Host-based Bottleneck Verification Efficiently Detects Novel Computer Attacks[1]

Robert K. Cunningham, Richard P. Lippmann, David Kassay, Seth E. Webster, Marc A. Zissman
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

***Abstract*-Bottleneck verification detects novel computer attacks by looking for users performing operations at a high privilege level without passing through legal "bottleneck" programs that grant those privileges. This approach has been used with network sniffing data to analyze telnet and rlogin network sessions to UNIX hosts and with Solaris Basic Security Module (BSM) host-based audit data. An off-line version of Bottleneck Verification performs at a false alarm rate more than two orders of magnitude lower than a reference key-string system, while simultaneously increasing the detection rate from roughly 20% to 80% for user-to-super-user attacks. Recent development of a real-time host-based version demonstrates that Bottleneck Verification can rapidly and accurately detect attacks, while adding very little load to the protected system. Furthermore, a simple extension allows a system to detect the use of backdoors installed prior to system installation.**

I. Introduction

Heavy reliance on the Internet has greatly increased the potential damage that can be inflicted by computer-based attacks, while worldwide connectivity has made it easier to launch those attacks from a distant and safe location. It is difficult to prevent such attacks by security policies, firewalls, or other mechanisms because software often contains unknown weaknesses or bugs, and because complex interactions between application software, operating system services and network protocols are exploited by attackers. Intrusion detection systems are designed to detect attacks, which inevitably occur despite security precautions. Some intrusion detection systems detect attacks in real time and can be used to stop an attack in progress. Others provide after-the-fact information about attacks and are used to repair damage, understand the attack mechanism, and reduce the possibility of future attacks of the same type. More complex intrusion detection systems detect never-before-seen, new attacks, while most commercial and government systems detect previously known and studied attacks.

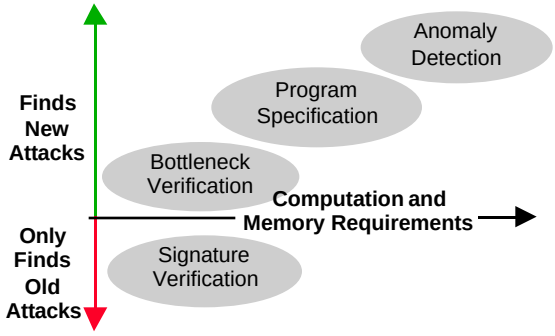

Fig. 1. Approaches to Intrusion Detection.

The many different approaches which have been pursued to develop intrusion detection systems are described elsewhere, including [4][16]. Figure 1 shows four major approaches to intrusion detection and the different characteristics of these approaches [15]. The figure is divided vertically into approaches that detect new, unseen attacks, and those that only detect previously known attacks. Simpler approaches are on the left and approaches that are computationally more complex, and have greater memory requirements are shown towards the right.

The most common approach to intrusion detection, denoted as "signature verification," is shown on the bottom of Figure 1. Systems using this approach find only previously seen, (known) attacks by looking for an invariant signature left by these attacks. This signature may be found either in host-based audit records on a victim machine or in the stream of network packets sent to and from a victim and captured by a "sniffer" which stores all important packets for on-line or future examination. The Network Security Monitor (NSM) was an early signature-based intrusion detection system that found attacks by searching for key-strings in network traffic captured using a sniffer. Early versions of the NSM [9][11] were the foundation for many government and commercial intrusion detection systems including NID [14] and NetRanger [5]. This type of system is popular because one sniffer can monitor traffic to many workstations and the computation required to reconstruct network sessions and search for keystrings is not excessive. In practice, these systems can have high false alarm rates (e.g. 100's of false alarms per day) because it is often difficult to select key-

[1] This work was sponsored by the Department of the Army under Air Force contract F19628-95-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Air Force.

strings by hand that successfully detect real attacks while not creating false alarms for normal traffic. In addition, these signature-based systems must be updated frequently to detect new attacks as they are discovered, and these and other systems which rely on network sniffing can be defeated by user or network encryption which makes reconstruction of network sessions effectively impossible. Most current commercial systems, including NetRanger [5], include some form of signature verification. Recent research on systems which rely on signature verification include BRO [18] which uses network sniffer data and NSTAT [12] which uses audit information from one or more hosts.

