Attacker Nodes.

After running the modified data simulation we get the following output on the terminal.
ns /home/jayant/simbodnew.tcl

Sending K...Sent in 5.000000 Seconds.

Sending E...Sent in 8.000000 Seconds.

Sending B...Sent in 11.000000 Seconds.

Sending C...Sent in 14.000000 Seconds.

Sending K...Sent in 17.000000 Seconds.

Sending E...Sent in 20.000000 Seconds.

Packet received

Transaction (set D) ==> asdfg

Sending E...Sent in 5.000000 Seconds.

Sending B...Sent in 8.000000 Seconds.

Sending C...Sent in 11.000000 Seconds.

Sending C...Sent in 14.000000 Seconds.

Sending B...Sent in 17.000000 Seconds

Packet received

Transaction (set D) ==> jkhgasdfg

Sending K...Sent in 5.000000 Seconds.

Sending L...Sent in 8.000000 Seconds.

Sending K...Sent in 11.000000 Seconds.

Sending P...Sent in 14.000000 Seconds.

Sending E...Sent in 17.000000 Seconds.

Sending K...Sent in 20.000000 Seconds.

Sending P...Sent in 23.000000 Seconds
Packet received

Transaction (set D) ==> locnnmasdfghf

Buffer Overflow Attack Detected

Detected in 4.000000 Seconds.

Packet received

Transaction (set D) ==> oqwdjhkcmd

SET L1={ (a,3) (s,3) (d,5) (f,4) (g,4) (j,2) (k,2) (h,3) (o,2) (c,2) (n,2) (m,2) }

Ln={ (sda,0) (sds,0) (sdd,0) (sdf,0) (sdg,0) (sdj,0) (sdk,0) (sdh,0) (sdo,0) (sdc,0) (sdn,0) (sdm,0) (asd,0) (ssd,0) (dsd,0) (fsd,0) (gsd,0) (jsd,0) (ksd,0) (hsd,0) (osd,0) (csd,0) (nsd,0) (msd,0) }

Cn={ (sdf,3) (asd,3) }

Cn={ (sdfg,3) (asdf,3) }

Cn={ (asdfg,3) }

The following table indicates various types of attacks that are generated in the network.

Generated in Simulation of	Type of Attack		
	Normal	Buffer Overflow	DoS
ns simnids.tcl	7	2	1

Table 1: Attacks detected by the implemented buffer overflow attack detection approach.

The values of precision and recall for the implemented buffer overflow detection system are shown in the table.

The precision and recall is calculated by the following formulae.

Precision

$$= \frac{\text{Total Attacks in Network} \cap \text{Total Attacks Detect}}{\text{Total Attacks Detected}}$$

Recall

$$= \frac{\text{Total Attacks in Network} \cap \text{Total Attacks Detect}}{\text{Total Attacks in Network}}$$

CONCLUSION

As the most serious attacks like code injection attacks, DoS attacks etc are most vulnerable to the networks, the SigFree can block the attacks without having the signature of the attack, thus it is free from most of the code obfuscation methods. And also be most useful for good economical Internet wide development. Also the maintenance cost of the signature free buffer overflow blocker is very less and the negligible throughput.

Hence ,we can conclude that the defined goal of our Implementation of buffer overflow attack blocker have been achieved by introducing cost effective and with negligible throughput for networking solutions that allow to control networks.

REFERENCES

[1] A. Pasupulati, J. Coit, K. Levitt, S.F.Wu, S.H. Li, R.C. Kuo, and K.P. Fan, "Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities," Network Operations and Management Symposium 2004(NOMS 2004).

[2] A. Smirnov and T. cker Chiueh, "Dira: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.*

[3] C. Cowan. Software security for open-source systems. IEEE Security & Privacy, 1(1):38–45, 2003.

[4] C. Kruegel, T. Toth, and E. Kirda, "Service Specific Anomaly Detection for Network Intrusion Detection," InSymposium on Applied Computing (SAC), Spain, March 2002.

[5] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software, vol. 19, no. 1, 2002.*

[6] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Seventh Ann. Network and Distributed System Security Symp. (NDSS '00), Feb. 2000.*

[7] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Network and Distributed System Security Symposium, pages 3–17, San Diego, CA, Feb. 2000.

[8] Fu-Hau Hsu and Tzi-cker Chiueh, "CTCP: A Transparent Centralized TCP/IP Architecture for Network Security," Annual Computer Security Application Conference (ACSAC 2004), Tucson, Arizona, Dec., 2004.

[9] GCC Extension for Protecting Applications from Stack-Smashing Attacks, http://www.research.ibm.com/trl/projects/security/ ssp, 2007.

[10] G. Holzmann. Static source code checking for user-defined properties. In Proc. IDPT 2002, Pasadena, CA, USA, June 2002.

[11] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," *Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS), 2004.*

[12] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 232–244.

ACM Press, 2003.

[13] Jempiscodes—A Polymorphic Shellcode Generator, http://www.shellcode.com.ar/ en/proyectos.html, 2007

[14] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.*

[15] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In Proceedings of the 10th Network and Distributed System Security Symposium, pages 149–162, Feb. 2003.

[16] Ke Wang and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," Recent Advance in Intrusion Detection (RAID), Sept. 2005.

[17] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. SIGSOFT Softw. Eng. Notes, 29(6):97–106, 2004.

[18] M. Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Aug. 2003.

[19] Manish Prasad, Tzi-cker Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflow Attacks," Usenix Annual Technical Conference, General Track, San Antonio, TX, June 2003

[20] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on Software Engineering, 30(12):859–872, Dec. 2004.

[21] S. Macaulay, Admmutate: Polymorphic Shellcode Engine, http:// www.ktwo.ca/security.html, 2007.