Dotenvx: Reducing Secrets Risk with Cryptographic Separation

Scott Motte
scott@dotenvx.com
www.dotenvx.com
June 18, 2025

Abstract. An ideal secrets solution would not only centralize secrets but also contain the fallout of a breach. While secrets managers offer centralized storage and distribution, their design creates a large blast radius, risking exposure of thousands or even millions of secrets. We propose a solution that reduces the blast radius by splitting secrets management into two distinct components: an encrypted secrets file and a separate decryption key.

1. Introduction

Modern software relies on secrets to operate—API keys, tokens, and credentials are essential for applications to interact with services like Stripe, Twilio, and AWS. The majority of these secrets are stored in platform-native secrets managers such as AWS Secrets Manager, Vercel Environment Variables, and Heroku Config Vars. These systems offer convenience by centralizing secrets and seamlessly injecting them into runtime environments. However, this centralization introduces significant risks. If breached, they expose all secrets stored within, resulting in a blast radius where thousands or even millions of secrets may be leaked. At the same time, alternatives such as .env files minimize blast radius but lack the safeguards necessary to prevent unauthorized access. Developers are left choosing between simplicity with higher risk or complexity with a larger blast radius.

What is needed is a secrets system based on hybrid cryptography instead of trust, allowing a developer to encrypt secrets without relying on any third party to remain secure. In this paper, we propose a solution to these risks using a library that decrypts an encrypted secrets file at runtime with a private key stored separately in the platform's secrets manager. This approach contains the blast radius of a breach while maintaining the simplicity of .env files. Even if one component—either the encrypted file or the secrets manager—is compromised, secrets remain secure. Only simultaneous access to both can expose them.

2. Secrets

We define a secret as a token or string value, typically issued by a third-party

service like Stripe, Twilio, or AWS, and used by applications to interact with their APIs. Secrets are essential for modern application functionality, enabling critical operations such as processing payments, sending emails, and accessing cloud resources.

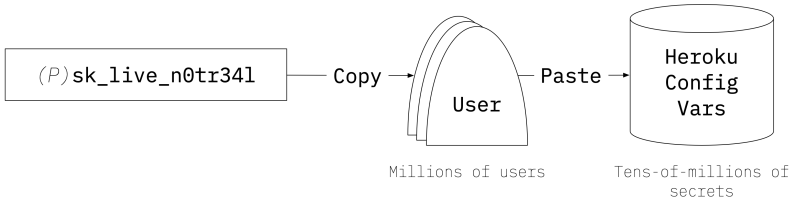
To ensure secrets are available to the application at runtime, they must be stored in a way that the codebase can access them.



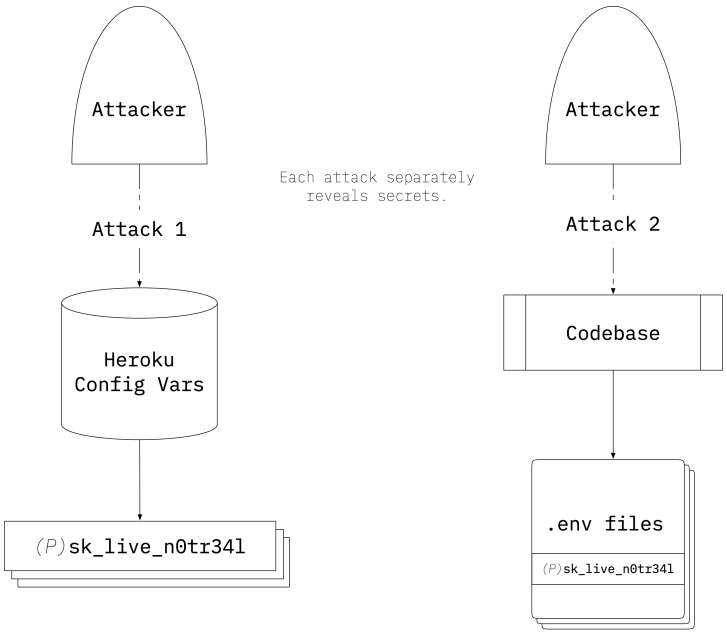
3. Storage

Not long ago, secrets were primarily stored as plaintext in code. If a codebase was breached, so were its secrets. The Twelve Factor App rightly introduced the concept of "strict separation of config from code," helping contain this fallout. If a codebase was breached, its secrets remained safe—since they were injected as environment variables. [1]

This, however, still required a place to store secrets at rest before injecting them into env. For development, .env files became the standard, and for production, platform-native secrets managers emerged. [2] Heroku Config Vars is the canonical example—providing both a CLI and UI to set production secrets. When code is deployed, Heroku Config Vars reads these secrets from storage and injects them into the running process as env. [3] Today, this remains the standard for all secrets managers, aside from certain file-based methods we address later in this paper.



