FloGuard: Cost-aware Systemwide Intrusion Defense via Online Forensics and On-demand IDS Deployment

Saman A. Zonouz†, Kaustubh R. Joshi‡, and William H. Sanders†
University of Illinois† and AT&T Labs Research‡

Abstract. Detecting intrusions early enough can be a challenging and expensive endeavor. While intrusion detection techniques exist for many types of vulnerabilities, deploying them all to catch the small number of vulnerability exploitations that might actually exist for a given system is not cost-effective. In this paper, we present FloGuard, an on-line intrusion forensics and on-demand detector selection framework that provides systems with the ability to deploy the right detectors dynamically in a cost-effective manner when the system is threatened by an exploit. FloGuard relies on often easy-to-detect symptoms of attacks, e.g., participation in a botnet, and works backwards by iteratively deploying off-the-shelf detectors closer to the initial attack vector. The experiments using the EggDrop bot and systems with real vulnerabilities show that FloGuard can efficiently localize the attack origins even for unknown vulnerabilities, and can judiciously choose appropriate detectors to prevent them from being exploited in the future.

1 Introduction

Automatic response to security attacks that exploit previously unknown vulnerabilities can help the majority of computer systems that are not supported by dedicated security teams. If an attack's initial infection point can be isolated to an individual process or file, response techniques such as online attack signature generation and filtering, e.g., [10, 31] can be effective. However, the usefulness of such approaches for unknown "zero-day" attacks is often hampered by lack of early and accurate detection of unknown vulnerability exploitations. There are approaches in the literature for detecting many different types of vulnerability exploitations such as buffer overflows [7], SQL injections and other format string attacks [30], and brute-force attacks [5]. Nevertheless, many computer systems today run with very little protection against unknown attacks, and often the first sign of compromise happens too late, either by users noticing degraded system performance, or ISPs detecting that the system is part of a BotNet or DDoS attack [23]. If such a range of detection options are available in the literature, why are they not used?

We hypothesize that there are two main challenges to early but effective attack detection: cost and precision. Many of the detection mechanisms cited earlier are simply too expensive to be continuously deployed. For example, bounds checking techniques such as CRED have overheads of as much as 300% [28], while taint-tracking can add as much as 20X (Section 7). As one broadens the range of vulnerability types to be detected and the number of system components to be protected, the overheads add up, and push the cost threshold beyond which a technique becomes infeasible even lower. The move to mobile devices with limited compute power and battery life further exacerbates this problem. Detection mechanisms such as anomaly detectors or syscall sequence detectors that have low precision (i.e., high false positive rates) are rarely used even if their computational costs are low. On the other hand, cheap detectors do exist, e.g., change detectors for critical files [32], or anomaly detectors [5], but they often only detect the consequences of an attack, not the actual vulnerability that was exploited.

In this paper, we introduce FloGuard, which extends our previous work [34], an online forensics and backtracking framework that takes a system-wide cost-sensitive

approach to attack detection and tracing. It uses the observation that although it may be difficult to notice the immediate effects of an exploit inexpensively, the ultimate consequences of attacks are often easier to detect. For example, inexpensive in-network scanning techniques such as BotGrep [23] can detect participation in a botnet or DDoS attack. Malware scanners such as ClamAV [17] can detect previously known payloads even if the attack vector is unknown, and anomaly detectors can detect deviations in a system's performance or modifications to its critical files. Once an attack is detected in this manner, FloGuard can roll the system back to a clean checkpoint[1], determine possible paths the attack could have taken to reach its detection point using an online forensics algorithm, and deploy additional security detectors to catch and detect the attack at an earlier stage the next time that it is attempted. By iteratively repeating this process, it can deploy detectors successively closer to the initial attack vector until such a time that the attack can be stopped by automatic prevention techniques, such as input signature generation or quarantining.

To determine the set of detectors to enable and disable in every iteration, FloGuard uses an *Attack-Graph-Template* (AGT), which is an extended attack graph that enumerates *possible* attack and privilege escalation paths in the system. AGTs include possible paths, not ones known to be in the particular implementation being protected. E.g., an AGT for a server written in C can include privilege escalation using a buffer-overflow exploit, even though there may be no such known vulnerability. Therefore, AGTs can be constructed automatically by using system call monitoring to track *all* information and control flow paths between a system's privilege levels via processes, sockets, and files. During the forensics phase, FloGuard solves an optimization problem based on the the deployed detectors' outputs, the cost and coverages of the unused detectors, and the paths encoded by the AGT to determine which detectors to deploy for the next round.

We believe that FloGuard is one of the first frameworks to effectively balance the cost of security detection mechanisms against their coverage. By invoking mechanisms "on-demand" only when they are needed to forensically evaluate an attack that is already known to exist for the target system, FloGuard can utilize expensive mechanisms such as buffer bounds checking and taint tracking that are known to have good coverage characteristics. Furthermore, since FloGuard is a whole-system tool, it can initiate its detection and forensics analysis based on a wide range of attack consequences that may be many processes and files away from the initial attack vector. Finally, the models we use are designed so that new security mechanisms can be easily integrated into its decision-making, making FloGuard a flexible and extensible detection tool that can be practically used for a wide variety of attack types.

The basic premise behind FloGuard is not that one cannot statically determine the necessary IDSes to install for "known" vulnerability (e.g., using CERT); instead, we content that statically deploying all the necessary IDSes or countermeasures to protect against the entire universe of "unknown" vulnerabilities that may be present in a system is not feasible from a performance and/or usability standpoint. We currently focus on scripted zero-day worms and malware exploiting both known and unknown vulnerabilities. These usually generate easy to detect consequences such as network scanning, and participation in botnets. Our focus is not on stealthy attacks (e.g., Stuxnet).

We demonstrate FloGuard's capabilities against a modified real-world bot, namely Eggdrop [20], and several attacks against multiple real applications with a number of actual vulnerabilities in them. We show how FloGuard can integrate several off-the-shelf IDSes to detect a range of attacks that lie beyond any single tool's capabilities.

