Trend Research

TREND MICRO

Unleashing Chaos

Real World Threats Hidden in the DevOps Minefield

Alfredo de Oliveira and David Fiser

Contents

Published by
Trend Research

For **Raimund Genes (1963-2017)**

Rise of Environment Variables

Environment variables were introduced in Unix version 7 in 1979 as key-value pairs to share variables to processes. These variables were adopted into successor operating systems that are UNIX- and Windows-based. Common use cases include the PATH variable, which is used for sharing the working directory, allowing us to execute our favorite tools without specifying the absolute path. Similarly, language-specific and terminal-related variables, which are used to define the running environment, make perfect sense to share among multiple processes. According to "The Open Group Base Specifications Issue 7, 2018 edition," an environment variable is defined as follows:[1]

The Open Group Base Specifications defines the value of an environment variable as a string of characters. For a C-language program, an array of strings called the environment shall be made available when a process begins. The array is pointed to by the external variable environ, which is defined as:

```
extern char **environ;
```

These strings have the form name=value; names shall not contain the character '='. For values to be portable across systems conforming to POSIX.1-2017, the value shall be composed of characters from the portable character set (except NUL and as indicated below). There is no meaning associated with the order of strings in the environment. If more than one string in an environment of a process has the same name, the consequences are undefined.

Other name= value pairs may be placed in the environment by, for example, calling any of the *setenv()*,[2] *unsetenv()*,[3] [XSI] or *putenv()*[4] functions, assigning a new value to the environ variable, or by using envp arguments when creating a process; see exec in the System Interfaces volume of POSIX.1-2017.

We can imagine environment variables as a static table. Every process has its own with variables that are accessible using the */proc* pseudo file system. For example, the command cat */proc/[PID]/environ* prints all of the process's environment variables.

There are two common ways to start a new process on Linux: fork or the exec variant *syscall*, both of which copy the environment variables table by default. To execute a process with custom-provided environment variables, we have to use the exec version with the "-e" suffix and provide appropriate parameters – *execve()*, *execle()*, or *execvpe()*.[5]

When we execute application, the kernel already places the environment variables on the stack of the running process. Because of this behavior, it is irresponsible to store sensitive information in environment variables.

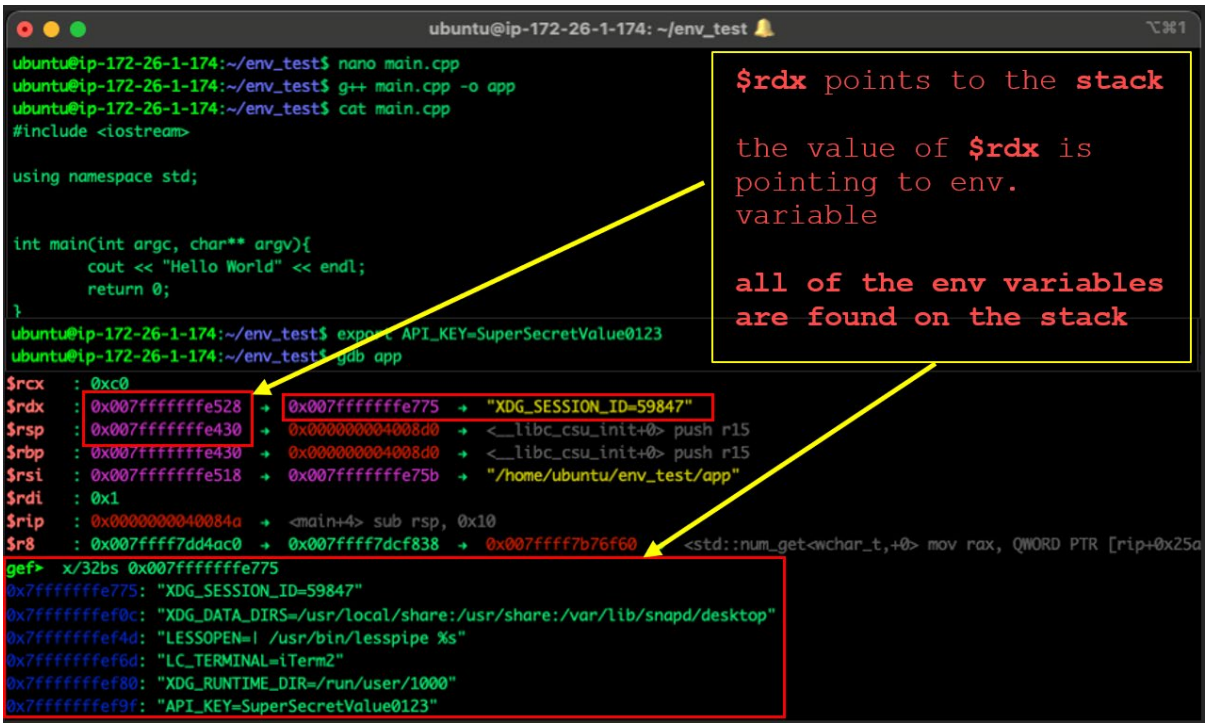


Figure 1. An example of a "Hello World" application with environment variables on the stack

The Unix system has multiple ways to define environment variables depending on the context. For instance, an export command[6] allows us to define an environment variable bound to the actual shell interpreter and its child processes. We can use the export command inside the .profile file in a user's home directory, having the variables accessible after the login.

The massive adoption of containers triggered an explosion of environment variable usage, providing a convenient way to pass a variable. The process namespace isolation prevented the variables from being shared within another process except for the children of the container process. The mentioned behavior is no issue for passing non-sensitive information.

The problem starts when containers power a managed platform, that inherently need to process secrets passed by environment variables. This application design copies every sensitive information stored within the variables into their child processes, such as worker nodes, which do not need them, providing additional opportunity for leakage.[7]

Environment variables have been successfully adopted inside the Windows operating system. Compared to UNIX-based systems, Windows-based systems' configurations are more user-friendly, and the initial idea of exchanging information between multiple processes remains, although implementation details differ.

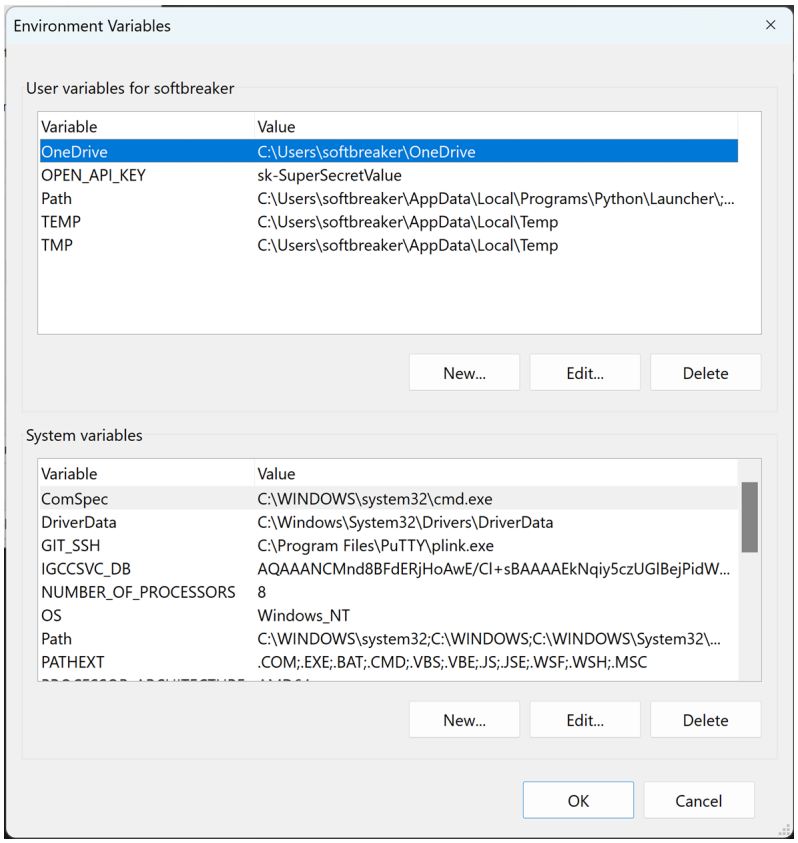


Figure 2. Specifying environment variables inside a Windows system

The Windows operating system distinguishes between user and system environment variables stored inside registry keys.

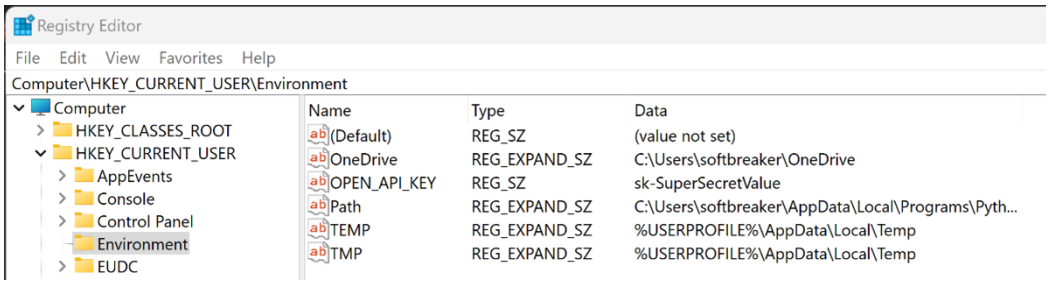


Figure 3. User environment variables stored within registry keys

Although environment variables are not present in every process by default, many processes load them in their virtual memory without necessarily using them. This behavior should raise a red flag in users when sensitive information is stored, as it might be leaked as part of a crash dump. To be more specific, let's say we store a secret of our favorite LLM provider inside a environment variable to avoid hardcoding and reference it in our script. Another application that we use is also working with environment variables. The application has the API key loaded in the process memory and unwillingly exfiltrates it as part of its telemetry.

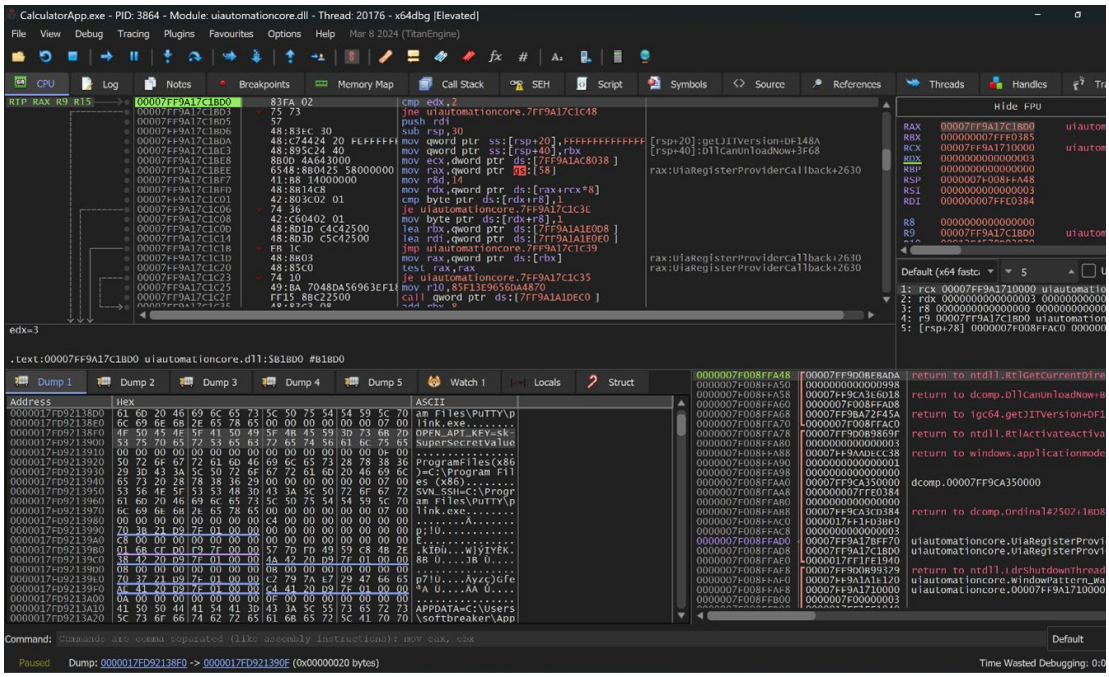


Figure 4. Example of an *OPENAI_API_KEY* environment variable present in the memory space of a Windows calculator

The greatest danger is that we adopted environment variables so rapidly that we didn't even consider examining their underlying technologies and somehow just intuitively deemed them safe.

How else would API gateway developers put them on the same level as vaults? The initial thought of removing hard-coded, file-stored secrets has now vanished.

Best Practices for Secure Secrets Management:

1. Environment Variables:

Use environment variables to store secrets separately from the codebase. This separation reduces the risk of accidental exposure and facilitates easy management.

Figure 5. A secure secrets management best practice is to store secrets separately from the codebase.

Source: Muhammad Ilyas on Medium[8]

The separation of secrets from the codebase is a fantastic approach. The devil is in the details and technology used. Environment variables are not a secure way to manage secrets – they are a convenient way to do so. We might limit the sharing scope to the root container process, but the nature of the technology makes it prone to misconfigurations. They also remain in memory during the entirety of the execution time, and most importantly, they must be passed to the container in the first place.

At its core, Linux inherits a powerful philosophy of modular configuration from its Unix ancestors. This design principle manifests itself in the concept of "Everything is a file."[9] Whether it's system-wide settings, individual user preferences, or the runtime context of a process, these elements are frequently controlled through the manipulation of plain-text configuration files. Environment variables are a core mechanism that enables this flexible configuration system.

Two concrete examples of very familiar files are */etc/profile* and *~/.bashrc*.

The */etc/profile* (system-wide) file plays a critical role in establishing the base environment for all users in a Linux system. The file is processed at system startup and whenever a user logs in.

Within the */etc/profile*, we find the initial definitions for a collection of system-wide environment variables:

- **PATH.** This variable dictates the directories the operating system will search to locate executable programs. By modifying PATH, administrators have fine-grained control over how commands are resolved.
- **LANG, LC_*.** This is a set of environment variables that collectively determine a system's locale. These settings govern elements such as language, date and time display formatting, and numerical conventions, ensuring that applications have culturally aware behaviors.
- **MAIL.** Historically, the MAIL variable would specify the path to a user's mail spool file, enabling command-line mail clients.

Upon launching an interactive login shell, a user's preferences are brought into play through the execution of the *~/.bashrc* (User-Specific Customization) file (located within their home directory). This file provides an avenue for extensive personalization:

- **Shell aliases.** Users commonly introduce aliases to create shortcuts to frequently used or complex commands. For instance, alias *ll='ls -la'* transforms a simple *'ll'* into a detailed directory listing.
- **Prompt customization.** The PS1 variable allows the modification of the command prompt. Users often set terminal colors, display the current working directory, or alter other information to enhance their terminal experience.
- **PATH additions.** Users might extend their PATH variable within their *~/.bashrc* to include directories containing their own installed tools and scripts.

It is worth noting that different Linux distributions might employ variations such as *~/.bash_profile* or *~/.profile*. The specific execution order of these files can have subtle nuances.

```
$ tail -n 30 ~/.bashrc

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Add an "alert" alias for long running commands.  Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*/
/;s/[;&|]\s*alert$//'\'')"'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
export PATH=/usr/bin:$PATH
$
$
```

Figure 6. Setting environment variable inside .bashrc

The .env Invention

The previously shown examples feature how aggregating variables in text files is inherently characteristic of Unix environments, and the rise of container technologies and the DevOps methodology fundamentally transformed how we develop, deploy, and manage software applications.

