2023 17th International Conference on Open Source Systems and Technologies (ICOSST) | 979-8-3503-8132-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICOSST60641.2023.10414205

LsStk: Lightweight solution to preventing Stack from buffer overflow vulnerability

Muhammad Ahsan
Department of Electrical Engineering
University of Engineering and Technology
Lahore, Pakistan

Muhammad Ali
Department of Electrical Engineering
University of Engineering and Technology
Lahore, Pakistan

***Abstract*—Buffer overflow is a significant vulnerability in computer systems, and despite numerous techniques and approaches proposed to protect programs from exploitation, many existing software and hardware-based prevention mechanisms have proven ineffective in mitigating this threat. The presented study introduces a lightweight prevention technique (LsStk). The approach involves implementing an encryption algorithm to safeguard the function return address, making it resilient to exploitation by adversaries. The algorithm integrated at the compiler level, utilizes epilogue and prologue functions to encrypt the return address and protect it from adversarial tampering. Given its minimal footprint, the solution is well-suited for adaptation in embedded IoT (Internet of Things) devices.**

To evaluate the effectiveness of our proposed technique, we conducted a statistical analysis using various 32-bit servers. We compared the performance of our method with existing prevention mechanisms. Our approach is designed to be adaptable at both the hardware CPU architecture level and as a software-level patch for the GCC compiler. Our analysis leads to the conclusion that our proposed technique significantly enhances security against such attacks.

***Index Terms*—Buffer overflow attacks, Return address encryption, Statistical analysis**

I. Introduction

Network security has been an ongoing research field for decades with new challenges emerging very often. To create a secure atmosphere for the user to work in, Internet security is the process of taking preventive measures to secure the system from potential misuse, malware, unauthorized access, etc.

Morris Worm 1988 [1], developed by an MIT graduate student, was one of the first internet attacks that got mainstream media attention due to the size and magnitude of the damage it caused. The worm uses overflow vulnerability and propagates through the internet. It infected around 60,000 systems and estimated damages were around $100,000–10,000,000 [2]. Heartbleed another major attack was caused due to buffer-overread vulnerability. The vulnerability occurred due to a bug in the Open SSL cryptography library. The exploit was made public in April 2014. Jun Wang et al. provided a complete risk assessment for the Heartbleed vulnerability [3]. Related to the bug another critical level patch was recently released to openSSLv3 (CVE-2022-37786) [4].

979-8-3503-8132-0/23/$31.00 ©2023 IEEE

Buffer overflow is an attack technique that emerges very often in the network and is used for penetrating and gaining unauthorized access to user data. Based on taxonomy, there are two basic types of buffer overflow vulnerabilities stack-based and heap-based. From the point of ease of application and practicality, stack-based vulnerabilities are more feasible to exploit as compared to heap-based attacks. A survey on various types of buffer overflow vulnerabilities, attacks, and various defensive measures and techniques to mitigate the vulnerability is presented by Padmanabhuni et al. [5].

Cyber-physical systems are gaining a lot of attention and becoming everyday commodities. These include IoT devices, smart vehicles, smart grids [6], and smart manufacturing [7]. These devices are being deployed in critical applications making their security paramount. A lot of research is available in securing such devices against adversarial attempts [8]–[10]. This paper presents a lightweight algorithm to protect the function return address to be exploited by the vulnerability. The algorithm uses lightweight encryption to protect the return address from being exploited. The technique integrated at the compiler level uses function epilogue and prologue functions for protecting the return address. Due to minimal resource consumption [11], the algorithm is well-suited to be implemented for resource-constrained IoT devices [12]. The effectiveness of the proposed technique is statistically analyzed by implementing multiple 32-bit servers and comparing its performance against the available lightweight technique in the literature. For the sake of simplicity, 32-bit system architecture is assumed, however, the proposed algorithm is generic and could be tweaked to be implemented on 64-bit system architecture.

The rest of this paper is organized as follows: Section II provides a survey of the available preventive measures, in Section III types and anatomy of buffer overflow attacks is detailed. Section IV explains the proposed prevention methodology. Section V provides a statistical analysis of the proposed algorithm, and in the end, Section VI concludes the paper along with future recommendations.

II. Literature Survey

A lot of work is being done and is available in the literature for buffer overflow vulnerability prevention. Preventive

measures can be implemented at 2 stages at the system kernel level or at the compiler level. Silberman et al. [13] presented a comprehensive overview of different prevention techniques and their weaknesses and a comparative study is provided for the available techniques. Some of these known prevention mechanisms are further discussed in this section.