[1] We define a clean snapshot as the last system snapshot before all potential attack entry points, e.g., a socket connection, that could have influenced the detection point.

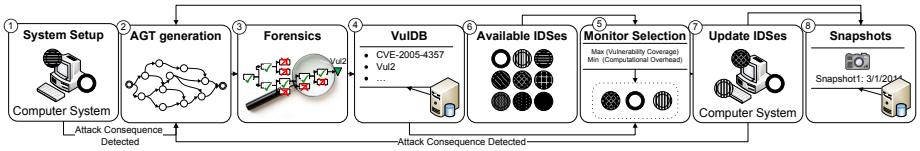


Fig. 1. FloGuard Architecture

2 Architecture

We begin by describing FloGuard's architecture and its overall operation. Figure 1 shows a bird's-eye view of different components in FloGuard. FloGuard assumes a virtual machine environment and requires the target system to operate as a guest VM[2]. FloGuard itself operates in the hypervisor/host OS of the VM environment to protect itself from attack and facilitate secure snapshotting facilities. Our prototype makes use of the Qemu [8] system emulator. The main inputs that FloGuard requires from a system administrator are a Vulnerability-Detector database (VulDB) and an initial set of "attack consequence detectors."

Attack consequence detectors are lightweight detectors that can operate continuously to detect the eventual symptoms of an intrusion, and cannot be disabled by an attack. Examples include in-network botnet/DDoS tracking done by ISPs and hypervisor-based file-integrity checkers. The output of an attack consequence detector is a process, port, or file that exhibits the symptoms of an attack. The VulDB encodes information about the kinds of vulnerabilities that FloGuard is to guard against and the intrusion detection systems available to it. The database does not require knowledge of the specific vulnerabilities that may actually be present in the target system (which are unknown), but just the high-level types, e.g., buffer overflow or SQL injection, that are possible and for which detection mechanisms are available.

During normal operation, FloGuard turns on the consequence detectors and periodically collects incremental snapshots of the system. It also keeps an append-only log of all system calls that are sent to a secure backend through a unidirectional communication link. When the attack consequence detector produces an alert (i.e., the detection point), FloGuard parses the syslogs, and determines the set of all potential attack sources (entry points) ((similar to [14, 16]). The last system snapshot taken before all the potential attack sources is marked as the last clean snapshot. FloGuard produces an Attack Graph Template (AGT), which by design consists of a superset of actual attack paths using the syslogs from the clean snapshot to the detection point[3].

Through an iterative forensics analysis process, FloGuard invokes intrusion detectors, during each iteration (attack repetition), to refine the AGT from possible attack paths into an actual path. For each iteration, FloGuard selects a new set of detectors, rolls the system back to a past clean snapshot, deploys the detectors, and waits for a repeat attempt of the attack. After the iteration, FloGuard uses the outputs of the intrusion detectors to determine which paths in the AGT can be implicated or eliminated, and produces a refined AGT that is a subset of the original one. Eventually, once the actual attack path gets identified, a mitigation mechanism can then be used to block similar attacks in future. Thus, only the detectors related to vulnerabilities that are present in the system and for which an exploit actually exists need to be deployed in the process.

[2] In fact, VMs are not strictly needed, but they make incremental snapshots more efficient.

[3] It is important to mention that FloGuard also addresses kernel vulnerabilities. The last possible exploitation by the attacker, which would give him or her the highest privilege, is the root escalation. And that last exploitation (i.e., a consequence) is also logged, because logs are sent real-time to a backend server, and are later used for forensics analysis.

Because FloGuard makes use of securely logged syscalls, it does not rely on any knowledge of what the attacker may do in the future. Additionally, because FloGuard works based on logged past activities within the system, it can work against social engineering attacks by tracing the attack path back to the executable which was downloaded during the social engineering phase of the attack.

3 Vulnerability-Detector DB

The vulnerability-detector database encodes all the domain knowledge about vulnerability types, detection mechanisms, and the applications in the system that are used by the forensics engine to make its detection decisions.

Vulnerability Types. In general, vulnerabilities are software flaws that are used by an adversary in the penetration step of an attack to obtain a privilege domain on a machine. The *vulnerability types* set in the VulDB includes all "possible" types of vulnerabilities that could potentially exist in different parts of the target system. A vulnerability type represents general vulnerability classes, e.g., buffer overflow, that encompass all target systems, without mentioning their context. It does not represent a *specific* vulnerability in the system, e.g., the vulnerable application's name and the exact location in the application's code. In addition, the VulDB also contains a target-system-specific mapping of vulnerabilities to processes in the system. The mapping can be positive or negative (e.g., the eVision application [4] could be vulnerable to a SQL injection attack while the sshd daemon could not). Since the system's actual vulnerabilities are unknown, these mappings represent *potential* vulnerabilities, not known ones. The mappings are optional, and FloGuard assumes that every process can be vulnerable to every vulnerability type if the mapping is missing.

Detection Mechanisms. The second element in the VulDB is the set of intrusion detection systems (IDSes) that are available to FloGuard to monitor different parts of the target system. Different kinds of IDSes that together cover as many different vulnerability types as possible are preferable. For each detection mechanism, FloGuard also requires a relative cost measure, e.g., CPU overhead, associated with the detector when it is deployed. The cost measure is only used to compare one intrusion detector against another; so as long as a uniform measure is used for all the detectors, it is not necessary for the cost to represent any specific performance or overhead measure. Ultimately, FloGuard's main objective is to protect a system from attack once its corresponding vulnerability gets identified. While each detector can be converted into a rudimentary intrusion mitigator (by restarting the target process once the detector detects an intrusion), specialized mitigation mechanisms that may not detect attacks but can block them can also be included in the VulDB database. For example, a "disable account" action cannot detect attacks per se, but can block password attacks.

