Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach*

Xuxian Jiang, AAron Walters, Florian Buchholz†, Dongyan Xu
Yi-Min Wang‡, Eugene H. Spafford

CERIAS and Dept. of Computer Science
Purdue University, W. Lafayette, IN 47907
{jiangx, arwalter, dxu, spaf}@cs.purdue.edu

† Dept. of Computer Science
James Madison Univ., Harrisonburg, VA 22807
buchhofp@jmu.edu

‡ Microsoft Research
Redmond, WA 98052
ymwang@microsoft.com

Abstract

To investigate the exploitation and contamination by self-propagating Internet worms, a provenance-aware tracing mechanism is highly desirable. Provenance unawareness causes difficulties in fast, accurate identification of a worm's break-in point, and incurs significant log inspection overhead. This paper presents the design, implementation, and evaluation of process coloring*, an efficient provenance-aware approach to worm break-in and contamination tracing. More specifically, process coloring assigns a "color", a unique system-wide identifier, to each remotely-accessible server or process. The color will then be either inherited by spawned child processes or diffused indirectly through process actions (e.g.,* read/write *operations). Process coloring brings two major advantages: (1) It enables fast color-based identification of a worm's break-in point even before detailed log analysis; (2) It naturally partitions log data based on their colors, effectively reducing the volume of log data that need to be examined for worm investigation. A tamper-resistant log collection method is developed based on the virtual machine introspection technique. Our experiments with a number of real-world worms demonstrate the advantages of processing coloring.*

1 Introduction

In the combat against worms, the following tasks are critical to the understanding of a worm's exploitation details and to the recovery of an infected host from worm contaminations: (1) identifying the *break-in point*, namely the vulnerable, remotely accessible service via which the worm infects the victim and (2) determining all contaminations and damages inflicted by the worm during its residence in the victim. To perform these tasks, various intrusion analysis tools can be used. For example, BackTracker [16] is an advanced forensic tool that traces back an intrusion starting from a "detection point" and identifies files and processes that could have affected that detection point. The tool takes the entire log file of the host as input for back-tracking.

Log-based intrusion analysis tools face the following challenges: (1) Many tools [3, 16, 27] rely on an *externally-determined* detection point, from which a forensic investigation will be initiated towards the break-in point of the intrusion. However, due to a worm's possibly long "infection-to-detection" duration, it may be days or even weeks later when such a detection point is identified. It is therefore desirable that the log data carry more information and provide "leads" to trigger more timely investigations. (2) Current operating systems lack a *provenance-aware* mechanism to pre-classify the log data before log analysis. On the other hand, log data generated by the system may be of large volume. The uncategorized bulk log data are likely to result in long duration and high overhead in worm investigation. (3) Many log-based tools do not address *tamper-resistant* log collection, which is essential in dealing with advanced worms. As shown in Section 2.3, a commonly used mechanism for collecting system call traces, syscall wrapping, can be easily circumvented.

In this paper, we present the design, implementation, and evaluation of *process coloring*, an efficient provenance-aware approach to worm break-in and contamination investigation. More specifically, process coloring associates a "color", a unique system-wide identifier, to each remotely-accessible server or process - a potential worm break-in point. The color will be either *inherited* directly by any spawned child process, or *diffused* indirectly through the processes' actions (e.g., *read* or *write* operations). As a result, any process or object (e.g., a file

*This work was supported in part by gifts from Microsoft Research and grants from the National Science Foundation OCI-0438246, OCI-0504261, CNS-0546173.

or directory) affected by a colored process will be tainted with the same color, as recorded in the corresponding log entry. Process coloring naturally leads to the following two key advantages:

Color-based identification of a worm's break-in point All worm-infected processes and contaminated objects will be tainted with the same color as the original vulnerable service, which is exploited by the worm as the break-in point. By simply examining the color of any worm-related log entry or any worm-affected object, the break-in point of the corresponding worm can be immediately identified before detailed log analysis.

Natural partition of log data The colors of log entries provide a natural way to partition the log. To reveal the contaminations caused by a worm, it is no longer necessary to examine the entire log file. Instead, only log entries with the same color as the worm's entry point will need to be inspected. Such partition can substantially reduce the volume of relevant log data, and thereby improve the efficiency of worm investigation.

