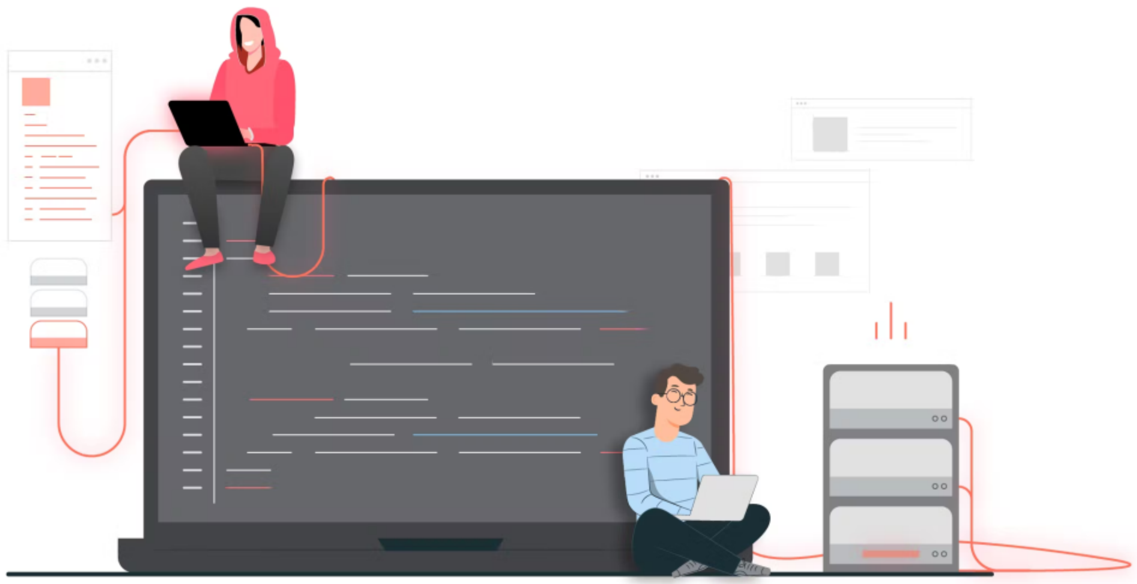
Hack the Stack with LocalStack: Code Vulnerabilities Explained

BY DENNIS BRINKROLF | MARCH 02, 2021

Security



LocalStack is a popular open source application that provides an easy-to-use test framework for cloud applications. It enables you to host a fully functional AWS cloud setup in your local network for developing and testing cloud and serverless apps. According to GitHub, it is one of the most popular open source Python applications.

During our security research into modern applications, we discovered critical code vulnerabilities in the latest LocalStack version. We reported all issues responsibly to the affected vendor. However, after the vendor assessed the risk it left the vulnerabilities we reported unpatched due to a limited attack scenario. In this blog post we analyze the attack scenario, the technical root cause of the code vulnerabilities, and how attackers are able to exploit these vulnerabilities.

Impact

We detected the following vulnerabilities in the latest LocalStack version 0.12.6:

- S5334: OS Command Injection (CVE-2021-32090)
- S5144: Server-Side Request Forgery (SSRF)
- S5131: Cross-Site Scripting (XSS) (CVE-2021-32091)
- S2631: Denial of Service via regular expressions (ReDoS)

A LocalStack instance typically runs in an internal network setup. As shown in this blog post, attackers who are not in this same network are still capable of attacking such application setups remotely. By combining different vulnerabilities, an attacker can completely compromise the local instance and execute arbitrary system commands.

Our video illustrates such an attack and shows how quickly and easily a server can be compromised.