Detector-Capability Matrix. The detector-capability matrix indicates the ability of a given IDS to detect various vulnerability types. The matrix is defined over the Cartesian product of the vulnerability type set and the set of IDSes. Each matrix element shows how likely it is that each IDS, due to false positives and negatives, could detect an exploitation of a specific vulnerability type. In our experiments, we have used discrete N, L, M, H and C notations to represent no, low, medium, high and complete coverages, respectively. The detection capability matrix is later employed by the forensics and detector selection algorithms in deciding on the minimum-cost set of intrusion detection systems with maximum exploit detection capability.

4 Attack Graph Templates

Generally, every cyber attack scenario (path) consists of a number of subsequent exploits. In other words, the adversary, with initially no access to the system, can subsequently exploit various vulnerabilities to achieve the privilege required for his or her

malicious goal, e.g., modifying a sensitive system file. Throughout this paper, exploits are represented as (process, vulnerability type) pairs in which the first and second elements denote, respectively, the vulnerable application, e.g., eVision, and the vulnerability type, e.g., buffer overflow, in the application.

Traditionally, an attack graph for a computer system represents a collection of known penetration scenarios according to a set of known vulnerabilities in the system [29]. In this section, we present the attack graph template (AGT), i.e., an extended attack graph, which is intended to cover all "possible" *attack types*. As an example, the attack graph template for a web server addresses the possibility that the server application might be vulnerable to buffer overflow attacks, even if there is no such known vulnerability in the application. The attack graph template is a state-based graph in which each state is defined to be the set of the attacker's privileges in that state. State transitions in AGT (each mapped to a vulnerability exploitation) represent privilege escalations. Formally, the AGT encodes all possible attack paths from the initial state, in which the attacker has no privilege, to the goal state, in which the attacker has achieved the required privilege to accomplish his or her malicious goals.

In general, unknown vulnerability exploitations in a given application could be in any part of the application code; however, almost all of them are of known *types*, such as buffer overflow or SQL injection. Furthermore, as each IDS can detect certain *types* of exploits, generating AGTs that consider all possible vulnerability exploitations in applications allows FloGuard to determine which state transitions in an AGT get detected if a particular set of IDSes are deployed. Additionally, as the AGT is designed to consider all exploit types that constitute a finite set, the order (#states) and size (#state transitions) of AGT are finite, and often manageable in practice (see Section 7).

Automatic AGT Generation. FloGuard is particularly interested in the syscalls that cause data dependencies among the OS-level objects[4]. A dependency relationship is specified by three things: a source object, a sink object, and a timestamp[5]. For example, the reading of a file by a process causes that process (sink) to depend on that file (source). Given a detection point and the syscall logs, the dependency graph is generated using an algorithm similar to BackTracker [16]. Syslogs are parsed and analyzed line by line from the beginning to the detection point; their corresponding source and sink objects are created or updated; and a directed edge, labeled with the event's occurrence time, is created between those nodes.

We run two optimizations on the dependency graph before converting it to AGT. First, using time-sensitive backward reachability analysis, we eliminate irrelevant vertices/edges that do not causally affect the state of the detection point [19]. Second, by calculating transitive closure of the graph, we get rid of all the non-process nodes; any pair of processes get connected if there is a causal directed path [19] between them consisting of only non-process nodes. Finally, the refined dependency graph is used to generate the AGT. The idea is to consider any dependency graph edge connecting two nodes from different privileges (security contexts), a potential vulnerability exploitation by the attacker to obtain the privilege of the process to which the edge directs.

To convert the dependency graph to AGT, we traverse the dependency graph and concurrently update the AGT. First, the initial state of AGT with an empty privilege set is created. Starting from each entry point node in the graph, we run a causal depth-first

[4] Throughout the paper, we use the term *OS-level objects* for processes, regular files, directories, symbolic links, character devices, block devices, FIFO pipes, and all thirty-five types of sockets, including internet sockets, i.e., `AF_INET`, interchangeably.

[5] We also log the security context of the objects. For instance, on a SE-Linux system, the web server directory is associated with the security type `httpd_sys_content_t`.

search (DFS), i.e., with increasing time tags on the edges of the paths being traversed. While DFS is recursively traversing the dependency graph, it keeps track of the current state in the AGT, i.e., the set of privileges already gained through the path traversed so far by DFS. When DFS traverses a dependency graph edge that crosses over privilege domains, a state transition in AGT happens if the current state in AGT does not include the privilege domain of the process to which the edge directs. The transition is between the current state and the state that includes exactly the same privilege set as the current state plus the privilege of the process directed by the dependent graph edge. In fact, more than one transition edge might be created, depending on how many vulnerability types could potentially exist in the process, according to the VulDB.

Once the depth-first search is over, AGT is generated such that all its terminal states include the privilege domain of the process that had caused the detection point event during the attack. The last step would be to add a *goal state* to the AGT and connect all the terminal states to it using NOP (No-OPeration) edges, meaning that no action is needed to make the transition. Once the AGT is generated, the forensics analysis (Section 5) tries to identify the exact path traversed by the attacker.

5 Intrusion Forensics

The forensics analysis by FloGuard requires two logging subroutines. First, we assume that an incremental snapshot of the system is taken periodically, e.g., once a day; therefore, we could go back to any time instant in the past that coincides with one of the snapshot times. Second, syscalls are being logged and stored in a secure back-end storage device while the system is operating. That enables FloGuard to generate the AGT for any past time interval.

FloGuard employs an iterative forensics algorithm; it restores a clean system snapshot and waits for attackers to launch similar attacks (exploiting the same vulnerability) several times. During each iteration, it deploys a different IDS to gather further evidence regarding the attack. The deployed IDSes are chosen based on their cost, detection capabilities, and the generated AGT. In particular, FloGuard chooses the intrusion detector d^* for each forensics iteration using the following equation: $d^* \leftarrow \arg\max_{d \in D}\{\texttt{Coverage}(AGT, d)/\texttt{Cost}(d)\}$, where D is the set of available IDSes; $\texttt{Coverage}(AGT, d)$ denotes the expected number of already-unmonitored transitions in AGT that are monitored by d. Using VulDB, FloGuard knows what vulnerability exploitations (state transitions) each IDS can detect; therefore, after each iteration, it can prune the AGT based on the deployed IDS and its alerts during the attack. More specifically, the detected vulnerability exploitations are marked, and the rest (the vulnerabilities whose exploitations did not get reported, while being monitored, by the deployed IDS) can safely be removed from the AGT. The refined AGT is used to choose the IDS for the next iteration (attack repetition). FloGuard continues the forensics iterations until the refined AGT consists of one marked path from its initial state to the goal state. The marked path is the actual path traversed during the attack.