Approaches shown in the upper half of Figure 1 can find novel new attacks. This capability is essential to protect critical hosts because new attacks and new attack variants are constantly being developed. Anomaly detection, shown in the upper right of Figure 1, is one of the most frequently suggested approaches to detect novel new attacks. NIDES was one of the first statistical-based anomaly detection systems used to detect unusual user [10] and unusual program [2] behavior. The statistical component of NIDES forms a model of a user, system, or network activity during a training phase. After training, anomalies or departures from normal behavior are detected and are flagged as attacks. Anomaly detection is most useful if normal user or system behavior is repetitive and easily modeled and less useful when behaviors vary widely. When behavior is regular, this technique can discover attacks that rely on human interactions not observed on the computer system or network. Social engineering attacks, such as those in which an attacker tricks the victim into revealing passwords, can only be found using this method. Recent research on anomaly detection includes the development of EMERALD [19] which combines statistical anomaly detection from NIDES with signature verification. Other research, motivated by the natural immune system, detects anomalous behavior of system programs by examining system calls and looking for unusual call sequences that didn't occur during normal training [7][8]. Finally, some researchers are beginning to use neural networks for anomaly detection [22]. This approach combines some of the good qualities of signature verification with anomaly detection because neural networks can be trained to simultaneously model both normal behavior and known attack behavior.

Specification-based intrusion detection [13] is a second approach from the top half of Figure 1 that can be used to detect new attacks. It detects attacks which make improper use of system or application programs. This approach involves first writing security specifications that describe the normal intended behavior of programs. Host-based audit records are then monitored to detect behavior that violates the security specifications. This approach was applied to 15 Unix system programs and successfully found many attacks [13]. Specification-based intrusion detection has the potential for providing very low false alarm rates and detecting a wide range of attacks including many forms of malicious code such as trojan horses and viruses, attacks that take advantage of race conditions, and attacks that take advantage of improper synchronization in distributed programs. Unfortunately, it has not become popular because security specifications must be written for all monitored programs. This is difficult because system and application programs are constantly updated, because all programs must be monitored for effective protection, and because many recent browser, mail, and word processing applications are extremely complex and are difficult to model. Specification-based intrusion detection is thus best applied to a small number of critical user or system programs that might be considered prime targets for exploitation.

The final approach to intrusion detection shown in Fig. 1. is bottleneck verification. Bottleneck verification is designed to detect major system-wide security policy violations, without monitoring every system or application program. It doesn't require specifications for all monitored programs as with specification-based approaches, it doesn't require signatures for attacks as with signature verification, and it doesn't require a model of normal user behavior as with anomaly detection. It is also has very low computational and memory requirements. Bottleneck verification detects a user transitioning to a privileged state without going through the normal system bottlenecks used to permit this type of transition.

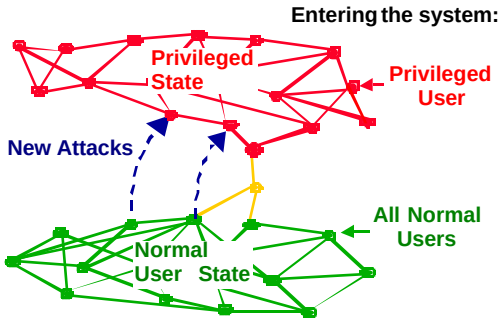

Fig. 2. System user state diagram (notional). Bottleneck verification detects a user transitioning to a privileged state without going through the normal system bottlenecks.

II. Bottleneck Verification

The bottleneck verification approach was motivated by careful examination of reconstructed telnet sessions, captured by network sniffers, of many actual attacks on government sites. It was found that the goal of many actual attacks on Unix systems and also of many exploits posted to web sites is to obtain an interactive shell running at the highest level of privilege (root). It was also noticed that, no matter which exploit was used to obtain such a root shell, evidence was present in the transcript that could be used to determine that a root shell had been created and that it had been created without following normal system procedures.