The practicality and effectiveness of process coloring are demonstrated using a number of real-world self-propagating worms and their variants. For each of these worms, we are able to fast identify the vulnerable networked service exploited by the worm. Moreover, reduction of inspected log data is achieved in each worm experiment. For example, for a detailed SARS worm [2] break-in and contamination investigation, only 12.1% of the entire log data need to be inspected. Our prototype also addresses the important requirement of tamper-resistant log data recording. We adopt a technique similar to Livewire [13] and develop an extension to the UML virtual machine monitor (VMM) for tamper-resistant logging.

The rest of the paper is organized as follows: Section 2 provides an overview of the process coloring scheme, whose implementation is presented in Section 3. Experimental evaluation results are presented in Section 4. Other applications and possible attacks are addressed in Section 5. Section 6 discusses related work. Finally, Section 7 concludes this paper.

2 Process Coloring Approach

2.1 Initial Coloring

Figure 1 shows a process coloring view of a networked host system running multiple servers. A unique system-wide identifier called *color* is assigned to each server process. The color assignment takes place after the server processes have started but before serving client requests. A worm breaking into the system will need to exploit a certain vulnerability of a (colored) server process. Because any action performed by the exploited process will lead to a corresponding *color diffusion* in the host (Section 2.2), the break-in and contaminations by the worm will be evidenced by the color of the affected processes and system resources and by the color of the corresponding log entries. Each remotely-accessible service is performed by one or more active processes in the host. For example, the Samba service will start with two different processes *smbd* and *nmbd*; and both *portmap* and *rpc.statd* processes belong to the NFS/RPC service. Such processes can be assigned the same color. However, if we need to further differentiate each individual process (e.g., "which Apache process is exploited by a Slapper worm?"), different colors can be assigned to processes belonging to the same service. One benefit of such assignment is that it provides a finer granularity in log data partition.

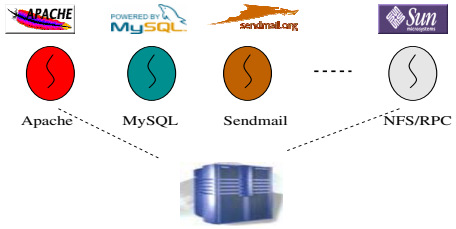

Figure 1. Process coloring view of a system running multiple servers

2.2 Color Diffusion Model

After the service processes are initially colored, the colors will be diffused to other processes according to the operations performed by the processes. To reveal worm contaminations, we are especially interested in process color diffusion via system-wide shared resources, such as files, directories, and sockets. For a worm to inflict contamination (e.g., backdoor installation), it needs to go through a number of system calls. Hence the process colors are diffused to the affected system resources via the operations performed by the system calls.

The color diffusion model is based on our more general *process label* framework [4], where audit information (defined as process label) is propagated and preserved in a system. We also note that process color diffusion reflects the information flow model [8]. In this paper, we only consider the information flow through syscall interfaces, with the processes as subjects and intermediate resources as objects. Other means such as using CPU utilization or disk space availability to convey information are beyond the scope of this paper. In the following, we describe two types of syscall-based color diffusion:

Direct diffusion involves one process directly affecting the color of another process. It can happen in a number of ways: (1) *Process spawning:* If a process issues the *fork*,

vfork, or *clone* system call, a new child process will usually be spawned and it will inherit the color of the parent process. (2) *Code injection:* A process may use code injection (e.g. via *ptrace* system call) to modify the memory space of another process to change its functionality. The color of the injected process will be updated accordingly. (3) *Signal processing:* A process may send a special signal (e.g., the *kill* command) to another process. If received and authorized, the signal will invoke corresponding signal handling and thus affect the execution flow of the signaled process.

