{JS-ON: Security-OFF}: Abusing JSON-Based SQL to Bypass WAF

By Noam Moshe

Executive Summary

- Team82 has developed a generic bypass of industry-leading web application firewalls (WAF).
- The attack technique involves appending JSON syntax to SQL injection payloads that a WAF is unable to parse.
- Major WAF vendors lacked JSON support in their products, despite it being supported by most database engines for a decade.
- Most WAFs will easily detect SQLi attacks, but prepending JSON to SQL syntax left the WAF blind to these attacks.
- Our bypass worked against WAFs sold by five leading vendors: Palo Alto Networks, Amazon Web Services, Cloudflare, F5, and Imperva. All five have updated their products to support JSON syntax in their SQL injection inspection process.
- Attackers using this technique would be able to bypass the WAF's protection and use additional vulnerabilities to exfiltrate data.

Introduction

Web application firewalls (WAF) are designed to safeguard web-based applications and APIs from malicious external HTTPs traffic, most notably cross-site scripting and SQL injection attacks that just don't seem to drop off the security radar.

While recognized and relatively simple to remedy, SQL injection in particular is a constant among the output of automated code scans, and a regular feature on industry lists of top vulnerabilities, including the OWASP Top 10.

The introduction of WAFs in the early 2000s was largely a counter to these coding errors. WAFs are now a key line of defense in securing organizational information stored in a database that can be reached through a web application. WAFs are also increasingly used to protect cloud-

based management platforms that oversee connected embedded devices such as routers and access points.

An attacker able to bypass the traffic scanning and blocking capabilities of WAFs often has a direct line to sensitive business and customer information. Such bypasses, thankfully, have been infrequent, and one-offs targeting a particular vendor's implementation.

Today, Team82 introduces an attack technique that acts as the first generic bypass of multiple web application firewalls sold by industry-leading vendors. Our bypass works on WAFs sold by five leading vendors: Palo Alto, F5, Amazon Web Services, Cloudflare, and Imperva. All of the affected vendors acknowledged Team82's disclosure and implemented fixes that add support for JSON syntax to their products' SQL inspection processes.

Our technique relies first on understanding how WAFs identify and flag SQL syntax as malicious, and then finding SQL syntax the WAF is blind to. This turned out to be JSON. JSON is a standard file and data exchange format, and is commonly used when data is sent from a server to a web application.

JSON support was introduced in SQL databases going back almost 10 years. Modern database engines today support JSON syntax by default, basic searches and modifications, as well as a range of JSON functions and operators. While JSON support is the norm among database engines, the same cannot be said for WAFs. Vendors have been slow to add JSON support, which allowed us to craft new SQL injection payloads that include JSON that bypassed the security WAFs provide.

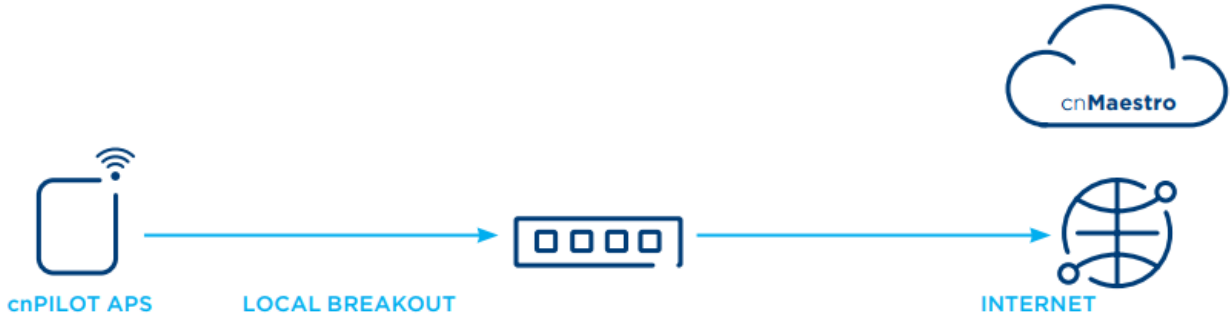
Attackers using this novel technique could access a backend database and use additional vulnerabilities and exploits to exfiltrate information via either direct access to the server or over the cloud.