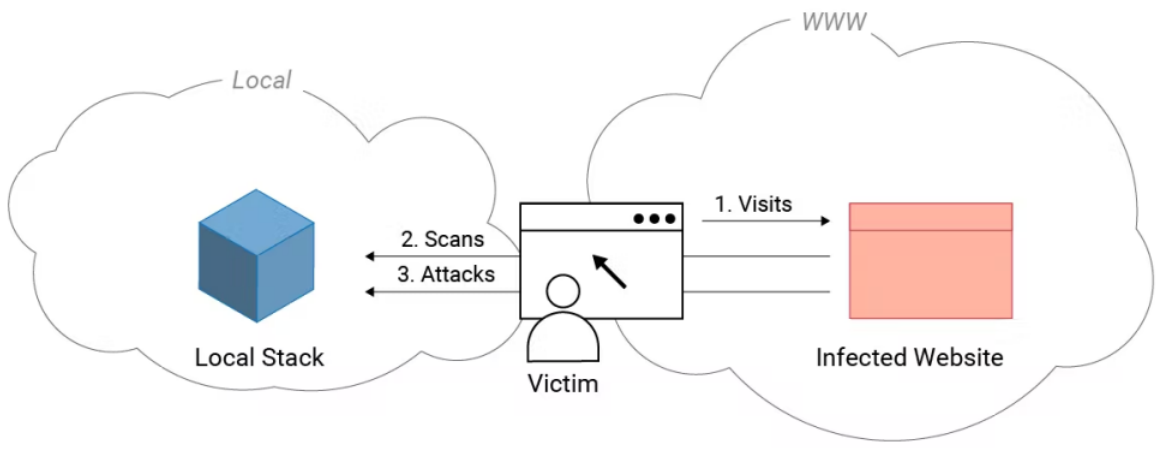
Technical Analysis

In this technical analysis, we first explain how applications that run locally are attacked. Then, we discuss two vulnerabilities that we found in the LocalStack code. The two vulnerabilities can be combined by an attacker to compromise and take over a LocalStack instance.

Remote Attacks on Local Instances

When using LocalStack, we noticed that it does not use any authentication. Probably that is because the LocalStack software is run locally or in a Docker environment, as recommended by the vendor, and is therefore not directly exposed to remote attackers. However, it is a common fallacy that this type of application cannot be attacked at all. Web interfaces of network routers are a popular example of local applications that have been attacked in the real-world by criminals (*Drive-by-Pharming*).

One way for a remote attacker to interact with LocalStack running locally is through a target user's browser. Typically, the browser of the developer who uses LocalStack is also connected to the internet to, for example, read documentation pages. When this victim visits (or is lured to) a malicious/infected website controlled by an attacker, it is possible to trigger cross-site HTTP requests to the victim's local network via JavaScript code (*Cross-Site Request Forgery* - CSRF).



This way, an attacker can send arbitrary requests from a website to a LocalStack instance but cannot read the respective responses. This is prevented by the *cross-origin resource sharing* (CORS) mechanism in the browser. However, merely sending requests to the vulnerable application - even without being able to read the responses - is sufficient to carry out a successful attack via CSFR. The attacker blindly sends the attack payload and hopes that the vulnerable application is reached.

Moreover, LocalStack explicitly allows the execution of cross-origin requests through any page by setting special HTTP headers in the response. This means that the attacker can detect and attack a LocalStack instance through the XHR response and does not actually operate blindly.

```
access-control-allow-origin: *
access-control-allow-methods: HEAD,GET,PUT,POST,DELETE,OPTIONS,PATCH
access-control-allow-headers: authorization,content-type,content-md5,cache-control,x-amz-content-sha256,x-amz-date,x-amz-security-token,x-amz-user-
access-control-expose-headers: x-amz-version-id
```

Note that modern browsers have recently further restricted cross-origin requests to reduce the potential of CSRF attacks. However, we also found a Cross-Site Scripting (XSS) vulnerability in LocalStack which allows an attacker to bypass these protections.

heir own port. All user requests are forwarded to the respective API via a central *edge router*. For this router it is possible to configure a proxy via the LocalStack settings.

