Discover and Tame Long-running Idling Processes in Enterprise Systems

Jun Wang[1], Zhiyun Qain[2], Zhichun Li[3], Zhenyu Wu[3], Junghwan Rhee[3], Xia Ning[4], Peng Liu[1], Guofei Jiang[3]
[1]Penn State University, [2]University of California, Riverside, [3]NEC Labs America, Inc. [4]IUPUI
[1]{jow5222, pliu}@ist.psu.edu, [2]zhiyunq@cs.ucr.edu, [3]{zhichun, adamwu, rhee, gfj}@nec-labs.com, [4]xning@cs.iupui.edu

ABSTRACT

Reducing attack surface is an effective preventive measure to strengthen security in large systems. However, it is challenging to apply this idea in an enterprise environment where systems are complex and evolving over time. In this paper, we empirically analyze and measure a real enterprise to identify unused services that expose attack surface. Interestingly, such unused services are known to exist and summarized by security best practices, yet such solutions require significant manual effort.

We propose an automated approach to accurately detect the idling (most likely unused) services that are in either blocked or bookkeeping states. The idea is to identify repeating events with perfect time alignment, which is the indication of being idling. We implement this idea by developing a novel statistical algorithm based on autocorrelation with time information incorporated. From our measurement results, we find that 88.5% of the detected idling services can be constrained with a simple syscall-based policy, which confines the process behaviors within its bookkeeping states. In addition, working with two IT departments (one of which is a cross validation), we receive positive feedbacks which show that about 30.6% of such services can be safely disabled or uninstalled directly. In the future, the IT department plan to incorporate the results to build a "smaller" OS installation image. Finally, we believe our measurement results raise the awareness of the potential security risks of idling services.

Categories and Subject Descriptors

K.6 [**Management of computing and information systems**]: Security and Protection

Keywords

Idling service detection; Attack surface reduction; Enterprise systems; Autocorrelation

1. INTRODUCTION

Managing the security of enterprise systems is always a challenging task, because the overall security of the entire enterprise is usually determined by the weakest link and today's enterprise systems are so complex that it is hard to understand which program/process can be the weakest links. Many security breaches start with the compromise of processes running on an inconspicuous workstation. Therefore, it is beneficial to turn off unused services to reduce their corresponding attack surface. Anecdotally, many security best practice guidelines [12, 9] suggest system administrators to reduce attack surface by disabling unused services. However, such knowledge needs to be constantly updated and it is unfortunate that no formal approach has been studied to automatically identify such services.

Prior research has mostly focused on the line of anomaly detection to learn the normal behavior of processes and then constrain them, *e.g.,* by limiting accessible IP/port ranges [32, 31, 33] or the system calls that are allowed to be made [20, 29, 22, 18].

While such approaches may detect abnormal behaviors of a process (*e.g.,* possibly attacks), they belong to reactive approaches rather than preventive ones. Obviously, reducing attack surface is a preventive measure that complements anomaly detection. The strong semantic of idling services enables more specific treatments (*e.g.,* disable or even uninstall them) as opposed to the general treatments to anomalies, *e.g.,* raising a security alert. Unfortunately, the frequent false alerts in anomaly detection make it rarely deployed in practice. Identifying idling services, on the other hand, represent a much more cost effective solution where each service only needs to be examined mostly once (*e.g.,* to determine if it can be safely uninstalled). Once a positive decision is made, it completely eliminates the entire attack surface of a process before an attack could ever happen. Unfortunately, due to lack of formal methods, in state-of-the-art practice, system administrators today could only depend on practical wisdoms from Internet forums [10], and their own experience.

One might argue that system administrators or end-users should setup their servers and desktop machines to only run useful services and applications in the first place. However, in practice, this is a daunting task. The modern OS vendors tend to stuff more and more functionalities into their distributions, and provide a "one size fits all" system. Thus, many services might not be useful for a particular user. Even worse, today's enterprises typically run a variety of OS distributions. As a result, it is not easy for system administrators to keep up with the purpose of each and every service. Furthermore, systems keep evolving and users' demands keep changing. Given the complex dependencies inside enterprise, people tend to keep things as they are. Such practice, therefore, negligibly leaves numerous idling services running, and thus exposes unnecessary attack surface.

ASIA CCS'15, April 14–17, 2015, Singapore..

http://dx.doi.org/10.1145/2714576.2714613.

In this paper, we propose an automated method to detect "idle/idling" services and also provide ways to reduce or minimize their attack surface. An idling service is a service that does not serve real workload, but rather running in a blocked or bookkeeping state. A simple example is that a long-running service may listen on a certain port, while in reality no one has connected, or will connect to it. This service thus should be disabled to eliminate possible remote exploits. In our study, we find 25% of long-running processes are idling services.

In general, there are two types of idling services. The first type is easy to detect – the service process is completely dormant, blocked waiting for events that would never arrive. For the second type, the service processes regularly wake up from blocked state and perform some "bookkeeping" tasks – simple operations not relevant to the service they provide. The second type is more difficult to identify, since the bookkeeping operations could make these processes appear as active and truly performing services. In order to address the problem of identifying both types of idling services, we propose LIPdetect (**L**ong-running **I**dling **P**rocess), an effective algorithm that is able to differentiate "bookkeeping" activities from the ones resulted from real workload.

While LIPdetect could identify and report idling services to system administrators, working with IT department of our company we find in some cases the system administrators are not 100% sure whether certain services can be disabled without negative consequences. It is therefore desirable to keep those services running while still reducing their attack surface. In such cases, we fall back to the traditional wisdom to constrain the runtime behavior of a process. To this end, we design a simple tool named procZip that constrains the idling service's execution to previous known states. Combining idling service detection (LIPdetect) and automated process constraining (procZip), our system is therefore called LIPzip.

We evaluate the results of idling service detection, and perform a measurement study in an enterprise with both desktop and server machines to show how much attack surface can be possibly reduced. Meanwhile, we work with the IT department personnels to confirm our results. We also cross validate our results in a university department.

In summary, our paper has the following contributions:

- We propose an approach to identifying the long-running "idling" services and show that we can accurately identify such processes (with less than 1.5% mis-classified).
- We deploy our solution in a real enterprise environment on 24 hosts. We find that 25% of the long-running processes are idling on average per host and they constitute about 66.7% of attack surface of all long-running processes. 51.1% of the 92 unique binaries have had publicly known vulnerabilities.
- In collaboration with on-site system administrators and owners of the hosts, we analyze 434 idling processes in detail and classify 30.6% of such services can be disabled to fully eliminate the attack surface. We categorize the common reasons why they can be safely disabled or uninstalled, and show that such results can be shared across enterprises to make recommendations.
- We design procZip and show that even though the system administrators do not have the confidence to disable all services discovered by us, 88.5% of the idling services can be safely constrained by the procZip during a two-week evaluation, indicating that they are continuing to be idling.

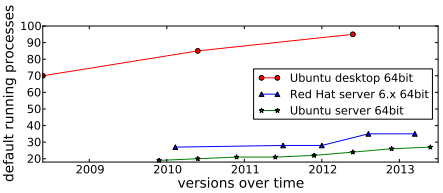


Figure 1: The number of default running processes

2. MOTIVATION AND OVERVIEW

Our motivation comes from the following observations:

1) Today's computer systems are equipped with an increasing number of functionalities (and thus also the attack surface) with newly developed technologies, perhaps much more than necessary for everyday use.

2) Systems are constantly evolving; more applications or features are installed (or updated) over time but they are left there even when no longer needed.

3) What makes the problem worse, in the Linux world, there are a large number of different OS distributions which make the IT systems largely heterogeneous and fragmented.

As a result, it is highly non-trivial to control or even understand all the systems precisely, let alone knowing whether a process is actually performing useful work or simply "idling". According to the IT department in our enterprise, they indeed do not keep track of what Linux distributions or applications are installed, the reason being that they do not want to get in the way of users doing what they want to do.

