Patrol: Revealing Zero-Day Attack Paths through Network-Wide System Object Dependencies

Jun Dai, Xiaoyan Sun, and Peng Liu

College of Information Sciences and Technology,
Pennsylvania State University, University Park, PA 16802
{jqd5187,xzs5052,pliu}@ist.psu.edu

Abstract. Identifying attack paths in enterprise network is strategically necessary and critical for security defense. However, there has been insufficient efforts in studying how to identify an attack path that goes through unknown security holes. In this paper, we define such attack paths as *zero-day attack paths*, and propose a prototype system named Patrol to identify them at runtime. Using system calls, Patrol builds a *network-wide system object dependency graph* that captures dependency relations between OS objects, and identifies *suspicious intrusion propagation paths* in it as candidate zero-day attack paths through forward and backward tracking from intrusion symptoms. Patrol further identifies highly suspicious candidates among these paths, by recognizing indicators of unknown vulnerability exploitations along the paths through rule-based checking. Our evaluation shows that Patrol can work accurately and effectively at runtime with an acceptable performance overhead.

1 Introduction

1.1 Zero-Day Attack Paths

When deploying enterprise network security defense, it is important to consider multi-step attacks. Given that today's network is usually under basic protection from security deployments like firewall and IDS, it's not easy for attackers to directly break into their final target. Instead, determined attackers patiently compromise other intermediate hosts as stepping-stones. That is, attackers often have to go through an *attack path* before they achieve their goal. An attack path is a sequence of vulnerability exploits on compromised hosts. It's necessary and critical to find the attack paths hidden in the network.

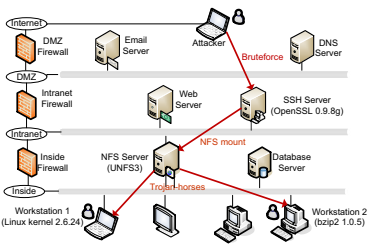


Fig. 1. An example attack scenario

Suppose that a host is compromised by a local or remote exploit. If this exploit is enabled by a known vulnerability, it's not zero-day. If this

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 536–555, 2013.

exploit is enabled by an unknown vulnerability, it is zero-day. If an attack path includes one or more zero-day exploits, it is a *zero-day attack path*.

Fig. 1 illustrates an example attack scenario including three steps. Step 1, a brute-force key guessing attack is used to exploit *CVE-2008-0166* on *SSH Server* to gain root privilege. Step 2, the export table on *NFS Server* is inappropriately configured to allow any user to share files through a public directory (*/exports*), so two crafted trojan-horses are uploaded to this directory. The trojan-horses contain exploit code of *CVE-2009-2692* and *CVE-2011-4089*. Step 3, once a trojan-horse file is mounted and installed by an innocent user like *Workstation 1* or *2*, arbitrary code is executed to create a hidden channel. Hence, two attack paths exist: *p1*{*CVE-2008-0166, NFS misconfiguration, CVE-2009-2692*} and *p2*{*CVE-2008-0166, NFS misconfiguration, CVE-2011-4089*}. Let's assume the time now is *August 1, 2009*, then *CVE-2008-0166* becomes the only known vulnerability. If the attackers are still able to exploit all the vulnerabilities in this scenario, then *p1* and *p2* both become zero-day attack paths.

Zero-day exploit problem is so important and challenging. Zero-day attack path problem is beyond zero-day exploit problem. This paper aims to take the first steps to address the zero-day attack path problem.

1.2 Possible Solutions

The literature is explored for possible solutions of zero-day attack path problem. However, we find that no existing technique can well address this problem due to the unknown nature of zero-day attack path.

Attack graph [1–3]. By considering vulnerabilities in combination (not merely in isolation), attack graph can generate attack paths that show exploit sequences to specific attack goals. But, this notion has been primarily applied to model causality dependencies among known vulnerabilities. Unknown vulnerabilities are not captured and zero-day attack paths will accordingly be missing in attack graph. Notable exceptions are recent research [4] [5], which have pioneered the attack graph based analysis and modeling of zero-day vulnerabilities. However, a solution to identify zero-day attack paths at runtime is further expected.

Penetration test [6–8]. This solution uses real exploits to reveal some speculated attack paths. It requires huge knowledge and operation input from human intelligence. Hence, the cost is usually too expensive. Besides, the attack paths in their discovery are largely known ones, because it's very difficult to exploit unknown vulnerabilities in penetration tests.

Alert correlation [9] [10]. This solution correlates isolated alerts to form potential attack paths. Although it has potentials to be automatic and inexpensive, it may induce high false rates. The false rates are twofold: 1) The correlation itself is inaccurate because it attempts to integrate possibly different contexts into a unified "story"; 2) The alerts that the correlation largely depends on genetically inherit false rates from security sensors like IDS. When the two folds of false rates are combined together, the accuracy of the whole solution gets worse.

Techniques to detect zero-day exploits may help the identification of zero-day attack path, such as anomaly detection [11–18] and specification-based detection

[19] [20]. By profiling normal behavior and detecting deviations, these techniques are capable of detecting novel exploits. However, they are hard to cope with false positives. Besides, the identification of novel exploits doesn't mean the identification of zero-day attack paths. As pointed above, IDS alert correlation needs to be involved and thus introduces one more fold of false rates.

1.3 Key Insights and Our Approach

This paper leverages a different strategy to identify the zero-day attack paths. Instead of first collecting vulnerabilities or alerts and then correlating them into paths, we first try to build a superset graph and identify the suspicious intrusion propagation paths hidden in it as candidate zero-day attack paths, and then recognize the highly suspicious candidates among these paths. Interested readers can refer to Fig. 2 for an example of a superset graph (Fig. 2a) and the suspicious intrusion propagation paths (Fig. 2b) hidden in it.

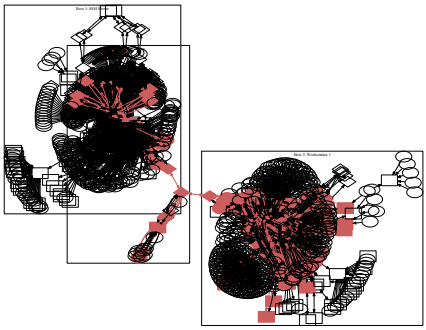
We make this decision for four *key insights*: 1) As the only way for programs to interact with OS, system calls are found hard-to-avoid and attack neutral; 2) We find that a network-wide superset graph can be generated from system calls, and zero-day attack paths are showing themselves in it. This graph is also attack neutral. It exists no matter whether any vulnerability is exploited or not; 3) The superset graph is inherently a set of paths. We find a way to get its appropriate subsets as candidate zero-day attack paths. These paths actually and naturally correlate vulnerability exploitations, different from the logical correlation in attack graph; 4) The candidate zero-day attack paths expose unknown vulnerability exploitations along them, and thus can orientate us to recognize such exploitations. With these paths serving as network-wide attack context, the accuracy and performance of detecting unknown vulnerability exploitations can be better than the detection with only isolated per-host context.

