WALLARM AI ENGINE: HOW IT WORKS

Does AI really improve security? We are often asked how Wallarm technology makes its decisions on what traffic is malicious and how we know where application vulnerabilities might exist.

Whether we are talking about a load balancer-based (NIDS, WAF), host-based or in-application solution (HIDS, RASP), the key aspect of the solution's effectiveness and efficiency is its detection mechanism and decision- making process. In all cases similar general detection techniques can be applied. In this paper we want to describe why Wallarm's approach is principally different from that of the legacy solutions.

"Intelligence involves a great deal more than ability to follow rules... it is also the ability to make up the rules for oneself, when they are needed, or to learn new rules through trial and error."

Steve Grand, author of Creation

TABLE OF CONTENTS

OVERVIEW

The main task of the run-time application security is to protect modern applications and APIs.
In this endeavor the solutions face a number of challenges:

- Applications are different both in structure and in content. Things that are harmful to one application may be perfectly normal for another.
- User behavior varies between both the applications and the individual application functions. For example several login calls every second may indicate a credential stuffing attack, while several data layer queries per second may be a normal function of building a correlated data set.
- The number of known attacks keeps growing, with attack patterns (or signatures) sometimes being hidden within nested protocols
- Straightforward implementation of attack detection based on signatures often results in a high rate of false positives and false negatives,

Wallarm relies on its AI engine to solve the above challenges. Wallarm approaches the application protection by implementing three important tasks.

1. BUSINESS LOGIC MARKUP

This is the first phase of the entire learning process. In this phase Wallarm applies machine learning to the task of pattern recognition to identify application functions (aka endpoints). This is done by identifying **features** in the application traffic. In machine learning a feature is an individual measurable property, characteristic of the pattern being observed. Choosing features that are both independent and discriminating is critical to the effectiveness of the ML model. String and graph features are common for syntactic pattern analysis.

In Wallarm's model, each feature represents a separate HTTP request field with a value. It can be a part of the URL path, a JSON parameter inside a request body, an HTTP header, or anything else.

By applying this model, Wallarm is able to derive a full application business logic profile. For example, the business logic may include specifying, that the login function must be called by HTTP requests to the auth.domain.com host, with an /api/call URI and JSON method inside with a "method" parameter with a value of "authenticate".

2. PROFILING NORMAL BEHAVIOR

For each of the different application functions recognized at the **Business logic markup** stage, Wallarm creates a behavior profile. This profile consists of two different machine learning models: data format model and user behavior model.

Data format is a statistical model for character combination (or n-grams) distribution functions for each of the data parameters related to this application call variant, a.k.a. endpoint. For example, it can describe that the username should be patterned like an email and the password should have eight or more characters with at least one special character inside.

The user behavior model is a machine-readable representation of all the normal user activity, including how often they are calling each particular endpoint, what the expected order is for calling the endpoints (like wizard steps), what the prerequisites are for what calls and so on.

3. ATTACK DETECTION

This phase comes into play if a request falls outside the normal behavior model identified in the previous stage. A request may contain either a data or a behavior anomaly. Data format statistical model anomalies can be present in one or many data fields. A user behavior anomaly may include an attempt to pass through the second step of the wizard without passing the first step, guessing passwords too frequently, site crawling, etc.

