A Tamper-Resistant Framework for Unambiguous Detection of Attacks in User Space Using Process Monitors

Ramkumar Chinchani and Shambhu Upadhyaya
Dept. of Computer Science and Engineering
University at Buffalo, SUNY
Amherst, NY 14260
Email: {rc27, shambhu}@cse.buffalo.edu

Kevin Kwiat
Air Force Research Laboratory
525 Brooks Road
Rome, NY 13441
Email: kwiatk@rl.af.mil

Abstract

Replication and redundancy techniques rely on the assumption that a majority of components are always safe and voting is used to resolve any ambiguities. This assumption may be unreasonable in the context of attacks and intrusions. An intruder could compromise any number of the available copies of a service resulting in a false sense of security. The kernel based approaches have proven to be quite effective but they cause performance impacts if any code changes are in the critical path. In this paper, we provide an alternate user space mechanism consisting of process monitors by which such user space daemons can be unambiguously monitored without causing serious performance impacts. A framework that claims to provide such a feature must itself be tamper-resistant to attacks. We theoretically analyze and compare some relevant schemes and show their fallibility. We propose our own framework that is based on some simple principles of graph theory and well-founded concepts in topological fault tolerance, and show that it can not only unambiguously detect any such attacks on the services but is also very hard to subvert. We also present some preliminary results as a proof of concept.

1. Introduction

With the advent of computer networks, user space services have not only become plausible but also very prevalent. Both academic and other organization environments alike use services of user space daemons such as *inetd*, *sshd*, *lpd*, etc. More complicated services are provided by intrusion detection systems, network management systems, etc. As these services become feature-rich, the code base becomes larger and hence these programs are likely to contain software bugs some of which may be exploitable. Empirical case studies in software engineering and experiences with large projects [14], [9], [12], [11] suggest this.

Maintaining high availability of these services is critical for the smooth functioning of any organization relying on those resources. These services can fail due to software faults or attacks by intruders. Occurrence of faults can result in the unpredictable failure of the system. Intrusions are often likened to faults and successful attacks, like faults, leave the system in an inconsistent or unusable state.

Fault detection and tolerance techniques have goals such as *dependability*, *reliability*, *availability*, *safety* and *performability* which are similar to the goals of intrusion prevention and detection, and other security measures. However, the correspondence is not always one to one. Given the gamut of possibilities through which intrusions or malicious behavior can manifest, the decision-making is very diffuse and hard, unlike detecting atomic faults where the problem is less non-deterministic. Also, faults are generally more random and occur independent of each other, whereas an intruder can target specific parts of a system deliberately in order to disable it as a whole.

Fault tolerance techniques profess redundancy and replication to maintain high availability with the assumption that even if a small minority of them fail, the majority will continue to render the services required of them. Byzantine agreement [23] and voting protocols [36], [19], [4] allow for consistent outputs in spite of failures in some replicated components. However, from the security point of view, it is safe to assume that if an intruder can compromise a particular copy of a service, he can compromise other copies of it as well with relative ease. This voids the assumption that a majority of these copies will be safe and that the detection of any such compromises will be unambiguous. From an intruder's point of view, taking over a service simply means disabling a majority of its copies. Although not trivial, it can still be done with careful reconnaissance and analysis of weaknesses in the service components.

Fault tolerance is the attribute that enables a system to

continue the correct performance of its specified tasks in the presence of hardware or software faults [31], [21]. It allows greater availability and reliability of the particular system component. This is true even in the context of intrusions. But in order to respond to such events, one must provide a mechanism to unambiguously detect these failures.

In this paper, we first discuss the pros and cons of building such detection systems at various levels of the operating system, and the related work. Then, we evolve some theoretical machinery to analyze the various relevant techniques. We propose a framework using some simple concepts in graph theory and show that it has the capability of detecting failures of services unambiguously and also making it very hard for an intruder to subvert it. While known solutions popularly implement this functionality inside the kernel, the crux of this paper is to provide an alternate solution by unambiguously detecting these failures in user space. Although, the focus of this paper is more on failures due to intrusions and attacks, this solution works equally well with inherent software faults and failures.

1.1. Paper Organization

Section 2 gives a more elaborate description of the related work done by peers. Section 3 explains the theoretical aspects of the framework and compares it with the current techniques. Section 4 analyzes the various configurations. Section 5 discusses the implementation issues. Preliminary experiments and results are presented in Section 6. The paper is concluded in Sections 7 and 8 by a discussion and overview of our future work.