Central to this transformation is the increased reliance on environment variables to configure applications. The .env file emerged and gained popularity with Docker's debut in 2013. It is the near-ubiquitous standard, yet it carries subtle security implications often overlooked in the quest for agility, time sensitivity, or, simply, convenience.

As tools, libraries, and clouds advanced to streamline development processes, the interaction between applications and environment variables required structured aggregation of these variables.

The .env file core concept is straightforward:

- **Key/value format.** The .env file follows the standard KEY=VALUE format (one per line).
- **Loading Variables.** Libraries like *python-dotenv* (and its equivalents in other languages) offer the ability to parse and load .env files into the application's environment at runtime. Also, tools such as docker compose automatically loads environment variables defined in a .env file when located in the same directory as the docker-compose.yml file. This file is used for variable substitution in the docker-compose.yml file and within the container environment.
- **Human readable.** The plain-text nature of the .env file enhances readability and maintainability.

The adoption of the .env file for making development easier has, not surprisingly, opened up new kinds of security problems, showing us how even simple things can become complicated and create issues. People like using the .env file because it helps manage environment variables in one place. However, this has led to accidents wherein small mistakes in control version systems or web servers can accidentally make these files available online. These files also sometimes get picked up by search engines, leading to breaches that show a considerable gap between what's convenient and what's secure.

There is also something that we call "echoes of the past." It's ironic because the .env file, which is meant to be kept away from the source code and, therefore, be safe, holds both sensitive and not-so-sensitive configuration data together. This mix brings back old security issues, as developers might feel complacent, thinking that keeping everything in a separate .env file is enough to keep it safe.

Also, just having a file named ".env" might cause developers to think that they are on the safe side,despite the fact that this file does not really protect or encrypt the data inside it by itself. Using an .env file is tricky because it makes people believe that they are doing enough for security simply because of how the file is named, when in reality they are not. The .env file being hidden in the operating system context is just an appeal to security through obscurity, not a valid strategy – it never was.

Frameworks that assume that the .env is the main or only place for configuration adds to the problem. These frameworks push everyone to use the .env file without discerning whether it is the best way to handle sensitive information. It's like putting all your eggs in one basket without considering if it's the safest place to put them.

Therefore, even though the .env file has been picked up by many for its promise of simplifying configuration management, using it in real life has shone a light on many security problems that we didn't expect. This tells us that developers and security experts need to look closely at the tools and practices that are being adopted and carefully consider all their components, especially when it comes to keeping information safe.

The Ugly Truth about .env

To illustrate some of the security traps developers can inadvertently fall into and exemplify previous claims, we need to take a look at Laravel, a popular PHP web framework that is used on 18% of the leaked codebases based in our telemetry.

Laravel encourages using the .env file as one of the configuration files to store the following:

- **Database credentials.** Based on our observation, database usernames and passwords are typically found in this file.
- **API keys.** Keys for external services, payment services, and other bits and pieces for connecting to different systems are often placed here for the sake of convenience.
- **Application secrets.** Things like Laravel's own encryption key, secrets for sessions, and other unique bits that the application needs to keep secure are usually kept in the .env file.
- **General settings.** Even things that are usually not a big secret, such as whether the app is in debug or production mode or the mail server settings, can end up in the .env file.

Even though Laravel gives tips on how to keep the .env file safe in its documentation, the file is used for almost everything, which might make developers a bit too comfortable or complacent when using it.

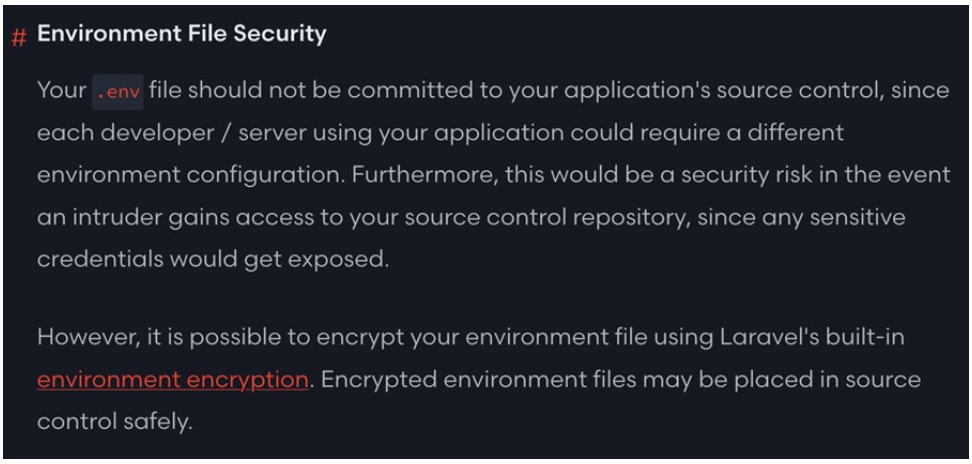


Figure 7. The .env file security warning in the Laravel documentation

Source: Laravel[10]

The .env file has become a substitute for configuration files in cloud-native projects. We adopted it so much that we stopped asking if it was the right place for it. The increasing number of configuration parameters tends to hide secrets among other arguments, forgetting that it's a data type unsuitable for environment variables.

Figure 8. A website powered by Laravel with an exposed .env file

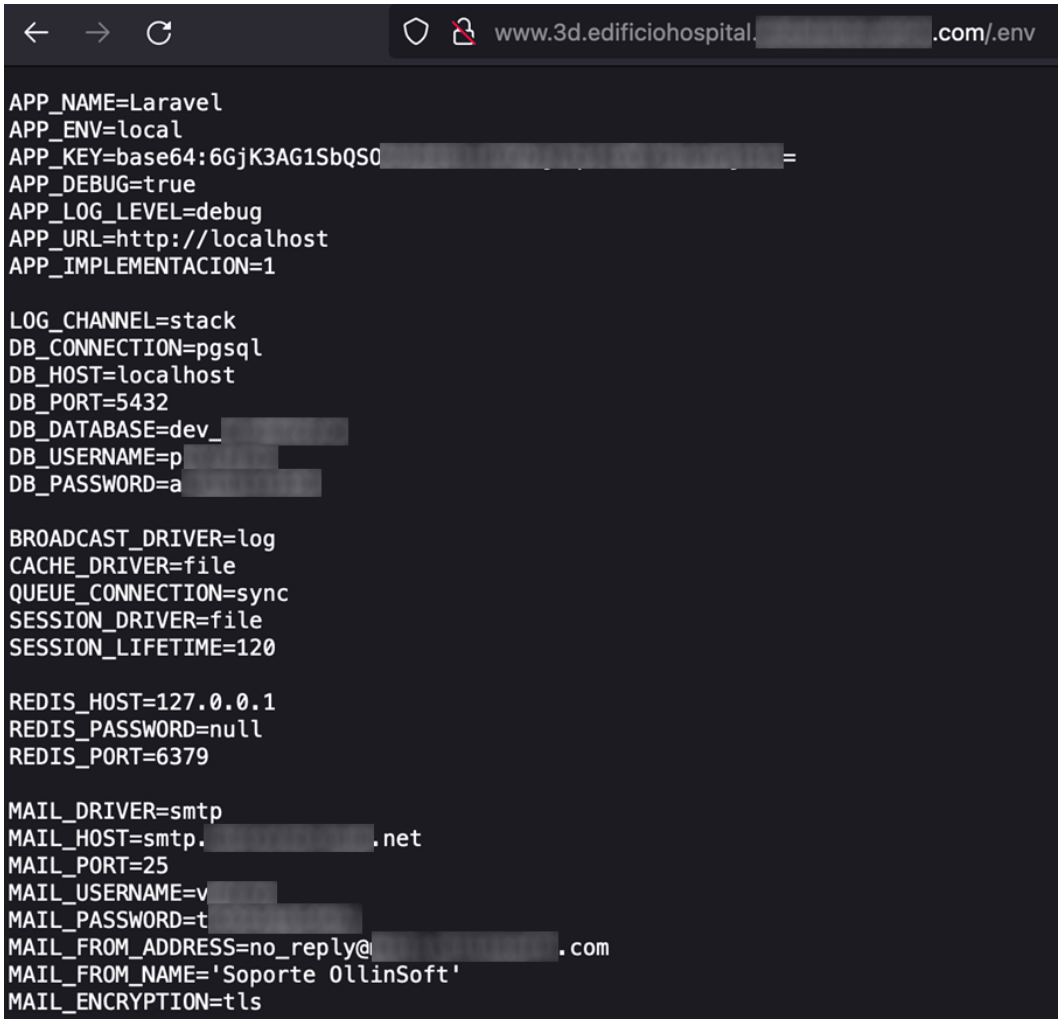


Figure 9. A website powered by Laravel with an exposed .env file

Usage Analysis

To understand the adoption of .env files in DevOps, we examined files sourced from VirusTotal and GitHub. After analyzing 2,000 files, our investigation confirmed that the .env file is mainly used as a universal configuration file, suggesting that its adoption is not always linked to the environment variables concept. This indicates that .env files are present inside deployment environments, including containers. The findings correlate with our previous research on container registries.

```
################################################################
banner2 "     R U N  C O N T A I N E R S"
echo
docker-compose -f /home/"$USER"/docker/docker-compose.yml --env-file /home/"$USER"/docker/.env up -d
```

Figure 10. An example of .env usage linked with spinning containers

This brings us to question the misconception of the .env file's intended usage, especially when sensitive information is stored within these files.

```
# Hinzufügen der Funktion zum Lesen der .env-Datei
def read_env_file():
    env_values = {}
    with open('.env', 'r') as env_file:
        for line in env_file:
            key, value = line.strip().split('=')
            env_values[key] = value
    return env_values

# Hinzufügen der Funktion zum Schreiben der .env-Datei
def write_env_file(env_values):
    with open('.env', 'w') as env_file:
        for key, value in env_values.items():
            env_file.write(f'{key}={value}\n')
    with open('backend/.env', 'w') as env_file:
        for key, value in env_values.items():
            env_file.write(f'{key}={value}\n')
```

```
config = dotenv_values(".env")

class Sniper:
    def __init__(self):
        self.vanity_url = config.get("VANITY_URL")
        self.guild_id = config.get("GUILD_ID")
        self.token = config.get("TOKEN")
```

Figures 11 and 12. The .env file used as a configuration file and the .env file used with the popular dotenv library

The intention of injecting dynamic configuration and secrets into a container environment, together with the massive adoption and misunderstanding of how containers should be used, led to the creation of the .env file. Unfortunately, developers tend to put secrets in .env files without giving it a second thought, creating opportunities for abuse by threat actors.

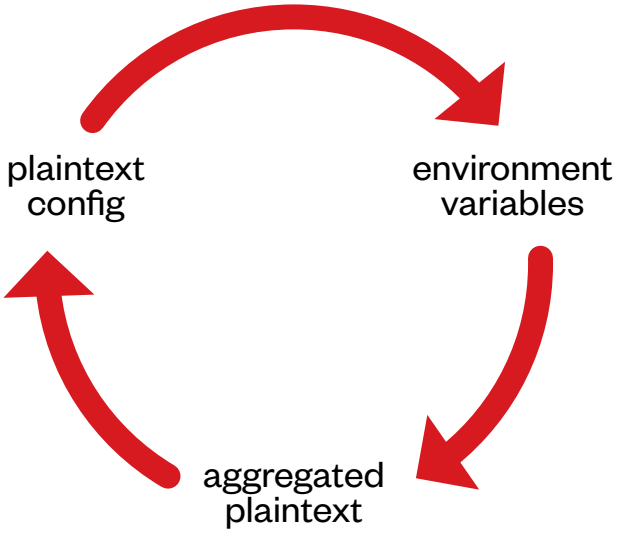


Figure 13. The life cycle of environment variables

Figure 14 features a code snippet that demonstrates how developers either ignore or misunderstand security concepts when it comes to storing secrets in .env files.

```
for project in Guidance Management OrderEntry ReportEngine
do
  (aws secretsmanager get-secret-value --region $1 --secret-id $2/pgx/${project,,}_env --query SecretString --output text) > env.json;
  (jq -r 'to_entries[] | (.key | tostring | ascii_upcase) + "=" + (.value | tostring )' < env.json) > .env;
  rm env.json
  mv .env /var/www/html/${project}/
done

exec apachectl -D FOREGROUND
```

Figure 14. A code snippet that shows how secret-storing security concepts are ignored

The AWS Secrets Manager is a secure vault for storing secrets. However, in this example, the developers creatively use it to obtain securely stored credentials and put them into unsecured plain-text files, which they transform into the .env file format and move to the production directory.

The directory is accessible through the Apache web server and poses a security vulnerability as the .env file exposes all secrets to the public without any access limitation.

Tracing .env on GitHub

As part of our investigation, we queried GitHub and confirmed the .env file's popularity, finding more than 20 million references to the keyword.

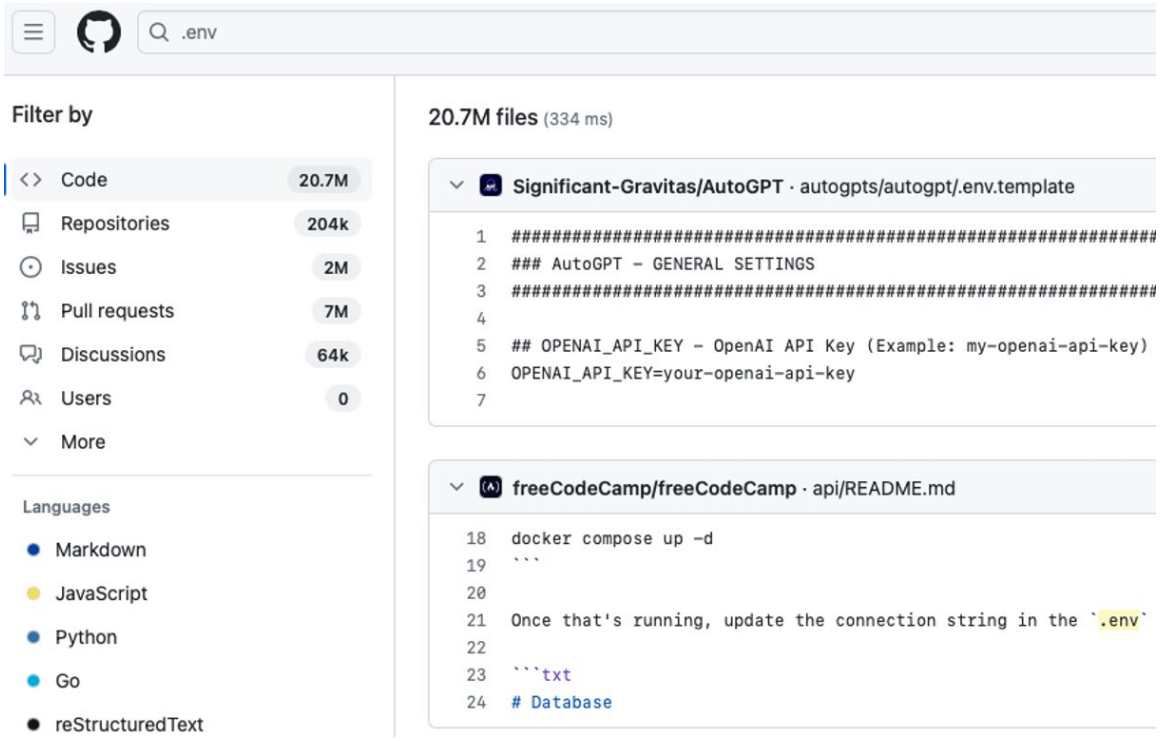


Figure 15. The prevalence of the .env keyword on GitHub

Although GitHub implements secret scanning[11] that runs automatically on public repositories and public npm packages, no detection logic is perfect. Users should focus on not putting secrets in public repositories and npm packages in the first place, as the consequences could be disastrous. Despite GitHub's secret scanning feature, we found API keys and other secrets hard-coded inside public repositories. It is important to note that although secrets are placed in public repositories even just for a brief period, it is enough time for threat actors to find and exploit them as they have automated tools waiting for such unintended exposures to happen.

