pactera EDGE

Coding Best practices for building Intelligent B2C Cloud Native Applications

September 4, 2020

By Ovidiu Mura
Software Engineer at PacteraEdge

ovidiu.mura@pacteraedge.com

In a highly interconnected world where technology changes rapidly data privacy and securing the applications from bad actors is essential. The B2C Cloud Native applications are opened to the public internet to be able to reach all possible customers but this comes with risks. The B2C Cloud Native applications can have vulnerabilities in design, source code, network security policies, or the vulnerability which comes with the deployment on the Cloud platforms. Bad actors can exploit vulnerabilities which can be exposed by a negligent development process, but if the software development engineers and security engineers work together and use the best coding practices in their software development process, vulnerabilities can be mitigated completely.

B2C (Business-to-Consumer) applications are applications which have a design model where the products or services move directly from a business to the customer who purchases them for personal use via the internet. Cloud-Native is an approach that takes advantage of the cloud computing delivery model for building and running applications. Cloud Native Applications are applications which are built from ground up, and are optimized for cloud scale and performance. They are built based on microservices architectures, use managed services, and take advantage of continuous delivery to achieve reliability and faster time to market. Cloud-native computing takes advantage of many modern techniques, including PaaS, multi-cloud, microservices, agile methodology, containers, CI/CD, and devops. [1]

Cloud-Native Applications run on cloud-based compute solutions so that the developers don't worry about the infrastructure details; the resources can be easily scaled up or scaled out as the application usage grows. Azure provides Infrastructure as a Service (IaaS) which gives the developer full control over the application hosting - customer-managed products such as Virtual Machines/Virtual Networks. Azure provides Platform as a Service (PaaS) which gives to developers fully managed services needed to power your applications such as platform-managed solution Service Fabric and App Service. Azure provides serverless hosting where all the developer needs to do is only write code; serverless solutions with Azure are Functions. [2] The other big Cloud service providers such as GCP (Google Cloud Platform), AWS (Amazon Web Services) deliver similar solutions with some differences. The software developer engineers need to define the Cloud Native Application architecture/functional requirements and then look for the best Cloud service provider solutions offered by different vendors.

The key characteristics of Cloud-Native applications are multiple services, elasticity, resiliency, and composability. Each application can be separated into loose coupled services called Microservices (multiple services) and they communicate to each other through a controller as a single application which interacts with the end user. The elasticity is the property of the Cloud service to scale vertically (up/down) and horizontally (in/out) the application deployed. The cloud platform services are dynamically managed and they guarantee an efficient usage of the resources per customer demand. The services which compose the Cloud-Native application are resilient because they still run properly when the Cloud platform services experience outages or failures. APIs can define composable behaviour for each microservice to be consumed by the other applications.

The Cloud-Native applications can be deployed fast and easy in the cloud using CI (Continuous Integration) and CD (Continuous Delivery/ Continuous Deployment) and the microservice and the applications are auto-scalable independently and automatically. The microservices which compose the Cloud-Native application provides an architecture for the application to be built independently of each

[1] Horizontally vs Vertically Scalling https://www.linkedin.com/pulse/scalability-horizontal-vs-vertical-scaling-scale-outin-khaja-shaik-/

other in such a way that they are updated, managed and deployed individually. The cost to build these applications is efficient and cheaper than Non-Cloud-Native applications because the account is charged only for what is used and the resources can be created and deleted on demand without leaving any overhead.

The APIs and other Cloud services can be managed and developed using platform specific SDKs and other additional SDKs such as .NET, Node.js, Java, PHP, Python, Ruby, Go and others. Many programming languages provided for the Cloud-Native application to be built opens more vulnerabilities in the application development and easier the functionality to be broken. Next, I will present a few best coding practices for building B2C (Business-to-Consumer) Cloud-Native applications. Software Developer Engineers and Security Engineers must work together to overcome the challenges which come with building B2C (Business-to-Consumer) Cloud-Native applications. The Cloud-Native applications are dependent on the internet and computer networks which makes the process of building reliable quality applications complex.

The broken object level authorization vulnerability is an access control mechanism that is implemented to ensure that only the desired user can access objects in the given environment. The APIs endpoints which receive object ID and perform any type of action on that object must implement object level authorization checks which validate that the logged-in user has proper permission access to perform the requested action on the requested object. If this mechanism is not implemented properly it will lead to unauthorized information disclosure, modification or destruction of all data. For example, an malicious attack scenario would be in the HTTP PATCH request when the attacker modifies a custom HTTP request header X-User-Id: 2 to X-User-Id: 1. The attacker will receive a successful HTTP response and will be able to modify the User-Id 1 account data if it exists.

