Continuous Monitoring Hardware for Proactive Computer System Security

Computer Measurement Laboratory
info@cmlab.biz

In the world of computer security, the threats to mission and safety critical systems evolve very rapidly. To further complicate matters, it is simply not possible to completely second guess and/or anticipate the direction that a potential antagonist might take to gain control of system infrastructure. Unfortunately, the notion that it is possible to anticipate and monitor for known threats is central to the current state of the art for computer security systems. A key element of this reactive philosophy is the notion that it is possible to recognize anomalous activity based on signatures or profiles of attacks that have been seen before. However, it is without question that the most effective security assaults are novel. They have never been seen before. Their signatures or attack profiles are nonexistent.

Two excellent examples are the Sasser worm released in 2004 and the Conficker worm released at the end of 2008. In the latter case, over 9 Million Windows based PCs were infected with the worm; including the UK Ministry of Defense. The reactionary approach was absolutely not effective in catching these previously unseen exploits prior to their leaving a path of destruction and mayhem behind.

It follows, then, that this reactive approach is not really viable where mission and safety critical security really matters. The reason for the lack of robustness of the reactive approach is that it is based on the detection of software anomalies. Unfortunately, the state space of these anomalies is very large, and is essentially unbounded. It is literally impossible to solve the problem this way.

The clear alternative to the reactive approach to computer security is a proactive one. The key element of our proactive approach is that it is based on measurement and control: two disciplines that traditionally play important roles in science and engineering. The proactive approach focuses on the normal state, or "set point" of the software system, which is described very succinctly and embodied in a mathematical model. Then, the execution of the software system is measured and monitored to determine the degree to which the current activity of the system departs from the set point or "certified model" of the normal behavior. When a departure from normality is detected, adjustments are then made at the appropriate control points to bring the system back to the normal or certified state.

It is uncommon to address the notions of measurement and control of computing systems in the way we shall present here. It is, however, imperative to do so because the historic growth in software complexity demands it. Software systems have grown to exceed the complexity of what are otherwise considered to be very complicated engineering problems. Unfortunately, far too many software systems come into being without the same measurement and control rigor exercised by the science and engineering communities.

Measurement is a notion that goes beyond simple observation. Measurement requires an understanding of not only what is being measured, but also how the observation process is conducted. The effect the observation process has upon the measured system must be understood. Measurement often implies the application of statistical methods to account for systematic inaccuracies as well as noise. These principles apply equally well to computer hardware and software just as they do any artificial, natural or social system.

Closed-loop control is commonly used where it is important to maintain stability of a dynamic system. In a typical closed loop control system, there exists an autonomous controller that observes system

state and modifies system inputs to maintain stability in the presence of input perturbations. Computing system dynamics are best characterized in terms of hardware and software functionalities. Perturbations of computing system functionalities, such as those caused by security exploits, can be compensated for via the application of proper system measurement and closed loop control.

The technology that we have developed closely follows a process control methodology for use on software systems. The software is measured at certain probe points and these measurements are compared against a standard model for the nominal execution for the software system being monitored. If the software begins to drift from a predetermined set point, control may be taken from the executing software and turned over to a feedback control element that may terminate or modify the software system based on observed system activity.

As applied to the problem of computer security, an execution model, or certificate, for the nominal or certified program execution may be established through a certification process. When the software is subsequently placed in service, its activity may be continually monitored and compared against the model of nominal operation. This will insure that the software system is continually operating within the range of normal activity.

This is really the dual of the traditional software security approach. In the traditional approach, the focus is on the detection of abnormal activity. There are two problems with this approach. First, it is reactive. That is, it is possible to detect attacks once they have been seen and classified. Secondly, the state space of abnormal behavior is unbounded. There are many ways that a system could be compromised most of which are unknown until the vulnerability is exposed.

The method that we have developed is proactive. We first model normal program activity. It turns out that this activity is typically highly constrained. That is, the state space of normal behavior is very small relative to that of abnormal behavior so the model is very sensitive to abnormal perturbations. That model is then referenced when the program is executed in an unsecured environment. When significant measurement perturbations are observed, closed loop feedback is applied to bring the system back to normal.

