Multivariant Execution Environment against Code Injection Attacks

R.Deepa[1], P.Thenmozhi[2], L. John Ritchie Immanuel[3]
[123]Asst. Prof, Dept. of CSE, M.I.E.T Engineering College.

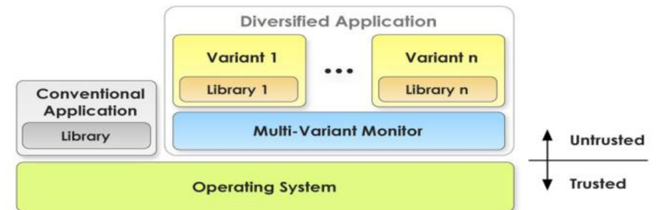
Abstract- The number and complexity of attacks on computer systems are increasing. This growth necessitates proper defense mechanisms. Intrusion detection systems play an important role in detecting and disrupting attacks before they can compromise software. Multivariant execution is an intrusion detection mechanism that executes several slightly different versions, called variants, of the same program in lockstep. The variants are built to have identical behavior under normal execution conditions. However, when the variants are under attack, there are detectable differences in their execution behavior. At runtime, a monitor compares the behavior of the variants at certain synchronization points and raises an alarm when a discrepancy is detected. We present a monitoring mechanism that does not need any kernel privileges to supervise the variants. Many sources of inconsistencies, including asynchronous signals and scheduling of multithreaded or multiprocess applications, can cause divergence in behavior of variants. These divergences cause false alarms. We provide solutions to remove these false alarms. Our experiments show that the multivariant execution technique is effective in detecting and preventing code injection attacks.

I. INTRODUCTION

Security vulnerabilities in software have been a significant problem for the computer industry for decades. As a result, the challenge of finding mechanisms to detect and remove vulnerabilities persists. Modern static analysis tools are capable of finding many varieties of programming errors, but a lack of runtime information limits their abilities. Dynamic and runtime tools are often not effective because they lack a baseline to use for detection. Also, the performance overhead of sophisticated algorithms used by such runtime tools is often prohibitively high in some production systems.

Multivariant code execution is a run time monitoring technique that prevents system damage resulting from malicious code execution and addresses the above problems with dynamic detection tools. Multivariant execution protects against malicious code execution attacks by running two or more slightly different versions of the same program, called variants, in lockstep. At defined synchronization points, the variants' behavior is compared against each other. Divergence among the behavior is an indication of an anomaly and raises an alarm.

II. THE MULTIVARIANT MONITOR



Multivariant execution is a monitoring mechanism that controls the states of the variants being executed and verifies that the variants are complying to defined rules. A monitoring agent, or monitor, is responsible for performing the checks and ensuring that no program instance has been corrupted. This can be achieved at varying granularities, ranging from a coarse-grained approach that only checks that the final output of each variant is identical, all the way to a checkpointing mechanism that compares each executed instruction. The granularity of monitoring does not impact what can be detected, but it determines how soon an attack can be caught.

This paper use a monitoring technique that synchronizes program instances at the granularity of system calls. This rationale for using this granularity is that the semantics of modern operating systems prevent processes from having any outside effect unless they invoke a system call. Thus, injected malicious code cannot damage the system without invoking a system call. Moreover, coarse-grained monitoring has lower overhead compared to fine-grained monitoring, as it reduces the number of comparisons and synchronization points.

The monitor runs completely in user space. The monitor is a process invoked by a user and receives the paths of the executables that must be run as variants. The monitor creates one child process per variant and starts executing all of them. It allows the variants to run without interruption as long as they do not require data or resources outside of their process spaces. Whenever a variant issues a system call, the request is intercepted by the monitor and the variant is suspended. The monitor then attempts to synchronize the system call with the other variants. All variants need to make the exact same system call with equivalent arguments within a small time window. The invocation of a system call is the synchronization point in this technique.

Note that argument equivalence does not necessarily mean that argument values are identical. When an argument is a pointer to a buffer, the contents of the buffers are compared

and the monitor expects them to be the same, whereas the pointers themselves can be different. Nonpointer arguments are considered equivalent only when they are identical.