These solutions work well enough but rely on trust—trust that a secrets manager will remain uncompromised, and trust that a .env file will not be exposed. However, history has repeatedly shown this trust to be fragile. [4][5][6] A single point of access—whether a secrets manager or a .env file—means that a breach at that point exposes every secret stored there.



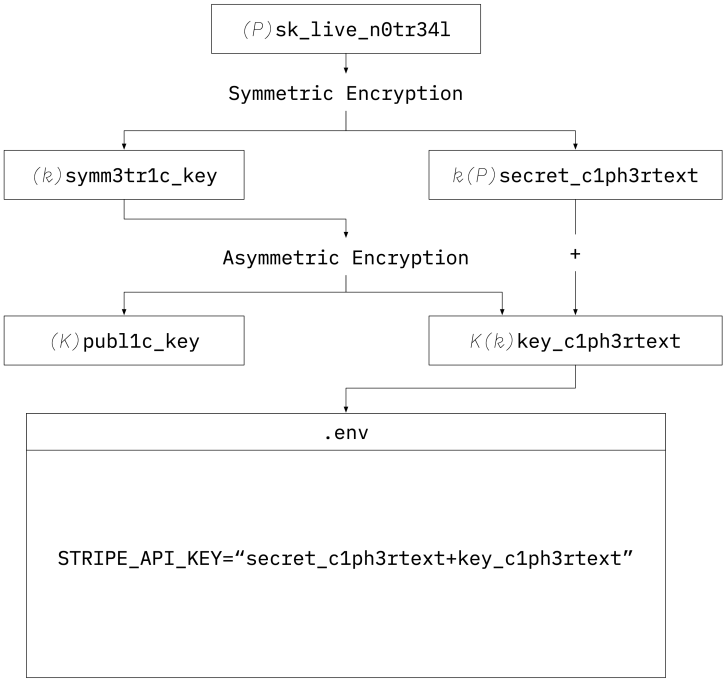
Secrets managers centralize storage, making them high-value targets. If breached, they reveal all secrets at once. Conversely, .env files are often scattered across local machines and repositories, increasing the likelihood of accidental exposure, but with a smaller blast radius.

4. Splitting Secrets

To reduce the blast radius of a breach, secrets should not exist in a single, retrievable location. Instead, they should be split into two parts. By ensuring that neither part is stored together, access to one does not grant access to the original secret.

Our solution uses Elliptic Curve Integrated Encryption Scheme (ECIES) to achieve this separation. [7] When a secret is encrypted, the process follows these steps:

1) A random symmetric key is generated for the secret.
2) The secret is encrypted using this key, producing a ciphertext.
3) The key itself is encrypted using a public key.
4) The encrypted key and ciphertext are stored together in a .env file.



At this point, the decryption key—the private key associated with the public key—is not stored in the codebase. Instead, it is kept separately in a .env.keys file or a platform-native secrets manager. This means:

- The .env file contains encrypted secrets but lacks the means to decrypt them.
- The .env.keys file holds the decryption key but contains no secret values.
- Neither the codebase nor the secrets manager alone can reconstruct the secrets—both are required.



By structuring secrets this way, we eliminate the risks associated with storing secrets in either the codebase or a secrets manager. A breach of one component is insufficient; an attacker would need access to both the encrypted .env file and the .env.keys file to recover the secrets.

5. Runtime

We will need a way to bring the two parts back together at runtime. A runtime library is designed to be installed anywhere and acts as a lightweight wrapper around the application process. Instead of secrets being known and injected by the platform-native secrets manager, the library retrieves the encrypted .env file, pulls only the private key from the .env.keys file or platform-native secrets manager, decrypts the secrets, and injects them into env just-in-time.

