1. Introduction

 Automated intrusion response is an important unsolved problem in computer security. A system called pH (for process homeostasis) is described which can successfully detect and stop intrusions before the target system is compromised. The pH monitors every executing process on a computer at the system-call level, and responds to anomalies by either delaying or aborting system calls.

 Until now intrusion detection systems (IDS) were focused mainly on prevention and detection and practiced on networks (terminating connections, blocking messages) but were not widely deployed because the risk of an inappropriate response is very high (false positives). To counter this problem we need a system that provides some definitions for normal and ambiguous behaviour about what constitutes an anomaly. This will be a system in which a computer autonomously monitors its own activities, routinely making small corrections to maintain itself in a “normal” state.

 We design pH which is a set of extensions to a Linux kernel which does not interfere with normal operation but can successfully stop attacks as they occur. In general, it connects system calls with feedback mechanisms that either delay or abort anomalous system calls.
The pH provides two main contributions; the feasibility of monitoring every active process at the system-call level in real-time, with minimal impact on overall performance. Second, it introduces a practical, relatively non-intrusive method for automatically responding to anomalous program behavior.

2. Background

The system call monitor is implemented by using Traces. Each time a process was invoked, we began a new trace, logging all the system calls for that process. Thus, for every process the trace consists of an ordered list (a time-series) of the system calls it made during its execution. For commonly executed programs, especially those that run with privilege, we collected such traces over many invocations of the program, when it was behaving normally. We then used the collection of all such traces (for one program) to develop an empirical model of its normal behavior.

The method must be suitable for on-line training and testing. That is, we must be able to construct the model “on the fly” in one pass over the data, and both training and testing must be efficient enough to be performed in real-time. The method must be suitable for large alphabet sizes (about 200 system calls). Finally, the method must create models that are sensitive to common forms of intrusion. Traces of intrusions are often 99% the same as normal traces.

Let's see a working example about traces, in which we have a series of system-calls: [execve, brk, open, fstat, nmap, close, open, nmap, munmap] and window (w) size of 4 (window is set by the user).
We slide the window across the sequence, and for each call we encounter, we record what call precedes it at different positions within the window, numbering them from 0 to w-1, with 0 being the current system call. When a call occurs more than once in a trace, it will likely be preceded by different calls in different contexts. We compress the explicit window representation by joining together lines with the same current value.

At testing time, system call pairs from test traces are compared against those in the normal profile. Any system call pair (the current call and a preceding call within the current window) not present in the normal profile is called a mismatch. Any individual mismatch could indicate anomalous behavior, or it could be a sequence that was not included in the normal training data (false positive). The current system call is defined as anomalous if there are any mismatches within its window.

### 3. pH Design

To minimize I/O requirements and maximize efficiency, stability, and security, we have implemented most of pH in kernel space.

For each running executable, pH maintains two arrays of pair data: A training array and a testing array. The training array is continuously updated with new pairs as they appear; the testing array is used to detect anomalies, and is never modified except by replacing it with a copy of the training array. Put another way, the
testing array is the current normal profile for a program, while the
training array is a candidate future normal profile. A new “normal” is
installed by replacing the testing array with the current state of the
training array. This replacement occurs under some conditions which
are setted by the user either by using specific system–call or
parameters that follow some pre-arranged formulas.

In pH the execve system call causes a new profile to be loaded.
Thus, if an attacker were able to subvert a process and cause it to
make an execve call, pH might be tricked into treating the current
process as normal, based on the data for the newly loaded
executable. Comparing Locality Frame Count (LFC) which stores the
number of the past n system calls which were anomalous, with the
parameter abort_execve, this possibility is countered by our system.

4. Implementation

The modified kernel is capable of monitoring every executed
system call, recording profiles for every executable. Program profiles
for each executable are stored on disk. Each profile contains both a
training and testing array, and so is actually two “profiles”. When a
new executable is loaded via the execve system call, the kernel
attempts to load the appropriate profile from disk; if it is not present,
a new profile is created. The profiles can be shared across processes.
When all processes using a given profile terminate, the updated
profile is saved to disk. A loaded profile consumes approximately 80K
of kernel (non-swappable) memory. The system call dispatcher is
also changed to call a pH function prior to the dispatching system
call.
5. Experimental Results

The researchers tested the effectiveness of pH system in intrusion response using a variety of attacks. To sum the results, pH can detect and stop attacks (or create huge delays) before an attacker can compromise it.

As for the overhead in performance when running pH, the table below shows the standard running time of a system-call when not using pH and when pH runs on the system (in μs). Although the table shows a significant performance hit, they are not indicative of the impact on overall system performance.

<table>
<thead>
<tr>
<th>System Call</th>
<th>Standard (μs)</th>
<th>pH (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid</td>
<td>1.1577 (0.000000)</td>
<td>5.8898 (0.00025)</td>
</tr>
<tr>
<td>getusage</td>
<td>1.9145 (0.000000)</td>
<td>6.6137 (0.00138)</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>1.6703 (0.00184)</td>
<td>6.3779 (0.00112)</td>
</tr>
<tr>
<td>sigaction</td>
<td>2.5609 (0.00010)</td>
<td>7.2928 (0.01029)</td>
</tr>
<tr>
<td>write</td>
<td>1.4135 (0.00187)</td>
<td>6.1637 (0.00075)</td>
</tr>
</tbody>
</table>

Additionally, if delays are not selected by the user, there is almost none overhead in performance. It is worth to mention, that some complex programs, were slow in terms of achieving a normal profile or even crashed. In those cases restarting the programs
some times or let them run for a day to let pH learn the normal
behaviour, was found to solve those problems.

6. Discussion-Conclusion

It is feasible with pH to use system-call delays to stop intrusions
in real-time, without prior knowledge about what form an attack
might take, for different types of attacks and for all kinds of
programs.

Although there is an ongoing risk that pH could be trained to
accept intrusions as normal behavior. It may be necessary to
implement a default timeout mechanism through pH, in which any
process that is delayed beyond a certain point is automatically
terminated. It may also be necessary to increase pH’s repertoire to
include actions such as system call parameter modifications.