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.

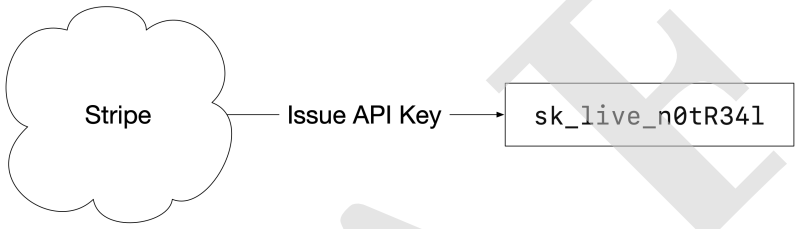
What is needed is a secrets system based on asymmetric cryptography instead of trust, allowing a developer to encrypt secrets without relying on any third party to remain secure. In this paper, we propose a solution to these risks using a library that decrypts an encrypted secrets file at runtime with 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. 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. Today, this remains the standard for all secrets managers, aside from certain file-based methods we address later in this paper.

Secrets Manager

id	appid	name
1	1	OPENAI_KEY
2	1	PAYPAL_KEY
3	3	SENDGRID_KEY
4	3	GITHUB_KEY
5	4	OPENAI_KEY
...		
999997	1	STRIPE_KEY

Large Exposure Risk of 999,997 Secrets

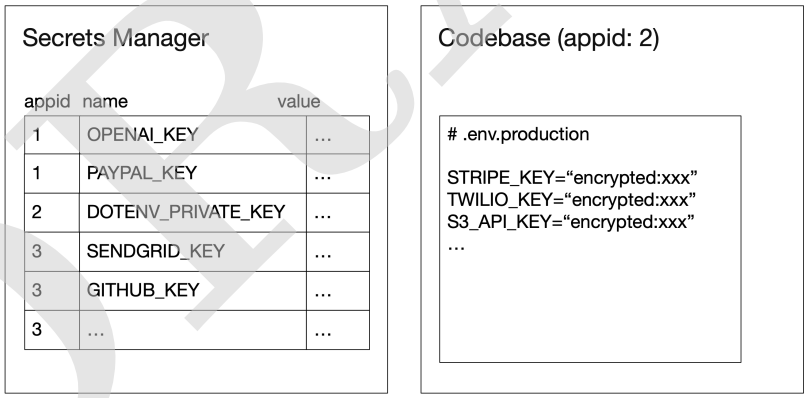
These solutions work well enough but rely on trust. You trust that your secrets manager is not breached, and that your .env file is not exposed. Unfortunately,

history has shown this trust to be fragile, as misconfigurations and human error repeatedly lead to leaks. [2][3][4] .env files are easy to expose unintentionally, but with a smaller blast radius, while secrets managers are breached less frequently but with a much larger scale of impact. The ideal solution would combine the best of both—minimizing breaches while limiting their impact.

4. Encryption

The solution we propose begins with asymmetric encryption. Instead of storing secrets in plaintext within a secrets manager or a .env file, we encrypt them before storage. Each secret is encrypted using a public key, producing an unreadable ciphertext that can only be decrypted with the corresponding private key. The encrypted secrets are then stored in the .env file, ensuring that even if the file is leaked, the secrets remain inaccessible.

The private key is stored separately, typically in a platform-native secrets manager. This effectively splits the secret into two distinct components—the encrypted .env file and the private decryption key—ensuring that no single breach grants access to plaintext secrets.

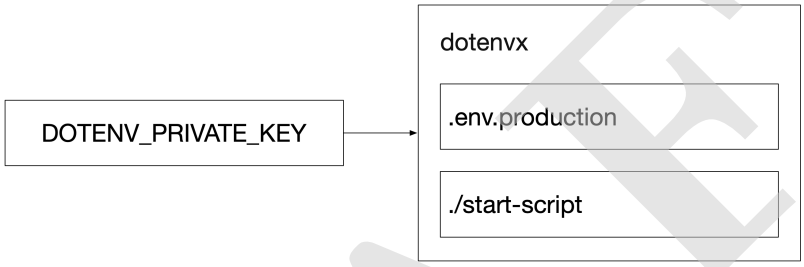


5. Runtime

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 secrets manager, the library retrieves the encrypted .env file, pulls the private key from the 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:

1) Retrieve private key from secrets manager
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



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 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 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. By encrypting .env files, secrets can now be committed safely alongside code, ensuring they follow the same versioning and history as application changes. This maintains the Twelve-Factor App principle of separating config from

code—while secrets exist in the repository, they remain encrypted, requiring a private key stored separately in a secrets manager to decrypt them. As a result, teams gain the benefits of versioning without compromising security.

8. Pull Requests

Pull requests are a cornerstone of modern software development, providing a structured way for teams to review, discuss, and approve changes before merging them into production. They serve as a crucial checkpoint for maintaining code quality, enforcing best practices, and catching potential issues early in the development lifecycle. Secrets management has been disconnected from this workflow, leading to secrets sprawl—where secrets are scattered across local machines, chat messages, shared documents, and different secrets managers without clear visibility or control.

By encrypting .env files and storing them in version control, teams can now bring secrets management into their existing pull request workflows. When a new secret is added, modified, or removed, these changes become visible as part of the pull request diff, ensuring that secrets are tracked and reviewed in the same way as code. This introduces several benefits:

- *Reduces Secrets Sprawl:* Instead of secrets being manually distributed across multiple locations, they are centralized in an encrypted .env file, versioned alongside code, and protected within the same workflow.
- *Built-in Approval Process:* Secrets changes now require the same level of review and approval as code changes, ensuring that production credentials are validated before deployment.
- *Collaboration Between Dev and Ops:* Development teams can propose secrets as part of their pull requests, while DevOps or security teams can verify and approve them, ensuring correct values are set in production before merging.
- *Prevention of Shadow Secrets:* Since all secret changes go through a review process, undocumented or improperly stored credentials are less likely to emerge, improving overall security posture.

This approach enforces an approval process for secrets changes while reducing the risks associated with secrets sprawl. Instead of secrets being an afterthought or handled informally, they become an integrated part of the development workflow, ensuring they are controlled, reviewed, and properly managed.

9. 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.

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 store 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.

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.

10. Calculations

TODO. Need some mathematical calculations to bolster the approach and research. Initial raw ideas:

- Do some calculation on how hard it would be for an attacker to coordinate associating each private key with an encrypted .env file and iterate through each, slurping up keys.
- Do some calculation or estimate/prediction/statistical stat on how these are separate systems and so decreases likelihood that an attacker gains access to both.
- Do calculations on a team using this vs a team not - which is attacker more likely to go after. When will attacker more likely get to your key - N time - as they try the plaintext keys first.
- Do a calculation on iterating over each process - and slurping up env.

References

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

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

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

[4] Margaret Kelley, Sean Johnstone, William Gamazo, Nathaniel Quist, "Leaked Environment Variables Allow Large-Scale Extortion Operation in Cloud Environments", https://unit42.paloaltonetworks.com/large-scale-cloud-extortion-operation/, 2024.