The following summarizes our main contributions:

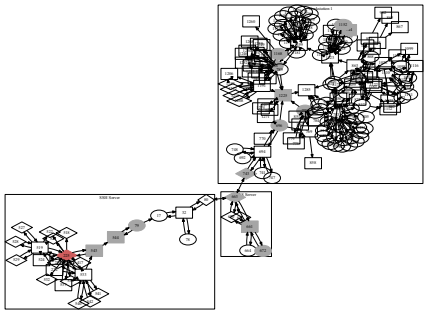
1. We propose to build a *network-wide system object dependency graph* (SODG) as the superset graph. Built from system calls, an SODG is made up of OS objects like processes/files/sockets (nodes) and dependency relations between them (edges). It neutrally captures the occurrence of vulnerability exploits.
2. We propose to identify *suspicious intrusion propagation paths* (SIPPs) in the network-wide SODG as candidate zero-day attack paths. The SIPPs actually and naturally correlate known/unknown vulnerability exploitations. We further coin the concepts of vulnerability shadow and shadow indicator to help recognize the highly suspicious candidates among the SIPPs.
3. We implemented a prototype system, called *Patrol*, which can work accurately and effectively at runtime with an acceptable performance overhead.

2 Models and Assumptions

We assume a network consists of Unix-like operating systems, in which system objects can be mainly classified into processes, files and sockets. We propose to



(a) An example 3-host SODG for the attack scenario in Fig. 1, with *1288* OS objects from *143120* system calls. The SIPPs hidden in it is highlighted in red.



(b) The red colored SIPPs hidden in (a), with *175* OS objects. The trigger node is highlighted in red and other verified malicious nodes in grey.

Fig. 2. This figure is to show what the SODG and SIPPs are like. A box contains a per-host SODG, in which a rectangle denotes a process, a diamond denotes a socket, and an ellipse denotes a file. *They look unreadable because of the fine granularity at OS-level and the scale of network. Readers are not expected to understand the details.* A main merit of Patrol is that it can dig out SIPPs from the network-wide SODG.

build a network-wide system object dependency graph (SODG) using system call traces. Since a system call is designed to be the only way to get service from OS in modern operating systems, attackers have to talk to the system via system calls. Therefore, although unknown exploits could not be seen by us, they can be seen by SODG. Fig. 2a gives an example of a 3-host SODG.

To build a network-wide SODG, we first need to construct the SODG for each host, namely per-host SODG. As in Definition 1, a per-host SODG is a directed graph made up of OS objects (nodes) and dependency relations (directed edges) between them. System calls are parsed to generate these nodes and edges. There are several types of dependency relations. For example, system call *read* infers that a process depends on a file (denoted as *file*→*process*), while *write* determines that a file depends on a process (*process*→*file*). Table 1 gives the dependency rules to help generate dependency relations from system calls. *start* and *end* respectively denote the timestamp at which a system call is invoked and returned.

Definition 1. *per-host System Object Dependency Graph*
If the system call trace for the i-*th* host is denoted as Σ_i, then the per-host SODG for the host is a directed graph $G(V_i, E_i)$, where:

- V_i is the set of nodes, and initialized to empty set $\varnothing$;
- E_i is the set of directed edges, and initialized to empty set $\varnothing$;
- If a system call $syscall \in \Sigma_i$, and *dep* is the dependency relation parsed from *syscall* according to dependency rules in Table 1, where $dep \in \{(src \rightarrow sink), (src \leftarrow sink), (src \leftrightarrow sink)\}$, *src* and *sink* are OS objects (mainly a process,

Table 1. System call dependency rules

Dependency	Events	System calls
process→file	process modifies file	write, pwrite64, rename, mkdir, linkat, link, symlinkat, symlink, fchmodat, fchmod, chmod, fchownat, mount
file→process	process uses but does not modify file	stat64, lstat64, fstat64, open, read, pread64, execve, mmap2, mprotect, linkat, link, symlinkat, symlink
process↔file	process uses and modifies file	open, rename, mount, mmap2, mprotect
process→process	process creation or termination	vfork, fork, kill
process↔process	process creation	clone
process→socket	process writes socket	write, pwrite64
socket→process	process checks or reads socket	fstat64, read, pread64
process↔socket	process writes socket	mount, connect, accept, bind, sendto, send, sendmsg, recvfrom, recv, recvmsg
socket↔socket	process reads or writes socket	connect, accept, sendto, sendmsg, recvfrom, recvmsg

file or socket), then $V_i = V_i \cup \{src, sink\}$, $E_i = E_i \cup \{dep\}$. *dep* inherits timestamps *start* and *end* from *syscall*;
- If $(a \rightarrow b) \in E_i$ and $(b \rightarrow c) \in E_i$, then c transitively depends on a.

As shown in Definition 2, the network-wide SODG is constructed by recursively concatenating the per-host SODGs. If and only if at least one directed edge exists between two nodes from two different SODGs, these two SODGs can be concatenated together (by the $\cup$ operation in Cantor set theory).

Definition 2. *network-wide System Object Dependency Graph*
If the per-host SODG for the i-*th* host is denoted as $G(V_i, E_i)$, then the network-wide SODG can be denoted as $\cup G(V_i, E_i)$, where:

- $\cup G(V_2, E_2) = G(V_1, E_1) \cup G(V_2, E_2) = G(\cup V_2, \cup E_2)$, iff $\exists obj_1 \in V_1$, $obj_2 \in V_2$ and $dep_{1,2} \in \cup E_2$, where $dep_{1,2} \in \{obj_1 \leftarrow obj_2, obj_1 \rightarrow obj_2, obj_1 \leftrightarrow obj_2\}$. $\cup V_2$ denotes $V_1 \cup V_2$, and $\cup E_2$ denotes $E_1 \cup E_2$;
- $\cup G(V_i, E_i) = \{\cup G(V_{i-1}, E_{i-1})\} \cup \mathrm{G}(V_i, E_i) = \mathrm{G}(\cup V_i, \cup E_i)$, iff $\exists obj_{i-1} \in \cup V_{i-1}$, $obj_i \in V_i$ and $dep_{i-1,i} \in \cup E_i$, where $dep_{i-1,i} \in \{obj_{i-1} \leftarrow obj_i, obj_{i-1} \rightarrow obj_i, obj_{i-1} \leftrightarrow obj_i\}$. $\cup V_i$ denotes $V_1 \cup \cdots \cup V_i$, and $\cup E_i$ denotes $E_1 \cup \cdots \cup E_i$.

The network-wide SODG is inherently a set of paths. A zero-day attack path will be one of them if it exists. Hence, we propose to identify suspicious intrusion propagation paths (SIPPs) in the network-wide SODG as candidate zero-day attack paths.