2.1 Measurement Study

To understand to what extent each observation is true, we conduct the following measurements based on two data sources. First, we look into the historical Linux distributions of Ubuntu and Red Hat from public images that are available on the Amazon EC2 service. Second, we collect information about 67 hosts in a real enterprise.

Default long-running processes. We look into the two popular Linux distributions Ubuntu and Red Hat over the past six years and count the number of daemon processes that are running by default after the system boots. Figure 1 shows that the number is steadily increasing over years. Note that the number from the server distributions is not very high due to the fact that these images are tailored to run on servers and thus many desktop features are not installed by default. In general, the increase of the default processes might render more and more idling processes.

User-installed long-running processes. We compare the default running processes with a snapshot of long-running processes in our own enterprise environment. We look at 13 Ubuntu 12.04 desktop machines and discover that on average 30 long-running processes on each host are not default running processes of the installation image, indicating that our employees tend to install many new services over time and the system size has been considerably growing.

Heterogeneity. Inside our enterprise, we study 67 Linux hosts including both desktops and servers. There are 7 Fedora, 11 CentOS, and 49 Ubuntu hosts respectively. These hosts include 3 unique Fedora versions, 6 unique CentOS versions, and 7 unique Ubuntu versions. This demonstrates the universal heterogeneity of Linux distributions. Indeed,

the management complexity rooted from the heterogeneity is partly echoed by users asking questions on public forums about what a specific process is for and why it is running [13, 7, 14].

In sum, all these evidences imply that there are potentially many idling services in average enterprise systems.

2.2 Problem and Solution

Problem statement. The key problem we are addressing is how to reduce the attack surface incurred by long-running daemon processes while minimizing the impact on the normal operations. The first challenge is the lack of a common definition on what types of processes can be safely disabled or uninstalled.

Long-running Idling process. Our definition of long-running idling processes comes from the following intuition. As shown in a recent study [24], long-running processes are typically structured as a giant loop, taking certain input in the beginning of the loop and performing different operations based on the input. Such a long-running process will be in one of the three states: (1) The process shows no program activity in terms of instructions or system calls, *e.g.,* blocked on certain system calls waiting for input. (2) The process repeatedly cycles through a loop to check certain input channels, *e.g.,* polling on sockets or checking the existence of a file. However, the input they receive has low entropy (same repeated input) and does not trigger any (useful) operation or they may not even receive any input at all (*e.g.,* the poll on sockets always timeout). As stated in §3, the challenge here is that such repeating events need to happen with perfect time alignment in order to be safely considered idling. (3) The process is taking input and perform useful operations based on the input.

In this work, we define the idling processes as processes in state (1) or (2). Our insight is that if we can identify a list of idling processes with decent accuracy, it is very likely to have little impact on normal operations if we disable them.

Workflow. Our system operates as shown in Figure 2. First, we collect data from hosts through monitoring agents that we developed. The details about the agent is described in §4. Then the data are taken as input to detect idling processes (details described in §3). Then we produce the list of idling processes and characterize their attack surface (explained in the following paragraphs). Next, we subject the process with attack surface[1] to two possible security actions.

Action 1: system administrators are usually in the best position to examine the list of idling processes and decide if they are needed. As shown in §7, we argue that such knowledge is required to determine the future necessity of a process with high confidence. In our evaluation, we show that administrators in our enterprise confirm many of the idling processes or services can indeed be disabled or uninstalled without interrupting normal operations. We further correlate the results with an IT group in a university department and show that the results can be shared effectively across organizations.

Action 2: similar to anomaly detection techniques, an alternative action is to "quarantine" or constrain their executions within the behaviors observed in the past (*e.g.,* checking the presence of a file). In this way, the process can still continue to receive any potential triggering events. When the triggering events do arrive in the future, we allow the process to be "freed" as long as the privileged user chooses to, *e.g.,* entering the root credentials or confirming a link of an email sent to a pre-registered address. Here, the idea is to reduce the attack surface of processes but still promptly allowing them to continue execution when they really need to. Note that the strategy is flawed if we simply un-quarantine the process whenever we see that the process is no longer idling as it could be possibly triggered by an attack that is actively trying to exploit a vulnerability of this process. Even though this action is operationally similar to anomaly detection in general, we point out that existing models from anomaly detection, *e.g.,* system call sequence or Finite State Automata (FSA), do not suit our problem (described in §3 and §8). So we need to develop new models and methods.

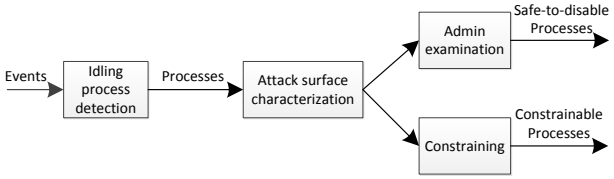


Figure 2: System workflow

Ideally, Action 1 is generally preferred since it can completely remove the attack surface. In practice, Action 2 can be used as a backup for usability purpose.

Attack surface. Similar to a previous work [32], we only consider system resources with "privilege discrepancy" such as files and sockets as attack surface. For instance, if a process is reading a file that is owned by the same user who started the process, we do not consider it an attack surface. As §6.2 shows, 29.3% of the idling processes we discover have direct attack surface exposed (*e.g.,* listening sockets or reading a file owned by a different user). More details in determining the attack surface is presented in §6 and Appendix. Based on the kind of attack surface they expose, we can prioritize them for system administrators. Note that the attack surface we compute is only the lower bound. In fact, many of them may expose additional attack surface once becoming active. As a result, the percentage of idling processes that have attack surface in idling state is only a conservative security metric.

Goals and non-goals. The goal of this paper is to investigate the security impact of idling processes and propose automated methods to identify them. We argue that the knowledge of the existence of such idling processes needs to be made more readily available to the community.

Non-goals are:

- To develop counter-measures against specific types of attacks. Rather, based on well-understood metrics of attack surface, our system is to reduce the likelihood of successful attacks.
- To come up with another anomaly detection algorithm. Instead, since the semantic of "idling processes" is much stronger than the abstract "normal behavior" in anomaly detection, we can apply more specific treatment to disable or uninstall such idling services. Even if we do constrain them still, there is a much less false alert rate.
- To design a perfect metric to compute the security risks or attack surface of a process. Instead, we leverage known metrics to achieve our goal.

[1]We are focusing on security and hence interested in attack surface. Different prioritizing metrics are also possible (*e.g.,* physical memory consumption).

3. DISCOVERY OF IDLING PROCESSES

As stated in §2.2, we are targeting at two categories of long-running processes. It is trivial to detect the **Category I (Completely Idle)** processes. As long as the process does not have any CPU time (available in the `procfs`) for longer than a significant time period (*e.g.,* one week) they are considered category I. This type of processes are usually blocked on certain blocking system calls and does not get a chance to return. However, it is much more subtle to detect the **Category II (Periodic/Repeating)**.

The rationale behind our method to detect category II processes is that an active process typically interacts with users and/or external inputs which should have non-trivial dynamics in size, inter-arrival time of requests, or connection timeout, *etc.* If a process repeats regularly over days and nights, it is highly likely that this process does not involve human-driven workflow, and thus can be considered idling. One might argue that batch jobs (*e.g.,* cron jobs) may also repeat daily or weekly, but they should not be considered idling. We would like to clarify that we consider repeating patterns at the system call level, including parameters and return values, instead of the "job" level. So even for a repeating cron job, *e.g.,* data backup, running daily could see different sets of files every time and therefore generate different sequence of system calls (and different parameters).