2. Background and Relevant Work

2.1. Fault Tolerance Techniques

A software service or component may fail due to different reasons. In order to expedite proper response, it is very important to ascertain the fact that the component has indeed failed. Detection of failures is a decision-making problem with varying degrees of hardness depending on what failure model is considered. For our work we take into account these models:

- Inherent software failures
- Failures due to attacks

Software errors and bugs inevitably creep into large projects due to the human factor. Even though these programs may be tested extensively, some of these software bugs may not be detected. Total elimination of these errors requires program verification that is often not computationally feasible. Consequently, since it is not possible to eliminate all the faults, a workaround is to achieve a high degree of fault tolerance.

Fault tolerant software architectures [5], [16], [22] have a common theme of redundant and replicated software entities or components that communicate with each other using some protocol to arrive at a consistent and correct outcome. These failure models assume failures that occur independent of some system activity.

Networks are often susceptible to outages because failure of components or nodes has resulted in some path or circuit becoming open or incomplete. The problem of developing fault-tolerant networks has been well-studied using topologies and graph theory [32], [6] [30]. Although, we use concepts in graph theory to model the problem, our effort attempts to provide a framework to ensure high availability of processes and services (in the context of intrusions and attacks) on a single host rather than across nodes on a network.

2.2. Intrusion Detection Techniques

The various facets central to providing protection to services are prevention, preemption, detection, deterrence and mitigation. Being a decision-making problem (often non-binary), intrusion detection is very hard to solve. Since the seminal work by Denning [13], significant efforts have been invested in devising new and effective techniques to perform intrusion prevention and detection. [34] presents a collection of about hundred such systems.

Even though the *trusted computing base* (TCB) of a component or process should be as minimal as possible [2], it is rarely the case [25]. A user space process relies on the operating system and if the kernel becomes unstable then the correct behavior of the process is highly suspect. However, kernels are typically well tested and secure from tampering making kernel space implementations a natural choice for achieving tamper-resistance. It then becomes an engineering tradeoff between safety of the intrusion detection itself and possible performance penalties of kernel level implementations. The various pros and cons are evaluated as follows.

- **Completely in the kernel space**

  Many intrusion detection system implementations fall into this category, e.g., [35]. When designing an intrusion detection system, the safety of the system becomes a critical question. Since, the processor privilege levels prevent user space programs to tamper with anything inside the kernel, it becomes an obvious choice to implement the system inside the kernel. Also, if some event immediately forces a user space

program to be woken up, then the context switches can be very expensive. On the other hand, there are some pitfalls too. A bad implementation inside the kernel is equally serious if not more. If the code is added to the critical paths of the kernel, then the performance impact could be very high. The Linux Security Modules (LSM) [1] framework provides a diffuse mechanism to perform checks at various places inside the kernel as compared to a more central system call interception based checks [35], which could be expensive.

- **Completely in the user space**

  If kernel implementation can be avoided, then it is best done outside the kernel in the user space. Some tools such as DWatch [15], [17], etc., are user space programs that watch other daemon programs. The advantage of implementing the detection system completely in the user space [20] is that there is very little overall performance degradation. But, completely exposing such a critical process to the elements does not read well with information assurance analysts. Therefore, the safety of the intrusion detection system becomes a very serious issue because channels of direct access to the user space components exist and the safety of the system depends on how soundly it has been implemented and good access restrictions. The task of protecting such a component relies on the operating system access control mechanisms [29].

- **A hybrid approach**

  The third approach seeks a middle ground [18], [26] straddling the both extremes. User space configuration and notification mechanisms are typically built into the intrusion detection systems for human interaction, e.g., [26] has a client hypervisor configuration mechanism in user space. Even though the intrusion detection system is securely implemented, compromising the notification mechanism could result in alarms not being raised even though an intrusion has occurred.

There are no rigid guidelines on how a detection system should be implemented. This is due to the complexity and variety of the problems. It has also been realized that no single technique is sufficient and a multitude of security systems have to be used in tandem to increase the probability of detection. Such a configuration is an application of the concept of heterogeneous replication or N-version programming [10], [7] to improve detection coverage. However, replication has the undesirable side effects of increased overheads and additional complexity of decision-making.

2.3. Summary

It is clear that protecting services using intrusion detection systems is not adequate since the IDS itself may come under attack and fail, leaving the system vulnerable. It therefore becomes a question of "who watches the protector?", "is it always necessary to deploy a kernel space IDS to protect user space services?" and "are fault-tolerant architectures capable of combating attacks?".