As in Definition 3, the SIPPs are a subgraph of the network-wide SODG, of which the OS objects are all "suspicious": given a trigger node *tn*, they either have affected *tn* through direct or transitive dependency relations before $lat(tn)$, or have been affected by *tn* after $eat(tn)$. Trigger nodes refer to SODG objects that are involved in the alerts from existing security sensors, such as Snort [21], Tripwire [22], or our system itself.[1] We assume trigger nodes can be noticed by administrators. The SIPPs inherently reveal the attacker's trace at OS level.

[1] To reduce dependency on efficiency of security monitoring tools, Patrol implements another mode: heavy mode, in which Patrol feeds itself with its own alerts as seeds.

Fig. 2b gives an example of the SIPPs hidden in the 3-host SODG, using the SSH socket (node *225*) noticed from the Snort alert "SSH potential brute force attack" as the trigger node.

Definition 3. *Suspicious Intrusion Propagation Paths* (SIPPs)
If the network-wide SODG is denoted as $\cup G(V_i, E_i)$, where $G(V_i, E_i)$ denotes the per-host SODG for the i-*th* host, then the SIPPs are a subgraph of $\cup G(V_i, E_i)$, denoted as $G(V', E')$, where:

- V' is the set of nodes, and $V' \subset \cup V_i$;
- E' is the set of directed edges, and $E' \subset \cup E_i$;
- V' is initialized to include *trigger nodes* only;
- For $\forall obj' \in V'$, if $\exists obj \in \cup V_i$ where $(obj \rightarrow obj') \in \cup E_i$ and $start(obj \rightarrow obj') \leq lat(obj')$, then $V' = V' \cup \{obj\}$ and $E' = E' \cup \{(obj \rightarrow obj')\}$. $lat(obj')$ maintains the *latest access time* to obj' by edges in E';
- For $\forall obj' \in V'$, if $\exists obj \in \cup V_i$ where $(obj' \rightarrow obj) \in \cup E_i$ and $end(obj' \rightarrow obj) \geq eat(obj')$, then $V' = V' \cup \{obj\}$ and $E' = E' \cup \{(obj' \rightarrow obj)\}$. $eat(obj')$ maintains the *earliest access time* to obj' by edges in E'.

A network-wide SODG can be unmanageably complex. A main merit of Patrol is that it can dig out SIPPs from the network-wide SODG. The size of the identified SIPPs is much smaller (see Table 4 in Appendix for the statistics of the 3-host SODG and SIPPs in Fig 2). The SIPPs will include almost all the zero-day attack paths. The only possible way for a zero-day attack path to escape SIPPs is that it includes only zero-day exploits on all compromised hosts. This is very rare and unlikely, because it's almost impossible for attackers to exploit only zero-day vulnerabilities along the path. Therefore, a zero-day attack path will be a path in SIPPs if it exists. Section 3.5 will propose a method to help recognize highly suspicious candidate zero-day attack paths among the SIPPs.

3 System Design

3.1 System Overview

Fig. 3 shows the overview of our system. It consists of four components:

System call auditing and filtering. We first perform system call auditing on each host, and then send the system call traces from individual hosts to the analysis machine after filtering (according to filtering rules). Among the four components, only system call auditing and filtering is on the fly. The other three are performed off-line, to reduce overhead imposed on individual hosts.

SODG graph generation. To construct a network-wide SODG, two steps are needed: per-host SODG generation and inter-host SODG generation. First, the collected system call logs are parsed based on dependency rules to build per-host SODGs. Then, per-host SODGs are concatenated into a network-wide SODG.

SIPPs identification. To dig out the SIPPs "hidden" inside the network-wide SODG, trigger nodes are used as seeds to track the forward and backward OS

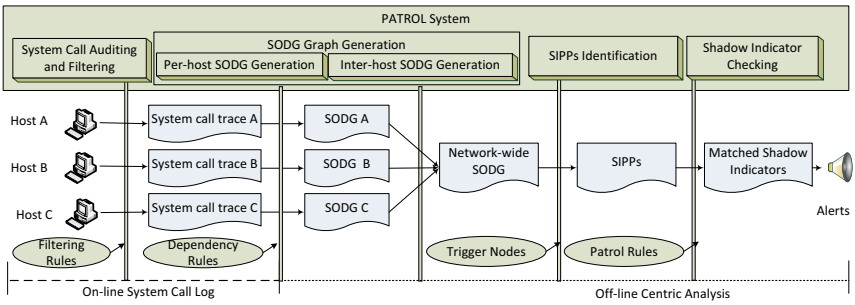


Fig. 3. System overview of Patrol

dependencies across the boundaries of individual hosts. These dependencies identify the nodes and edges of SIPPs.

Shadow indicator checking. To help identify highly suspicious candidate zero-day attack paths among the SIPPs, we also perform shadow indicator checking, which is a new technique that we will present in Section 3.5.

3.2 System Call Auditing and Filtering

Several requirements are expected for system call auditing: 1) System call auditing should be done against all running processes, rather than against specific processes. It's hard to pre-determine which process to audit, so process-specific system call auditing could miss important system calls that carry critical intrusion information. 2) System call auditing should be network-wide, meaning that: first, all hosts of the network should be audited; second, the socket communications between hosts should be captured. Network-wide system call auditing is the basis for identifying suspicious paths across hosts. 3) Sufficient OS-aware information should be preserved for accurate OS object identification. Due to the reuse of process ID and file descriptor numbers, it's inaccurate to identify system processes and files solely by their IDs or descriptor numbers. 4) The time that a system call is invoked and returned should be recorded. Time information can later help determine whether a system call is involved in intrusion propagation.

Considering unfiltered data would cause more bandwidth/CPU costs on data transfer and analysis, system calls are filtered before being sent to the analysis machine. Some filtering rules are applied to prune system calls which involve OS objects that are either highly redundant or possibly innocent. This is called *filtering preprocessing*, which can boost the speed of graph generation and reduce the complexity of resulted graphs. For example, we currently perform pruning for the following objects: 1) The dynamic linked library files like *libc.so.*$*$ and *libm.so.*$*$. They are loaded every time an executable is run, and thus cause a lot of redundancy; 2) Dummy objects like *stdin/stdout* and */dev/null*; 3) Objects about pseudo-terminal master and slave (*/dev/ptmx* and */dev/pts*); 4) Log relevant objects like *syslogd* and */var/log/*$*$; 5) Objects relevant with system maintenance (*apt-get* and *apt-config*). More filtering rules could be specified to

prune more system calls, gaining better speed boosting. However, it also takes more risk filtering out objects involved in vulnerability exploitations. Due to this tradeoff, filtering rules for preprocessing are enabled as options.

After filtering, system call traces are sent to the analysis machine. Considering accumulative data may cause bigger latency on data transfer and analysis, we set a parameter called *time window* to tune the frequency of sending system call logs. It is the periodic time span during which system calls are logged.

