Asian Journal of Computer Science and Technology
ISSN: 2249-0701 (P) Vol.1 No.1, 2012, pp.43-46

DOI: https://doi.org/10.51983/ajcst-2012.1.1.1672

# RDACIA: Runtime Defence Against Code Injection Attack Using N-Variant Approach

**K.A. Sheik Mydeen and V. Bala Murugan**

*Department of Computer Science and Engineering, Mohamed Sathak Engineering College, Kilakkarai, Tamil Nadu, India*

E-mail: mydeen.kas@gmail.com

***Abstract* - Software vulnerabilities have been a major threat for decades. Security vulnerabilities in software permit attackers to compromise and misuse computer systems for various malicious purposes. Intrusion detection systems have an important role in detecting and disrupting attacks before they can compromise software. Multi- variant execution is an intrusion detection mechanism that executes several slightly different versions or 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 run time, 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. As a result, the monitor runs entirely in user space. Our experiments show that the multi-variant execution technique is effective in detecting and preventing code injection attacks.**

***Keywords* – Code injection attack, malicious attack, n-variant execution, multi-variant execution, software fault tolerant**

## I. Introduction

Code injection is the general name for a lot of types of attacks which depend on inserting code, which is interpreted by the application. Such an attack may be performed by adding strings of characters into a cookie or an argument values in the URI. The concept of Code injection is to add malicious code into an application which then will be executed. Added code is a part of the application itself. It is not external code which is executed as like he command injection. Intrusion detection systems play an important role in detecting and disrupting attacks before they can compromise software applications. Multi-variant execution is an intrusion detection mechanism that executes several slightly different versions called variants of the same program in lock step. We provide reasons why certain types of applications suffer from higher performance degradation in a multi variant environment.

### *A. Problem Definition*

Multi-variant code execution is a run-time monitoring technique that prevents malicious code execution and addresses the problems mentioned above. Vulnerabilities that allow the injection of malicious code are among the most dangerous forms of security flaws since they allow attackers to gain complete control over the targeted system. Multi- variant execution protects against malicious code execution attacks by running two or more slightly different variants of the same program in lock step. At certain synchronization points, their behavior is compared against each other. Divergence among the behavior of the variants is an indication of an anomaly in the system and raises an alarm. An obvious drawback of multi-variant execution is the extra processing overhead, since at lease two variants of the same program must be executed in lockstep to provide the benefits mentioned above. Our experimental results show that extra computational overhead imposed by multi-variant execution is in the range afforded by most security sensitive applications where performance is not the first priority, such as government and banking software. Our proposed architecture allows running conventional applications without engaging the MVEE (see Fig. 1).

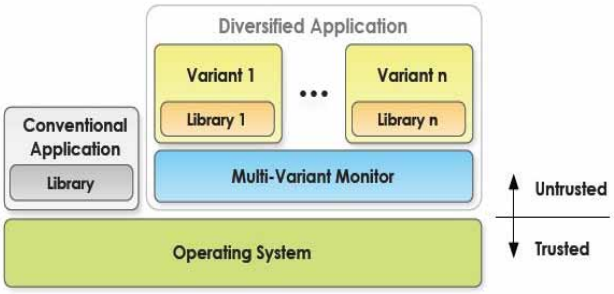


Fig.1 The proposed architecture

Thus, normal applications may run conventionally on he system and in parallel with security sensitive applications which are executed on top of the MVEE.

## II. Module Implementation

One of the most common forms of security attacks involves exploiting a vulnerability to inject malicious code into an executing application and then cause the injected code to be executed. Adaptive software has attracted much research interest in recent years. Two key features of adaptive software are (1) the ability to monitor its own execution and (2) the ability to reconfigure itself based on the result of runtime monitoring. Self adaptation is essential to improve system survivability for a range of applications from safety critical embedded software to mission-critical web services that shall be resilient to malicious attacks

### *A. Multi Variant Code Executions*

It is a runtime monitoring technique that prevents system damage resulting from malicious code execution and address

the above problems with dynamic detection tools. Multi-variant execution protects against malicious code execution attacks by running two or more slightly different versions of the same program, called variants in the lock step. 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. Unlike many previously proposed techniques to prevent malicious code execution that use random and/or secret keys in order to prevent attacks, a multi-variant execution is a secret-less system. It is designed on the assumption that program variants have identical behavior under normal execution conditions but their behavior differs under abnormal conditions. Therefore the choice is what to vary defines which class of attacks can be stopped and which vulnerabilities still can be exploited (False negatives). It is important that every variant be fed identical copies of each input from the system simultaneously. This design makes it difficult for an attacker to send individual malicious inputs to different variants and compromise them one at a time. If the variants are chosen properly, a malicious input to one variant causes collateral damage in some of the other variants, causing them deviate from each other. The deviation is then detected by a monitoring agent that enforces a security policy and raises an alarm.