Address Space Layout Randomization (ASLR) [14] is a kernel-level prevention mechanism developed by the Pax Team as a patch for Linux in 2001. The aim is to create randomness in the process memory space. Since the exploit's success depends upon correctly knowing or guessing the current location of the process in memory, by creating randomness in the address space, the attacks are prevented. If the attacker tries to attack the wrong address of the memory the program will be crashed and an alert will be generated for the system. Hovav Shacham et al. presented a paper on the effectiveness of the ASLR technique [15]. Different techniques have been studied in the literature to bypass ASLR techniques including brute force attacks. A more recent technique to bypass ASLR security is presented by Dmitry et al. [16] where along with a possible vulnerability to bypass ASLR they also presented potential hardware and software protection techniques to mitigate the problem described.

Data Execution Prevention (DEP) [17] is a kernel-level prevention mechanism and was first introduced in 2003 in Windows XP service pack 2. The main objective was to prevent the exploitation caused due to stack-based buffer overflow attacks. In the proposed technique pages are marked to make certain areas of the program memory non-executable. If an adversary attempts to execute the malicious code in the protected memory area an exemption is generated causing the process to terminate. DEP works well with other protection schemes but can be easily bypassed when working alone. Return into libc and push esp, ret attack proposed by Stojanovski et al. [18] suggests that DEP is vulnerable and can therefore be bypassed to execute buffer overflow attacks.

Stack Guard is one of the earliest techniques proposed at the compiler level for securing the return address to be exploitable from buffer overflow attacks. It was proposed in 1998 by Crispan Cowan et al. [19]. Stack Guard puts a canary value at the top of the return address and ensures its integrity before the function hits return to the main routine. Different techniques are available to bypass Stack Guard few of these were presented by Gerardo Richarte [20]. One of the techniques to bypass the canary is to brute force the canary value itself making it vulnerable to be exploited by buffer overflow attacks. DynaGuard [21] is a technique to protect the canary value from such brute-force attempts. Another technique named RCR for preventing stack smashing attacks by bypassing canary values is proposed by Shehab et al. [22].

Stack Shield [23] is a preventive technique to counter stack smashing attacks. In this approach, the return address is saved in a non-overflowable memory location at the function call, which then checks the return address at the function return for any corruption. A technique presented by Bulba and Kil3r [24] shows that the attacker can use indirect pointer overwrite to bypass the prevention mechanism.

Function pointer/return address encryption is another technique to protect the pointer from being exploited by buffer overflow attacks. Crispin Cowan et al. [25] presented a technique named PointGuard to protect the pointers from vulnerabilities. The presented technique is based on a compiler where the pointers are encrypted before being stored in the memory and are decrypted when put into CPU registers. Another similar approach is presented by Ge Zhu et al. [26] for protecting pointers against indirect overflow attacks. They proposed to randomize the key for each run of the program so that the adversary would not be able to get the desired knowledge to break encryption. The proposed modifications lead the attacker to perform a brute force attack with a cost of 2^{32} for a 32-bit system architecture.

Zhu [27] in his Ph.D. dissertation proposed another encryption technique where he used RC5 encryption rather than XOR. RC5 is a block encryption and decryption algorithm proposed by Rivest [28]. The encryption is implemented for a 32-bit system architecture and overhead is calculated to be around 15% which is more than XOR encryption but adds more security and is a good trade-off. Madan et al. [29] presented a technique named StackOffence. The integrity of the return address is ensured by pushing an encrypted version of the return address along with the original address and is compared for any discrepancies at the time of return from the subroutine. An in-depth analysis to bypass the preventive measures is presented by Butt et al. [30].

Hardware-based approaches [31]–[35] have also been proposed in the literature. Tuck et al. [32] presented a hardware technique to protect code pointers from buffer overflow attacks. Another similar technique is presented by Shao et al. named Hardware/Software address protection (HSAP) [33] and Hardware/Software Defender (HSDefender) [34] to protect function pointer from buffer overflow vulnerability.

III. State of the Art

Buffer overflow attacks are triggered by causing the program to jump to a specified malicious address. This jump to the address can be done by one of the many methods [36] including by corrupting the: function return address, function pointer, and longjmp buffers. The details of which are provided in the following subsections.

A. Function Return Address Corruption

During the execution of the process whenever a call to a subroutine happens (function call), a stack frame is created where the function return address to the main routine, function arguments, function frame pointer, and local variables are pushed. In this frame, the return address to the main routine can be exploited and vulnerable to attack using buffer overflow vulnerability. In case there is a buffer that is vulnerable to overflow due to no bound checking the adversary can use this in their favor and corrupt the return address by overflowing the buffer and pointing it to their desired location.

