APIGEE Scan Report

Project Name	APIGEE
Scan Start	Tuesday, November 10, 2020 11:48:30 AM
Preset	Checkmarx Default
Scan Time	00h:01m:09s
Lines Of Code Scanned	2163
Files Scanned	20
Report Creation Time	Tuesday, November 10, 2020 11:50:05 AM
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14
Team	APIGEE
Checkmarx Version	9.0.0.40085 HF12
Scan Type	Full
Source Origin	GIT
Branch	/refs/heads/jehnsen
Density	6/1000 (Vulnerabilities/LOC)
Visibility	Public

Filter Settings

Severity

Included: High, Medium, Low, Information

Excluded: None

Result State

Included: Confirmed, Not Exploitable, To Verify, Urgent, Proposed Not Exploitable

Excluded: None

Assigned to

Included: All

Categories

Included:

Uncategorized	All
Custom	All
PCI DSS v3.2	All
OWASP Top 10 2013	All
FISMA 2014	All
NIST SP 800-53	All
OWASP Top 10 2017	All
OWASP Mobile Top 10 2016	All
OWASP Top 10 API	All

Excluded:

Uncategorized	None
Custom	None
PCI DSS v3.2	None

OWASP Top 10 2013	None
FISMA 2014	None
NIST SP 800-53	None
OWASP Top 10 2017	None
OWASP Mobile Top 10 2016	None
OWASP Top 10 API	None

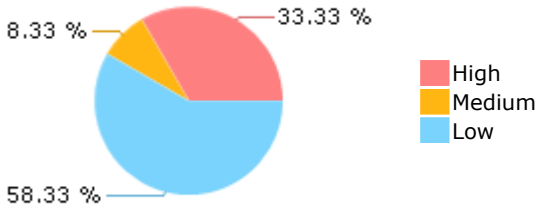
Results Limit

Results limit per query was set to 50

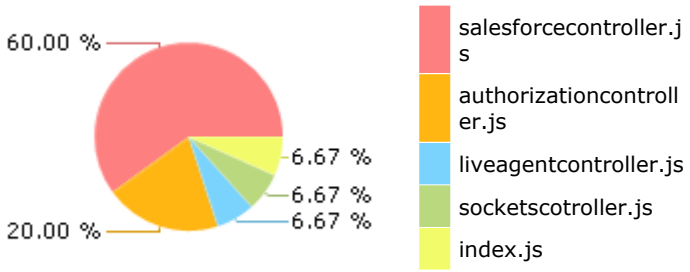
Selected Queries

Selected queries are listed in Result Summary

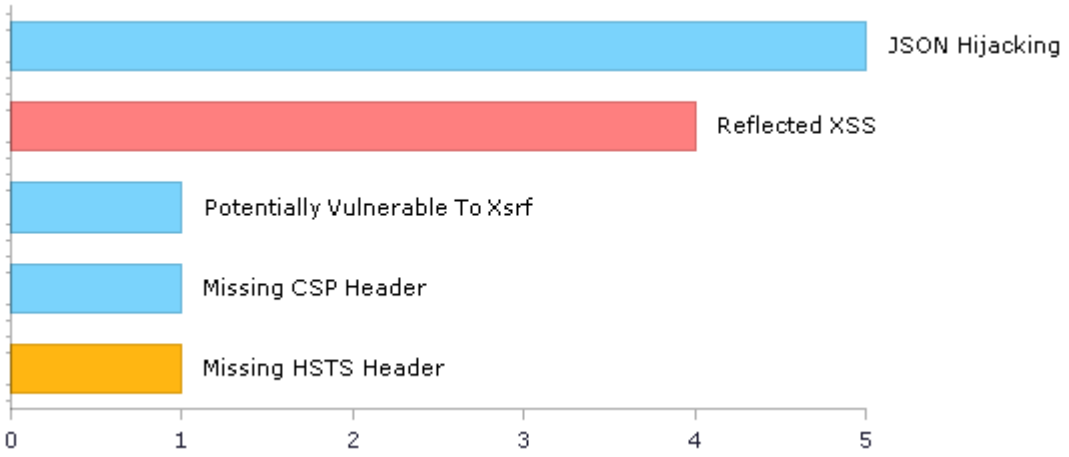
Result Summary



Most Vulnerable Files



Top 5 Vulnerabilities



Scan Summary - OWASP Top 10 2017

Further details and elaboration about vulnerabilities and risks can be found at: OWASP Top 10 2017

Category	Threat Agent	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impact	Issues Found	Best Fix Locations
A1-Injection	App. Specific	EASY	COMMON	EASY	SEVERE	App. Specific	0	0
A2-Broken Authentication	App. Specific	EASY	COMMON	AVERAGE	SEVERE	App. Specific	0	0
A3-Sensitive Data Exposure	App. Specific	AVERAGE	WIDESPREAD	AVERAGE	SEVERE	App. Specific	0	0
A4-XML External Entities (XXE)	App. Specific	AVERAGE	COMMON	EASY	SEVERE	App. Specific	0	0
A5-Broken Access Control*	App. Specific	AVERAGE	COMMON	AVERAGE	SEVERE	App. Specific	0	0
A6-Security Misconfiguration	App. Specific	EASY	WIDESPREAD	EASY	MODERATE	App. Specific	0	0
A7-Cross-Site Scripting (XSS)*	App. Specific	EASY	WIDESPREAD	EASY	MODERATE	App. Specific	4	2
A8-Insecure Deserialization	App. Specific	DIFFICULT	COMMON	AVERAGE	SEVERE	App. Specific	0	0
A9-Using Components with Known Vulnerabilities	App. Specific	AVERAGE	WIDESPREAD	AVERAGE	MODERATE	App. Specific	0	0
A10-Insufficient Logging & Monitoring	App. Specific	AVERAGE	WIDESPREAD	DIFFICULT	MODERATE	App. Specific	0	0

***** Project scan results do not include all relevant queries. Presets and\or Filters should be changed to include all relevant standard queries.