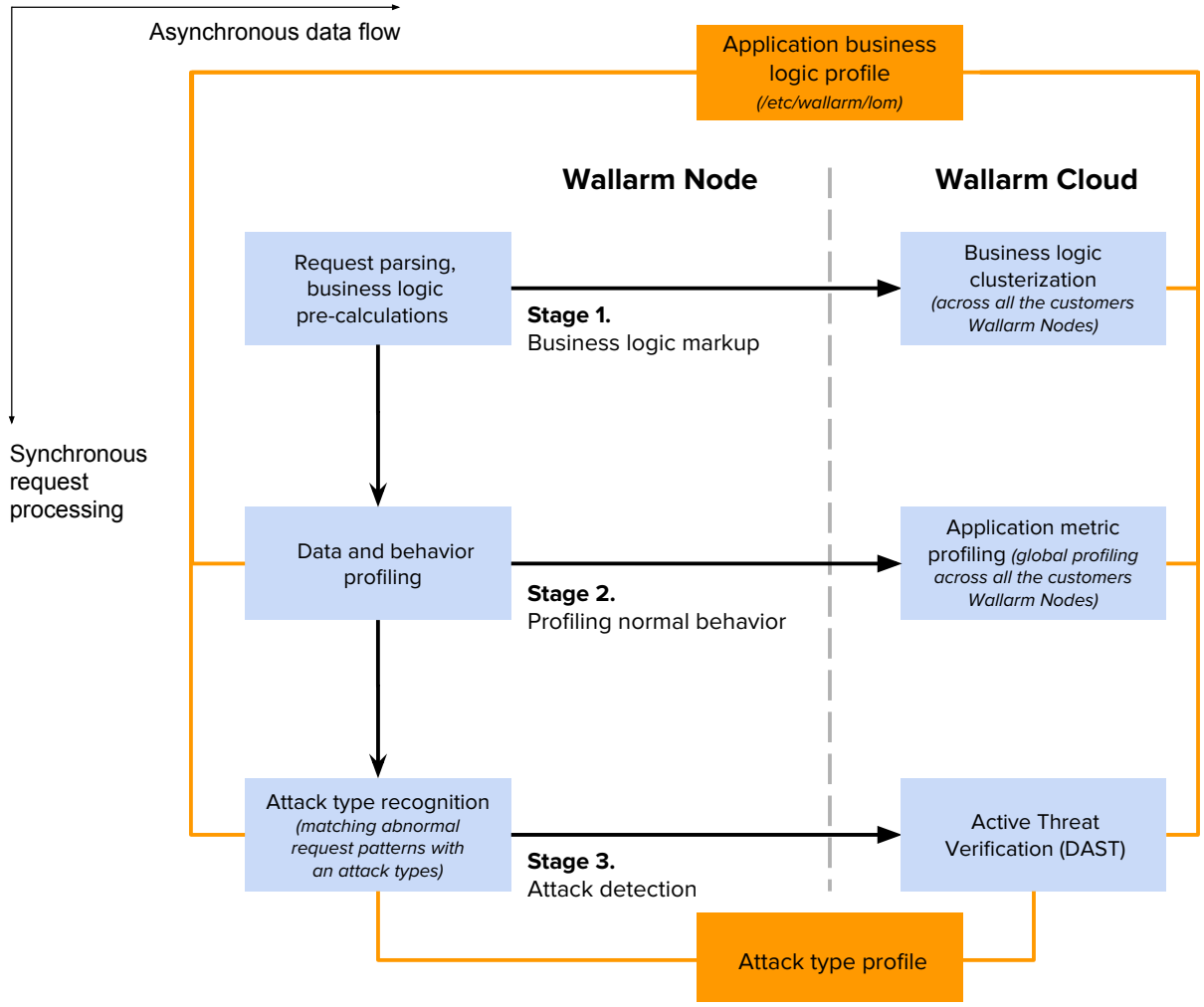
For all these cases, once the anomaly is detected Wallarm needs to classify an attack type. Data anomalies are mainly related to SQL injections, XSS, SSRF, XXE and other input validation attacks. User behavior anomalies are related to password bruteforce, credential stuffing attacks, authentication issues, etc.

Further details of the Wallarm attack detection logic algorithm are described in the "Evolution of Real Time Attack Detection" whitepaper.

While the high level description of the three tasks is fairly straightforward, the actual implementation is far from easy. Part of the challenge is that Wallarm splits its decision making between the locally installed Wallarm node and the AI-engine in the Wallarm cloud.

This separation is driven by the considerations of performance and data privacy.

An aerial bird's eye view of Wallarm architecture is shown in the following diagram:



