FUDMA Journal of Sciences (FJS)
ISSN online: 2616-1370
ISSN print: 2645 - 2944
Vol. 3 No. 1, March, 2019, pp 49 - 55

AN EXPERIMENTAL STACK ATTACKS DETECTION AND RECOVERY FRAMEWORK USING AGENTS, CHECKPOINTS AND ROLLBACK

[1]Agaji, I., [1]Mikailu, H. & [2]Kile A. S.

[1]Department of Mathematics/Statistics/Computer Science, Federal University of Agriculture, Makurdi, Nigeria
[2] Department of Computer Engineering, University of Maiduguri, Maiduguri, Nigeria

Correspondence Author's Email: ior.agaji@uam.edu.ng & sasemiks@gmail.com

ABSTRACT
Stack based attacks are on the increase. This work generally studied stack-based vulnerabilities and attacks and focused on attacks which employ the modification of return addresses used by control stacks. A control stack keeps track of the point in which a function returns control to after its execution. We proposed a framework that mitigates control stack attacks which utilizes kernel-controlled agent, checkpoints and rollback mechanisms. In the framework once a function is called the same return address (RA) is pushed to the control stack and also passed to the kernel-controlled agent. When a function call terminates the RA in the control stack is popped and passed to the kernel protected agent for comparison and if there is any disparity in the values of the RAs then there is an attack. In such cases the kernel protected agent directs execution of the process to stack at the latest checkpoint. The framework was implemented using Java Netbeans 7.2.1. Experimental results of the framework indicated successful detection of attacks and rollbacks in case of the attacks. Rollback indicated recovery from the attacks.

Keywords: *Checkpoints, Rollback, Kernel protected agent, Control stack, Stack smashing attacks*

INTRODUCTION
Most today's Operating Systems utilize stacks in their operations. A stack holds immediate results of an operation or data that is waiting processing. A stack has a capacity and data received beyond its capacity is corrupted. Recently there has been increased stack smashing attacks using various approaches. While some attackers concentrate on introducing too much data than the capacity of the stack can hold, others focus on modifying the return address of functions and redirecting the address to their own code. Siberman and Johnson (2004) explored two approaches for applying a generic protection against buffer overflow attacks. With increased utilization of computers and internet in work places and the development of e-commerce, there has been growing concern over stack smashing issues which usually lead to huge loss in terms of revenue to organizations

LITERATURE REVIEW
There has been growing research in stacks mitigation attacks. This has led to the development of countermeasures against these attacks. In this section a review of the various methods adopted against such attacks and their shortcomings will be explored.
Murugan and Alagarsamy (2011) suggested many ways of detecting buffer overflows. These include entering extra data than asked for by a program that accepts input, entering malformed data for a program that accepts data in a standard format and the use of data block larger than the one specified in the size field. They further suggested a perfect coding style that would eliminate unchecked buffers which will eliminate buffer overflows

Patil and Chavan (2017) presented a systematic study on ways to make a browser secure. They listed attacks on a browser to include buffer overflow, browser cache poisoning, man-in-middle, session hijacking and clickjacking. They suggested many prevention measures against these attacks. These measures included the modification in the stack-allocated data and use of canary values for buffer overflow attacks, the use of input validation techniques that ensures correct entry of data, use of strong session ID to avoid hijacked sessions and use of frame busting defense against clickjacking. They further suggested the use of web browsers with electrolysis and sandboxing feature which restrict access to file systems.

Mirdula and Manivannan (2013) discussed commonly occurring online attacks in web applications. These attacks included phishing and pharming, IP spoofing, non-binding spoofing, binding spoofing, SQLinjection and cross cite scripting. They used a tool called BACKTRACK for checking SQL injection. Their method for performing attacks test using BACKTRACK involved gathering of information, carrying out vulnerability assessment, carrying out target assessment and maintenance and finally carrying out track clearance. They demonstrated how to use BACKTRACK to prevent SQL injection

Francillon et al (2009) presented a control flow enforcement known as Instruction Based Memory Access Control (IBMAC) which prevented low cost embedded systems against manipulation of the control of flow and accidental stack overflow. Their system divided the stack into two, that is, the data and return stack and these sections grow in opposite directions. The return section of the stack was used to store the control flow information while the data section was used to store regular data. This was achieved by a simple hardware modification to carry out the division. Their system did not totally prevent modification and control of information but makes it more cumbersome since control flow data was not close to stack allocated buffer. Their system was implemented as a modification of an existing simulator and also on a soft core on a field programmable gate array(FPGA)

Sahel et al (2013) proposed a software-based solution for stack-based vulnerabilities and attacks. Their solution created a random number of return addresses and stored them in random locations. In their method when a pointer was used all stored copies of the return addresses were read and compared. This successfully mitigated the attacks since it was difficult for an attacker to know all the locations where the return addresses were stored so as to modify all of them simultaneously. Their method proved good for mitigation of stack attacks, however, a lot of time was spent on comparison in order to ascertain if the return addresses stored in multiple locations were the same.