3.3 SODG Generation and Concatenation

System calls from individual hosts are used to construct per-host SODGs. A per-host SODG can be constructed by first parsing system calls into OS objects (process/file/socket) and dependency relations between them. OS objects then become SODG nodes and dependency relations become SODG edges. Dependency rules are proposed and used in related works [23–25] to help determine dependency relation types according to specific system calls. Table 1 lists the dependency rules used in Patrol. In addition to dependency rules, system call arguments also contribute to the parsing. They are used to uniquely recognize and name SODG nodes, and help infer the edge direction between them. For example, system call "*sys_open, start:470880, end:494338, pid:6707, pname:scp, pathname:/mnt/trojan, inode:9453574*" from our trace is transformed to (*6707, scp*)←(*/mnt/trojan, 9453574*), where *pid* and *pname* are used to recognize the process, and *pathname* and *inode* are used to identify the file.

Hosts communicate with each other, hence a per-host SODG may have directed edges to or from other per-host SODGs. This insight can be leveraged to build the network-wide SODG by concatenating per-host SODGs. If and only if there exists at least one directed edge between two nodes from two different per-host SODGs, these two SODGs can be concatenated together. Such edges can serve as the glue for concatenation. We find that directed edges between per-host SODGs are usually caused by socket-based communications. A local program can communicate with a remote program through message passing, which can be captured by system call *socketcall*. Hence, two per-host SODGs can be concatenated together by identifying and pairing socket objects. For example, system call "*sys_accept, start:681154, end:681162, pid:4935, pname:sshd, srcaddr:172.18.34.10, srcport:36036, sinkaddr:192.168.101.5, sinkport:22*" results in a directed edge (*172.18.34.10, 36036*)→(*192.168.101.5, 22*), where a socket object is denoted as a tuple (*ip*, *port*). This edge can be used to concatenate the per-host SODGs of *172.18.34.10* and *192.168.101.5*. The network-wide SODG is constructed by recursively concatenating the per-host SODGs. First, two per-host SODGs can be concatenated into a 2-host SODG. Then, the 3rd per-host SODG can be glued to the 2-host SODG, the 4th per-host SODG glued to the 3-host SODG, and so on. The algorithm goes on recursively and ends when no edge exists between any per-host SODG and the resulted network-wide SODG.

3.4 SIPPs Identification

SIPPs identification is designed to dig out SIPPs from the network-wide SODG. A benefit of the network-wide SODG is that intra-host forward and backward dependency tracking can be extended across the boundaries of individual hosts. Using trigger nodes as seeds, such inter-host dependency tracking identifies all network SODG objects that have direct or transitive dependency relations to or from trigger nodes, i.e. SIPPs by Definition 3. Hence, the SIPPs identification begins with the recognition of trigger nodes. Trigger nodes could be files that are deleted, added, or modified in unexpected ways, and processes that behave in an unusual or malicious manner, such as conducting abnormal port scanning, or making disallowed system calls. They are usually raised by security sensors like Snort, Tripwire, etc., and noticed by administrators.

Trigger nodes are not necessarily the start of an intrusion. For example, what an IDS detects could be later manifestation of the start. In that case, Patrol will use trigger nodes to first perform backward tracking to find the intrusion start, and then use the start to perform forward tracking. Basically, backward dependency tracking is used to identify all the SODG objects that have directly or transitively affected trigger nodes, and forward tracking is to identify objects that have been affected by trigger nodes. In patrol, backward and forward dependency tracking are both implemented based on breadth-first search (BFS) algorithm [26], as depicted in Definition 3. In simple words, the SIPPs is initialized to include only trigger nodes, and then BFS is recursively invoked to add new nodes and edges from the network-wide SODG. For each object obj' in SIPPs, the latest and earliest access time are respectively maintained in $lat(obj')$ and $eat(obj')$. In backward tracking, if obj' depends on another object obj in SODG, and the timestamp *start* of this dependency relation is earlier than $lat(obj')$, it means that obj has affected obj'. So, obj and the dependency relation should be added into SIPPs. Similarly, in forward tracking, if another object obj in SODG depends on obj', and the timestamp *end* of this dependency relation is later than $eat(obj')$, it means that obj has been affected by obj' and should be added into SIPPs together with the dependency relation.

3.5 Shadow Indicator Checking

The SIPPs could still be complex. To further identify highly suspicious candidate zero-day attack paths among the SIPPs, we propose the concepts of vulnerability shadow and shadow indicator. These concepts are based on the observation that vulnerabilities share some features. CWE [27] enumerates 693 common weaknesses, and CAPEC [28] classifies 400 common attack patterns. These common features could exist in vulnerabilities found in a long time span, and even in some future unknown vulnerabilities.

The concept of vulnerability shadow is much in the same spirit. But instead of directly characterizing vulnerabilities, we propose to characterize *exploitations* of them at the OS level. This is because, due to the existence of shared features, exploitations of some vulnerabilities often result in similar characteristics

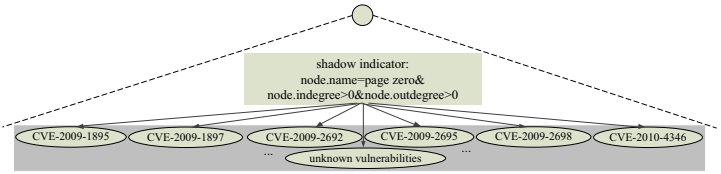


Fig. 4. A vulnerability shadow example: *bypassing mmap_min_addr*

in SODG. The insight here is that, the characteristics extracted from previous exploitations of known vulnerabilities can be applied to detect the exploitation of unknown vulnerabilities. We leverage this insight as follows: we define such common characteristics as an indicator function, which is used to indicate membership of elements in set theory, and use this function to build a set. The resulted set is a set of known and unknown vulnerabilities, whose exploitations all have the common characteristics. Such a set is named *vulnerability shadow*, and its set indicator function is called *shadow indicator*.

Definition 4. *Vulnerability Shadow* and *Shadow Indicator*
A vulnerability shadow is a Cantor set denoted as $S = \{v | p(SODG(v))\}$, where:

- v is a known or unknown vulnerability, whose exploitation is part of the SODG represented as $SODG(v)$;
- p, the shadow indicator for S, is a boolean-valued set indicator function: $SODG(v) \rightarrow \{\text{true, false}\}$. p can be a conjunction of several predicates, in a form like $p = p_1 \& p_2 \& \cdots \& p_n$ (n is a natural number), where for $\forall 1 \leq i \leq n$, p_i is predicating an attribute of a node or edge in $SODG(v)$, and & stands for *AND* operation in logic (p is true, iff p_i is true for $\forall 1 \leq i \leq n$);
- $v \in S$, iff $p(SODG(v)) = true$.