Indirect diffusion from process s_1 to s_2 can be represented as $s_1 \Rightarrow o \Rightarrow s_2$, where o is an intermediate resource (object). Various types of intermediate resources exist: some resources are dynamically created and will not exist after the process is terminated (e.g., UNIX sockets); other resources such as files can persistently exist and may later affect another process if that process acquires some input from these resources. To support indirect diffusion, the system data structure for an intermediate resource will be enhanced to record the influence of a process (i.e. its color). Later, when another process gets input from the "tainted" resource, the process will be tainted the same color. Common resource types supported in current Linux systems include files, directories, network sockets (including UNIX sockets), named pipes (FIFO), and IPC (messages, semaphores, and shared memory).

2.3 Log Information Collection

Process coloring employs system call (syscall) interception to generate log entries and tag them with process colors. As demonstrated in [16, 22], syscall interception is effective in revealing and understanding intrusion steps and actions. However, a simple syscall-based hooking mechanism may be vulnerable to the *re-hooking* attack, where attackers can easily avoid or even subvert [9] the log collection function. Instead, our design is based on the virtual machine introspection technique [13], where the interception of system calls happens *not* in the syscall dispatcher, but *on the virtual machine virtualization path*. As such, the interceptor is an integral part of the underlying virtual machine implementation (Section 3) achieving stronger tamper-resistance. Information about each intercepted system call (e.g. current process, syscall number, parameters, return value, and return address) forms a log entry, which is tagged with the color of the current process.

3 Implementation

Our prototype leverages User-Mode Linux (UML), an open-source VM implementation where the guest OS runs directly in the unmodified user space of the host OS, and only considers the *ext2* file system[1]. In order to support process coloring, a number of key data structures (e.g., $task\_struct$, $ext2\_inode\_info$) are modified to accommodate the color information. Implementation details of color setting and diffusion can be found in [14].

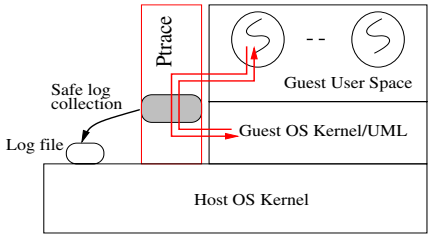

Figure 2. Tamper-resistant log collection by positioning the interceptor on the system call virtualization path

The log collection mechanism is based on the underlying virtual machine implementation, i.e. UML, as shown in Figure 2. UML adopts a system call-based virtualization approach and supports VMs in the user space of the host OS. Leveraging the capability of *ptrace*, a special thread is created to intercept the system calls made by any process in the VM, and to redirect them to the guest OS kernel. The interceptor for system call log collection is located on the system call virtualization path. Therefore, it is tamper-resistant from malicious processes running inside the VM. Moreover, once the interceptor has collected a certain amount of log data (e.g., $16K$), the log data will be pushed down to the host domain. One important benefit is that the analysis on the log file within the host domain will not interrupt the normal execution of the VM. This creates the possibility of external *runtime* system monitoring based on the colors of log data.

4 Evaluation

We evaluate the effectiveness of process coloring using a number of real-world Internet worms: Adore, Ramen, Lion, Slapper, SARS, and their variants. Each worm experiment is conducted in a virtual distributed worm playground called *vGround* [15], which is a realistic, confined, and scaled-down Internet environment consisting of network entities and end hosts realized as VMs with process coloring capability. vGround makes it easy and efficient to create VMs running real-world services as well as VMs running as service clients. vGround enables safe worm experiments by confining all traffic within the vGround. It also facilitates experiments with a multi-vector worm

[1] We are currently implementing process coloring on another VM platform Xen [10] and we expect even better performance than our UML-based prototype due to Xen's para-virtualization approach.

	Lion Worm	*Slapper Worm*	*SARS Worm*
Exploited Service (CVE references)	BIND (bind-8.2.2_P5-9) (CVE-2001-0010)	Apache (apache-1.3.19-5) (CAN-2002-0656)	Samba (samba-2.2.5-10) (CAN-2003-0201)
Time period being analyzed	24 hours	24 hours	24 hours
Number of log entries	129,386	293,759	166,646
Size of log data	8.0M	18.5MB	10.7MB
Number of worm-relevant log entries	66,504	195,884	19,494
Size of worm-relevant log data	3.9MB	12.2MB	1.3MB
Number of files "touched" by the worm	120,342	62	200
Percentage of worm-relevant logs	48.7%	65.9%	12.1%

Table 1. Statistics of process coloring log data in three worm experiments

(e.g., Ramen worm), which infects hosts (VMs) via different break-in points[2].