Possible solution from existing process behavior models. As described in §8, there are several mainstream process behavior models proposed before. Technically, we consider our problem a special case to model certain aspects of a process. We investigate if existing models are suitable to solve our problem. Not to disrupt the flow of the paper much, we present such discussion in more details in §8. In short, existing models do not fit the idling process detection problem due to two main reasons: 1) Many approaches model the behavior of a process based on a short history of events. For instance, the n-gram-based model only considers the n consecutive events [20]. The Finite State Automata (FSA) models only constrain the next event based on the current state. 2) While the sequence of events have been modeled in several ways, the time information such as the arrival time of events has not been fully considered. We should be able to analyze periodic behavior which is composed of a collection of diverse arrival time of events to accurately characterize idleness.

We note that the timing of events can be critical. For example, a busy web server may run in a loop and serve the same page and carry out the same set of operations repetitively. Therefore, the sequence of events may be perfectly repeating without considering the event timing. Hence, to address our problem, one has to have a careful treatment of timing, which is typically not entailed in anomaly detection.

3.1 Periodicity Detection: Background and Challenges

To find repeated patterns from sequence data is a common problem in many research areas such as data mining and signal processing [17]. A popular way to find the period from data sequences is via Fast Fourier Transform (FFT). However, only when the sequence is known as periodic as a priori, FFT can give the period. FFT is not able to detect whether a sequence is periodic or not.

In LIPdetect, we adopt the idea of autocorrelation to detect periodicity. Autocorrelation measures the correlation relation between a signal and itself after shift with a given lag [17]. For a sequence $S = \{X_1, X_2, \cdots, X_n\}$ and a lag k ($0 \leq k < n$), autocorrelation of S with lag k is calculated between sequence $\{X_{k+1}, X_{k+2}, \cdots, X_n\}$ and $\{X_1, X_2, \cdots, X_{n-k-1}, X_{n-k}\}$ using the following formula,

$$r(k) = \frac{1}{(n-k)\sigma^2} \sum_{t=1}^{n-k} (X_t - \mu)(X_{t+k} - \mu), \qquad (1)$$

where μ and σ^2 are the mean and variance of the sequence S. The value of autocorrelation falls in $[-1, 1]$, where larger absolute values mean stronger correlations and the positive (negative) sign means positively (negatively) correlated.

If a sequence is perfectly periodic with period p, then the autocorrelations of the sequence will be maximized (*i.e.,* autocorrelation equals to 1) when the lag k is a multiple of p, that is, $k = mp$, where m is an integer ($m = 1, 2, 3, \cdots$). Due to this property, autocorrelation can be used to detect periodicity of sequences. Curious readers can refer to Figure 3 to get an early sense of how autocorrelation looks like.

However, autocorrelation cannot be directly applied in our case to detect periodicity of a process. The primary reason is that as in Equation 1, X_t's are numerical values and thus their mean and variance, as well as the multiplication and addition operations on the values, are well defined. However, in event sequences, X_t's are categorical values (*i.e.,* event types) and all the operations as in Equation 1 are not well defined on categorical values (e.g., it does not make sense to add event type 1 and event type 2 to get event type 3). Another issue with applying autocorrelation on event sequences is that autocorrelation cannot capture the time information between two events. In a process, different system calls may introduce different intermediate idle time intervals, and the length of the time intervals carries useful information in inferring process periodicity. Autocorrelation has no mechanism to encode such timing information.

3.2 Periodicity Detection on Event Data

We propose a new autocorrelation measure, denoted as E-Autocorrelation, particularly for event data, which have categorical values and varying time gap between events.

3.2.1 E-Autocorrelation

Correlation measure. To handle categorical value, we changed the correlation measure from Equation 1 to:

$$r_e(k) = \frac{1}{n-k} \sum_{t=1}^{n-k} \mathbb{I}(X_t, X_{t+k}), \qquad (2)$$

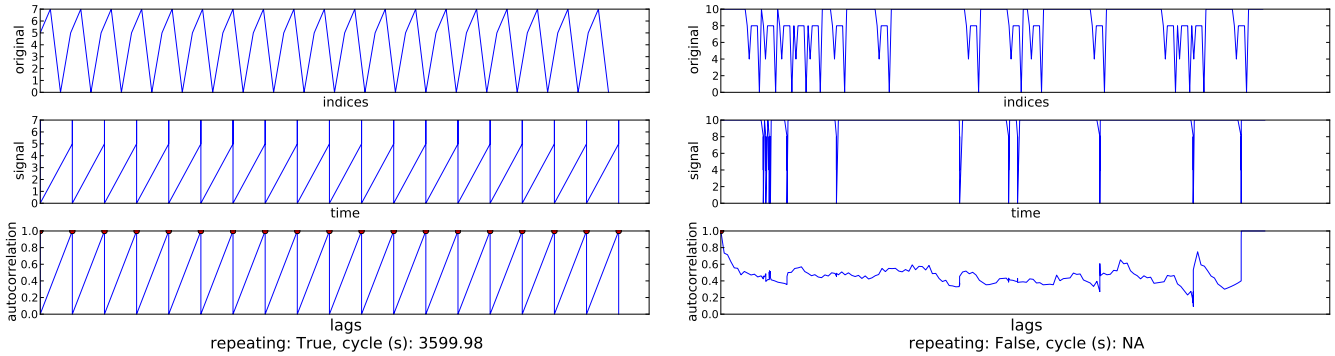
where $\mathbb{I}(\cdot, \cdot)$ is an identity function defined as follows:

$$\mathbb{I}(x, y) = \begin{cases} 1 & \text{if } x \text{ and } y \text{ are of same type} \\ 0 & \text{otherwise.} \end{cases} \qquad (3)$$

The value of r_e falls in $\{0, 1\}$. By r_e, an event sequence is highly correlated if its shifted version and itself has many same event types aligned.

E-Autocorrelation shares the same property with the conventional autocorrelation, that is, when a periodic event sequence is shifted with a lag that is exactly a multiple of the period, r_e achieves its maximum. Thus, E-Autocorrelation is suitable for periodicity detection in event sequences.

Two-step periodicity detection. E-Autocorrelation employs a two-step approach to find periodic events in terms



(a) Periodic event sequence of an idling `sendmail` instance.

(b) Aperiodic event sequence of a `mysqld` instance in active use.

Figure 3: An example of a process's original event sequence, converted sequence as signal in time domain, and autocorrelation (top to bottom). Periodic and aperiodic event sequences are presented respectively in the left and right sub figures.

of both event types and timing. For example, a pattern of interest would look like `open-read-close`. Patterns `open-read-`idle-`close` and `open-`idle-`read-close` will be considered as different patterns since the time interval for idling is at different places or has different length.

Step-1 is to identify periodicity from event sequences using E-Autocorrelation with all timing information ignored. For an event sequence $\{X_1, X_2, X_3, \cdots, X_n\}$, we first calculate the E-Autocorrelation value sequence $\{r_e(1), r_e(2), r_e(3), \cdots, r_e(n-1)\}$. Then from the E-Autocorrelation sequence, we find the values that are no less than a threshold α ($\alpha \leq 1$) and consider them as peaks. Note that in a perfectly periodic sequence, E-Autocorrelation values as 1 indicate periodicity. However, since the event sequences could be noisy, choosing smaller values than 1 as peaks provides a chance for our approach to better tolerate noises and thus remain robust. We use $\alpha = 0.95$ based on our experiments using a one-month training dataset.

After the peaks are located, whether the event sequence (again, without time information) is periodic is determined by looking at the standard deviation σ of the distances between consecutive peaks, where the distances are calculated as the difference of lag values corresponding to the peaks. Our analysis on datasets with ground truth shows that for periodic sequences σ is typically smaller than 0.1, whereas for non-periodic sequences σ is typically larger than 10 or even 100. This indicates that σ could serve as a parameter to differentiate periodic sequences from non-periodic sequences. Thus, we test σ against a pre-defined threshold T (we set T as 0), and only when $\sigma \leq T$, the sequence is considered periodic.