Fig. 4 shows an example vulnerability shadow *bypassing mmap_min_addr*, with $node.name = page\_zero \& node.indegree > 0 \& node.outdegree > 0$ as its shadow indicator.[2] This indicator was first observed in exploiting CVE-2009-1895 and CVE-2009-1897, and then can be used to recognize the exploitations of CVE-2009-2692, CVE-2009-2695, CVE-2009-2698, etc. A very intriguing implication of vulnerability shadow is that, unknown vulnerabilities that do not have a CVE ID yet could exist in this shadow, if and only if their exploitations can make the shadow indicator become true.

Shadow indicators imply occurrence of an exploitation and should not appear in legitimate paths. In addition to the trigger node, if other shadow indicators appear on a path in SIPPs, the path is very likely to be an attack path. If

[2] The kernel variable *mmap_min_addr* is tunable to specify the minimum virtual address that a process is allowed to *mmap*. *Bypassing mmap_min_addr* makes a violation to map user-land *page zero*, which can be triggered later by null pointer dereference to gain privileges. Page zero is parsed from *mmap2*(*null, 4096,* $*$*,* $*$*,* $*$*)=0* or *mprotect*(*0, 4096,* $*$*)=0*, where $*$ is the wildcard.

no alerts from vulnerability scanners or traditional IDS can be associated with any of these indicators, this path is reported as a highly suspicious candidate zero-day attack path. Rule-based checking is employed to recognize the shadow indicators in SIPPs. As Snort rules are developed for Snort to capture attack signature at packet level, Patrol rules are invented for Patrol to capture shadow indicators at OS level. A Patrol rule is like this: *indicator indicator_object* (*function: indicator_function; msg: "vulnerability_shadow_name"*).

Each rule specifies the object to check upon in *indicator_object*. If no object is specified, "any" is used to check on every object. Each rule contains the indicator function in *indicator_function*. The function specifies unexpected attribute values of the nodes or edges in SIPPs. A message will display the name of the vulnerability shadow when the function returns true. The following gives the Patrol rule for checking the shadow indicator of bypsssing mmap_min_addr: *indicator page_zero* (*function: indegree>0&outdegree>0; msg: "bypassing mmap_min_addr"*).

The attributes used to specify *indicator_function* include graph attributes and system call attributes. The graph attributes like *indegree* (a node's inward edge number) and *outdegree* (outward edge number) allow us to characterize exploitations from the perspective of graph. In addition to graph attributes, system call attributes such as *syscall* (system call name), *argument* (arguments) and *rtn* (return value) can also be taken into consideration. Patrol maintains association between graph edges and corresponding system calls. Hence, system calls can be revisited for inspection of its arguments and return values. For example, the following Patrol rule is used to detect symlink inconsistency: *indicator any* (*function: outdegree=0&∃(syscall=linkat&rtn=0); msg: "symlink inconsistency between request and creation"*).[3]

4 Implementation

The system design is implemented into a prototype named Patrol, through approximately 5493 lines of code, which include about 2411 lines of C code for a loadable kernel module auditing 39 system calls, and 3082 lines of gawk code for data analysis which produces dot-compatible [29] output for graph visualization.

System Call Auditing and OS-Aware Reconstruction. Patrol hooks system calls via a loadable kernel module, which can audit all running processes. Interested system calls are audited, including those encapsulated in system call *socketcall*, such as *sys_accept*, *sys_sendto*, etc. In the module, codes are inserted to each system call to 1) record its arguments and return values; 2) refer OS kernel data structures, retrieving process descriptor from *task_struct* and file descriptor from *files_struct*. The OS-aware information such as process descriptors, absolute file paths and inode numbers are preserved for accurate OS object identification. The timestamps *start* and *end* respectively record the time that the

[3] If a symbolic link created is inconsistent with the one requested, an attacker can exploit race condition to make arbitrary code executed as the requested link is referenced. Because *linkat* has other alternatives like *symlinkat*, *link*, and *symlink*, this rule has several siblings.

system call is invoked and returned. The resulted kernel module supports Linux kernel versions 2.6.24 through 2.6.32.

Graph Representation and Edge Aggregation. We represent our graphs with an adjacency matrix (*Map*) because during SODG generation and SIPPs identification we need to quickly look up if there is already an existing edge connecting two nodes. With adjacency matrix, the query takes only O(1) time, while with other data structures it may take O($|v|$) or O($|e|$) time, where $|v|$ and $|e|$ are respectively the number of nodes and edges in a graph. For each pair of SODG nodes (*srcObj* and *sinkObj*), there could be a large number of edges between them. The edges are caused by different system calls or the same system call with different timestamps. Our implementation aggregates them into a single one, maintaining the matrix cell (*Map*[*srcObj, sinkObj*]) to count the number of edges, and a timestamp list (*tMap*[*srcObj, sinkObj*]) to associate this aggregated edge with different timestamps.

Light Mode and Heavy Mode. To reduce dependency on efficiency of the traditional security sensors, Patrol implements another mode: heavy mode, in which Patrol feeds itself with its own alerts as seeds. In light mode, Patrol gets fed with trigger nodes, identifies SIPPs, and continues with rule-based checking against SIPPs to detect if shadow indicators exist. In heavy mode, it doesn't use any trigger nodes from other tools. Instead, it directly matches shadow indicators against the whole network-wide SODG. If any shadow indicators are matched, they are then used as trigger nodes to initiate the light-mode running. That is, a heavy mode can be run to replace the role of security sensors, but it also causes heavier workload. For example, the heavy mode can detect the brute-force attack exploiting CVE-2008-0166 in seconds after the SODG is built, without relying on any Snort alert. This paper focuses on illustration of light mode.

5 Evaluation

5.1 Experimental Setup

The ideal environment to evaluate Patrol is a real-world enterprise network. However, accesses to production kernels are tightly controlled by policy. We therefore built a web-shop test-bed for evaluation. Fig. 1 illustrates the test-bed network, which is set up with firewalls, Nessus [30], Oval [31], Snort, Wireshark [32], Ntop [33] and Tripwire. The hosts are typically deployed with Dell PowerEdge T310 with two 2.53GHz Intel(R) Xeon(R) X3440 quad-core processor and 4GB of RAM running 32-bit Linux 2.6.24 through 2.6.32.

We implemented the attack scenario in Fig. 1. In order to produce zero-day attack paths, the attacks have to exploit unknown vulnerabilities. However, a typical zero-day attack can remain undisclosed for 312 days on average [34]. Due to such lack of zero-day resources, we emulate unknown vulnerabilities by using published vulnerabilities. Our strategy is to tune the "time" back to a history date and assume vulnerabilities published after that date are still unknown.