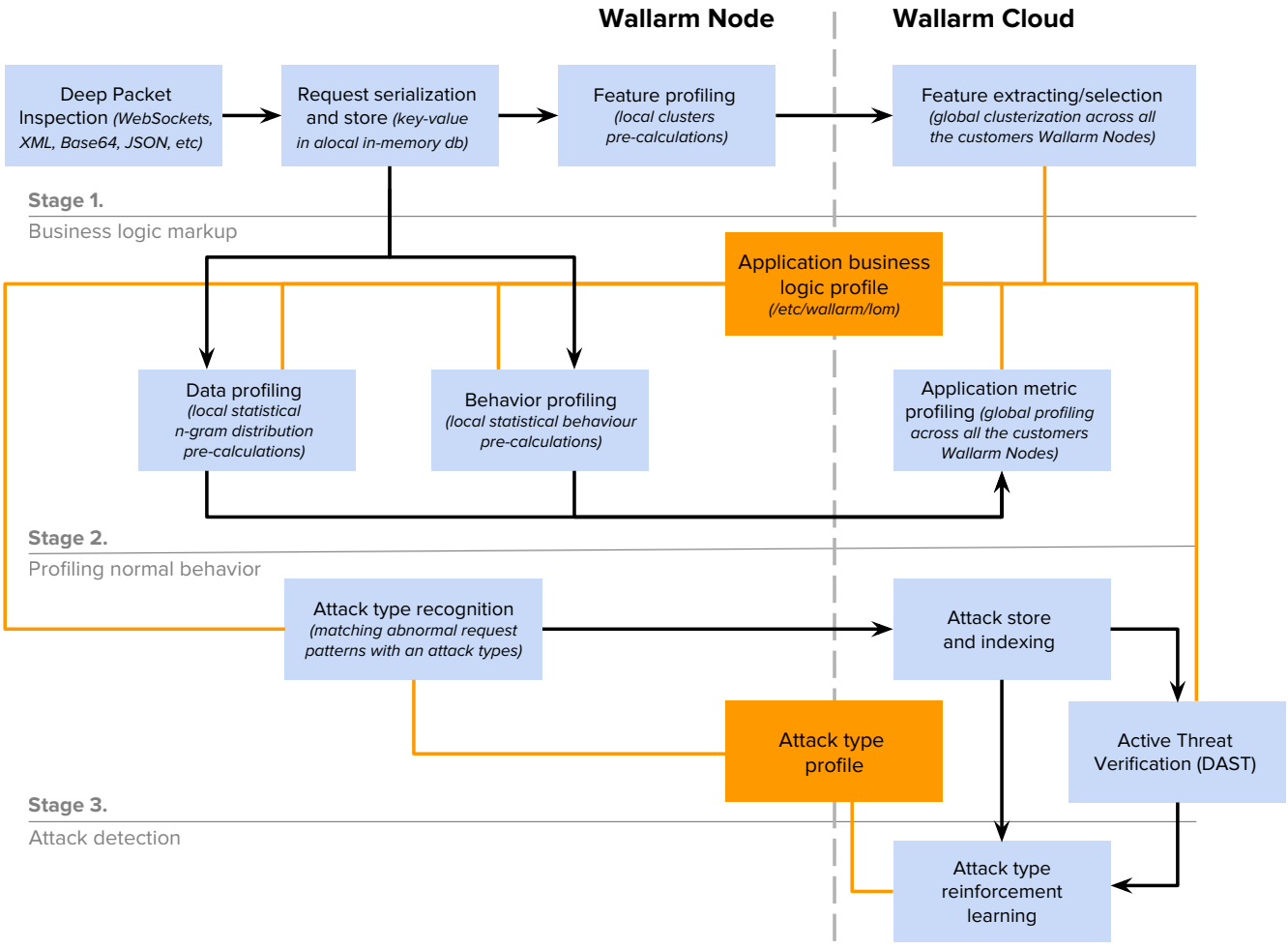
With this more sophisticated approach Wallarm is much more efficient in protecting web applications than legacy solutions. **An overall comparison table is shown below:**

	Legacy attack detection	**Machine learning attack detection**
Products/libraries	• PCRE • CoreRuleSet/ mod_security • libinjection • Naxsi	Wallarm approach
Performance	low	superior
Accuracy	Poor detection of both false positives and false negatives	Lowest false positives rate and automated detection of legal false negatives: avoiding blocking control panel, security-related blog posts. False positives training across the cloud.
Maintenance	Requires manual rule updates and tuning when new signatures are published	Security rules are updated automatically adapting to traffic
Handling of encoded APIs	Manual configuration required for each API call in case of data encoding. Multiple nested encoded API calls (i.e. base64 inside JSON) not supported	No configuration required for any data encodings including multiple nested encoding.
Active Threat Verification (DAST)	No	Yes
Other considerations	Easy to understand and visualize	Requires production traffic to train

UNDER THE COVERS

As described in the overview, Wallarm machine learning engine is roughly composed of three stages, each using different machine learning models. It relies on a unique combination of the statistical and deep learning principles including hierarchical clusterization, statistical n-gram based models, recurrent neural networks and reinforcement learning.