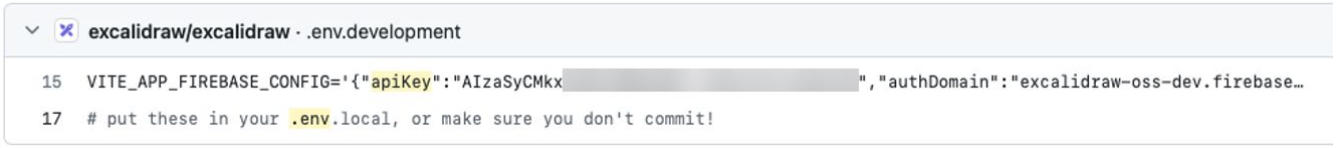


Figure 16. An example of an API key found on GitHub

Source: Excalidraw's .env.development repository on GitHub[12]

The results include not only exposed secrets but also mainly the .env file use cases in various coding languages, giving us valuable insights on tools that the DevOps community has adopted.

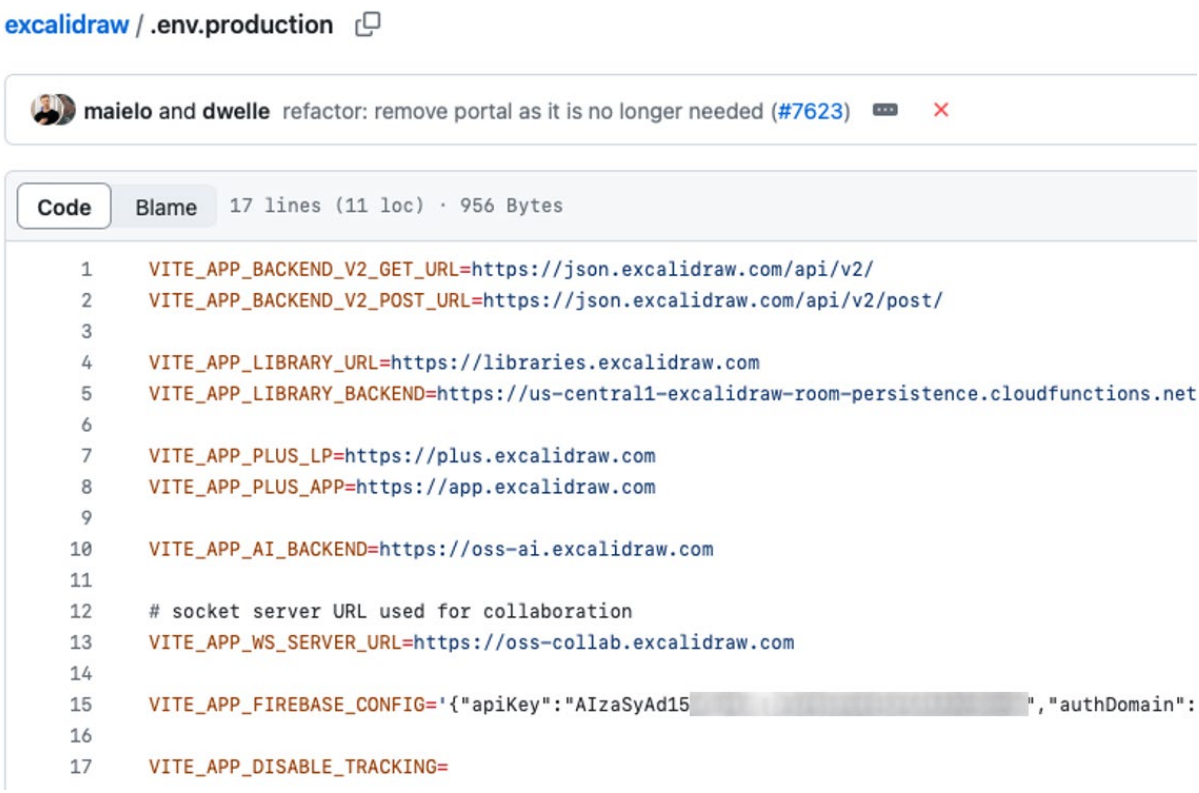


Figure 17. Production .env file present inside a public GitHub repository

Source: Excalidraw's .env.production repository on GitHub[13]

Secrets exposure is only the tip of the iceberg. One of our research goals was to understand developers' .env file practices. Our GitHub query revealed the massive popularity of dotenv-related libraries.

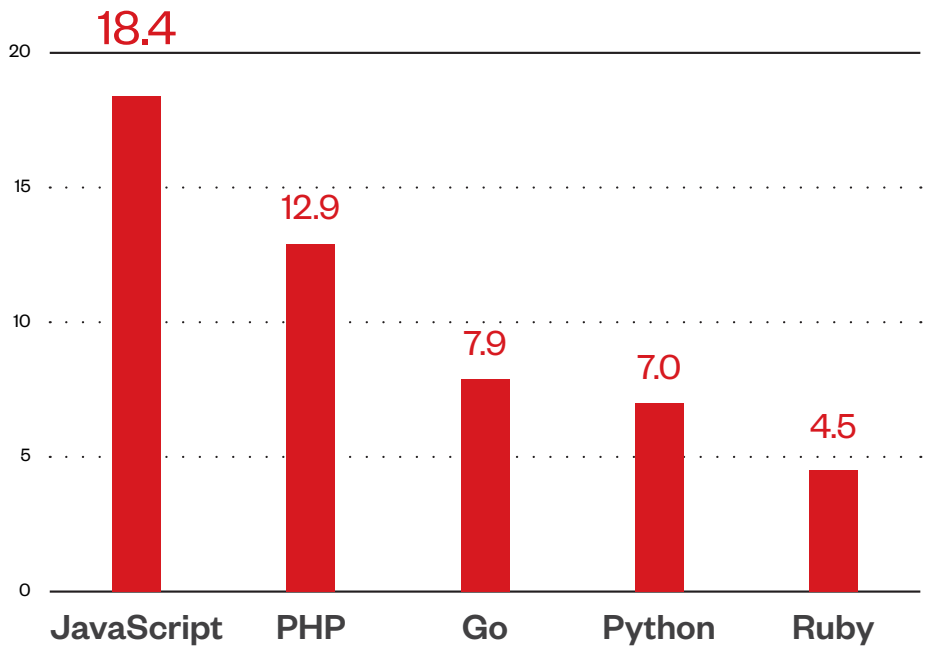


Figure 18. Dotenv-related libraries' popularity data from GitHub obtained from the second quarter of 2024

Looking into the documentation of five of the most popular libraries, we observed that they primarily encourage storing secrets in .env files. One suggested the possibility of encryption using another project, while four of them suggested adding the .env file to the .gitignore file to prevent unwanted secrets from committing.

Multiline values

If you need multiline variables, for example private keys, those are now supported (`>= v15.0.0`) with line breaks:

```
PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
...
Kh9NV...
...
-----END RSA PRIVATE KEY-----"
```

Alternatively, you can double quote strings and use the `\n` character:

```
PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----\nKh9NV...\n-----END RSA PRIVATE KEY-----\n"
```

Comments

Comments may be added to your file on their own line or inline:

```
# This is a comment
SECRET_KEY=YOURSECRETKEYGOESHERE # comment
SECRET_HASH="something-with-a-#-hash"
```

Figure 19. Documentation encouraging the storing of secrets inside the .env file

Source: motdotla's dotenv repository on GitHub[14]

This might create an assumption that adding .env into the .gitignore file is a secure practice that will prevent committing secrets into Git repositories. However, doing so will not prevent the storing of a set of plain-text secrets inside a single file in production or test environments or adding plain-text secrets to a container image.

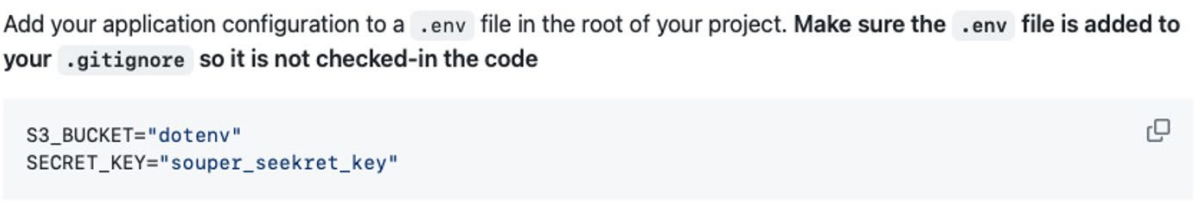


Figure 20. Documentation recommending adding .env files into the .gitignore

Source: phpdotenv's repository on GitHub[15]

We do not recommend using environmental variables or .env files for storing secrets because of the reasons we previously described. However, there are scenarios where it is less dangerous to use .env files for storing secrets. For instance, using containers as short-lived processes and injecting them from the parent environment. Doing so won't allow the production container to access the .env files, which means that they are not present within the container image.

If secrets must be stored within a file, the values should be encrypted and decrypted only for the short lifespan of the container.

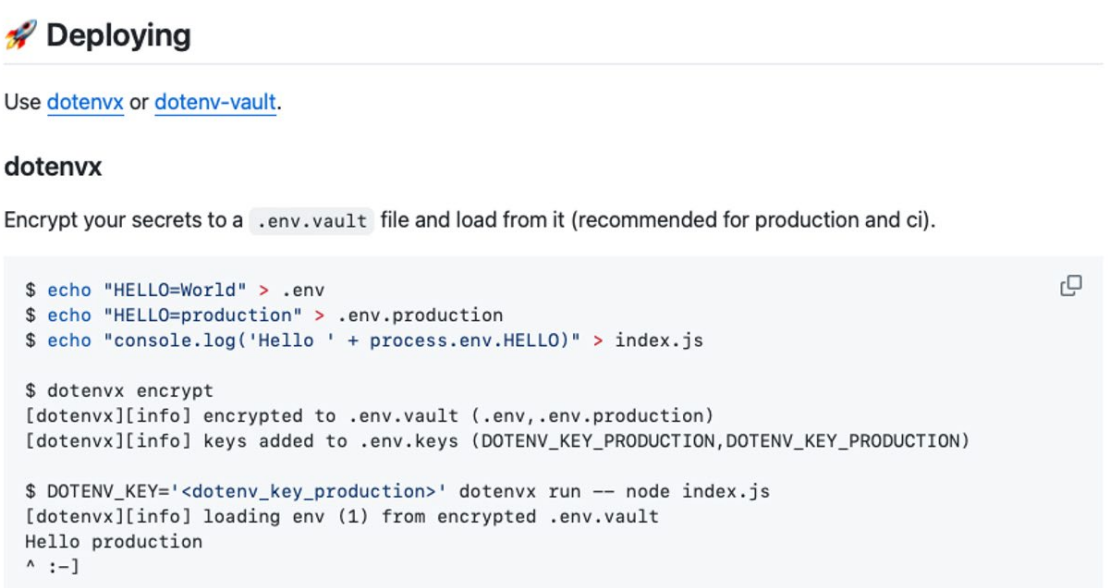


Figure 21. An example of how to encrypt .env files

Source: phpdotenv's repository on GitHub[16]

Our short exploration of GitHub repositories suggests that developers are encouraged to store sensitive information inside a .env file. The lack of security awareness and understanding of the full scope of the problem allows malicious actors to take advantage of and download exposed .env files. This practice also provides easy access to cybercriminals after a compromise occurs as it aggregates project credentials within a single plain-text file.

VirusTotal and .env

VirusTotal was our second source, where we did not expect to find non-malicious .env references. We performed a retrospective hunt, searching for a generic .env keyword and expecting many false positives. After the first run, we removed the most prevalent false positives, and the query returned 2,000 results that we manually categorized. We sorted 1,500 samples as irrelevant to our research due to duplicities or false positives. We categorized 89 of them as malicious, and 25 referenced OpenAI API usage. The remaining 386 served as .env file usage analysis samples.

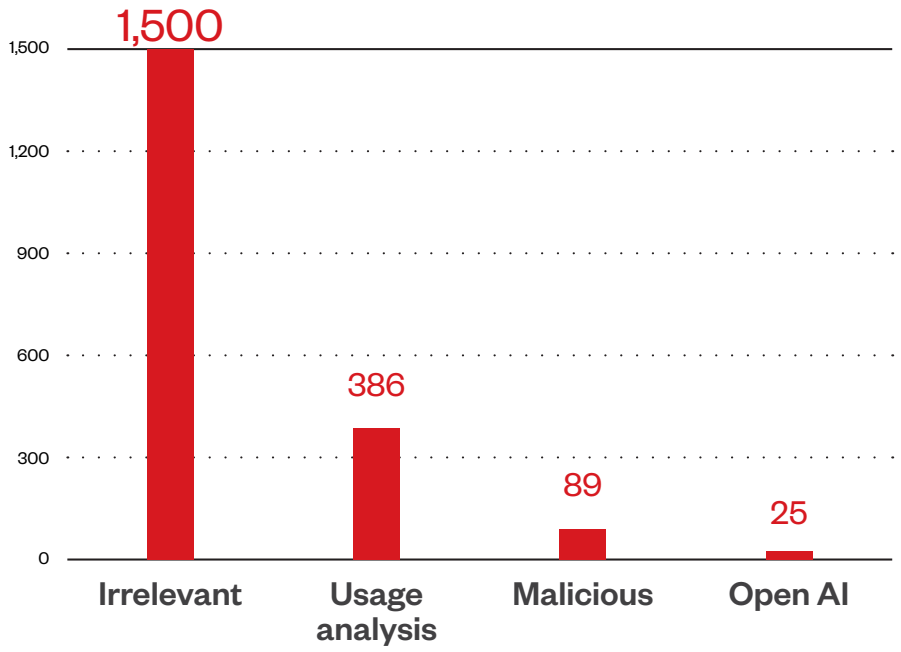


Figure 22. VirusTotal .env sample categorization data obtained from _____ to _____, 2024

The usage analysis revealed that 313 of 411 were unique, non-malicious scripts referencing one of the following keywords:

- "api"
- "token"
- "password"
- "key"

Although some used the term "encrypt" inside their content, none was related to the .env file.

How did the samples appear on VirusTotal in the first place? VirusTotal is a service that the security community uses to evaluate and scan files with various security engines. Security professionals also use the service for sample hunting.

To answer the question, the samples were uploaded to the service, either via an individual user or a software that automatically utilizes the VirusTotal service. In one of our previous searches, we identified hard-coded API tokens uploaded to VirusTotal. It is another reason why we should not save credentials in a file in plain text.

Exploitations of .env Files

We first described secrets inside environment variables in our white paper "Securing Weak Points in Serverless Architectures: Risks and Recommendations," which was published in 2020.[17] The implicit pre-authorization tokens stored within the variables allowed a threat actor to exploit the tokens and use the services upon compromise; since then, we have continuously observed sensitive information being stored inside environment variables.