*B. Multi-Variant Monitor*

Multi-variant execution is a monitoring mechanism that controls states of the variants being executed and verifies that the variants are complying with the earned 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 potentially hardware-assisted check pointing mechanism that compares each executed instruction to ensure that the variants execute semantically equivalent instructions in lockstep. The granularity of monitoring does not impact what can be detected, but it determines how soon an attack can be caught. We use a monitoring technique that synchronizes program instances at the granularity of system calls. Our rational for using this granularity is that the semantics of modern operating systems prevents 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. As mentioned before, our 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 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 called a synchronization point in our 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. Non-pointer arguments are considered equivalent only when they are identical.

*C. System Call Execution*

A multi-variant environment and all the variants executed in this system must act as any one of the variants running conventionally on the host operating system. The monitor is responsible for providing this characteristic by running certain system calls on behalf of the variants and providing the variants with the results.

We have examined the system calls of the host operating system one by one and considered types and the number of possible arguments that can be passed to them. Depending on the effects of these system calls and their results, we have specified which ones can be executed by the variants and which ones should be run by the monitor. The decision as to who should run the system calls has generally been made 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 should not be allowed to run it.
- Non-state changing system calls that return non-immutable 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 preventing false- positives.
- Non-state changing system calls that produce immutable results allowed to be executed by the variants. For example, uname that returns information about the operating system is executed by all the variants.

These are general rules for system call execution, but running system calls are more complicated in practice and the decision as to who should run a system call sometimes need more investigations.

## III. Inconsistencies And Non- Determinism

Internal conditions and behavior of the system that runs the variants, as well as system events, can cause divergence in behavior of the variants. These divergences cause the monitor to raise false alarms and interrupt execution of the variants. There are several sources of inconsistencies among the variants that can cause false positives in multi-variant execution. Scheduling of child processes and threads, asynchronous signals, file descriptors, process IDs, time, and random numbers must be handled properly to prevent false positives.

### *A. 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. In a multi-threaded monitor, any monitoring thread may receive signals or events encountered in any traced process. This means that a monitoring thread can receive signals raised for the processes monitored by other monitoring threads. We use *wait4* to tackle this problem. wait4 allows a monitoring thread to wait for a specific process whose PID is passed to *wait4*. Using this wait function, a monitoring thread receives notifications of signals or system calls only for the processes under its supervision.

### *B. Synchronous Signal Delivery*

Handling asynchronous signals is one of the major challenges in multi-variant execution, as it can cause the variants to execute different sequences of system calls. This behavior is detected as a discrepancy and raises a false alarm in the system. For example, assume variant $p_1$ receives a signal and starts executing its signal handler. $p_1$'s signal handler then invokes system call $s_1$, causing the monitor to wait for the same system call from $p_2$. Meanwhile, variant $p_2$ has not received the signal and is still running its main program code. When $p_2$ calls system call $s_2$, the monitor detects the difference between $s_1$ and $s_2$ and raises an alarm. This scenario is depicted in Fig.2 .

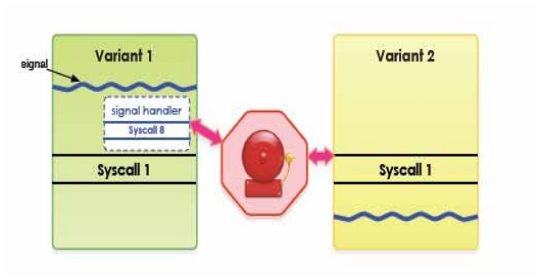


Fig.2 Asynchronous signals could cause the monitor to observe different sequences of system calls and raise a false alarm.