In a more formal way, the monitor determines whether the variants are in complying state based on the following rules. If p1 to pn are the variants of the same program p, they are considered to be in conforming states if at every synchronization point the following conditions hold:

1. $\forall_{si,sj} \varepsilon$ S: si = sj where S={s1,s2,….sn} is the set of all invoked system calls at the synchronization point and si is the system call invoked by variant pi.

2. $\forall$aij, aik ε A : aij ≡ aik where A ={a11, a12, . . . , amn} is the set of all the system call arguments encountered at the synchronization point, aij is the ith argument of the system call invoked by pj, and m is the number of arguments used by the encountered system call. A is empty for system calls that do not take arguments. When an argument is a pointer to a buffer, the contents of the buffers are compared and the monitor expects them to be the same, whereas the pointers (actual arguments) themselves can be different. Formally, the argument equivalence operator is defined as

$$a \equiv b \leftrightarrow \{ \text{ if type} \neq \text{buffer} : a = b \quad \text{else:content}(a)= \text{content}(b)$$

with type being the argument type expected for this argument of the system call. The content of a buffer is the set of all bytes contained in it

$$\text{content}(a) :=\{a[0]\ldots.\ a[\text{size}(a) - 1]\}$$

with the size function returning the first occurrence of a zero byte in the buffer in case of a zeroterminated buffer, or the value of a system call argument used to indicate the size of the buffer in case of buffers with explicit size specification.

3. $\forall$ti ε T : ti - ts ≤ ω where T ={t1, t2, . . . ; tn} is the set of times when the monitor intercepts system calls, ti is the time that system call si is intercepted by the monitor, and ts is the time that the synchronization point is triggered. This is the time of the first system call encountered at this synchronization point. ω is the maximum amount of wallclock time that the monitor waits for a variant. ω is specified in the policy and depends on the application and hardware. As an example, the ratio of the number of variants to the number of available processor cores can increase or decrease ω.

If any of these conditions is not met, an alarm is raised and the monitor takes an appropriate action based on a configurable policy. Terminate and restart all the variants, but other policies such as terminating only the nonconforming ones, based on majority voting, are possible.

III. SYSTEM CALL EXECUTION

An MVEE and all the variants executed in this system must act as if only one variant was running conventionally on the host operating system. The monitor is responsible for providing this behavior by running certain system calls on behalf of the variants and providing the variants with the results.

Depending on the effects of these system calls and their results, we specified which ones can be executed by the variants and which ones must be run by the monitor. The decision is based on the following parameters:

- System calls that change the state of the system are executed by the monitor and the results are copied to the variants. For example, a system call that creates a file on the system must be executed once by the monitor and the variants are not allowed to run it.
- Non-state-changing system calls that return volatile results must also be executed by the monitor, and the variants must receive identical results of the system call. For example, reading the system time (gettimeofday) must be performed by the monitor and the variants only receive the results. This is necessary to keep the variants in conforming states in the course of execution and to prevent false positives.
- Non-state-changing system calls that produce immutable results can be executed by the variants. For example, uname, which returns information about the operating system, is executed by all the variants.

These are only general rules for system call execution. Some system calls, such as chdir, must be executed by all the variants and also by the monitor. The monitor needs to run this system call to synchronize its working directory with that of the variants. This is required because the variants may later perform a file operation that is intercepted and executed by the monitor, but they may not provide the full path of the file.

The system call write must sometimes be executed by the monitor and sometimes by the variants. When the variants read input data, the monitor intercepts the input, and then sends identical copies of the data to all the variants. This is not only required to mimic the behavior of a single application, but it is also essential to prevent attackers from compromising one variant at a time.

File, socket, and standard I/O operations are performed by the monitor and the variants only receive the results. When a file is opened for writing, for example, the monitor is the only process that opens the file and sets the registers of the variants so that it appears to them that they succeeded in opening the file. All subsequent operations on such a file are performed by the monitor and the variants are just recipients of the results. This method would fail if the variants tried to map a file to their memory spaces using mmap, because the file descriptor received from the monitor was not actually opened in their contexts and, hence, mmap would return an error. This would cause a major restriction because shared libraries are mapped using this approach. Therefore, we allow the variants to open files locally if requested to be opened read only. Mapping shared libraries is allowed, but mapping a file opened for writing fails. However, mmap is rarely used in this manner.