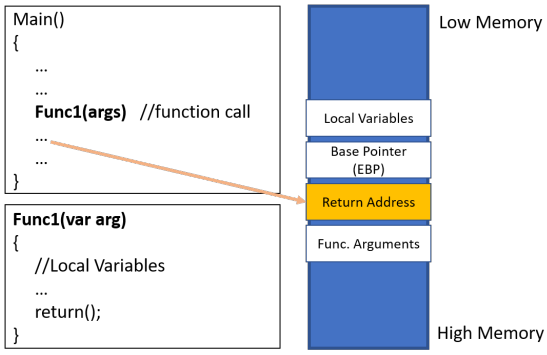

Fig. 1: Stack during Function Call

B. Function Pointer Corruption

A pointer is used to store the address of an array, a function, or any other data structure being used in the program. These pointers can be modified to alter the flow of the program to the attacker's shell code. In this type of attack, the adversary modifies the value of the pointer (function or data pointer) which then can be used to trigger stack-based or heap-based buffer overflow attacks. These attacks are effective even in case the return address modification is protected because these attacks don't involve modifying the return address.

C. Longjmp Buffer

C99 standard C library defines a setjmp/longjmp procedures for executing non-local jumps (non-local goto). The purpose was to avoid using the function call and return procedures for these types of jumps. The setjmp() saves the function's calling environment in a separate jmp_buf array type variable which is later used by the longjmp() to restore the environment for the latest setjmp() call. An adversary can use this in his favor by overflowing the jmp_buf with the desired address which points to the attacker's code. So, when the longjmp() is called the corrupted address will take the flow of the program to the attacker code [32].

D. Buffer Overflow Attack Anatomy

Exploiting a buffer overflow vulnerability starts with understanding the stack, which follows the Last In First Out (LIFO) principle. It uses Push and POP operations to add or remove elements from the top, controlled by an Extended Stack Pointer (ESP) in 32-bit systems. The stack temporarily stores data like function arguments, return addresses, and local variables during program execution, making it a prime target for manipulation in buffer overflow attacks.

The stack grows from higher to lower memory addresses. Fig.1 presents the working of a stack during a function call. The main routine function (Func1) is being called having arguments (arg). During Func1 execution the items pushed into the stack include function arguments, function return address, Extended Base Pointer (EBP), and then followed by space allocated for local variables to be used in the subroutine.

To prepare the stack to be used during function call and return there are a set of assembly language instructions that

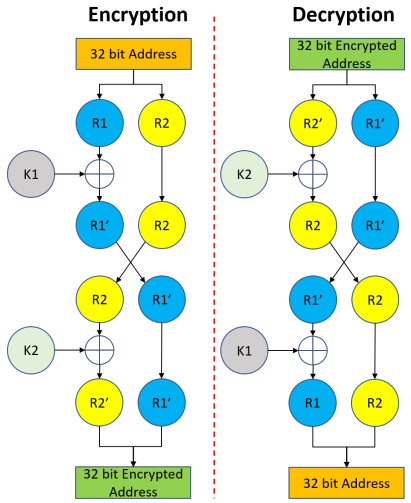

Fig. 2: Block Diagram for Proposed Algorithm

are executed. These instructions are termed function prologue and epilogue procedures and are given below.

Function Prologue

$Push \quad EBP$
$MOV \quad ESP, EBP$
$SUB \quad ESP, N$

Function Epilogue

$MOV \quad ESP, EBP$
$POP \quad EBP$
RET

The $'RET'$ instruction when executed POP the return address from the stack and passes it to the program Instruction Pointer (EIP). The presented study is focused on securing the function return address to be exploited using buffer overflow vulnerability. The attacks are also termed direct overflow attacks or stack smashing attacks.

Following the case as presented in Fig. 1 for the adversary to change the follow of the program to its desired location it has to corrupt that return address in the stack with the malicious address. So, when the calling function returns to the main routine and POP the return address into EIP instead the control flow of the program will be changed to the adversary code where a malicious code will be present ready to be executed.

IV. Proposed Prevention Algorithm

The encryption and decryption algorithms commonly used for pointers tend to either exhibit weaknesses in security or come with high time costs. In this section, we introduce a cost-sensitive algorithm (LsStk) designed for encrypting return addresses. This algorithm can be seamlessly integrated by modifying the prologue and epilogue procedures within the GCC compiler, making it a practical and efficient choice for enhancing security.