In practice, detection systems are not always accurate, and sometimes either produce false positives or miss some misbehaviors (false negatives). FloGuard takes such inaccuracies into account by using their corresponding rates provided in VulDB[6] (otherwise, a default value is used) and defining the edge weights w^e (which are all initially set to 1). In particular, if an IDS d reports a vulnerability exploitation e during a forensics iteration i, the corresponding edge (state transition) in AGT is not completely removed;

[6] Before the calculations, the qualitative values in VulDB are mapped to their corresponding crisp values as follows. *N* and *C* are mapped to 0 and 1, respectively. *L*, *M*, and *H* are, respectively, mapped to 0.25, 0.5, and 0.75.

instead, its weight is updated using the equation $w_i^e \leftarrow w_{i-1}^e \times [1 - \texttt{FPR}(d,e)]$, according to its previous weight and the false positive rate (FPR) of the IDS d in monitoring the exploit e. Similarly, in case no incident is reported, the weight is updated based on the IDS's false negative rate using the equation $w_i^e \leftarrow w_{i-1}^e \times \texttt{FNR}(d,e)$.

Essentially, to provide a precise automated forensics analysis, FloGuard must traverse the time dimension back and forth. FloGuard's implementation provides two different solutions, which are conceptually identical. 1) If the infrastructure supports system-wide restore/replay, FloGuard uses it to restore the past snapshot and replay the whole system several times, running the same forensics analysis as mentioned above, until the exact attack path in AGT is identified. 2) If the system-wide restore/replay is not supported, as described in this section, instead of making use of a restore/replay mechanism, FloGuard waits for the attacker to repeat the attack in the future. As our main target is scripted attacks, worms, and malware threats, so the repetition assumption is reasonable. In the unlikely scenario that an unprotected exploit is never re-exploited (i.e., an attack never repeats), forensics may not even be required - a simple rollback to pristine state will suffice. In this paper, we focus on the second situation due to the space limit; however, the main concept is the same for both solutions.

6 Monitor Selection

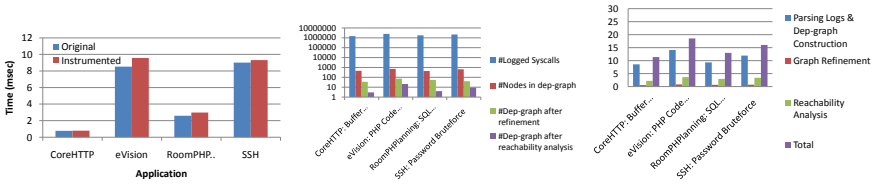
Once the attack path in the AGT is identified, FloGuard chooses the cost-optimal set of IDSes, as mitigation mechanisms (unless it is statically defined in VulDB) to deploy in the system permanently until the administrators install suitable patches. FloGuard selects and deploys a subset of IDSes that cooperatively detect and block exploitations of the *known* vulnerabilities represented by the refined AGT after the forensics analysis.

Formally, FloGuard decides upon the subset of IDSes to handle known vulnerabilities using $D_k^* = \arg\min_{D_i \subseteq D}[\sum_{d \in D_i} \texttt{Cost}(d) \times \arg\max_{AP \in AGT} \texttt{C}(AP, D_i)]$[7], where AP represents an attack path (sequence of exploits) from the initial state to the goal state in AGT. D is the set of available IDSes. The $\texttt{C}(AP, D_i)$ function denotes the overall cost if the system operates with IDSes $d \in D_i$ turned on. The overall cost is determined through consideration of two factors: 1) detection cost (performance penalty); and 2) damage cost by the attacker trying to get from the initial to the goal state through the attack path AP. Depending on the exploits in AP and the detection capability of the IDSes in D_i, AP might, but would not necessarily, be cut at some point between the initial and goal state by one of the detectors. Formally, the C function is defined as follows:

$$\texttt{C}(AP, D_i) = \sum_{e \in AP} \left[\texttt{Depth}(AP_e) . \prod_{e' \in AP_e} [w^{e'} . \min_{d \in D_i} \texttt{FNR}(d, e')] \right], \quad (1)$$

which formulates the damage by the attacker before he or she gets caught by any of the deployed IDSes D_i. Briefly, we used the detection latency from the initial exploit node to the detection point as the damage cost, since it determines how much rollback (and data loss) is needed. More formally, Equation (1) formulates the expected depth of penetration (number of subsequent vulnerability exploitations) the attacker can cause following the attack path AP without being detected by any of the deployed intrusion detectors D_i; to do so, the algorithm considers the penetration depth of each subattack AP_e (defined as the subpath of AP from the initial state to e) and the likelihood of it not being detected. The penetration depth for a subattack AP_e, denoted by Depth, is defined as the length of the path from the initial state to e through the attack path AP. The probability that the attack at a specific step AP_e is not yet detected is calculated by considering 1) the weights on each exploit e' in the subpath AP_e that have been

[7] In effect, the equation picks the subset of IDSes, that minimize the maximum possible cost that would result (according to AGT) if the system operated with that IDS subset deployed.



(a) Instrumentation Overhead (b) Logs and Graph Sizes (c) Time Requirements

Fig. 2. Automated AGT Generation for Four Attack Scenarios

updated through the forensics iterations; and 2) the false negative rates (denoted by FNR) of the deployed IDSes (more specifically, the IDS with the lowest false negative rate) regarding each exploit e' in the subpath AP_e. Consequently, the D_k^* equation above solves the tradeoff and selects the IDS set that minimizes the overall cost according to the AGT's structure. In practice, AGTs of actual attacks are small enough to permit a brute-force optimization of Equation (1).