We made similar observations in other cloud serverless frameworks. Although one cloud service provider opted out of using environment variables to store secrets within their serverless framework, we observed the dangerous uses of environment variables inside Azure Functions. We described them in our white paper, "The State of Serverless Security on Microsoft Azure," released in 2023.[18]

The rise of the TeamTNT threat actor, which targeted cloud service misconfigurations and harvested credentials, was, for us, an expected evolution instead of a surprise. The 2021 white paper titled "Tracking the Activities of TeamTNT: A Closer Look at a Cloud-Focused Malicious Actor Group"[19] thoroughly tracked the evolution of the malicious TeamTNT threat actor group that targeted credentials found inside an infected system.

Meanwhile, the increased attention on TeamTNT caused the group to keep a low profile, which then led to its announced retirement. Despite its inactivity, the problem of secrets being placed inside environment variables did not disappear. One thing is certain: The legacy of TeamTNT is alive and has inspired various threat actors. To this day, we continue to observe the evolution of malicious samples that this group has inspired. However, this time our observation is marked with an extended functionality: the harvesting of the .env file.

```
CRED_FILE_NAMES=("authinfo2" "access_tokens.db" "" ".smbclient.conf" ".smbcredentials" ".samba_credentials" \
".pgpass" "secrets" ".boto" ".netrc" "netrc" ".git-credentials" "api_key" "censys.cfg" \
"ngrok.yml" "filezilla.xml" "recentservers.xml" "queue.sqlite3" "servlist.conf" "accounts.xml"\
"kubeconfig" "adc.json" "azure.json" "clusters.conf" "docker-compose.yaml" ".env")


DBS_CREDFILES=("postgresUser.txt" "postgresPassword.txt")



AWS_CREDS_FILES=("credentials" ".s3cfg" ".passwd-s3fs" ".s3backer_passwd" ".s3b_config" "s3proxy.conf" "awsAccessKey.txt" "awsKey.txt")
```

Figure 23. Malicious credential harvester targeting .env files

The use of the .env file among the DevOps community is widely popular. However, a lack of security awareness, poorly established processes, and the very nature of the .env file itself make it easy for developers to include the file in the content management system (CMS) and, tragically, into the production environment. Threat actors know this and have developed an arsenal for scanning for such security misconfigurations. The .env file is a perfect target for cybercriminals because it can aggregate multiple cloud secrets.

Not only can we encounter the .env files deployed in production environments accessible by HTTP endpoints, but our previous analyses of exposed container registries[20, 21] also show the presence of .env files inside container images and container registries, contradicting the original intention of injecting secrets to environment variables.

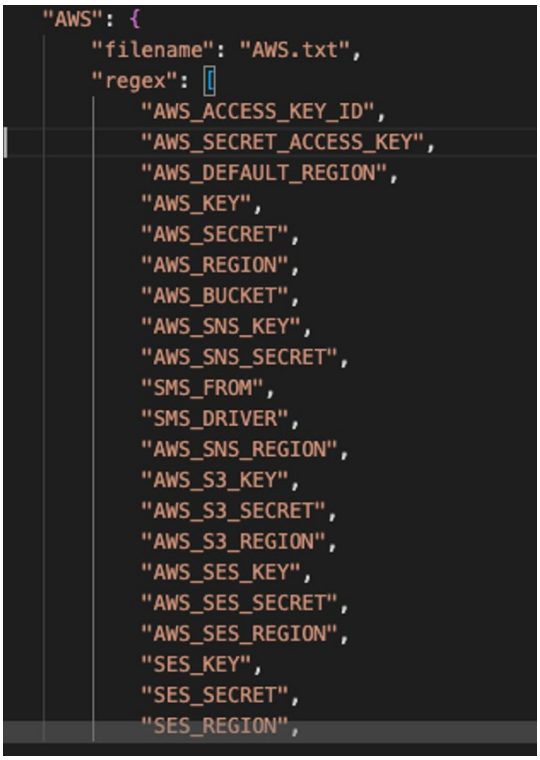


Figure 24. An example of the secrets that threat actors look for inside an *.env* file

We found nearly 90 samples specifically targeting .env file misconfigurations. Laravel is one of the examples that stood out multiple times during our research. The fact that threat actors specifically target Laravel suggests a high prevalence of security misconfigurations.

```
endpoints = [
    "/.env", "/core/.env", "/web/.env", "/app/.env", "/laravel/.env",
    "/crm/.env", "/backend/.env", "/local/.env", "/application/.env",
    "/admin/.env", "/prod/.env", "/api/.env", "/phpinfo",
    "/_profiler/phpinfo", "/phpinfo.php", "/info.php"
]

found = False
for endpoint in endpoints:
    response = requests.get(site + endpoint, verify=False, allow_redirects=False, timeout=8)
    if 'APP_URL' in response.text:
        found = True
        file_name = endpoint.split('/')[-1]
        with open(f'found_{file_name}.txt', 'a') as file:
            file.write(f'{site}{endpoint}\n')
        print(f"{site} | {Fore.GREEN}FOUND{Fore.RESET} - {file_name}")
        break
```

Figure 25. Script hunting Laravel .env files

The malicious payloads fall into one of two categories:

A. The script enumerates the HTTP endpoint from previously specified ranges, seeking .env files inside multiple URL paths.

B. The victim's system was already infected, and the scripts are looking for traces of .env files left by the developers.

Data from our honeypots confirmed the prevalence of this problem by logging .env file requests hours after their deployment without any additional advertisement.

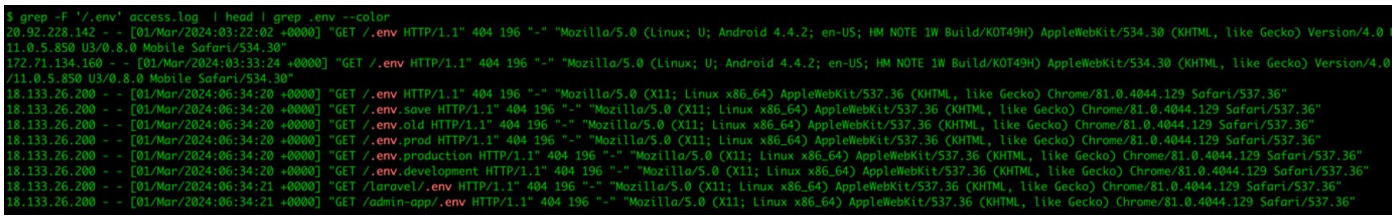


Figure 26. Honeypot records with multiple HTTP requests referencing *.env*

Thinking like a Threat Actor

This section aims to gather empirical data pertaining to the extent to which attackers actively seek out .env files.

Methodology

We set up an HTTP honeypot to simulate a realistic DevOps project environment, intentionally leaking credentials through .env files to monitor and log access attempts by unauthorized users. This setup was chosen to mimic common security oversights in real-world projects. The honeypot was set from 1st of March to 15th of April 2024.

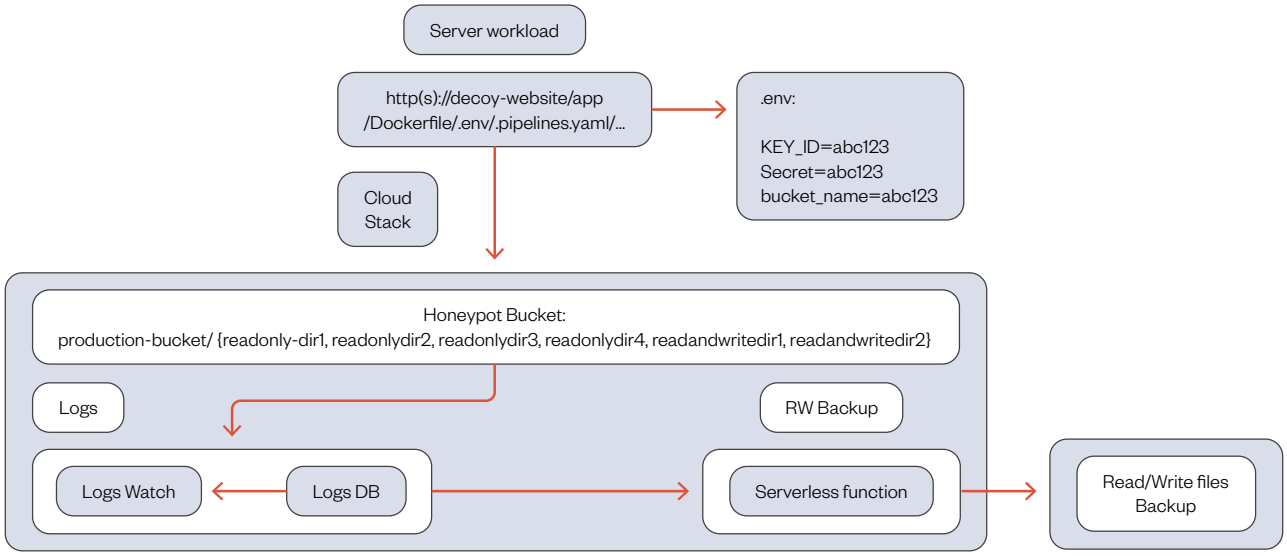


Figure 27. Honeypot workflow diagram

Execution

The honeypot was deployed with a decoy .env file containing real cloud credentials and fake API keys, secrets, and database connection settings similar to what we encountered in actual .env file usage in the wild that contained real credentials, API keys, and database connection settings.

We implemented native logging and analysis mechanisms to track access attempts to these files, aiming to identify patterns in the attackers' behavior, such as frequency of attempts, origin, and methods used to discover the .env files.

Findings

The data collected from the honeypot operation provided clear evidence of targeted attempts to access .env files. Key findings include:

- **Increased access attempts.** Throughout the experiment, there was a noticeable increase in the number of requests for .env files, suggesting a growing interest among attackers in exploiting these vulnerabilities.

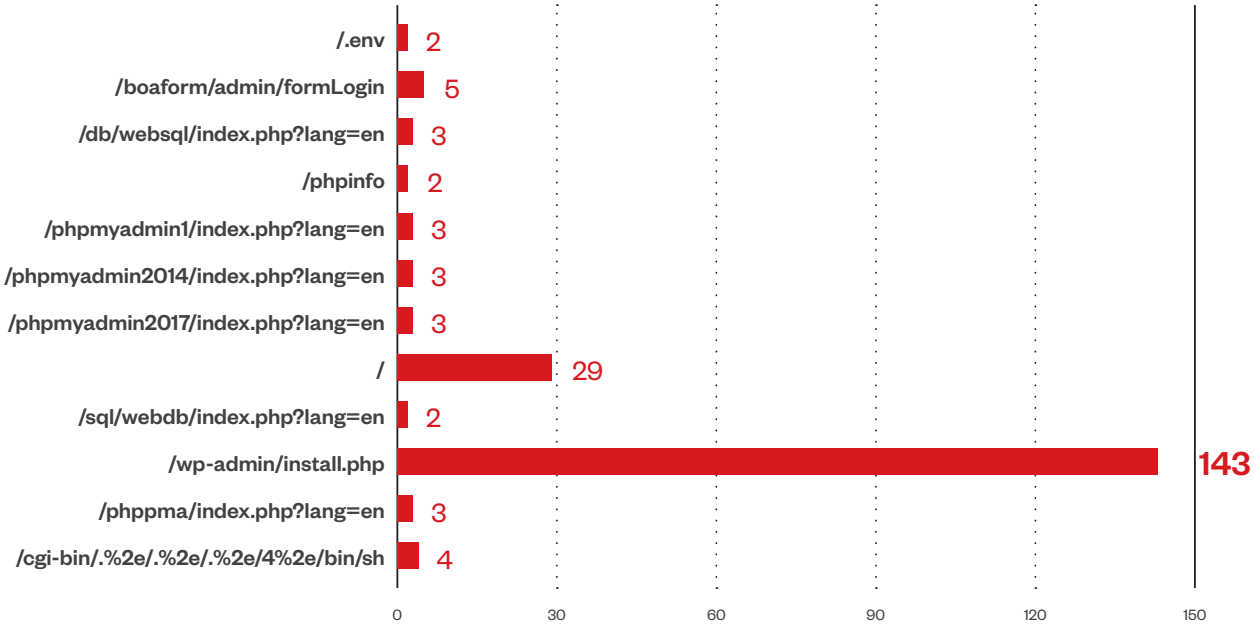


Figure 28. Different purpose Honeypot receiving access on Feb. 5, 2023, showing a small number of .env requests, not noticeable at that time

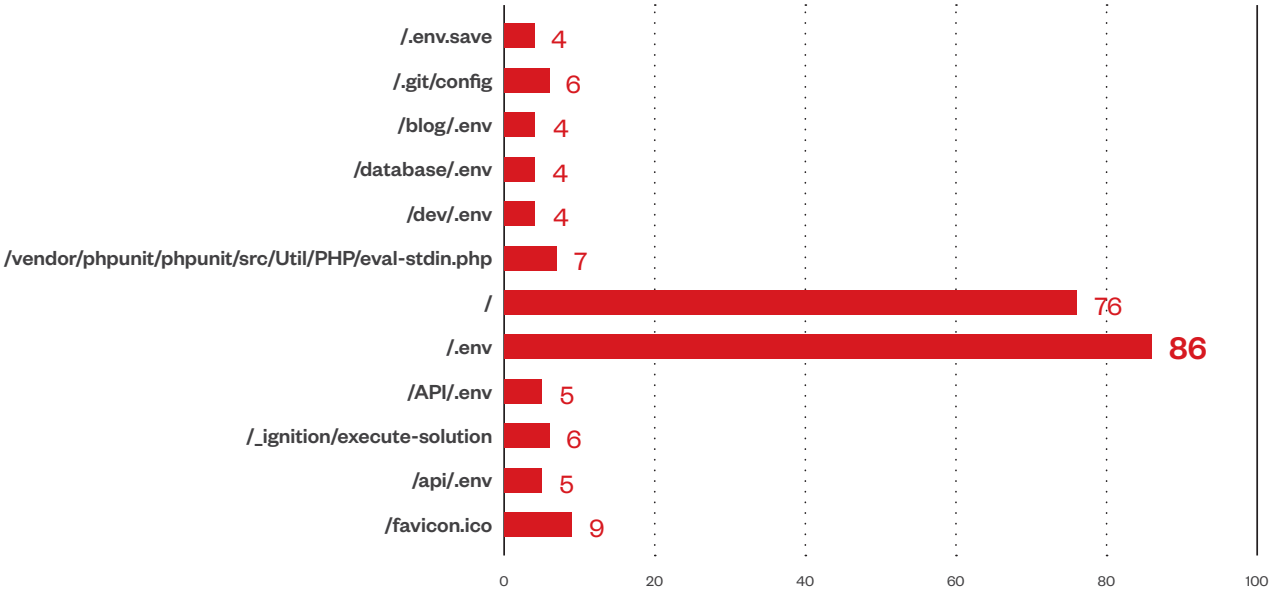


Figure 29. Honeypot access on March 17, 2024, showing the prevalence of .env requests

- **Reconnaissance techniques.** The analysis of logs revealed various techniques used to locate and access .env files, including directory traversal attacks and automated scanning tools.

```
"python-requests/2.31.0"
"Go-http-client/1.1"
"curl/8.1.2"
"python-requests/2.28.2"
"fasthttp"
"masscan/1.0
"Go-http-client/2.0"
"python-requests/2.26.0"
"python-requests/2.25.1"
"https://gdnplus.com:Gather
```

Figure 30. List of user agents from the tools used to fingerprint the honeypot

- **Origin of attacks.** Requests originated from a diverse set of IP addresses, indicating a broad interest across different attacker profiles and geographies.