A possible solution is to deliver signals synchronously at synchronization points, which are in fact the same as 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 during the execution. This could cause some signals to be delivered with a long delay which might not be acceptable for certain types of signals, such as timer signals. We present a solution to the problem of asynchronous signal delivery which removes false positives caused by asynchronous signals and is not based on delivering signals at system calls. The variants are monitored after each system call and the following rules are applied to them:

- If all the variants are paused as a result of receiving a signal and none of them invokes any system call before receiving the signal, the signal is delivered to all the variants.
- If at least half of the variants receive a signal, but the rest invoke a system call, the monitor makes the latter variants skip the system call and forces them to wait for the signal. The monitor then delivers the signal to all the variants and restores the system call in those variants that have been made to skip it. The variants that are forced to wait for a signal and do not receive it within a configurable amount of time are considered as non-complying.
- If fewer than half of the variants receive a signal and the rest invoke a system call, the signal is ignored and the variants which are stopped by the signal are resumed. The monitor keeps a list of pending signals for each variant. All received signals are added to these lists by the monitor. As more variants receive the signal, the monitor checks the lists and then half of the variants have received the signal, the signal is delivered using the method mentioned in the above rule.

## IV. Implementation

To demonstrate the effectiveness of the multi-variant execution environment, we create a customized test suite which includes common benchmarks and frequently used applications. This suite allows us to evaluate the security claims and assess the computational tradeoff in CPU- and I/O- bound operations.

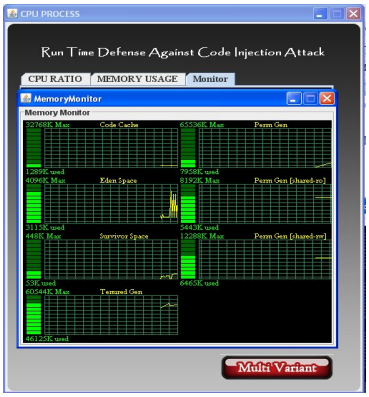


Fig.3 Multi-variant monitor during the run-time against code injection attack

One of the key features of our multi-variant execution technique that distinguishes it from n- version programming

is automated variant generation. The variants of a program are generated automatically from the same source code eliminating the need to rewrite the variants manually. This feature significantly reduces the costs of development and maintenance of the variants. Running two variants that grow the stack in opposite directions in a multi-variant environment helps preventing exploitation of stack-based buffer overflow vulnerabilities. Buffer overflow vulnerabilities give the opportunity to remote attackers to inject and execute malicious code. This phenomenon makes exploiting of this type of vulnerability appealing and, as a result, these vulnerabilities are still among the main sources of exploited software security flaws. The simplest and most common form of buffer overflow attacks is stack smashing. In this type of attack, an attacker overwrites the return address of the currently running function, and causes the program to jump to a desired location in memory that contains the injected code, and execute it. Stack smashing is shown in Fig.4.

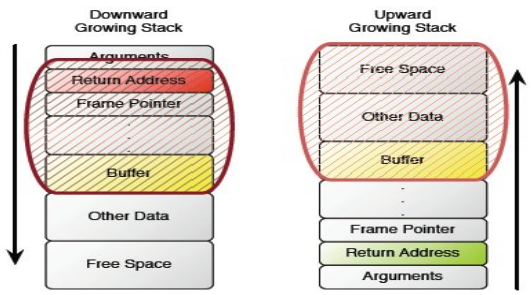


Fig. 4 The return address of the current function cannot be overwritten by exploiting buffer overflow vulnerabilities when the stack grows upward (right side).

When stack grows downward, an input larger than the size of the Buffer is given to the program and overwrites the return address of the current function. On the right side of this figure, we can see that the same vulnerability cannot be exploited to overwrite the corresponding return address on an upward growing stack. Function pointer overwrite is a similar attack in which vulnerabilities are exploited to overwrite function pointers rather than return addresses. When the function whose pointer is overwritten is called, control is transferred to the overwritten address which usually contains the malicious code.

## V. CONCLUSION

Multi variant execution is effective even against sophisticated polymorphic and metamorphic viruses and worms. The mechanism proposed in this dissertation trusts the operating system and protects against vulnerabilities in applications and their libraries.

A major benefit of this approach is that it enables us to detect and prevent a wide range of threats, including "zero-day" attacks. The multi-variant execution is an effective mechanism to thwart viruses, worms, and other exploitation of vulnerabilities. The technique discussed in this dissertation targets code injection attacks and is based on detecting "out-of- specification" behavior. Cross site scripting and SQL injection attacks constitute a large number of attacks in recent years. Although attack vectors used in these types of exploits also cause "out-of-specification" behavior, the vectors are not illegal inputs. This is in contrast to code injection attacks in which attack vectors are illegal inputs. Expanding the idea of multi- variant execution to cover cross site scripting and SQL injection can thwart a large spectrum of attacks; however the feasibility of this idea needs further investigation.

## 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.