This is especially important for OT and IoT platforms that have moved to cloud-based management and monitoring systems. WAFs offer a promise of additional security from the cloud; an attacker able to bypass these protections has expansive access to systems.

Previous Work Leads to New Technique

Our journey to developing this technique began last year during unrelated research on Cambium Networks' wireless device management platform, including its `cnMaestro` wireless network manager that is sold either on-premises or in the cloud.

A wireless access point made by Cambium Networks.



cnMaestro cloud architecture allows users to configure and control their AP wifi devices remotely, from the cloud.

In order to understand how the platform is built and many of its internal APIs and routes, we downloaded an Open Virtualization Format virtual machine of cnMaestro's on premises deployment from Cambium's website.

We learned that cnMaestro is built from many different NodeJS backend services that handle users' requests to specific routes. Each of those services is lightly obfuscated to make researching the platform difficult. In order to proxy each request to the correct service, Nginx is used to pass the requests by the requested URL.

cnMaestro offers two different deployment types:

1) **On-Premise Deployment:** A dedicated cnMaestro server is created that is hosted and managed by the user.
2) **Cloud Deployment:** A cnMaestro server hosted on Cambium Networks' cloud infrastructure; all such instances of cnMaestro are hosted on Amazon AWS' cloud under Cambium's organization in a multi-tenant architecture.

When we started fiddling with the cnMaestro application, we noticed a few interesting things with regard to the cloud deployment.

Cloud Deployment

cnMaestro cloud deployments hosted on Amazon's AWS include a main instance of cnMaestro (hosted on https://cloud.cambiumnetworks.com) that handles logins, device deployment, and saves most of the platform's data inside a main database.

Any user who registers to the cnMaestro Cloud application is given a personal Amazon AWS instance, with a personal URL (a sub-domain of Cambium's main cloud), and an organizational identifier. This helps to separate the different users in a multi-tenant design. In order to access your cnMaestro instance, a unique URL is generated following this scheme:

```
https://us-e1-sXX-XXXXXXXXXX.cloud.cambiumnetworks.com
```

At the end of our research into Cambium cnMaestro, we discovered seven different vulnerabilities, which can be seen here and on Team82's Disclosure Dashboard. However, one vulnerability in particular made us go down a huge rabbit hole that led us into discovering and developing this new technique.

Getting Stuck With a Zero Day You Can't Exploit

One particular Cambium vulnerability we discovered proved more difficult to exploit: CVE-2022-1361. At the core of the vulnerability is a simple SQL injection vulnerability, however the actual exploitation process required us to think outside the box and create a whole new SQL technique. Using this vulnerability, we were able to exfiltrate users' sessions, SSH keys, password hashes, tokens, and verification codes.

The core issue of this vulnerability was that in this particular case, the developers did not use a prepared statement to append user-supplied data to a query. Instead of using a safe method of appending user parameters into an SQL query and sanitizing the input, they simply appended it to the query directly.

```
function a(i, a, r, o) {
    var d;
    d = a.serialNo ? '\'{"serial_no":"' + a.serialNo + "\"}'" : '\'{"mac":"' + a.mac + "\"}'", utils.dbQuery(e, "SELECT DISTINCT device_id from
    device_history WHERE EXTRACT(EPOCH FROM timestamp) < $1 AND data @> " + d, [r], function (e, a) {
        if (e) return o(e);
        var d = [];
        a.rows.forEach(function (e) {
            d.push(_.partial("history" === i ? t : n, e.device_id, r))
        }), async.parallel(d, function (e, i) {
            return e ? o(e) : o(null, i)
        })
    })
}
```

The SQL Injection sink point we abused in [CVE-2022-1361](CVE-2022-1361).

As we can see in the sink point above, the application takes user-supplied data (in this case: a.serialNo or a.mac) and appends it to a SQL query. Our goal using this vulnerability was to exfiltrate sensitive data stored in the database. However, while this seemed simple enough, after a quick analysis of this vulnerability we realized it had three key weaknesses/limitations:

1) We can only retrieve integers as the returned rows
2) The returned rows are returned in random order
3) We can only return a limited number of rows in each request.