Sharazi and Kalaji (2010) applied information theory measures like entropy and mutual information and ranked 41 connection features according to their attack class after normalization. The connection features were ranked according to their importance in detecting attacks. They also designed network traffic linear classifiers based on Genetic algorithm which were trained using KDD99 data set. These classifiers were utilized in building a detection engine whose experimental results showed a detection rate of up to 92.94%

Mansour and Amir (2010) investigated the performance of rule extracting module from a dynamic cell structure (DCS) neural network in intrusion detection applications and compared it with fast multilayer perceptron (MLP)-based intrusion detection which utilized Output Weight Optimization – Hidden Weight Optimization (OWO-HWO) and selected 25 input features. They used a modified version of the LERX algorithm for rule extraction from DCS, the detection rate of their model was higher and the cost per example was lower than other models

Vadirelmurugan and Alagarsamy (2013) classified buffer overflow attacks into first generation, second generation and third generation. According to their classification first generation attacks focused on stack smashing, second generation focused on heap overflow and third generation involved format string attacks crashing the program printf. Printf is an in-built function in C used to print or display output on the screen. They suggested tools to prevent buffer overflow vulnerability which included Address space randomization, canaries, deep packet inspection, executable space protection and pointer protection

Alam et al (2010) presented different buffer overflow techniques exploited and methods used to mitigate them. The buffer overrun methods discussed in their work included arch injection which is used to invoke a number of functions including chain functions in sequence with arguments supplied by them. They also discussed heap smashing which overruns a heap buffer to change the control flow of a program which could overwrite function pointers stored on the heap thereby redirecting the control flow. They discussed various mitigating techniques such as StackGuard, protection of function pointers and use of high quality code

Leon and Bruda (2016) discussed buffer overflows attacks in GNU/Linux OS. Their proposed solution worked using ptrace system call as the main engine. Ptrace is a tool built in GNU/Linux that allowed for the interception of certain resources during process execution for analysis. Their method has the advantage of not requiring hardware modification as required by many other such similar systems.

Shinagawa (2006) presented an efficient mechanism for protection against buffer overflow attacks that utilized pointer copying. Copies of code pointers were stored in safe memory locations and used to detect and prevent manipulation of code pointers. To protect the copied code pointers from modifications attacks segmentation hardware of IA-32 processors were exploited. The scheme involved a small overhead of time used in copying a code pointer.

Shacham, et al. (2004) studied the effectiveness of randomizing address space and observed that its utilization in 32-bits architectures was limited by the number of bits available for address randomization. They explored many other ways of strengthening the method and observed the weaknesses associated with each. They concluded that on a 32-bit architecture the only gain of a PaX-like address space randomization was a small slowdown in worm propagation speed

Sandeep et al. (2003) carried out a systematic study of address obfuscation that randomized the location of victim program data and code. They presented their implementation which transformed object files and executables at link and load times.

Their method required no changes to the OS kernel or compilers and the method was applied to individual application programs without affecting the rest of the system. Their system reduced the probability of successful attacks to the barest minimum and it also ensured that an attack that succeeded against one victim was not likely to succeed against another or even for the second time against the same victim. Their system was particularly useful against large scale attacks as each failed attempt typically crashed the victim's program

Cowan et al. (1998) described StackGuard which was a compiler technique that eliminated buffer overflow vulnerabilities with modest performance and penalties. With the system privileged programs that were recompiled with StackGuard extension no longer yielded control to attackers since such programs required no source code changes at all and were binary compatible with existing OS and libraries. Their system provided an adaptive response to buffer overflow attacks. In their system performance was traded for survivability of systems.