The LsStk algorithm is based on the Feistel Cipher design model, incorporating two encryption keys and an additional

Algorithm 1 LsSTK Encryption Algorithm

Input: K_1, K_2, R
Output: R'
Initialisation : $tmp \leftarrow R$
1: $R \leftarrow R \oplus K_1$
2: $R \leftarrow R$ & $0xffff0000$
3: $tmp \leftarrow tmp$ & $0x0000ffff$
4: $R << 16$
5: $tmp >> 16$
6: $R \leftarrow$ Concatenate R and tmp
7: $tmp \leftarrow R$
8: $R \leftarrow R \oplus K_2$
9: $R \leftarrow R$ & $0xffff0000$
10: $tmp \leftarrow tmp$ & $0x0000ffff$
11: $R' \leftarrow$ Concatenate R and tmp
12: **return** R'

rotation step. This design enhances its security compared to XOR encryption, offering two levels of encryption through the use of these two keys. While the LsStk algorithm introduces some computational overhead, it is still more efficient than the RC5 encryption algorithm, albeit less so than XOR encryption. Algorithm 1 details the LsStk encryption algorithm wherein the input is the two keys K_1 and K_2 along with function return address R, and the output is the encrypted return address R'.

For a 32-bit system architecture the encryption algorithm involves the following steps

- Step 1: The address is partitioned into two equal parts of 16 bits as R1 and R2.
- Step 2: R1 is XORed with KEY-1 and the result is R1'.
- Step 3: Circulation of R1' and R2.
- Step 4: XOR R2 with second key KEY-2 to give R2'.

The same steps are followed for decrypting the return address as shown in the block diagram in Fig. 2. The new function prologue and epilogue after incorporating the algorithm are shown in Fig. 3. The proposed algorithm is generic and can be applied to a 64-bit system architecture, however, for the sake of simplicity, the design and evaluation are limited to 32-bit architecture.

V. Statistical Analysis

For statistically analyzing the proposed algorithm in comparison with XOR encryption we will be considering three types of servers

- Single process server
- Inetd-based server
- Forking server

Each of these servers will be statistically analyzed on a given attack for the proposed algorithm and XOR Encryption.

For the case of 32-bit system architecture $d_1 = d_2 = 32$ which gives a total probability p_0 as;

$$p_0 = p_1 p_2 = 1/2^{32} * 1/2^{32}$$
$$p_0 = 1/2^{64} \quad where \quad D_0 = 2^{64}$$

```
SUB ESP,N
MOV EBX,DWORD PTR SS:[EBP+4]        #EBX Contain the return address
XOR DWORD PTR SS:[EBP+4],KEY-1      #(RET Address) XOR (KEY-1)
AND DWORD PTR SS:[EBP+4],FFFF0000   #To use right most 4 bytes
AND EBX,0FFFF                       #To use left most 4 bytes
SHR DWORD PTR SS:[EBP+4],8
SHR DWORD PTR SS:[EBP+4],8          #Sift right 16 bits
SHL EBX,8
SHL EBX,8                           #Sift left 16 bits
OR DWORD PTR SS:[EBP+4],EBX         #Combine both parts
MOV EBX,DWORD PTR SS:[EBP+4]        #Move the combined part to EBX
XOR DWORD PTR SS:[EBP+4],KEY-1      #(Modified return) XOR (KEY-2)
AND DWORD PTR SS:[EBP+4],FFFF0000   #To use right most 4 bytes
AND EBX,0FFFF                       #To use left most 4 bytes
OR DWORD PTR SS:[EBP+4],EBX         #Combine both parts
```

(a) Function Prologue

```
MOV EBX,DWORD PTR SS:[EBP+4]        #EBX Contain the return address
XOR DWORD PTR SS:[EBP+4],EBP        #(RET Address) XOR (KEY-2)
AND DWORD PTR SS:[EBP+4],FFFF0000   #To use right most 4 bytes
AND EBX,0FFFF                       #To use left most 4 bytes
SHR DWORD PTR SS:[EBP+4],8
SHR DWORD PTR SS:[EBP+4],8          #Sift right 16 bits
SHL EBX,8
SHL EBX,8                           #Sift left 16 bits
OR DWORD PTR SS:[EBP+4],EBX         #Combine both parts
MOV EBX,DWORD PTR SS:[EBP+4]        #Move the combined part to EBX
XOR DWORD PTR SS:[EBP+4],EBP        #(Modified return) XOR (KEY-1)
AND DWORD PTR SS:[EBP+4],FFFF0000   #To use right most 4 bytes
AND EBX,0FFFF                       #To use left most 4 bytes
OR DWORD PTR SS:[EBP+4],EBX         #Combine both parts
```