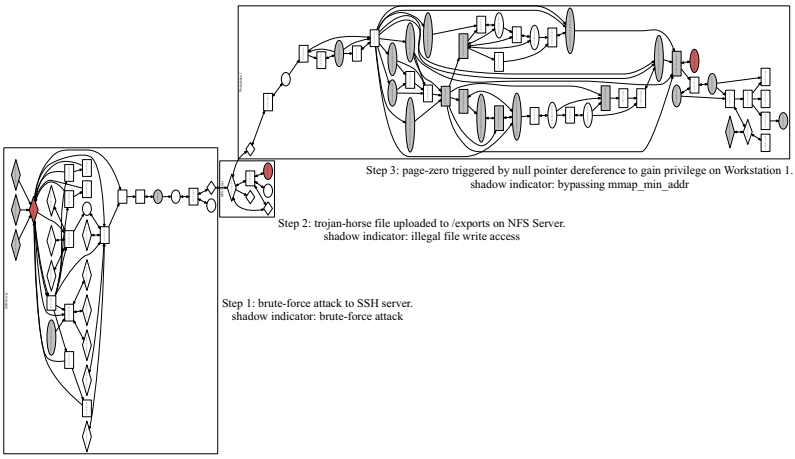


Fig. 5. The zero-day attack path *p1* dug out from the SIPPs (Fig. 2b) by Patrol, capturing the 3-step attack in the attack scenario. The identified shadow indictors are highlighted in red color. The grey nodes are proved to be malicious during verification.

Such emulation enables us to evaluate the correctness of our approach, because 1) timelines can be maintained for vulnerability shadows to make sure that no specific knowledge of the emulated vulnerabilities is needed; 2) the exploit code and other information about the emulated vulnerabilities can be available for verification. This paper assumes that the time is tuned to August 1, 2009, so that CVE-2008-0166 becomes the only known vulnerability in the attack scenario.

5.2 Correctness

Of all the vulnerabilities in the attack scenario, only the exploit of CVE-2008-0166 triggered an alert "SSH potential brute force attack" from Snort. Hence, both of the zero-day attack paths *p1* and *p2* in the attack scenario were missing. In contrast, using the SSH socket (node *225* in the figures) noticed from the Snort alert as the trigger node, Patrol successfully identified both *p1* and *p2* at the OS level. Fig. 5 and Fig. 7 respectively illustrate *p1*[4] and *p2*. Since *p2* and *p1* share the same Step 1 and Step 2, Fig. 7 only shows the Step 3 of *p2*.

We verified the correctness of *p1* and *p2*, by comparing the nodes and edges on them with the

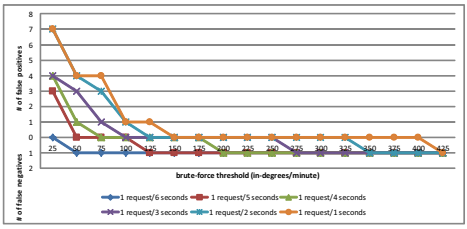


Fig. 6. False positives and negatives of shadow indicator checking for brute-force attack

[4] There were hundreds of socket communications coming from different ports of the same malicious IP (*192.168.202.2*) to node *225*. For simplicity, only three of them are illustrated.

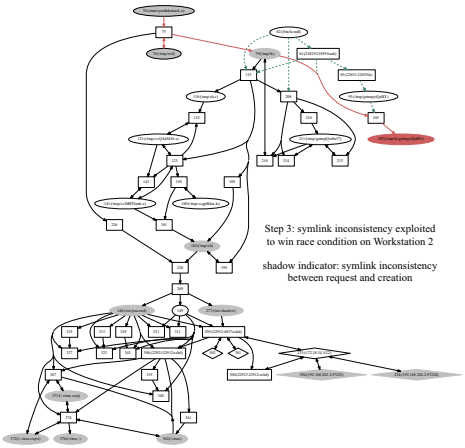


Fig. 7. Step 3 of the zero-day attack path *p2* identified by Patrol. The red and green dotted lines respectively denote the execution of the attack processes and innocent processes. The red lines replaced the requested symlink */tmp/ls* (*79*) with malicious code */tmp/evil* (*78*), which was later referenced by the the innocent process *ls* (*115*). The identified shadow indictor is highlighted in red color. The grey nodes are proved to be malicious during verification.

intrusion knowledge extracted from the exploit code, the CVE entries in NVD [35] and the documentation of corresponding vulnerable applications. We marked the nodes in Fig. 5 and Fig. 7 with grey color if they were verified to be malicious. It shows that Patrol correctly captured the malicious objects interacting with each other to accomplish the intrusion break-in and propagation.

We also evaluated the false positives of the shadow indicator checking on the two identified paths. For this, we kept Patrol running intensively for 72 hours against a variety of applications and services in the test-bed. It turns out that the false positive rate of shadow indicator checking is indicator-specific. For example, the indicator checking for *bypassing mmap_min_addr* and *symlink inconsistency* got 0 false positives, while the indicator checking for *brute-force attack* had false positives varying with the setup of a parameter and the workload of the host. The brute-force shadow indicator ($node.indegree > threshold_{bruteforce}$) uses $threshold_{bruteforce}$ to specify the maximum in-degree per minute allowed for a SODG node. Fig. 6 illustrates the impact of $threshold_{bruteforce}$ on false positives and negatives for SSH Server. As the threshold increases, the false positives first decrease and then stay at 0 until the false negative appears. As the request speed increases, the false positives increase and a bigger threshold is needed.

Furthermore, the above false positives can be tolerated by Patrol to some extent. For example, for brute-force shadow indicator checking, the false alarmed objects include: 1) DNS related process (avahi-daemon) or sockets (port 53 or 5353); 2) uninitialized sockets (port 0); 3) dynamic linked library files. However, none of them were on the same SIPPs with other shadow indicators. Hence, with the help of SIPPs, most of the false positives could be eliminated.

5.3 Efficiency

Time window size and filtering preprocessing are two important factors impacting the efficiency of Patrol data analysis. Time window is the periodic time span

Table 2. Statistics for time window based tests on data analysis

time window size (mins)	5	15	25	35	45	55
# of syscalls in filtered log	17550	52649	87748	122848	157947	193047
time overhead of SODG generation (s)	44.94	108.88	278.52	601.93	1097.33	1836.47
# of objects in SODG	526	1425	2326	3227	4101	4977
time overhead of SIPPs identification (s)	0.91	7.54	22.75	48.79	76.75	107.02
# of objects on SIPPs	374	1094	1811	2519	3209	3903
time overhead of indicator checking (s)	0.004	7.033	18.149	31.159	60.932	76.497
total overhead (s)	52.26	142.30	350.90	726.58	1291.95	2089.31
syscall generation speed (KB/s)	1.02	1.027	1.034	1.033	1.032	1.034
data analysis speed (KB/s)	5.839	6.498	4.418	2.985	2.157	1.634
storage size (raw)(MB)	2.731	8.189	13.65	19.107	24.568	30.026
storage size (compressed)(MB)	0.298	0.903	1.514	2.118	2.722	3.333

during which system calls are collected and analyzed. All the evaluation results in this subsection use the arithmetic mean averaging over 10 runs of tests.