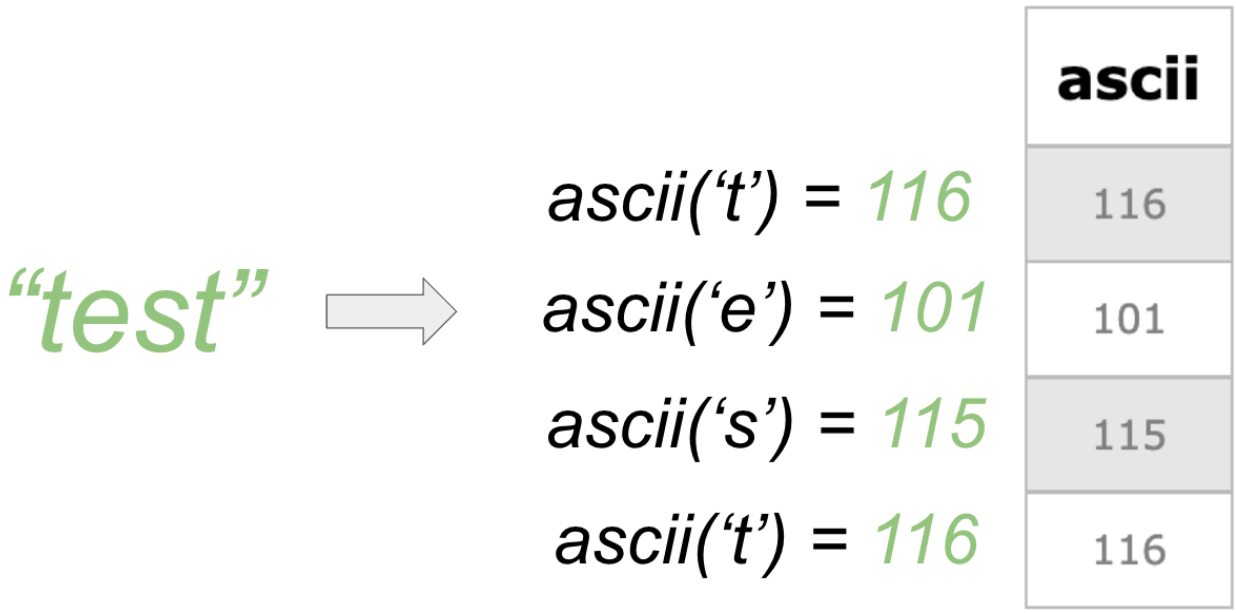
Let's analyze the limitations in-depth.

Limitation 1: We Can Only Retrieve Integers

The first limitation returns only integers, and not strings. Since the original request returns integers, any union statement we will use must also return integers. In SQL, if you perform a union operation, you must make sure both columns are of the same type, and since one side fetched integers, we had to return integers as well. Since the data that we will want to exfiltrate will most likely be strings (session tokens, SSH keys etc.), we must somehow gain the ability to exfiltrate strings.

This limitation was easily overcome by casting any string we want to exfiltrate into an integer array, returning each character as a separate row. To do so, we used the `string_to_array` and `ASCII` SQL functions.

```
SELECT ASCII(c) FROM unnest(string_to_array('test',NULL)) AS c;
```



A SQL query returning a string as an integer list of its characters.