Step-2: The periodic patterns recognized from Step-1 could be incorrect since time interval information is completely ignored. It is still possible that, for example, the sequence $\{A, B, C, A, B, C, A, B, C\}$ is determined as periodic from Step-1. However, the duration of the first A, B, C is 1,000 seconds, whereas the duration of the second A, B, C is 1 seconds and the duration of the last one is 0.001 seconds. In this case, it is not reasonable to claim that the entire sequence is periodic. Thus, in the second step, the results from Step-1 are post-processed so as to filter out the cases that show no periodicity (or with incorrect period) in terms of time. To do so, we find the events that appear first in the periodic pattern identified from Step-1. In the example of sequence $\{A, B, C, A, B, C, A, B, C\}$, such events are the first event (the first A), the fourth event (the second A) and the seventh event (the third A) since the periodic pattern is A, B, C. This can be done by looking at the lags corresponding to peaks, that is, if a peak occurs at lag value k, then the k-th event is the one that appears first in the periodic pattern. Such events are denoted as peak events.

Next, for each peak event at lag k, we need to validate the corresponding period by checking if any event always repeats itself after time t_k. If the validation fails, we drop the corresponding peak. After iterating through all peak events, we are left with a subset of peak events, which correspond to the true periodic pattern. In order to do the validation, we construct a time sequence $\{t_1, t_2, t_3, \cdots, t_m\}$ where t_i is the timestamp of the i-th peak event. Then we compute the standard deviation of a time interval sequence $\{t_2 - t_1, t_3 - t_2, \cdots, t_m - t_{m-1}\}$. If this standard deviation is close to 0, then we consider the original sequence periodic and the true period is the mean of the time interval sequence. Otherwise, we calculate and test the standard deviation of sequence $\{t_3 - t_1, t_4 - t_2, \cdots, t_m - t_{m-2}\}$. The above procedure is repeated until a period is identified or the length of the time interval sequence is 1 (*i.e.,* all the peak events are tested).

Figure 3 shows an example of original event sequences and their autocorrelations. The raw event sequences are shown in the figures in the top. Their converted time-domain signals and autocorrelations are presented below. The left sub figures show a periodic event sequence and the corresponding autocorrelation; the right ones show a non-periodic event sequence. Our E-Autocorrelation can effectively differentiate periodic sequences from non-periodic ones and accurately identify the repeating cycles for periodic sequences.

4. SYSTEM DESIGN

In this section, we describe the design of the two components of LIPzip – data collection and process constraining.

4.1 Data Collection

While it is possible to monitor all activities of every process by using kernel debugging or auditing facilities, such as `ptrace` or `auditd`, doing so would incur significant performance penalty and seriously impact the normal usage of the system being monitored. In order to limit the monitoring overhead to an acceptable level, and allow for large scale deployment in a real working environment, we design a two-stage data collection and screening mechanism.

In the first stage, we perform light-weight monitoring for all processes on a system, by collecting partial information

Event Type	System Calls
process event	`fork,vfork,clone,execve,exit,exit_group`
network event	`socket,bind,connect,accept`
file event	`open,creat,stat,access,dup,close`
IPC event	`pipe,socketpair,shmget,msgget,` `socket,bind,connect,accept`

Table 1: System calls being monitored (partial list)

(such as time stamp, system call type, and error code) for a subset of security-relevant system calls. For instance, our observation indicates that `read` and `write` are among the most frequently used system calls. However, a process must first open a file before it could perform read/write operations on the file, and the usage of `open` and `close` system calls are over an order of magnitude less. And thus, by inferring read/write operations from `open` and `close` system calls, we deduct more than 90% monitoring overhead. We systematically studied the properties of every system call, and selected system calls that are crucial for our analysis, categorized into four types, as shown in Table 1.

With data collected from the first stage monitoring, we apply our E-autocorrelation to screen all processes and discover those that are likely to be "idling". However, due to the incomplete set of system calls, this screening stage produces a small number of false positives (around 18.2% according to our empirical observation) – *i.e.,* some processes classified as "Category I" (complete idle) do in fact make very light-weight system calls which we do not monitor; similarly, some processes classified as "Category II" (periodic) make unmonitored system calls in non-periodic manner.

In order to remove the false positives, we subject those processes to the second stage monitoring. In this stage, we collect more comprehensive information (such as call stacks, parameters and return values) for all system calls. The heavy-weight monitoring would not induce significant impact to the system, because the targeted processes have little or no activity, thanks to the first stage screening.

4.2 Process Constraining: procZip

We define the "constrained" state of an idling process as the following: (1) The process is allowed to continue its execution without any interruption, as long as it behaves consistently with its previously observed idling behavior; (2) When the behavior of the process deviates from its idling behavior, its execution is interrupted, and human interventions are needed to either allow the process to continue execution, or be terminated. Compared with simplistic mitigations, such as directly terminating an idling process, process constraining is a much more practical approach to reduce attack surface, because it leaves room for recourse in case of useful processes being mistakenly classified, and avoids serious negative impact on availability and user experience.

Different system mechanisms, such as *memory protection* and *code instrumentation* which build a program state machine [29, 22], can be leveraged to achieve process constraining to different degrees. Developing a precise and efficient program behavior model is not the focus of our study. In this work, we simply choose to design a simple model based on *system call instrumentation* only to illustrate that idling processes can be easily constrained. In particular, we intercept all system calls of the target idling process, and build its "idling behavior" profile, which comprises of the variety of system calls as well as their *calling context (CC)* [35].

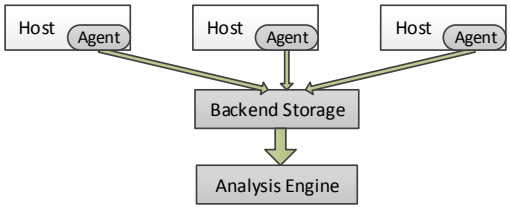


Figure 4: System Architecture

When the target process tries to make a system call that does not match the "idling behavior" profile, this incident is captured and corresponding manual intervention routine (*e.g.,* a pop-up window) is triggered. Because CC is a rich source of process internal state information, this approach could establish precise behavior constraining. Meanwhile, although such detailed system call instrumentation looks heavy weight, since the idling processes are generating little to no system calls, we do not observe any noticeable system slowdown in practice.

For category I processes, because of their complete idleness, the constraining will be in effect immediately. In other words, any system call from these process will be considered as anomaly and thus will trigger manual intervention. For category II processes, a "profile building" phase, which lasts one repeating cycle, will be carried out before the constraining become effective. During the "profile building" phase, any system call and its CC will be recorded as allowed "idling behavior". Afterwards, if a system call is detected with unmatched system call and/or CC in the "idling profile", manual intervention is triggered.

5. IMPLEMENTATION

In this section, we first provide an overview of our light-weight data collection system, which logs and collects process activities. Then, we describe notable implementation details of the process constraining mechanism.

5.1 Data Collection System

We implement and deploy a client-server system to achieve ubiquitous and light-weight data collection. The infrastructure being used for process events collection has a three-tier architecture, *i.e.,* the monitoring agent, the backend storage, and the analysis engine. Figure 4 illustrates the system architecture. The agent deployed on each host is responsible for performing both stages of data collection, as described in §4. For the first stage, the agent monitors a subset of system calls of every process on each host using two major data sources. One is the system call information provided by the kernel built-in Linux Audit subsystem [8]. The other is the auxiliary information from the `proc` file system. Because the majority of the monitoring overhead comes from the processing of data streams coming from the Linux Audit subsystem, we build a custom audit log handler, which achieves 14.3× speedup compared with the stock audit library. For the second stage, the agent leverages the standard `strace` utility to perform the monitoring of all system calls for selected processes.

From real system deployment, we observe that the agent introduces unnoticeable latencies to commonly used programs (*e.g.,* bash, vim, Firefox, Thunderbird, *etc.*). The average resource consumption of the agent is negligible under today's main-stream system configurations. On an idle system, the agent processes about 30 system calls per sec-

ond, and consumes under 1.0% of one processor core, and less than 100MB of memory. With intense workload, such as Linux kernel compilation, the agent processes about 5800 system calls per second, and use up to 14.8% of one processor core and 300MB of memory. For data reporting over the network, each agent on average consumes under 13kbps of network bandwidth.

