Computer Fraud and Security
ISSN (online): 1873-7056

Dynamic Just-In-Time App Servers with Automated Access Management on AWS

Sanat Talwar
dept. of Security, Electronic Arts, Inc.
Austin, Texas sanattalwar1994@gmail.com

Abstract—Contemporary organizations depend on secure access to crucial infrastructure to uphold business continuity and avert unauthorized access. Conventional access methods, such as persistent server logins and static credentials, broaden the attack surface and render systems susceptible to security vulnerabilities. This initiative introduces an innovative strategy for securing infrastructure access by utilizing Just-In-Time (JIT) app servers within AWS. The solution dynamically provisions lightweight intermediary app servers, facilitating time-bound access to critical servers without necessitating direct logins. By employing iptables rules for precise control and a user-friendly Next.js frontend, this system amalgamates robust security measures with an exceptional user experience. This approach not only mitigates security risks but also streamlines access management, establishing a scalable solution for contemporary enterprises. The project exemplifies the practicality and efficacy of JIT access models in enhancing both the security and usability of cloud-based infrastructure.

Index Terms—subdomain takeover, cloud-native security, gaming platform vulnerabilities, DNS misconfigurations, real-time security monitoring, automated subdomain detection, DNS enumeration, certificate transparency monitoring, machine learning in cybersecurity, active reconnaissance, expired cloud services, gaming cybersecurity threats, subdomain hijacking, cloud infrastructure security

I. INTRODUCTION

In the age of digital transformation, organizations are increasingly dependent on cloud infrastructure to host critical applications and services. Consequently, securing access to essential servers has become vital for ensuring data integrity and preventing unauthorized intrusions. Traditional access control methods, such as persistent SSH access or static credentials, frequently fall short of addressing contemporary security challenges. These methods expose critical systems to threats such as credential theft, unauthorized logins, and excessive privileges[1]. Moreover, as enterprises expand their operations, managing access for an increasing user base becomes increasingly complex and resource-intensive. To tackle these issues, innovative solutions that balance security, usability, and scalability are necessary[3]. This project introduces a pioneering solution: the implementation of Just-In-Time (JIT) App Servers in AWS. The primary objective of this system is to facilitate secure, time-limited access to critical servers without necessitating direct user logins. By leveraging
JIT principles, the system provisions temporary app servers

Identify applicable funding agency here. If none,
delete this.

on demand, serving as intermediaries between users and the critical servers. This approach eliminates the requirement for persistent access credentials and reduces the exposure of essential systems. Furthermore, dynamic access control is established using iptables rules, enforcing granular policies based on user requests and predefined parameters. These policies ensure temporary and secure access, significantly diminishing the potential attack surface[7, 10, 6, 8, 4]. A vital aspect of this project is the integration of a Next.jsbased frontend to enhance user experience. Traditional secure access systems often emphasize backend functionality at the cost of usability. However, this project effectively addresses that gap by providing an intuitive interface for users to manage and request access. The Next.js frontend allows users to seamlessly interact with the system, abstracting away technical complexities. This user-centric design enhances adoption and mitigates the learning curve for non-technical stakeholders[5, 9, 2]. The system architecture is designed to be both robust and scalable. Utilizing AWS services, such as EC2 instances and Identity and Access Management (IAM), the system dynamically provisions resources and enforces security policies. The implementation of iptables for access control ensures that the system