To securely handle this process, the runtime follows a straightforward sequence of operations [8]:

1) Retrieve private key from .env.keys file, platform-native secrets manager, or pre-existing env.
2) Retrieve encrypted .env file.
3) Decrypt encrypted .env file with private key.
4) Inject decrypted secrets into process env.
5) Application runs, accessing secrets from env.

Here is a reproducible example:



This approach ensures maximum compatibility, working in any environment where secrets are traditionally injected into env, while guaranteeing that secrets exist in plaintext only during runtime (within the process's isolated environment), never persisting in third-party storage or intermediary locations.

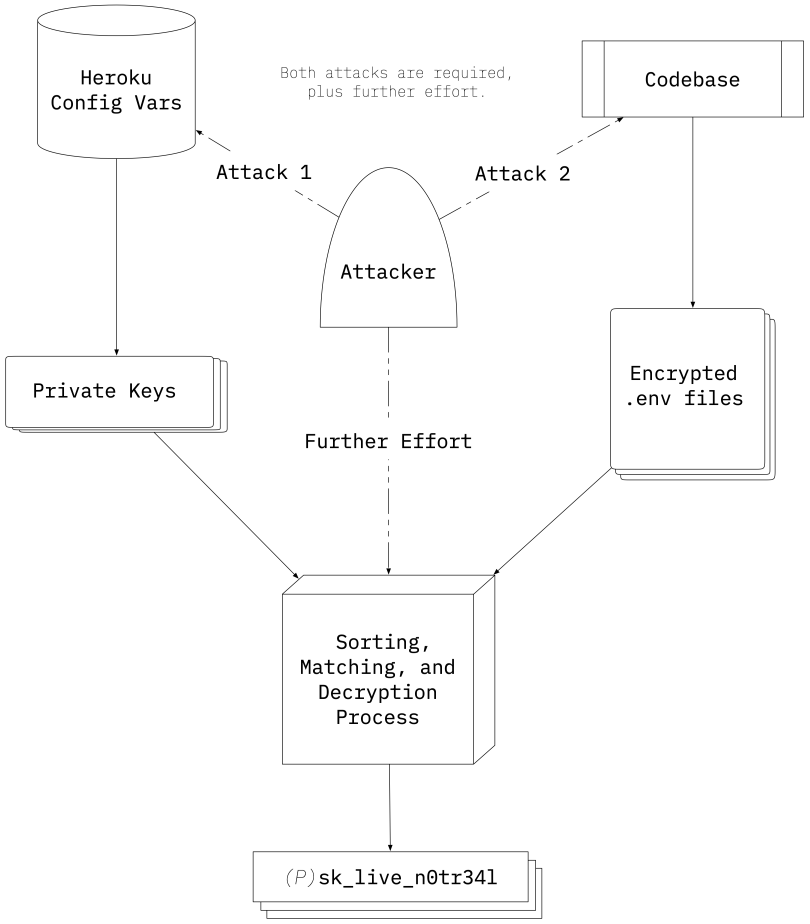
6. Dual Breach

Our solution significantly reduces the risks associated with isolated breaches. If the platform-native secrets manager is compromised, the attacker would gain access only

to the private key, which is useless without the encrypted secrets file. Conversely, if the codebase or .env file is exposed, the attacker would only obtain encrypted secrets, which cannot be decrypted without the private key. This separation ensures that a single point of failure does not expose plaintext secrets.

However, no system is entirely immune to all threats. In the case of a dual breach, where both the encrypted secrets file and the private key are compromised, the attacker can decrypt the secrets. That said, this setup forces an attacker to compromise two entirely separate security domains—such as application storage (where the .env file is committed) and infrastructure secrets management (where the private key is stored). In real-world attacks, breaching both independently controlled domains is significantly harder than compromising a single, centralized secrets manager.

Furthermore, even if an attacker gains access to both components, they may not be trivially linked–especially in complex systems. This added layer of separation makes attacks more difficult and increases the likelihood that detection mechanisms will intervene before decryption occurs.