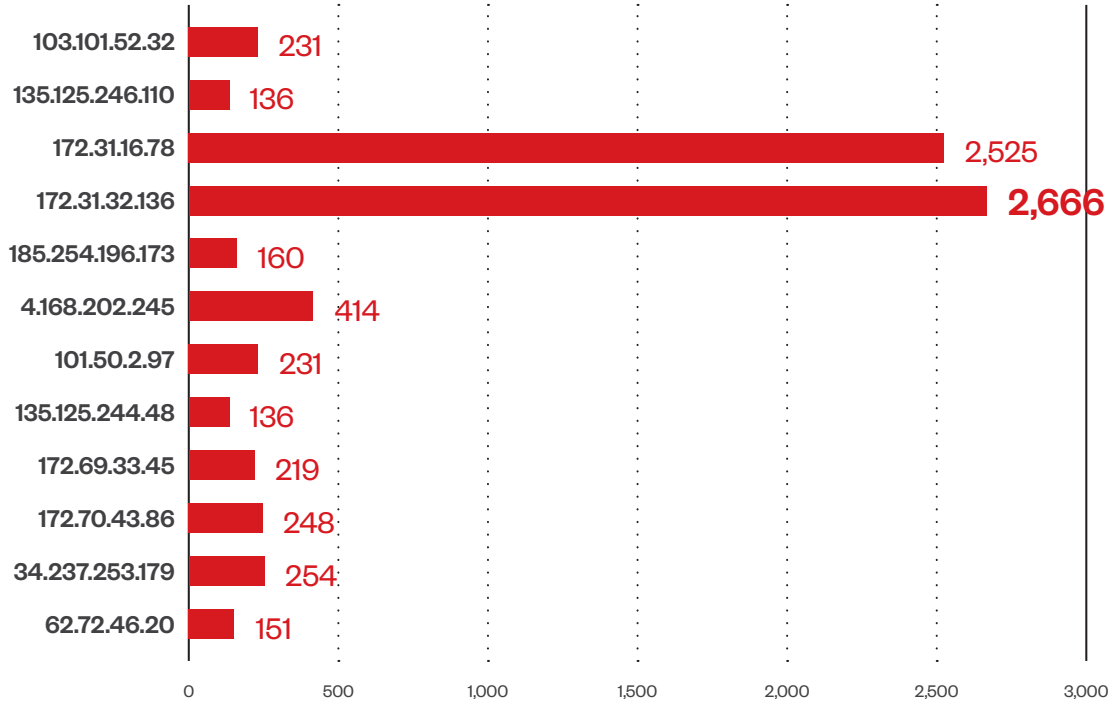


Figure 31. Honeypot access by IP addresses

An analysis of the honeypot baits taken

What happens when the attackers finally find the secrets?

To answer this question, we devised a honeypot that simulates a misconfigured cloud environment. This setup intentionally accepts cloud access tokens and IDs, serving as bait to lure attackers into our monitored environment. The goal was to observe attack patterns, identify common entry points, and understand the types of resources most frequently targeted.

Upon deploying the honeypot, we recorded all interactions, which were then analyzed to extract meaningful data on attacker behavior. During the 45 days experiment, we observed a significant number of unauthorized access attempts, with the total reaching 24,488 connections.

This volume of activity underscores the attractiveness of such credentials to cybercriminals and the constant threat faced by cloud environments.

The examination of this dataset revealed fascinating patterns and preferences in the attackers' methods. Among the most prevalent events attempted by these unauthorized entities were *HeadObject* and *GetObject* requests, both accounting for 18,680 incidents. This suggests a primary interest in accessing and exfiltrating data, underscoring the critical nature of securing object storage services.

Another noteworthy event was *GetAccount*, with 1,720 attempts, showing that cybercriminals want to gather more information about the compromised account, possibly to escalate privileges or explore further exploitable resources.

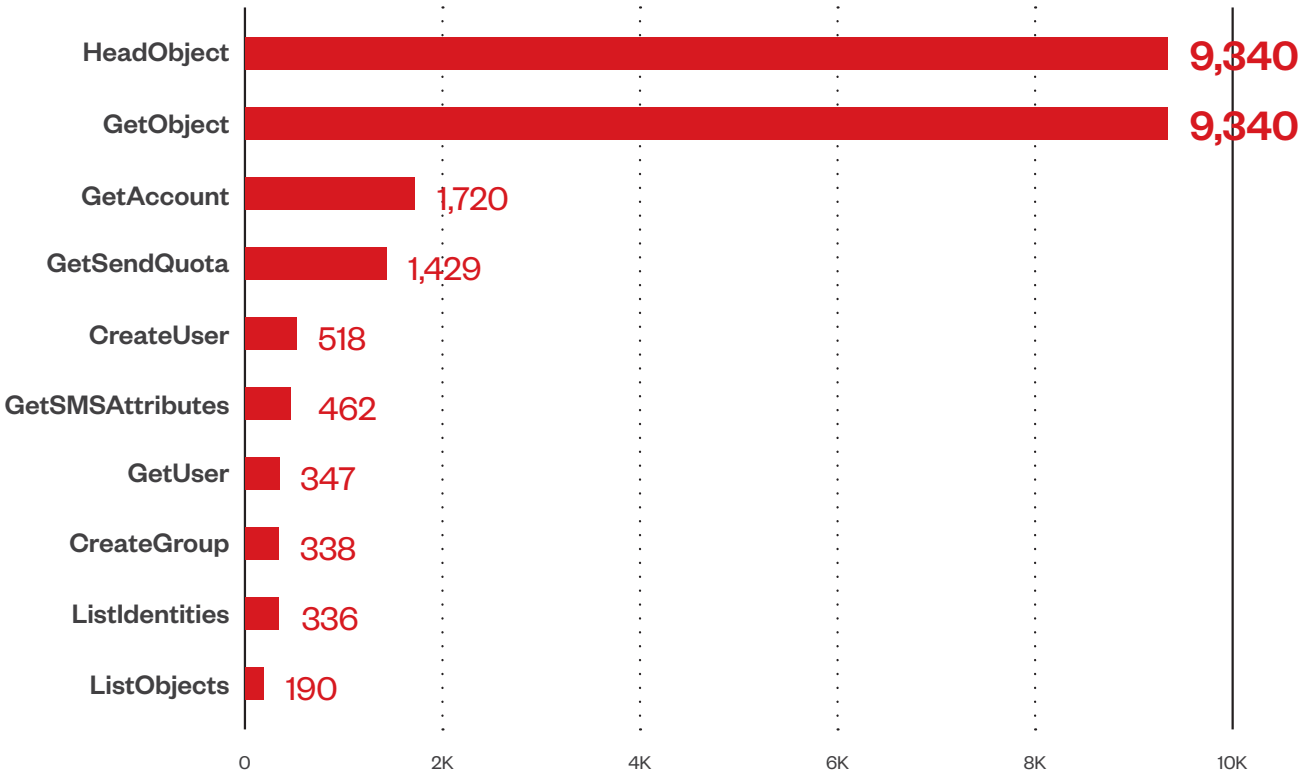


Figure 32. Top 10 most used service commands in our honeypot

An analysis of the source IP addresses from which these requests originated revealed a concentrated pattern of activity. A significant majority of the requests, 18,747, originated from a single IP address (139[.]28[.]177[.]186), suggesting the operation of a dedicated adversary or a botnet controller.

Other IP addresses, while far less active, also contributed to the unauthorized request pool, indicating a dispersed but focused effort to exploit the exposed credentials. Notable IP addresses include 185[.]254[.]196[.]173 and 185[.]254[.]196[.]186.

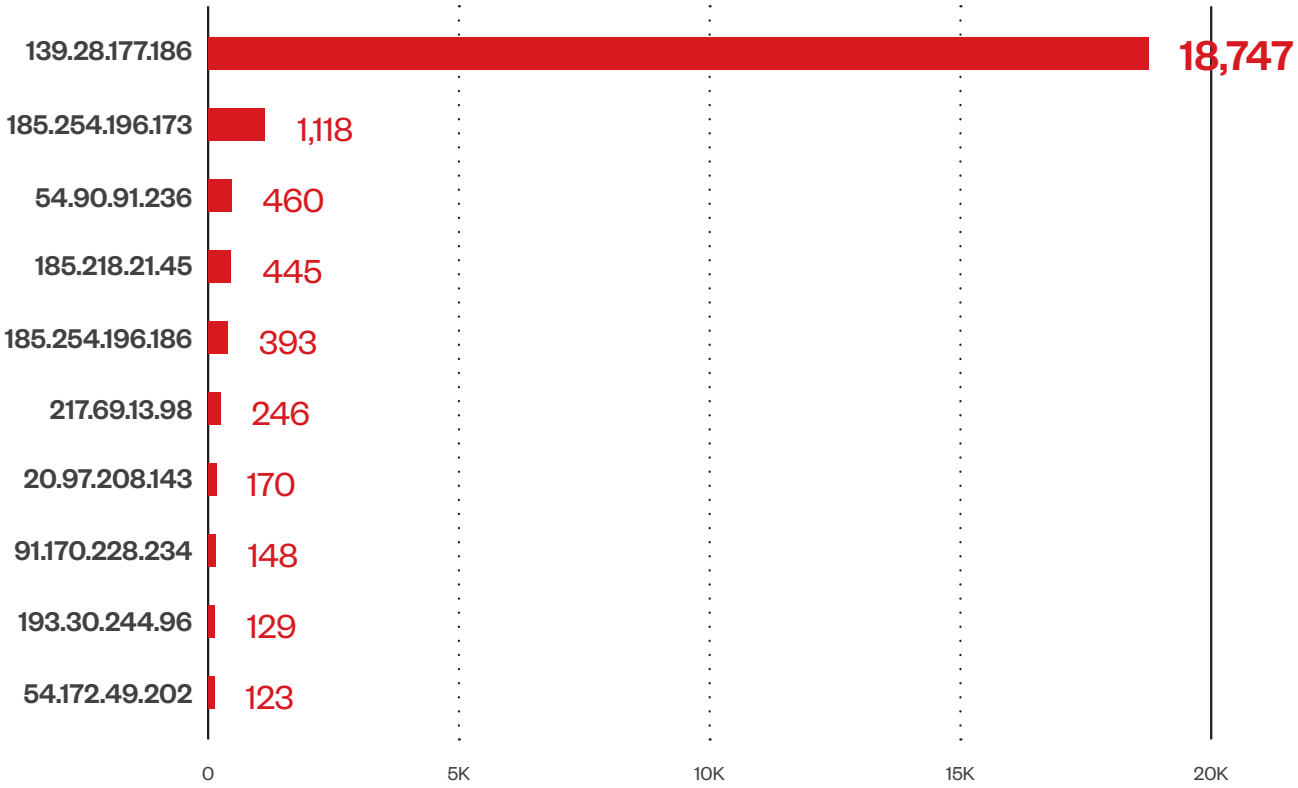


Figure 33. Top 10 IP addresses with the most honeypot access

Looking into the user-agent header in the http request, we got an insight on the tools and technologies favored by the threat actors with a dominant value "S3 Browser/10.7.1" involved in 18,992 requests, pointing to the utilization of popular, legitimate software tools for illicit activities. This is followed by various versions of "Boto3," an AWS library used in different tools. We also identified several other tools via the Boto3 user-agent, thus emphasizing the attacker's preference for scripting and automation in their exploitation efforts.

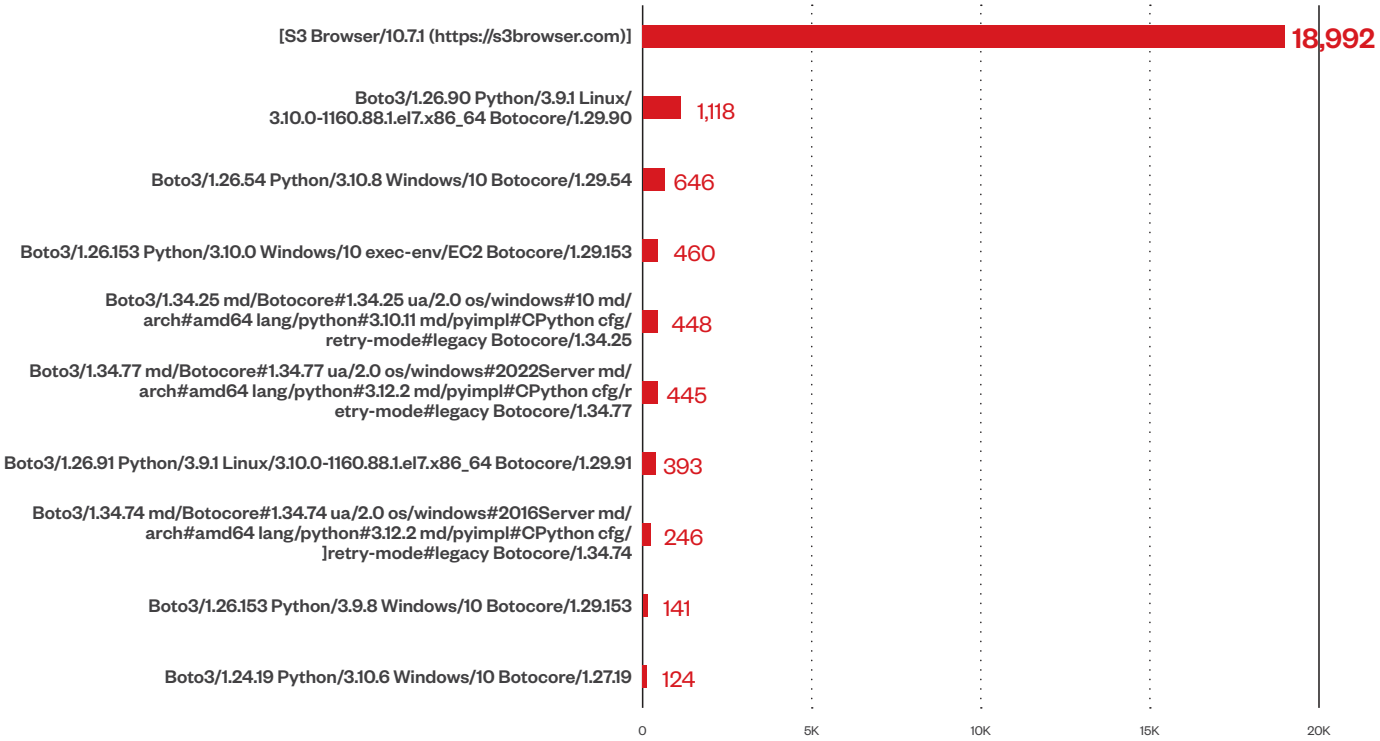


Figure 34. Top 10 most used user-agent/tool that accessed the honeypot

The data captured and processed from this honeypot experiment brings several critical insights:

- The high volume of unauthorized requests highlights the attractiveness of exposed cloud credentials to attackers. The focus on data access and account information requests reflects a strategic approach to exploit misconfigured environments for data exfiltration and further reconnaissance.
- The concentration of requests from specific IP addresses and the use of standard automation tools like "Boto3" library and "S3 Browser" underscore the operational tactics of attackers, leveraging CSP-provided tools and widely used software to scale their efforts.

This operation not only sheds light on the tactics used by attackers but also reinforces the necessity of rigorous security practices in cloud environments.