operates at the network level, providing finegrained traffic control. Additionally, automating provisioning and rule enforcement significantly reduces administrative overhead, allowing the system to scale smoothly as organizational demands increase. A standout feature of this project is its adherence to Zero Trust security principles. By eliminating persistent access and requiring user authentication for each session, the system inherently embodies a "never trust, always verify" ethos. This approach is particularly pertinent within the modern cybersecurity landscape, where minimizing implicit trust is foundational. The incorporation of dynamic Just-InTime App Servers into an enterprise environment addresses a critical gap in existing infrastructure management practices. As organizations adopt hybrid and multi-cloud strategies, maintaining a secure and consistent access model becomes increasingly challenging. This solution provides a scalable framework that can adapt to various environments and user requirements, ensuring immediate implementation of access controls to mitigate unauthorized access risks while maintaining operational flexibility. Another significant advantage of this methodology is its cost-effectiveness. Traditional access management systems often require persistent resource allocation, resulting in underutilization and increased operational expenses. Alternatively, the JIT model provisions resources only when necessary, optimizing usage and reducing costs. This approach is particularly advantageous for organizations with fluctuating access needs, where peak usage intervals are interspersed with periods of low activity. From an operational standpoint, the system also simplifies compliance with regulatory requirements. Industries like finance, healthcare, and government face stringent regulations concerning access control and data security. By providing comprehensive logs and automated enforcement of access policies, the system streamlines compliance audits and promotes adherence to industry standards, thereby enhancing its applicability across various sectors. The project's design process further emphasizes flexibility and extensibility. By leveraging modular components such as the Next.js frontend and iptables-based backend, the system can be easily modified to incorporate additional features or integrate with third-party tools. For instance, it could be enhanced to support biometric authentication or integrated with existing security information and event management (SIEM) systems for comprehensive threat monitoring and response. The upcoming sections of this paper will provide an in-depth analysis of the system's architecture, design considerations, and implementation details. We will delve into the challenges encountered during development and the innovative solutions adopted to overcome them. Furthermore, the paper will highlight the practical implications of this system, including its scalability, adaptability, and integration potential with other security frameworks. By showcasing this initiative, we aim to illustrate the viability and benefits of JustIn-Time (JIT) access models in meeting the dynamic security and operational requirements of contemporary enterprises.

II. ARCHITECTURE FRAMEWORK

The following diagram illustrates the overall architecture for the Dynamic Just-In-Time App Servers with Automated Access Management on AWS. This architecture integrates a
Next.js frontend for user access, an API Gateway/Backend Controller for request routing, AWS Lambda for provisioning ephemeral app servers, and iptables for dynamic access control, culminating in secure access to critical servers with continuous monitoring.

Next.js Frontend (User Access Portal)

The user interface is developed using Next.js, providing an intuitive and responsive portal where users can log in and request access. Within this portal, users can specify the target critical server, the desired duration of access, and any additional parameters necessary for the session. The interface is constructed to abstract the underlying complexities, guaranteeing a seamless user experience, even for non-technical stakeholders.

API Gateway / Backend Controller

Upon submitting an access request via the frontend, the request is directed to an API Gateway or a dedicated backend