7. Version Control

Secrets have traditionally been excluded from version control due to security risks, forcing teams to rely on manual distribution of .env files or third-party secrets managers. This lack of visibility contributes to secrets sprawl—where secrets end up scattered across local machines, shared documents, or chat messages, making them harder to track and secure. [9]

By encrypting .env files, secrets can now be committed safely alongside code, ensuring they follow the same versioning and history as application changes. Unlike full-file encryption approaches that obscure both keys and values, this method encrypts only the values, keeping the keys visible. This allows teams to track which secrets exist without exposing their actual contents, improving security without sacrificing visibility. [10]

Bringing encrypted secrets into version control naturally integrates them with pull request workflows, a core practice of modern software development. Changes to secrets become part of the code review process, ensuring that modifications are tracked, reviewed, and approved before merging. This introduces several key benefits:

- *Reduces Secrets Sprawl:* Secrets remain in a single, versioned location rather than being informally distributed.
- *Built-in Approval Process:* Changes to secrets require explicit review, preventing unauthorized modifications.
- *Collaboration Between Dev and Ops:* Developers can propose secret changes, while DevOps or security teams can review and approve them before they reach production.
- *Prevents Shadow Secrets:* Because every secret change is visible in pull request diffs, undocumented or improperly stored secrets are less likely to emerge. [11]



This structured approach not only minimizes the risk of secrets leaking but also enforces a clear and auditable process for managing them. Rather than being handled as an afterthought, secrets management becomes an integral part of software development—ensuring both security and operational efficiency.

8. Decryption At Access

While injecting secrets into env is the standard approach for most secrets managers, it is not without risks. Environment variables, though convenient, can be accessed by any process with sufficient privileges, making them a potential attack vector in case of a system compromise. Additionally, some file-based secrets management solutions, such as Docker Secrets, mitigate some risks by mounting secrets as files instead of using environment variables, but they still leave secrets exposed in plaintext at rest on the filesystem.

Our solution offers an alternative mode of operation—*decryption at access*. Instead of injecting decrypted secrets into env at process startup, this approach defers decryption until the moment a secret is needed. The runtime library reads the encrypted .env file, retrieves the corresponding private key from the secrets manager, and decrypts the secret just-in-time before returning it to the application. [12]

app.js

```
const dotenvx = require('@dotenvx/dotenvx')
const express = require('express')

const app = express()
const port = 3000

app.get('/', (req, res) => {
  const hello = dotenvx.get('HELLO') // decryption-at-access
  res.send(`Hello, ${hello}!`)
})

// simulates a badly controlled (leaky) log - leaking env
app.get('/leakylog', (req, res) => {
  // this will NOT leak because using `dotenvx.get`
(decryption-at-access) does not inject to env
  const env = JSON.stringify(process.env, null, 2)
  res.send(env)
})

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`)
})
```

This method provides additional security advantages:

- *Minimizes Secrets Exposure:* Secrets only exist in plaintext in memory for the brief moment they are being used, rather than persisting in environment variables or plaintext files.
- *Beyond Docker Secrets:* Unlike Docker Secrets, which stores secrets in plaintext on disk once retrieved, this approach ensures secrets remain encrypted at rest and only become readable when needed.
- *Defends Against Memory Scraping Attacks:* Because secrets do not persist as environment variables, attackers relying on inspecting process environments (e.g., env or /proc/PID/environ) will find nothing of value.
- *Enables Audit Logging:* Every decryption event can be logged, providing a traceable record of when and how secrets are accessed. This enhances observability and aids in detecting anomalous behavior or unauthorized access attempts.
- *Supports Dynamic Secret Reloading:* Since secrets are decrypted on demand, applications can reload them dynamically without restarting the process. This is particularly useful for long-running services, such as real-time voice or video applications, where secret rotation must occur without service disruption.

This is not the default mode, as most workflows prioritize performance and compatibility with existing env-based practices. However, for teams operating in high-security environments, decryption at access provides an extra layer of protection—ensuring secrets remain encrypted until the precise moment they are required, and never lingering in env or plaintext files.

9. Write-Only Access

Write-only access is *"the idea that I can change [a secret] or I can add [a secret] of some sort, but I can't necessarily read it back out again"*. [13] This property is useful for limiting exposure, ensuring that even those with permission to store or update secrets do not automatically gain the ability to retrieve them. It reduces the risk of accidental leaks and unauthorized access, enforcing a stricter separation of duties.

Our approach inherently enables write-only access–by default. The public key is embedded in the encrypted .env file, allowing anyone with access to encrypt new secrets while only the private key holder can decrypt them. This decouples secret storage from retrieval, preventing unauthorized decryption while enabling external systems, such as CI/CD pipelines or third-party services, to inject encrypted secrets without ever having read access to existing ones.