Due to space constraint, we only present experiments with Lion, Slapper, and SARS worms. Table 1 shows key statistics of their respective log data. Each log file contains log entries collected during a 24-hour period, including both worm-related and normal service access entries. During each experiment, process coloring demonstrates its key benefits: (1) We are able to identify the worms' break-in points before performing detailed log analysis. The break-in points are the BIND server (bind-8.2.2_P5-9) for Lion worm, the Apache server (apache-1.3.19-5 with openssl-0.9.6b-8 package) for Slapper worm, and the Samba server (samba-2.2.5-10) for SARS worm. (2) The log data that need to be inspected for detailed worm investigation is only 48.7% (Lion worm), 65.9% (Slapper worm), and 12.1% (SARS worm) of the total logged events, respectively. We note that, since log entries are naturally partitioned by their colors, increasing background service accesses (i.e. accesses to unrelated services) in the experiments will further *reduce* the percentage of worm-related log. (3) Since the worm break-in point (vulnerable service) is identified *before* log analysis, it is possible to further filter the log entries that record normal accesses to the vulnerable service, which have *known and different* footprint from that of a worm infection.

4.1 Lion Worm Contamination Investigation

Figure 3 shows a process coloring view of an *uninfected* system running a BIND server vulnerable to the Lion worm. There are also a number of other services hosted at the same system: NFS/RPC service (*portmap* and *rpc.statd*), printer service (*lpd*), and mail service (*sendmail*). A different color is assigned to each service.

[2]For example, Ramen worm has three possible break-in points: LPRng (CVE-2000-0917), rpc.statd (CVE-2000-0666), and wu-ftp (CVE-2000-0573) - the last one cannot lead to a successful exploitation as shown by our vGround experiment.

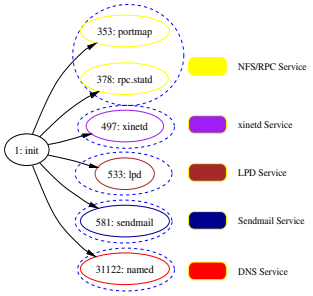

Figure 3. A process coloring view of a vulnerable system *BEFORE* Lion infection

Process *named* has the color "RED". The Lion worm is unleashed from a different VM in the vGround[3]. After the experiment, we obtain a log file whose entries are conveniently partitioned by their colors. Among the "RED" entries whose provenance is the *named* process, we observe an abnormal event that a *shell* process was spawned. This is *one* of the contaminations inflicted by the Lion worm. To further reduce the inspected log volume, entries generated by normal accesses to the BIND server from other legitimate VM clients in the vGround are filtered. We then use the remaining "RED" log entries to derive a Lion worm contamination graph as shown in Figure 4.

We confirm that Figure 4 reveals *all* Lion worm contaminations by comparing our results with a detailed Lion worm report [1]. The leftmost oval is the vulnerable *named* daemon (PID: 31122). After a successful exploitation of the *named* process, a worm replica is downloaded (Circle 2 in Figure 4). The worm then overwrites all HTML files named *index.html* in the system with a self-carried HTML file for web defacement (Circle 3). Interestingly, we observe from the log that the worm attempts to execute the file replacement *twice* - a detail *not* reported in [1]. The first attempt to replace files is within the shell

[3]This "seed" worm is instrumented to target the vulnerable VM for infection. However, the worm copy transferred is unmodified.

