API and Modern Application Security

A Traceable Whitepaper

Outline

I. Introduction
II. Traditional vs Modern Application Architecture
    A. Traditional
    B. Modern: Microservices / APIs / Cloud
III. Overview of Traditional Application Security
IV. Requirements for Modern Application Security
V. Conclusion

[I] Introduction

Over the last 15 years, businesses have been rapidly adopting modern application architectures and operational practices for developing and managing software. The term cloud-native has been widely adopted in the industry as an umbrella category that encapsulates many of these new technologies and processes. While not exhaustive, the most significant aspects of cloud-native include:

- Cloud infrastructure (e.g. Amazon Web Services, Microsoft Azure, Google Public Cloud)
- Containerization (e.g. Docker)
- Automation of deployment and scaling (e.g. Kubernetes)
- Microservices
- APIs
- DevOps
- Continuous integration, deployment, and delivery

With change comes great opportunity not just for your business but also for cybercriminals. Security technology and operational practices have not been able to keep pace with these changes, leaving applications and businesses vulnerable to attack. Attackers have innovated and found new methods to infiltrate applications to take over accounts, access private data, and cause businesses to lose the trust of their customers.

The focus of this paper is to highlight how application security systems and processes must evolve to protect and monitor production cloud-native applications. Application security testing, code-analysis, and other pre-production techniques are not discussed. To fully understand modern application security requirements, we'll begin by discussing how application architectures have evolved and highlight the key differences between traditional 3-tier and modern microservices architectures. We'll then review the state of traditional application security, followed by a thorough analysis of modern application security requirements.

[II] Traditional vs. Modern Applications

Before we discuss the requirements for securing modern applications, we first need to understand what constitutes a modern application and how traditional architectures have evolved to become 'modern'.

Traditional Application Overview

Let's start with a quick refresher on the "traditional" application architecture so we're all on the same page. There was hardware. Servers with operating systems and software running on them and networking gear and storage. This was all hosted in a private or co-located data center. Applications generally consisted of:

1. **Clients** were web browsers presenting the user interface through HTML with modest use of JavaScript.
2. **Application and web servers**, typically behind a load balancer, received client requests, ran server-side code, interacted with a relational database (hopefully running on a different server), and returned the HTML to the client.
3. **Database servers** stored the application data and responded to queries from the application server.

Traditional applications were monolithic, which simply means that all of the server-side code was organized in a single, self-contained application. A firewall was deployed in front of the web server. The client was almost always a web browser. SOAP was the standard protocol for exchanging data between applications.

Modern Application Overview

Fast forward to today and we've got, what at present time is referred to as 'modern', cloud-native application architectures. As the term cloud-native would suggest, modern applications run in the cloud. Firewalls still exist and virtual private clouds provide a level of isolation to make running applications in multi-tenant public clouds safer.

Microservices have made monolithic applications obsolete. A modern application is the combination of multiple, often hundreds or thousands, of microservices. Each microservice typically has a very specific and narrowly scoped job. Microservices access data from various datastores and communicate with other internal microservices and 3rd party services using REST APIs including JSON, GraphQL, etc.

Microservices are deployed in software packages called containers that, using operating system-level virtualization, include all of the necessary software and configuration files needed to run the microservice application code. Kubernetes and other container-orchestration systems are used to manage and automate the deployment and scaling of containers.

Browsers are still a popular client but modern applications also serve IOT, native apps, other internal microservices, and 3rd party services.

- **The Cloud:** Modern applications live “in the cloud”. Cloud providers offer different levels of abstractions (IaaS, PaaS, SaaS) but long story short, these apps run in the cloud.
- **Microservices:** Large applications are broken down into smaller components or services.
- **Business Logic is Highly Distributed:** Applications are decentralized with business logic spread across services.
- **Deployment Orchestration:** Containers and Kubernetes (K8S) make it easier to connect and manage large numbers of microservices.
- **Continuous Integration / Continuous Delivery (CI/CD):** Software teams develop features using agile methodologies and continuously deliver new code to production.
- **DevOps:** Developers and technical operations work together to manage and operate the application and development toolchain.
- **Data Volumes:** There’s no shortage of useful data these days and it comes in many shapes and sizes that require more databases than your standard RDBMS (sorry Oracle).
- **Cloud / 3rd Party Services:** Not all application functionality needs to be created in-house. Most apps integrate 3rd party services, using APIs, to beef up application functionality without reinventing the wheel.