.env

```
#/--------------------[DOTENV_PUBLIC_KEY]--------------------/
#/            public-key encryption for .env files           /
#/       [how it works](https://dotenvx.com/encryption)      /
#/-----------------------------------------------------------/
DOTENV_PUBLIC_KEY="026d4945b6513baec60f68b207f203ba534fb54d2b0c9952557d240815e42a7d83"

# .env
# Database configuration
DB_HOST="encrypted:BMO83g2fEtr66gcFvUs2+/ZuccCQuBbZwSW3JfCLvoUiACmusxCbTfG2dvc2LxenPhUtgWapO8f9BCcBVAcTnMcrd3kndvk+acWytRjIWRUvsSezdD340/OT5EQgbqJtwXfuRz0i2t8PVA=="
DB_USER="encrypted:BBrXv55qxgA19sEqqNnZzS/C0WguVk6ROQmfxnGhBhafLoc0XwpKprk/J3hJCVq7s45WyBSXGUz9U9rHxCBeVkl27WFzzgZkDewX0gBLt+Cc37K0EVU2hZ1GPbax5mzpI5Jwwi65be6+"
```

10. Encryption

Here's a code-level breakdown of how our solution (using ECIES) encrypts each secret in a .env file with a unique ephemeral keypair while still being decryptable with a long-lived private key. [15]

1) Long-Term Keypair. Before encrypting secrets, a long-lived keypair is generated:

long_term_private_key = generate_private_key() # Place in .env.keys
long_term_public_key = derive_public_key(long_term_private_key) # Placed in .env

- The private key is never exposed.
- The public key is used in the encryption process.

2) Ephemeral Keypair. For each secret, a new ephemeral keypair is generated.

```
ephemeral_private_key = generate_private_key() # New per secret
ephemeral_public_key = derive_public_key(ephemeral_private_key)
```

• This provides cryptographic isolation between individual encrypted values, each using a uniquely derived AES-256-GCM key.

3) Shared Secret. Using the ephemeral private key and the recipient's long-term public key, we generate a shared secret:

```
shared_secret = diffie_hellman(ephemeral_private_key, long_term_public_key)
```

• This shared secret is NOT the final AES-256-GCM key yet–it still needs to be processed through a KDF.

4) Key Derivation. To produce a strong, fixed-length AES-256-GCM key, we apply SHA-256 (or HKDF) to the shared secret:

```
aes_key = derive_aes_key(shared_secret) # SHA-256 based KDF
```

• This ensures the AES-256-GCM key is uniformly random and resistant to attacks.
• If needed, HKDF can also be used instead of PBKDF2.

5) Encryption. With the derived AES-256-GCM key, we encrypt the secret:

```
ciphertext = aes_encrypt(aes_key, secret_value)
```

6) Combine. The ephemeral public key is prepended to the encrypted value, so the decryption process can reconstruct the AES-256-GCM key later:

```
final_ciphertext = ephemeral_public_key || ciphertext
```

• The ephemeral public key is used to recompute the shared secret during decryption.

11. Decryption

Here's a code-level breakdown of the decryption process.

1) Extract. Parse the ephemeral public key & ciphertext from the final ciphertext.

```
ephemeral_public_key = extract_first_part(final_ciphertext)
ciphertext = extract_remaining_part(final_ciphertext)
```

2) Shared Secret. The recipient reconstructs the shared secret using their long-term

private key and the ephemeral public key:

shared_secret = diffie_hellman(long_term_private_key, ephemeral_public_key)

3) Key Derivation. The same KDF is applied to derive the AES-256-GCM key:

aes_key = derive_aes_key(shared_secret)

• Since both encryption and decryption use the same KDF, the AES-256-GCM key matches.

3) Decryption. Decrypt the secret using AES-256-GCM:

plaintext_secret = aes_decrypt(aes_key, ciphertext)

• Secret is successfully decrypted.

12. Rotation

This encryption mechanism unlocks rotation capabilities, allowing secrets to be re-encrypted with newly generated key pairs while remaining secure and accessible. The rotate command seamlessly re-encrypts all stored secrets using a fresh asymmetric key pair. This process enhances security without requiring changes to the underlying secrets themselves. While rotation here does not modify secrets from external providers like Twilio, users can pair our solution's rotation with provider-specific secret rotation for a more comprehensive security strategy. [16]

13. Calculations

Our approach significantly reduces the risk of secret exposure by requiring an attacker to breach two independent security domains—application storage (encrypted .env file) and infrastructure secrets management (private key). This dual-layered security makes attacks twice as hard compared to a traditional secrets manager.