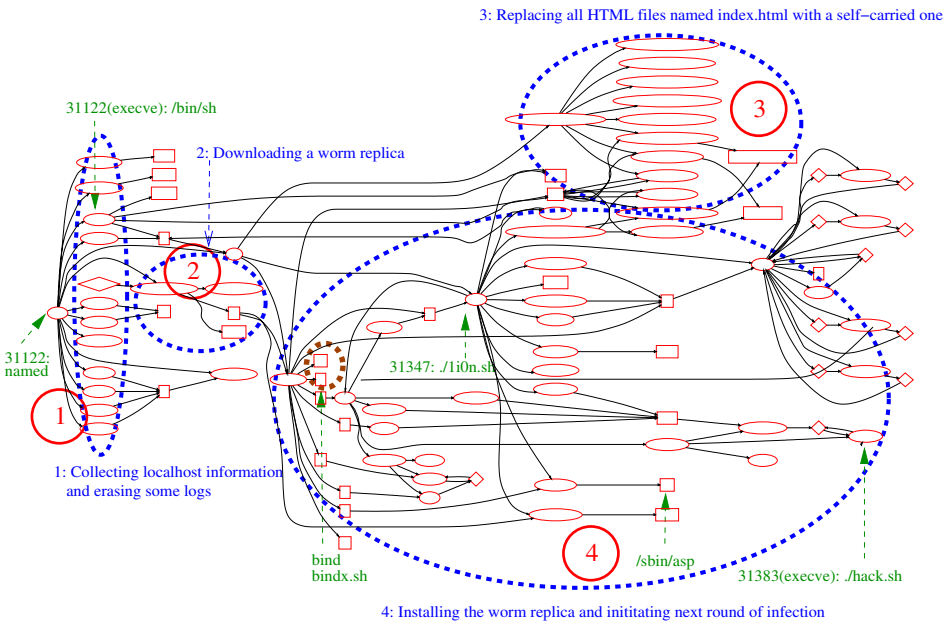

Figure 4. Lion worm contaminations reconstructed from "RED" log entries

code (PID: 31181) after executing the malicious buffer overrun code (Circle 2 and Circle 3). The second attempt happens when the driving script *./1i0n.sh* (PID: 31347) is executed (Circle 4). The worm then tries to initiate the next round of infection (Circle 4). In the thick dotted circle inside Circle 4, we find two "RED" *dangling* files *bind* and *bindx.sh*, which are introduced by the worm but never accessed by any worm-related process. Such anomaly deserves a further investigation. A forensic analysis of the VM reveals that these two files contain the exploitation code for the BIND vulnerability. As there is only one VM running the vulnerable BIND service in the vGround, the worm cannot find another host to infect and the file *bindname.log* storing IP addresses of possible victims is empty. As a result, the exploitation code is never launched.

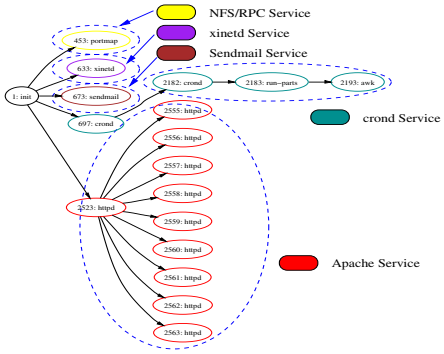

Figure 5. A process coloring view of a Slapper-vulnerable system *BEFORE* infection

4.2 Slapper Worm Contamination Investigation

The Slapper worm experiment is conducted in a different vGround. We initially assign colors to service processes in an uninfected VM. Especially, the vulnerable Apache service is assigned "RED". Through direct diffusion, all spawned httpd worker processes are also colored "RED". A process coloring view of the system *before* the Slapper infection is shown in Figure 5. The experiment involves accesses to the other services as well as normal web accesses requesting a 2890-byte *index.html* file.

After the experiment, an examination on the log file shows a flurry of "RED" log entries ($>$ 10000) within a very short period (1 minute) - an anomaly indicating a possible infection. As the "RED" color is associated with the Apache web server, we select all "RED" log entries, which constitute 65.9% of the entire log file. A quick review of these log entries shows that the Slapper worm infection has a large and distinct footprint in the infected host. During the transmission of a Slapper worm, a *uuencoded* source file is sent from the infecter to the victim. More specifically, the sender issues a *sendch* call for *each byte* of the uuencoded file. Correspondingly, the receiver uses a *sys_read* for each byte received (total 94320 calls). Moreover, each encoded byte is then written (the *cat* command) to a local file named */tmp/.uubugtraq*, leading to another 94320 *sys_write* system calls. In sharp contrast, each normal web access only generates 15 log entries, record-