Leaked Codebase Experiment

Since most code repository platforms perform secret scanning to avoid commits with sensitive information, we focused our research efforts on searching for production-ready codebases hosted and accessible over misconfigured platforms.

We performed queries in two different search engines, Shodan and Censys, which led us to the discovery of secrets across 1,754 unique hosts. The results showed the indexed file names commonly associated with production code, such as .gitignore and .env.

We also found several projects, both self-hosted and inside the GitHub repository. Although the GitHub repositories referenced the .env file inside the .gitignore, preventing them from committing, the self-hosted repositories still had the .env file present and available for download.

Host Criteria

To include a host in our study, it had to meet the relevance standards described here:

- **Presence of .env files or equivalent.** The host must contain exposed .env files or other configuration files that explicitly store environment variables.
- **Containment of recognizable secrets.** The exposed files must include one or more recognizable secrets, such as API keys, access tokens, or passwords.
- **Accessibility.** The codebase or configuration files must be publicly accessible without requiring authentication, indicating an evident leak rather than a controlled share.

The 1,754 hosts combined contained 710 GB of data, and sensitive information was found in 82,687 files. The .env file was the most popular by a large margin, as shown in Figure 35:

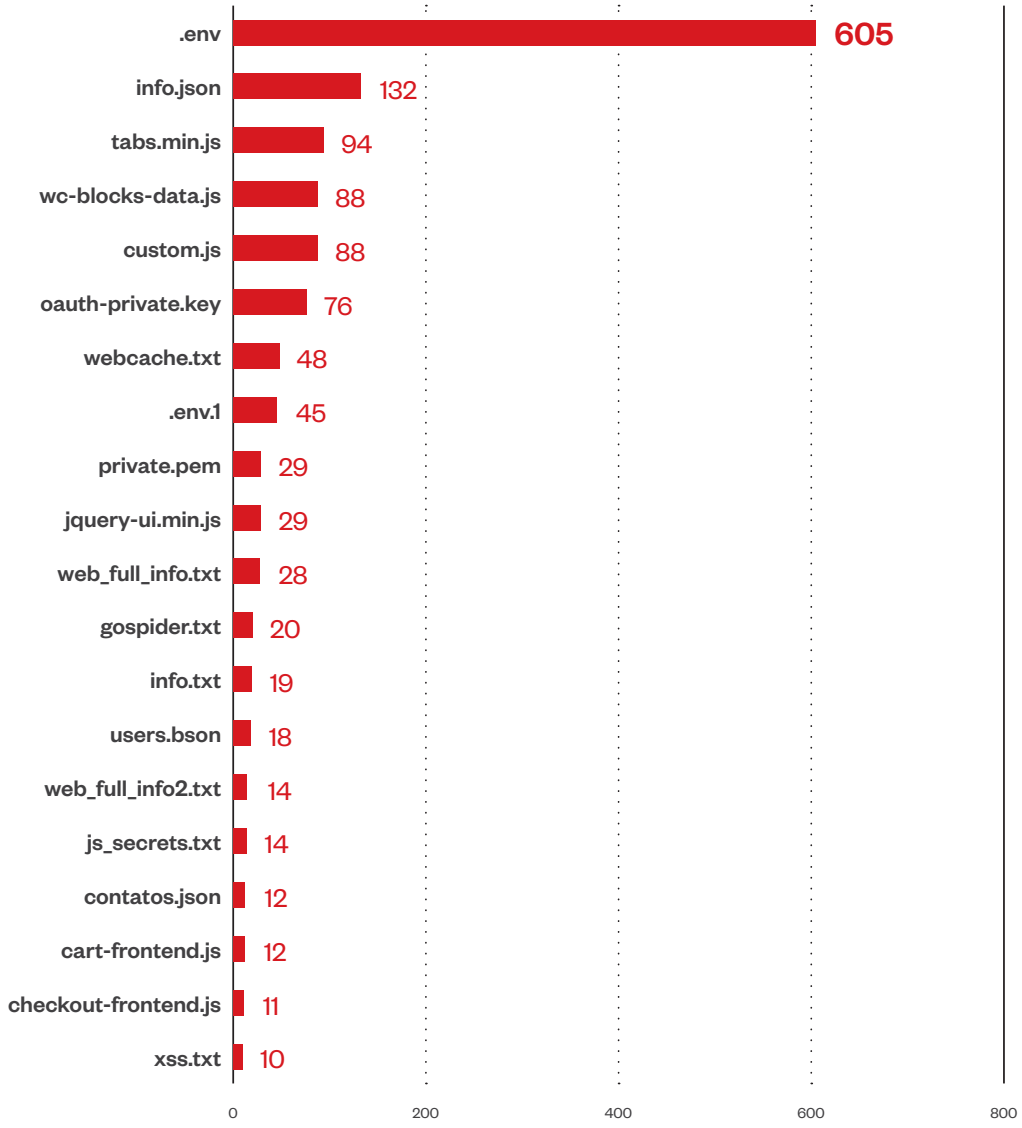


Figure 35. The number of files where secrets were found

The Secrets Analysis

We found 677,426 secrets among 1,754 different sources – secrets that shouldn't have been there in the first place.

In this part, we describe the types of secrets, their location, occurrences, and the impact of their leakage. Our analysis uncovered that most of the unique secrets categories are *generic-api-key*, *jwt*, and cloud *access Tokens*.

Methodology

We used a matching tool with an extensive database of regexes to search for potential secrets on the leaked codebases. The tool with the best performance and database for our purposes was Gitleaks.

The regex database contains very accurate rules for secrets with fixed patterns and with a more generic approach, which could lead to some false positives. All the results were submitted to a manual analysis.

For instance, the most notable pattern for matching AWS Access Key ID is too generic, matching AKIA examples and leading to many false positives. The AWS-recommended regex rule also generates false positives, as seen in Figure 36.

```
$ grep -RP '(?<![A-Z0-9])[A-Z0-9]{20}(?![A-Z0-9])' iam-2010-05-08.examples.json | grep EXAMPLE
            "AccessKeyId": "AKIAIOSFODNN7EXAMPLE",
            "RoleId": "AKIAIOSFODNN7EXAMPLE",
            "UserId": "AKIAIOSFODNN7EXAMPLE",
            "UserId": "AKIAIOSFODNN7EXAMPLE",
              "AccessKeyId": "AKIA111111111EXAMPLE",
              "AccessKeyId": "AKIA222222222EXAMPLE",
              "GroupId": "AGP2111111111EXAMPLE",
          "AccessKeyId": "AKIAIOSFODNN7EXAMPLE",
$
```

Figure 36. False positives with the AKIA pattern

The final numbers presented in this research were manually reviewed and did not include notable false positives.

Since we were scanning a large amount of data and several different code projects, we used the *detect* and *–no-git* parameters to analyze all the code in one run.

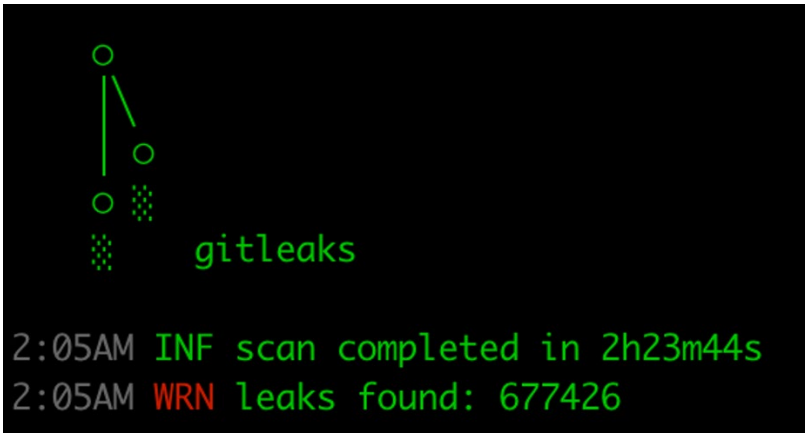


Figure 37. Gitleaks' end of execution showing the total time of scanning and number of leaks found

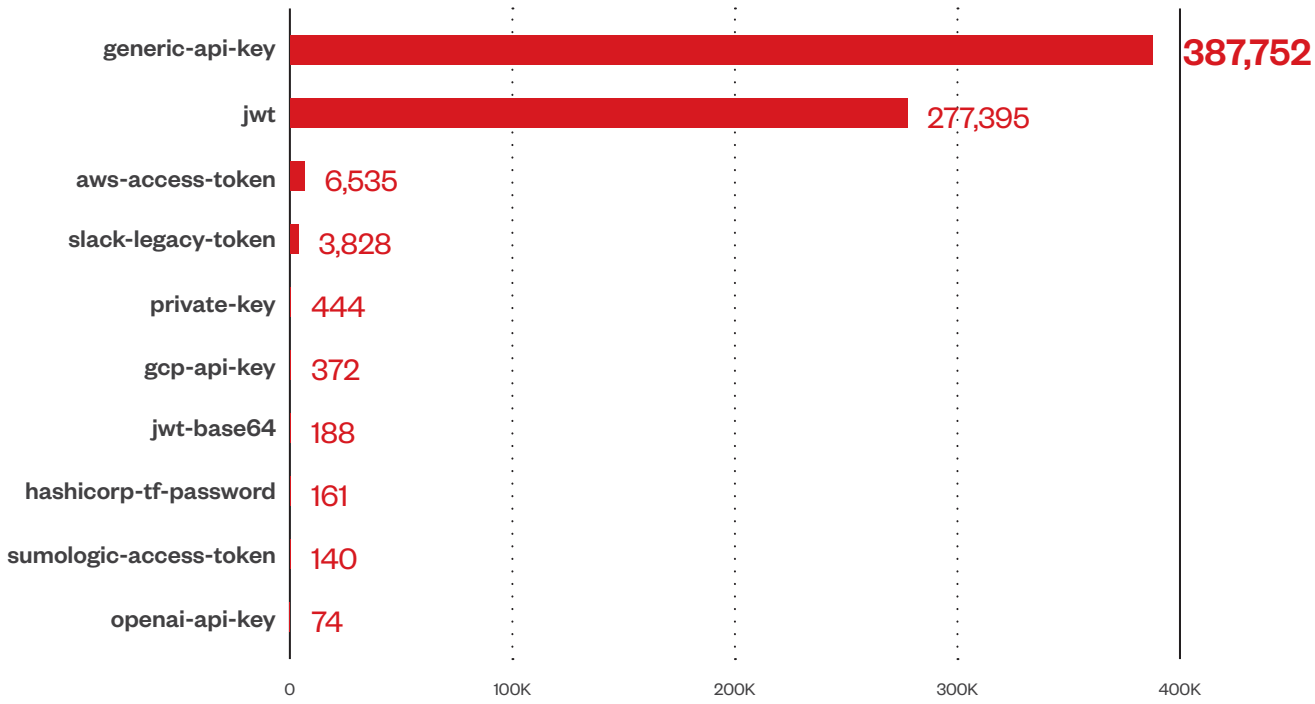


Figure 38. Top 10 leaked secrets with generic-api-key and JWT being the top two secret types

Our analysis has led us to the following important observations:

- We found 387,752 *generic-api-key* secrets. Those are the secrets that do not match any specific regex rule from the database but match a more generic construction of a key or a secret, plus the combination of keywords in the variable name commonly used for this purpose. The large number of generic API keys shows the popularity of their production hard-coding.

```
"Match": "KEY=8m5ytQsBcyAdqAEm",
"Match": "SECRET_KEY=6Le8UFUhAAAAAMgzj63wiY1dPo09JxrAKxR4GWDI",
"Match": "API_TOKEN=cd63f50a-139b-44b0-9bd8-2fdf51816aa53f65be7d-c5e3-4349-933f-d2d53e4ee52d",
"Match": "PASSWORD=0jlRPsD8TaJ2",
```

Figure 39. A few examples of generic secrets matched by the rule

- The second-most prevalent type of secret, with 277,395 occurrences, were JWT tokens, which are used for authentication and enable users to access resources more securely. JWTs are widely used in web applications and APIs for authentication and authorization purposes, ensuring token authenticity by their properties.

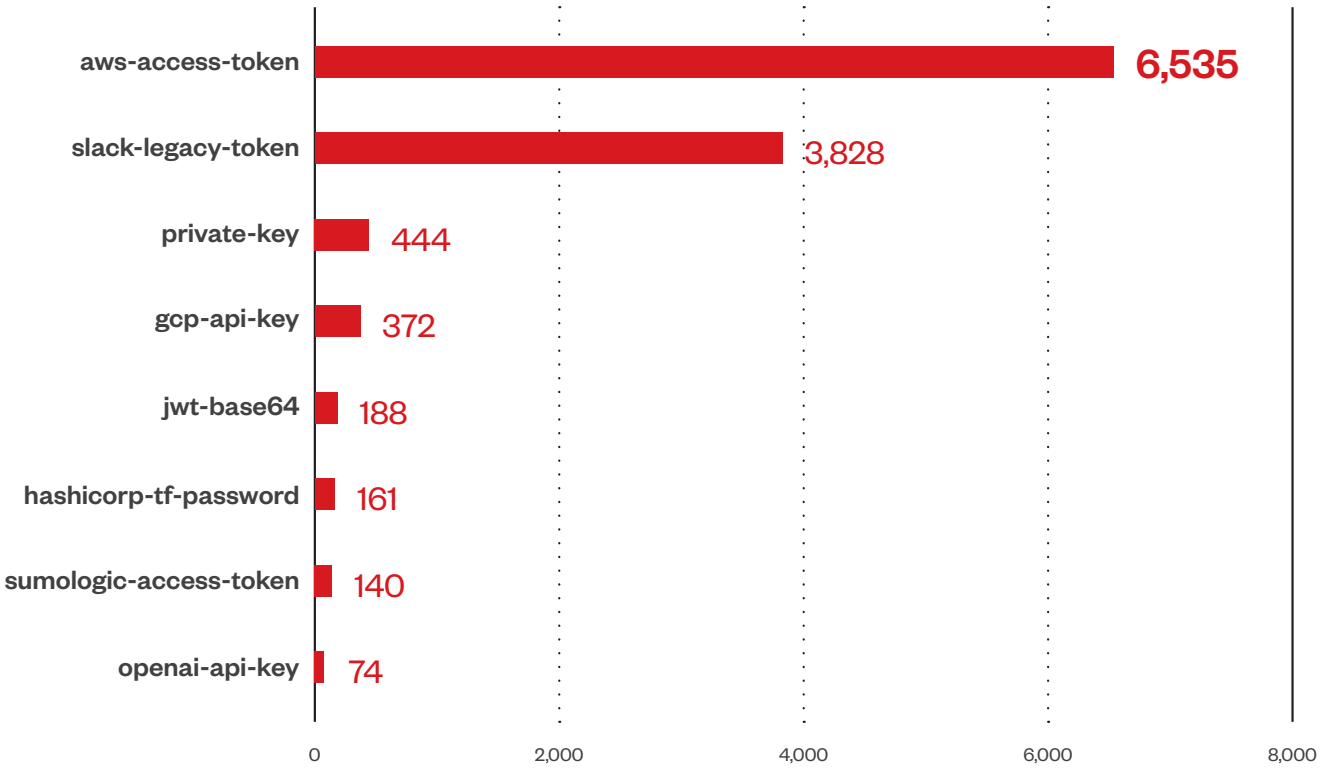


Figure 40. Top leaked secrets excluding generic-api-key and JWT

We highlight the following observations from the top leaked secret, the AWS access token:

- AWS access tokens appeared 6,535 times, allowing threat actors to get into several AWS products and access sensitive data.
- We also found 3,828 slack-legacy-tokens, 444 private-keys, and a few other tokens and keys for different services including GCP, HashiCorp, and Sumo Logic.

