An Introduction to Opus HMS

Computer Measurement Laboratory, Inc.
info@cmlab.biz

The lack of basic principles of engineering design and discipline in software development has led to some very serious problems in the computer security arena. A very good example of this is the Supervisory Control and Data Acquisition (SCADA) software systems. These systems are the heart of modern power distribution systems. They provide real time control and data management. It is interesting to note that every aspect of a hardware system that is under control of the SCADA architecture is fully instrumented and the operation of the monitored system is totally under management of the control system. However, the *software* that controls the entire power grid is essentially open loop and uncontrolled. It would be unthinkable to build a complex hardware system without embedding monitors for process control throughout the whole architecture. Ironically, the backbone of modern control systems, SCADA for example, is almost entirely dependent on a software system that is running out of control

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 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.

The lack of monitoring of complex software systems opens the door for problems in both security and reliability. From a security perspective, the software may be compromised and made to perform the bidding of a hostile agent. From a reliability perspective, the software may begin to execute software components that are not thoroughly tested. The end result of either of these two scenarios is the same. The nominal operation of the software system will be compromised. What is worse, because the processes that constitute the software system are not being monitored, the integrity of the hardware component is most certainly compromised as well.

The foundation of our current work is the design and development of software control systems. These systems are built to provide engineering integrity for executing software systems. It is our philosophy

that software systems must be subject to the same rigorous scrutiny during their operation as would any mission critical or safety critical hardware system.

A most pressing problem looms in the immediate future. In an information warfare scenario, the enemy will deploy an attack that has never been seen before. This attack will completely debilitate both defensive and offensive software capabilities. Unfortunately, the state space of possible attacks is at once large and unknown for a complex software system. If a system is monitored for nominal operation, then it is possible to recognize instantly any abnormal activity. Once the abnormality is recognized, it is then possible to restore the system to a known certified state. Our project's major innovation is a comprehensive approach for tolerating security violations in mission/safety critical software systems by closing the software feedback control loop. To the best of our knowledge, this project is the first project that uniformly applies the real-time measurement paradigm to the reliability and security of mission critical software.

A control policy relates a measured deviation in process control flow to a modification of process behavior. A simple policy that indicates process termination when measured deviation exceeds a pre-determined threshold implements a comprehensive dual of traditional signature analysis. It allows functionality to continue for all measurements that lie within an acceptable distance from the expected behavior. In contrast, signature analysis denies functionality to a limited set of points enumerated outside the acceptable distance. In a more controlled environment, a policy might reserve termination only for the most egregious deviations, and it may even be inhomogeneous: i.e. sensitive to the current program execution context. For smaller more innocuous deviations, administration could decide that simple reductions in program priority would suffice.

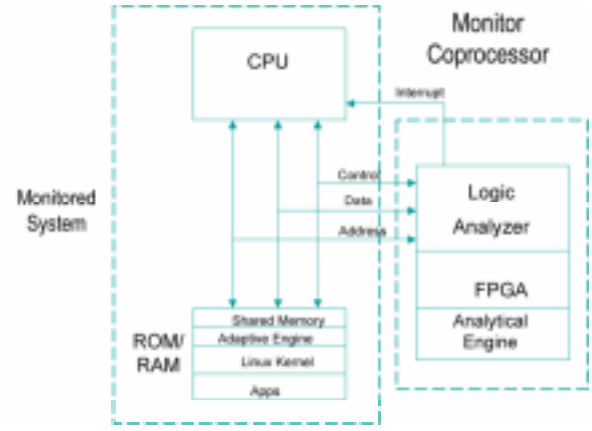

Figure 1.

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 measure its activity while it is running. The basic architecture of this hardware monitor is shown in Figure 1. 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 trying to model and understand 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.

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 performs the detection role. At the point that the software under observation crosses the critical threshold for execution, the monitor system generates an interrupt. This interrupt is 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 alter the executing process so that it may continue to execute. If a section of code has been compromised, it is possible to map around that code at run time. Alternatively, a mission code segment may be duplexed and run in parallel threads. If one thread fails, the ACS can simple eliminate that thread and continue with an uninterrupted execution of the program.

Process control, then, 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. Access control, the, 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 process monitoring. It is simply not possible to build systems that are free from vulnerabilities. Engineers have long understood that it is simply not possible to build a defect free bridge or building. Mistakes and human error are a fact of the construction process. Thus, defect free software should never be an objective 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.

In response to the need for a software process control system, we have developed the CML Hardware-enabled Monitoring of Software, HMS, under the aegis of a contract from the Office of the Secretary of Defense. The primary objective of the HMS project is to create the infrastructure for an autonomic kernel protection system and then productize this infrastructure. There are essentially two types of attacks that can be made on an operating software system: those that have been deployed against existing systems and those that have never been seen before. Current approaches to the kernel defense rely on the knowledge of the attack signature. That is, there are recognizable symptoms of the effect of the attack on the kernel. Clearly, this approach renders most systems vulnerable to the very type of attack that is most likely in an information warfare scenario.

We have implemented the HMS architecture on a PCI express card for use in Linux based PCs. In its security role, the Opus HMS system is used as a software process controller. It measures the continuing activity of the software running on a host machine and it uses the telemetry that it gathers from this process to compare against a model of standard or certified software activity. At the point that a software system begins to deviate from its standard model of operation, the Opus HMS system may be empowered to alter the execution of the offending process.