(b) Function Epilogue

Fig. 3: Function Prologue and Epilogue for proposed algorithm

TABLE I: List of Acronyms

d = Entropy of Encryption Key
D= 2^d (Total possible encryption keys)
d_1 = Entropy of Encryption Key k_1
D_1 = 2^{d_1} (Total possible encryption keys for k_1)
d_2 = Entropy of Encryption Key k_2
D_2 = 2^{d_2} (Total possible encryption keys for k_2)

These assumptions will be used for statistical analysis of all the servers to be discussed later in this section. For statistical analysis, we will follow the procedure used by Hector et al. [37] for comparing the effectiveness of defensive techniques like DEP and ASLR.

A. Single Process Server

In single-process servers, the clients are serviced by a single process. Based on internal architecture, a single process server is further divided into:

- Event Based
- Sequential
- Multi-Threaded

For statistical analyses of these servers, in the event of an attack, all of them exhibit the same behavior. Once the process is crashed due to an incorrect or faked request the services will be stopped and are not restarted again. So, breaking into these types of servers is practically impossible.

Given the behavior of the single process server, the attacker is given only one attempt to break into the system since in

TABLE II: Inetd Server (Trial and Test) result comparison

	XOR Encryption	Proposed Algorithm
$E[X]/\mu$	2^{32}	2^{64}
$Var[X]/\sigma^2$	$2^{32}(2^{32}-1)$	$2^{64}(2^{64}-1)$
CDF	$Pr(X \leq k) = 1-(1-1/2^{32})^k$	$Pr(X \leq k) = 1-(1-1/2^{64})^k$
100%	∞	∞
95%(3μ)	$3((2^{32})$	$3((2^{64})$
50%(0.68μ)	$0.68(2^{32})$	$0.68(2^{64})$

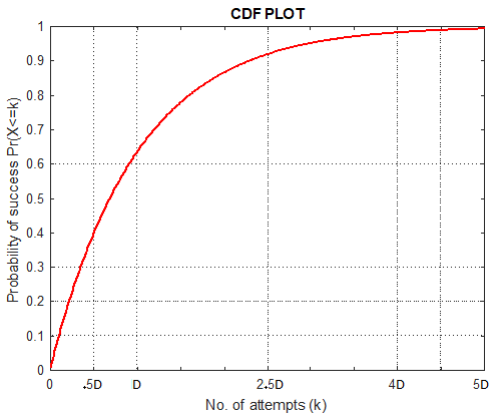

Fig. 4: CDF Plot for Trial and Test attack

case of an incorrect attempt the services will be stopped and are not restarted by the administrator. So, the attack model for this type of server follows Bernoulli's equation.

$$Pr(X=n) = \begin{cases} 1-p & if \quad n=0, "failure" \\ p & if \quad n=1, "success" \end{cases}$$

Where, $p = 1/(2^{32})$ for XOR encryption and $p = p_0 = 1/(2^{64})$ for the proposed algorithm. Given these very low probabilities of success, it is highly unlikely for the attacker to gain access to the vulnerable machine.

B. Inetd Based Server

The inetd also known as the 'Internet super-server' listens for the connections on certain host sockets (like FTP, telnet POP3, etc.) and connects the available requests to the corresponding program to service that request. In these types of servers, the client request is serviced by a new process initiated by the server via the execution of $fork() \Longrightarrow exec()$ sequence. Therefore, whenever a new process is created the secrets as used by the child processes are also changed for each new request by the client. Given the server behavior, an adversary can successfully conduct a Trail and Test attack the details of which are given below.

1) Trial and Test attack: In this technique, the attacker cannot discard the already tested keys since the keys are replaced after every incorrect attempt. This process is also known as 'sampling with replacement'. In the case of an Inetd-based server adversary can attack the key for infinite many times as the services are restarted for every incorrect attempt but the attacker needs to remember the incorrect keys as the new process to service the request will have different secrets as used by the child process.