5.2 Process Constraining

To implement the procZip, we leverage the kernel `ptrace` facility to intercept all system calls of the constrained process. For each intercepted system call, we uncover its calling context (CC) by walking up the call stack using `libunwind`, and concatenate each call site address into a chain. During the "profile building" phase, we store all observed system call and corresponding CC in an associative array of CC-syscall pairs, which is used as the "idling profile".

When a constrained process is detected to attempt "out-of-profile" system calls, we implement a pop-up dialog window to prompt the local user for either approval (*i.e.,* allowing process to continue out-of-profile behavior) or termination. For our evaluation purpose, we also implement an alternative routine, which logs the event while silently allow the out-of-profile behavior to continue. This implementation enables us to non-intrusively perform evaluation on real server systems and workstations.

Of course, procZip will consume some system resources like CPU and memory on top of that of the idling process. Based on our experiment, such resource consumption is negligible.

6. EVALUATION

Data collection. The lightweight agent data was collected from a total of 64 Linux hosts. The data used for analysis was collected from December 2013 to July 2014, a period of over 180 days. The average days for each host is about 155 days during which each machine has rebooted at least once (due to power outage and so forth). Interestingly, most of these machines were never turned off by human even for desktop machines (this phenomenon is also observed in other environment [27]). Out of 64 hosts, we further investigate 24 hosts to study in more details about the process behaviors.

Determining attack surface. After determining idling processes, we prioritize high-impact attack surface cases (that are more likely to be exploited) based on the resource types and the process user ID. Similar as many existing works [31, 32, 33, 5, 3, 4], we define the *attack surface* as a set of communication channels (resources) provided by OS, including files, Internet sockets, and Unix domain sockets, through which an attacker can potentially gain additional privilege.

We summarize what kind of resources we consider and how we determine if they are considered attack surface as a set of rules presented in Appendix. We rank the severity of the attack surface in the following order:

1. A root process with an open public network port.
2. Open files that are writable by anyone or a root process with local listening sockets accessible to everyone.
3. Other attack surface (see Appendix).

Evaluation methodology. We present the following results in the evaluation:

- How many processes are found by our LIPdetect technique as idling? What are their associated security risks (*i.e.,* attack surface)? Results are discussed in §6.1 and §6.2.

Workflow	Role of Human
Operational	Determine safe-to-disable processes
Non-operational	Tag hosts and processes (§6.1)
	Search for historical vulnerabilities of idling processes (§6.2)
	Examine why some idling processes are not constrainable (§6.3)
	Perform case studies to investigate the reasons of the existence of idling processes (§6.4)
	Conduct survey to cross validate our results (§6.5)

Table 2: Clarification of human involvement in the operational workflow and non-operational (evaluation) workflow

- What percentage of discovered idling processes can be easily "quarantined" to reduce attack surface? Results are discussed in §6.3.
- Can these idling processes be safely disabled in the eyes of system administrators and users? What are the reasons for their existence? Results are discussed in §6.4
- Can our findings be shared across different enterprises? Results are presented in §6.5.

For the first result, we use the lightweight agent data to first gather a list of potential idling processes on 64 hosts. Next, we further select 24 hosts to deploy the second-stage monitoring to perform the end-to-end analysis.

For the second result, we deploy procZip to constrain the execution of the idling processes for two weeks and log any deviation attempts or new behaviors.

For the third and fourth result, we perform case study in collaboration with sysadmins and users in our enterprise and a survey with sysadmins in a university department.

It should be pointed out that the system administrators are mainly involved for evaluation purpose. In the operational workflow of our approach, system administrators are only involved (as the last step shown in Figure 2) when one really wants to kill a process or uninstall a program. There is no better way because it is extremely difficult to predict whether an idling process may become useful in the future. Therefore, it has to be human to judge the reasons case by case and determine whether a process can be safely killed or a program uninstalled. Our detection system itself (including data collection, idling process analysis, attack surface measurement) is automated and does *not* require any human intervention. To avoid confusion, we make a summary in Table 2 to differentiate human involvement in the operational workflow and non-operational (evaluation) workflow.

6.1 Idling Processes Summary

First, by analyzing the lightweight agent data, we get totally 5839 long-running processes, among which 2095 were identified as potentially idling. Here, we consider a process long-running if it has been running for longer than 80% of the host uptime. Next, due to privacy and policy concerns, we select 24 hosts to deploy the second-stage monitoring. With `strace` data from 24 hosts (1774 long-running processes), 434 out of the 546 potentially idling processes are determined to be really idling. Among them, Category I has 328 (overall 18.5%) and Category II has 115 (overall 6.5%).

Unique binaries. Even though the number of idling processes is high, they come from only 92 unique programs (*i.e.,* executable names). Some idling programs are commonly seen on many hosts, while some others only exist on a few hosts. Figure 5 plots the fraction of instances of each unique executable in a decreasing order. It is interesting to see that there is a long tail where about 42 binaries occur less than twice in the 24 hosts, indicating a wide variety of idling processes and services. While it would take some time to investigate all of them for system administrators, one can

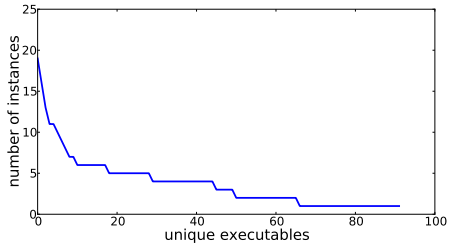


Figure 5: Distributions for number of instances of unique executables

Category I	Instances	Category II	Instances
acpid	19	atd	16
rpc.idmapd	13	rpcbind	5
hald	11	sendmail	5
avahi-daemon	11	mdadm	4
gconf-helper	10	smartd	4

Table 3: Number of unique instances of idling processes and the corresponding category

prioritize which processes to look at first is based on how popular it is inside the enterprise. Table 3 lists the top ranked programs, the corresponding number of instances, and their categories.

Idling process distribution. We further study the distribution of idling processes from the 24 hosts with regard to host types. We want to answer the questions like whether it is desktop OSes or server OSes that have more idling processes and which Linux distributions (*e.g.,* CentOS, Fedora, Ubuntu) would normally include more idling processes. To achieve this, we tagged each host as "User" to indicate a desktop OS, "Server" to represent a backend server, and "Testbed" to mark those testbed servers that are less frequently used. We did this based on our own knowledge and also with the help of the IT personnel. For the type of Linux distributions, our agent actually has a built-in functionality to retrieve this information. Figure 6(a) shows the distributions of idling processes over "User" OSes, "Server" OSes, and "Testbed" OSes, and Figure 6(b) shows the distributions over three Linux distributions. We can see that the "Testbed" machines have much more idling processes than others. And due to the fact that Fedora is only installed on "Testbed" machines, we got a higher ratio for Fedora than CentOS and Ubuntu.

Intuition tells us that the characteristics of an idling process should also affect on its distribution. For instance, the idling processes on a server that is mostly accessed through remote shells might have more programs without GUI; while for a desktop host, GUI processes would instead take a larger portion. Due to the lack of a standard way to classify GUI and non-GUI programs, we manually tag each process. Figure 6(c) shows that the "User" machines do have more idling GUI processes than those "Server" machines.

Syscall patterns. For category I processes, as shown in Figure 7, we observe a variety of blocking system calls. We can see that most of the them are `select`, `read`, and `ppoll` which are operated on file descriptors. Such system calls have a parameter where a timeout value can be specified. We are surprised to see many programs choose to block indefinitely, given that such programming practice may lead to responsiveness issues [6].