Baratloo et al (2000) presented two methods to detect and handle buffer overflow vulnerability attacks. Their first method intercepted all calls to library functions known to be vulnerable. A substitute version of the corresponding function implemented the original functionality in such a manner that guaranteed that no overflows were contained within the current stack frame. The second method used binary modification of the process memory thereby forcing verification of critical elements of the stack before use. Their methods were implemented on Linux as dynamically loadable libraries. Their methods detected many known attacks and their performance overhead were not more than 15%

Younan et al (2006) presented an efficient countermeasure against stack smashing attacks that utilized the splitting of the standard stack into multiple stacks in which the allocation of data types to any of the stack was based on the chances that a data element was either a target of attack or an attack vector. They implemented their method using C-compiler for Linux and their method showed negligible overhead.

A similar method to Stack Shield was suggested by Xu et al. (2002). The method of their countermeasure divides the stack into a control and a data stack. The control stack is to store the return addresses while the data stack comprises the remaining data stored on the stack. Their method, before any function call, copies the return address to the control stack and copies it back from the control stack where it was stored onto the data stack ahead of return from the function call. The researchers provided performance results that showed a minimum performance overhead.

METHODOLOGY

The proposed framework detects stack smashing attacks and recovers from such attacks using kernel protected agent, checkpoints and rollback mechanisms. The framework was designed using a UML tool called sequence diagram. During function call the RA is pushed onto the control stack and also passed as a message to the kernel protected agent. A checkpoint represents a milestone in the execution sequence of a process. The RAs are compared after termination of function call and if there are disparities then an attack has occurred and the latest checkpoint is located and process execution is rolled back to the checkpoint and process execution continues. The work is similar to the work proposed by Sahel et al (2013) which makes use of random locations for the storage of return addresses. However, the work utilized kernel protected agent which can store return addresses from different function calls. The work also utilizes checkpoints and rollback mechanism to enable recovery from control stack attacks. The architecture of the proposed framework is depicted in fig 1. The architecture shows process execution with zero or more occurrences of function calls. Once a function call occurs the return address (RA) is pushed onto a stack as well as passed to a kernel protected (KP) agent and servicing of the function call is carried out. Upon return from servicing of the function call the RA in the stack is popped and passed to the KP agent for comparison and once a disparity is discovered the agent redirects the process execution to start at the most recent checkpoint so that execution will start from there. A disparity is an indication that an attack has occurred. If there is no disparity then there is no attack and normal execution of the process continues. The difference between the use of agents and other methods that employ memory locations to store return address is that a single agent can store many return addresses from different function calls.

The sequence diagram for the framework is as shown in fig 2. Actors/Objects identified in the framework are Process Executor, Process, Kernel protected Agent (KPA), Control Stack, Checkpoints and functions. The interactions between the actors/objects are depicted with respect to time. The Process executor executes process instructions and creates checkpoints at certain intervals, when a function call occurs the process executor agent branches to execute the function after saving the RA to the stack and passing the same RA to the kernel protected agent. On returning from the execution of the function, it compares the two addresses in the stack and the kernel protected agents. If a variation is observed, it rolls back the execution to the latest checkpoint else it continues with normal program execution.

This approach will enhance detection and recovery from any stack smashing attacks. Moreover, this proposed approach will have better performance as opposed to the compiler-based approach that has the limitation of runtime overhead as well as the architectural approach that requires changing the instruction set semantics, and adding new registers into the processor when implemented as proposed by some researchers.