When the mmap system call is used to map a file into the address space of a process, reads and writes to the mapped memory space are equivalent to reads and writes to the file, and can be performed without calling any system call. This could allow an attacker to take control over one variant and compromise the other variants using shared memory. To prevent this vulnerability,wedeny anymmap request that can create potential communication routes between the variants and only allow MAP_ANONYMOUS and MAP_PRIVATE. MAP_ SHARED is allowed only with read-only permission. In practice, this does not seem to be a significant limitation for most applications.

Variants are allowed to create anonymous pipes, but all data written to the pipes are checked by the monitor and must conform to the monitoring rules. Named pipes are created and operated by the monitor and the variants just receive the results.

IV. SCHEDULING

Scheduling of child processes or threads created by the variants can cause the monitor to observe different sequences of system calls and raise a false alarm. To prevent this situation, corresponding variants must be synchronized to each other. Suppose p1 and p2 are the main variants, and p1_1 is p1's child and p2_1 is p2's child. p1 and p2 must be synchronized to each other and p1_1 and p2_1 must also be synchronized to each other. We may choose to use a single monitor to supervise the variants and their children or we can use several monitors to do so.Using a single monitor can cause unnecessary delays in responding to their requests. Suppose p1 and p2 invoke a system call whose arguments take a large amount of time to compare. Just after the system call invocation and while the monitor is busy comparing the arguments, p1_1 and p2_1 invoke a system call that could be quickly checked by the monitor, but since the monitor is busy, the requests of the children cannot be processed immediately and they have to wait for the monitor to finish its first task.

Simple solution is to spawn a new monitoring thread for each set of new child processes or threads. This is done by the monitor responsible for the parent variants whenever the variants create new child processes or threads. Monitoring of the newly created children is handed over to the new monitor.

As mentioned before, we use ptrace to synchronize the variants. Unfortunately, ptrace is not designed to be used in a multithreaded debugger. As a result, handing the control of the new children over to a new monitor is difficult. We let the parent monitor start monitoring the new child variants until they invoke the first system call. After this point, we create a new monitoring thread and let the new thread take the control of the new variants.

V. SYNCHRONOUS SIGNAL DELIVERY

Handling asynchronous signals is one of the major challenges in multivariant execution, as it can cause the variants to execute different sequences of system calls. For example, assume variant p1 receives a signal and starts executing its handler. p1's signal handler then invokes system call s8, causing the monitor to wait for the same system call from p2. Meanwhile, variant p2 has not received the signal and calls system call s1 in its normal code flow. This behavior is considered a discrepancy and raises a false alarm in the system.

A possible solution is to deliver signals synchronously only at synchronization points, i.e., at system calls. The problem with this approach, however, is that CPU-intensive applications may not invoke any system call for a long period of time. Empirical results show that this technique adds 0.5 millisecond delay in delivering signals, while delivering signals at system calls could cause hundreds of milliseconds of delay in CPU-intensive applications. Such a long delay might not be acceptable for certain types of signals, such as timer signals, and could also reduce responsiveness of certain applications.

We provide a solution that is not based on delivering signals at system calls. Our solution benefits from the fact that whenever a signal is sent to a variant, the operating system pauses the variant and notifies the monitor. The monitor can either deliver the signal to the variant, or save it and ignore it for now.

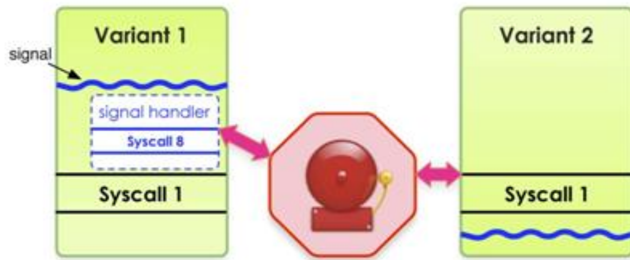
The monitor immediately delivers signals that terminate program execution, such as SIGTERM, and signals generated by CPU exceptions, such as SIGSEGV. If the CPU exception is caused by the normal flow of an application, it must appear in all the variants and, therefore, all of them receive it in the same execution state. Hence, the signal is automatically delivered to all the variants in the same state and delivering the signals immediately does not cause false alarms even if variants use user-defined signal handlers for the exceptions. If the exception is raised only in one or more variants but not all of them, immediate signal delivery causes an alarm in the system. This is a true alarm because it is an actual divergence in the behavior of the variants.