A non-negligible fraction (16.3%) of them are not blocked on file descriptors, *e.g.,* `wait4`, `waitid`, and `futex`. At first glance, no attack surface seems to exist. However, it is plau-

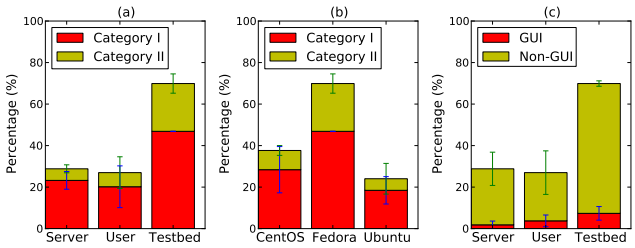


Figure 6: Distributions of idling processes in different aspects. (a) OS purposes: User/Server/Testbed; (b) OS distributions: CentOS/Fedora/Ubuntu; (c) Processes types: GUI/non-GUI

sible that after the syscall returns, the program will start taking input (*e.g.,* by calling `open`), which may constitute attack surface.

For category II processes, out of 115, 49 repeatedly try to take input through system calls like `read` and `recv`, but always failed with errors (*e.g.,* timeout). 18 processes that successfully take input, yet do not produce any output (*e.g.,* no `write` syscall). When the process does produce output, the output is most likely to be exactly the same across time. We measured the entropy [28] of the parameters for the output relevant system calls (*e.g.,* `write`, `writev`, `recv`, `recvmsg`, `recvfrom`) and found that among the 32 processes that produce outputs, 14 have the same exact output, indicating zero entropy.

6.2 Attack Surface and Vulnerabilities

Attack surface. Idling processes not only waste system resources but may also expose attack surface. Here, we study what is the percentage of idling processes which expose attack surface and what kind they are. Note that the attack surface we measure is only a lower bound, as the attack surface is measured only when a process is idling. Once become active, it may start accessing additional resources and open up new attack surface.

With that in mind, we still want to understand the attack surface exposed when these processes are idling. Overall, 127 of 434 (29.3%) processes have direct attack surface through files that are owned by a different user, or listening sockets which can be either accessed by all processes on the same host (if local socket) or by other hosts (if listening on the public interface).

Further, we find that the attack surface of the idling processes consists of 66.7% of attack surface exposed by all long-running processes, based on a snapshot of all the currently open resources on each host. Table 4 provides a breakdown of the attack surface. As we can see, the majority of the attack surface comes from the listening sockets (both remote-

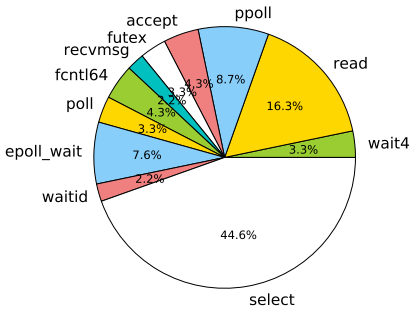


Figure 7: Breakdown of system calls that Category I processes are blocked on

Resource	File	Listening sockets (public)	Listening sockets (local)
Root	10	43	66
Non-root	19	51	53

Table 4: Attack surface breakdown of 434 idling processes

facing and local-facing). It is surprising to see that there are 94 public open TCP/UDP ports (without one connecting to them) from just 24 hosts, which raises significant security concerns, especially considering 43 of them are opened by root processes. Note that these processes are idling and no one has connected to these ports for a long time. For the listening local sockets, they are either TCP/UDP ports bound to loopback address or UNIX domain sockets. Such listening sockets present a threat on local privilege escalation attacks, which is less concerning but still important given the popularity of local privilege escalation attacks in Linux [32, 33]. Note that the number of file attack surface is smaller, indicating the more popular use of sockets as communication mechanisms in long-running processes.

Known vulnerabilities. Out of all the processes, based on their executable names, we find that 47 out of 92 (51.1%) unique binaries have had known vulnerabilities in their development history according to the CVE database [2] and other online sources [1]. It indicates that there can be real threats to the idling processes that we discover. Just to give one example, remote attackers can trigger a vulnerability (CVE-2009-1490) in sendmail by crafting a long X-header to make a buffer overflow, which allows the attackers to execute shellcode and gain root privilege.

6.3 Constrainability Evaluation

Next we want to evaluate how constrainable the idling processes are. Through deploying procZip for two weeks on the 434 idling processes on 24 hosts, we find 384 processes (88.5%) did not have any "new behavior" or deviation from the previously observed behaviors – with a new syscall or an existing syscall at a different call stack, which means that they are very well constrained. 96.3% processes have less than 10 times where new behaviors are observed. More than 93.5% processes converge within one day.

Figure 8 shows the distributions of the number of times that new behaviors appear and the converge time across the 434 processes. The results give a strong indication that the majority of idling processes are indeed constrainable. Of course, ideally, if a process is really idling and the procZip is given the correct repeating cycle, they should be 100% constrainable and should not have any new behaviors observed. In reality, however, why are there still a few idling processes not constrainable? To answer this question, we further investigated 13 such processes from 10 hosts and found that they can be attributed to the following reasons:

First, 3 cases we find are due to the fact that some system calls are issued inside signal handler which may cause new call stack to be observed, depending on which instruction is interrupted. Note that this is not a fundamental limitation of our approach. We could support the signal handling cases by interpreting the parameters of signal handler registration calls (*e.g.,* `signal`, `sigaction`), which allows us to know the address of the handler function. In that case, when we observe a new call stack in the future and see the address of handler function in one of the stack frame, we can simply ignore the immediate stack frame next to it of which the return address can be anywhere in the program.

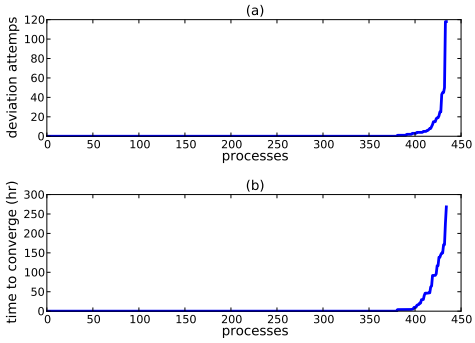


Figure 8: Distributions of (a) deviation attempts and (b) converge time

Second, due to the inherent noise tolerance characteristics of autocorrelation, 3 cases are detected with a repeating cycle shorter than the reality. These 3 cases are considered the inaccuracy of our idling process detection algorithm.

Third, 7 cases are due to real workload change which breaks its previous idling pattern. These 7 cases are considered the false positives of our approach. On one hand, we can argue that such cases are less likely to occur if we monitor their historical behavior long enough. On the other hand, we may want to admit that it is fundamentally hard by any means to predict the future.

6.4 Safe-to-disable Processes and Case Studies

Many idling processes are considered safe-to-disable in the eyes of system administrators and owners. We conduct detailed case studies in our enterprise to identify safe-to-disable processes and the reasons of their existence. The key challenge to perform the study is that a process needs to be examined in the typical runtime context (job-mix) of the host (or cluster of hosts) under examination. To achieve this, we have had intensive conversation with the system administrators and owners of the machines. Finally, we gather a list of processes that are considered safe-to-disable and we provide analysis and insights about such processes. Hopefully, our investigation results can serve as a helpful reference to average system administrators and end users in general.

In our case studies, 133 out of the 434 processes are considered safe-to-disable (30.6%). The top reasons are listed below (without specific order):

- **Supporting unused hardware devices**, *e.g.,* bluetooth devices, 3G/4G data card. Given that we are in an enterprise environment with fairly standard hardware setup: USB keyboard/mouse and Ethernet cables, such processes are very unlikely to be used. After checking with our system administrators, we conclude that they do not actually support the use of bluetooth devices or 3G/4G data cards and therefore these processes can be disabled. As a more detailed example, mdadm is a Linux utility used to manage software RAID devices. In strace log, the mdadm process is repeatedly calling `dup`, `fcntl`, and `close` on a configuration file `/proc/mdstat`, while this file shows that no RAID has been configured. We further check the file system and realize that there is only one physical volume, which does not require RAID management. Interestingly, the reason that the process is running is that the host were previously used to manage RAID devices. When the RAID devices were removed, the person forgot to terminate this service. Besides

this example, most of these processes are actually bundled in the default installation image of the Linux distributions.