Clients have differences too!

- **Devices became more powerful:** When I was a kid, iPhones only had 1 camera! What a time.
- **Client Varietals:** Today there are more types of clients - from traditional web browsers to native mobile apps to (purposely) exposing your own APIs to other developers.
- **Client Muscle:** Modern frontend frameworks (Angular, React.js) allow developers to write complex logic that runs efficiently in the client. Clients now render visual components that used to render server-side.
- **More Calls, More Parameters:** Clients maintain user state locally and trigger API calls when data or an action is needed. This is done at the component level vs the page, so the number of calls is higher and the number of parameters sent to the server is higher as well.

[III] Overview of Traditional Application Security

Securing traditional applications was, in comparison to modern applications, relatively straightforward considering the more simplistic monolithic architecture. That doesn't mean preventing application attacks was easy or bulletproof. What it does mean is that security teams were able to understand the application components, how they interacted, and the expected ingress and egress traffic patterns.

Security teams benefited greatly from the simplicity of the traditional application architecture for a multitude of reasons. First, the clients were dumb. It was difficult to garner a rich understanding of the application's business logic from looking at the HTML code parsed by your browser. Hackers could learn a little from the form input fields and URL parameters, but typically not enough to reconstruct the business logic. Second, there were fewer unique parts. Users accessed applications almost exclusively through a web browser; native apps and IOT were not yet mainstream. The monolithic application consolidated the code and business logic into a single application; microservices were not yet widely adopted..

WAFs analyzed the traffic between the client and the application, blocking client requests as dictated by a set of manually created rules. As you recall, clients were browsers displaying HTML and executing limited JavaScript (compared to today) and application servers simply returned HTML from the application server to the browser. WAFs, therefore, didn't have access to application code or business logic, thus limiting the number of signals available to the security pros to distinguish between safe and malicious users. Security teams had to choose between high false-positive rates resulting in unhappy customers and high false-negative rates resulting in uhappy Chief Security Officers.

Hackers were also limited by the simplicity of the traditional architecture, but managed to find no shortage of opportunities to attack. Flaws in application design such as improper handling of SQL injection or cross-site scripting (XSS) and poorly implemented authentication, authorization, or session management systems opened the doors for hackers to succeed in their mission. The application's attack surface also increased if data was exchanged with 3rd parties. By exposing the application's business logic, these primitive web services, like microservices today, gave hackers more information to work with. Additionally, there are vulnerabilities in authentication and authorization as 3rd parties required a different, more complex set of permissions and access requirements.

[IV] Requirements for Modern Application Security

In the previous chapters, we developed an understanding of how application architectures have evolved, compared the differences between traditional and modern architectures, and summarized traditional application security. In this chapter, we'll describe the requirements for securing modern, cloud-native applications, including the APIs that interface with client applications, other internal APIs, and 3rd party services.

One of the biggest challenges in securing cloud-native applications and their APIs is understanding application context at runtime to determine if client requests are legitimate. Application context includes all of the activity that occurs when a client session makes requests to the application. With traditional applications, the business logic was well contained in the backend application code. With cloud-native applications and APIs, the client connects to APIs and makes requests to backend and 3rd party services to implement the business logic.

Take a typical consumer web application as an example. A user wanting to view their stored billing address would visit the application's "Manage Account" page. This page includes, among other things, a module that displays the user's billing address. In a traditional application, the client requests the "Manage Account" page from the application server. The backend code executes the business logic and constructs the page, including the data for the billing address module, and returns it to the browser as a full HTML page. Therefore, all of the business logic is hidden from the client.

Conversely, with a cloud-native application, the client calls all of the individual backend application services, including the service responsible for retrieving billing address data. Each service returns the requested data in JSON format back to the client. The client is then responsible for taking in the JSON and stitching together the components in the resulting page or screen. ***Understanding application context is a fundamental security requirement for cloud-native applications.*** If the pattern of client API requests, subsequent internal API requests, and/or calls to 3rd party services is inconsistent with normal application behavior, then the user may be malicious.