The developers implement proper authorization mechanisms using user policies and hierarchy to prevent broken object level authorization vulnerability. Developers use authorization mechanisms to check logged-in users and their access is allowed to perform the requested action on the object. Developers can prevent by using random and unpredictable values as GUIDs for object or records IDs. Also, developers must

write and execute tests which validate the authorization mechanisms before the artifacts are deployed and they deploy only the quality code which passes these tests into the production.

Another vulnerability in the code of Cloud-Native applications can be broken user authentication which can appear in the APIs endpoints and flows unprotected such as “Forgot password / reset password”. They need to be implemented as authentication mechanisms. The endpoints and the flows can be attacked if they permit credential stuffing [8] - attacker attacks with a list of valid users and passwords. The attacker also can perform a brute force attack on the same user account if there is no captcha or account lockout implementation. An attack can be successful on weak passwords, or if the sensitive authentication details (auth tokens and passwords) are sent in the URL, or the authenticity of tokens are not validated. The APIs are also vulnerable if they accept JWT tokens which are unsigned or weakly signed, or the tokens are not validated for the expiration date. The passwords must not be used in plain text and they must be strongly encrypted with powerful encryption keys. A well known attack on this type of vulnerability is the credential stuffing which uses a list of known username and passwords for which the attacker uses the application as a password tester to determine if the credentials are valid.

To prevent broken user authentication vulnerability, the developers must know all the possible flows to authenticate to the API and carefully examine peers to verify if the possible flows to authenticate are correct and they are implemented as they intended. The authentication APIs, tooken generation, password storage flows must be implemented by specifications of the corresponding standard. It is recommended to implement multi-factor authentication if the system allows, and strict anti brute force mechanisms to mitigate credential stuffing, dictionary attacks, and brute force attacks on the authentication endpoints. The brute force attack can also be prevented by implementing account lockout and captcha mechanisms for more vulnerable users and the API keys should be used strictly for client app and project authentication access, Figure 1.

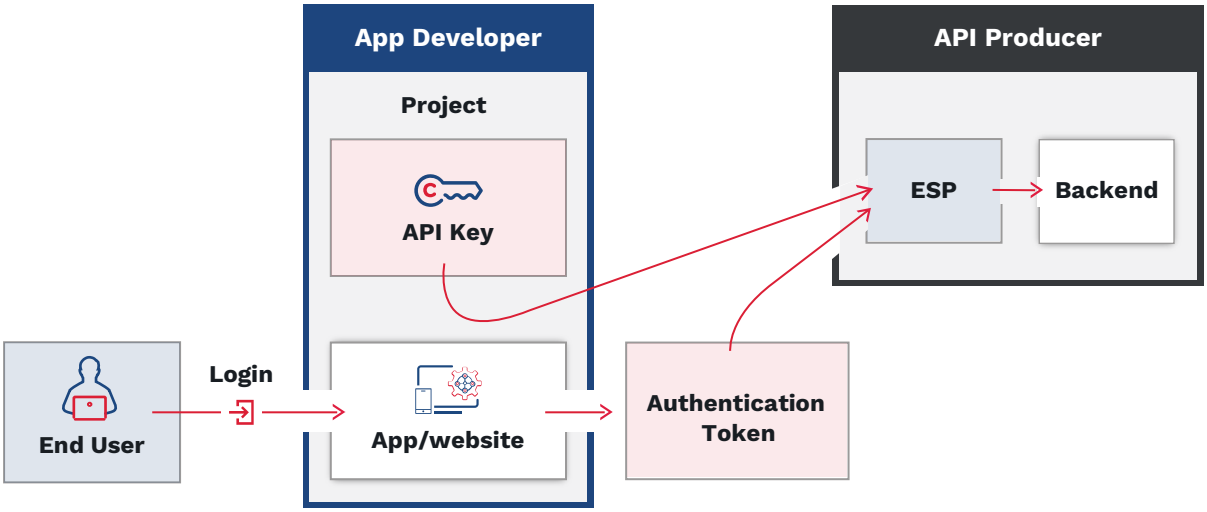

Figure 1 - Authentication tokens identify a user that is using the app or the site [9].