We show that given a set of process monitors in various commonly known fault-tolerant configurations, it is possible to disable them without being detected. The focus of our work is to provide a tamper-resistant framework to detect the failure of any such user space components with a very high accuracy. The following sections deal with the development and implementation of such a framework.

3. Theoretical Framework

In this section, we state some assumptions and terms, and present a theoretical analysis based on them.

3.1. Basic Assumptions

- **A host**

  A host consists of resources and an operating system that mediates their access. There may be some system level access controls such as file permissions in place but they are static. The framework that we propose resides completely on a host.

- **A service**

  A service is a user space component, which could be any program or parts of it that are implemented in the user space. For example, it could be a daemon, an intrusion detection system component, etc. The goal is to maintain high availability of this component and to achieve this, it is essential to detect its failure unambiguously and respond.

- **A protective framework**

  A protective framework monitors and protects a service or a user space component. It consists of one or more active process monitors interacting with each other in some configuration and monitor some user space component. This framework is completely constructed in the user space.

- **A failure event due to intrusions**

  In our analysis, we assume that no software component is 100% secure. Any such component can be successfully compromised with some probability, viz., $P(\mathbf{X})$

but it is not important how this probability is constituted. In general, calculating intrusion probabilities is very hard.

- **Intruder's skills**

  It is only a matter of time before some intruder learns how some system functions, and discovers its weakness and vulnerabilities. We assume that the intruder has full knowledge of the system to begin with.

3.2. Additional Terms

We quantify tamper resistant properties with the following metrics.

3.2.1 Probabilities of Subversion

If a protective framework consists of various components each with a probability p_i of being compromised, then we can speak of the following probabilities of subversion:

- **Total Probability of Subversion**

  The *total probability of subversion* is defined as the probability with which the entire framework consisting of various components can be disabled. Ideally, given n such components each with an individual probability of subversion equal to p_i, the total probability of subversion would be $\prod_{i=0}^{n} p_i$. But, this is the ideal case. A framework design may be such that disabling a part of the system may bring down the whole system.

- **Per-Stage Probability of Subversion**

  The *per-stage probability of subversion* is defined as the probability with which a subset of all components constituting the framework can be subverted at any instant of time without raising any signals of suspicion or intrusion.

  In a general configuration where there can be various components each with some probability of subversion, the per-stage probability of subversion is not necessarily uniform. Due to design flaws in a configuration, it may become possible to sequentially disable the components resulting in the subversion of the entire system. Some parts of the framework may offer greater resistance than others. The smaller the per-stage probability of subversion, the harder it is to subvert that particular component or module. We can define a *minimum* per-stage probability of subversion as the minimum of the per-stage probabilities of subversion of all components or modules constituting the framework. This quantifies the maximum resistance the framework can put up at any given time. For example, let the protective framework consist of two components A and B, and let it be possible that they can be disabled in a sequence, A followed by B. So, the per-stage probability of subversion of this framework would be p_A at some instant of time t_0 followed by p_B at some time t_1. If $p_A < p_B$, then p_A is the minimum per-stage probability of subversion.

  Additionally, we can also define a *maximum* per-stage probability of subversion as the maximum of all per-stage probabilities of subversion in a group of subsets of components. This quantifies the fact that the security of a system is only as strong as its weakest link.

  If the configuration of the protective framework is such that it is possible to disable the whole protective framework by disabling its components in some sequence, then the total probability of subversion becomes less relevant. The per-stage probability of subversion overshadows the total probability of subversion because even if the latter is very low, the effective probability of subversion is the former. Consequently, it is desirable to maintain both of them as low as possible. For the purposes of completeness of analysis, we mention both the probability aspects although the per-stage probability of subversion may be more significant that the total probability of subversion.

- **Degree of Incidence**

  When some entity or process is monitored by d other entities, then that entity is said to have a *degree of incidence* equal to d (Fig. 1(c)). It is desirable to keep the degree of incidence of a process as high as possible since it would become necessary to disable all the processes that are monitoring that particular process to successfully subvert the system.

3.2.2 Framework Overhead

Any process monitor that is introduced in the framework performs some specific task such as monitoring and interacts with other entities. Therefore, its operations incur some overhead. This overhead is due to two components: the overhead when the process is running in isolation and the overhead due to the interaction with other entities or processes. Scalability of a system is determined by the overheads its components generate when they grow in number. Hence it is important to take this factor into account for performance analysis.