Program Monitoring

In regards to the monitoring activity, there are two distinctly different approaches that can be taken. One, we may choose to monitor either the data stream as input to the program or the data stream as output from the program. In this approach the software is viewed as a black box whose internal characteristics are not known. Again, the problem with this approach is that the state space of the input data is very large. Two, we may chose to monitor the software activity as a process control problem. Clearly, the data induce different behavior patterns on the software system itself. In this view, the internal structure is visible as a gray box. Control points within the software are made visible to a monitoring system.

The actual monitoring process can be performed in one of two different ways. The software system itself can be modified to insert probe points in it. Alternatively, the execution of the software may be monitored through modifications to the hardware that is running the software.

Various points in the software may be instrumented to collect telemetry on the flow of program execution. Typically, this instrumentation will be implemented through the use of a call statement. Each call will transmit a unique number to a monitor function that will record the call number and then return control to the program.

The downside of the hardware monitoring function is that there is a relatively high initial cost. The monitor function must be integrated into new hardware elements. Given that there is no performance hit for the monitoring function and the fact that it is simply not possible to compromise the

monitor/security system, there are many mission/safety critical applications for which there is simply no other viable solution.

The best way to monitor the activity of software executing on a machine is to analyze the CPU bus traffic generated by that software. To perform this monitor function, a logic analyzer will be connected to the hardware bus to trap control events that are of interest. It is possible to trap, say, call instructions together with the call address for that instruction. With these data, an ancillary processor could then monitor transfer of control from one program module to another. Alternatively, it is possible to track branch instructions in a similar manner to monitor control flow within a program module.

The most important aspect of the hardware monitoring strategy from a security perspective is that the process is totally unobtrusive. It is not visible to the system being monitored. Its monitoring role is strictly observational. Thus, there is no performance hit for the monitor system. It operates on its own hardware system.

The hardware monitoring strategy has another very important feature. It is possible to monitor literally every executing instruction at machine speeds. There are no problems with latency. The hardware can literally detect abnormalities down to individual machine instructions.

There is a hybrid monitoring strategy that sits somewhere between the pure software and hardware monitoring strategies. In this approach, the compiler is modified to insert memory probes at the point of each call statement. These memory probes are then sensed on the control bus by a special hardware system component. We have implemented this strategy with our CML Hardware-enabled Monitoring (HMS) system.

The PCI HMS is a hybrid monitoring system. It is implemented on a PCI Express card. In this guise, the HMS system will monitor the PCI bus for specific write activities of the software running on the PC system itself. To implement this strategy, the software to be monitored by PCI HMS must be compiled with our modified GCC compiler. This version of the GCC compiler will place the necessary hooks in the software for the analytical engine. Modifications are also built into the Linux kernel task switcher. These modifications will permit HMS to monitor the transitions from one process to another.

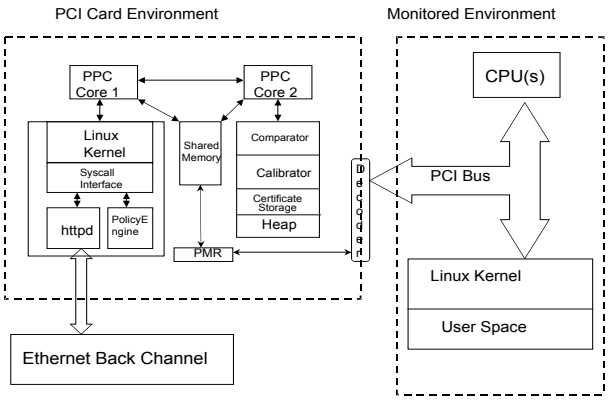

Figure 1. The PCI-HMS System

Ultimately each task or process that is to be monitored by the system will carry its own operational certificate. In this guise, as each process is loaded and readied for execution by the operating system, the certificate for its operation will be opened by the Analytical Engine. When it detects off-nominal activity, the Analytical Engine will return control to the Linux kernel via an interrupt. The corresponding interrupt service routine (ISR) will vector to the adaptive engine that will, in turn, decide what action is to be taken by the operating system.