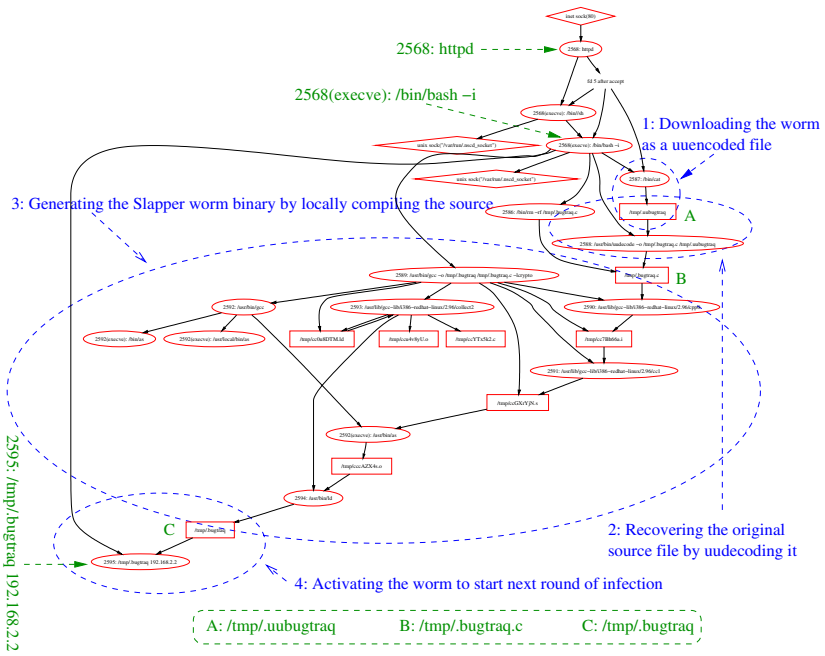

Figure 6. Slapper worm contaminations reconstructed from "RED" log entries

ing the known normal sequence of Apache server actions. Therefore, we remove these ("RED") entries before constructing the Slapper worm contamination graph (Figure 6)[4].

By comparing our results with a detailed Slapper worm analysis [20], we confirm that Figure 6 reveals *all* contaminations by the Slapper worm. We first observe that the worm exploits an httpd worker process (PID:2568) to gain system access. After that, a uuencoded version of the worm source code is downloaded (Circle 1 in Figure 6) and *uudecoded* (Circle 2) to reconstruct the original code, which is then compiled (Circle 3) to generate the worm binary. The binary is executed (Circle 4) to attempt to infect other hosts. The collected log data further reveal that the exploitation of the Slapper worm is rather sophisticated. Before the httpd worker process (PID: 2568) is exploited, 23 TCP connections have already been established with different http worker processes between the infecter and the victim. Interestingly, 21 of these connections have *no* payload; one connection is an invalid HTTP request, which turns out to be a request to obtain the Apache server version; and the last connection has a short interaction. From [20], we know that one of the 21 plain connections is used to validate the reachability of the Apache server, while the other 20 connections are made for depleting the Apache server pool to make sure that the two subsequent exploitations will have the same heap layout. The first exploitation aims at reliably deriving the over-writable heap address in the vulnerable Apache server. This heap address is then reused in the second exploitation. All these connections and interactions are recorded by "RED" log entries.

SARS worm contamination investigation Due to space constraint, the results of SARS worm experiment are presented in [14]. Only 12.1% of the log data need to be processed to fully reveal SARS worm contaminations. SARS worm performs rootkit and backdoor installation as well as system information collection, reflecting the recent trend in the underground evolution of increasingly *stealthy* self-propagating worms.

5 Possible Attacks and Counter-Measures

Jamming attack A worm could intentionally introduce many noise log entries to hide its actual intention. For example, a worm could invoke a large number of "innocuous" or unrelated syscalls just to hide its real infection attempts. However, tactically speaking, these actions still need to be considered as a part of the worm's behavior in the infected system, even though they may not contribute to any real damage. Also, to the worm's *disadvantage*, these noise log entries deviate from the normal log pattern of a specific color and will trigger an alarm. Finally, the capability of color-based identification of a worm's entry point is still valid under this attack, though it will take a

[4]We note that a general intrusion may mimic the normal sequence of service access actions [25]. However, it is more difficult for self-propagating worms to do so because their *outgoing propagation* behavior is semantically different from a normal service access.

more careful analysis to uncover the obfuscated intention.