- **Overhead when a process runs in isolation**

  When a process is initiated, it causes some overhead in terms of memory and processor usage. It is represented as a function δ (Fig. 1(a)). Since, the process typically runs in an information processing loop, this overhead is more or less a constant per process.

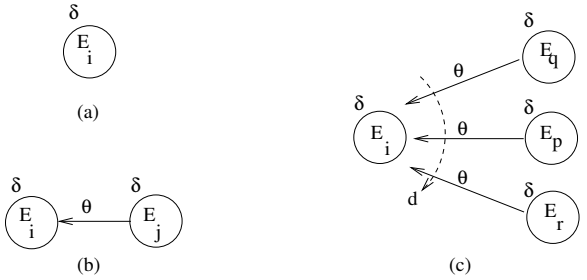

Figure 1. (a) Overhead δ due to isolated execution (b) Overhead θ of monitoring another process (c) Degree of incidence d for a process

- **Overhead when a process monitors another process**

  When a process monitors another process, it incurs an overhead due to information processing. Let the function that describes it be θ (Fig. 1(b)). This overhead depends on the number of processes that it interacts with.

3.3. Mutual Trust

Problems requiring protocols of communication are hard to solve since any entity or process can emanate false data and "lie" about the information that it sends. Hence, we consider models that do not require any communication among processes or in other words, there is no *mutual trust* between them. Any given process monitors a subset of its neighbors but does not participate in any direct communication with them. Actual monitoring and information gathering occurs indirectly via mechanisms that the operating system provides.

3.4. Problem Transformation

With the above conceptual machinery, one can transform the framework design problem into a graph topology problem. The transformation proceeds as follows:

1. Each process becomes a node. Associated with each node is the overhead due to isolated execution.

2. If a process E_i monitors another process E_j, then there is a directed edge from node i to node j. This edge exists as long as the process that is monitoring is not disabled. Associated with this edge is the overhead of monitoring.

3. The degree of incidence is the number of incoming edges to a node. Note that there are a few rules by which edges can be formed. An edge from a process to itself (see Fig. 2(a)) is useless since disabling the process disables this edge also. Having multiple redundant edges from one process to another process (see Fig. 2(b)) is also undesirable since it only increases overhead without increasing the overall monitoring capability.

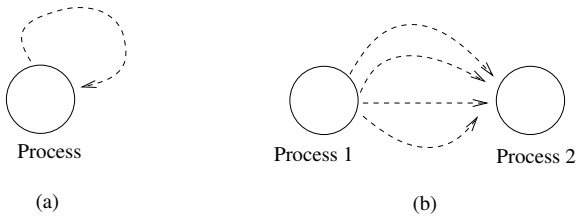

Figure 2. Undesirable edge formations (a) self-loop (b) redundant edges between the pair of processes

The goal is to minimize the total and per-stage probabilities of subversion while keeping the overhead as low as possible. Since no process trusts the other and there is no direct communication, inter process communication does not exist and need not be represented.

4. Topological Configurations

In this section, we describe and analyze the various relevant techniques and finally present the most optimum solution to the problem. Each configuration is composed of a few processes in some topological arrangement. We make some claims [1] and justify them subjectively.

4.1. Simple Replication

Claim 1 *The simple replication scheme has the weakest tamper resistant properties.*

One common technique in fault tolerance domain to increase availability is by service replication. Multiple copies of a process can be executed on a host to increase availability per host. These n number of processes directly monitor the user space component E (ref. Fig. 3(a)).

This may initially appear as a good solution but it has some serious drawbacks. It is possible for an attacker to disable each of the processes in the protective framework with relative ease. There is hardly any self-protective capability.

4.2. Layered Hierarchy

Claim 2 *The layered hierarchy scheme is only marginally better than the simple replication scheme.*