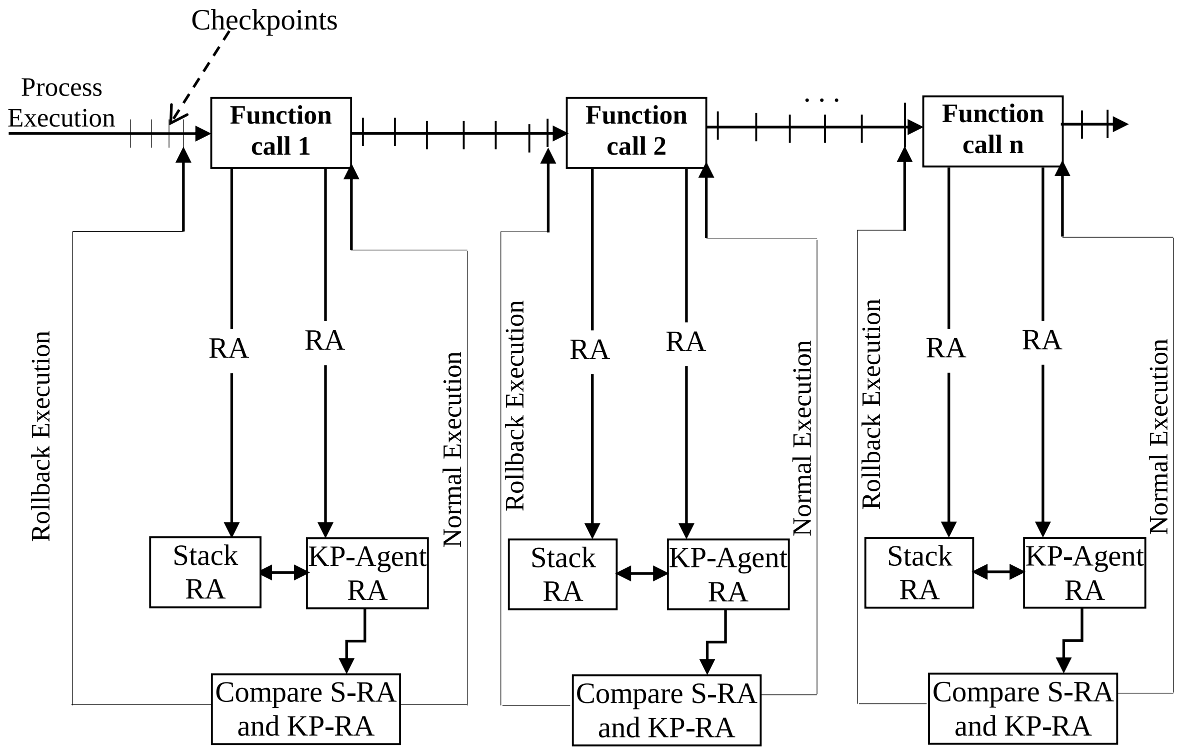


Fig.1: The Architecture of the Proposed Framework

RESULTS

The framework was simulated using Java NetBeans. Process execution was modeled using a sequence generating positive integers with each integer representing an executable statement in the process. Checkpoints were created at regular intervals using the integers. Random numbers were used in triggering events. Events in the framework were the occurrence of function call and the occurrence of an attack. For an event to occur a threshold was set on the random number generated and once that threshold was exceeded the event was said to occur.

Five simulation runs were carried out using the framework. Sample outputs from the simulation are shown in figs 3.