Low-level attack The integrity of colors associated with active processes and intermediate resources are critical to worm investigation. As the current prototype maintains the color information within the kernel of the system under inspection, it is possible that this information be manipulated through certain low-level attacks. For example, if the process color is associated with the *task_struct* PCB structure, a method called direct kernel object manipulation (DKOM) [5] can be leveraged to explicitly change the color value (e.g., by writing to the special device file */dev/kmem*). Fortunately, solutions such as CoPilot[21], Livewire[13], and Pioneer [23] have been proposed to address the issue of kernel integrity. Another possible counter-measure is to create a *shadow* structure, which is instead maintained by the virtual machine monitor (VMM) and is totally inaccessible from inside the VM. Compared with the current prototype, the shadow solution poses significantly greater challenge in deriving VM operation semantics from low-level information collected via virtual machine introspection, which may affect the accuracy and completeness of worm investigation results.

Diffusion-cutting attack It is possible that a worm might use a hidden channel to undermine the diffusion. For example, a worm could use an initial part of an attack to crack a weak password, which is later used in a *separate* session to gain the system access and complete the rest of worm contamination. Process coloring can track any action performed within each break-in, but it cannot automatically associate the second break-in with the first one. However, any anomaly within the second break-in will immediately expose the responsible login session, which may lead to the identification of the cracked password. Based on the log data from the first break-in, the administrator may still be able to correlate those two disjunct break-ins.

Color saturation attack If a worm is aware of the coloring scheme, it might attempt to acquire more colors from different services right after its break-in. As a result, the associated colors can not uniquely identify the break-in point. However, to the worm's *disadvantage*, the color saturation attack will immediately lead to an alarm of *color mixing* - an anomaly triggering further investigation. Color saturation attack does expose a weakness of our current prototype, which uses a single color field. Although our prototype is able to accommodate multiple colors (each bit in the color field represents a different color), it is not able to differentiate between an *inherited* color and a *diffused* color. The inherited color of a process can only be inherited from its parent and will not be changed by its own or others' behavior. The diffused colors, on the other hand, reflect the color diffusions through its own or others' actions (e.g., *sys_read*/*sys_write*). With this distinction, the inherited colors can be used to partition log data, while the diffused colors can be used to detect color saturation attacks and identify all color-mixing points for further examination in affected partitions.

6 Related Work

Process coloring is inspired by the concept of transitive dependency tracking [12] originally proposed for failure recovery. Process coloring also reflects the information flow model [8]. With these concepts and models as theoretical underpinnings, a spectrum of taint-based techniques have recently been proposed: Process coloring operates at the system call level to reveal worm break-in and contamination; TaintCheck [19] works at the instruction level to detect overwrite attacks and generate exploit signatures; TaintBochs [6] focuses on lifetime tracking of sensitive data in a system. While sharing the same design philosophy, these techniques differ in their goals, design, implementation, and usage.

Process coloring can be integrated into existing log-based intrusion investigation tools [16, 18] so that they become provenance-aware. BackTracker [16] is able to automatically reconstruct the sequences of steps that occurred during an intrusion. More specifically, starting with an external detection point, BackTracker identifies files and processes that could have affected this detection point and displays chains of events in a dependency graph. The follow-up work [18] of Backtracker proposes a forward tracking capability that identifies all possible damages caused by the intrusion after the back-tracking session. Both BackTracker and its forward tracking extension require the entire log data as input. With process coloring enhancement, the break-in point of a worm can first be identified by the color of the detection point, and the volume of input log data will be reduced by color-based log partition, resulting in more efficient back-tracking and forward-tracking. In addition, the colors and patterns of log entries may provide alerts at runtime, triggering more timely investigations.