The architectural overview of the PCI HMS system for Linux is shown in Figure 1. The system is comprised of two processing systems that are linked through the PCI bus. The analytical engine resides on the PCI card as do the certificates for the certified software systems. All of the processes in the user space run under the aegis of the PC Core. On this side of the system is the Linux kernel and all of the user processes. The monitor function resides entirely on the PCI card.

There are actually two separate computer systems that reside on the PCI card. The analytical engine resides in local memory and functions without an operating system on one of the two on-board

processors. The Web Interface system is a simple Linux kernel that runs on a second processor on the PCI card. Its sole function is to handle the communications between the policy engine that is running external to the PC environment and the adaptive engine that is running under the aegis of the PC host Linux kernel.

Communication between the two onboard CPUs is through a shared memory segment. This shared memory is also used for communicating between the policy engine and the adaptive engine.

The policy engine is the security administrator's (SA) interface to the system. This is a web-based system. As such the engine is actually run by the `httpd` server (Apache). It is through this system that the system security administrator may interact with the analytical engine. The SA will establish a security policy for the analytical engine. In that the analytical engine will operate at machine speeds, it must be empowered to take action when processes begin to diverge from their certified operation. The precise nature of the action to be taken is a matter of policy.

The HMS system is designed to monitor the execution of the Linux kernel on the host computer system. It may also be used to monitor any other executing software system running on the host machine. Functionally, there are two modes of operation of the HMS system. First, there is the certification mode. In this operational mode, a specified software system will be monitored by HMS to build a certified certificate of execution for that system.

The second mode of execution for the HMS system is the monitor mode. In this mode, the execution activity of a monitored software system will be validated against the software certificate. When departures are observed from the certified software behavior, the HMS system will transfer control to an embedded routine in the host Linux kernel known as the adaptive engine. The adaptive engine will then take the necessary steps to control the future execution of the software that is executing outside of its certified range.

The adaptive engine operates on the basis of a set of rules that are predetermined by a software security administrator. These rules are prepared through the interaction of the security administrator with a web based policy engine. The policy engine then downloads the security policy to the Linux kernel on the PCIe card that will, in turn, load these data onto the adaptive engine in the host computer that is responsible for implementing the security policy.

The analytical engine performs the actual monitoring function within the HMS system. There are two distinct roles that the analytical engine serves. First, it communicates with the PC Linux kernel to identify processes that are in execution. Second, it monitors the execution of those processes.

Measurement Domains

It is our thesis is that it is possible to measure software systems when they are running. When these software systems are running normally, the range of behavior that they will exhibit in terms of measurable characteristics of the program is highly constrained. When the systems are disturbed either intentionally or as a result of program failure, their behavior will change fairly dramatically. This change in behavior can be detected through the dynamic measurement of the executing software.

There are quite a number of different measures that can be taken on executing software systems. Some of these measures are at a fairly high level of granularity such as program call activity. Some measures are at a really low level of granularity such as program branching activity. The question is "What should be measured and at what level of granularity does this measurement have to occur?" The answer to this question is very simple. The measurement activity must be good enough to detect program anomalies 1) when they occur and 2) in sufficient time to arrest the mal-operation before it can do any damage. It is a simple engineering problem.

We deal with similar measurement problems on a daily basis. How good does a clock have to be? Well, a wristwatch that is accurate to plus or minus one minute is probably good enough for most people. A clock that will be used in the Global Positioning System much be accurate to many more decimal places for it to do its job.

For the level of resolution that we will need to monitor the activity of a system for security purposes, we will consider two different measurement activities at two distinct levels of measurement granularity. First, we may measure program activity within its traverse of its call tree. We will call these Between Module Measures. Then, we will consider the prospect of measuring program activity at a lower level of granularity by looking at Within Module Measures. At this level we would actually monitor branching activity with each module as it executes.