Scan Summary - OWASP Top 10 2013

Further details and elaboration about vulnerabilities and risks can be found at: OWASP Top 10 2013

Category	Threat Agent	Attack Vectors	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impact	Issues Found	Best Fix Locations
A1-Injection	EXTERNAL, INTERNAL, ADMIN USERS	EASY	COMMON	AVERAGE	SEVERE	ALL DATA	0	0
A2-Broken Authentication and Session Management	EXTERNAL, INTERNAL USERS	AVERAGE	WIDESPREAD	AVERAGE	SEVERE	AFFECTED DATA AND FUNCTIONS	0	0
A3-Cross-Site Scripting (XSS)*	EXTERNAL, INTERNAL, ADMIN USERS	AVERAGE	VERY WIDESPREAD	EASY	MODERATE	AFFECTED DATA AND SYSTEM	4	2
A4-Insecure Direct Object References*	SYSTEM USERS	EASY	COMMON	EASY	MODERATE	EXPOSED DATA	0	0
A5-Security Misconfiguration	EXTERNAL, INTERNAL, ADMIN USERS	EASY	COMMON	EASY	MODERATE	ALL DATA AND SYSTEM	0	0
A6-Sensitive Data Exposure	EXTERNAL, INTERNAL, ADMIN USERS, USERS BROWSERS	DIFFICULT	UNCOMMON	AVERAGE	SEVERE	EXPOSED DATA	0	0
A7-Missing Function Level Access Control	EXTERNAL, INTERNAL USERS	EASY	COMMON	AVERAGE	MODERATE	EXPOSED DATA AND FUNCTIONS	0	0
A8-Cross-Site Request Forgery (CSRF)*	USERS BROWSERS	AVERAGE	COMMON	EASY	MODERATE	AFFECTED DATA AND FUNCTIONS	0	0
A9-Using Components with Known Vulnerabilities	EXTERNAL USERS, AUTOMATED TOOLS	AVERAGE	WIDESPREAD	DIFFICULT	MODERATE	AFFECTED DATA AND FUNCTIONS	0	0
A10-Unvalidated Redirects and Forwards	USERS BROWSERS	AVERAGE	WIDESPREAD	DIFFICULT	MODERATE	AFFECTED DATA AND FUNCTIONS	0	0

***** Project scan results do not include all relevant queries. Presets and\or Filters should be changed to include all relevant standard queries.

Scan Summary - PCI DSS v3.2

Category	Issues Found	Best Fix Locations
PCI DSS (3.2) - 6.5.1 - Injection flaws - particularly SQL injection	0	0
PCI DSS (3.2) - 6.5.2 - Buffer overflows	0	0
PCI DSS (3.2) - 6.5.3 - Insecure cryptographic storage	0	0
PCI DSS (3.2) - 6.5.4 - Insecure communications	0	0
PCI DSS (3.2) - 6.5.5 - Improper error handling	0	0
PCI DSS (3.2) - 6.5.7 - Cross-site scripting (XSS)	4	2
PCI DSS (3.2) - 6.5.8 - Improper access control	0	0
PCI DSS (3.2) - 6.5.9 - Cross-site request forgery*	0	0
PCI DSS (3.2) - 6.5.10 - Broken authentication and session management	0	0

***** Project scan results do not include all relevant queries. Presets and\or Filters should be changed to include all relevant standard queries.

Scan Summary - FISMA 2014

Category	Description	Issues Found	Best Fix Locations
Access Control	Organizations must limit information system access to authorized users, processes acting on behalf of authorized users, or devices (including other information systems) and to the types of transactions and functions that authorized users are permitted to exercise.	0	0
Audit And Accountability	Organizations must: (i) create, protect, and retain information system audit records to the extent needed to enable the monitoring, analysis, investigation, and reporting of unlawful, unauthorized, or inappropriate information system activity; and (ii) ensure that the actions of individual information system users can be uniquely traced to those users so they can be held accountable for their actions.	0	0
Configuration Management	Organizations must: (i) establish and maintain baseline configurations and inventories of organizational information systems (including hardware, software, firmware, and documentation) throughout the respective system development life cycles; and (ii) establish and enforce security configuration settings for information technology products employed in organizational information systems.	0	0
Identification And Authentication	Organizations must identify information system users, processes acting on behalf of users, or devices and authenticate (or verify) the identities of those users, processes, or devices, as a prerequisite to allowing access to organizational information systems.	0	0
Media Protection	Organizations must: (i) protect information system media, both paper and digital; (ii) limit access to information on information system media to authorized users; and (iii) sanitize or destroy information system media before disposal or release for reuse.	0	0
System And Communications Protection	Organizations must: (i) monitor, control, and protect organizational communications (i.e., information transmitted or received by organizational information systems) at the external boundaries and key internal boundaries of the information systems; and (ii) employ architectural designs, software development techniques, and systems engineering principles that promote effective information security within organizational information systems.	0	0
System And Information Integrity*	Organizations must: (i) identify, report, and correct information and information system flaws in a timely manner; (ii) provide protection from malicious code at appropriate locations within organizational information systems; and (iii) monitor information system security alerts and advisories and take appropriate actions in response.	4	2

***** Project scan results do not include all relevant queries. Presets and\or Filters should be changed to include all relevant standard queries.