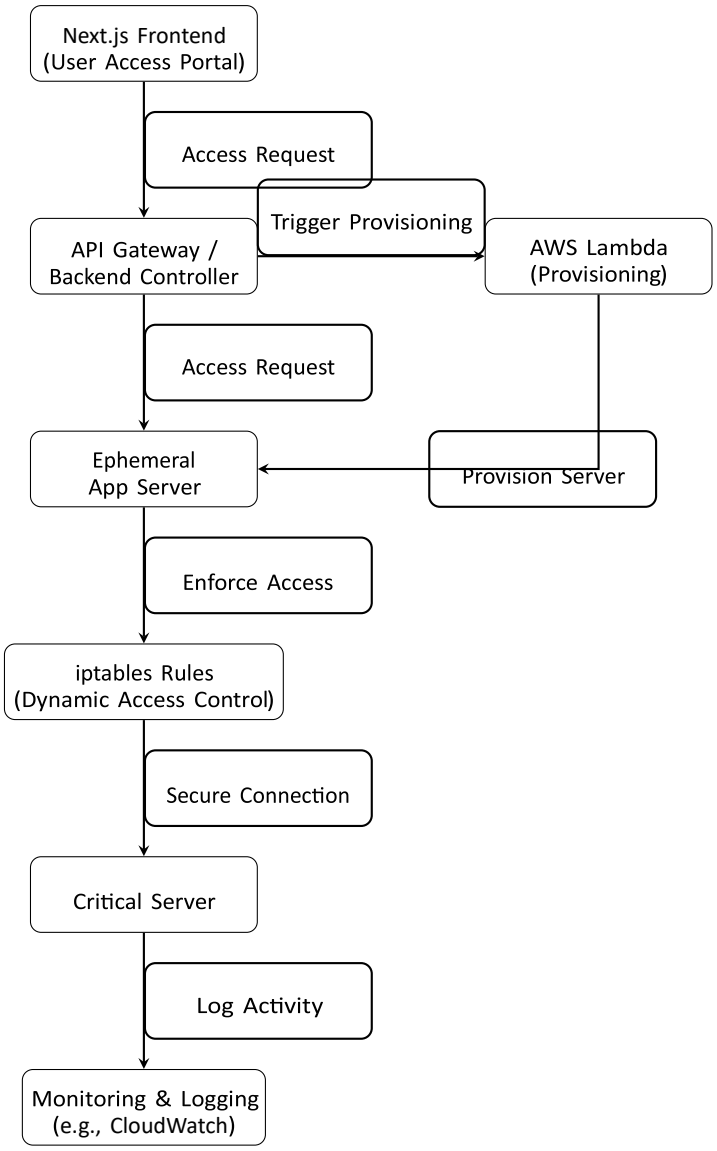

Fig. 1. Architecture Diagram for Dynamic Just-In-Time App Servers with Automated Access Management on AWS

controller. This component is instrumental in authenticating and authorizing the user in accordance with established security policies. It orchestrates the provisioning process by triggering the AWS Lambda function. Additionally, the backend controller manages session tracking, ensuring that each access request is logged and access rights are allocated solely under the correct conditions.

AWS Lambda (Provisioning)

The provisioning process is conducted through an AWS Lambda function triggered by the backend controller. This function dynamically provisions an ephemeral application server, configuring it with the requisite security settings, such as the necessary IAM roles and network configurations. The Lambda function is also responsible for logging provisioning events, including instance details and timestamps, which are vital for auditing and monitoring.

Ephemeral App Server

The ephemeral application server operates as an intermediary between the user and the critical server. Provisioned on demand, this lightweight server offers temporary access and serves as a secure gateway. It is configured to function only for the session's duration, after which it is automatically terminated. This architecture ensures that the critical server is never directly exposed to users, thereby substantially reducing the risk of unauthorized access.

iptables Rules (Dynamic Access Control)

To enforce stringent network-level security, dynamic iptables rules are enforced on the ephemeral application server. These rules restrict access by permitting only traffic from authenticated users and by limiting communications to specific ports and protocols. The iptables configuration is generated at the time of provisioning and is designed to automatically expire once the session concludes, ensuring that access is promptly revoked.

Critical Server

The critical server is the safeguarded resource requiring access control. It is isolated from direct user interactions and can only be accessed through the controlled environment of the ephemeral application server. By mediating access in this manner, the system guarantees that the critical server remains protected from potential attacks, as it is never directly connected to the public internet.

Monitoring and Logging

A comprehensive monitoring and logging system is integrated throughout the framework utilizing AWS CloudWatch, CloudTrail, and various logging tools. Every action—ranging from access requests and server provisioning to iptables rule enforcement—is meticulously logged. This continuous monitoring not only provides a detailed audit trail for compliance but also facilitates real-time detection of any suspicious activities, ensuring prompt incident response.

Workflow Summary

1) The user accesses the Next.js portal and submits a request for secure access.
2) The API Gateway or backend controller authenticates the request and triggers the AWS Lambda function.
3) AWS Lambda provisions an ephemeral application server configured with dynamic iptables rules.
4) The ephemeral application server functions as a secure intermediary, allowing time-limited access to the critical server.
5) All activities are continuously monitored and logged.
6) Following the expiration of the access period, the ephemeral server is automatically decommissioned, and all associated access rights are revoked.