Impact of Time Window Size. We set the time window size to values from 5 mins to 55 mins. Table 2 illustrates the statistics of Patrol data analysis for SSH server. To get overhead under heavy workload, requests were loaded to SSH Server at the speed of 1 request per 5 seconds. Data analysis spends time mainly on SODG generation, SIPPs identification and shadow indicator checking. Fig. 8 plots the time overheads. The results show that SODG generation dominates the time overhead, and its computation cost increases approximately quadratic with the time window size. The time overheads of SIPPs identification and shadow indicator checking tend to be linear and relatively much smaller. Fig. 9 shows that the speed of Patrol data analysis is maximized when time window size is *15 mins*. This speed is far beyond the system call generation. We also noticed that the caused latency is about *2.37 mins*, and the storage requirement is about *0.085 GB/day*. Today's hard disk is large enough to accommodate this substantial amount of log traffic. Considering the test is done in quite request-intensive workload, both the time and storage overheads are reasonable. We therefore determine the time window size for the test-bed network to be 15 mins.

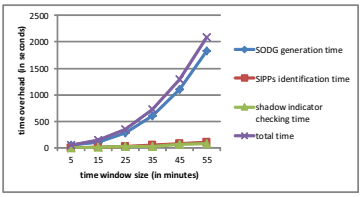


Fig. 8. Time overhead of Patrol data analysis varying with time window size

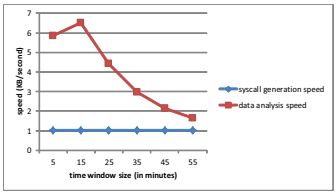


Fig. 9. Patrol data analysis speed vs. system call generation speed

The above results are theoretically supported. SODG generation checks each existing object to avoid duplication before adding new objects. Hence, the computational complexity of SODG generation can be $O(|v|^2)$. The SIPPs identification is using the BFS algorithm, thus its time complexity is $O(|v| + |e|)$ [26],

Table 3. Comparison results between filtered and unfiltered data analysis

	SSH Server		NFS Server		Workstation 1	
filtered/unfiltered	filtered	unfiltered	filtered	unfiltered	filtered	unfiltered
# of syscalls in log	22249	82133	11761	14944	21722	46043
time overhead on SODG(s)	58.38	1812.966	42.286	48.447	51.012	101.138
# of objects	650	15960	34	210	604	1007
# of processes	230	273	7	121	106	138
# of files	248	15515	17	79	473	844
# of sockets	171	171	10	10	23	23
# of dependencies	18697	97805	11813	15056	19649	43712

where $|v|$ and $|e|$ are respectively the number of nodes and edges in SODG. The shadow indicator checking checks each object and dependency of SIPPs in worst case, therefore its complexity is also $O(|v'| + |e'|)$, where $|v'|$ and $|e'|$ are respectively the number of nodes and edges in SIPPs.

Impact of Filtering Preprocessing. Table 3 summarizes the SODG generation time with filtering enabled and disabled respectively. The results show that unfiltered data costs more time than filtered data. The worst case overhead is the unfiltered SODG generation for SSH Server. It spent about half an hour. The large overhead is mainly because the algorithm checks each existing object to avoid duplication before adding new objects. When the system object number reaches very high, such as *15960* in this case, the time cost rises very quickly. We also noticed that among these objects, the number of files is extremely large as *15515*. The filtered SODG generation costs less than one minute because a large number of these files are effectively pruned by filtering rules.

5.4 Performance Overhead

We use LMBench [36] to measure the performance impact of Patrol on individual core kernel system calls. The outputs show that the addon overhead of most modified system calls in Patrol is within 10%. Some of them are even working with negligible overhead, such as *sys_read* and *sys_write*. The worst case overhead is 52.7% for *sys_stat* and 175% for *sys_fstat*. These results are to be expected, because of the relatively small amount of work done in each call compared to the work of recording OS-aware object information. For example, 175% is larger than 52.7% because of the smaller denominator, but in both cases the imposed overhead was equally 0.3-0.4 microseconds. The common case is much better.

We use UnixBench to measure the slow-down of the whole system that orchestrates the above individual system calls together. The outputs show that the performance overhead of Patrol is 20.8% for the whole system, with larger overhead to I/O-intensive applications than CPU-intensive applications. We also use kernel decompression and kernel compilation to measure the system performance of Patrol in intensive workload. The results show that the two workloads impose 15.93% overhead and 20.34% overhead on the system.

5.5 Scalability

Regarding the scalability, let's consider the main overhead imposed on bandwidth, SODG generation and SIPPs identification for an enterprise network equipped with 10000 hosts, 10 GB/s network bandwidth and a HPC cluster of 640 processor cores (20 processors with 32 cores per processor).

With converging traffic from hosts to the cluster, the bandwidth cost will be about 10000 times the system call generation speed for each host. Taking the speed 1.027 KB/s from Table 2, the *bandwidth overhead* is about 10.029 MB/s which only occupies less than 1% of total bandwidth.

The SODG generation costs time mainly on per-host SODG generation which is a parallelizable task (α=0 in Equation 1). Given the data collected in 1 time window, the *SODG generation time* for 10000 hosts is estimated to be 28.35 minutes according to the following Gustafson's law, taking single-host SODG generation overhead as 108.88 seconds from Table 2.

$$\frac{t_1}{t_p} = p - \alpha(p-1) = \alpha + p(1-\alpha) \tag{1}$$

where p is the number of processors for parallel computing, α is the fraction of running time a program spends on non-parallelizable parts, t_1 is the execution time of the sequential algorithm, and t_p is the execution time with maximum speed-up under parallelization of the program.

SIPPs identification from a trigger node is non-parallelizable (α=1) due to the sequential nature of dependency tracking. Hence, the SIPPs identification time increases linearly with the host-length of SIPPs (l). Its maximum can be estimated by constructing service dependency transitive closure ("host A can reach host B through one or more service dependencies") in enterprise network. Let's suppose l=100, and the *SIPPs identification time* will be about 12.57 minutes, taking time overhead of single-host SIPPs identification as 7.54 seconds from Table 2. SIPPs identification from different trigger nodes and branching in SIPPs identification can be done in parallel. As long as the number of trigger nodes and branches don't exceed p, SIPPs identification can be easily handled within 12.57 minutes. We make conservative estimation by α=1, hence the efficiency for parallel computing can be better in reality than estimated.

6 Related Work

Patrol draws inspirations from previous research such as system call-based intrusion detection and system object dependency tracking.