TABLE III: Forking Server (Brute Force) results comparison

	XOR Encryption	Proposed Algorithm
$E[X]/\mu$	$2^{32}/2$	$2^{64}/2$
$Var[X]/\sigma^2$	$2^{32}/12$	$2^{64}/12$
CDF	$Pr(X \leq k) = k/2^{32}$	$Pr(X \leq k) = k/2^{64}$
100%	2^{32}	2^{64}
95%(3μ)	$0.95(2^{32})$	$0.95(2^{64})$
50%(0.68μ)	$2^{32}/2$	$2^{64}/2$

This attack is modeled as Bernoulli's trial equation with the total of 'k' attempts made and the probability of success for each attempt is 'p'. To find the number of attempts till we have 100% success the process is modelled as Geometric distribution. So, for successfully breaking the encryption key the attempts required 'k' will be in the range from $[1, \infty]$

The comparative results for the proposed and XOR encryption techniques for Inetd-based server Trial and Test attacks are presented in Table II with CDF plot presented in Fig. 4. It can be inferred that it is practically impossible to achieve a 100% success rate for these attacks to be successful. For the 50% success rate, the results show that the XOR encryption is more vulnerable than the proposed with the factor of 2^{32}.

C. Forking Server

Forking-type servers are a subcategory of multi-process servers. In these servers, the client is serviced directly by the child process from the parent process. So, to service a new request a new child process is created and when the request is terminated the child process also terminates.

The behavior of forking servers matches closely to that of Inetd servers in terms of operation but unlike Inetd servers as explained earlier the child processes are directly responsible for servicing the client requests and no new process image is loaded into the memory when a new service request is generated. So, the behavior of these servers allows the attackers to execute attacks more effectively. Two types of attacks have been analyzed on these servers.

- Brute Force (BF) attacks
- Byte For Byte (BFB) attacks

The detailed analysis of these attacks is given in the following sub-sections.

1) Brute Force (BF) attacks: Given the behavior of Forking servers, the attacker can implement brute force attacks on the encryption key. So, the attacker can attack an infinite number of times because after each crash the services get restarted and each time if the guess is wrong it can discard the present combination and move on to the next. This process in statistics is also termed 'sampling without replacement'. The attack is modeled using Bernoulli's trial equation with the total of 'k' attempts made and the probability of success for each attempt is 'p'. To find the number of attempts till we have a 100% success the process is modeled as uniform distribution.

Table III presents the comparative results for the proposed and XOR encryption techniques for forking-based server brute force attacks with the CDF plot presented in Fig. 5. For the

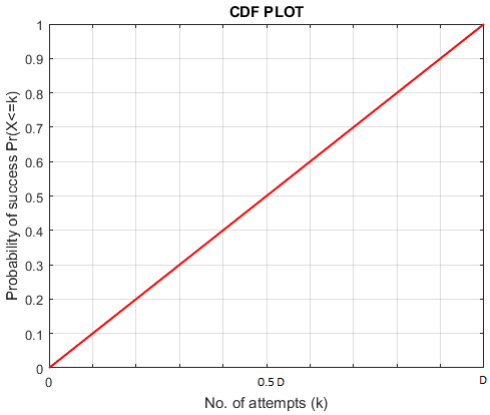

Fig. 5: CDF Plot for Brute force attack

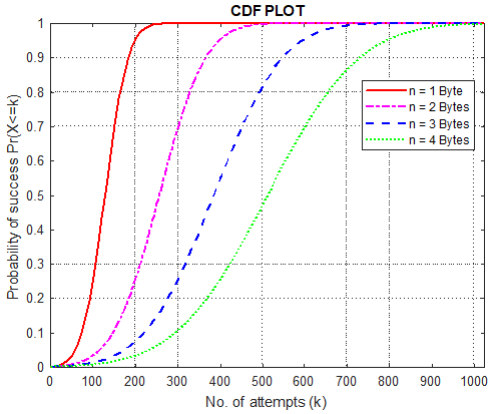

Fig. 6: CDF plot for byte-for-byte attack on XOR encryption

TABLE IV: Forking Server (Byte For Byte) results comparison

	XOR Encryption	Proposed Algorithm
$E[X]/\mu$	$256n/2$	$65536n^2/2$
$Var[X]/\sigma^2$	$256n^2/12$	$65536n^2/12$
CDF	$Pr(X \leq k) = 1/2(1 + erf\frac{k-\mu}{\sqrt{2\pi\sigma^2}})$	
100%	$256n$	$(256 * 256)n^2$
$95\%(3\mu)$	$0.95(256n)$	$0.95(256 * 256)n^2$
$50\%(0.68\mu)$	$256n/2$	$(256 * 256)n^2/2$