The first step in the adaptive control process will be to develop some notion of normal program activity. To do this, a software system must first be calibrated. This calibration process will involve observing the software execution in clean room conditions while an execution certificate is prepared for it. The precise nature of this certificate is determined by the set of attributes that are being measured. In the PCI HMS™ system, for example, we have chosen to monitor at the between module level of granularity. Experience and experimentation has shown that this level of granularity of measurement will easily identify abnormal program activity in a sufficiently timely fashion.

The amount of time necessary to perform this calibration step is a direct function of the entropy of the monitored software system. It is a key feature of most software applications that they are very low entropy applications, which in turn, means that the calibration phase of a typical application is also very short.

Continuous Monitoring

The actual measurement activity occurs within a software program called the analytical engine. There are two distinct ways that this system may be implemented. In systems based on software probes, the analytical engine will typically run as a function in the software space. On hybrid systems or hardware based monitoring systems, the analytical engine will reside on a separate processor that is actually performing the monitoring function.

The calibration functionality is a key component of the analytical engine. Its purpose is to monitor the nominal execution of a process and prepare a certificate representing the nominal operating characteristics of that process. At the conclusion of the calibration process, the analytical engine will store the certificate in local, possibly flash, memory.

There are three distinct software processes that operate within the analytical engine. The calibrator is used to build an execution certificate for any process that is to be subsequently monitored. The comparator is a software process that will monitor the activity of any process running on the host Linux side system that has been certified. The task switcher follows the context switching activity on Linux side. As each context switch occurs in the Linux kernel, an appropriate context switch must also be made on the analytical engine side. Each process executing on the Linux side has its own certificate or is in the process of calibration to build a certificate for that process. The actual execution certificates for all certified processes will reside in flash memory of the PCI card.

Any process that will be monitored by the PCI HMS system, for example, must first be calibrated. As part of this calibration process, the monitored process must be carefully exercised so that all possible nominal execution scenarios including all error handling conditions are exercised in a clean room scenario. Any program activity not observed during the calibration process will be, by definition, abnormal behavior.

The monitoring function is a vital one for the security environment of the system. In hardware base monitoring systems, the boot loader is typically modified to prevent the host OS kernel from booting until the analytical engine is enabled and fully operational. Once the OS on the host machine is fully operational, no process will be permitted to execute on the PC side unless it is either 1) in the process of being calibrated or 2) has be calibrated in the past.

Task Switching

At any point in time the operating system may be scheduling a very large number of processes for execution. This means that the bus activity may span multiple processes during any observation interval. As the operating system switches from one task to another, the analytical engine component of the system must be able to follow this switching process.

There are two ways that a new process can be started in Linux. A process may be forked or it may be executed. In the event of a process fork, the new process will have the same name as the previous process but the Process ID will be different. In addition, the entire adaptive engine environment including all of the local registers and the execution certificate must be replicated. In this event, the parent process will likely go quiescent. In which case, the state of the adaptive engine for the parent process must be captured and pushed into a backing store.

In the event that a new process is initiated via an execute, the new process will retain the same PID as the initiating process but the process name will be different. It is clear, however, that the state of the initiating process will no longer be needed. It may be totally discarded. What happens next depends heavily on the process status. If the new process is an exempt process, the analytical engine will simply monitor the PID shared memory location for evidence of a context switch.

The actual context switch is managed in the analytical engine by the task switcher. It will drive the activities of the calibrator and the comparator. Basically, the task switcher will read one entry at a time from the FIFO. It will then vector control to the appropriate routine. There are seven distinct context switch cases. They are shown in the table below.

There is one significant departure from this basic structure of task switching. This is related to the interrupt service on the Linux side. When an interrupt is detected, all subsequent calls will be stacked and unstacked until the stack empties, signifying that the interrupt service is complete. In that the interrupt service routines operate very deterministically, there is no need to certify their operation.