The exposure of such secrets is a high security risk. Our research shows the lack of proper secret management among DevOps practitioners and emphasizes the need for appropriate secret management.

The most confidential details found among the .env files are as follows:

- **Private keys (444 instances).** Private keys are used for various purposes, including SSH access, signing digital certificates, and decrypting confidential information. The exposure of private keys compromises the integrity and confidentiality of communication and data.

  AWS access tokens (6,535 instances). Given how organizations extensively use cloud computing services, the exposure of AWS access tokens is a significant cybersecurity threat. For example, attackers could gain access to sensitive data stored inside S3 buckets, launch instances to mine cryptocurrency, or even take over the entire cloud account.
- **Slack legacy tokens (3,828 instances).** These tokens provide access to Slack workspaces, exposing sensitive messages and files. Having those tokens allows threat actors to impersonate users or deploy malicious bots within an organization's communication channels.
- **OpenAI (74 instances).** Although relatively new, we identified a substantial number of OpenAI API keys. Unauthorized users can use those keys to abuse the service, leaving the rightful owner with monetary loss and leading to account cancellation.

Another piece of data that caught our attention is the high number of database and email configurations inside the .env files. These are examples of predominantly sensitive data, which represents a significant portion of the secrets found in the *generic-api-key* category stored in plain text.

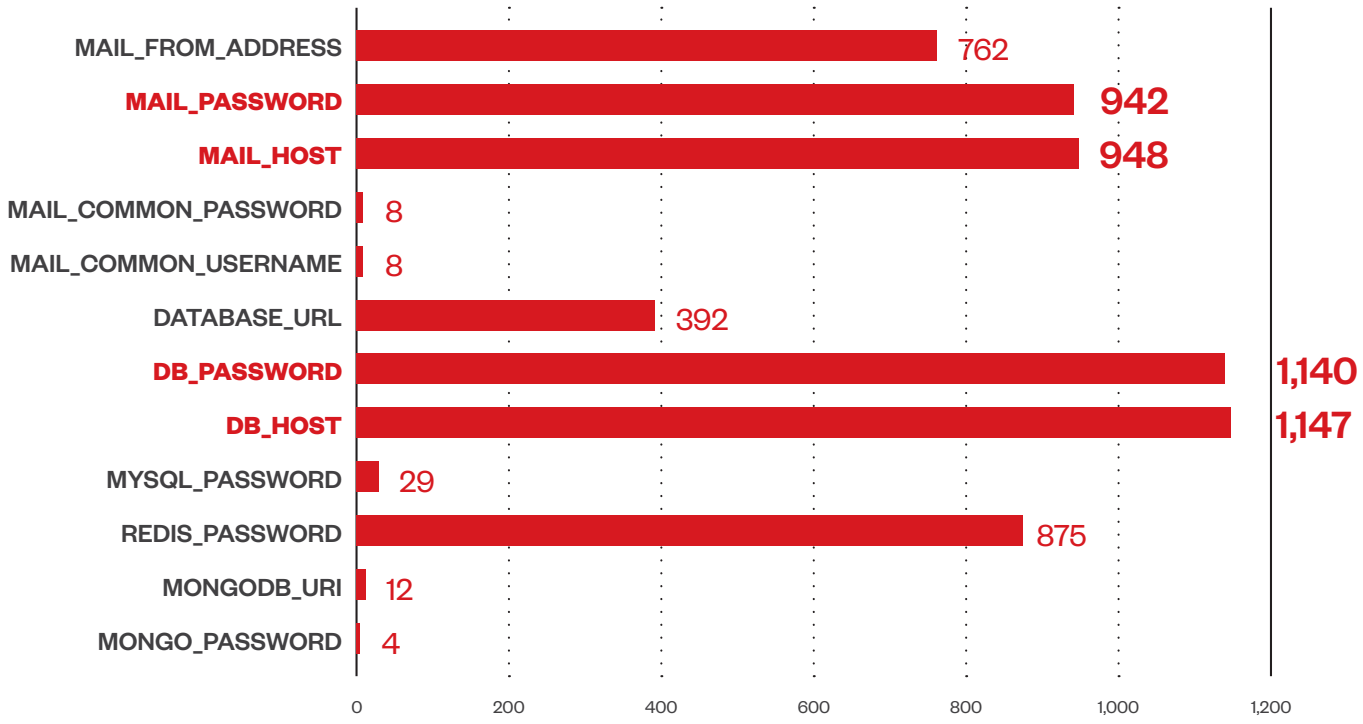


Figure 41. Top environment variable names in .env files, highlighting variable names with "mail" and "database"

Threat Scenarios

If you are not yet concerned about the possible misuse of .env files, we describe the possible consequences here. Let's assume that your organization is using a cloud storage service for application data. It does not matter which CSP you choose, as you will need credentials to access your storage account. Since your organization keeps abreast with ongoing development trends, you use a .env file to configure your application, including your cloud access credentials.

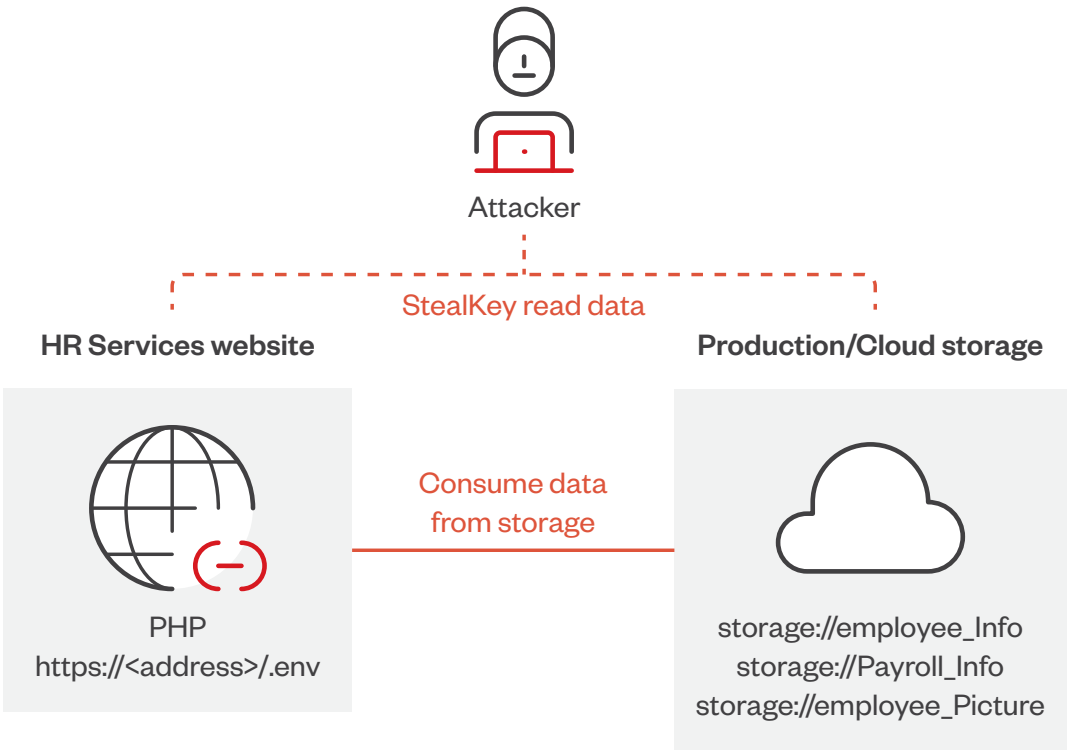
Your organization has a basic understanding of security, and your developers follow the recommendation and reference the .env file inside .gitignore. However, the deadline is approaching, and your organization hires a junior developer. A blocking issue appears late Friday afternoon when the senior developers are already home with their families. The junior developer wants to make an impression and decides to fix the issue. The issue is successfully fixed. Unfortunately, he commits the .env file into production, making it available for everyone.

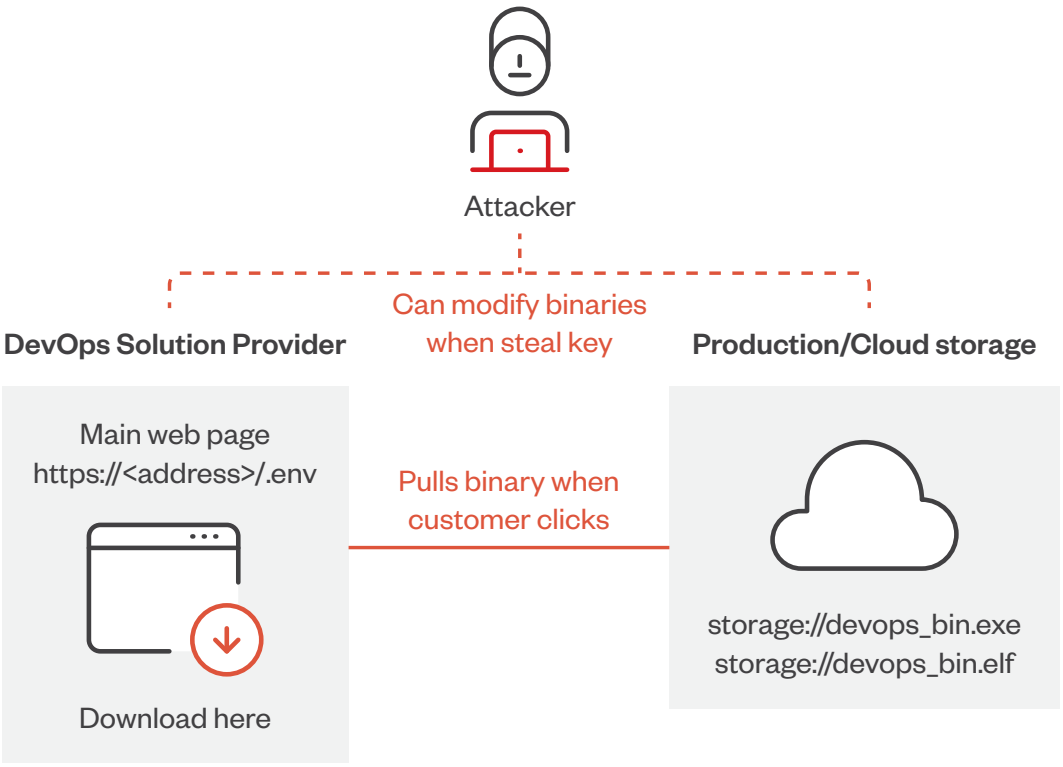
Malicious actors are actively looking out for such mistakes and have more than 48 hours to find them. The threat remains unnoticed unless protection measures are implemented.

What do the threat actors gain?

At this stage, it depends on what the company stores in the storage account and what permissions the token possesses.

Starting with read-only permissions, let's say that your organization develops HR software that stores accounting data for customers and employees. This data includes employee IDs, loan agreements, contracts, payrolls, and images proving workday attendance. As you can imagine, the leakage of a token with just read-only permissions can lead to massive data breaches.





Here is another example: Your company develops a software product, and its binaries are present in the cloud storage account, and those binaries are used for software distribution and referenced by the company website. Your company stores only website static data on the cloud storage account. Unfortunately, the leaked token has write permissions. This means the threat actors can replace the original binaries with malicious code and launch a successful supply-chain attack on your company's customers.

Lastly, not all secrets inside the .env file are related to cloud storage services. With the increasing adoption of AI and LLM, API tokens for those services will become increasingly prevalent.

Let's say your company stores the *OPENAI_API_KEY*, which leaked via an improperly used .env file. Consequently, threat actors can access the platform on the company's behalf. More importantly, since the OpenAI API has limited monitoring capabilities (only the number of requests, not their source at the time of writing), they can silently benefit from using ChatGPT using the company's resources unless they reach a budget cap.

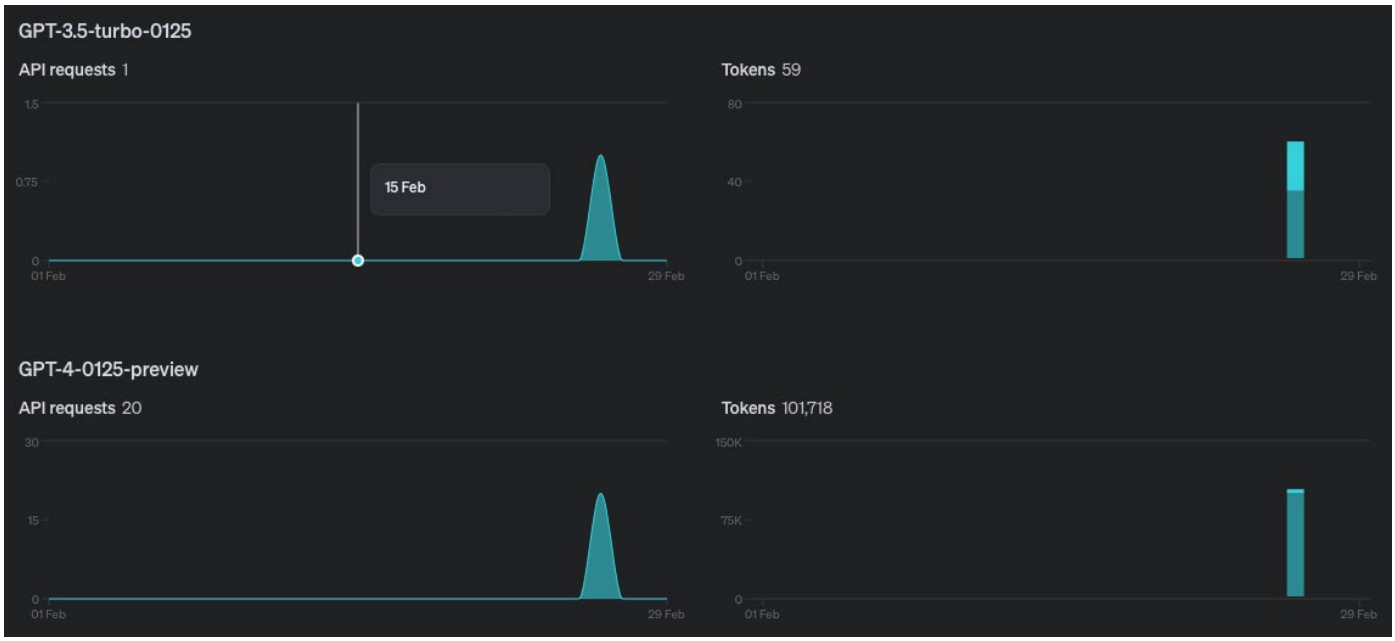


Figure 42. OpenAI monitoring capabilities

Keeping Secrets Secure

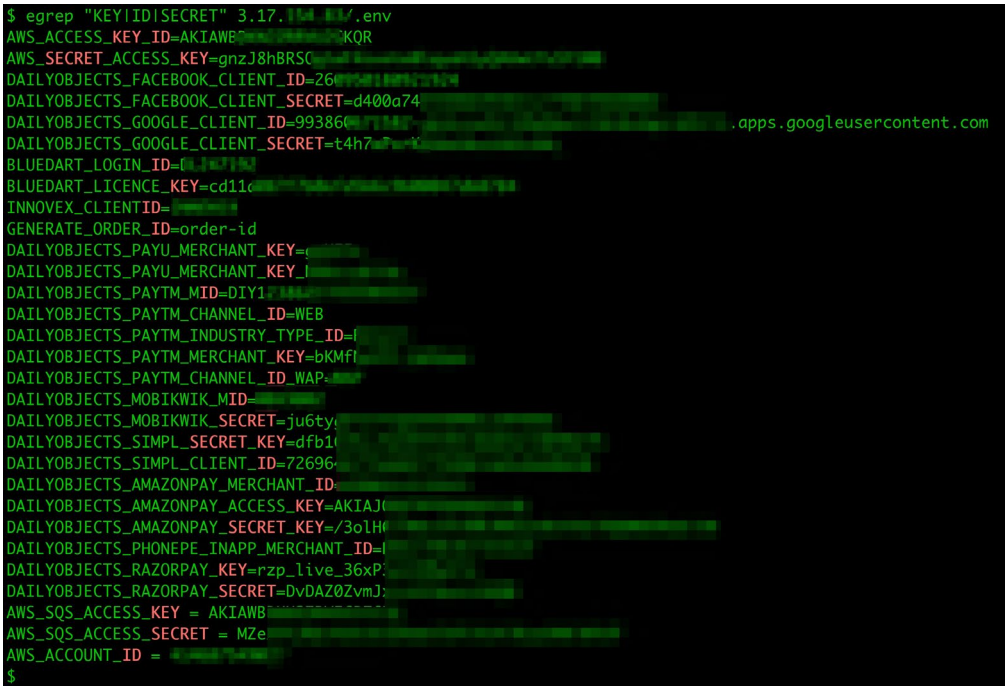
In previous sections, we described various issues surrounding the use of .env files that require our attention. Even though there is never a silver-bullet solution, developers should consider employing the following security mitigations that we elaborate on here.