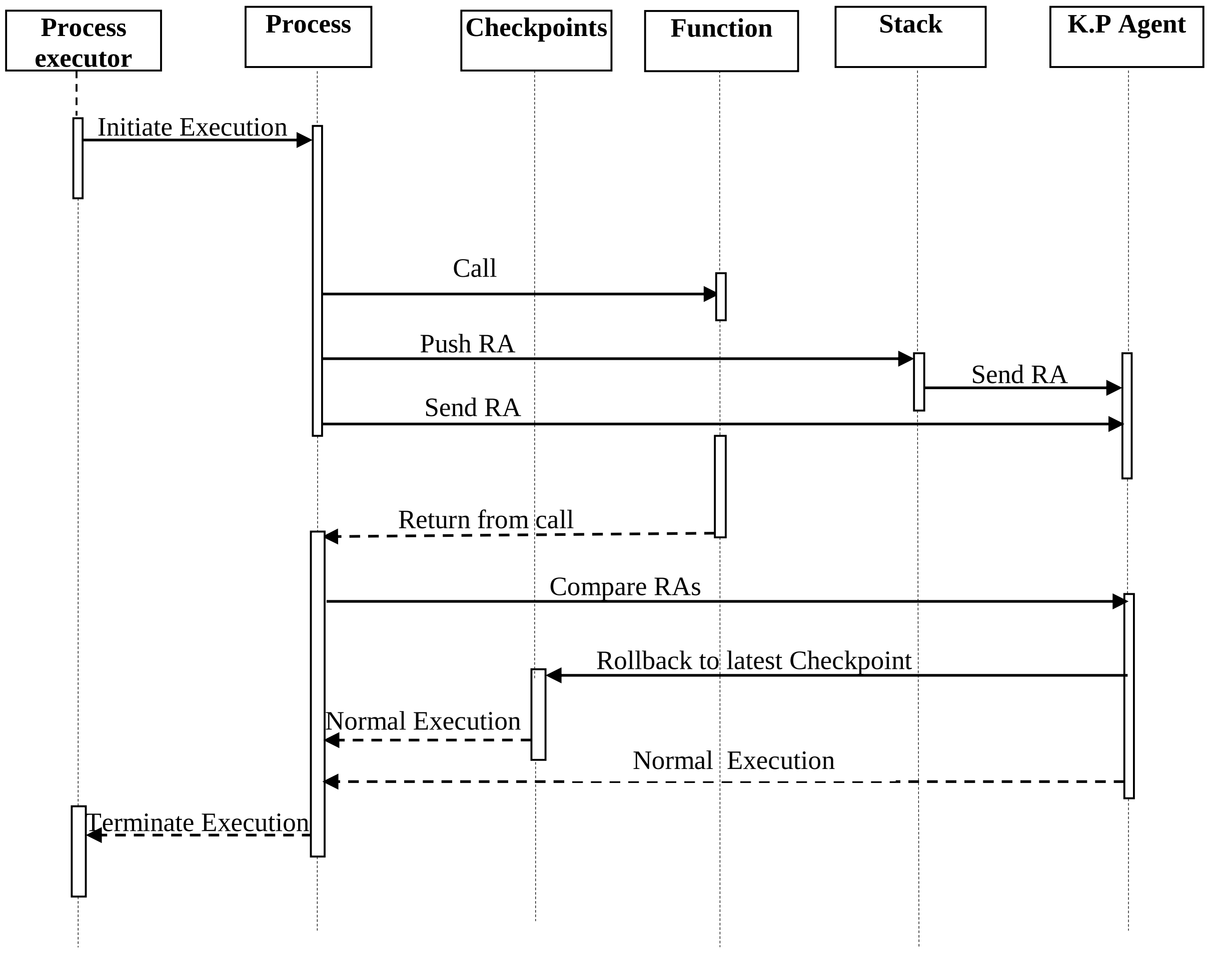
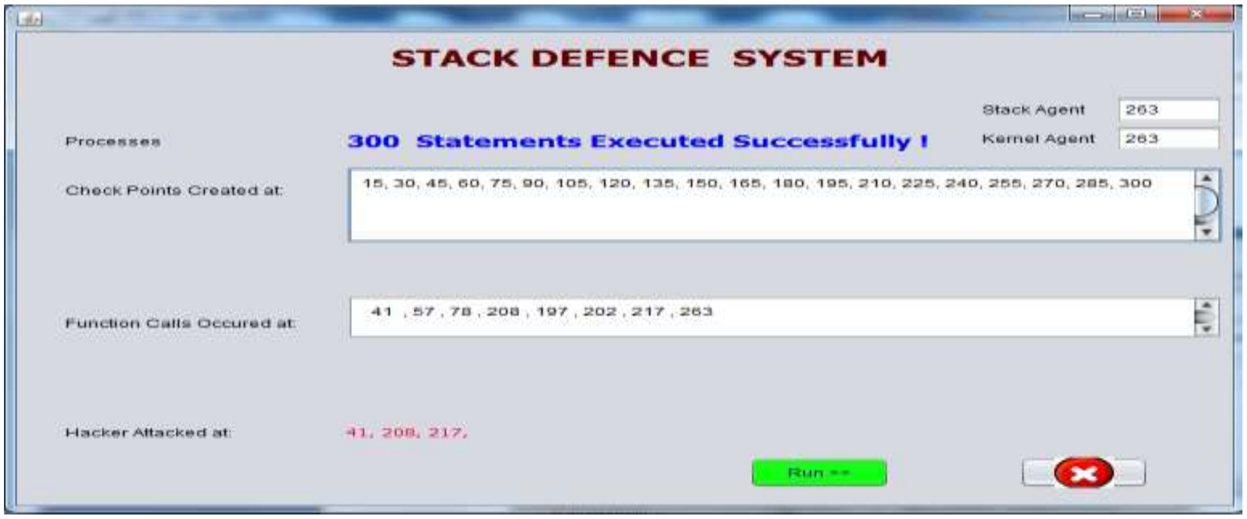


Fig. 2: A Sequence Diagram of the Proposed Framework


STACK DEFENCE SYSTEM

Stack Agent 263

Kernel Agent 263

Processes 300 Statements Executed Successfully !

Check Points Created at: 15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, 210, 225, 240, 255, 270, 285, 300

Function Calls Occured at: 41 , 57 , 78 , 208 , 197 , 202 , 217 , 263

Hacker Attacked at: 41, 208, 217,

Fig. 3: Sample Output from the Simulation runs.

Fig 3 shows the number of statements in the process, the number of checkpoints created during the execution of the process, number of functions called during process execution and points in the execution where attacked occurred. The results of the five simulation runs are summarized in table 1.