Figure 2 illustrates the bottleneck verification approach to intrusion detection. This approach applies to well-designed

operating systems where there are only a few legal "bottleneck" methods to transition from a lower privilege level (the lower, normal user states) to a higher privilege level (the upper, privileged user states) and where it is relatively easy to determine when a user is at a higher level. The key concept is to detect the use of legal bottleneck methods to transition to higher privilege levels and user activity indicative of a high privilege level. High privilege activity that originates from a session that did not pass through a bottleneck is indicative of an attack on the system. This can theoretically detect any novel attack that illegally transitions a user to a high privilege level, without prior knowledge of the attack mechanism. New attacks shown on the left of Figure 2 can thus be detected even when the attack mechanism is not understood.

Initial work on bottleneck verification has focused on detecting when users illegally obtain interactive root shells on UNIX hosts by examining sniffing data or host audit data, as described in the next few sections. The general approach, however, can be extended to other operating systems and to determine whether users illegally access or modify data or illegally use specific application or system programs.

A recent addition to bottleneck verification, indicated on the right side of Figure 2, makes use of the observation that most security policies restrict the number of privileged users permitted to access a system. (For UNIX, often only "root" is allowed to be the super-user with high privilege.)

When users other than the select few permitted by security policy appear to have super-user privilege, it is likely that a successful attack occurred that either predated installation of the Bottleneck Verification system, or that used a mechanism that was not yet covered by the implementation.

This approach was used to develop a successful (low false alarm rate, high detection rate) off-line implementation of a sniffer-based bottleneck verification intrusion detection system [15]. The following sections describe the design and implementation of real-time host-based Bottleneck Verification, and the performance of the system on two different computing platforms.

III. Real-time Host-Based System Design

Bottleneck Verification is applicable to any well-designed operating system that has at least two different levels of privilege. All UNIX-style operating systems (e.g., AIX, FreeBSD, Linux, MacOS Server, SCO, and Solaris) and the Windows/NT operating systems enforce this distinction. Unfortunately, system actions are recorded in different formats in different files for each of the different operating systems. To permit research and development of Bottleneck Verification independent of a given operating system's recording mechanism, a multi-step approach is taken. (See Fig. 3.) In the first stage, an operating system's (OS) native process reporting format is converted to an OS-independent data structure that represents information about each running process. This data structure represents who is running the process and whose permissions are used during the execution of the process (in UNIX parlance, both real and effective user ids), which process started the current process, and the name of the command. Some additional information is maintained to support attack reporting, including the start time of the process, and if the session originates on a different computer, the source IP address and the port from which the connection started.

As before, operating system events are mapped to generic process events, to permit development of Bottleneck Verification without regard for a specific system's event. For Unix systems, the POSIX standards have helped unify many aspects of the programming and user interfaces. Unfortunately, the application programmer's interface (API) for C2-class monitoring has not been standardized, so each operating system has a slightly different tool to accomplish this task. Luckily, to implement Bottleneck Verification, the only events that need to be monitored are those that create, initialize, or destroy a process, or those events that modify the processes' operating permission level. Create and destroy events are used to allocate and free process objects, initialization events indicate the permissions that the process will run at, and events that modify the operating permission level are used to verify that the bottleneck has been respected. Processes that spring into life at a higher permission level than their parents are flagged as attacks, if they haven't passed through the bottleneck.

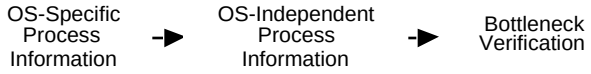

Fig. 3. Separating Bottleneck Verification from OS-specific process information.