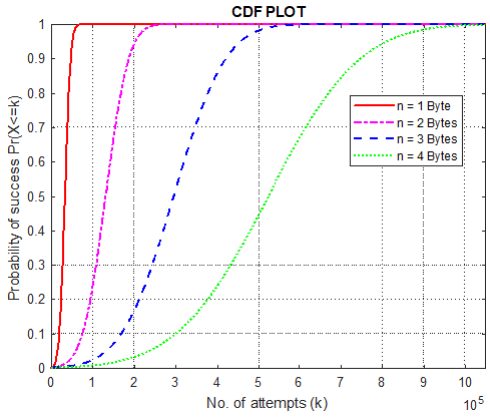

Fig. 7: CDF plot for byte-for-byte attack on LsStk algorithm

attack to be 100% successful the XOR encryption is more vulnerable than the proposed protection scheme with the factor of 2^{32}. The factor 2^{32} is the same for the case of 50% and 95% success rate because of the uniform distribution.

2) Byte For Byte (BFB) Attacks: In this attack, the adversary tries to exploit by guessing one byte at a time. So, when exploiting the buffer overflow vulnerability, the attacker only changes one byte of the address. If the process crashes the attacker will change and try with another byte. So, to get a 100% success rate for 1st byte it will take 256 attempts, as the new child process has the same secrets [38]. Therefore, for a 32-bit system architecture the required number of attempts becomes 4*256 and 8*256 for a 64-bit operating system. These attacks are modeled as Irwin Hall's normal distribution.

Table IV presents the comparative results for the proposed and the XOR encryption techniques for forking-based server Byte for Byte attack. The CDF plot for the proposed and XOR algorithm is also presented in Fig. 6 and Fig. 7 respectively. From the results, we can infer that for these attacks to be 100% successful XOR encryption is more vulnerable than the proposed prevention technique by the factor of $256n$. The factor is the same for the 50% success rate.

VI. Conclusion and Future Works

Buffer overflow is one of the top vulnerabilities in the list of security threats and alerts. These vulnerabilities if exploited could result in giving complete administrative access to the attacker/adversary which might result in deadly consequences. Many software and hardware-based prevention techniques have been proposed in the literature to prevent such exploitation. Our presented study focused on the direct overflow type of vulnerability where the adversary corrupts the return address to effectively change the control flow of the program to its desired location. We proposed a lightweight algorithm (LsStk) with the Fiestal Cipher design model for encryption and decryption of the return address to prevent them from being exploited by the adversary.

As a comparison, a statistical analysis was performed on the proposed technique and XOR encryption. The results from the comparative analysis show that the proposed technique is less vulnerable to trial and test, brute force, and byte-for-byte attacks than XOR encryption. The proposed prevention technique like XOR encryption can be implemented at the CPU architecture level or by simple modification to the compiler's prologue and epilogue code. The introduced overhead using the proposed prevention technique will be higher than XOR encryption but as the statistical analysis suggests it is a good trade-off over security.

LsStk being a lightweight algorithm is suitable to be implemented for contained IoT devices. One of the future works would be testing the algorithm on contained embedded devices and comparing the induced overhead using benchmarking techniques against available algorithms. LsStk could also be implemented at the compiler level e.g. as a patch to the GCC compiler or it could introduced at the hardware level. Implementation and induced overhead for the software and hardware levels are also left as future work.

REFERENCES

[1] M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the internet virus of november 1988," in *Proceedings. 1989 IEEE Symposium on Security and Privacy*, 1989, pp. 326–343.

[2] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro, "The cornell commission: on morris and the worm," *Communications of the ACM*, vol. 32, no. 6, pp. 706–709, 1989.

[3] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu, "Risk assessment of buffer" heartbleed" over-read vulnerabilities," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 555–562.

[4] O. S. Advisory, "X.509 Email Address 4-byte Buffer Overflow (CVE-2022-3602)," 2022. [Online]. Available: https://www.openssl.org/news/secadv/20221101.txt

[5] B. M. Padmanabhuni and H. B. K. Tan, "Defending against buffer-overflow vulnerabilities," *Computer*, vol. 44, no. 11, pp. 53–60, 2011.

[6] M. Ahsan, N. Ahmad, and H. M. W. Badar, "Simulation of Solar angles for maximizing Efficiency of Solar Thermal Collectors," in *2019 3rd International Conference on Energy Conservation and Efficiency (ICECE)*, 2019, pp. 1–5.

[7] M. Ahsan, M. H. Rais, and I. Ahmed, "Sok: Side channel monitoring for additive manufacturing - bridging cybersecurity and quality assurance communities," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 1160–1178.

