Wix Engineering · Nov 14, 2022 · 5 min read

Threat and Vulnerability Hunting with Application Server Error Logs

Introduction

When doing application security at scale, you have to make peace with the fact that some issues may as well find their way into production. While we work hard to make sure this almost never happens, we understand that it's just a fact of life that sometimes some things can slip into production without proper security controls.

Photo by Mahdi Bafande on Unsplash

That's why in addition to constantly "shifting left" our efforts and trying to secure all frameworks "by design", one of the core missions of the application security team in Wix is to monitor and assess our production environment at all times for application level issues that might put our data (or our users data) at risk.

However, while we have many security monitoring processes that focus on the runtime environment (server / operating system) or on the perimeter (WAF, HTTP access logs), we found commercial solutions tend to neglect to monitor the 'application runtime', that is the behavior, logs & metrics of the application stack.

At Wix, we added a simple, yet effective monitoring process that helps us detect application level vulnerabilities in the production environment in real time. By monitoring specific runtime exceptions (e.g. "SQL syntax error'') we are able to easily identify applications exposing ***exploitable vulnerabilities*** already running in production. This monitoring process generates effective alerts with a very minimal false positive rate.

In comparison to "shifting left", this monitoring approach can be considered as a "safety net" in which the misbehaving applications are caught in the production & testing phases at runtime, covering for the pitfalls of the traditional "prevention" AppSec methodologies.

Appsec monitoring via error logs

The basic assumption of our monitoring idea is that certain exceptions should never appear in production applications that were written securely. For example - SQL 'syntax errors' should not occur when a query is written 'properly' (i.e. using a valid parameterized query library). Such exceptions may indicate that the actual syntax of the SQL query changed due to a runtime aspect (for example user input) that was not properly handled or was unexpected. In other words, such errors occur only when using dynamic SQL queries (which are always considered a very bad practice, even if they are not directly affected by user input), and if it's indeed an input that "breaks" the syntax - the query is vulnerable to SQL Injection.

If you 'reverse' this logic, the process becomes clear - instead of searching for injections using penetration tests (and then finding evidence in the log), find errors in the logs and then reverse-engineer them to build the injection payload and find the vulnerability.

Having understood this concept with SQLi, we can extend the approach to other vulnerability classes that share similar characteristics, such as deserialization bugs, XXE, server-side template injection etc.

They all have "symptoms" capable of triggering unexpected errors which are part of the application's runtime / core dependencies. Such errors can be considered as application-level Indicators Of Compromise (IOCs).

We put our theory to the test and were amazed by the results - by defining some application-level IOC's and monitoring the behavior of all our production services we were able to identify and fix several nasty vulnerabilities across many services.

Let's break down an error containing a stack trace with an SQL Syntax Exception:

The first line contains the Exception Class & the Exception Details.
Each new line contains the following details: Full Class, Invoked Method , Class File , Line in the source code

When analyzing the stack trace for SQL injection, we're interested in finding the first **non-generic** class that leads to the exception. By **non-generic** we mean a custom class that is **unique to your codebase** (in our case - unique to Wix). It will often start with '*com.wix.**' rather than part of a generic library like '*com.mysql.**'

In the example above, the suspicious query is fired at MySqlQuestionDao.scala at line 324 in the **getQuestions** function.

This is just an example, the same logic can be applied to other runtimes (such as NodeJS, Ruby, .Net, Python etc.), and of course to other vulnerability classes.

Scaling up to a production-grade solution

One of the key engineering principles in Wix products is the minimal latency and the fast loading time of user sites. Even the slightest impact on the application latency is unacceptable, therefore we decided to avoid the potential overhead that comes

with common *application runtime protection* (RASP) solutions and to deploy the application behavioral monitoring logic out-of-band in an external environment outside the production clusters.

As you can imagine, over 3000 services running on ~20K kubernetes pods generate quite a lot of application logs, all of which are aggregated into a central log database that adds up to an average storage of 35 Terabytes a day. Therefore analyzing all the logs, all the time is extremely resource intensive.