IV. Real-time Host-Based System Implementation

To verify the ideas described above, a real-time implementation of Bottleneck Verification was developed using the Sun Solaris Basic Security Module (BSM) as input to the system. Solaris BSM events were mapped to the generic events as follows: fork, fork1, vfork create; exec and execve initialize process; kill, exit destroy process. In addition, the inetd connection creation event is monitored to provide information about source IP addresses and ports for reporting attacks, although this is not strictly necessary for implementing BV.

BSM intercepts calls between applications and the kernel, usually reporting all events to an audit file. The events that are recorded can be controlled via an interface that communicates with the audit daemon. By default, many events are turned on. This, along with the attempt to write these events to disk, results in a system that noticeably slows when BSM auditing is started in its default configuration. To achieve good performance, we do two things: first, only those events that are monitored by Bottleneck Verification are

enabled, and second, the audit daemon is told to write to a pipe that is read by the Bottleneck Verification process. Under most conditions, this data is never written to disk.

V. Host-Based System Performance

A. Evaluation Using Data from the DARPA 1998 Off-Line Intrusion Detection Evaluation

1. Data Source Description

In 1998 MIT Lincoln Laboratory created the first large-scale realistic data base that could be openly distributed and used to evaluate intrusion detection systems [6][17]. The full corpus was designed to evaluate both the false alarm rate and the detection rate of intrusion detection systems using many types of both known and new attacks embedded in a large amount of normal background traffic. Actual network traffic was generated to be similar to the type of traffic observed flowing between U.S. Air Force Bases and the publicly accessible Internet.

Traffic and attacks were generated using conventional Unix system and application programs by humans and automatic traffic generators on a network which simulated 1000's of Unix hosts and 100's of users using fewer than 20 actual machines. This network simulates the inside of an Air Force base connected through a Cisco router to outside machines on the Internet. The inside contains Linux, SunOS, and Solaris UNIX victim hosts and a gateway to 100's of simulated PC and Unix Workstation hosts. Host-based audit data was collected on the Solaris victim machine using Sun's Basic Security Module (BSM). The outside contains a gateway to 100's of simulated PC and Unix Workstation hosts, a gateway to 1000's of web servers and a sniffer workstation that captures all packets on the local area network connected to the outside of the router. Most attacks were launched from outside workstations through the router against one or more of the inside victim workstations.

Seven weeks of training data were provided with labeled attacks to train intrusion detection systems and two weeks of test data were then provided with unlabeled attacks to perform a blind evaluation. Automatically generated background traffic used more than 20 network services including dns, finger, ftp, http, ident, ping, pop, smtp, snmp, telnet, time, X. More than 38 different types of attacks were included in the test data with a total of 114 attack instances. Attacks were divided into four categories with regard to their purpose. Denial-of-service attacks made it difficult to use a system or some network service over a short interval, remote-to-user attacks provided a user at a outside workstation access to one of the victim machines, user-to-root attacks allowed a user who had already obtained user-level access to a victim machine to transition to a root privilege level, and surveillance or probe attacks provided outside users information about the configuration or existence of inside hosts. Attacks in the test data had either appeared in the training data, were new to the test data, or were new and developed by MIT Lincoln Laboratory.

2. Accuracy Results

An off-line version of host-based bottleneck verification was evaluated using the Solaris BSM audit information provided with Lincoln Laboratory corpus. The performance evaluation of host-based systems focused on user-to-root attacks because this is the attack type for which sufficient data was present. All sessions that were recorded by the auditing Solaris system were processed, however this is only a small subset (4600) of the total number of sessions. Of these sessions, only 22 were attacks, and all of these attacks obtained a root shell as part of the attack. The results presented in this section should be considered "unofficial," as some members of the intrusion detection team interacted with members of the evaluation corpus development team. Nevertheless, this data was processed only once following the formal evaluation procedures described in [17].

The off-line host-based bottleneck verification algorithm detected more than 80% of the attacks, with no false alarms. Further investigation of the results revealed that the algorithm could have found all attacks with no false alarms, were it not for an implementation bug. In the tested implementation we assumed that if any shell obtained root legally, then all subsequent processes spawned from this shell or its descendants should be allowed to obtain root. This assumption was too permissive: in the test data the inet daemon was restarted by the system administrator (who had legally obtained root). The inet daemon starts processes, connecting them to incoming sessions, so subsequent attacks that came over the network were missed. The fix actually simplified the implementation: instead of maintaining a list of all descendants, only immediate parent-children relationships need to be examined.