Scan Summary - NIST SP 800-53

Category	Issues Found	Best Fix Locations
AC-12 Session Termination (P2)	0	0
AC-3 Access Enforcement (P1)	0	0
AC-4 Information Flow Enforcement (P1)	0	0
AC-6 Least Privilege (P1)	0	0
AU-9 Protection of Audit Information (P1)	0	0
CM-6 Configuration Settings (P2)	0	0
IA-5 Authenticator Management (P1)	0	0
IA-6 Authenticator Feedback (P2)	0	0
IA-8 Identification and Authentication (Non-Organizational Users) (P1)	0	0
SC-12 Cryptographic Key Establishment and Management (P1)	0	0
SC-13 Cryptographic Protection (P1)	0	0
SC-17 Public Key Infrastructure Certificates (P1)	0	0
SC-18 Mobile Code (P2)	0	0
SC-23 Session Authenticity (P1)*	5	2
SC-28 Protection of Information at Rest (P1)	0	0
SC-4 Information in Shared Resources (P1)	0	0
SC-5 Denial of Service Protection (P1)	0	0
SC-8 Transmission Confidentiality and Integrity (P1)	0	0
SI-10 Information Input Validation (P1)*	0	0
SI-11 Error Handling (P2)	0	0
SI-15 Information Output Filtering (P0)*	4	2
SI-16 Memory Protection (P1)	0	0

***** Project scan results do not include all relevant queries. Presets and\or Filters should be changed to include all relevant standard queries.

Scan Summary - OWASP Mobile Top 10 2016

Category	Description	Issues Found	Best Fix Locations
M1-Improper Platform Usage	This category covers misuse of a platform feature or failure to use platform security controls. It might include Android intents, platform permissions, misuse of TouchID, the Keychain, or some other security control that is part of the mobile operating system. There are several ways that mobile apps can experience this risk.	0	0
M2-Insecure Data Storage	This category covers insecure data storage and unintended data leakage.	0	0
M3-Insecure Communication	This category covers poor handshaking, incorrect SSL versions, weak negotiation, cleartext communication of sensitive assets, etc.	0	0
M4-Insecure Authentication	This category captures notions of authenticating the end user or bad session management. This can include: -Failing to identify the user at all when that should be required -Failure to maintain the user's identity when it is required -Weaknesses in session management	0	0
M5-Insufficient Cryptography	The code applies cryptography to a sensitive information asset. However, the cryptography is insufficient in some way. Note that anything and everything related to TLS or SSL goes in M3. Also, if the app fails to use cryptography at all when it should, that probably belongs in M2. This category is for issues where cryptography was attempted, but it wasnt done correctly.	0	0
M6-Insecure Authorization	This is a category to capture any failures in authorization (e.g., authorization decisions in the client side, forced browsing, etc.). It is distinct from authentication issues (e.g., device enrolment, user identification, etc.). If the app does not authenticate users at all in a situation where it should (e.g., granting anonymous access to some resource or service when authenticated and authorized access is required), then that is an authentication failure not an authorization failure.	0	0
M7-Client Code Quality	This category is the catch-all for code-level implementation problems in the mobile client. That's distinct from server-side coding mistakes. This would capture things like buffer overflows, format string vulnerabilities, and various other code-level mistakes where the solution is to rewrite some code that's running on the mobile device.	0	0
M8-Code Tampering	This category covers binary patching, local resource modification, method hooking, method swizzling, and dynamic memory modification. Once the application is delivered to the mobile device, the code and data resources are resident there. An attacker can either directly modify the code, change the contents of memory dynamically, change or replace the system APIs that the application uses, or	0	0

	modify the application's data and resources. This can provide the attacker a direct method of subverting the intended use of the software for personal or monetary gain.		
M9-Reverse Engineering	This category includes analysis of the final core binary to determine its source code, libraries, algorithms, and other assets. Software such as IDA Pro, Hopper, otool, and other binary inspection tools give the attacker insight into the inner workings of the application. This may be used to exploit other nascent vulnerabilities in the application, as well as revealing information about back end servers, cryptographic constants and ciphers, and intellectual property.	0	0
M10-Extraneous Functionality	Often, developers include hidden backdoor functionality or other internal development security controls that are not intended to be released into a production environment. For example, a developer may accidentally include a password as a comment in a hybrid app. Another example includes disabling of 2-factor authentication during testing.	0	0

Scan Summary - OWASP Top 10 API

Category	Issues Found	Best Fix Locations
API1-Broken Object Level Authorization	0	0
API2-Broken Authentication	0	0
API3-Excessive Data Exposure	0	0
API4-Lack of Resources and Rate Limiting	0	0
API5-Broken Function Level Authorization	0	0
API6-Mass Assignment	0	0
API7-Security Misconfiguration	0	0
API8-Injection	0	0
API9-Improper Assets Management	0	0
API10-Insufficient Logging and Monitoring	0	0