Process coloring can also be applied to enhance file and database repair/recovery. The Repairable File Service [27] identifies possible file system level corruptions caused by a root process, assuming that the administrator has already identified such a root process. It then uses the log to identify the files that may have been contaminated by that process. The repairable file service implements a limited version of the forward-tracking capability by only tracking file system level corruptions. Meanwhile, there has been technique in the database area [3] that is capable of recording contaminations at the transaction level and rolling back the damages if the transaction is later found malicious. This technique also requires external identification of malicious processes or transactions. Process coloring can enhance these techniques by tracking more

sophisticated contamination behavior via color diffusion, raising anomaly alarms based on log colors and patterns, and achieving tamper-resistant log collection.

Recent advances in virtual machine technologies have created new opportunities for intrusion detection and replay [11, 13], system diagnosis [17, 26], attack recovery and avoidance [11, 24], and data life-time tracking [6, 7]. For example, ReVirt [11] is able to replay a system's execution at the instruction level. Time-traveling virtual machines [17, 26] provide a highly effective means of re-examining and troubleshooting system problems. Process coloring complements these efforts by leveraging virtual machine technologies for worm break-in and contamination investigation. In addition, process coloring, as an advanced logging mechanism, can be integrated into other VM-based networked systems to add provenance-awareness to these systems.

7 Conclusion

We have presented the design, implementation, and evaluation of process coloring, a novel systematic approach to provenance-aware tracing of worm break-in and contaminations. By associating a unique color to each remotely-accessible service and diffusing the color based on actions performed by processes in the system, process coloring achieves two key benefits: (1) color-based identification of a worm's break-in point before detailed log analysis and (2) color-based partitioning of log data. Process coloring improves log-based worm investigation tools by reducing the amount of log entries to be processed and by providing color-related "leads" for more timely investigation. Experiments with a number of real-world Internet worms demonstrate the practicality and effectiveness of process coloring.

References

[1] SANS Institute: Lion worm. *http://www.sans.com/y2k/lion.htm*.

[2] SARS Worms. *http://www.xfocus.net/tools/200306/413.html*, June 2003.

[3] P. Ammann, S. Jajodia, and P. Liu. Recovery from Malicious Transactions. *IEEE TKDE, Volume 14, Issue 5*, 2002.

[4] F. Buchholz. Pervasive Binding of Labels to System Processes. *Ph.D. Thesis, CERIAS TR 2005-54, Purdue University*, 2005.

[5] J. Butler. Direct Kernel Object Manipulation (DKOM). *http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf*, 2004.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. *USENIX Security Symp.*, Aug. 2004.

[7] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. *USENIX Security Symp.*, Aug. 2005.

[8] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM 19, 5 (May), 236-243*, 1976.

[9] M. Dornseif, T. Holz, and C. Klein. NoSEBrEaK - Attacking Honeynets. *Proccedings of the 5th Annual IEEE Information Assurance Workshop, Westpoint*, June 2004.

[10] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. *ACM SOSP 2003*, Oct. 2003.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *USENIX OSDI*, Dec. 2002.

[12] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Computing Survey*, 34(3), 2002.

[13] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection . *NDSS 2003*, Feb. 2003.

[14] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y. Wang, and E. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. *CERIAS Technical Report 2005-81, Purdue University*, 2005.

[15] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *RAID 2005*, Sept. 2005.

[16] S. T. King and P. M. Chen. Backtracking Intrusions. *ACM SOSP 2003*, Oct. 2003.

[17] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *USENIX Technical Conference*, Apr. 2005.

[18] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts Through Multi-Host Causality. *NDSS 2005*, Feb. 2005.

[19] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS 2005*, Feb. 2005.

[20] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper*.

[21] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. *USENIX Security Symp.*, Aug. 2004.

[22] N. Provos. Improving Host Security with System Call Policies. *USENIX Security Symp.*, Aug. 2003.

[23] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. *ACM SOSP 2005*, Oct. 2005.

[24] A. Stavrou, A. D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein. MOVE: An End-to-End Solution To Network Denial of Service. *NDSS 2005*, Feb. 2005.

[25] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. *ACM CCS 2002*, Nov. 2002.

[26] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. *USENIX OSDI 2004*, Dec. 2004.

[27] N. Zhu and T. Chiueh. Design, Implementation and Evaluation of Repairable File Service. *IEEE DSN 2003*, June 2003.