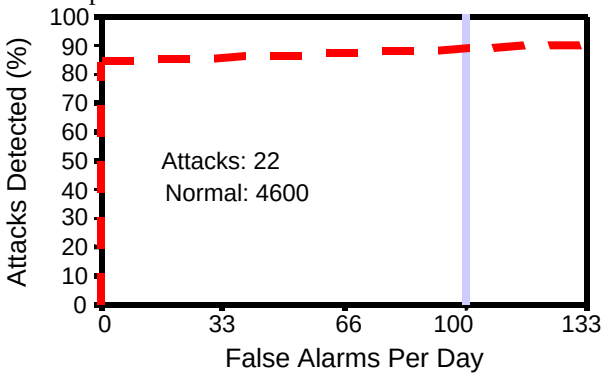

Fig. 4. Host-based Bottleneck Verification Accuracy. (Unofficial.)

B. Effect of installing Real-time Bottleneck Verification

Host-based intrusion detection systems have two inherent drawbacks: they alter the configuration of the system software, and they use up processing power and disk space of the machine on which they are installed.

Commercial companies that develop host-based intrusion detection systems address these problems. To reduce the number of changes required to install an intrusion detection

system, some companies provide an option to read audit data generated by operating systems in their default configuration (e.g., [1]). Commercial host-based intrusion detection systems typically require 10s to 100s of Mbytes of disk space to store the output of the audit and syslog daemons. For reasons of portability, some first allow the audit daemon to save records, and periodically and asynchronously process these records with a polling intrusion detection process [1].

For our implementation of bottleneck verification, we assume that if one is willing to modify the configuration of the system by installing an intrusion detection system, then it is advantageous to install a C2-class auditing system and receive the most accurate and timely process information possible. To reduce the amount of disk storage required, we cause Sun's BSM to write directly to the Bottleneck Verification process, only recording the events required to detect bottleneck violations.

Because we want to continue to improve the algorithm, it is important to us to have a system that can easily and rapidly be modified. As a result, our implementation of Bottleneck Verification is in perl, with C extensions to control the audit process. Perl is an outstanding language for prototyping systems; it has a simple object model that allows one to rapidly experiment with new ideas by altering the behavior of a limited number of program objects. It is not the best language for system software where memory use and processor utilization is to be minimized.

Nevertheless, we have installed our implementation of Bottleneck Verification on several different computer systems, including a portable Intel x86 system and a desktop Sun Sparc system. The portable Intel-based system was a Tecra 740 CDT portable system with a 166 MHz Pentium processor with 128 MB of memory running Sun Solaris 5.5.1. The desktop Sparc system was an Ultra 5 275/250MHz system with 128 MB of memory running Sun Solaris 5.6. The systems have run for several weeks, usually not consuming more than a few percent of processing, while consuming a relatively stable 4 Mbytes of memory. Because Bottleneck Verification is event driven, it only consumes CPU time when the requested audit events are produced.

We ran the integer SPEC marks to measure the overhead of the Bottleneck Verification daemon in a nearly quiescent system, and found it to be essentially zero. (There were no significant differences between SPECint marks with and without BV present.) During the few weeks that BV was installed on the abovementioned systems, we noticed the highest loads (~8% of the CPU utilization) during the execution of a parallel distributed make, which spawned many processes on the local and remote machines. During the past two weeks, no attacks were found, and no alerts were issued.

VI. Conclusions

Host-based Bottleneck Verification rapidly and efficiently detects novel attacks at exceedingly low false alarm rates. Host-based BV has been tested using the 1998 DARPA Intrusion Detection data for which it detected more than 80% of the attacks against an audited Solaris host that advanced the privilege of the attacker. A real-time host-based version has been developed and tested on Solaris systems running on two different processors. To date, the real-time version of Bottleneck Verification has had no significant effect on normal operation of Solaris hosts. Future work will focus on extending Bottleneck Verification to find other abuses of privilege and to find more stealthy attacks.