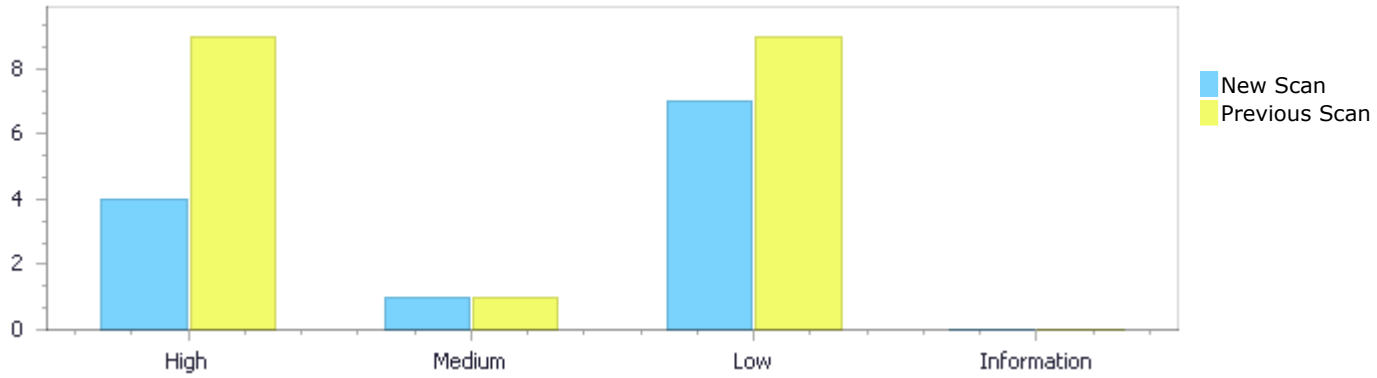
Scan Summary - Custom

Category	Issues Found	Best Fix Locations
Must audit	0	0
Check	0	0
Optional	0	0

Results Distribution By Status

Compared to project scan from 11/10/2020 8:38 AM

	High	Medium	Low	Information	Total
New Issues	0	1	1	0	2
Recurrent Issues	4	0	6	0	10
Total	4	1	7	0	12
Fixed Issues	5	1	3	0	9



Results Distribution By State

	High	Medium	Low	Information	Total
Confirmed	0	0	0	0	0
Not Exploitable	0	0	0	0	0
To Verify	4	1	7	0	12
Urgent	0	0	0	0	0
Proposed Not Exploitable	0	0	0	0	0
Total	4	1	7	0	12

Result Summary

Vulnerability Type	Occurrences	Severity
Reflected XSS	4	High
Missing HSTS Header	1	Medium
JSON Hijacking	5	Low
Missing CSP Header	1	Low
Potentially Vulnerable To Xsrf	1	Low

10 Most Vulnerable Files

High and Medium Vulnerabilities

File Name	Issues Found
salesforcecontroller.js	4
liveagentcontroller.js	1

Scan Results Details

Reflected XSS

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\Reflected XSS Version:2

Categories

PCI DSS v3.2: PCI DSS (3.2) - 6.5.7 - Cross-site scripting (XSS)
OWASP Top 10 2013: A3-Cross-Site Scripting (XSS)
FISMA 2014: System And Information Integrity
NIST SP 800-53: SI-15 Information Output Filtering (P0)
OWASP Top 10 2017: A7-Cross-Site Scripting (XSS)

Description

Reflected XSS\Path 1:

Severity	High
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=1
Status	Recurrent

The application's function embeds untrusted data in the generated output with send, at line 410 of salesforcecontroller.js. This untrusted data is embedded straight into the output without proper sanitization or encoding, enabling an attacker to inject malicious code into the output.

The attacker would be able to alter the returned web page by simply providing modified data in the user input message, which is read by the function method at line 410 of salesforcecontroller.js. This input then flows through the code straight to the output web page, without sanitization.

This can enable a Reflected Cross-Site Scripting (XSS) attack.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	413	421
Object	message	send

Code Snippet
File Name salesforcecontroller.js
Method offlineCase: async function(req, res) {

```
....
413.              let message = req.body.message
....
421.                  return res.status(200).send(
```

Reflected XSS\Path 2:

Severity	High
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=2
Status	Recurrent

The application's function embeds untrusted data in the generated output with send, at line 410 of salesforcecontroller.js. This untrusted data is embedded straight into the output without proper sanitization or encoding, enabling an attacker to inject malicious code into the output.

The attacker would be able to alter the returned web page by simply providing modified data in the user input sessionId, which is read by the function method at line 410 of salesforcecontroller.js. This input then flows through the code straight to the output web page, without sanitization.

This can enable a Reflected Cross-Site Scripting (XSS) attack.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	412	421
Object	sessionId	send

Code Snippet
File Name salesforcecontroller.js
Method offlineCase: async function(req, res) {

```
....
412.              let sessionId = req.body.sessionId
....
421.                  return res.status(200).send(
```

Reflected XSS\Path 3:

Severity	High
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=3
Status	Recurrent

The application's function embeds untrusted data in the generated output with send, at line 410 of salesforcecontroller.js. This untrusted data is embedded straight into the output without proper sanitization or encoding, enabling an attacker to inject malicious code into the output.

The attacker would be able to alter the returned web page by simply providing modified data in the user input body, which is read by the function method at line 410 of salesforcecontroller.js. This input then flows through the code straight to the output web page, without sanitization.