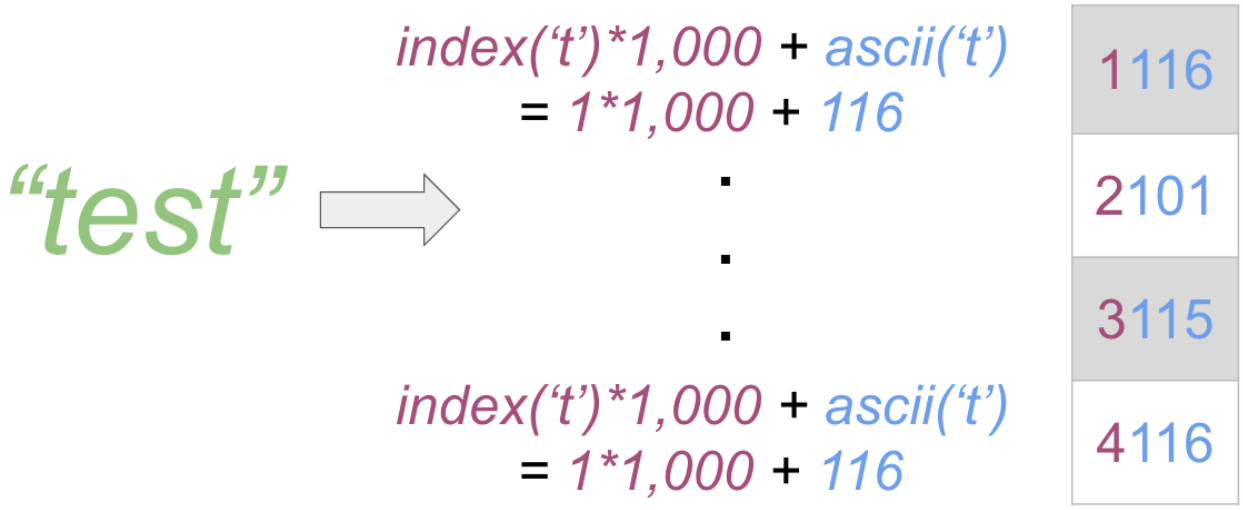
Limitation 2: The Returned Rows Are Returned In Random Order

The second limitation was that when we return multiple rows, the web server will return it to us in random order. When we looked at the code that is executed after the vulnerability, we saw that for each row that the SQL query returned, the server will perform a few other asynchronous actions (which can be seen by the `async.parallel` function being called). This means that the original order of the returned rows will not be kept, instead the order will be the order of the asynchronous action being finished.

This meant that if we were to exfiltrate a string as an integer array, we would lose the character order thus rendering the exfiltration irrelevant.

We managed to overcome this limitation by appending the row index, which translates the index of the character in the string to the returned integer, using the `row_number` SQL function. Because we only return ASCII characters, each character value is limited to 128. By adding the index number multiplied by a thousand (`i * 1000`) and appending it to the result, we can always be sure of the character index using a simple division and module actions.

```
SELECT (c + 1000 * index) FROM (SELECT ASCII(c) AS c,row_number()
over() AS index FROM unnest(string_to_array('test',NULL)) c) AS aa;
```



A SQL payload that returns the ascii value of each letter in a string, with the character's index multiplied by 1,000.

After we retrieve the exfiltrated data, we can simply divide each returned row by a thousand in order to know the character index. We can also recover the original character ASCII value by using the module action on the returned value.

Limitation 3: We Can Only Return a Limited Number Of Rows In Each Request

The final limitation was the most difficult to overcome: a timeout issue. For each row we returned, the server performed a few other actions, including another SQL query and data manipulation. When we tried to retrieve a large number of rows, the request was timed out. To make matters worse, the API endpoint was fairly slow, so retrieving one row at a time was too time consuming.

Our solution was actually very elegant: instead of returning one row for each character, we would instead construct an integer out of many rows. This is possible because of the difference in byte size between integers and characters. In PostgreSQL, an integer is 4 bytes long, while the character we try to exfiltrate is up to 1 byte long (as long as we are talking about ascii characters). This means that by performing simple byte operations, we can house four different

characters in each integer. Furthermore, if we cast our integer into a `BIGINT` in our union command, which is possible to do in PostgreSQL, we can expand each row into 8 bytes.

8.1. Numeric Types

8.1.1. Integer Types
8.1.2. Arbitrary Precision Numbers
8.1.3. Floating-Point Types
8.1.4. Serial Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals. **Table 8.2** lists the available types.