Our approach is a dual of the traditional security paradigm. We are able to model the normal activity of a system and detect departures from that normal model. This methodology permits each implementation of a software system to be calibrated for operation in a particular environment. Once a departure from the nominal model of software activity has been detected, the offending activity may be arrested before it has compromised the system. The problem that we have solved is the mathematical representation of normal system activity. The state space of normal activity for large number of mission critical systems is very small and well defined. On the other hand, the state space of abnormal activity is unbounded. It can never really be known.

There are four fundamental components to the HMS system that we have developed to implement this approach. These are the Logic Analyzer, the Analytical Engine, the Policy Engine and the Adaptive Engine. The Logic Analyzer consists of the set of monitor operations that will be performed by the adjunct CPU running on a separate bus architecture. The Analytical Engine will provide a dynamic analysis of the currently executing process to insure that it is executing within certified limits. The Calibration Engine will generate the nominal execution certificate for a process that is to be monitored subsequently by the HMS system. The Policy Engine will manage the rule sets for determining the actions that are to be taken when a process is found to be executing outside the bounds of its nominal operation. The Adaptive Engine will act to alter the execution environment when a process is found to be out of bounds. The action may include, for example, suspending the executing process, termination of an executing process, or modifying an executing process to eliminate suspected malware.

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 transferred to 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 will be vectored to the Policy Engine that will, in turn, decide what action is to be taken by the operating system.

In its simplest form, the HMS system is comprised of two central processing systems that are linked with a control interface and also with a shared memory component as is shown in Figure 2. All of the processes in the user space run under the aegis of the main CPU. On this side of the system is the Linux kernel and all of the user processes. The analytical engine resides entirely on the second CPU system. There is, however, no operating system on the analytical engine side. There is an element of shared memory that is visible to both of the CPUs. The two systems communicate by passing information through the shared memory element. There is also a control interface between the two CPU cores.

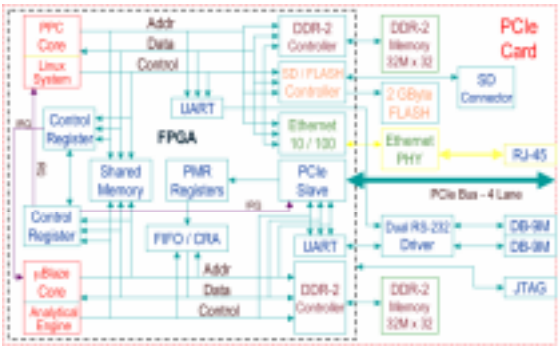

Figure 2.

It is the role of the adaptive engine to modify the execution environment of a process to allow the adapted process to continue to function in a safe manner. For the purposes of the Opus HMS architecture, the role of the adaptive engine is fairly simple. This engine is embedded in the Linux kernel as an interrupt service. It simply implements the user's policy as determined by the user's interaction with the policy engine. In this guise, a process may 1) be terminated in the event that it deviates significantly from its certified execution profile, 2) it may be niced to a lower priority to slow it down and 3) it may be suspended pending action by the security administrator.

The important issue, here, is that the Opus HMS system is in complete control of the host Linux environment. HMS is monitoring the activity of all executing processes included the host Linux kernel. The kernel will not launch without authorization from the Opus HMS card, nor will any uncertified process be permitted to execute in this environment.

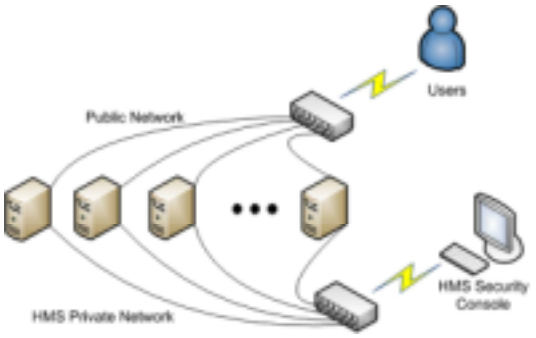

Figure 3.

The Opus HMS software process control system is administered through a secure communication channel. The Ethernet port on the Opus HMS card is designed for the specific use of the HMS system. It is not intended to be connected to the public Internet. The HMS system is designed to function with its own secure backchannel communication between each Opus HMS system and the Security Console as is shown in Figure 3.

This technology will enhance and harden the kernel software and other executing software systems from malicious alteration and assaults from the communication network. In that the monitoring functionality

is implemented out of band in a hardware co-processor it is simply not possible for an external program activity to access 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 adversary.

It is our security thesis that software vulnerabilities are not the fundamental issue. It is very expensive and impractical to try to build a vulnerability free system. No civil engineer would ever expect to build a perfect bridge. An engineer would, instead, expect to build a bridge within practical tolerances and design in compensation for environmental variation. The problem in software, then, is not the vulnerabilities. The problem is that the vulnerabilities get expressed because no one is watching and controlling. We choose to trust no software. We monitor and control everything in the software execution environment.

Apart from the specifics, the paradigm shift described here is intended to change 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 focus should shift from the stygian task of classification of malware to the process control approach. It is not really important to know and to name an exploit. The set of potential exploits for a modern complex software system is huge and possibly unknown. It is simply a problem that cannot be solved.

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.