This can enable a Reflected Cross-Site Scripting (XSS) attack.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	417	421
Object	body	send

Code Snippet
File Name salesforcecontroller.js
Method offlineCase: async function(req, res) {

```
....
417.              let contact = await this.createContact(apigeeToken,
salesforceToken, req.body)
....
421.                   return res.status(200).send(
```

Reflected XSS\Path 4:

Severity	High
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=4
Status	Recurrent

The application's function embeds untrusted data in the generated output with send, at line 453 of salesforcecontroller.js. This untrusted data is embedded straight into the output without proper sanitization or encoding, enabling an attacker to inject malicious code into the output.

The attacker would be able to alter the returned web page by simply providing modified data in the user input caseId, which is read by the function method at line 453 of salesforcecontroller.js. This input then flows through the code straight to the output web page, without sanitization.

This can enable a Reflected Cross-Site Scripting (XSS) attack.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	457	459
Object	caseId	send

Code Snippet
File Name salesforcecontroller.js
Method caseNumber : async function(req, res){

```
....
457.              let caseNumber = await this.getCaseNumber(apigeeToken,
salesforceToken, req.body.caseId)
....
459.              return res.status(200).send(
```

Missing HSTS Header

Query Path:
JavaScript\Cx\JavaScript Medium Threat\Missing HSTS Header Version:1

Description

Missing HSTS Header\Path 1:

Severity	Medium
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=5
Status	New

The web-application does not define an HSTS header, leaving it vulnerable to attack.

	Source	Destination
File	liveagentcontroller.js	liveagentcontroller.js
Line	18	18
Object	json	json

Code Snippet
File Name liveagentcontroller.js
Method chatMessage : async function(req, res){

```
....
18.             res.json({ response })
```

JSON Hijacking

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\JSON Hijacking Version:1

Categories

NIST SP 800-53: SC-23 Session Authenticity (P1)

Description
JSON Hijacking\Path 1:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=7
Status	Recurrent

The application returns sensitive data in a JSON object in salesforcecontroller.js at line 410.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	393	421
Object	stringify	send

Code Snippet
File Name salesforcecontroller.js
Method generateOfflineCase : async function(apigeeToken, salesforceToken, config, sessionId, contact, message){

```
....
393.              body     : JSON.stringify(newCase)
```

File Name salesforcecontroller.js
Method offlineCase: async function(req, res) {

```
....
421.                    return res.status(200).send(
```

JSON Hijacking\Path 2:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=8
Status	Recurrent

The application returns sensitive data in a JSON object in salesforcecontroller.js at line 410.

	Source	Destination
File	salesforcecontroller.js	salesforcecontroller.js
Line	543	421
Object	stringify	send

Code Snippet
File Name salesforcecontroller.js
Method createContact: async function(apigeeToken, salesforceToken, clientInfo){

```
....
543.                body     : JSON.stringify(newContact)
```

File Name salesforcecontroller.js

Method offlineCase: async function(req, res) {

```
....
421.                    return res.status(200).send(
```

JSON Hijacking\Path 3:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=9
Status	Recurrent

The application returns sensitive data in a JSON object in salesforcecontroller.js at line 595.

	Source	Destination
File	authorizationcontroller.js	salesforcecontroller.js
Line	59	602
Object	stringify	send

Code Snippet
File Name authorizationcontroller.js

Method getSalesforceToken : async function(){

```
....
59.                 body     : JSON.stringify({})
```

File Name salesforcecontroller.js

Method getCaseByChatKey : async function(req, res){

```
....
602.              res.status(200).send(
```

JSON Hijacking\Path 4:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=10
Status	Recurrent

The application returns sensitive data in a JSON object in salesforcecontroller.js at line 410.

	Source	Destination
File	authorizationcontroller.js	salesforcecontroller.js
Line	59	421
Object	stringify	send

Code Snippet

File Name authorizationcontroller.js

Method getSalesforceToken : async function(){

```
....
59.                 body     : JSON.stringify({})
```

File Name salesforcecontroller.js

Method offlineCase: async function(req, res) {

```
....
421.                  return res.status(200).send(
```

JSON Hijacking\Path 5:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=11
Status	Recurrent

The application returns sensitive data in a JSON object in salesforcecontroller.js at line 453.

	Source	Destination
File	authorizationcontroller.js	salesforcecontroller.js
Line	59	459
Object	stringify	send

Code Snippet
File Name authorizationcontroller.js
Method getSalesforceToken : async function(){

```
....
59.                    body     : JSON.stringify({})
```

File Name salesforcecontroller.js
Method caseNumber : async function(req, res){

```
....
459.               return res.status(200).send(
```

Missing CSP Header

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\Missing CSP Header Version:2

Description

Missing CSP Header\Path 1:

Severity Low
Result State To Verify
Online Results https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=6
Status New

A Content Security Policy is not explicitly defined within the web-application.

	Source	Destination
File	socketscotroller.js	socketscotroller.js
Line	22	22
Object	send	send

Code Snippet
File Name socketscotroller.js
Method socketsRoom : async function(req, res) {

```
....
22.               res.status(200).send()
```

Potentially Vulnerable To Xsrf

Query Path:
JavaScript\Cx\JavaScript Server Side Vulnerabilities\Potentially Vulnerable To Xsrf Version:2

Description

Potentially Vulnerable To Xsrf\Path 1:

Severity	Low
Result State	To Verify
Online Results	https://cebamsgcmmgr01.cebupacificair.com/CxWebClient/ViewerMain.aspx?scanid=1000772&projectid=14&pathid=12
Status	Recurrent

Method express at line 2 of index.js gets a parameter from a user request from app. This parameter value flows through the code and is eventually used to access application state altering functionality. This may enable Cross-Site Request Forgery (XSRF).