Injection is another API vulnerability and it occurs when Client supplied data is not validated, filtered, or sanitized. The client-supplied data is directly used or concatenated to queries, commands, XML parser, ORM/ODM. The injection vulnerability also occurs when the data is coming from the external systems and is not validated, filtered, or sanitized. An example of an injection scenario is when a booking application with basic CRUD functionality has one of the delete booking request's parameters changed, Figure 2.

```
router.delete('/bookings', async function (req, res, next) {
  try {
    const deletedBooking = await Bookings.findOneAndRemove({_id' : req.query.bookingId});
    res.status(200);
  } catch (err) {
    res.status(400).json({
      error: 'Unexpected error occured while processing a request'
    });
  }
});
```

Figure 2 - The API server delete request for the Injection vulnerability scenario

The changed delete request's parameter is bookingId from the query string; the injection changed parameter bookingId delete another user's booking object.

To prevent the injection vulnerability, the data must be separated from the commands and queries. The data must be validated using a single, trustworthy, and actively maintained library. Data coming from client or integrated systems must be validated, filtered and sanitized properly. The developer should use safe APIs that provide parameterized interfaces and the number of returned records must be limited to a small number to prevent the mass disclosure in case of injection. Also, the incoming data must be validated with filters to allow only valid values for each input parameter and define data types and strict patterns for all string parameters.

Next, I want to talk about the vulnerability on the server-side called server-side request forgery (SSRF). The attacker makes HTTP requests from the server-side to an arbitrary domain of the attacker's choosing. When the attacker gains access to the server-side, it can make requests back to itself or to a different web-based service on the organization's infrastructure or third-party systems. SSRF attacks perform unauthorized actions or access to data within the organization, either in the vulnerable application itself or on other back-end systems that the application can communicate with and the attacker can execute arbitrary commands. SSRF attacks often exploit trust relationships to escalate an attack from unauthorized actions in relation to the server itself or in relation to other back-end systems within the same organization.

I will talk about the SSRF attacks against the server itself scenario in which the attacker induces the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. The supplied URL which has the hostname 127.0.0.1 (localhost). The trust relationship where the requests originated from a local server are processed differently than the other requests makes the SSRF critical vulnerability. For example, let's present a scenario in which we have a shopping application that the user can view if an item is in stock in a particular store. The application queries back-end REST APIs for a given product and the store. The implementation passes the url to the corresponding back-end API endpoint via a front-end HTTP request, Figure 3.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

Figure 3 - Back-end endpoint via a front-end HTTP request

The server will make the request of the given URL, retrieve the stock status and then return this response to the user. The attacker can modify the request by setting a URL relative to the local server, Figure 4.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

Figure 4 - An attacker can modify the request to /admin url

The server will fetch the content of the /admin URL and return the response to the user or the attacker can also access the /admin URL directly which is available usually only for the admin users. When the attacker visits th url directly, it will not see the /admin content, but when the request to the /admin url comes from the local machine itself, the normal access controls are bypassed and the application allow full access to the admin functionality since the request appears to originate from the trusted location. The reasons for this type of vulnerability can be that the access control check might be implemented in a different component that sits in front of the application server and when a connection is made back to the server itself, the check is bypassed. Another reason for this type of vulnerability is that the app allows admin access without logging in to the users coming from a local server, for disaster recovery purposes, considering these users fully trusted. The admin instance on the local server can be listed on a different port number than the main app, and it might not be reachable directly by users.

I will present the SSRF attacks against other back-end systems. This type of vulnerability occurs on the server-side when the application server is able to interact with other back-end systems that are not accessed directly by the users. These back-end systems usually are not protected by the network topology, and they have sensitive functionality with weaker security - that can be accessed without authentication by anyone who interacts with these systems. For example, let's assume that an administrative interface at the back-end URL https://192.168.0.68/admin can be exploited with SSRF vulnerability to access this interface by submitting the request from Figure 5.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
stockApi=http://192.168.0.68/admin
```

Figure 5 - SSRF vulnerability to access the administrative interface

SSRF vulnerabilities can be prevented with defenses intended to prevent malicious exploitation but sometimes these defenses can be overcomed. For example, SSRF uses backlist-based input filters to block input containing 127.0.0.1, /admin and localhost. These filters can be bypassed using alternative IP representation of 127.0.0.1, such as 2130706433, 017700000001, or 127.1. The attacker can register his own domain name that resolves to 127.0.0.1 using Burp Collaborator , or obfuscating blocked strings using URL encoding or case variation. The developer can prevent these attacks by writing code to detect the attacker's registered own domain name or obfuscating blocked strings.

Another method to protect SSRF (Server-Side Request Forgery) is using whitelist-based input filters. The application input filter only allows input that matches, begins with, or contains a whitelist of permitted values. The filter can be attacked by exploiting inconsistencies in url parsing which are failed to be noticed, such as embedding credentials in a url before the hostname, using the @ character . The # character can be used to indicate a url fragment . The DNS naming hierarchy can be

replaced a fully-qualified DNS name that the attacker can control . The url can be encoded to confuse the url parsing code section, this is very useful when the code that implements the filter handles url encoded characters differently than the code that performs the back-end HTTP request.

SSRF filters-based defenses can be overcome by open redirection techniques. For example, the user submitted url is validated to prevent malicious exploitation of the SSRF behaviour. The allowed url contains an open redirection vulnerability such as if the API used to make the back-end HTTP request supports redirections, the attacker can construct a url that satisfies the filter and results in a redirected request to the desired back-end target, Figure 6.

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://weliketoshop.net/product/nextProduct?currentProductId=6&path=http://192.168.0.68/admin
```

