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. In its current form, pH monitors every executing process on a computer at the system-call level, and responds to anomalies by either delaying or aborting system calls.

Computer security research has focused almost entirely on prevention (e.g., cryptography, firewalls and protocol design) and detection (e.g., virus and intrusion detection). Response has been an afterthought, generally restricted to increased logging and administrator email. Commercial intrusion detection systems (IDSs) are capable of terminating connections, killing processes, and even blocking messages from entire networks; in practice, though, these mechanisms cannot be widely deployed because the risk of an inappropriate response is too
high. Thus, we must seek solutions that enable our computers to respond to attacks autonomously.

However, good anomaly detection comes at the price of persistent false positives, which can not be eliminated completely, due to the dynamic nature of computers. Therefore, any automated response system must be designed to account for persistent false-positives, evolving definitions of normal, and ambiguity about what constitutes an anomaly. So, to approach the automated response, we need a system that will be able to autonomously monitor its own activities and and will routinely make small corrections to maintain its homeostasis.

2. The System

The system of pH, is a set of Linux kernel extensions, it does not interfere with the normal operation and is able to successfully stop attacks. pH is implemented to connect system calls with feedback mechanisms that can delay and abort the anomalous ones. By implementing the delays as an increasing function of the number of recent anomalous sequences, pH can smoothly transition between normal execution and program termination.

This paper proposed two main contributions. Firstly, it demonstrates the feasibility of monitoring every active process at the system-call level in real-time with minimal performance overhead. Secondly, it introduces a practical, relatively non-intrusive method for automatically responding to anomalous program behavior.

The main idea of the system is that for each process that will be invoked, a new system-call trace will begin. The system will collect these traces
over many invocations of a certain program, while behaving normally. Then, this collection of traces will be used to develop an empirical model of its normal behavior, so as to categorize most normal behavior as “normal”, whereas most attacks will be categorized as “abnormal”.

There are some constraints to take into consideration. First, 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. Next, the method must be suitable for large alphabet sizes. Our alphabet consists of all the different system calls (about 200 for UNIX systems). 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, with very small, temporally clumped deviations from normal behavior.

Let’s see an example of the traces. Suppose we had as normal the following sequence of calls: “execve, brk, open, fstat, mmap, close, open, mmap, munmap” and a window size of 4.

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 (a true positive), or it could be a sequence that was not included in the normal training data (a false positive). The current system call is defined as anomalous if there are any mismatches within its window.

3. Design

pH performs two important functions: It monitors individual processes at the system-call level, and it automatically responds to anomalous behavior by either slowing down or aborting system calls.

In addition, most of pH is implemented in kernel space, so as to minimize I/O requirements and maximize efficiency, stability and security.

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. pH responds to anomalies by delaying system call execution.
Because pH monitors process behavior based on the executable that is currently running, 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.

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. 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. If another process runs the same executable, the profile is shared between both processes.

They, also, modified the system call dispatcher so that it calls a pH function (pH process syscall) prior to dispatching the system call. pH process syscall implements the monitoring, response, and training logic.

5. Experimental Results

pH can detect and stop attacks in time to prevent system compromise.

Concerning the overhead of running pH, although the following table shows a significant performance hit (latency in microseconds), it is not indicative of the impact on overall system performance. Also, if delays are turned off, a user can use the modified workstation without noticing any differences in system behavior.
Furthermore, programs which make large numbers of system calls in a short period of time tend to acquire normal profiles, even when a true sampling of behavior has not yet occurred.

<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>

6. Conclusions

pH can use system-call delays to stop intrusions in real-time, even for very different types of attacks.

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.

However, pH in its current form monitors and responds to anomalies in all programs.