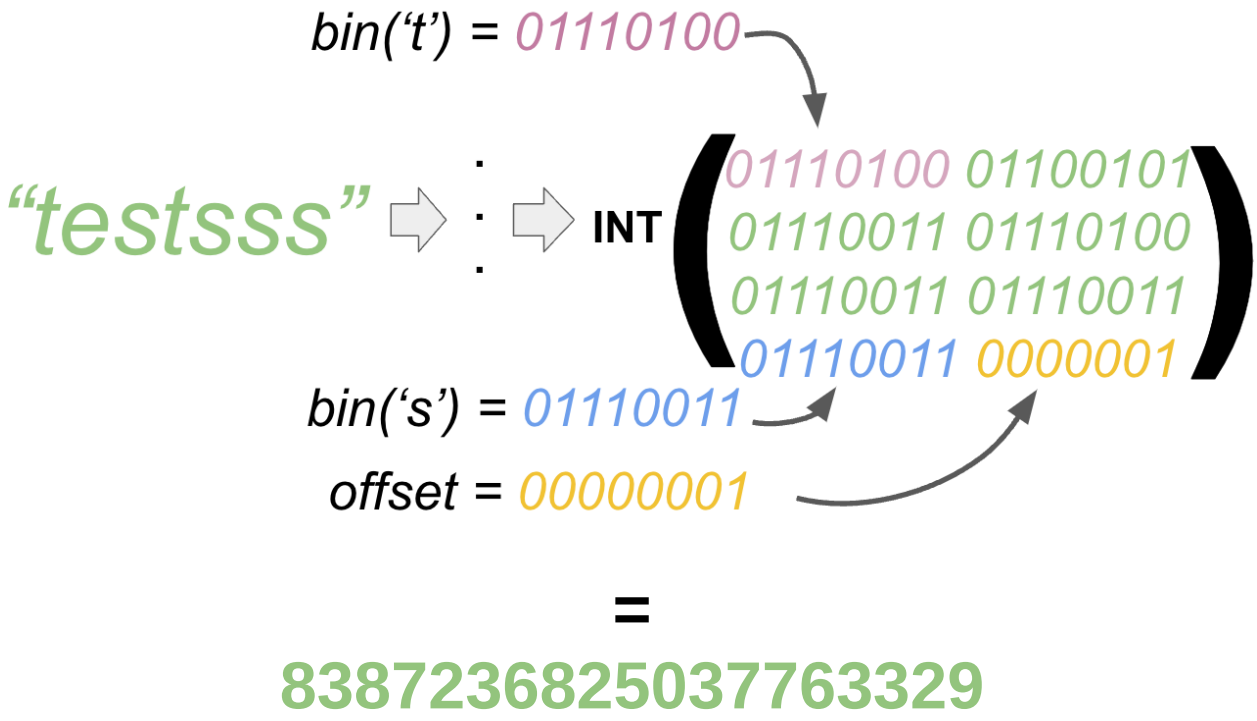
Table 8.2. Numeric Types

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

PostgreSQL types sizes, taken from [PostgreSQL site](#).

This means that if we were to append 8 bytes for each character we exfiltrate, and append it into a BIGINT, we could exfiltrate 7 times more characters in each request (1 byte is reserved to the character index).

```
SELECT (id+num) FROM (SELECT ((ASCII(a[7])::BIGINT<<8) + ASCII(a[6])::BIGINT<<16) +
(ASCII(a[5])::BIGINT<<24) + (ASCII(a[4])::BIGINT<<32) + (ASCII(a[3])::BIGINT<<40) +
(ASCII(a[2])::BIGINT<<48) + (ASCII(a[1])::BIGINT<<56)) AS num,row_number() over()
AS id FROM regexp_matches((SELECT 'testsss'),'(.)(.)(.)(.)(.)(.)(.)','g') AS a) bb
```



A SQL query that takes a string, and creates a BIGINT out of every few characters.

Using this methodology, we were able to exfiltrate up to 8 times more data in each request. This reduced the time it would take us to exfiltrate a meaningful amount of data and make the attack scenario plausible.