	Source	Destination
File	index.js	index.js
Line	2	2
Object	app	app

Code Snippet
File Name index.js
Method const app = express();

```
....
2.  const app = express();
```

Reflected XSS

Risk

What might happen

A successful XSS exploit would allow an attacker to rewrite web pages and insert malicious scripts which would alter the intended output. This could include HTML fragments, CSS styling rules, arbitrary JavaScript, or references to third party code. An attacker could use this to steal users' passwords, collect personal data such as credit card details, provide false information, or run malware. From the victim's point of view, this is performed by the genuine website, and the victim would blame the site for incurred damage.

The attacker could use social engineering to cause the user to send the website modified input, which will be returned in the requested web page.

Cause

How does it happen

The application creates web pages that include untrusted data, whether from user input, the application's database, or from other external sources. The untrusted data is embedded directly in the page's HTML, causing the browser to display it as part of the web page. If the input includes HTML fragments or JavaScript, these are displayed too, and the user cannot tell that this is not the intended page. The vulnerability is the result of directly embedding arbitrary data without first encoding it in a format that would prevent the browser from treating it like HTML or code instead of plain text.

Note that an attacker can exploit this vulnerability either by modifying the URL, or by submitting malicious data in the user input or other request fields.

General Recommendations

How to avoid it

- Fully encode all dynamic data, regardless of source, before embedding it in output.
- Encoding should be context-sensitive. For example:
  - HTML encoding for HTML content
  - HTML Attribute encoding for data output to attribute values
  - JavaScript encoding for server-generated JavaScript
- It is recommended to use the platform-provided encoding functionality, or known security libraries for encoding output.
- Implement a Content Security Policy (CSP) with explicit whitelists for the application's resources only.
- As an extra layer of protection, validate all untrusted data, regardless of source (note this is not a replacement for encoding). Validation should be based on a whitelist: accept only data fitting a specified structure, rather than reject bad patterns. Check for:
  - Data type
  - Size
  - Range
  - Format
  - Expected values
- In the `Content-Type` HTTP response header, explicitly define character encoding (charset) for the entire page.
- Set the `HTTPOnly` flag on the session cookie for "Defense in Depth", to prevent any successful XSS exploits from stealing the cookie.

Source Code Examples

JavaScript

No Input Sanitization while Outputting User Input to HTML Context in Express.js

```
app.get(welcomeUrl, function(req, res) {
      var name = req.query.name;
      res.send("<h1>Welcome, " + name + "!</h1>"); // Setting the payload
name=<script>alert(1)</script> will result in an alert prompt in browsers without XSS
protection, such as Firefox, demonstrating XSS
});
```

Using a Template Engine, Such as Handlebars, to Write User Input to An HTML Context, Where User Input is Encoded by The Template Engine

```
// In Web-Server code
app.get(welcomeUrl, function(req, res) {
      var name = req.query.name;
      res.render('home', {name: name});
});

// In 'home' View code
<h1>Welcome, {{name}} </h1>"
```

Missing HSTS Header

Risk

What might happen

Failure to set an HSTS header and provide it with a reasonable "max-age" value of at least one year may leave users vulnerable to Man-in-the-Middle attacks.

Cause

How does it happen

Many users browse to websites by simply typing the domain name into the address bar, without the protocol prefix. The browser will automatically assume that the user's intended protocol is HTTP, instead of the encrypted HTTPS protocol.

When this initial request is made, an attacker can perform a Man-in-the-Middle attack and manipulate it to redirect users to a malicious web-site of the attacker's choosing. To protect the user from such an occurence, the HTTP Strict Transport Security (HSTS) header instructs the user's browser to disallow use of an unsecure HTTP connection to the the domain associated with the HSTS header.

Once a browser that supports the HSTS feature has visited a web-site and the header was set, it will no longer allow communicating with the domain over an HTTP connection.

Once an HSTS header was issued for a specific website, the browser is also instructed to prevent users from manually overriding and accepting an untrusted SSL certificate for as long as the "max-age" value still applies. The recommended "max-age" value is for at least one year in seconds, or 31536000.

General Recommendations

How to avoid it

- Before setting the HSTS header - consider the implications it may have:
  - Forcing HTTPS will prevent any future use of HTTP, which could hinder some testing
  - Disabling HSTS is not trivial, as once it is disabled on the site, it must also be disabled on the browser
- Set the HSTS header either explicitly within application code, or using web-server configurations.
- Ensure the "max-age" value for HSTS headers is set to 31536000 to ensure HSTS is strictly enforced for at least one year.
- Include the "includeSubDomains" to maximize HSTS coverage, and ensure HSTS is enforced on all sub-domains under the current domain
  - Note that this may prevent secure browser access to any sub-domains that utilize HTTP; however, use of HTTP is very severe and highly discouraged, even for websites that do not contain any sensitive information, as their contents can still be tampered via Man-in-the-Middle attacks to phish users under the HTTP domain.
- Once HSTS has been enforced, submit the web-application's address to an HSTS preload list - this will ensure that, even if a client is accessing the web-application for the first time (implying HSTS has not yet been set by the web-application), a browser that respects the HSTS preload list would still treat the web-application as if it had already issued an HSTS header. Note that this requires the server to have a trusted SSL certificate, and issue an HSTS header with a maxAge of 1 year (31536000)
- Note that this query is designed to return one result per application. This means that if more than one vulnerable response without an HSTS header is identified, only the first identified instance of this issue will be highlighted as a result. If a misconfigured instance of HSTS is identified (has a short lifespan, or is missing the "includeSubDomains" flag), that result will be flagged. Since HSTS is required to be

enforced across the entire application to be considered a secure deployment of HSTS functionality, fixing this issue only where the query highlights this result is likely to produce subsequent results in other sections of the application; therefore, when adding this header via code, ensure it is uniformly deployed across the entire application. If this header is added via configuration, ensure that this configuration applies to the entire application.