The detailed architecture is depicted in the following diagram:



BUSINESS LOGIC ACTION MARK-UP

In the first stage, Wallarm first parses HTTP requests and determines what application business logic is represented by each of the requests. This process is sometimes called Deep Packet Inspection (DPI) in firewall and IPS solutions, but technically we are talking about L7 request inspection, not L3 packets.

It works in the following way. Each HTTP request is parsed, serialized and stored and then preprocessed in the in-memory database which is a part of the Wallarm node. With its strong DPI facilities, Wallarm is capable of identifying and decoding all the modern web application data formats including: XML, JSON, WebSockets, Base64, GZIP, VIEWSTATE, PHP, Java serialization formats and more. Moreover, Wallarm does not require any configuration to be able to decode all the data formats and their combinations. Wallarm can easily parse even complicated nested data encodings like Base64 inside a JSON field inside an XML and do so at an incredible parsing speed.

The speed of processing is enabled by the algorithms based on statistical data profiling, which is what allows Wallarm to make decisions about the applicable data format for each HTTP request field without parsing it. As a result, parsers only run when they are needed.

It is important to note here that a request can be parsed into one or more serialized objects which happens because some encodings are mutually exclusive. For example, a request with a “Content-Type: application/form-urlencoded” may have a JSON body, meaning that it can be decoded as form data and as JSON as well. These two variants of initial raw request decoding generate two different decoded request objects. It’s important because otherwise it would be possible to bypass this detection logic (**see details**).

After the initial parsing and data decoding, all the requests are serialized to the key-value objects (including XML, JSON and other data inside). Technically this means that the raw requests are now transformed into decoded and serialized request objects. These request objects are stored in the in-memory database co-located with Wallarm Node for further processing at the next stages. It is important to note here that the store procedure is non-blocking. The request will be handled by the real-time protection logic (passed to the backend or blocked in case of attack), regardless of whether it is being committed to a local in-memory database in parallel. If the database is not available, it doesn’t affect the request processing mechanism at all.

Once the traffic is parsed, Wallarm applies machine learning to syntactically analyze the application and identify the application endpoints. First up is the request features profiling pre-calculation phase, which happens in Wallarm Node. This calculation happens asynchronously inside the in-memory database and follows the same algorithms as clusterization. During this phase Wallarm calculates correlation metrics between request objects to understand which HTTP parameters represent different application functions. The entire task of application business logic markup is split between Wallarm Node and Wallarm Cloud and the final decision is made on the cloud side. All Wallarm Nodes pre-calculate local clusters and send the results to the Wallarm Cloud for feature second-order clusterization. As a result, marked up business logic calls, called “actions” are produced.

Below is an example of how one API call is represented in Wallarm:

Request body	Business logic representation

POST /api HTTP/1.1
HOST: api.local
Content-Type: application/json
...
 	
{“method”:”login”,
“username”:”admin@local”,
“password”:”s3cr3t!”}
	Action: {
	ID => 31337,
	Conditions => [
	METHOD eq “POST”
	URI_PATH -> 0 eq “api”
	HEADER -> HOST eq “api.local”
	POST -> BODY -> JSON -> method eq “login”
	]
	}

PROFILING NORMAL BEHAVIOR

The second stage of machine learning takes the results of the first stage and the current application profile as inputs. Storing request objects at the previous phase is important because all the behavior statistical metrics are calculated there. These statistical metrics and their correlations between request objects enables detection of brute force, credential stuffing, authentication and other behavior-based attacks when it’s impossible to make a decision about blocking by analyzing only one request without user session tracking. We will explain this in detail below. Starting with version 2.6, Wallarm Node also has the ability to work without the in-memory database at all. However, taking away the database means that the solution would no longer be able to detect behavior-based attacks.

The AI engine uses another machine learning model based on characters and it’s combinations distribution functions. These character combinations, also known as n-grams or shingles (**see details**) are also initially generated by machine learning algorithm on the Cloud side to completely cover all the known attack payloads and data structures. Altogether there are over two hundred different character distribution functions used to calculate a dynamic data template for each of the request fields.