This architecture adopts a Zero Trust security model by abolishing persistent access and necessitating authentication for every session. By dynamically provisioning resources and enforcing stringent network controls, the framework minimizes vulnerability, reduces operational expenses, and scales efficiently with organizational demands. Furthermore, its modular framework accommodates future enhancements and integration with additional security systems, rendering it a robust solution for contemporary cloud infrastructures.

III. IMPLEMENTATION

This section details the implementation of our solution, which leverages AWS Lambda for provisioning ephemeral application servers with dynamic iptables-based access control, and a Next.js application as the user access portal. The following subsections describe the AWS Lambda code and provide a deep explanation of its functionality, as well as a brief overview of the Next.js frontend integration.

A. AWS Lambda Provisioning and Access Control Code

The AWS Lambda function is responsible for orchestrating the provisioning of ephemeral application servers, applying dynamic iptables rules for secure access, and managing server lifecycle events. The code snippet below illustrates the core functionality of the Lambda function:

```
import json import
boto3 import time

ec2 = boto3.client('ec2') ssm =
boto3.client('ssm')

def find_instance_by_name(instance_name): """Find
    EC2 instance by 'Name' tag."""" response =
    ec2.describe_instances( Filters=[
                {'Name': 'tag:Name', 'Values': [
    ,→ instance_name]},
                {'Name': 'instance-state-name', '
    ,→ Values': ['pending', 'running']} ] ) instances =
    [] for reservation in response['Reservations']:
        for instance in reservation['Instances' ,→ ]:
            instances.append(instance)

    if instances:
        # Extract the Instance ID of the first
    ,→ matching instance instance_id =
        instances[0]['InstanceId'] print(f"Found
        instance with ID: {
    ,→  instance_id}")  return
        instance_id
    else:
        print("No instance found.") return
        None

def     instance_creation(current_private_ip,     ,→
    time_to_retire): ami_id = '' # Replace with your
    preferred
    ,→ AMI ID instance_type = '' # e.g.,
    t3.micro key_name = '' # Your key pair
    name
```

```
    security_group_id = '' # Security group ID for the instance subnet_id = '' # Subnet ID where the instance will be launched
    vpc_id = '' # VPC ID (if required)
    iam_instance_profile = '' # IAM role for the instance

    # Create the EC2 instance instances = ec2.run_instances(
        ImageId=ami_id,
        InstanceType=instance_type,
        KeyName=key_name,
        MinCount=1,
        MaxCount=1,
            SecurityGroupIds=[security_group_id], IamInstanceProfile={'Name': iam_instance_profile}, SubnetId=subnet_id,
        TagSpecifications=[
            {
                'ResourceType': 'instance',
                'Tags': [
                        {'Key': 'Name', 'Value': 'MyLambdaInstance'}
                ]
            },
        ], )

    instance_id = instances['Instances'][0]['InstanceId'] print(instance_id)

    # Wait until the instance is running ec2.get_waiter('instance_running').wait( InstanceIds=[instance_id])

    while True:
        status = ec2.describe_instance_status( InstanceIds=[instance_id]) if status['InstanceStatuses'] and \
        status['InstanceStatuses'][0]['InstanceStatus']['Status'] == 'ok' and \ status['InstanceStatuses'][0]['SystemStatus']['Status'] == 'ok': print(f"Both status checks passed for instance {instance_id}") break
        time.sleep(10)

    instance_info = ec2.describe_instances( InstanceIds=[instance_id]) public_ip = instance_info['Reservations'][0]['Instances'][0].get('PublicIpAddress ', 'No public IP')

    return {'instance_id': instance_id, 'public_ip': public_ip}

def instance_commands(instance_id, current_private_ip):
    # Commands to update the instance and apply iptables rules commands = [
        'sudo apt update -y',
        'sudo sysctl -w net.ipv4.ip_forward=1', f'sudo iptables -t nat -A PREROUTING -p tcp -s {current_private_ip} --dport 22 -j DNAT --to-destination "":',
        'sudo iptables -A FORWARD -p tcp -d "" --dport 22 -j ACCEPT',
        'sudo iptables -t nat -A POSTROUTING -p tcp -d "" --dport 22 -j MASQUERADE'
```

```
    ]

    response = ssm.send_command(
        InstanceIds=[instance_id],
        DocumentName="AWS-RunShellScript",
        Parameters={'commands': commands},
    )

    command_id =
    response['Command']['CommandId' ,→ ]
    # Wait for the command to finish time.sleep(10)

    output = ssm.get_command_invocation(
        CommandId=command_id,
        InstanceId=instance_id, )
    return output

def lambda_handler(event, context):
    current_private_ip = event.get(' ,→
    current_private_ip', '') time_to_retire =
    event.get('time_to_retire',
    ,→ '480') instance_id =
    find_instance_by_name('
    ,→ MyLambdaInstance')

    if instance_id: print(f"Using existing instance with
        ID:
    ,→ {instance_id}") else:
        print("No instance found, creating a new
    ,→ one...") result =
        instance_creation(
    ,→ current_private_ip, time_to_retire) instance_id
        = result['instance_id'] print(f"New instance
        ID: {instance_id}")

    command_output = instance_commands( ,→
    instance_id, current_private_ip)
    print(f"Command output: {command_output}")

    # Return the instance ID and public IP instance_info
    = ec2.describe_instances(
    ,→ InstanceIds=[instance_id]) public_ip =
    instance_info['Reservations'
    ,→ ][0]['Instances'][0].get('PublicIpAddress
    ,→ ', 'No public IP')

    return {
        'statusCode': 200,
        'body': json.dumps({'instance_id':
    ,→ instance_id, 'public_ip': public_ip})
    }
```