- **GUI-related processes that run on server machines**, *e.g.,* notification of sound level change. The servers in our environment are never intended to be used as desktop hosts. As a result, many GUI-related processes can be disabled. For instance, strace logs show that the indicator-sound-service process is completely blocked on a system call on three machines and never return. Actually we find out that they sit in a server room where no one is using it to output any audio, let alone observing the sound level changes.
- **Supporting unused miscellaneous software services**, *e.g.,* virtualization daemon (no VMs are supposed to run on them), super server which is not configured to start any daemon on demand. We find that some machines are pre-installed with the xinetd (extended Internet daemon) which listens for incoming requests over a network and launches the appropriate service for that request. By examining this process in detail, we realize that the configuration file indicates that there is no service registered at all, which indicates that this process is not intended to run. This is also confirmed with system administrators.
- **Applications not properly setup/configured**, *e.g.,* location tracking service, web servers serving default page. Many programs, when installed, are not fully configured to run with intended behaviors, possibly due to human negligence or mistakes. For example, we see an instance where the web server thttpd is installed and running for a long time yet idling. We further check the web directory and find that it only contains the default pages that come with the default installation. In other words, one can only browse the default page (which is a 404 page). Another example is sendmail which is normally used as mail servers and mail relays. Desktop users, in our environment, do not really need this program as our email systems go through the company mail servers (Microsoft exchange server). We typically use IT-supported programs such as Thunderbird to receive emails.

Although rare, another interesting case is that certain application/service is in the process of being retired and its functionalities are to be replaced by a new one. In this case, we observe that the old application/service can still be running for some reason. For example, HAL (Hardware Abstraction Layer) was aimed for providing hardware abstractions for UNIX-like OSes. Since 2011, major Linux distributions are in the process of deprecating HAL and hald is in the process of being merged into udev and Linux kernel. Permitted by the owner, we disable hald on one desktop machine and the user did not notice any influence. The CD and USB icon can still automatically appear on the desktop when being inserted. The discovery of this type of case is very surprising and not originally anticipated by us.

Overall, a large portion of safe-to-disable processes can be attributed to the fact that OSes are packing more and more general purpose applications and utilities. Some of them are installed at a later time, yet they are either not really properly configured to begin with, or they are no longer needed but forgotten to be uninstalled.

Boundary cases There exist a large number of processes that are in the grey area even in the eyes of system administrators. Here are some examples to illustrate such cases: the irqbalance process is a daemon process that balances interrupts across multiple CPUs or cores, which is supposed to provide better performance and IO balance on SMP systems. However, we find an intensive and insightful discussion among the Ubuntu package management people in their online forum about whether to start this process by default [11]. Our survey indicates that irqbalance seems to be targeted at a specific server environment and may sometimes downgrade the performance. Further, on desktop machines, such performance concern is barely serious.

To more objectively evaluate whether the processes are indeed safe to disable. We have asked a few users who have agreed to have a few services disabled, which include: bluetoodhd, mdadm, indicator-sound-service, xinetd, thttpd, irqbalance, sendmail, smartd, hald, and winbindd over the course of a few days. The users do not observe any disruption to the daily use of the machines.

6.5 Cross Validation

Since enterprise systems usually share some common characteristics, we extend our evaluation to another organization to see whether our results can also be shared. Particularly, we conduct a survey with three system administrators in a university department. One of them is the head of the IT group and the other two are both senior system administrators with multiple years of experience. The IT group is responsible for managing computers in the classrooms, faculty offices, and student laboratories. They are also in charge of the department web server, email server, databases, *etc.*

We create a questionnaire containing 20 programs selected from our safe-to-disable list together with the specific reasons of their existence. We ask them to 1) give a score from 0-10 (0-strongly disagree, 10-strongly agree) to indicate to what extent they can confirm our result based on their own experience, and 2) note down whether they happen to find the exact same program on their machines and also think it is unnecessary but somewhat were not aware of before.

Due to page limit, we put the detailed survey results at `https://sites.google.com/site/lipzip15/survey`. Overall, the average score is 9.6, showing a reasonably large degree of agreement. Two sysadmins even state that they find all the 20 programs on their machines that they believe are unnecessary but did not notice before.

7. DISCUSSION

Idling processes and their necessity. The concepts of "idling" and "unnecessary" are not exactly the same thing. An idling process describes a process that is in a state given past observations. Whether or not it is still necessary to keep in the system is a question that requires future knowledge. For instance, an idling daemon process that supports cellular network cards may be idling for the past one year. However, it is impossible to predict if someone may need to use the cellular network card in the next day. Therefore, the necessity of a process can only be answered by prophet, in our case, the system administrators.

In addition, the concept of "necessary" is subjective. An process that is deemed unnecessary may be actively performing useful operations. Consider a process which outputs a pop-up notification window (*e.g.,* for instant messenger applications) to the screen from time to time. Some user may find it useful while others will not and may even find it disturbing. Such problem is outside the scope of this paper.

Idling process and its usage. In this paper, we study a number of idling processes from our automated method and then confirm that a large fraction of them can be safely

disabled or uninstalled. We envision that such knowledge can be shared across enterprises or organizations. In §6.4, we notice that every safe-to-disable process has its reason to be idling. We could actually construct a knowledge base of such safe-to-disable processes. One way to share such knowledge is to compile a list of questions such as whether the organization requires the usage of special hardware (*e.g.,* cellular network connection) or whether the host is to be used as a server or desktop. Answering these questions will then lead to an automated selection of services to install in order to minimize the attack surface while at the same time satisfy the functionality requirement.

8. RELATED WORK

Unnecessary service. Few studies have looked at what services are idling or unnecessary inside an enterprise environment. Guha et al. [19] examined network traffic traces collected from laptops in an enterprise network and looked at services that generate useless traffic (*e.g.,* broadcast discovery messages with no response, failed TCP flows). Although network traffic can also indicate whether a process is idle or not, but it is by no means accurate. For example, a process might have its listening socket idle but still do useful work (*e.g.,* a database server has an idle Internet socket but is serving a local web server via a Unix socket). Our approach looks at both CPU time and system calls, which can precisely describe the idleness of processes.

Attack surface metrics and reduction. Traditionally, attack surface has been evaluated at different system layers. Manadhata et al. proposed a software system's attack surface measurement [26]. This system uses several metrics such as system channels (*e.g.,* sockets), system methods (*e.g.,* API), and the transferred data items to determine the level of risk. Szefer et al. [30] takes a similar approach for virtualization. Authors proposed NoHype, a hypervisor that eliminates the hypervisor attack surface. Kurmus et al. [23] measured attack surface metrics on Linux kernels. Moreover, they reduced the attack surface and kernel vulnerabilities using automated compile-time OS kernel tailoring.

In our work, we choose a set of simple attack surface metrics to measure at the system layers. Most of the above approaches can complement our method to provide more complete and precise attack surface.

Anomaly detection and process behavior models. Numerous attempts have been made to study how to model the behavior of a process and learn its normal behavior for the purpose of performing anomaly detection [20, 29, 22, 18]. Even though our study also models the behavior of a process, it incurs a different set of challenges. Specifically, we examine if existing models can directly be applied in our case. In general, there are three categories of process behavior models: 1) sequence of system calls [20, 25], 2) deterministic Finite State Automata (FSA) [29, 22], and 3) probabilistic FSA (*e.g.,* Hidden Markov Model) [34, 18]. For 1), the idea is to use n-gram model to learn the common n consecutive system calls. The problem is that an idling process needs not only to have common n-grams, but also the exact time alignment of events. For 2) and 3), the model only learns what the next system call is allowed based on the current "state", it does not learn the linkage between a long sequence of events. For instance, if the model has two states A and B, with the possible transition of $A \rightarrow B$, $B \rightarrow A$, and $A \rightarrow A$. It is unclear if this leads to a repeating sequence at all. For example, the allowed sequence could be either $A \rightarrow B \rightarrow A \rightarrow A \rightarrow B \rightarrow ...$ (non-repeating) or $A \rightarrow A \rightarrow B \rightarrow A \rightarrow A \rightarrow B \rightarrow ...$(repeating). Furthermore, such models do not consider time information at all. Our time-augmented version of autocorrelation approach fits well the problem of identifying repeating patterns.