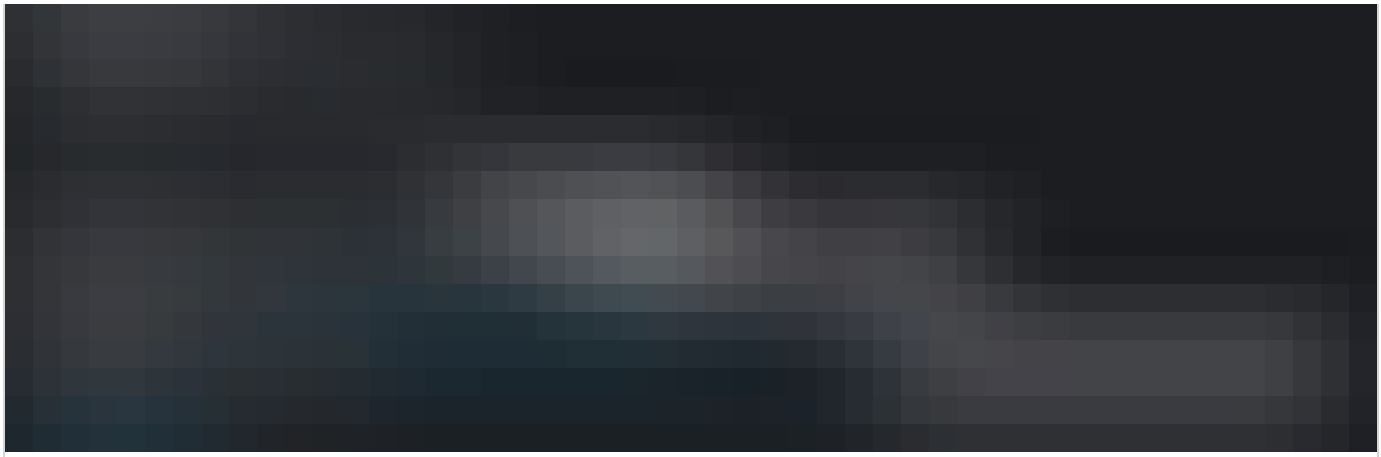
To make the problem more difficult, in addition to the main production clusters in Wix we have several “external” production environments maintained by different organizational units (such as subsidiaries, internal systems etc.). Our monitoring process provides visibility into their application stack as well; all environments are considered as first-class citizens when it comes to security monitoring.

In order to overcome the scalability & latency problems, we have set up a monitoring process based on the existing central log database. The monitoring process extracts a subset of the application server logs that contain suspicious application-level IOCs, extracts the relevant metadata and alerts via Slack when a log matches the rule-set we defined. These alerts are then analyzed manually by an application security specialist and a bug is born.

Having used and matured this process for more than a year, we reached a stable and reliable set of detection rules that ensure our false positive rate is quite low. It varies depending on the vulnerability class and the accuracy of the detection rule. For XXE & SSTI we experienced a detection rate of 100%, meaning that there were zero false positives.

For SQL Injection the case is a bit different, 26% of the alerts resulted in a real vulnerability, while the other 74% are due to edge cases in the queries (either typos or plain bugs such as an empty *in()* clause)

An example for such as an alert:



Conclusion and future work

Identifying and preventing application vulnerabilities is a multi-layer process. A good application security program aims to eliminate vulnerabilities in the development process, but it also must accept that bugs “infiltrate” into the production environment.

Cyber Security

This very naïve and simple monitoring process turned out to be super-effective at scale, the raw logic behind the monitoring rules helped us to remedy many vulnerabilities already present in production.

Due to the scaling & configuration issues that we experienced with our current SIEM solution, we built an internal log processing pipeline that allows us to seamlessly apply our detection rules to our server logs, regardless of the scale or number of log sources.

Recent Posts See All

Improving Our Security Posture with Ethical Hackers

5

Wix Continuous Security Posture Management- Pt.2

4

At Wix Engineering we develop some of the most innovative cloud-based web applications that influence our +200 million users worldwide.

Have any questions?
Email: wixeng@wix.com

Home
Life at Wix
Blog
Open Source
Podcast
Newsletter

Meetup @TLV
Meetup @Vilnius
Meetup @Kyiv
Meetup @Dnipro
Wix Jobs

Twitter
YouTube
Facebook
Github
Linkedin
Wix Kickstart
Wix Enter

- Subscribe to our monthly newsletter
- Subscribe to our YouTube channel
- Follow our Medium publication
- Listen to our podcast on Apple, Spotify or Google