[1] Complete mathematical proofs to these claims are omitted

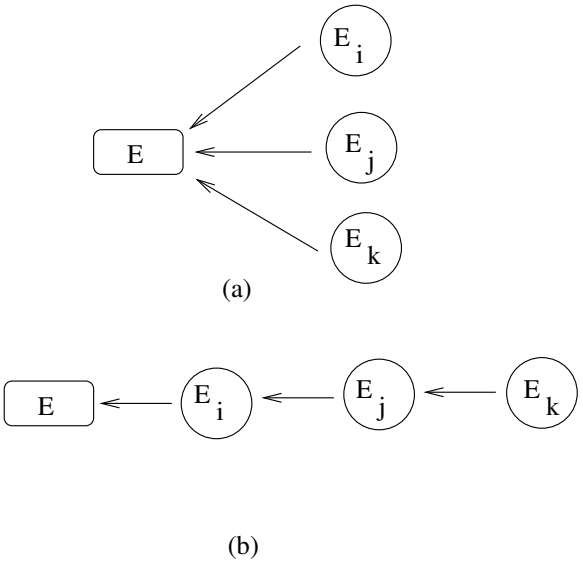

Figure 3. (a) Simple replication of processes (b) A simple layered hierarchy

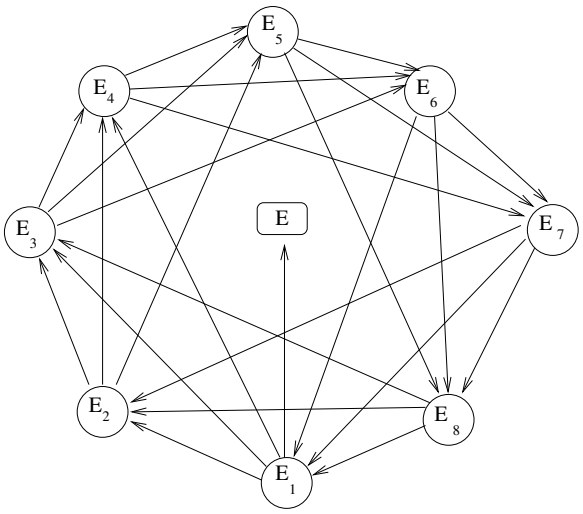

Figure 4. A circulant digraph configuration with $n = 8$ and $d = 3$

Multiple wrappers can be defined around a user space component to monitor it. [28] speaks of an "onion peel" model to provide stronger deterrence.

Let each layer be composed of one process and let there be n such layers. Each layer monitors its lower layer and the lowest layer directly monitors the user space component E. This forms a layered hierarchy of processes (Fig. 3(b)). This configuration is a little better in terms of self-protective capability since an attacker has to detect the sequence and disable the processes in that strict order.

4.3. Circulant Digraph

In this section we show that the most optimum configuration is that of a circulant digraph.

A *circulant graph* is defined as a graph $C_{i_l}(n)$ of n vertices in which the ith vertex is adjacent to the $(i+j)$th and the $(i-j)$th vertices for each vertex j in the list l. For example, $C_{i_{1,2,3,\cdots,\lfloor n/2 \rfloor}}(n)$ is a complete graph and $C_{i_1}(n)$ is a cyclic graph.

Relaxing the properties of symmetry and adding the requirement of directed edges, it can be equivalently defined as a graph $C_{i_l}(n)$ of n vertices in which the ith vertex has a directed edge to the $(i+j)$th vertex for each j in l. Then, $Ci_{1,2,3,\cdots,(n-1)}$ is a complete digraph. In general, a circulant digraph of in-degree or out-degree d can be defined as a graph $Ci_{1,2,3,\cdots,d}$, where $1 \leq d \leq (n-1)$.

In this scheme, the processes are arranged in a circulant digraph topology with some degree of incidence d. Figure 4 shows a framework where processes are arranged in a circulant digraph configuration with $n = 8$ and $d = 3$. At least one process in the group has an additional responsibility of monitoring the user space component. This configuration has a strong self-protective capability.

Claim 3 *A circulant digraph configuration provides the strongest* (in the theoretical context) *tamper resistance properties.*

It is not possible to subvert any subset of processes without alerting the processes monitoring them. If the entire framework has to be subverted, all the processes have to be subverted at exactly the same time.

Consider the trivial case when the degree of incidence $d = 1$. Disabling the entire framework requires subversion of every process and also its parent before it can respond to that event. There is a definite sequence in which this can be done and it is some circular permutation of the processes. Therefore, the intruder has to detect the correct circular permutation and disable the processes in that order. But in order to do so, he must make a correct guess from a large number[2] of circular permutations.

As such, a successful attack requires precise synchronization, and when the degree of incidence $d > 1$, the situation is further complicated, since subversion of any process alerts multiple processes.

5. Implementation Issues

The circulant digraph framework requires the implementation of a *closed loop*, where events or messages are delivered in real time. Operating systems such as FreeBSD and

[2]Number of circular permutations is of exponential order

Linux support direct interaction between processes through the *ptrace(2)* family of system calls [20] or the *proc file system* (which *truss* uses). In either approach it is possible for a process to trace the execution of another process. However, taking control of the traced process on each event makes it undesirable to form such a closed loop (see Fig. 5(a)). This is due to the fact that there is a possibility of an exponential cascade of events and deadlocks [3] Therefore, design decisions occlude such closed loop relationships between traced processes.