Signals that do not terminate program execution and are not caused by CPU exceptions are delivered to all the variants synchronously, meaning that signals are delivered to all of them either before or after a synchronization point, i.e., a system call, but not necessarily at the synchronization point. In other words, if we call the time span between any two consecutive system call invocation a "signal time frame," our algorithm guarantees that a signal is delivered to all the variants in the same signal time frame.

This algorithm postpones delivery of a signal until at least half of the variants receive the signal. At such a point, the signal is delivered to all the variants in the current signal time frame.

Variants that have not received the signal at such a point and have invoked a system call are forced to skip the system call and spin wait for the signal. The skipped system call is later restored and the variants run them. The subsequent section provides more details about the algorithm.

We use majority voting to determine when to deliver signals and also to find noncompliant variants. Using majority voting in signal delivery works well in multivariant execution systems that terminate all variants upon detection of one or more noncompliant variants. However, as explained in Section 2, terminating only noncompliant variants and continuing with the compliant majority cannot always guarantee correct results.



The synchronous signal delivery mechanism guarantees that the same sequence of system calls is observed in all the variants. However, if a signal handler invokes a system call and passes a frequently changing value from the program context to the system call, a false alarm may still be raised. A frequently changing value is a value that changes more than once between two system call invocations.

VI. CONCLUSIONS

A multivariant execution environment runs multiple versions of a program simultaneously and monitors their behavior. Discrepancy in behavior of the variants is an indication of an attack. Using this technique, this paper prevents exploitation of vulnerabilities at runtime. It is complementary to other methods that remove vulnerabilities, such as static analysis. Instead of finding and removing the vulnerabilities, this method accepts the inevitable existence of vulnerabilities and prevents their exploitations. A major advantage of this approach is that it enables to detect and prevent a wide range of threats, including “zero-day” attacks.

Many everyday applications are mostly sequential in nature. At the same time, automatic parallelization techniques are not yet effective enough on such workloads. Even in parallel applications, such as webservers, limited I/O bandwidth prevents from putting all available processing resources into service. As a result, parallel processors in today’s computers are often partially idle. By running programs in MVEEs on such multicore processors, this paper put the parallel hardware in good use and make the programs much more resilient against code injection attacks.

VII. REFERENCES

[1]. B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz. (2010) on the effectiveness of multi-variant program execution for vulnerability detection and prevention. In International Workshop on Security Measurements and Metrics (MetriSec).

[2]. B. Salamat, T. Jackson, A. Gal, and M. Franz. (2009) Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In Proceedings of the European Conference on Computer Systems, pages 33–46. ACM Press.

[3]. B. Salamat, C. Wimmer, and M. Franz. (2009) Synchronous signal delivery in a multi-variant intrusion detection system. Technical report, School of Information and Computer Sciences, University of California, Irvine.

[4]. B. Salamat, A. Gal, and M. Franz. (2008) Reverse stack execution in a multi-variant execution environment. In Workshop on Compiler and Architectural Techniques for Application Reliability and Security.

[5]. D. Evans, B. Cox, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser (2006) “N-Variant Systems: A Secretless Framework for Security through Diversity,” Proc. USENIX Security Symp., pp. 105-120.

AUTHOR PROFILE

Ms. R Deepa is working as an Asst. Professor in M.I.E.T Engg College,Trichy. She has done undergraduate degree in Computer Science and Engineering from Ann University, Chennai in 2006, and completed her postgraduate in Computer Science and Engineering from Anna University, Trichy in 2013. She has 3 years of teaching experience. Her areas of interest include Data Structures, Database and Computer Networks. She has a number of research papers in the field of Computer Networks.

Ms. P Thenmozhi is working as an Asst. Professor in M.I.E.T Engg College,Trichy. She has done undergraduate and postgraduate degree in Computer Science and Engineering from Anna University, Trichy. She has 3 years of teaching experience. Her areas of interest include Artificial Intelligence and Computer Networks. She has a number of research papers in the field of Computer Networks.

Mr. L John Richie Immanuel is working as an Asst. Professor in M.I.E.T Engg College, Trichy. He has done undergraduate and postgraduate in Computer Science and Engineering from Anna University, Chennai. He has 1 year of teaching experience. His area of interest includes Web Technology and Computer Networks.