7 Evaluations

We implemented FloGuard and evaluated it on a real botnet worm and different attack scenarios against four applications, each with a specific vulnerability. The vulnerability exploitations included buffer overflow exploitation, a SQL injection vulnerability, PHP remote code execution, and password attacks.

Experimentation Setup. The experiments were conducted on a system with 2.20 GHz AMD Athlon ^{TM}64 Processor 3700+ CPU, 1 MB of cache, and 2.0 GB of RAM. The host and guest OSes running on the machine were Ubuntu 9.04 with Linux 2.6.22 kernels. The production system included a web server with several PHP applications, including eVision content management system [4], and the RoomPHPlanning [3] scheduling application. Furthermore, the applications could connect to a MySQL database, and the trusted remote clients made use of SSH to obtain access to the system.

We used a set of IDSes that fall into the following categories. To block malicious use of library functions and malformed network packets, we used LibSafe [7] and Snort [27], respectively. To detect viruses and malicious actions on file system objects, we employed ClamAV [17] and Samhain [32], respectively. We employed Zabbix [5] and Memcheck in Valgrind [24] to detect anomalous activities, such as DoS or brute-force attacks, and general memory access violations, respectively. We used the TEMU [30] system-wide taint-tracking engine, which runs on the host OS. More specifically, using TEMU, one can mark some input data, such as network interfaces, as tainted, and then TEMU will track the information flow and store the executed instructions in a trace file on the host OS. To actually make use of TEMU, we had to improve its capability to produce higher-level information (not only instruction-level) regarding file-system objects, such as names of the files that are being dynamically tainted. We improved its implementation by using The Sleuth Kit (TSK) [9] to read the file system in the virtual machine's image file; this enabled us to dynamically translate disk-level tainted addresses, which are generated by TEMU, to file system object names, such as file names and their absolute addresses.

Services and Vulnerabilities. Next, we describe the vulnerabilities and the affected services in our experiments. *SQL injection*: According to CVE-2009-4669, multiple SQL injection vulnerabilities in RoomPHPlanning 1.6 allow attackers to execute arbitrary SQL commands. *Buffer overflow*: Based on CVE-2007-4060, an off-by-one error in the CoreHTTP 0.5.3.1 web server allows remote attackers to execute arbitrary code via an intelligently handcrafted HTTP request. *PHP remote code execution*: According to CVE-2008-6551, multiple directory traversal vulnerabilities in e-Vision 2.0 allow attackers to include and execute arbitrary PHP files. *The weak password*: Our production

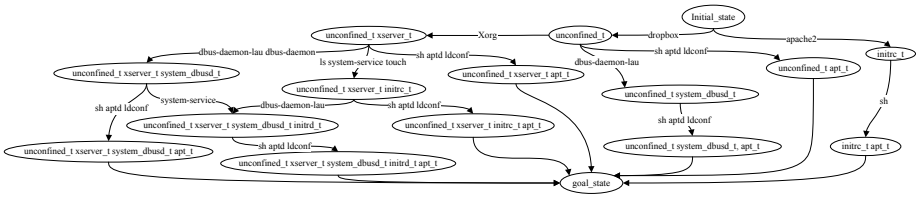


Fig. 3. An Automatically Generated AGT for Remote PHP Code Execution

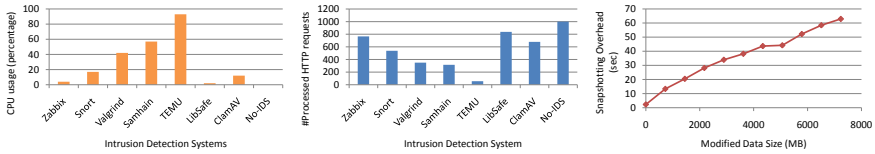
system includes accounts with weak passwords that open up the opportunity to make use of password-cracking software tools, e.g., John the Ripper [2]. We implemented a Perl password brute-force script that reads a file storing a large number of passwords and tries them against the target system.

Instrumentation Overhead. We implemented the syscall interception as a loadable kernel module; however, recently, other approaches like [18] have proposed interception of system calls from within the hypervisor using the previously generated signatures of the process memory images. However, that approach also assumes that the root domain is tamper-proof, since the attacker with root access can modify kernel data structures and consequently make the signatures useless. We configured the `/etc/syslog.conf` file such that all the syslogs are directly sent through a uni-directional link to a secure backend machine; therefore, all the system calls logged before the attacker gets access to the root domain are trusted.

The syscall interception module is always loaded while the system is operating. We measured the performance overhead by FloGuard's syscall interception module (see Figure 2(a)). For the first three applications, the figure shows the average response time for subsequent HTML requests for the CoreHTTP server, and PHP page requests for the eVision and RoomPHPlanning applications. The SSH results show the time required to transfer 100 KB of data from the server. In fact, our logging requirements are very modest. We only log a subset of syscalls that indicate data flow between processes/files (15 out of 350 call types) and only the first interaction when multiple syscalls are present between the same nodes. We ignore all syscall arguments except source or destination process/file IDs, and our encoding requires 10 bytes per syscall. Our 4 attack scenarios produced 40k records per minute (i.e., 576MB/day). Furthermore, syscall types repeat frequently, and after compression, each trace required an average of 10MB per day.

Attack-Graph Template Generation. We have collected the intercepted syscalls for the four abovementioned attacks. Figure 2(b) shows the number of syscalls for each attack on the system (the first columns). For each attack, once the attack consequence detector reports the detection point event, which was a sensitive file modification for all the attacks, the parser started reading the syslogs line by line (approximately 300K lines per second) and automatically creates the directed dependency graph. The second columns show the number of nodes in the generated dependency graph for each attack. The third columns illustrate the number of nodes in the generated dependency graph after non-process nodes are removed. The last phase (fourth columns) before generation of the AGTs is to further prune the dependency graph using reachability analysis.