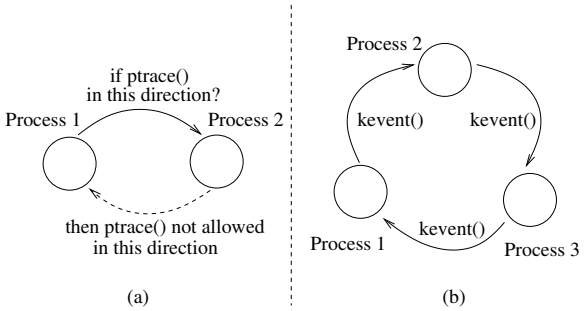

Figure 5. Monitoring process events using *ptrace(2)* and *kevent(2)* (a) *ptrace(2)* doesn't allow a loop (b) *kevent(2)* being asynchronous allows such loops

5.1. Kernel Event Subsystem

Both the approaches discussed above result execution and control flow in a lock-step manner. Since each process monitor in our framework executes in a tight sense-decide-act loop, it calls for a more asynchronous event notification mechanism. Currently, the operating system with such a support and the convenience of experimentation is FreeBSD. The event notification mechanism on FreeBSD is called *kqueue* [24] . It is a highly scalable and generic event notification mechanism proposed as a replacement for *poll(2)* and *select(2)*. Web servers have been reported to achieve significant performance gains [24] when using the kqueue subsystem for socket events. However, it provides only a limited set of events that can be monitored such as *exit(2)*, *fork(2)* and the *exec* family.

When a process wants to listen to the events generated by another process, it first creates a *kevent* handle and then registers a listener for the target process. It then enters a loop, waiting for events to occur. The *kqueue* subsystem allows multiple processes to register listeners for the same event. The edges discussed in the earlier sections manifest as the listeners registered for some specific events. Since the event delivery is asynchronous unlike *ptrace(2)*, it is possible to define loops among the process monitors (see Fig.5(b)).

Process level behavior can be captured by monitoring the system calls the process makes [20], [35]. Other relevant ways of run time verification of process behavior are proof carrying code [27] and model carrying code [33]. Hence, run time monitoring of processes becomes an instance of the problem for which these techniques are proposed as solutions. It must be noted that the support at this level to monitor a larger set of events is not currently present in FreeBSD. Since *kqueue* doesn't support every possible event at the process level, we have to patch the operating system to provide the additional functionality such as retrieving arguments to the *exec(2)* family of system calls, etc.

6. Experiments and Results

Since the simple replication and layered hierarchy schemes are flawed from the security point of view, we do not consider them for empirical evaluations. On the other hand, the circulant digraph has been theoretically proven to be very good in terms of protective capabilities and only this configuration is considered and verified empirically for feasibility and strength of the approach.

6.1. Experimental Setup

The simulations were conducted on a uniprocessor Pentium III 450MHz PC with 64 MB RAM running FreeBSD 4.5. This version of FreeBSD has adequate support for preliminary implementation through the *kqueue* subsystem to demonstrate the strength of the approach. However, a complete implementation would require monitoring of a larger set of events than it currently supports.

The target of the monitoring framework is *inetd* daemon. This is a generic service that spawns off other appropriate daemons to handle network connections. Each active entity or node of this monitoring framework is implemented as a process and there is an edge in the topology graph if a process monitors another process. Monitoring multiple processes at the same time, i.e., when the degree of incidence is greater than one is made possible by multi-threading. The attacker is given all information regarding the process monitors and their relationships. In reality, this information about the edges is actually hard to obtain. This setup satisfies all the basic assumptions and the developed theoretical framework.

6.2. Attack Scenarios

Current limitations of the *kqueue* subsystem allow us to test the framework against crash attacks. An attacker can cause any number of the process monitors to crash. This is made possible by implementing the process monitors in such a way that they do not handle any signals and they can

each be killed via the `kill` command. Each process monitor registers a handler for the *exit(2)* system call. An intruder is successful if he can cause all the monitors to crash before one of them can raise a signal.

When a process receives an event that it is listening for, it simply prints a message to the screen and this is perceived as an intrusion signal. This configuration is tested under light and heavy loads. It is still theoretically possible to subvert this configuration if the intruder can chance upon the right sequence of processes. This is an instance of a *time of check to time of use* (TOCTOU) attack [8]. While the responses may be quick under light load, the window between the event and response widens under high load. However, most TOCTOU attacks are successful only when one such window exists. In our case, there are multiple such windows and all of them have to be predicted and exploited. This makes the entire exercise very hard for an intruder.