• *Traditional Secrets Manager Risk:* If a centralized secrets manager is breached, 100% of its stored secrets are immediately exposed.
• *Dotenvx Dual-Breach Risk:* Since an attacker must breach both the encrypted .env file and the private key store, the probability of full exposure is lower.

Given estimated annual breach probabilities:

• Traditional secrets manager breach risk = 1% per year
• Codebase leakage risk (exposed .env file) = 0.5% per year
• Dual breach risk (both events happening together) = $0.01 \times 0.005 = 0.00005$

(0.005% per year)

We see that the probability of a breach exposing secrets in a traditional secrets manager is 200× higher than with Dotenvx. This means Dotenvx reduces the likelihood of total secret exposure by 99.5% compared to centralized storage.

Over different time spans, the probability of full secret exposure is:

Years	Traditional Secrets Manager Risk	Dotenvx Risk (Dual Breach)
1	1.00%	0.005%
5	4.90%	0.025%
10	9.56%	0.049%
20	18.21%	0.098%

By requiring two independent breaches rather than a single-point of failure, Dotenvx effectively mitigates risk by a factor of 200×, making secret exposure significantly less likely over time.

While precise annual breach probabilities can vary and are challenging to pinpoint, the 1% risk for traditional secrets managers and 0.5% risk for codebase leaks used in our calculations are hypothetical estimates for illustrative purposes. These figures are intended to demonstrate the enhanced security posture achieved through our dual-layered approach. It's important to note that actual breach probabilities depend on numerous factors, including an organization's security measures, industry, and threat landscape. Therefore, while our example provides a conceptual understanding of risk reduction, organizations should assess their specific environments to determine accurate risk metrics.

References

[1] A. Wiggins, "The Twelve-Factor App", https://12factor.net, 2011.

[2] D. Dollar, "How to specify environment?", https://github.com/ddollar/foreman/issues/17, 2011.

[3] A. Wiggins, "Config Vars for Deploy-Specific Settings", https://blog.heroku.com/config-vars, 2009.

[4] R. Zuber, "CircleCI security alert: Rotate any secrets stored in CircleCI", https://circleci.com/blog/january-4-2023-security-alert, 2023.

[5] NIST, "CVE-2021-41077", https://nvd.nist.gov/vuln/detail/CVE-2021-41077, 2021.

[6] M. Kelley, S. Johnstone, W. Gamazo, N. Quist, "Leaked Environment Variables Allow Large-Scale Extortion Operation in Cloud Environments", https://unit42.paloaltonetworks.com/large-scale-cloud-extortion-operation/, 2024.

[7] V. Shoup, "A Proposal for an ISO Standard for Public Key Encryption", https://www.shoup.net/papers/iso-2_1.pdf, 2001.

[8] S. Motte, "Runtime Example", https://github.com/dotenvx/runtime-example, 2025.

[9] A. Dadgar, "What is "secret sprawl" and why is it harmful?", https://www.hashicorp.com/en/resources/what-is-secret-sprawl-why-is-it-harmful, 2018.

[10] G. Lederrey, "My griefs with hiera-gpg", https://slashdevslashrandom.wordpress.com/2013/06/03/my-griefs-with-hiera-gpg/, 2013.

[11] J. Marshall, "A guide to developer secrets and shadow IT for security teams", https://blog.1password.com/secrets-management-for-developers/, 2024.

[12] S. Motte, "Decryption At Access Example", https://github.com/dotenvx/decryption-at-access-example, 2025.

[13] L. Rice, "Your Secret's Safe with Me. Securing Container Secrets with Vault", https://youtu.be/j3QJRdiTr1I?t=270, 2017.

[14] W. Diffie, M. Hellman, "New Directions in Cryptography", https://ee.stanford.edu/~hellman/publications/24.pdf, 1976.

[15] W. Li, "Mechanism and Implementation (ECIES) in JavaScript", https://github.com/ecies/js/blob/master/DETAILS.md, 2025.

[16] J. Leon, "Deleting leaked API keys isn't a solution", https://trufflesecurity.com/blog/remediate-leaked-api-keys-with-key-rotation, 2023.