Figure 6 - Open redirection vulnerability to bypass the URL filter

We can see an example of open redirection vulnerability to bypass the url filter to exploit the SSRF. The application will validate the supplied stockAPI url is on the allowed domain (yes, it is in the same domain), then the application requests the supplied url, which triggers the open redirection. The application will redirect successfully and make the request to the malicious url. The developer must validate the supplied url together with the supplied stockAPI url to prevent attacks using open redirection techniques.

[2] https://portswigger.net/burp/documentation/collaborator

[3] https://expected-host@evil-host

[4] https://evil-host#expected-host

[5] https://expected-host.evil-host

[6] /product/nextProduct?currentProductId=6

[7] http://evil-user.net

When the application can be manipulated to issue a back-end HTTP request to a given url and the response is not returned in the front-end of the application, this type of vulnerability is called blind SSRF. They are harder to exploit but they are very powerful if they are successfully executed on the remote servers or other back-end components with elevated privileges. The server-side request forgery vulnerabilities are pretty easy to find because the application normal traffic contains request parameters containing full urls, but other types of server-side request forgery are harder to identify if the full urls are not present in the request's parameters. The request parameters have sometimes only hostnames or part of a url path, then the submitted value is concatenated to the server-side value forming a full target url. The values are identified as potential attacks if they are hostnames or some form of url paths.

Blind SSRF vulnerability can be prevented by response handling technique. The response handling technique limits the content allowed in the response body for preventing leakage of data and checks the response before sending it to the bad actors and filters out data not expected by the server. Another way to prevent blind SSRF vulnerabilities is by disabling unused URL schemas such as "ftp://" or "file://". Whitelists and DNS resolution can also be used very effectively to prevent SSRF attacks by whitelisting only the required DNS names or IP addresses that the application uses.

Applications also transmit XML data in structural formats with the specification URLs included, then they are passed to the data parser for processing the request. The data is transferred from the client to server and when the application accepts/parses data in XML format, it can be vulnerable to XML external entity injection combined with the server-side request forgery vulnerability.

XML external entity injection attacks can be prevented by disabling the corresponding features (XXE). Application's XML parsing library can have dangerous XML features that the application doesn't need.

Other forms of blind SSRF vulnerabilities are performed using Referer header. Applications sometimes use server-side analytics software that tracks visitors which

logs the Referer header in requests for tracking incoming links. The analytics software will visit any third-party urls that are found in the Referer header for scanning and analyzing the content of referring sites and the text used in the incoming links. The attacker can use the Referer header to exploit SSRF vulnerabilities.

To prevent SSRF Referer header attacks the developer can implement checks using whitelists and allow only headers with Referer contained in the whitelists. Also the developer must validate the Referer header against META tag attacks, adding subdomains of the main request domain attacks, or validate against attacks which place the vulnerable domain elsewhere in the URL.

Over seventy percent of the business of the world operate fully or in part on the cloud stated in the CSA report. Ninety percent of organizations are moderately or very concerned about public cloud security. These concerns are the result of code vulnerabilities, hijacked accounts, malicious insiders, and full-scale data breaches, which occur in the cloud in the last years. I will mention a few. Data breaches occur on cloud and the security measure to protect data are very low according to a Ponemon Institute study . Attackers hijack accounts using user login information to remotely access sensitive data stored on the cloud and they can falsify and manipulate information through hijacked credentials. Attackers from inside the organizations can misuse their authorized access in scenario such as misuse of information through malicious intent, accidents or malware . Cloud services are vulnerable for malware injections (scripts or code embedded) running as SaaS on cloud servers . Cloud services can be abused by storing

---

[8] SSRF Remediations, https://medium.com/cybersecurityservices/server-side-request-forgery-aashna-jain-bcc42aea0479

[9] XXE, https://portswigger.net/web-security/xxe

[10] According to Cloud Security Alliance (CSA), https://www.imperva.com/blog/top-10-cloud-security-concerns

