What follows is a working draft of dotenvx's whitepaper.

Please leave any thoughts or comments at:

https://github.com/dotenvx/dotenvx/issues/526

Dotenvx: A Cryptographic Approach to Reducing Secrets Risk

Scott Motte
mot@dotenvx.com
www.dotenvx.com

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

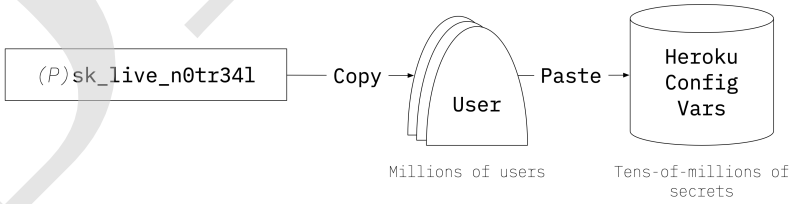
service like Stripe, Twilio, or AWS, and used by applications to interact with their APIs. Secrets are essential for modern application functionality, enabling critical operations such as processing payments, sending emails, and accessing cloud resources.

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

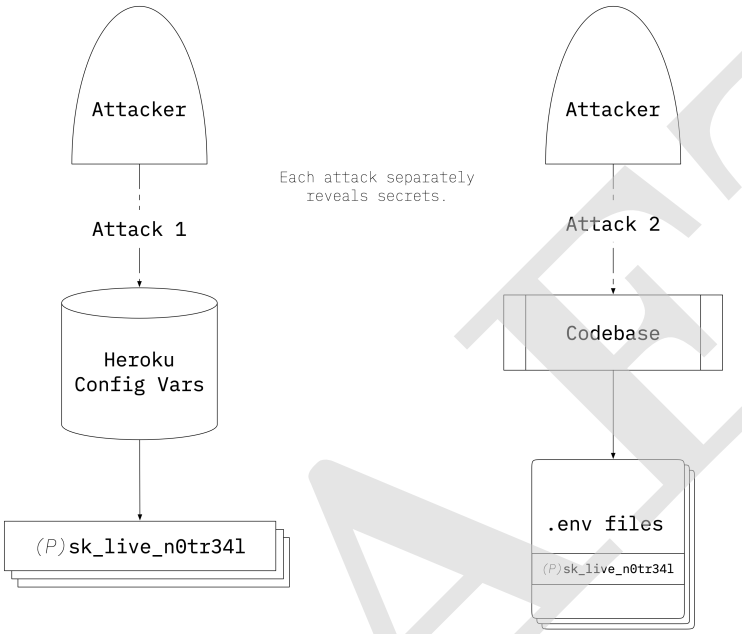
3. Storage

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.



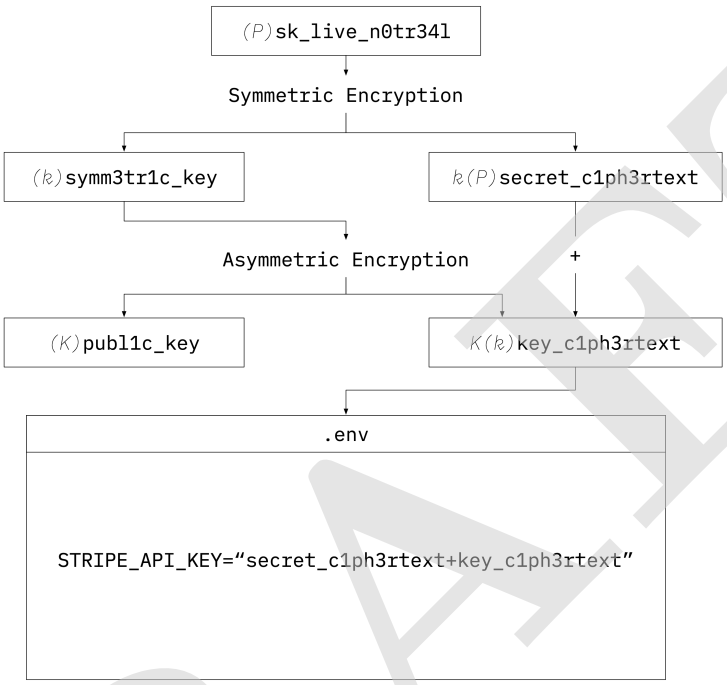
Secrets managers centralize storage, making them high-value targets. If breached, they reveal all secrets at once. Conversely, .env files are often scattered across local machines and repositories, increasing the likelihood of accidental exposure, but with a smaller blast radius.

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.