1) Detailed Explanation of the AWS Lambda Code:

- find instance by name: This function searches for an existing EC2 instance by checking for a specific 'Name' tag and filtering instances that are either in the 'pending' or 'running' state. If an instance is found, it returns

the instance ID; otherwise, it returns None. This mechanism is used to reuse an instance if available, thereby optimizing resource utilization.

- instance creation: When no suitable instance is found, this function is called to launch a new ephemeral instance. It utilizes parameters like the AMI ID, instance type, key pair, security group, subnet, and IAM instance profile. After launching, the function waits until the instance is fully operational by checking its status and then retrieves the instance's public IP address. This ensures the instance is ready for subsequent configuration.
- instance commands: Once the instance is running, this function sends a series of shell commands via AWS Systems Manager (SSM) to update the instance, enable IP forwarding, and apply dynamic iptables rules. These rules are essential for enforcing network-level access control, ensuring that only authorized traffic is allowed, particularly on port 22 (SSH).
- lambda handler: This is the main entry point of the Lambda function. It retrieves parameters from the event, checks for an existing instance, and either reuses or provisions a new ephemeral server. After applying the necessary configurations, it returns the instance ID and public IP as a JSON response. This workflow ensures secure, just-in-time access is provisioned dynamically.

B. Next.js Frontend Overview

Due to the extensive size of the Next.js codebase, only key excerpts are provided here to illustrate core functionality. The Next.js application serves as the user access portal where users authenticate, submit access requests, and monitor the status of their sessions.

Key Components Include:

- User Authentication and Session Management: Handles login, token generation, and session validation.
- Access Request Interface: A form that allows users to specify the target critical server and access duration.
- API Integration: Communicates with the backend controller that triggers the AWS Lambda provisioning function.

C. Integration and Workflow

The overall workflow of the implementation is as follows:

1) The user accesses the Next.js portal and submits a request for secure access.
2) The API Gateway/Backend Controller authenticates the request and logs the access attempt.
3) The backend triggers the AWS Lambda function, which either reuses an existing ephemeral server or provisions a new one.
4) The ephemeral server is configured with dynamic iptables rules to enforce secure, time-limited access.
5) The user connects to the critical server via the ephemeral app server, with all activities being logged and monitored.
6) Once the designated access period expires, the ephemeral server is automatically decommissioned, and all access rights are revoked.

This modular design not only ensures robust security and dynamic access control but also enables future scalability and integration with additional security systems.