[11] Man In Cloud Attack, http://go.netskope.com/rs/netskope/images/Ponemon-DataBreach-CloudMultiplierEffect-June2014.pdf

[12] Inside Track on Insider Threats, https://www.imperva.com/resources/resource-library/white-papers/an-inside-track-on-insider-threats/

[13] Security Threats On Cloud Computing Vulnerabilities, http://airccse.org/journal/jcsit/5313ijcsit06.pdf

huge amounts of data easily with the cloud storage services . Cloud APIs can be a threat to cloud security because the security risks grow proportional to infrastructure of APIs size. The communication between applications exposes the APIs to vulnerabilities increasing the security risks. Denial of Service (DoS) is another form of attacks which attempt to make the B2C Cloud-Native applications and servers unavailable to their customers. Another source of vulnerabilities in the cloud is the insufficient due diligence when an organization migrates their applications to Cloud-Native applications; it is common to companies with data under regulatory laws like PII, PCI, PHI, and FERPA. Providers such as Box, Dropbox, Microsoft, and Google require the client to take preventative actions to protect their data while the providers do have standardized procedures to secure their side. Share vulnerabilities occur when the cloud security is shared between the provider and the client. Losing data on cloud service can also be through malicious attacks, natural disaster, or a data wipe by the service provider. Securing data means carefully reviewing provider back up procedures as they relate to physical storage locations, physical access, and physical disasters.

The common vulnerabilities presented above are a serious threat and a strong reason for the development teams to build a cloud strategy to protect the Cloud-Native Applications. I talked about the B2C design model and the benefits of building Cloud-Native applications and how the cloud providers deliver services to the consumers. Then I presented code best practices by discussing vulnerabilities of the Cloud-Native applications implementation and how the attackers plan to attack the systems. Next, I discussed in detail the necessary steps required to be taken to secure the applications from the presented vulnerabilities. Finally, I shared report results of studies conducted to measure the cloud security current issues which shows that B2C Cloud-Native applications have a low security overall. To increase the security in the cloud, software development engineers and the security engineers must work together, and split responsibilities and implement coding best practices in the applications. Let’s build Intelligent B2C Cloud Native Applications!

---

[14] The Notorious Nine
https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf

[15] DoS, https://www.imperva.com/learn/ddos/denial-of-service/

ABOUT THE AUTHOR

Ovidiu is a Software Engineer at PacteraEdge. He has a Graduate MS degree in Computer Science from Portland State University and a Graduate certification degree in Cybersecurity from Portland State University. He also has over ten years experience working in various technology domains such as financial delivery systems, healthcare integrated systems, eCommerce systems, automation, cybersecurity, artificial intelligence, machine learning and algorithms.

REFERENCES

[1] Andy Patrizio. (June 14, 2018). What is cloud-native? The modern way to develop software. InfoWorld. https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html

[2] Microsoft Docs. (November 18, 2019). Get Started guide for Azure developers. https://docs.microsoft.com/en-us/azure/guides/developer/azure-developer-guide

[3] Saviant Intelligent Solutions. (2020). Everything you need to know about Cloud-Native Applications. Saviantconsulting.com. https://www.saviantconsulting.com/blog/everything-about-cloud-native-applications.aspx

[4] Anubhav Dwivedi. (2020). Saviant intelligent Solutions. 3 Reasons Why Developing Cloud-Native Applications are Worth the Time and Money. https://www.saviantconsulting.com/blog/3-reasons-why-developing-cloud-native-applications-is-worth-the-time-and-money.aspx

[5] Paulo A Silva. (Dec 26, 2019). OWASP API Security Top 10. OWASP. https://github.com/OWASP/API-Security

[6] DevSpecOps.org. (2012-2015). Manifesto. https://www.devsecops.org/

[7] OWASP Cheat Sheet Series. (2020). Authentication Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[8] CloudFlare. (2020). What is Credential Stuffing? https://www.cloudflare.com/learning/bots/what-is-credential-stuffing/

[9] Google Cloud. (2020). Why and when to use API keys. https://cloud.google.com/endpoints/docs/openapi/when-why-api-key

[10] PortsSwigger. (2020). Server-Side Request Forgery (SSRF). (2020). https://portswigger.net/web-security/ssrf

[11] Joy Ma. (December 14, 2015). Top 10 Security Concerns for Cloud-Based Services. Imperva. https://www.imperva.com/blog/top-10-cloud-security-concerns/

[12] OWASP. (2020). OWASP Cheat Sheet Series.CheatSheetSeries.owasp.org. https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html