[8] B. Imran, B. Afzal, A. H. Akbar, M. Ahsan, and G. A. Shah, "Misa: Minimalist implementation of onem2m security architecture for constrained iot devices," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[9] M. H. Rais, M. Ahsan, and I. Ahmed, "Fromepp: Digital forensic readiness framework for material extrusion based 3d printing process," *Forensic Science International: Digital Investigation*, vol. 44, p. 301510, 2023, selected papers of the Tenth Annual DFRWS EU Conference. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281723000112

[10] M. H. Rais, M. Ahsan, V. Sharma, R. Barua, R. Prins, and I. Ahmed, "Low-magnitude infill structure manipulation attacks on fused filament fabrication 3d printers," in *International Conference on Critical Infrastructure Protection*. Springer, 2022, pp. 205–232.

[11] A. Muhammad, B. Afzal, B. Imran, A. Tanwir, A. H. Akbar, and G. Shah, "onem2m architecture based secure mqtt binding in mbed os," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019, pp. 48–56.

[12] G. Mullen and L. Meany, "Assessment of buffer overflow based attacks on an iot operating system," in *2019 Global IoT Summit (GIoTS)*, 2019, pp. 1–6.

[13] P. Silberman and R. Johnson, "A comparison of buffer overflow prevention implementations and weaknesses," *IDEFENSE, August*, 2004.

[14] P. Team, "Address space layout randomization (ASLR)," 2013. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[15] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298–307.

[16] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[17] Microsoft Corporation, "Data Execution Prevention." [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx

[18] N. Stojanovski, M. Gusev, D. Gligoroski, and S. J. Knapskog, "Bypassing data execution prevention on microsoftwindows xp sp2," in *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE, 2007, pp. 1222–1226.

[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[20] G. Richarte *et al.*, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.

[21] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Dynaguard: Armoring canary-based protections against brute-force attacks," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 351–360.

[22] D. A.-H. Shehab and O. A. Batarfi, "Rcr for preventing stack smashing attacks bypass stack canaries," in *2017 Computing Conference*. IEEE, 2017, pp. 795–800.

[23] S. S. Vendicator, "A stack smashing technique protection tool for linux," *World Wide Web, http://www. angelfire. com/sk/stackshield/info. html*, 2000.

[24] Bulba and Kil3r, "Bypassing STACKGUARD and STACKSHIELD," 2000. [Online]. Available: http://phrack.org/issues/56/5.html

[25] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguardtm: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, 2003, pp. 91–104.

[26] G. Zhu and A. Tyagi, "Protection against indirect overflow attacks on pointers," in *Second IEEE International Information Assurance Workshop, 2004. Proceedings.* IEEE, 2004, pp. 97–106.

[27] G. Zhu, "Protection against overflow attacks," in *Retrospective Theses and Dissertations*. Iowa State University, USA, 2006, p. 3042.

[28] R. L. Rivest, "The rc5 encryption algorithm," fast software encryption, lncs 1008, b. preneel, ed," 1995.

[29] B. B. Madan, S. Phoha, and K. S. Trivedi, "Stackoffence: a technique for defending against buffer overflow attacks," in *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*, vol. 1. IEEE, 2005, pp. 656–661.

[30] M. A. Butt, Z. Ajmal, Z. I. Khan, M. Idrees, and Y. Javed, "An in-depth survey of bypassing buffer overflow mitigation techniques," *Applied Sciences*, vol. 12, no. 13, p. 6702, 2022.

[31] S. Sayeeshwari and E. Prabhu, "A simple countermeasure to mitigate buffer overflow attack using minimalistic hardware-integrated software simulation for fpga," in *2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2022, pp. 1–4.

[32] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 2004, pp. 209–220.

[33] Z. Shao, Q. Zhuge, Y. He, and E.-M. Sha, "Defending embedded systems against buffer overflow via hardware/software," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.* IEEE, 2003, pp. 352–361.

[34] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, and E.-M. Sha, "Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 443–453, 2006.

[35] C. Liu, Y.-J. Wu, J.-Z. Wu, and C. Zhao, "A buffer overflow detection and defense method based on risc-v instruction set extension," *Cybersecurity*, vol. 6, no. 1, p. 45, 2023.

[36] D. Fu and F. Shi, "Buffer overflow exploit and defensive techniques," in *2012 Fourth International Conference on Multimedia Information Networking and Security*. IEEE, 2012, pp. 87–90.

[37] H. M. Gisbert and I. Ripoll, "On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows," in *2014 IEEE 13th International Symposium on Network Computing and Applications*. IEEE, 2014, pp. 145–152.

[38] pi3'Zabrocki, A, "Scraps of notes on remote stack overflow exploitation," 2010.