System calls are used in pioneer works by Forrest et al. [11] and Lee et al. [12] for intrusion detection. System call-based IDS mainly leverages statistical properties of system call sequence [13] [14] and system call arguments [16] [17]. Bhatkar et al. further takes into account the temporal properties involving arguments of different system calls [18]. Instead of providing individual intrusion

alerts, the aim of Patrol is to identify zero-day attack paths through network-wide dependencies parsed from system calls. These paths provide network-wide attack context, and help detect unknown vulnerability exploitations.

System object dependency tracking is first proposed by King et al. [23] to automatically identify sequences of intrusion steps. The follow-up works [37] [38] further propose to integrate system object dependency tracking and alert correlation techniques. Given a large number of existing IDS alerts, these works target on identifying their correlations. In contrast, Patrol takes an inverse strategy to first identify SIPPs hidden in the network-wide SODG, and then recognize unknown vulnerability exploitations on these paths.

7 Discussion and Conclusion

In addition to the promising potentials, the current version of Patrol may face challenges such as 1) If an attack path goes through a victim machine hosting kernel mode service like nfs-kernel-server, Patrol may lose trace halfway since it relies on system call interface; 2) If an attack is a long-term attack, Patrol may successfully capture its intrusion propagation paths at different time spans, but fail to correlate them.

In conclusion, this paper identifies the problem of zero-day attack paths in practical network defense. This paper proposes a prototype system named Patrol. By building a network-wide system object dependency graph, identifying suspicious intrusion propagation paths in it, and recognizing shadow indicators on these paths, Patrol can dig out the zero-day attack paths at runtime.

Acknowledgments. We want to thank the anonymous reviewers for their valuable and helpful comments. This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, AFOSR W911NF1210055, and ARO MURI project "Adversarial and Uncertain Reasoning for Adaptive Cyber Defense: Building the Scientific Foundation".

References

1. Sheyner, O., Haines, J., Jha, S.: Automated generation and analysis of attack graphs. IEEE Oakland (2002)
2. Jajodia, S., Noel, S., O'Berry, B.: Topological analysis of network attack vulnerability. Managing Cyber Threats: Issues, Approaches and Challanges (2003)
3. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: A logic-based network security analyzer. In: USENIX Security (2005)
4. Wang, L., Jajodia, S., Singhal, A., Cheng, P., Noel, S.: k-Zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. In: TDSC (2013)
5. Albanese, M., Jajodia, S., Singhal, A., Wang, L.: An efficient approach to assessing the risk of zero-day vulnerabilities. In: SECRYPT (2013)
6. Long, J.: Google Hacking for Penetration Testers. Syngress (2007)
7. McClure, S.: Hacking Exposed: Network Security Secrets and Solutions. McGraw-Hill (2009)
8. Network Penetration Testing. MosaicSecurity.com. `https://mosaicsecurity.com/categories`

9. Debar, H., Wespi, A.: Aggregation and correlation of intrusion-detection alerts. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 85–103. Springer, Heidelberg (2001)
10. Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 54–68. Springer, Heidelberg (2001)
11. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. IEEE Oakland (1996)
12. Lee, W., Stolfo, S.J., Chan, P.K.: Learning patterns from unix process execution traces for intrusion detection. In: AI Approaches to Fraud Detection and Risk Management (1997)
13. Kosoresow, A.P., Hofmeyer, S.A.: Intrusion detection via system call traces. IEEE Software (1997)
14. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security (1998)
15. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. IEEE Oakland (2001)
16. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
17. Tandon, G., Chan, P.: Learning rules from system call arguments and sequences for anomaly detection. In: ICDM DMSEC (2003)
18. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. IEEE Oakland (2006)
19. Sekar, R., Gupta, A., Frullo, J., Shanbhag, T.: Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions. In: ACM CCS (2002)
20. Ko, C., Ruschitzka, M., Levitt, K.: Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. IEEE Oakland (1997)
21. Snort. Sourcefire, Inc., `http://www.snort.org`
22. Tripwire. Tripwire, Inc., `http://www.tripwire.com`
23. King, S.T., Chen, P.M.: Backtracking intrusions. In: ACM SOSP (2003)
24. Xiong, X., Jia, X., Liu, P.: Shelf: Preserving business continuity and availability in an intrusion recovery system. In: ACSAC (2009)
25. Goel, A., Po, K., Farhadi, K., Li, Z., de Lara, E.: The taser intrusion recovery system. In: ACM SOSP (2005)
26. Knuth, D.E.: The Art Of Computer Programming (1997)
27. CWE. MITRE, `http://cwe.mitre.org`
28. CAPEC. MITRE, `http://capec.mitre.org`
29. Graphviz, `http://www.graphviz.org`
30. Nessus. Tenable Network Security, `http://www.tenable.com`
31. Oval. MITRE, `http://oval.mitre.org`
32. Wireshark. Wireshark Foundation, `http://www.wireshark.org`
33. Ntop, `http://www.ntop.org`
34. Bilge, L., Dumitras, T.: An Empirical Study of Zero-Day Attacks In The Real World. In: ACM CCS (2012)
35. NVD. MITRE, `http://nvd.nist.gov`
36. McVoy, L.W., Staelin, C.: lmbench: Portable Tools for Performance Analysis. In: USENIX (1996)
37. King, S.T., Mao, Z.M., Lucchetti, D.G., Chen, P.M.: Enriching intrusion alerts through multi-host causality. In: NDSS (2005)
38. Zhai, Y., Ning, P., Xu, J.: Integrating IDS alert correlation and OS-Level dependency tracking. In: IEEE Intelligence and Security Informatics (2006)

Appendix

Table 4. Statistics for the 3-host SODG and SIPPs in Fig 2

metrics	SSH Server	NFS Server	Workstation 1
time window size (in minutes)	15	15	15
# of syscalls in unfiltered log	82133	14944	46043
# of syscalls in filtered log	22249	11761	21722
growth rate of compressed syscall log (GB/day)	0.126	0.019	0.065
# of objects in graph	650	34	604
# of processes in graph	230	7	106
# of files in graph	248	17	473
# of sockets in graph	171	10	23
# of dependencies in graph	18697	11813	19649
# of inter-host dependencies from last host in graph	50	11	1
# of inter-host dependencies to next host in graph	1	11	0
average indegree/outdegree in graph	29	347	33
max indegree in graph	8640	8478	12909
object index of max indegree in graph	543	661	1123
max outdegree in graph	9908	8294	12784
object index of max outdegree in graph	225	663	1153
# of objects in SIPPs	26	6	143
# of processes in SIPPs	8	1	62
# of files in SIPPs	3	2	75
# of sockets in SIPPs	15	3	5
# of dependencies in SIPPs	8905	11664	4059
# of inter-host dependencies from last host in SIPPs	14	1	1
# of inter-host dependencies to next host in SIPPs	1	8	0
average indegree/outdegree in SIPPs	343	1944	28
max indegree in SIPPs	8581	8442	410
object index of max indegree in SIPPs	543	661	808
max outdegree in SIPPs	8686	8280	2373
object index of max outdegree in SIPPs	225	663	783