The statistical-based approach allows Wallarm to analyze each data field with one single-run operation in contrast to the regular expressions-based approach which requires re-read operations in many cases. The resulting n-gram distribution values are put into the request object in the in-memory database and serve as input into the next iteration of the Wallarm AI.

The result of this can be visualized in this way:

Request body	Data Profile generated by Wallarm

POST /api HTTP/1.1
HOST: api.local
Content-Type:application/json
...
 	
{“method”:”login”,
“username”:”admin@local”,
“password”:”s3cr3t!”}
	Hint: {
	ID => 525,
	Action => 31337,
	Type => :data_profile,
	Profile => {
	POST -> BODY -> JSON -> username = [11726.394, 737.7364, ...]
	}
	}
	
	Hint: {
	ID => 526,
	Action => 31337,
	Type => :data_profile,
	Profile => {
	POST -> BODY -> JSON -> password = [726.394, -57.7564, ...]
	}
	}

As we can see in the example, each data profile is described with a “Hint” notation and a type of “data_profile”. This proprietary Wallarm notation describes different properties of the behavior and business logic such as the data profile for each particular request field.

“Hints” are generated based on the analysis of requests over time and describe different properties of the application, allowing for better anomaly detection.

A few examples of hints:

- Binary data (hint is not to try to parse this data because it’s a binary stream)
- Data format (Base64, GZIP, XML or JSON parser required)
- Upload file format (Documents, Images, Video or other file types are normal there)

The user behavior profile is also defined by a number of “Hints” objects of different types. This approach covers behavior patterns, such as how frequently a user normally uses this application action, the sequence and order in which the actions should be called, and many other important metrics. We believe that this approach can cover any business logic of any application. It can also be easily extended by adding new types of “Hints”. User behavior hints allow Wallarm to protect against bots, application DoS attacks and other behavior-based attacks like password bruteforce, credential stuffing and so on.

ATTACK DETECTION

During the final stage of the request analysis, Wallarm applies a machine learning-based algorithm to classify previously detected anomalies by attack types and eliminate false positives.

Anomalies are identified by applying fuzzy search to a statistical character distribution model. Attack types are recognized by analyzing the current request against the ML model of the application graph. This allows Wallarm to identify a broad set of possible issues automatically without having to generate signatures manually. This approach is also more resistant to most of the WAF-bypass-techniques.

It is important that the attack type recognition applies only to the abnormal requests detected at the previous stages. This means, for example, that the SQL-related commands in legitimate control panels will not be detected. We called these cases “legal true negatives” to identify that it is normal according to application behavior to have attack-related payloads in some HTTP request fields. For example, it is normal to post some SQL-injection payloads in a security-related blogpost.

The attack classification process can be visualized in the following way:

Request body	**Data Profile generated by Wallarm**

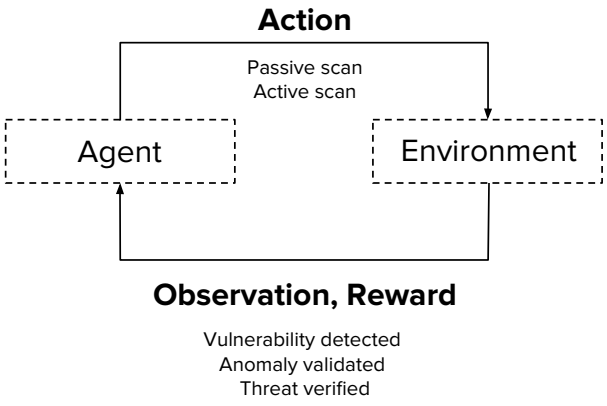
`POST /api HTTP/1.1`
`HOST: api.local`
`Content-Type:application/json`
`...`
 	
`{“method”:”login”,`
`“username”:”t@local’or 7=7--a-”,`
`“password”:”secret”}`
	`[x] Abnormal data check triggered for the application action #31337`
	
	`[x] SQL injection grammar profile stamp 2549 (logic-based injection variant) detected with a probability of 84.72%`

A detailed description of the Wallarm detection logic is available in the “Evolution of Real Time Attack Detection” whitepaper.

REINFORCEMENT LEARNING

The most differentiated part of the Wallarm's machine learning approach is reinforcement learning.