Committed to code *Separate, stays on machine*

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:

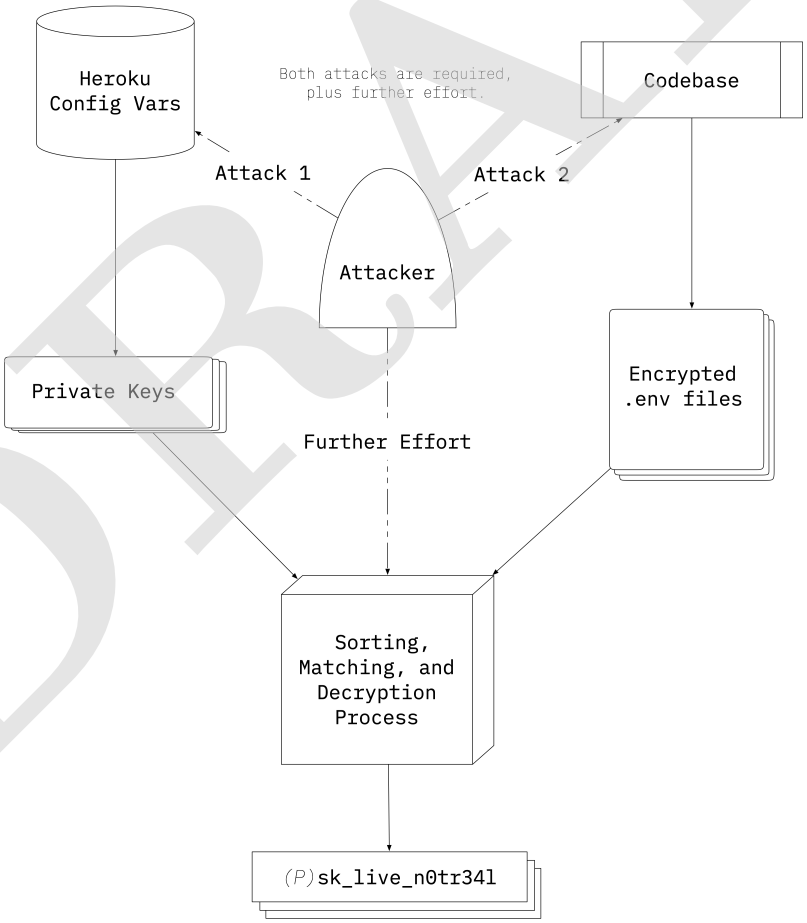
This approach ensures maximum compatibility, working in any environment where secrets are traditionally injected into env, while guaranteeing that secrets exist in plaintext only during runtime (within the process's isolated environment), never persisting in third-party storage or intermediary locations.

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 still raises the bar for an attacker. The attacker must compromise two independent systems—both the platform-native secrets manager and the codebase—often maintained by separate teams or governed by different security policies. Furthermore, even if the attacker gains the encrypted file, they must identify and locate the corresponding private key in a separate system, which might not be trivially linked.



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="026d4945b6513baec60f68b207f203ba534fb54d2b0c99
52557d240815e42a7d83"

# .env
# Database configuration
DB_HOST="encrypted:BMO83g2fEtr66gcFvUs2+/
ZuccCQuBbZwSW3JfCLvoUiACmusxCbTfG2dvc2LxenPhUtgWapO8f9BCcBVAcTnMc
rd3kndvk+acWytRjIWRUvsSezdD340/OT5EQgbqJtwXfuRz0i2t8PVA=="
DB_USER="encrypted:BBrXv55qxgA19sEqqNnZzS/
C0WguVk6ROQmfxnGhBhafLoc0XwpKprk/
J3hJCVq7s45WyBSXGUz9U9rHxCBeVkl27WFzzgZkDewX0gBLt+Cc37K0EVU2hZ1GP
bax5mzpI5Jwwi65be6+"
```

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.

1) Generate a long-lived keypair

When running *dotenvx encrypt* on a .env file, a static 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
```

This long-lived keypair is used for all encryptions of this .env file, but the encryption itself is not directly done with it.

2) Encrypting a Secret (Per-Secret Ephemeral Keypair)

When encrypting each secret, dotenvx does not use the long-lived keypair directly. Instead, it generates a new ephemeral keypair for that encryption session:

```
ephemeral_private_key = generate_private_key() # New for each secret
ephemeral_public_key = derive_public_key(ephemeral_private_key)
```

Now, we derive a shared AES key using Diffie-Hellman key exchange with the recipient's (your long-term) public key [14]:

```
shared_secret = diffie_hellman(ephemeral_private_key, long_term_public_key)
aes_key = derive_aes_key(shared_secret) # Hash shared secret to get AES key
```

This AES key is unique per secret and is never reused. Next, the plaintext secret is encrypted:

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

Finally, the ephemeral public key is prepended to the encrypted value:

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

This means every secret gets a unique public key, which is included in its encrypted value.

3) Decrypting a Secret

To decrypt, you extract the ephemeral public key from the ciphertext:

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

Now, we derive the shared AES key again, but this time using your long-lived private key:

```
shared_secret = diffie_hellman(long_term_private_key, ephemeral_public_key)
aes_key = derive_aes_key(shared_secret)
```

Since Diffie-Hellman is symmetric, this derived key matches the one used for encryption. Finally, decrypt:

```
plaintext_secret = aes_decrypt(aes_key, ciphertext)
```

Summary

- Every secret is encrypted with a unique ephemeral keypair.
- The recipient's long-term public key is used for key exchange.
- The AES key is derived per secret, so forward secrecy is built in.
- The ephemeral public key is included in the ciphertext, allowing decryption later.
- If one secret is compromised, other secrets remain secure.

11. Calculations

WIP (demonstrate risk-off calculations from our solution)

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?utm_source=chatgpt.com, 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.