Figure 2(c) shows how long (seconds) each of the abovementioned steps took. Because in our implementations, the syslogs parsing and the initial dependency graph creation are done concurrently, we report the time they took as a single result (first columns in Figure 2(c)). The second and third columns report the results for the graph refinement steps. Most of the total time is spent on parsing the syslogs and producing the initial dependency graph. As the reachability analysis significantly prunes the dependency graph, the time required to convert the resulting graph to AGT was less than 1 second in our experiments. Figure 3 shows the automatically generated AGT (using



(a) CPU Usage Cost (b) Impact on Availability (c) Snapshotting Overhead

Fig. 4. Cost Evaluation of Individual Intrusion Detection Systems and the Snapshotting Procedure

the `Graphviz-dot` tool) for the remote PHP code execution attack scenario. The AGT is represented using the SE-Linux privilege domains; however, for systems with traditional two-level discretionary access control, i.e., user and root, the nodes in AGT will have those two privilege levels only.

IDS Computational Cost. We measured the CPU overhead for individual IDSes. As illustrated in Figure 4(a), the TEMU engine puts the highest load on the system. Second, we deployed all of the IDSes and evaluated their impact on the system's overall throughput (see Figure 4(b)), which is defined as the maximum number of client requests, generated by HTTPTrafficGen [1], that could be processed within a fixed amount of time.

Periodic Snapshots. We measured the average performance overhead of the incremental system-wide snapshotting. Figure 4(c) illustrates the time needed for the engine to snapshot the whole system given the amount of data modified since the previous snapshot. As a case in point, if 2 GB of data are modified, e.g., downloaded, in the virtual machine between two successive snapshots, i.e., 30 minutes in our experiments, it takes about 13.4 seconds on average to pause the system and take and store a complete system-wide snapshot[8]. In our experiments, the snapshot restoration process was done quite fast, i.e., 3.2 seconds on average.

Intrusion Forensics. Finally, we present iterative intrusion forensics analysis results for six different attack scenarios. As the consequence detector, Samhain was configured to check the marked sensitive files and directories against its database and fire an alert upon identifying a modification or access.

First, we start with the buffer overflow attack scenario. While the CoreHTTP web server was operating after the snapshot, we remotely launched a buffer overflow exploit, which we had created manually using GDB, and got shell access to the machine. We then modified the web server's configuration file, i.e., `chttp.conf`, which had been marked to be monitored by Samhain. Upon receiving the Samhain alert, FloGuard started its forensics analysis by parsing the syscall logs from the last snapshot to the detection point, and generating the AGT. As shown in Table 1, the initial AGT consisted of 6 possible attack paths based on the intercepted syscalls during the attack. Having employed the monitor selection algorithm, FloGuard picked Valgrind as the first detector, as it maximized the coverage/cost measure, to deploy to monitor the web server. Consequently, the past clean snapshot was retrieved, Valgrind was deployed, and the system started its normal operation while FloGuard was waiting for the next repetition of the attack. We then relaunched the attack. Valgrind did not detect the buffer overflow in CoreHTTP, since it does not perform bounds checking on static arrays (allocated on the stack). After the first iteration, AGT was pruned, and the resulting AGT consisted of 3.7 expected number of attack paths (the fractional number is due to IDS uncertainties). Using the refined AGT and the detector-capability matrix, FloGuard chose the next detector, i.e., Valgrind on the `sh` process, and the iterative forensics continued until Libsafe was picked to monitor the web server that successfully detected the buffer

[8] Using the VirtualBox framework, taking live snapshots, w/o pausing the system, is possible.

Table 1. Iterative Intrusion Forensics Analysis

Attacks	CoreHTTP: Buffer Overflow				eVision: Remote Code Execution				SSH: Password Brute-force				RoomPHPlanning: SQL Injection			
DP	/var/www/chttp.conf				/etc/passwd				/var/log/auth.log				/var/lib/mysql/RMP/rp_resa.MYD			
	#Paths	IDS	Overhead	Dctd?	#Paths	IDS	Overhead	Dctd?	#Paths	IDS	Overhead	Dctd?	#Paths	IDS	Overhead	Dctd?
It.1	6	V(corehttp)	1.8X	No	53	TEMU(sh)	16.8X	No	4	V(ssh)	0.1X	No	5	V(mysql)	1.2	No
It.2	3.7	V(sh)	1.3X	No	9	V(apache2)	1.7X	No	1.7	Zabbix	0.3X	No	2.6	Zabbix	0.3	No
It.3	1.4	LS(corehttp)	0.2X	Yes	6.8	V(sh)	0.2X	No	0.9	LS(ssh)	0.1X	No	1.7	Snort	0.8	Yes
It.4	0.9				4.4	Zabbix	0.3X	No	0.4	Snort	0.3X	Yes	0.9			
It.5					3.4	ClamAV	0.5X	Yes	0.3							
It.6					2.4											

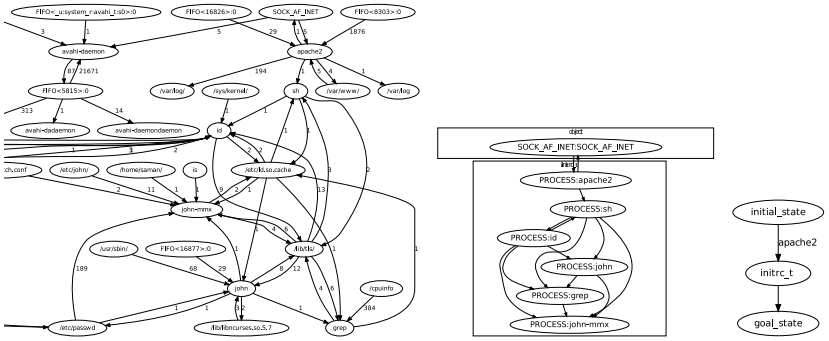
overflow in CoreHTTP. Consequently, LibSafe was permanently turned on to detect and block similar attacks until the administrator manually patches the system.