Process Switch	This represents the change from one active process to another. This context switch would also include the execution of the kernel itself.
Thread Switch	In this case the OS switches to an execution thread of an existing process
Initial Process Starts	In this case the OS initiates a new process with a new PID
Initial Thread Start	The OS task switcher initiates a new thread for an existing process
Thread Termination	The OS task switcher terminates the current thread a of a process
Termination of Active Process	The OS terminates the execution of an active PID
Fork of Current Process	The system now has two active processes of the same name but different PIDs.

Security Policy

From the standpoint of a measurement based security system, the notion of security policy becomes very well defined. Security policy for these systems has two components. First, critical thresholds must be established. It is clearly possible to measure the state of the system as it is currently executing against a standard model, as embodied by a set of program certificates. In this sense, it is possible to measure the distance between the standard model or certified model of program execution and the current activity of the system. Thus, the first component of the security policy will be a distance threshold. Once the activity of the software has been forced into a new operational mode that has crossed the pre-established threshold of nominal activity, control of the system will be transferred automatically to the adaptive software component.

This aspect of policy is analogous to the operation of the security gates that are employed at airport terminals. The sensitivity of these gates to detect metal may be adjusted up or down in relation to the perceived threat.

The second aspect of security policy in this environment relates to the action that is to be taken to adapt the system when the critical threshold for a given program has been crossed. This security system will operate at machine speeds. The system must, therefore, be equipped to deal with abnormalities, at least initially, without human intervention. In the near term, there are three alternatives. The offending process may be killed, it may have its priority reduced, or it may be suspended until its activities have been reviewed by a human security analyst.

The considerations associated with a compromised operating system or underlying hardware are, of course, different. It certainly is possible to halt the OS or reset the hardware, but to do so, to a large extent, effectively defeats the purpose of shielding the system from an attack. Furthermore, it doesn't make sense to kill or reduce the priority of every process affected by a low-level exploit since it is our strategic objective to maintain system operation under all possible conditions.
This is where the notion of software adaptation comes into play. Rather than simply changing the status of an executing process, here, the process is dynamically modified or otherwise repaired. With the application of system survivability analysis, it is possible to determine a policy that mirrors redundant copies of key OS components and hardware register settings. These can be rapidly patched into the system dynamically with much less interruption of critical system operation.

Social Security

Social engineering plays an important role in enabling attacks upon computer system infrastructure. Having the ability to quiescently measure and characterize process activities within the user context can provide an important tool in this regard. Calibrating system operation in the user context can provide critical computer installations an important edge for maintaining a trusted environment.

In our approach to software process monitoring, we are interested in the activity of the software system. We have designed a security coprocessor that monitors the running software system and measures its activity while it is running. In this context the data may be seen as stimuli for the program that exhibits some activity in response to each datum. Data in the normal range of user activity will induce normal activity on the software. Data outside of this range will induce different or unusual activity on the system that may be readily observed. In this approach the focus shifts from modeling and understanding the data space to which a program may be subjected, to the activity of the program in response to the data input.

The central issue here is that once we understand the notion that a program exhibits activity that can be measured, we can begin to assert control on the program execution. We can certify certain activities as nominal and reject activities that are outside of this range. We can do this in real time because we are monitoring the activity of the system in real time.

Social process control should be a major component of computer security. It is also a totally neglected component of computer security. It has long been accepted that data control in the form of encryption is a necessity to preserve the integrity of information flowing from one agency to another. Controlling access to system resources has also shown great value for imposing a security regime.

Access control has been used over time as a means of attaining some modicum of security. In the middle ages, castles were constructed to limit the access of marauding bands of itinerant soldiers to the populace of a region. These castles were effective if and only if they were sufficiently strong. This made them a nightmare for the occupants. The castles were cold, drafty and very restrictive in terms of the movement of their inhabitants. With the advent of the trebuchet and the cannon, even these imposing and uncomfortable structures became obsolete. There are other real good examples of failed access control in the Great Wall of China or the Maginot Line constructed before World War II to defend France against the Germans. Access control is a deterrent but not a solution.