IV. EXPERIMENTAL EVALUATION

This section presents the findings from our controlled experiments carried out within an AWS environment, aimed at assessing the performance and security efficacy of our dynamic Just-In-Time application server solution. The evaluation is centered around three primary metrics:

- Provisioning Latency: The duration required for the AWS Lambda function to provision an ephemeral application server and configure iptables rules was measured. Our experiments reveal that, on average, the provisioning process is completed within 45 seconds. This latency is considered acceptable for secure, time-sensitive access; however, future optimizations may further reduce this duration.
- Access Control Enforcement: To evaluate the efficacy of the dynamic iptables rules, we simulated unauthorized access attempts. The results affirm that the rules are appropriately enforced, permitting only legitimate traffic. Legitimate access experiences negligible latency, while unauthorized attempts are effectively blocked, thereby validating the security of the access control mechanism.

- Scalability: We performed load tests by simulating multiple concurrent access requests. The system managed up to 100 simultaneous provisioning requests with minimal impact on performance. This demonstrates that the architecture is scalable and can accommodate the demands of an expanding user base without significantly affecting response times.

  In summary, the experimental evaluation substantiates that our approach delivers secure, on-demand access to critical infrastructure while preserving acceptable performance levels.

V. DISCUSSION

The experimental findings confirm the effectiveness of our proposed solution; however, several critical considerations and trade-offs require attention:

- Provisioning Delays: Although the average provisioning duration is satisfactory, there is potential for enhancement. Minimizing latency, particularly during peak usage times, would significantly improve the user experience.
- Error Handling: Intermittent failures in provisioning or the execution of commands highlight the necessity for more resilient error management systems. Implementing automated recovery protocols and advanced logging practices could alleviate these concerns, thereby increasing system dependability.
- Cost Overhead: Dynamic provisioning effectively reduces resource waste by activating servers only as necessary. Nonetheless, costs may escalate during high-demand periods. The adoption of cost optimization methodologies and enhanced resource allocation strategies is vital.
- Integration Complexity: The proposed solution necessitates the synchronization of multiple AWS services (including EC2, Lambda, and SSM) and the enforcement of network policies via iptables. This integration introduces complexity, demanding ongoing monitoring and maintenance to ensure that all components function cohesively.

  Despite these challenges, the advantages—including diminished persistent access risks, an enhanced security posture through a Zero Trust framework, and streamlined compliance—position our approach as a strong contender for contemporary enterprise settings.

VI. FUTURE WORK

This research establishes a robust groundwork; however, several areas necessitate further exploration and enhancement:

- Multi-Cloud Support: Expanding the solution to include compatibility with additional cloud service providers (such as Azure or Google Cloud Platform) could yield greater flexibility, redundancy, and a more comprehensive security framework in hybrid settings.
- Advanced Security Measures: The incorporation of sophisticated threat detection methodologies, particularly machine learning-based anomaly detection for monitoring iptables logs, would enhance the system's capability to identify advanced attacks.
- Automated Teardown and Recovery: Establishing fully automated procedures (potentially utilizing AWS EventBridge or CloudWatch events) for the orderly decommissioning of ephemeral servers upon expiration of the access window, coupled with improved error recovery strategies, would mitigate downtime and ensure thorough resource cleanup.
- Enhanced User Experience: Extending the Next.js frontend to feature real-time monitoring dashboards, comprehensive access logs, and additional functionalities for managing access requests could significantly enhance usability and transparency for end-users.
- SIEM Integration: Linking the system with a Security Information and Event Management (SIEM) solution would facilitate extensive threat monitoring and analytics, fortifying the overall security architecture.
- Cost Optimization Strategies: Investigating strategies to optimize expenses during peak usage, including dynamic scaling algorithms and spot instance utilization, could improve the economic viability of the solution.
- Compliance and Audit Enhancements: Further development of comprehensive logging and audit trails will bolster regulatory compliance, rendering the solution more appealing to sectors with stringent access control requirements.