localstack/services/edge.py

```
88    def do_forward_request(api, port, method, path, data, headers):
89        if config.FORWARD_EDGE_INMEM:
90            result = do_forward_request_inmem(api, port, method, path, data, headers)
91        else:
92            result = do_forward_request_network(port, method, path, data, headers)
93        if hasattr(result, 'status_code') and result.status_code >= 400 and method == 'OPTIONS':
94            # fall back to successful response for OPTIONS requests
95            return 200
96        return result
```

The function `do_forward_request()` is executed every time a request is sent to the edge router of LocalStack. In line 89 it is checked if the config entry `FORWARD_EDGE_INMEM` is set. In this case, the request is processed locally, otherwise the request is forwarded to the network. Because an attacker can set `FORWARD_EDGE_INMEM` to *False* via a CSRF attack, we reach line 92 every time. Consequently, all requests to the edge router are processed by the function `do_foward_request_network()`. Also, the HTTP responses of the respective requests are printed without sanitization.

localstack/services/edge.py

```
112    def do_forward_request_network(port, method, path, data, headers):
113        connect_host = '%s:%s' % (config.HOSTNAME, port)
114        url = '%s://%s%s' % (get_service_protocol(), connect_host, path)
115        function = getattr(requests, method.lower())
116        response = function(url, data=data, headers=headers, verify=False, stream=True)
117        return response
```

In line 113, the `HOSTNAME` that is used for the forwarded request is read from the configuration. Since an attacker can configure the `HOSTNAME` via CSRF attack, this host is now an attacker-controlled IP. In the following lines the request is constructed and in line 116 the request is executed which leads to a (persistent) SSRF vulnerability.

An interesting point about this feature is that the server copies the entire HTTP request from the client and forwards it to the server. This also means that the HTTP headers of the client are sent to the attacker-controlled server, including the Authorization header. This header is used for authentication in the AWS Cloud which can lead to session hijacking and stealing sensitive data from the test cloud.

As an attacker we can now go even one step further. As mentioned above, the HTTP response of the SSRF request is printed unsanitized in LocalStack. In other words, the attacker can send an XSS payload as a response via his controlled server which leads to a (persistent) Cross-Site Scripting vulnerability in LocalStack. With this, the attacker has installed a persistent man-in-the-middle proxy in LocalStack that controls every HTTP request and response of the LocalStack instance. This enables abuse of further features and to trigger other code vulnerabilities.

Command Injection Vulnerability (CVE-2021-32090)

One possible way to go further is to exploit vulnerabilities in the LocalStack dashboard. When it is active, an attacker can permanently infiltrate the system and compromise the developer's machine via a Command Injection vulnerability. Let's have a look at the affected code lines.

localstack/dashboard/api.py

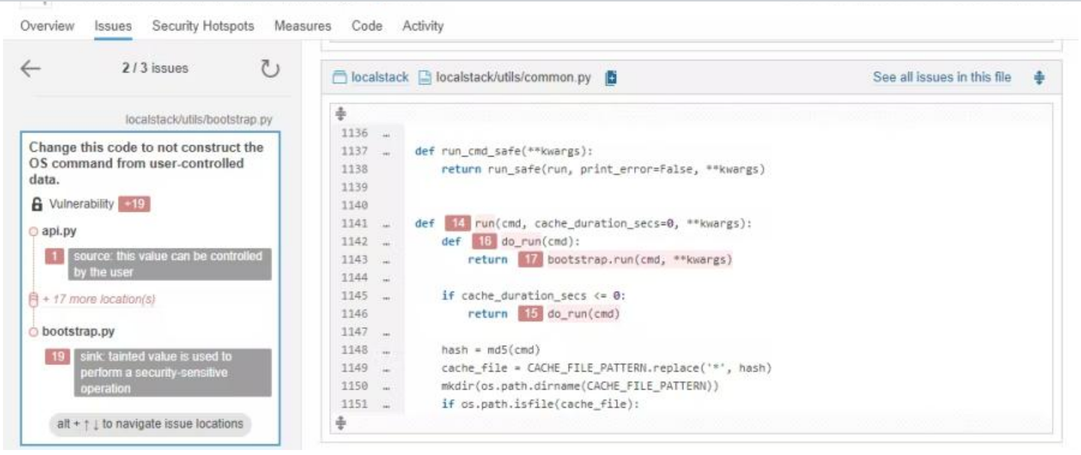
```
85    @app.route('/lambda/<functionName>/code', methods=['POST'])
86    def get_lambda_code(functionName):
...
98        result = infra.get_lambda_code(func_name=functionName, env=env)
```