Another cornerstone of computer security is encryption. This technology has been with society almost since the inception of the written word. It remains an extremely valuable security tool.
The missing piece of the computer security paradigm is that of social process control. It is simply not possible to build systems that are free from vulnerabilities in this regard. It has long been understood that it is simply not possible to engineer a defect free bridge or building. Mistakes and human error are a fact of the construction process. Thus, defect free software should never be assumed as a part of software development. Normal users of a system do not exploit vulnerabilities. Only the deliberate misuse of systems will exploit vulnerabilities. This misuse can be detected and acted on immediately, if the systems are being monitored in real time. The problem is not the fact that there are vulnerabilities in the software. The problem is that the software is not monitored and the vulnerabilities open the door for exploits.

Closed Loop Control

It is one thing to identify the fact that a program has begun to depart from its normal or certified activity. It is quite another to move to control the process. The monitoring system will perform the detection role. At the point that the software under observation crosses the critical threshold for execution, the monitor system will generate an interrupt. This interrupt will be handled by a special interrupt service routine that we call the adaptive control system ACS. Once the adaptive control system has been called, it will grab the process control information of the offending software process from the monitor system. At this point the ACS will make a determination of the nature of the action that is to be taken on the offending process. This is an aspect of security policy as discussed earlier. The ACS will then communicate the necessary action to the operating system kernel to handle the process in accordance with the established security policy.

To this point the ACS has been cast into a very simple role. It treats each of the monitored processes as a single entity. This entity can be killed or its access to system resources may be limited. That is all. It is possible, however, for the ACS to actually alter the executing process so that it may continue to execute.

The internal structure of the executing process is known to the AE. It is part of the certificate. Let us suppose, for example, that the AE is monitoring the call tree structure of a process for abnormal activity. It might well be that this abnormal activity consists of a new call sub tree being grafted onto the certified execution call tree. With this knowledge in hand, it would be possible for the ACS to modify the executing code to prune the offending call sub tree from the execution path and then return control to the software system at the point where the call occurred. In this hypothetical scenario, the ACS could execute yet another aspect of security policy. That is, it could modify the executing software to eliminate the abnormal activity and then let it continue to execute.

Conclusion

Our measurement-based technology will enhance and harden application software, OS software and the underlying hardware from malicious alteration and assaults. In that the monitoring functionality is implemented in a hardware co-processor it is simply not possible for external program activity to alter the monitoring system. In that the monitoring function is an external activity, the knowledge of its activity and design will not provide comfort to an enemy. The monitoring activity operates in real time. It can detect even the most rapid assaults such as buffer overflows and take action to ameliorate or eliminate their negative consequences.

To this end, an important problem that we have solved is the mathematical representation of normal system activity. The state space of normal activity as projected onto the process call tree is compact and well defined. On the other hand, the state space of abnormal activity is unbounded and thus easier to identify as being inconsistent with a certified call tree.

Many of the problems that we must now confront in the arena of computer security relate directly to the lack of engineering discipline of the designers and developers of these software-based systems. Perhaps the most dangerous aspect of this simplicity is the notion that it is of value to classify and catalog specific faults (vulnerabilities) in the many software systems in use. This is neither useful nor productive. As long as there are humans in the software development process, there will be faults (vulnerabilities) in the code. It is an anathema to the notion of engineering discipline to believe that it is either necessary or possible to remove all of these problems from every piece of software. The vulnerabilities are not the problem. The problem is that the software is simply running out of control. The solution to the problem is to bring the software under process control just as we would do for any complex dynamic system.

Apart from the specifics, the paradigm shift described here changes the name of the security game. Security strategies should move from a reactive view of a world to a proactive one. One that maintains dynamic control over software execution rather than manually adapting ex-post-facto, or worse yet, simply running open loop with no feedback at all.

The most threatening attacks to our national computational infrastructure are those that we haven't seen yet. This, we believe, is the most pressing problem confronting computing security in our nation. These attacks will certainly come as a surprise. Our approach addresses this problem by recognizing the divergence from normal system activity. This makes it possible to identify abnormal activity whether it is novel, well known, or a new form of attack. It is our believe, then, that software process control is vital to our national security interests.