Vulnerability discovery. A number of studies have systematically revealed a class of vulnerabilities related to resource-access. Vijayakumar *et al.* [32] studied name resolution vulnerabilities in Linux systems. For instance, if a root process reads a file in a public directory, an attacker may be able to remove the file and creates a link to `/etc/passwd` to steal sensitive data. According to their study, a significant number of such vulnerabilities exist. Such file-based attack surface is also considered in our evaluation. Process firewall [33] has been proposed to help defend against such class of vulnerabilities. Our study aims at identifying idling processes which can lead to an easier treatment (*e.g.,* disable the service) so as to reduce their attack surface .

Security risk analysis. There exist a number of studies focusing on the security risk of services [21, 15, 16]. Homer et al. [21] presented a sound and practical modeling of security risk utilizing attack graphs and vulnerability metrics. Chakrabarti et al. [15] models the spread of malware and assesses the benefits of immunization of certain nodes. Chan et al. [16] models the attack and defense as interdependent defense games to study the cost-effectiveness of certain security decisions. Such assessment and modeling is abstract using probability to evaluate the likelihood of compromise. Their risk models can be integrated with our system to provide more accurate risk and impact analysis.

9. CONCLUSION

In this paper, we present LIPzip, a system that identifies idling services, an under-studied research area, and explore its security application of attack surface reduction. We define idling process as long-running processes that are in a blocked or bookkeeping state, and we propose an adaptive autocorrelation-based algorithm which can robustly detect idling process from system call information. We evaluate the effectiveness of our algorithm in a real enterprise working environment, using a custom built light-weight and scalable data collection system. With both case studies and automatic validation through process behavior constraining, 30.7% of the identified idling processes can be safely disabled to fully eliminate their attacker surface, and 93.5% can be safely and automatically constrained to reduce attack surface without system administrators' manual intervention.

10. ACKNOWLEDGEMENTS

We would like to thank Bill Kanawyer, Chris Milito, Shannon Johnson, and Eric Treece for their participation in our survey and anonymous reviewers for their insightful feedback. Jun Wang was partially supported by NSF CNS-1223710 and Peng Liu was supported by ARO W911NF-09-1-0525 (MURI) and NSF CNS-1223710. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any organization.

11. REFERENCES

[1] Bugs: Ubuntu. `https://bugs.launchpad.net/ubuntu`.

[2] Common vulnerabilities and exposures. http://cve.mitre.org/cve/.
[3] Cve vulnerabilities : file with the write mode. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=file+write+mode.
[4] Cve vulnerabilities : remote ports. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=remote+port.
[5] Cve vulnerabilities : unix domain sockets. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=unix+domain+socket.
[6] Guide to network programming. http://beej.us/guide/bgnet/output/html/multipage/advanced.html.
[7] How to remove geoclue-master? http://ubuntuforums.org/showthread.php?t=1957331.
[8] The linux audit framework. https://www.suse.com/documentation/sled10/audit_sp1/data/book_sle_audit.html.
[9] MSDN best practices for security. http://msdn.microsoft.com/en-us/library/ms912889(v=winembedded.5).aspx.
[10] Securing a linux desktop part 1: removing unwanted services. http://www.ihackforfun.eu/index.php?title=improve-security-by-removing-services&more=1&c=1&tb=1&pb=1.
[11] Start irqbalance by default? http://ubuntu.5.x6.nabble.com/Start-irqbalance-by-default-td732222.html.
[12] System administrator - security best practices. http://www.sans.org/reading-room/whitepapers/bestprac/system-administrator-security-practices-657.
[13] what is avahi-daemon? http://forums.fedoraforum.org/showthread.php?t=124837.
[14] What's this at-spi-registryd? https://blogs.oracle.com/jmcp/entry/what_s_this_at_spi.
[15] D. Chakrabarti, Y. Wang, C. Wang, J. Leskovec, and C. Faloutsos. Epidemic thresholds in real networks. *ACM Trans. Inf. Syst. Secur.*, 2008.
[16] H. Chan, M. Ceyko, and L. Ortiz. Interdependent defense games: Modeling interdependent security under deliberate attacks. In *In Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012.
[17] C. Chatfield. *The analysis of time series: an introduction*. CRC press, 2013.
[18] A. Frossi, F. Maggi, G. L. Rizzo, and S. Zanero. Selecting and improving system call models for anomaly detection. In *DIMVA*, 2009.
[19] S. Guha, J. Chandrashekar, N. Taft, and K. Papagiannaki. How healthy are today's enterprise networks? In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08.
[20] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. In *Journal of Computer Security*, 1998.
[21] J. Homer, X. Ou, and D. Schmidt. A sound and practical approach to quantifying security risk in enterprise networks. Technical Report (2009): 1-15, Kansas State University.
[22] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. In *IEEE Software*, '97.
[23] A. Kurmus, R. Tartler, D. Dorneau, B. Heinloth, V. Rothberg, A. Ruprecht, W. SchÃűder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *NDSS 2013*.
[24] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS'13)*.
[25] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 2010.
[26] P. K. Manadhata and J. M. Wing. An attack surface metric. In *IEEE Transactions on Software Engineering (2011)*, pages 371–386.
[27] J. Reich, M. Goraczko, A. Kansal, and J. Padhye. Sleepless in seattle no longer. In *USENIX Annual Technical Conference (ATC)*, 2010.
[28] A. Rényi. On measures of entropy and information. In *Fourth Berkeley Symposium on Mathematical Statistics and Probability*, pages 547–561, 1961.
[29] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.
[30] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. CCS, 2011.
[31] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. ASIACCS '12.
[32] H. Vijayakumar, J. Schiffman, and T. Jaeger. Sting: Finding name resolution vulnerabilities in programs. In *Proceedings of the 21st USENIX Security*, 2012.
[33] H. Vijayakumar, J. Schiffman, and T. Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems (EUROSYS 2013)*, April 2013.
[34] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.
[35] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. Deltapath: Precise and scalable calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, New York, NY, USA, 2014.

APPENDIX

Determining Attack Surface. We summarize what kind of resources we consider and how we determine if they are attack surface as a set of rules: (1) If a process has opened a resource in readable mode, and (2) we consider three main resource types: Internet listening sockets (either bind to local or public interface), Unix domain listening sockets, and regular files. (2.1) If it is an Internet listening socket, we directly consider it attack surface as it can be accessed by anyone (host-based firewalls are not configured in our enterprise). Many known vulnerabilities are exploited by first connecting to a listening socket [4]. (2.2) If it is a Unix domain listening socket, we first check if it is file-backed. If so, the file permissions apply when connecting to such a socket, and we treat it similar to a regular file (as described next). Otherwise, if it is not file-backed, we consider it attack surface since anyone local can access it. Vulnerabilities have been reported when accessing Unix domain sockets [5]. (2.3) If it is a file, and the file is not a directory or a special file such as `/dev/null` and files under `/proc`. We consider it attack surface if one of the three following conditions is satisfied: (2.3.1) Everyone can write to the file. (2.3.2) The file owner group ID is different from the process group ID and the group-writable permission bit is set. (2.3.3) The file owner user ID is different from the process user ID and the file owner user ID is not root (as we assume the root is trusted). Vulnerabilities related to files have also been widely discussed [32, 3].