Reinforcement learning is a discipline of AI which focuses on making sequences of decisions. Within the reinforcement learning model an agent observes the environment and takes actions to optimize rewards. After every action the agent makes another observation to understand if the overall reward metric is improving. The key part of reinforcement learning is that it deals with previously unknown environments, through a learning process that usually involves lots of trial and error.



In the case of Wallarm, the application itself and its request and response traffic represent the environment. Wallarm's filtering node is the agent.

Imagine that a protected app is a resource like **stackoverflow.com** or a personal blog of a security expert. This means that some request fields like a message/blogpost text may include a payloads/exploit as a matter of course because this type of information is frequently shared there. Based on this normal application behavior, these payloads-related data would normally pass through the detection engine. But how does the AI engine know whether these payloads are dangerous for the application? This is a difficult problem because there could be a lot of requests with payloads from many different sources that look like normal traffic but in fact could be malicious.

To be able to allow certain payloads to pass as safe, the detection logic should determine beforehand that this data can not exploit any vulnerability in the protected application. This type of data may look like a payload or even be an actual payload (in a case of control panel and other legal true negatives). This task is actually equivalent to a vulnerability detection problem, because if we know that a payload affects the app at a particular endpoint, we know that there is a vulnerability there. Worth noting: the attack detection logic must have vulnerability detection capabilities to train itself. This is an example of reinforcement learning observation which affects action.

Vulnerability detection is typically a job for scanners. However, many scanners do not work well with modern applications and APIs because of the sophisticated L7 data protocols and encoding formats involved. Active scanners can only uncover detailed information about requests during the crawling phase, when they attempt to mirror a web site by recursively downloading pages and emulating browser behavior. These old-fashioned scanning methods are ineffective for single page applications and APIs because these are impossible to crawl. Conversely, Wallarm already has a full map of all the available API calls and data structures. This map is created during the profiling stage described earlier. Thus, Wallarm makes active scanning possible. Additionally, Wallarm implements a passive vulnerability detection approach. Some of the vulnerabilities can be detected by the sequence of requests and responses as well as correlation analysis of the requests' and responses' content. This allows Wallarm to detect vulnerabilities even before they are tested for by the active scanner. Even though passive scanning is important, some of the vulnerabilities that can be exploited out-of-bounds (OOB) are not detectable by passive correlation and require the active scanner. Among these are such important vulnerabilities as Remote Code Execution (RCE), Remote FIle Inclusion (RFI), Server Side Request Forgery (SSRF) and XML eXternal Entity (XXE).

Overall, it's a unique Wallarm feature to reinforce machine learning by passive and active vulnerability scanning results to deliver a very low level of false positives in attack detection and to distinguish exploitable attacks from aggressive noise. The detection logic now uses feedback from the application to be able to tune itself.

CONCLUSION

As applications and attacks grow in complexity and sophistication, traditional protection methods become more cumbersome or break altogether. Wallarm's approach is principally different and relies on AI to profile applications and to decide what's normal within the application profile.

In this paper we have looked into the inner workings of several stages of Wallarm AI both in Wallarm Cloud and locally on the Wallarm filtering node.

The combination of on-site analysis of application requests and responses with cloud-driven reinforcement learning is what allows Wallarm to be:

- Fast
- Accurate
- Comprehensive
- Auto-configurable

Automated application environment learning makes Wallarm a good fit for highly distributed infrastructures that change often. It also means that it can detect Zero-day threats in addition to threats that are known and eliminate many of false positives and false negatives by generating a smaller number of security rules that are specifically relevant to the applications under protection..

ABOUT WALLARM

Wallarm is an innovative AI startup focused on application security. With the help of machine learning, Wallarm reconstructs application context and API logic by looking at application requests and responses and uses this information to automatically create custom security protection rules for each application release, making it a great fit for CI/CD environments with frequent releases. Founded in 2013, Wallarm has already helped hundreds of SaaS and enterprise customers discover and fix critical vulnerabilities, automate web application and API runtime protection and prioritize security risks. Wallarm is a privately held company headquartered in Menlo Park, California and backed by Y-combinator, Partech Ventures, and other investors.

https://wallarm.com
155 Constitution Dr., Menlo Park, CA, 94025
(415) 940-7077