WARATEK

WHITEPAPER

An Introduction to Scalable Security-as-Code

This whitepaper aims to provide a clear starting point to Security-as-Code knowledge. We make a conscious effort to describe Security-as-Code objectively and optimistically for what it can be.

A Shift Has Happened

We live in a world where 84% of software exploits happen at the application layer. Yet we continue to rely on vintage security techniques at the network layer to protect enterprise applications and the millions of users that use them.

Whether your organization uses a WAF, RASP, or a combination of SAST, DAST, or IAST, the only reliable approach to address these vulnerabilities is to patch the codebase.

Still, we make assumptions about risk in the form of heuristics that require a significant amount of manual investigation. Today enterprises deploy code multiple times a day, and Security teams must keep pace with each deployment where each code change can introduce new and previously patched vulnerabilities.

Three factors make this increased speed unsustainable for Security teams:

1. Fixing vulnerabilities is manual
2. Existing tooling adds noise rather than value
3. Code changes lead to vulnerability regressions

This never ending effort keeps them running on a metaphoric treadmill where they move fast but go nowhere.

The reality is simple - the current approaches are not sustainable for security professionals. If you need proof that the status quo is no longer adequate, look no further:

- A record-breaking number of 1,862 data breaches occurred in 2021.
- Gartner predicts that 99% of successful cyberattacks will continue to result from known but unpatched software vulnerabilities.
- NIST adds a new software vulnerability to the National Vulnerability Database every 30 minutes on average.

A 2022 report from Palo Alto Networks stresses how hackers start scanning for vulnerabilities within just 15 minutes once a new CVE is published.

With hackers becoming more dangerous than ever in recent years, it can take them mere minutes to find a weak point in their target's applications.

In contrast, the average time to fix these critical software vulnerabilities sits at an all-time high of 205 days.

If traditional cybersecurity approaches aren't decreasing the frequency of attacks or the time it takes to patch the vulnerabilities, how do we, as a community, improve our approach?

Introducing Security-as-Code

Security-as-Code (SaC) is the practice of leveraging machine-readable definition files that use high-level descriptive coding language to apply immutable and continuous security behavior.

This approach drastically reduces reliance on human intervention and grants security teams autonomy while allowing engineers to focus on development rather than vulnerability remediation.

While Security-as-Code is still in its early days, many resources are available despite a need for more consensus around what exactly it is. The issue with these resources is that their content could be more organized and less self-serving.

There needs to be a clear and objective path to Security-as-Code knowledge. To be as objective as possible, we'll use Infrastructure-as-Code as a blueprint for Security-as-Code.

Using Infrastructure-as-Code as the blueprint means there are three key pillars that any SaC solution needs to exemplify:

1. Immutability

The rules you define should remain accurate to prevent vulnerability regressions with each deployment. Security must be continuous to achieve immutability - not at a single instance like in the CI/CD pipeline.

To perform security continuously, the protection applied to your applications and APIs must be inseparable. Due to this nuance, utilization of the runtime or VM is necessary.

2. Scalability

Cost and human capital are the reasons current security solutions don't scale. WAFs and RASPs, for example, are essentially massive data pipelines that ingest, analyze, and make assumptions on HTTP payloads to determine risk.

This focus on HTTP payloads, a lagging indicator, results in false positives and negatives, exasperating the scalability issues of existing Security solutions.

Static analysis tools only do marginally better by telling you what code to fix. Then your teams are left with the manual task of patching the vulnerabilities.

The only way to improve security scalability is to automate the process of patching code. Compilers have operated on this philosophy for decades to improve performance.

The same philosophy can also correct known vulnerabilities and insert security rules that block unknown vulnerabilities (zero days). When you fix the vulnerable code, you eliminate false positives and negatives and the need for expensive hardware.

This decreased friction enables security teams to, for the first time, scale with modern software development.

3. Performance

In theory, an infallible security solution works, but we need to meet business demand in the real world. Because of that, most security solutions recommend running in sample mode, where only a handful of requests are protected.

Companies shouldn't have to choose between being protected and remaining competitive.

Companies no longer have to make this choice when your security solution has proximity to the application and APIs through living in the runtime or VM and adopting the philosophies from compilers.

Security-as-Code implementations average less than 2% performance impact at scale.

How does Security-as-Code Work?

Security-as-Code solutions do not require direct interaction with the application code or prior knowledge of the application.

When executing a method on your application for the first time and attempting to exploit a vulnerability, the JVM or CLR replaces the execution in milliseconds by performing a checksum check without downtime, source code changes, or tuning.

On any additional calls to that same method, only the protected version of your code is available, resulting in even faster execution.

Compared to the complex nature of a WAF or RASP and the overhead of managing large numbers of rules, the business benefits are clear: unprecedented accuracy, low maintenance, virtually zero performance overhead, and no disruption of service once deployed.

Declarative and Immutable Security-as-Code

In declarative approaches, you specify the desired final state of the security you want to apply without dictating how to get it.

This behavior is ideal for Common Weakness Enumerations (CWEs), which act as dictionaries of vulnerabilities and refer to software weaknesses rather than specific instances of vulnerabilities.

Declarative programming minimizes the factors that could affect the behavior of a piece of code to ensure that only a function's input should affect the output, not what's happening elsewhere in the program.

This approach enables Security-as-Code's immutability and makes it possible to prevent vulnerability regressions.