Token Permissions

Overly permissive tokens are often the root of disastrous security scenarios. Even if we do everything in our power to secure our systems, certain things are out of our control. This is where the "assume breach" paradigm comes in to limit further damage.

This paradigm promotes the limiting of token permissions and validity. The following are some important questions that developers should ask under this paradigm:

- Does the token need to be valid for the entire month?
- Does the web application need write permissions to our cloud storage account?
- Is there a token rotation plan?
- Does the 30-, 60-, or 90-day rotation plan suffice, or do we need a shorter lifespan based on information confidentiality?



Figures 43 and 44. Examples of a good security practice: Each service has its own key.

Secret Hard-Coding

We should never hard-code secrets inside files and store them in plain text. Developers should follow security best practices even in non-production environments. Do we trust all our applications and believe none of them would exfiltrate secrets to services such as VirusTotal or similar? Have we read all the lines of the End-User License Agreement (EULA) of the services you're using?

Even though most of the code examples suggest storing API keys or other secrets inside the environment, consideration should be made depending on the actual use case. Do we know all the consequences, for example, of sharing our API key with other processes when we are developing on Windows?

Figure 45. An OpenAI Cookbook example that suggests using environment variable as a best practice when storing API keys

Source: OpenAI's repository on GitHub[22]

At this point, it's important to ask: Isn't it better to store secrets inside places specifically designed for that purpose, such as vaults or local keyrings?

In Figure 46, we use the Python keyring library to obtain our API token from the secure store managed by our operating system. This is a secure way to manage secrets on development machines, especially when they store cloud service credentials. It is no surprise that many applications are already using this approach (such as CSP portal tokens).

```python
import keyring
from openai import OpenAI


client = OpenAI(
  api_key=keyring.get_keyring().get_credential("openai_apitoken","token").password
)

response = client.chat.completions.create(
  model="gpt-3.5-turbo-0125",
  response_format={ "type": "json_object" },
  messages=[
    {"role": "system", "content": "You are a helpful assistant designed to output improper jokes in JSON."},
    {"role": "user", "content": "Tell me the most nasty joke you can imagine."}
  ]
)
print(response.choices[0].message.content)
```

Figure 46. An example of using OpenAI API to obtain secrets from the operating system secrets manager

Secure Alternatives

The greatest danger of the .env file is using it to store secrets. Doing so is the same as storing them in plain text, with a few exceptions: Threat actors know where and what to look for, and the only surprise in this scenario is the number of stored secrets they will find. Our research confirms that it is not unusual to find secrets inside production environments or container images.

Using the .env file to store non-sensitive information is perfectly okay. However, leaving it inside the production environment might lead to the disclosure of unnecessary information. We discuss other ways to pass secrets that developers should explore in the next sections.

Environment variables

If you read our report carefully, you already know that we do not recommend using environment variables for passing secrets. However, we are aware that many developers would disagree with us. In this case, ensure that your application runs inside a container and has an isolated process namespace. Remember the default inheritance of your environment variables and ensure that the secrets are not passed into child processes when unnecessary. The application should be designed in a way that the container process will be short-lived, which minimizes the time secrets are present in memory. Secret injection should be done from an environment that is unreachable from the container and secrets should be stored in an encrypted vault.

Vaults

Compared to environment variables, vaults could have performance penalties for retrieving secrets. Thus, it makes for a more sensible option for long-living processes. Vaults require code execution to retrieve vault-stored secrets, which is one security advantage of vaults compared to environment variables. Simply put, a malicious actor must trigger vault retrieve code execution instead of reading an environment variable or a file.

Proxies

Instead of passing secrets to the production environment, we can create an extra back-end layer for communicating with third-party services, shielding the secrets and implementation details from production environments. This is especially useful in applications where users can introduce vulnerabilities by encoding their content. The implementation can be as simple as using an extra container.

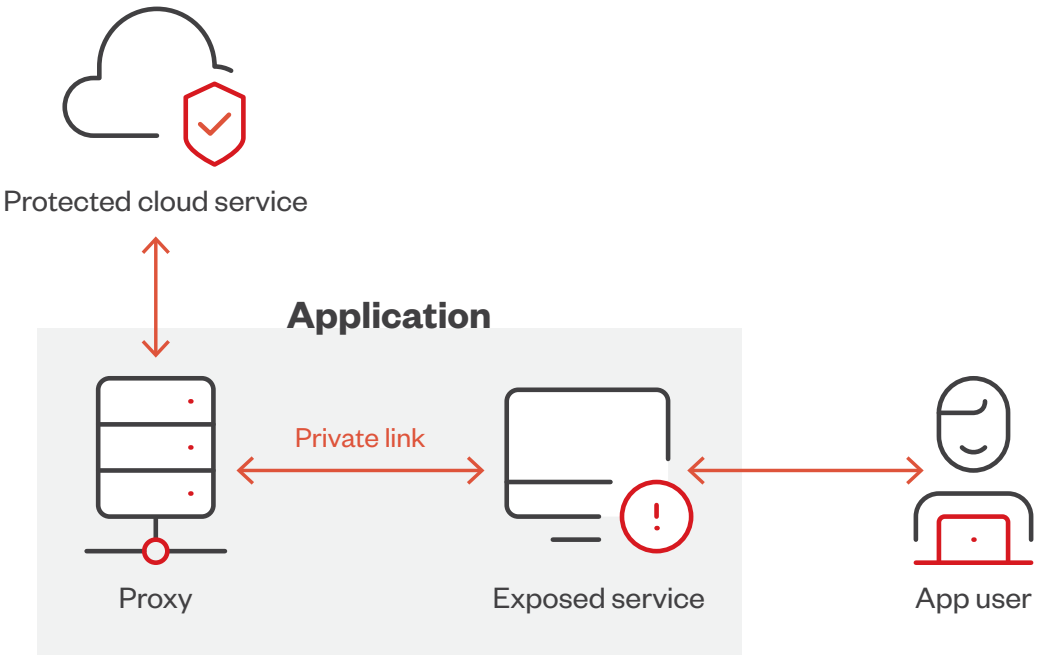


Figure 47. An example of a proxy architecture shielding cloud credentials from the exposed application service

Secretless

We get used to authenticating for every service. But we should ask ourselves if we really need to use secrets for every use case. It might sound contradictory, as using authentication and role-based access control (RBAC) are two of the most emphasized security clichés, but when we think about a use case of two containers sharing the same private network, do they need a secret for authentication? Or are we only creating a false sense of security while the real protection is still missing?

Secrets handling	Risks	Security boundary
Environment variables	Inheritance, long-lived	I/O or memory read access
.env file	Aggregated secrets, production commits + environment variables	File read access, process memory access
Vault	Aggregated secrets, single secret access	Network access/Code execution
Proxy	Proxy compromise, network attacks	Network access/Code execution
Secretless	Breach of trust, vulnerabilities	Code execution, network/Device compromise

Table 1. Secrets handling risk matrix

Secrets Scanning

Even though we minimize the number of secrets in our environments by using access policies, we will not become completely secretless in the near future. What we should focus on is minimizing the risk of exposure. We have, on multiple occasions, written about the importance of secrets management. The more research we conduct on the topic, the more apparent it becomes to us that successful secrets management does not merely involve storing secrets in the vault or knowing about the dangers of environment variables or the .env file – rather, the key is tracking the entire life cycle. This means tracking everything at every layer: from generation, usage, until the final rotation and disposal.

Our research shows that proper secrets management will not be achieved if safe and secure measures are applied in only one layer but are completely ignored in another. For instance, we reference the secrets inside one container image to prevent hard-coding. At the same time, we hard-code the same secrets inside a build container and, more tragically, push them into the same container registry, where we let them be exposed. Another example described in this report is when we stored the secrets locked in a vault and then irrationally retrieved them to put them into the .env file stored in plain text and add them into a container image[23] or leave them exposed[24] inside a production environment.

One way to tackle this problem is to implement secrets scanning to mitigate the risk of these issues. However, no scanning engine is perfect, and developers must be aware and educated.

We should therefore never fall for convenience in place of security, especially when cloud service credentials are at stake.

Zero-Trust Architecture

Adopting a zero-trust architecture involves designing services on the assumption that breaches are inevitable. Therefore, no device inside or outside the network or environment should be trusted by default. This model requires strict identity verification, least-privilege access, and continuous monitoring for every access request to resources and secrets. Zero-trust principles extend to secrets management by enforcing authentication and authorization for every attempt to access a secret, regardless of the requester's location or role. This approach minimizes the attack surface and reduces the risk of unauthorized access to sensitive information. Implementing a zero-trust architecture requires a comprehensive strategy adapted to specific scenarios.[25]

Conclusion and Best Practices

Unfortunately, all systems are not impregnable to threats; we will never achieve one-hundred-percent security, which is why we should always have a plan in case of a security breach. Prompt, plan-based actions reduce our reaction time and minimize the aftermath costs when a breach occurs.

Assuming we discovered that our credentials referenced in the .env file have been leaked, we must rotate all the secrets from the .env file and invalidate them. We should also rotate all the secrets present or reachable from the affected environments and services. This includes passwords, private keys, and access tokens.

In this sample case, we can imagine how painful it would be to replace all the hard-coded secrets instead of performing secrets regeneration inside a vault and their dynamic usage. Having a prepared scenario in automated form dramatically reduces the reaction time needed to address security incidents.

Endnotes

1 IEEE and The Open Group. (n.d.) *The Open Group*. "The Open Group Base Specifications Issue 7, 2018 edition." Accessed on June 3, 2024, at Link.

2 IEEE and The Open Group. (n.d.) *The Open Group*. "The Open Group Base Specifications Issue 7, 2018 edition: setenv." Accessed on June 3, 2024, at Link.

3 IEEE and The Open Group. (n.d.) *The Open Group*. "The Open Group Base Specifications Issue 7, 2018 edition: unsetenv." Accessed on June 3, 2024, at Link.

4 IEEE and The Open Group. (n.d.) *The Open Group*. "The Open Group Base Specifications Issue 7, 2018 edition: putenv." Accessed on June 3, 2024, at Link.

5 Man7.org. (n.d.). *Man7.org*. "exec(3) — Linux manual page." Accessed on June 3, 2024, at Link.

6 Prasoon_mishra. (May 15, 2019). *GeeksforGeeks*. "export command in Linux with Examples." Accessed on June 3, 2024, at Link.

7 David Fiser and Alfredo Oliveira. (Mar. 28. 2023). *Trend Micro Research, News, and Perspectives*. "The State of Serverless Security on Microsoft Azure." Accessed on June 3, 2024, at Link.

8 Muhammad Ilyas. (Jan. 14, 2024). *Medium*. "Efficient Ways to Handle the Environment Secrets Variables in Large Scale Applications." Accessed on June 3, 2024, at Link.

9 Youdiowei Eteimorde. (Dec. 13, 2023). *DEV Community*. ""Everything is a file" Explained." Accessed on June 3, 2024, at Link.

10 Laravel. (n.d.). *Laravel*. "Configuration." Accessed on June 6, 2024, at Link.

11 GitHub Docs. (n.d.). *GitHub Docs*. "About secret scanning." Accessed on June 7, 2024, at Link.

12 David Luzar. (Feb. 1, 2024). *GitHub*. "excalidraw/.env.development." Accessed on June 7, 2024, at Link.

13 Milos Vetesnik and David Luzar. (Jan. 29, 2024). *GitHub*. "excalidraw/.env.production." Accessed on June 7, 2024, at Link.

14 Motdotla. (May 28, 2024). *GitHub*. "motdotla/dotenv." Accessed on June 7, 2024, at Link.

15 Pieter Frenssen. (Nov. 12, 2023). *GitHub*. "PHP dotenv/README.md." Accessed on June 7, 2024, at Link.

16 Motdotla. (May 28, 2024). *GitHub*. "motdotla/dotenv." Accessed on June 7, 2024, at Link.

17 Alfredo Oliveira. (Aug. 11, 2020). *Trend Micro Security News*. "Shedding Light on Security Considerations in Serverless Cloud Architectures." Accessed on June 7, 2024, at Link.

18 David Fiser and Alfredo Oliveira. (Mar. 28, 2023). *Trend Micro Research, News, and Perspectives*. "The State of Serverless Security on Microsoft Azure." Accessed on June 7, 2024, at Link.

19 David Fiser and Alfredo Oliveira. (July 20, 2021). *Trend Micro Security News*. "TeamTNT Activities Probed Credential Theft, Cryptocurrency Mining, and More." Accessed on June 7, 2024, at Link.

20 David Fiser and Alfredo Oliveira. (Sept. 26, 2023). *Trend Micro Security News*. "Exposed Container Registries: A Potential Vector for Supply-Chain Attacks." Accessed on June 7, 2024, at Link.

21 David Fiser and Alfredo Oliveira. (Oct. 9, 2023). *Trend Micro Security News*. "Mining Through Mountains of Information and Risk: Containers and Exposed Container Registries." Accessed on June 7, 2024, at Link.

22 Logan Kilpatrick. (Jan. 25, 2024). *GitHub*. "openai-cookbook/examples/vector_databases/supabase/semantic-search.mdx." Accessed on June 11, 2024, at Link.

23 David Fiser and Alfredo Oliveira. (Sept. 26, 2023). *Trend Micro Security News*. "Exposed Container Registries: A Potential Vector for Supply-Chain Attacks." Accessed on June 11, 2024, at Link.

24 David Fiser and Alfredo Oliveira. (Oct. 9, 2023). *Trend Micro Security News*. "Mining Through Mountains of Information and Risk: Containers and Exposed Container Registries." Accessed on June 11, 2024, at Link.

25 Trend Micro. (March 31, 2023). *Trend Micro Security News*. "Securing Cloud-Native Environments with Zero Trust: Real-World Attack Cases." Accessed on June 11, 2024, at Link.