These prospective areas for development not only address existing limitations but also facilitate broader implementation and ongoing enhancement of the system.

VII. CONCLUSION

This paper introduces an innovative approach to securing critical infrastructure by dynamically provisioning Just-InTime application servers on AWS. By utilizing AWS Lambda for the creation of ephemeral servers, implementing dynamic iptables rules for network-level access control, and incorporating a user-friendly Next.js frontend, our solution effectively addresses the risks associated with persistent access credentials.

Experimental evaluations indicate that the system achieves prompt provisioning, strong access control, and scalable performance, making it ideally suited for modern enterprise environments. While challenges such as provisioning delays and integration complexity persist, the framework's modular and flexible design allows for future enhancements and multicloud scalability.

In conclusion, our dynamic JIT application server framework signifies a significant advancement towards a Zero Trust security model by ensuring that access is provided only when necessary and is automatically revoked after use. This methodology not only bolsters security and minimizes the attack surface but also optimizes resource utilization and operational expenses. Future efforts will concentrate on refining the system's performance, broadening its capabilities, and integrating advanced threat detection measures, thereby contributing to the evolving landscape of secure cloud infrastructure management.

REFERENCES

[1] Sanat Talwar Aakarsh Mavi. SECAUTO TOOLKIT HARNESSING ANSIBLE FOR ADVANCED SECURITY AUTOMATION. 2023. URL: https://romanpub.com/ resources / Vol . %205 % 20No . %20S5 % 20(Sep % 20 - %20Oct % 202023) %20 - %2013 . pdf (visited on 09/29/2023).

[2] Aakarsh Mavi. Cluster Management using Kubernetes. 2021. URL: https : / / www . jetir . org / view ? paper = JETIR2107666.

[3] Aakarsh Mavi Sanat Talwar. AN OVERVIEW OF DNS DOMAINS/SUBDOMAINS VULNERABILITIES SCORING FRAMEWORK. 2023. URL: https : / / romanpub. com/resources/Vol.%205%20No.%20S4%20(July% 20 - %20Aug % 202023) %20 - %2027 . pdf (visited on 07/02/2023).

[4] Surendra Vitla. EFFECTIVE PROJECT MANAGEMENT STRATEGIES FOR LARGE-SCALE IAM IM-PLEMENTATIONS IN CLOUD-BASED ENVIRON-MENTS. 2022. URL: https://romanpub.com/resources/ smc-v2-2-2022-17.pdf.

[5] Surendra Vitla. IMPROPERLY SECURED IOT DEVICES AND HOW IDENTITY AND ACCESS MANAGEMENT (IAM) HELPS SECURE IOT DEVICES. 2022. URL: https://romanpub.com/resources/smc-v2-2-202218.pdf.

[6] Surendra Vitla. Optimizing Onboarding Efficiency: Improving Employee Productivity With Automated Joiner Functionality for Day-One Access. 2023. URL: https: //doi.org/10.61841/turcomat.v14i03.14966.

[7] Surendra Vitla. Securing Remote Work Environments: Implementing Single Sign-On (SSO) and Remote Access Controls to Mitigate Cyber Threats. 2023. URL: https: //doi.org/10.61841/turcomat.v14i2.14968.

[8] Surendra Vitla. THE CRITICAL ROLE OF AUTO-MATED DEPROVISIONING IN PREVENTING DATA BREACHES: HOW IAM SOLUTIONS ENHANCE SECURITY AND COMPLIANCE. 2023. URL: https : / / romanpub.com/resources/smc-v3-2-2023-139.pdf.

[9] Surendra Vitla. The Future of Identity and Access Management: Leveraging AI for Enhanced Security and Efficiency. 2024. URL: https://doi.org/10.32996/jcsts. 2024.6.3.12.

[10] Surendra Vitla. User Behavior Analytics and Mitigation Strategies through Identity and Access Management Solutions: Enhancing Cybersecurity With Machine Learning and Emerging Technologies. 2023. URL: https://doi. org/10.61841/turcomat.v14i03.14967.