For example, if SQLi vulnerabilities are rampant in your applications, it's possible through Security-as-Code platforms to declaratively tell your applications that you never want to see another SQLi again.

From the time you deploy your SQLi rule, there's no code that developers can add to the codebase that will override your rule.

Due to the nature of these rules, they inherently remediate zero-days in some instances. This inherent protection is the case for CVE-2022-42889, in which the process forking rule is more than sufficient for payloads supplied as Javascript code, as seen below:

```
app("nashorn CVE-2022-42889"):
    requires(version: ARMR/2.2)
    process("Deny any process execution"):
        execute("*")
        protect(message: "", severity: 7)
    endprocess
endapp
```

Security-as-Code solutions also provide imperative rules for more involved vulnerabilities like Log4shell (CVE-2021-44228), where conditionality and context are needed.

Imperative and Instant Time-to-Remediate

In the imperative approach, the SaC solution helps you prepare automation scripts that apply your security one specific step at a time.

Imperative rules require a higher level of domain experience with your applications. Still, the reward is more control over how you

accomplish vulnerability remediation, which is ideal when you need to make small changes, optimize for a specific purpose, or account for software quirks.

Log4shell is an excellent example where following the recommended patching of removing the JNDI Lookup class can break some applications.

Rather than completely removing the class, it's possible with imperative rules to specify specific conditions in your applications where you want the JNDI Lookup class to function but place constraints around what you expect from the outputs.

Below is an example imperative rule for CVE-2021-44228:

```
app("APACHE LOG4J - CVE-2021-44228 - v1.2, b2"):
    requires(version: ARMR/2.2)

    patch("CVE-2021-44228 :01"):
    function("org/apache/logging/log4j/core/net/JndiManager.
lookup(Ljava/lang/String;)Ljava/lang/Object;",
        checksums: ["da55340ac1",
                    "02c6120d62",
                    "b04cf027e3",
                    "d3ad3c6d00",
                    "bb0462e72d"])
        entry()

        code(language: java):
            public void patch(JavaFrame frame) {
                String payload = frame.
loadStringVariable(1);
                log("Forcing JndiManager.lookup() to return
'null' due to CVE-2021-44228", payload);
                frame.returnObject(null);
            }

            private static void log(String msg, String
payload) {
                ArmrEvent event = ArmrEvent.load("ALERT",
"HIGH");
                event.addExtension("msg", msg);
                event.addExtension("payload", payload);
                event.commit();
            }
        endcode
    endpatch
endapp
```

Legacy Application Modernization

Businesses often struggle to update out-of-support applications incompatible with newer versions of their languages.

The only option for mitigating risks posed by EOL languages and libraries has been to rewrite or replace the applications at a high cost in time and money. However, until replaced, they pose a significant security and compliance risk to the business.

Security-as-Code can virtually upgrade out-of-support applications without source code changes by wrapping the entire application stack in a modern version of the programming language, helping you take advantage of a modern tech stack's security and compliance benefits.

A prime example is upgrading legacy applications from TLS 1.0 to 1.2.

Summary

Enterprises like yours use Waratek to provision, apply, and orchestrate application security for all their applications where it's most impactful - the live binary.

Our products are standardized across the ecosystem, used by security practitioners and leaders worldwide, and trusted by enterprises everywhere.

Prior to the introduction fo Security-as-Code, it's never been economically feasible to secure every application and significantly lower your risk profile.

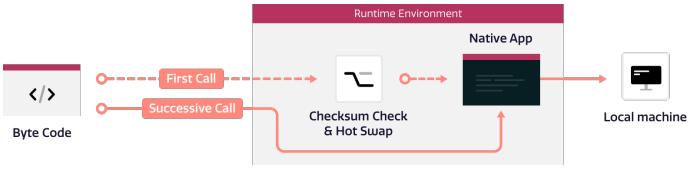
However, Security-as-Code enables you to fix the executing code rather than make assumptions on a lagging indicator like network data, and the following happens:

- False positives and negatives are non-existent
- The human capital needed to secure applications decreases
- The economics of securing every app becomes possible

For more information on Security-as-Code visit Waratek.com.

> **Utilizing security as code enables organizations to scale with modern software development by codifying security and policy into development processes and workflows.**
>
> – Melinda Marks, Senior Analyst, ESG

How it Works



When an action is performed on your applications for the first time, and an attempt is made to execute vulnerable code, Waratek Secure performs a checksum check and tells your application to ignore the code.

A healthy version of the code is returned in real-time as defined in your Policy Config file or the Waratek Portal. Only the healthy version will be made available on any additional call to that same piece of code, resulting in even faster execution.

Technical Specs

Feature	Notes
Agent Size	3MB
CPU Utilization	< 2%
Memory utilization	25MB
Network Utilization	Negligible at scale

See first hand how Waratek can help you

Immutable Security
Say goodbye to regressions after deployments. Once your policy is defined, no code added to the codebase can supersede the policy.

Deployment Agnostic
Securing your runtime instead of your codebase or CI/CD pipeline enables critical patches to be applied instantly instead of during the next deployment window.

Economically Scalable
Remove the toil of false positives and long feedback loops between security and engineering to transform the economics of AppSec.

Ready to Scale Security with modern Software Development?

Take a guided tour to learn how to accelerate your adoption of Security-as-Code to deliver AppSec at scale. No email required.

Start tour