Monitoring and reacting to application context requires that your application's security system can understand expected, or normal, application behavior. Knowing what is normal is not a trivial task. Most cloud-native applications have many backend services and APIs, in some cases hundreds or even thousands. There are usually multiple types of clients, browsers, mobile apps, admin tools, other services, etc., that are customers of the backend services, each with different application flows. Further, teams building and operating cloud-native applications are typically

practicing agile development and continuous delivery, making normal application behavior fluid and dynamic. Expected application behavior evolves as rapidly as the application evolves with code changes, updated APIs, and new services being added to the application. ***These factors dictate that the second fundamental security requirement for modern applications is the ability to continuously learn the application's expected behavior across client types and as application components change.*** The system must then be able to compare application behavior in real-time with the expected behavior to detect malicious activity. The application security system, therefore, must include a method for collecting real-user application activity and have an artificial intelligence component that is capable of dynamically learning from this data and distinguishing between expected and abnormal application activity.

The OWASP API Top Ten documents the most common attack vectors facing APIs today, and includes:

1. Broken Object Level Authorization
2. Broken User Authentication
3. Excessive Data Exposure
4. Lack of Resources & Rate Limiting
5. Broken Function Level Authorization
6. Mass Assignment
7. Security Misconfiguration
8. Injection
9. Improper Assets Management
10. Insufficient Logging & Monitoring

While we won't go into the details of these attack vectors in this paper, it is critical to understand that they all take advantage of either flaws in the application code, application configuration, or miscommunication or lack of communication between development and security teams. ***The next requirement for securing modern applications, as you may have guessed, is that these two teams work closely together to protect applications from attack and to respond if and when attacks do occur.*** DevSecOps is an industry movement that focuses on addressing this need. Whether this movement will evolve, similar to DevOps, into a formal job function is yet to be determined. Regardless, it is imperative that these teams work together, use common tools, have clear communication protocols, and share in the responsibility of keeping applications secure. One of the easiest and most successful ways of accomplishing this tight coordination is to leverage tools that provide capabilities for both security and software development professionals. While not an exhaustive list, these tools should include:

- Application topology maps that automatically map production components from the client all the way through to individual microservices, data stores, and even 3rd party services
- API specifications produced from live production traffic to understand actual usage compared to documented specifications
- The ability to track how data flows across various application components
- Understand application users and roles and the different API activity of users with different roles and permissions.
- Consolidated forensic data from all application components with drill-down reporting
- Integrations with cloud-native infrastructure, such as Kubernetes, Envoy, Docker, etc. to orchestrate the security solution and automatically block malicious users
- Lightweight agents or other data collection mechanisms to collect data from production applications without introducing performance overhead

The final, and admittedly not new, requirement is high accuracy in detecting malicious activity. While this isn't a new requirement, it is a critical requirement and requires a brief overview. Accuracy is typically measured in false-positive rates, indicating the percentage of valid users that have been treated as malicious users incorrectly. Successful application security has to find the right balance between preventing malicious activity and delivering a high quality user experience. Securing an application and having zero false-positives would be easy if every request was reviewed manually. Of course, security teams don't have the time and more importantly our legitimate users don't have the patience for that level of inspection. Application security monitoring systems aim to automate this review and can even automate blocking malicious users. Traditional security systems depend on manually derived rules that are either too stringent and cause a bad user experience or are too lenient and let malicious activity slip past their defenses. Modern security solutions, likely using artificial intelligence techniques, must be able to effectively prevent malicious activity without derailing user experience and achieve a near zero false-positive rate.

To summarize, the critical requirements of modern application security include the ability to:

- Continuously and automatically learn normal application behavior
- Track application context in real-time and distinguish legitimate and malicious activity
- Automatically adapt to application changes, including code, API, and configuration changes
- Bring software development and security teams together to protect applications and coordinate responses to attacks

- Accurately protect applications without jeopardizing actual user experience

[V] Why Traceable?

Traceable(™) answers most of the requirements for a modern API and Application security tool.

It has deployment options and data collection vehicles for anything presenting application logic on layer 7, from legacy monolithic PHP applications to Kubernetes instrumented with Istio service mesh. It is optimized for DevSecOps and provides a platform for security pros to find anomalies and teach developers how to make use of the tool themselves. It has a rich set of data and a powerful ML platform to learn the developer's intent, discover the APIs and other assets, and adapt to the environment - all while improving the observability for human decision making. It monitors both north-south and east-west APIs to provide as much zero-trust information as the team can handle. It monitors the application logic, user, and data behavior in context to become one of the best and most comprehensive application security monitoring tools.