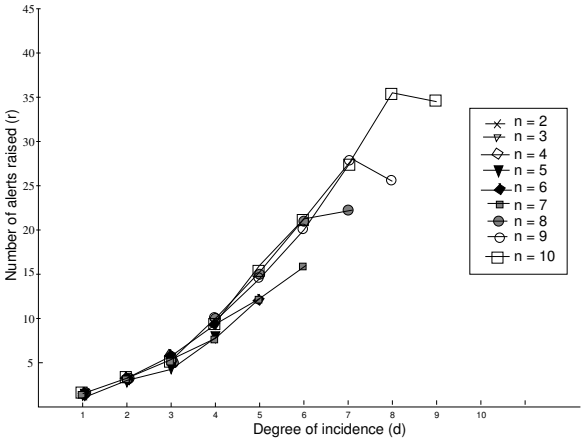

Figure 6. A plot of number of responses (r) against the degree of incidence (d) for various values of number of processes (n) under heavy system load

6.3. Recognizing Attacks

The plot of number of intrusion signals with respect to the degree of incidence for each configuration under heavy system load is given in Fig. 6. The load was increased from 0 to 20 (approximately) processes per minute. The graph characteristics under light system load are very similar if not the same.

We have plotted only the minimum of responses seen when trying various sequences of attacks. Averages may have been higher but they do not correctly reflect the nature of success or failure of attacks, hence we did not choose to plot the averages. In the weakest possible configuration, i.e., with degree of incidence equal to 1, there was consistently one response regardless of the number of processes. At least in our experiments, it was not possible to successfully subvert this setup. While theoretical analysis showed that this configuration fails if all the processes are subverted at the same time, implementation of the operating system and the kernel event mechanism prevents such parallelism due to the strict serial execution in critical sections of the operating system code, e.g., locking and unlocking of the run queues, kernel event queues, etc. This is true even in multiprocessor environments.

Empirical results suggest that the number of intrusion signals is more closely tied to the degree of incidence than the number of processes in the setup. For the same degree, when the number of processes increases, the number of responses remains the same. But the number of responses increase when the degree of incidence increases.

6.4. System Overhead

The memory footprint of each process monitor was only 1.4 MB. The *kqueue* subsystem was introduced to counter the scalability problems due to *poll(2)* and *select(2)*. All the processes are in sleep state until an event occurs and they are woken up. Even with a large number of processes, there is hardly an increase in processor usage. This just shows that the overhead of monitoring is negligible.

6.5. Summary

Initial experiments show as a proof of concept that not only is this framework very strong in terms of tamper resistance but also incurs very small overheads making it highly scalable.

7. Discussion

This paper shows that currently known techniques in the fault tolerance domain are not sufficient to solve the problem of ensuring high availability of a user space service because of the inherent differences between faults and attacks. By formulating the problem differently, we are able to devise an alternate scheme to kernel based protection. This framework theoretically provides good tamper resistance against attackers. The preliminary implementation and results show the feasibility of such a framework.

This monitoring framework has certain restrictions and limitations. The processes in the framework rely on the operating system to provide a secure event notification mechanism. Consequently, this framework can be deployed only when it is known that the system it is being installed on, has not been compromised. Also, current operating systems provide only limited support for the framework's implementation in terms of asynchronous event monitoring.

The strength of this framework lies in the fact that none of the processes require any direct communication since they do not trust each other and there is no single hierarchy (every process monitor monitors some process and it by itself is always being monitored). It is not possible to subvert a subset of them and succeed in an attack; all of them have to be disabled and at exactly the same time. Hence, merely knowing the vulnerabilities in implementation of the framework is not adequate.

It must be noted that while it was not possible to subvert the framework during our experiments, this should not be misread as a tamper-proof (instead of tamper-resistant) property. It is still theoretically possible to subvert it, only that it is non-trivially hard.

We are currently investigating the feasibility of fully implementing this framework and conjecture that it may be possible to provide this generic wrapper around most daemons to make them tamper-resistant. While the focus of the protective framework was to safeguard a user space service, the concept is applicable and can be extended to those domains which use sensors and monitors and their safety becomes an important issue. In other words, it addresses the question of who watches the watcher, wherever this framework is relevant.

8. Future Work

The most outstanding future goals of this ongoing project are as follows:

- Construct a more complete kernel event subsystem by supplementing the existing *kqueue* subsystem.
- Provide sufficient primitives to construct complex filters.
- Proper criteria to decide parameters such as number of processes, etc., on a per system basis.
- Testing and deployment on different user space services.