References

[1] "Axent Intruder Alert User Manual," Version 3.0, October 1998.

[2] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. "Safeguard final report: detecting unusual program behavior using the NIDES statistical component," Computer Science Laboratory, SRI International, Menlo Park, CA, Technical Report, December 1993.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0," RFC 1945, 1996.

[4] M. Bishop,, S. Cheung, C. Wee. "The Threat from the Net", IEEE Spectrum, 1997, 38(8).

[5] Cisco Systems, Inc. "NetRanger Intrusion Detection System Technical Overview,"http://www.cisco.com/warp/public/778/security/netranger/ntran_tc.htm, 1998.

[6] R. Cunningham, R. Lippmann, D. Fried, S. Garfinkel, I. Graf, K. Kendall, S. Webster, D. Wyschogrod, and M. Zissman, "Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation," in Proceedings of ID'99, Third Conference and Workshop on Intrusion Detection and Response, San Diego, CA: SANS Institute, 1999.

[7] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for Unix processes," in Proceedings of 1996 IEEE Symposium on Computer Security and Privacy, 1996.

[8] S. Forrest, S. Hofmeyr, and A. Somayaji, "Computer Immunology," Communications of the ACM, 40(10), 88-96, 1997.

[9] T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A Network Security Monitor", in IEEE Symposium on Research in Security and Privacy., 1990, pp. 296-304.

[10] H. Javitz, and A. Valdes, "The NIDES statistical component description and justification," Computer Science Laboratory, SRI International, Menlo Park, CA, Technical Report, March 1994.

[11] T. Heberlein, "Network Security Monitor (NSM) - Final Report", U.C. Davis: Feb. 1995, http://seclab.cs.ucdavis.edu/papers/NSM-final.pdf.

[12] R. Kemmerer. "NSTAT: A Model-based real-time network intrusion detection system," Computer Science Department, University of California, Santa Barbara, Report TRCS97-18, http://www.cs.ucsb.edu/TRs/TRCS97-18.html.

[13] C. Ko, M. Ruschitzka, and K. Levitt, "Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach," in Proc. IEEE Symposium on Security and Privacy, 1997, pp. 134-144, Oakland, CA: IEEE Computer Society Press.

[14] Lawrence Livermore National Laboratory (1998). "Network Intrusion Detector (NID) Overview," Computer Security Technology Center, http://ciac.llnl.gov/cstc/nid/intro.html.

[15] R. Lippmann, R. Cunningham, S. Webster, I. Graf, D. Fried, "Using Bottleneck Verification to Find Novel New Computer Attacks with a Low False Alarm Rate", in review.

[16] T. Lunt,, "Automated Audit Trail Analysis and Intrusion Detection: A Survey", in Proceedings 11th National Computer Security Conference., 1998, pp. 65-73.

[17] MIT Lincoln Laboratory, "Intrusion detection evaluations" http://www.ll.mit.edu/IST/ideval/index.html.

[18] V. Paxon, "Bro: A System for Detecting Network Intruders in Real-Time," in Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998, http://www.aciri.org/vern/papers.html.

[19] P. Porras, and P. Neumann, "EMERALD: Event Monitoring Enabling Response to Anomalous Live Disturbances," in Proceedings 20th National Information Systems Security Conference, Oct 7, 1997.

[20] J. Postel, "Transmission Control Protocol," RFC 793, USC/Information Sciences Institute September 1981.

[21] T. Ptacek, and T. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Secure Networks, Inc. January, 1998.

[22] J. Ryan, L. Meng-Jang, and R. Miikkulainen, "Intrusion Detection with Neural Nets," in Advances in Neural Information Processing Systems 10, Edited by M. Jordan, M. Kearns, and S. Solla, MIT Press: Cambridge, MA, 1998, pp. 943-949.