Constructing Our Payload

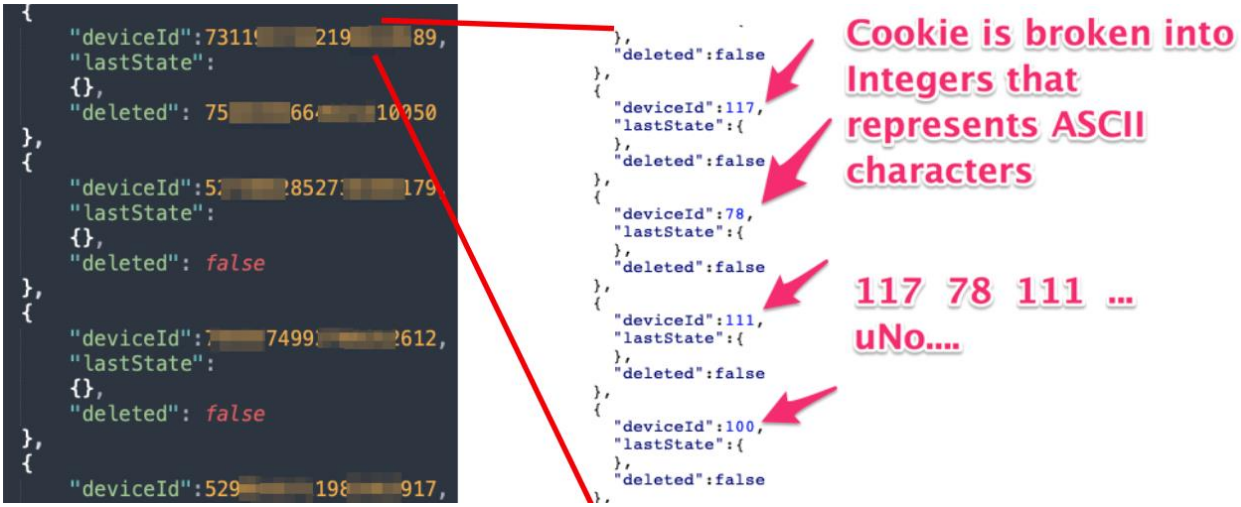
After we bypassed all three limitations, we were left with a big payload allowing us to extract any data we chose:

```
' and '{"C":2}' <@ '{"a":1, "b":2}' union select (id+num) from
(select ((ASCII(a[1])::BIGINT<<8) + (ASCII(a[2])::BIGINT<<16) +
(ASCII(a[3])::BIGINT<<24) + (ASCII(a[4])::BIGINT<<32) +
(ASCII(a[5])::BIGINT<<40) + (ASCII(a[6])::BIGINT<<48)) as
num,row_number() over() as id from regexp_matches((select 'this
is a test'),'(.)(.)(.)(.)(.)(.)','g') as a) bb-- -;
```

Colormap:
WAF Evasion
Solving Limitation 1 - casting strings to integers
Solving Limitation 2 - Exfiltrating character index
Solving Limitation 3 - Exfiltrating multiple characters in a single returned row.

`The actual data we want to exfiltrate`

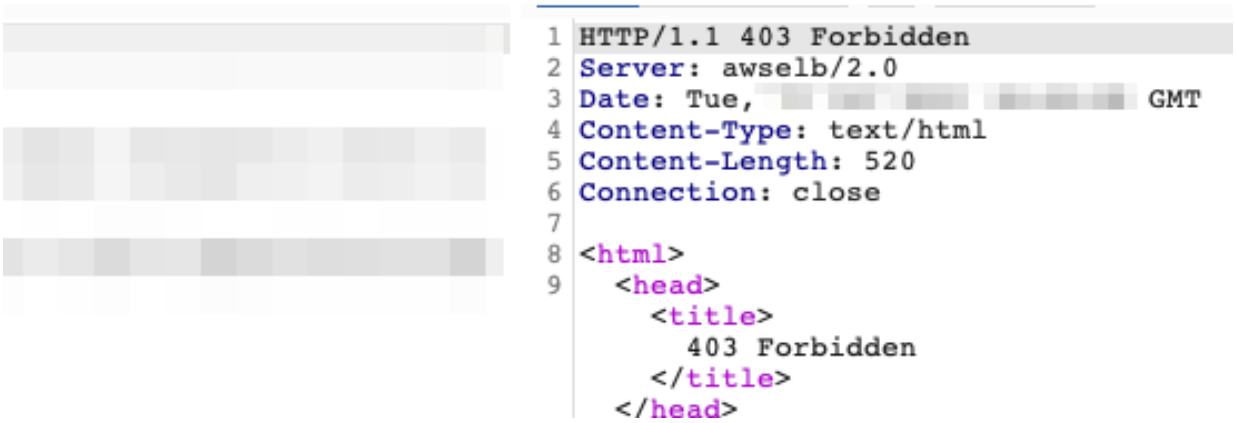
And indeed, when we used this payload we managed to exfiltrate sensitive information stored in the database ranging from session cookies to tokens, SSH keys and hashed passwords.