The second attack scenario was the PHP remote code execution in the eVision-2.0 CMS application. We launched the attack using the Perl exploit from http://www.exploit-db.com against eVision-2.0 that enabled us to upload an arbitrary file using the local file inclusion. Consequently, we could remotely execute any arbitrary command on the server. As shown in Table 1, we modified the `/etc/passwd` file, which was marked to be monitored by Samhain. The first detector to turn on for the forensics analysis was TEMU, because of its capability in tracing back the data from the process `sh` that caused the detection point event (see Figure 3). The sh process was then traced by TEMU's `tracebyname` command, and the actual data source, i.e., the apache2 process (see Figure 3), was identified via the `list_tainted_files` command, which we had implemented by translating disk-level addresses to filenames. TEMU helped FloGuard to prune the AGT to include only the paths exploiting possible vulnerabilities in the apache2 process. Finally, the ClamAV detection system detected the uploaded file during the attack, and FloGuard, using the VulDB, decided to turn off the `magic_quotes` (even though this might have affected other applications).

The third attack, i.e., SSH password brute-force, was remotely launched using a Perl password trial script. Subsequent password trials made Samhain fire an alert upon the `/var/log/auth.log` file modification. In forensics analysis, Snort finally detected the password brute-force attack. The next attack scenario was to modify a sensitive database file through the exploitation of a SQL injection. Upon receiving the Samhain alert, FloGuard, as shown in Table 1, selected Valgrind and Zabbix to be deployed in the first and second iterations, respectively; however, neither detected any misbehavior in the system. Finally, in the last iteration, FloGuard picked Snort, which successfully detected the SQL injection by identifying SQL meta-characters in the incoming data.

Both Snort rules picked by FloGuard in attacks #3 and #4 are non-standard rules (one written by us and one disabled by default) that cannot be permanently deployed because they have a high likelihood of false positives. On the other hand, FloGuard correctly identifies the specific rule that must be enabled, and which port to enable it on only when an attack is detected, thus eliminating false positives. Furthermore, because FloGuard can keep multiple detectors for the same type of attack on standby, it degrades gracefully. E.g., If the Snort rule had missed the MySQL injection attack #4 because HTTPS was used, then FloGuard would have picked the much more expensive TEMU as the detection mechanism. Such graceful degradation cannot be achieved by static deployment. For attack #2, the ClamAV detector checks only for the presence of a commonly used PHP payload. Since the actual exploit is assumed to be zero-day without a patch, and the mitigation action turns off PHP magic quotes. Doing so permanently can impair system functionality. Finally, other "detectors" such as disabling an account or quarantining a process are even more disruptive and can never be deployed permanently. But FloGuard can use them. We will add these clarifications in the prose.

We also experimented with FloGuard on a multi-step attack scenario. We deployed the Zabbix consequence detector in the target machine. First, having exploited the eVision vulnerability, we got shell access on the target system. Then, to maintain control over the system, through reboots and software patches, we tried to get the adminis-



(a) The Dependency Graph (b) The Refined Graph (c) AGT

Fig. 5. The Generated Graphs for the Multi-step Attack

trative password of the system using the John the Ripper password cracker. The tool started subsequent password trials that over-consumed the system's computational resources (96% CPU usage on average); this caused the Zabbix anomaly detector to fire an alert, making FloGuard start the forensics analysis. Due to space limits, only a portion of the generated dependency graph (which had 286 vertices) is shown in Figure 5(a). `john-mmx` was marked as the process causing the detection point. Figure 5(b) shows the graph, after the refinement procedures, which had a total of 7 vertices. Figure 5(c) shows the generated AGT, in which the `apache2` process is the only possible entry point (initial vulnerability exploitation) for the attack. FloGuard went through the iterative forensics analysis and picked Valgrind for the first detector that was unable to detect any misbehavior; however, the second chosen detector, i.e., ClamAV, detected the downloaded file `osirys.txt.gif` during the PHP code execution. Consequently, FloGuard then turned off the magic_quotes in the target machine.

Finally, we evaluated FloGuard against an updated real and well-known IRC botnet worm, namely Eggdrop [20]. We remotely launched the worm, hitting the target system as one of its victims. The worm accomplished several actions: it exploited the buffer-overflow vulnerability in the CoreHTTP web server; it downloaded, installed, and launched the Eggdrop package; the bot got connected to the `EFNet` IRC channel and then started listening on the tcp port `3355` to commands received through telnet from the remote attackers; later, the bot scanned the network to find the next victim machine to compromise; finally, it received remote commands trying to access some sensitive files in the system. The Samhain file integrity checker detected the file access and marked the event as the detection point. Figure 6 shows the generated AGT for the attack. On the third forensics iteration, FloGuard chose to deploy LibSafe, which successfully detected the buffer overflow violation. Consequently, LibSafe was deployed permanently to block similar attacks in the future.

8 Related Work

Several papers in the literature propose parts of what FloGuard achieves. However, we are not aware of any other approach that can perform on-demand, cost-aware intrusion detector deployment to defend against multi-stage attacks affecting multiple parts of a system. Several approaches perform alert correlation across multiple IDSes like [12]. Also, BotHunter [15] introduce a aggregation algorithm to recognize successful bot infections. However, both methods assume that all the appropriate intrusion detectors have already been chosen and deployed.