- Note that misconfigured HSTS headers that do not contain the recommended max-age value of at least one year or the "includeSubDomains" flag will still return a result for a missing HSTS header.

Source Code Examples

JavaScript

Using Helmet with Express

```
var express = require('express')
var helmet = require('helmet') // Helmet includes HSTS, defined to one year and with
"includeSubDomains", as a built-in header

var app = express()
app.use(helmet())
```

Using Explicit HSTS Package - Built into Helmet, So Either 'HSTS' or 'Helmet' Can Be Used

```
var hsts = require('hsts')

app.use(hsts({
  maxAge: 31536000,
  includeSubDomains: true // Also enabled by default
}))
```

Explicitly Setting HSTS Header in Code

```
res.setHeader("Strict-Transport-Security", "max-age=31536000; includeSubDomains");
```

Missing CSP Header

Risk

What might happen

The Content-Security-Policy header enforces that the source of content, such as the origin of a script, embedded (child) frame, embedding (parent) frame or image, are trusted and allowed by the current web-page; if, within the web-page, a content's source does not adhere to a strict Content Security Policy, it is promptly rejected by the browser. Failure to define a policy may leave the application's users exposed to Cross-Site Scripting (XSS) attacks, Clickjacking attacks, content forgery and more.

Cause

How does it happen

The Content-Security-Policy header is used by modern browsers as an indicator for trusted sources of content, including media, images, scripts, frames and more. If these policies are not explicitly defined, default browser behavior would allow untrusted content.

The application creates web responses, but does not properly set a Content-Security-Policy header.

General Recommendations

How to avoid it

Explicitly set the Content-Security-Policy headers for all applicable policy types (frame, script, form, script, media, img etc.) according to business requirements and deployment layout of external file hosting services. Specifically, do not use a wildcard, '*', to specify these policies, as this would allow content from any external resource.

The Content-Security-Policy can be explicitly defined within web-application code, as a header managed by web-server configurations, or within <meta> tags in the HTML `&#x3C;head&#x3E;` section.

Source Code Examples

JavaScript

Setting The CSP Header Explicitly

```
app.use(function(req, res, next) {
    res.setHeader("Content-Security-Policy", "script-src 'self'");
    return next();
});
```

JSON Hijacking

Risk

What might happen

An attacker may craft a malicious web-page and, via social engineering, get a user to access it. If that user uses an outdated browser, and is logged in to the vulnerable application, the attacker may retrieve the contents of JSON objects, bypassing SOP and reading these objects' contents, and any sensitive information therein. Note that, while this issue generally applies to outdated, legacy and obsolete browsers, JSON hijacking issues also periodically arise in more modern browsers.

Cause

How does it happen

The following is required for the application to be vulnerable:

- The application must be using cookie-based authentication
- The application responds in an authenticated manner to simple GET request
- This sensitive data is returned as a JSON, specifically in array form (enclosed in square [] brackets), and containing objects inside within the array

By returning a JSON array, an attacker may create a malicious website, which incorporates a <script> tag in their page in the following manner:

```
&#x3C;script src="https://example.com/path/to/vulnerable/page"&#x3E;&#x3C;/script&#x3E;
```

The browser will interpret the returned value as an object, causing it to temporarily exist in the malicious web-page's DOM; however since this object it is not assigned or referenced, it will be ephemeral, and would normally be immediately discarded. This is similar to any other declaration or return value without an assignment, and the malicious web-page would be unable to reference it in any way.

To actually exploit this issue in a vulnerable browser, an attacker would have to be able to override the prototype function for setters, which is normally restricted. If the browser is vulnerable and does allow overwriting certain core prototypes for setters in Javascript within the web-page context, they could craft a Javascript that, once a vulnerable page is included with the <script src=></script> method, an object will initiate construction, trigger the overridden setter prototypes with the JSON object's contents as values, and an attacker would then be able to access these values in the context of their malicious web-page. From that point on - accessing them would be trivial.

General Recommendations

How to avoid it

There are multiple methods to address this issue:

- Do not respond with JSON arrays as they are wrapped with square brackets, which immediately get evaluated as objects; if required, wrap the array with an external object (e.g. {"array":[]}) or add some sort of prefix to prevent this issue
- If required, respond with JSON arrays only to POST request; ensure no sensitive information is ever returned as an array to a GET request
- Prefix the JSON object with either JavaScript (`for(;;);`) or an unparseable JSON (`{};`) , and, before processing it on the client, strip this prefix; the latter will cause importation to fail, and the former will

cause importation to hang forever - either way, this will prevent an attacker from importing a JSON object as a script

---

Source Code Examples

JavaScript

JSON Array Containing Sensitive Data and Vulnerable to JSON Hijacking

```
app.get('/security_answers', function(req, res) {
	var userId = req.session.userId; // Assume userId is assigned only if authenticated
	if (userId) {
		connection.query('SELECT security_question, security_answer' +
			'FROM security_qa WHERE userId=?',[userId],
		function(err,results) {
			/* JSON response would be of the form [{"security_question":"Q1",
"security_answer":"A1"},
		* "security_question":"Q2", "security_answer":"A2"}], implying it can be loaded as an
Object
		* using <script src="path/to/this/JSON"></script>
			*/
				res.json(results);
			});
	} else {
		res.json({"error":"Not authenticated"});
	}
});
```