9. Acknowledgments

Our acknowledgments to the various people on the FreeBSD hackers group for their valuable input and indirect guidance in shaping this effort. We also thank the anonymous reviewers whose constructive suggestions led to an improved paper.

References

[1] Linux security modules, http://lsm.immunix.org/.

[2] Trusted computer security evaluation criteria. *DOD 5200.28-STD*, 1985.

[3] Freebsd problem report kern/29741, http://www.freebsd.org/cgi/query-pr.cgi?pr=kern/29741, April 2002.

[4] G. Aggrawal and P. Jalote. An efficient protocol for voting in distributed systems. In *Proc. 12th IEEE International Conference on Distributed Computing Systems*, pages 640–647. Yokohama, Japan, 1992.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high avaliablity. In *FTCS-22*, pages 76–84, 1992.

[6] B. W. Arden and H. Lee. Analysis of chordal ring network. *IEEE Trans. Computers*, 30:291–295, April 1981.

[7] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, December 1985.

[8] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[9] J. B. Brown. Standard error classification to support software reliability assessment. In *AFIPS Conference Proceedings, 1980 National Computer Conference*, pages 697–705, 1980.

[10] L. Chen and A. Avizienis. N-version programming: A fault tolerance approach to reliability of software operation. In *8th International Symposium on Fault Tolerant Computing Systems*, 1978.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Symposium on Operating Systems Principles (SOSP)*, pages 73–88. Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*. Hilton Head Island, SC, January 2002.

[13] D. Denning. An intrusion detection model. *IEEE Transactions of Software Engineering*, 13(2):222–232, February 1987.

[14] A. Endres. An analysis of errors and their causes in system programs. *IEEE Transactions on Software Engineering*, SE-1(2):140–149, June 1975.

[15] U. Eriksson. Daemon watcher, http://siag.nu/dwatch/.

[16] L. E. M. et al. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(12):54–63, 1996.

[17] J. Fox. Argus, http://staff.washington.edu/fox/argus/.

[18] T. Fraser, L. Badger, and M. Feldman. Hardening of cots software with generic software wrappers. In *Symposium on Security and Privacy*, 1999.

[19] D. K. Gifford. Weighted voting for replicated data. In *7th ACM Symposium on Operating Systems*, pages 150–162, December 1979.

[20] K. Jain and R. Sekar. A user level infrastructure for system call interception: A platform for intrusion detection and confinement. In *Network and Distributed Systems Security Symposium (NDSS)*, 2000.

[21] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall PTR, Englewood Cliffs, NJ 07632, 1994.

[22] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, 1999.

[23] L. Lamport, R. Shostak, and M.Pease. The byzantine generals problem. *ACM Tans. on Prog. Lang. and Syst.*, 4:382–401, 1982.

[24] J. Lemon. Kqueue: A generic and scalable event notification facility for freebsd. In *BSDCon 2000*. Monterey, CA, 2000.

[25] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and John F. Farrell (National Security Agency). The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*, pages 303–314, October 1998.

[26] T. Mitchum, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Annual Computer Security Conference*, December 1997.

[27] G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. Paris, January 1997.

[28] U. Nerurkar. Security analysis & design: A strategy that's both practical and generic. *Dr. Dobbs Journal*, featuring Computer Security, November 2000.

[29] T. Onabuta, T. Inoue, and M. Asaka. A protection mechanism for an intrusion detection system based on mandatory access control. *Information Processing Society of Japan*, 42(8), August 2001.

[30] J. M. Peha and F. A. Tobagi. Analyzing the fault tolerance of double-loop networks. *IEEE/ACM Transactions on Networking (TON)*, 2(4), August 1994.

[31] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.

[32] C. S. Raghavendra and M. Gerla. Optimal loop topologies for distributed systems. In *Seventh Data Communications Symposium*, pages 218–223. Mexico City, Mexico, October 27-29 1981.

[33] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (mcc): A new paradigm for mobile-code security. In *New Security Paradigms Workshop (NSPW'01)*. Cloudcroft, New Mexico, September 2001.

[34] M. Sobirey. Intrusion detection systems (a repertoire of known intrusion detection systems), http://www-rnks.informatik.tu-cottbus.de/ sobirey/ids.html, 2000.

[35] A. Somayaji and S. Forrest. Automated response using system call delays. In *Usenix Security Symposium*, 2000.

[36] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.