The concept of on-demand re-execution in a modified environment has also received some attention. Rx [26] and First-Aid [13] re-execute applications in a different execu-

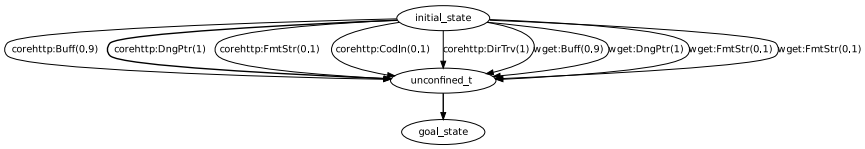


Fig. 6. The Generated AGT for Eggdrop Botnet Worm

tion environment, but they are not targeted towards security exploits. Bouncer [10], Vigilante [11], and Sweeper [31] combine an IDS along with program slicing or symbolic execution to trace-back from the detection point and produce input filters that can block the exploit packets. FLIPS [21] employs re-execution with instruction set randomization to detect root vulnerabilities. Shadow Honeypots [6] use re-execution in space (i.e., another machine) instead of time. However, most of these techniques only support detection of a single type of vulnerability (usually memory errors), and rely on the ability to detect attacks within the same process as the exploit entry-point. They cannot trace multi-step attacks that are detected in other parts of the system. Moreover, they do not consider multiple types of detectors and associated cost factors. FloGuard complements systems such as Sweeper by tracing detection points across multiple processes.

Attack graphs [35] have been extensively used to document known system vulnerabilities and attack paths. Two main drawbacks of the current approaches are 1) their inability to address unknown attacks, e.g., zero-day attacks, and 2) to improve scalability, their logic-based state notion does not represent system-level detailed information, significantly limiting their practical usage.

There has also been work on intrusion forensics analysis. Mukkamala et al. [22] use neural networks to discover sources of information breaches. Carrier [9] presents file-system-based forensics techniques to determine the source of security breaches by investigating their effects on the file-objects. Taser [14], BackTracker [16] and Panorama [33] aid off-line forensic analysis by producing taint-traces of file and process connections that led to a detected security breach. Because these forensics tools are based on passive data collection, they are either very pessimistic, marking most activities as malicious, or optimistic, thus missing many malicious activities that occur during an attack. In comparison, because FloGuard can actively deploy additional detection mechanisms to validate or refine its suspected attack paths, it can support much more realistic analysis. Nevertheless, pessimistic techniques that automatically produce system-level taint-graphs can be used to automatically produce initial AGTs for FloGuard.

intrusion prevention solutions (IPS) [35] have mainly focused on how to recover from attacks after the system is compromised. Zonouz et al. [35] introduce RRE, a game-theoretic IPS, whose goal is to take cost-optimal responsive actions against the adversary. EMERALD [25], a dynamic cooperative IPS, introduces a layered approach to correlate monitor reports through different abstract layers of the network. Unlike FloGuard, IPS solutions assume that a complete set of monitors are already deployed, and their main objective is not to identify previously unknown system vulnerabilities and exploitations to avoid identical attacks in the future.

9 Conclusion

We presented FloGuard, a cost-aware intrusion forensics system that uses online forensics and on-demand IDS deployment. FloGuard enables systems to defend against attacks that exploit various classes of previously unknown vulnerabilities. Our experiments show that FloGuard can deploy off-the-shelf IDSes only when they are needed and help protect systems against previously unknown vulnerabilities with minimal snapshotting overheads during normal operation.

References

1. HTTPTrafficGen, http://www.nsauditor.com/, 2008.
2. John the Ripper, http://www.openwall.com/john/, 2008.
3. RoomPHPlanning, http://www.beaussier.com/, 2008.
4. e-Vision, http://sourceforge.net/projects/e-vision/, 2009.
5. Zabbix, http://www.zabbix.org/, 2010.
6. K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *USENIX-Security*, page 9, 2005.
7. A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX-ATC*, pages 251–62, 2000.
8. F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX-ATC*, page 41, 2005.
9. B. Carrier. *File System Forensic Analysis*. Addison-Wesley Prof., 2005.
10. M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *SOSP*, pages 117–30, 2007.
11. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, pages 133–47, 2005.
12. H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *RAID*, pages 85–103, 2001.
13. Q. Gao, W. Zhang, Y. Tang, and F. Qin. First-aid: Surviving and preventing memory management bugs during production runs. In *EuroSys*, pages 159–72, 2009.
14. A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *SOSP*, pages 163–76, 2005.
15. G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *USENIX-Security*, pages 1–16, 2007.
16. S. T. King and P. M. Chen. Backtracking intrusions. *SOSP*, 37(5):223–36, 2003.
17. T. Kojm. ClamAV: http://www.clamav.net/, 2009.
18. S. Krishnan, K. Z. Snow, and F. Monrose. Trail of bytes: Efficient support for forensic analysis. In *CCS*, pages 50–60. ACM, 2010.
19. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *ACM-Comm.*, 21(7):558–65, 1978.
20. C. Li, W. Jiang, and X. Zou. Botnet: Survey and case study. *ICICIC*, pages 1184–87, 2009.
21. M. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. Flips: Hybrid adaptive intrusion prevention. In *RAID*, pages 82–101, 2005.
22. S. Mukkamala and A. H. Sung. Identifying significant features for network forensic analysis using artificial intelligent techniques. *IJDE*, 1, 2003.
23. S. Nagaraja, P. Mittal, C. yao Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis.
24. N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Runtime-Verification WS*, 2003.
25. P. Porras and P. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. *Proc. of the Info. Systems Security Conf.*, pages 353–65, 1997.
26. F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies: A safe method to survive software failures. In *SOSP*, pages 235–48, 2005.
27. M. Roesch. Snort: Lightweight intrusion detection for networks. In *USENIX-LISA*, pages 229–38, 1999.
28. O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, pages 159–169, 2004.
29. B. Schneier. Attack trees. *Dr. Dobb's Journal*, 1999.
30. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.
31. J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. *EuroSys*, 41(3):115–28, 2007.
32. B. Wotring, B. Potter, M. Ranum, and R. Wichmann. *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.
33. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, pages 116–27, 2007.
34. S. A. Zonouz, K. R. Joshi, and W. H. Sanders. Cost-aware systemwide intrusion defense via online forensics and on-demand detector deployment. In *CCS-SafeConfig*, pages 71–74, 2010.
35. S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley. RRE: A game-theoretic intrusion Response and Recovery Engine. In *DSN*, pages 439–48, 2009.