In line 85 the route is defined that calls the `get_lambda_code()` function in line 86. Here, the parameter `functionName` is passed to the `get_lambda_code()` function via the path of the route. Then, in line 98, this function is executed with the user controlled input.

localstack/dashboard/infra.py

```
258    def get_lambda_code(func_name, retries=1, cache_time=None, env=None):
...
264        out = cmd_lambda('get-function --function-name %s' % func_name, env, cache_time)
```

In line 264, the user-controlled input `func_name` is concatenated into a system command using a format string. Without any sanitization it is passed to the `cmd_lambda()` function. When we follow the user controlled input via further functions, we end up in the `run()` function. This *data flow analysis* is exactly what our security analyzers automate for you (open issue on SonarCloud).


Overview
Issues
Security Hotspots
Measures
Code
Activity
2 / 3 issues
localstack
localstack/utils/common.py
See all issues in this file
alt + ↑ ↓ to navigate issue locations

localstack/utils/bootstrap.py

Change this code to not construct the OS command from user-controlled data.

Vulnerability +19

api.py

1 source: this value can be controlled by the user

+ 17 more location(s)

bootstrap.py

19 sink: tainted value is used to perform a security-sensitive operation

```
def run_cmd_safe(**kwargs):
    return run_safe(run, print_error=False, **kwargs)


def 14 run(cmd, cache_duration_secs=0, **kwargs):
    def 18 do_run(cmd):
        return 17 bootstrap.run(cmd, **kwargs)

    if cache_duration_secs <= 0:
        return 15 do_run(cmd)

    hash = md5(cmd)
    cache_file = CACHE_FILE_PATTERN.replace('*', hash)
    mkdir(os.path.dirname(CACHE_FILE_PATTERN))
    if os.path.isfile(cache_file):
```

localstack/utils/bootstrap.py

```
596    def run(cmd, print_error=True, stderr=subprocess.STDOUT, env_vars=None, inherit_cwd=False, inherit_env=True):
...
613        output = subprocess.check_output(cmd, shell=True, stderr=stderr, env=env_dict, cwd=cwd)
```

In line 613, the shell command is finally executed via `subprocess.check_output()`. Here, the `cmd` parameter contains the user-controlled input that ends up unsanitized in a system command. This leads to a Command Injection vulnerability since the attacker can terminate the original command and execute his own. For example, after a Command Injection, the final `cmd` parameter in line 613 can look like the following to create a new file on the system:

```
cmd =  { test `which aws` || . .venv/bin/activate; }; aws lambda get-function --function-name test;touch sonarsource.txt
```

Summary

In this blog post we analyzed two code vulnerabilities found in the latest **LocalStack (0.12.6)**, a widely used Python application. We outlined how local applications can be attacked remotely, and how the combination of these vulnerabilities can lead to a complete takeover of a LocalStack.

We reported these vulnerabilities to the vendor in October 2020. After reaching out a couple of more times, we received notice in January that these threats are not considered a key concern since LocalStack is executed on a local machine. While we agree that real-world attacks against local instances are less likely than against directly exposed applications, we believe that developers should be aware of these risks in order to protect their setups and to write secure code for their own applications.

DENNIS BRINKROLF
Security Researcher
in