JSON Array Wrapped in JSON Object, Preventing JSON Hijacking

```
app.get('/security_answers', function(req, res) {
	var userId = req.session.userId; // Assume userId is assigned only if authenticated
	if (userId) {
		connection.query('SELECT security_question, security_answer' +
			'FROM security_qa WHERE userId=?',[userId],
		function(err,results) {
			/* JSON response would be of the form
{"results":[{"security_question":"Q1", "security_answer":"A1"},
				* {"security_question":"Q2","security_answer":"A2"}]}, implying
it will always be formulated
		* like a JSON object, not a JSON array, and would not be loaded externally
				*/
				res.json({"results":results});
			});
	} else {
		res.json({"error":"Not authenticated"});
	}
});
```

JSON Array Preceded by JavaScript That Freezes Page, Preventing Loading Array

```
app.get('/security_answers', function(req, res) {
      var userId = req.session.userId; // Assume userId is assigned only if authenticated
      if (userId) {
            connection.query('SELECT security_question, security_answer' +
                  'FROM security_qa WHERE userId=?',[userId],
            function(err,results) {
                              res.setHeader('Content-Type', 'application/json');
                              /* JSON response would be of the form
"for(;;);[{"security_question":"Q1", "security_answer":"A1"},
                               * {"security_question":"Q2","security_answer":"A2"}]", implying
if it is loaded as an Object using
                               * <script src="path/to/this/JSON"></script>, an endless loop will
occur before the object is loaded
                               */
                        res.send("for(;;);"+JSON.stringify(results));
                        });
      } else {
            res.json({"error":"Not authenticated"});
      }
});
```

Potentially Vulnerable To Xsrf

Risk

What might happen

An attacker could cause the victim to perform any action for which the victim is authorized, such as transferring funds from the victim's account to the attacker's. The action will be logged as being performed by the victim, in the context of their account, and potentially without their knowledge that this action has occurred.

Cause

How does it happen

The application performs some action that modifies database contents, based purely on HTTP request content, and does not require per-request renewed authentication (such as transaction authentication or a synchronizer token), instead relying solely on session authentication. This means that an attacker could use social engineering to cause a victim to browse to a link which contains a transaction request to the vulnerable application, submitting that request from the user's browser. Once the application receives the request, it would trust the victim's session, and would perform the action. This type of attack is known as Cross-Site Request Forgery (XSRF or CSRF).

A Cross-Site Request Forgery attack relies on the trust between a server and an authenticated client. By only validating the session, the server ensures that a request has emerged from a client's web-browser. However, any website may submit GET and POST requests to other websites, to which the browser will automatically add the session token if it is in a cookie. This cross-site request can then be trusted as arriving from the user's browser, but does not validate that it was their intent was to make this request.

In some cases, XSRF protection functionality exists in the application, but is either not implemented, or explicitly disabled.

General Recommendations

How to avoid it

Mitigating XSRF requires an additional layer of authentication that is built into the request validation mechanism. This mechanism would attach an additional token that only applies to the given user; this token would be available within the user's web-page, but will not be attached automatically to a request from a different website (e.g. not stored in a cookie). Since the token is not automatically attached to the request, and is not available to the attacker, and is required by the server to process the request, it would be completely impossible for the attacker to fill in a valid cross-site form that contains this token.

Many platforms offer built-in XSRF mitigation functionality which should be used, and perform this type of token management under the hood. Alternatively, use a known or trusted library which adds this functionality.

If implementing XSRF protection is required, this protection should adhere to the following rules:

- Any state altering form (Create, Update, Delete operations) should enforce XSRF protection, by adding an XSRF token to every state altering form submission on the client.
- An XSRF token should be generated, and be unique per-user per-session (and, preferably, per request).
- The XSRF token should be inserted into the client side form, and be submitted to the server as part of the form request. For example, it could be a hidden field in an HTML form, or a custom header added by a Javascript request.

- The XSRF token in the request body or custom header must then be verified as belonging to the current user by the server, before a request is authorized and processed as valid.

Always rely on best practices when using XSRF protection - always enable built-in functionality or available libraries where possible.

When using application-wide XSRF protection, never explicitly disable or subvert XSRF protection for specific functionality unless said functionality has been thoroughly verified to not require XSRF protection.

Source Code Examples

JavaScript
Applying 'csurf' Library to HTML Forms in Express

```
app.get('/profile/email/edit', (req, res) => {
  // Add csrfToken() as a value inside the form, so that the token is submitted in the
request
  res.send(`
  <h1>Change E-Mail Form</h1>
  <form action="/profile/email/edit" method="POST">
   <input id="email" name="email" type="text" />
       <input id="email" name="verify_email" type="text" />
   <input type="submit" value="Ok" />
   <input type="hidden" name="_csrf" value="${req.csrfToken()}" />
    </form>
  `);
});

// If the _csrf token in the request body is not equal to the synchronizer token, the POST
request will be rejected, and this method will not trigger
app.post('/profile/email/edit', (req, res) => {
      // Authenticate user session
      // Update email address
}
```

Scanned Languages

Language	Hash Number	Change Date
JavaScript	0109410431041810	2/17/2020
Common	0174166543303234	10/30/2020