Table 1: Summary of Results

Process Number	Number of Executable statements	Number of function calls that occurred	Number of attacks	Number of Successful rollbacks	Points where rollbacks occurred
Process1	300	8	3	3	41, 208, 217
Process2	1200	25	6	6	120,360,375,630, 825,1005
Process3	800	14	3	3	405,465,660
Process4	600	0	0	0	0
Process5	1500	21	5	5	75,285,915,1110, 1395

In the table 1 only Process4 made up of 600 executable statements was without any stack smashing attacks. All other processes had varying number of attacks. In all the processes that there were attacks the number of attacks was the same as the number of successful rollbacks signifying recovery from such attacks.

CONCLUSION

The work generally examined the various stack-based attacks and concentrated on the control stack attacks. The research was motivated because of the increased stack smashing attacks by hackers and attackers. In this work, an experimental framework for the mitigation of stack smashing attacks that makes use of kernel protected agent, rollback and checkpoint mechanisms was proposed. Simulations with the framework indicated successful detection and recovery from control stack attacks. The framework is recommended because of its detection and recovery from control stack attacks.

REFERENCES

Alam, M., Johri, P., & Rastogi, R.(2010). Buffer overrun: techniques of attacks and its prevention. *International Journal of Computer Science and Engineering,* 1(3): 1-6.

Baratloo, A., Singh, N., & Tsai, T. (2000). *Transparent run-time defense against stack smashing attacks.* In USENIX 2000 Annual Technical Conference Proceedings, San Diego, CA.

Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. & Zhang, Q. (1998). *Stack-Guard: automatic adaptive detection and prevention of buffer-overflow attacks.* In Proc. of the 7th USENIX Security Symp., San Antonio, TX.

Francillon, A., Perito, D., & Castelluccia, C.(2009). Defending embedded systems against control flow attacks. http://s3.eurecom.fr/docs/secucode09_francillon.pdf. Retrieved 7th December 2018.

Leon, E., & Bruda, S.D.(2016). Counter-measures against stack buffer overflows in GNU/Linux operating system. *Procedia Computer science, 83(2016): 1301-1306.*

Mansour, S., & Amir, K.(2010). Intrusion detection based on rule extraction from dynamic cell structure neural networks. *Majlesi Journal of Electrical Engineering,* 4(4): 24-34.

Mirdula, S., & Manivannan, D. (2013). Security vulnerabilities in web applications – an attack perspective. *International Journal of Engineering and Technology,* 5(2):1806-1811.

Murugan, P. V. & Alagarsamy, K. (2011). Buffer overflow attack – vulnerability in stack. *International Journal of Computer Applications,* 13(5): 1-2.

Patil, S. S. & Chavan, R.K. (2017). Web browser security: different attacks detection and prevention techniques. *International Journal of Computer Applications,* 170 (9): 35-41.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. & Boneh, D. (2004). *On the effectiveness of address-space randomization.* In the 11th ACM Conference on Computer and Communications Security (CCS).

Sahel, A., Mazen, K. & Rana, A. (2013). Stack memory buffer overflow protection based on duplication and randomization. *Procedia Computer Science,* 21(2013):250-256.

Sandeep, B,. Daniel, C. D., & Sekar, R. (2003). *Address obfuscation: an efficient approach to combat a broad range of memory error exploits.* In the 12th USENIX Security Symposium. Washington DC

Sharazi, H.M. & Kalaji, Y. (2010). An intelligent intrusion detection system using genetic algorithms and features selection. *Majlesi Journal of Electrical Engineering,* 4(1): 33-43.

Shinagawa, T. (2006). *SegmentShield: exploiting segmentation hardware for protecting against buffer overflow attacks*. 25th IEEE Symposium on Reliable Distributed Systems, Leeds, UK.

Siberman, P., & Johnson, R. (2004). Comparison of buffer overflow prevention implementations and weaknesses. iDEFENSE, https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf. Retrieved 29th January, 2019.

Vadirelmurugan, P. & Alagarsamy, K. (2013). Cataloguing and avoiding the buffer overflow attacks in network operating systems. *International Journal of Computer and Organization Trends*, 3(4): 66-70.

Xu, J., Kalbarczyk, Z., Patel, S. & Ravishankar, K. I. (2002). *Architecture support for defending against buffer overflow attacks.* In Second Workshop on Evaluating and Architecting System dependability, San Jose, CA.

Younan, Y., Pozza, D., Pissens, F. & Josen, W. (2006). Extended protection against stack smashing attacks without performance loss. https://www.acsac.org/2006/papers/43.pdf. Retrieved 7th December 2018.