An example of data we exfiltrated using our SQLi payload.

Going To The Clouds (and Falling Down)

After managing to fully exploit this vulnerability on the on-prem version, our next step was to try the same vulnerability on Cambium's cloud. Soon enough we found the corresponding cloud route, and we managed to confirm it is vulnerable to the same vulnerability. We then tried a safe version of our payload, and we received this response:



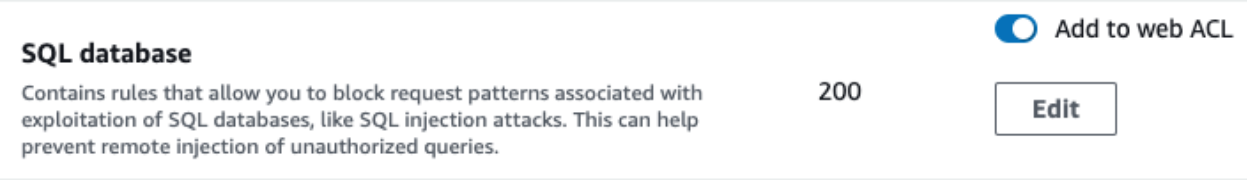
```
HTTP/1.1 403 Forbidden
Server: awselb/2.0
Date: Tue, [redacted] GMT
Content-Type: text/html
Content-Length: 520
Connection: close

<html>
  <head>
    <title>
      403 Forbidden
    </title>
  </head>
```

The response to our SQL Injection vulnerability. We can see that our request was dropped, returning a 403 Forbidden.

After a short panic, we noticed the `HTTP Server` header, containing `awselb/2.0`. This clued us in that our request was not stopped by the application, instead the AWS WAF dropped our request because it probably flagged it as malicious. This stumped us for a minute, however soon enough we set our goal on bypassing this WAF. This ignited our current research.

Researching AWS WAF

In order to research the AWS WAF, we first created our own setup where we control all moving parts: the application, the client and the WAF settings and logs. We created a simple machine on the AWS cloud, and set up the AWS WAF to protect the application from malicious requests (we set up the WAF).



The interface for configuring the WAF ruleset.

Then, we created a web application with a SQLi vulnerability, and hosted it on AWS.

```python
@app.route("/")
def home():
    args = request.args
    p = args.get("password")
    if p is None:
        return f"Hello admin, what is your password?<br><p style='color:red'>No password supplied</p>"

    con = psycopg2.connect(database="bh_playground", user="postgres", password='[redacted]', host="127.0.0.1")
    cur = con.cursor()
    query = f"select * from accounts where username='admin' and password='{p}';"

    try:
        cur.execute(query)
```

SQLi

The vulnerable Flask web application we created.

Lastly, we started sending hundreds of specially crafted requests to try and analyze how the WAF flags requests as malicious.

Requests the WAF flagged as malicious were blocked. In this request we pass a common SQLi payload, which is flagged by the WAF

From our testing, we concluded that in general, there are two methodologies for WAFs to flag a request as malicious:

1) Search for blacklisted words: The WAF can search for words it recognizes as SQL syntax, and if too many matches exist in a request, it will flag the request as a malicious SQLi attempt.
2) Parse SQL syntax from the request: The WAF can try and parse valid SQL syntax using different parts of the request. If the WAF successfully identifies SQL syntax, it will flag the request as a malicious SQLi attempt.

While most WAFs will use a combination of both methodologies in addition to anything unique the WAF does, they both have one common weakness: they require the WAF to recognize the SQL syntax. This triggered our interest and raised one major research question: what if we could find SQL syntax that no WAF would recognize?

JSON in SQL

Our answer came quickly in the form of a (major) SQL feature: JSON. In modern times, JSON has become one of the predominant forms of data storage and transfer. In order to support JSON syntax and allow developers to interact with data in similar ways to how they interact with it in other applications, JSON support was needed in SQL.

Currently all major SQL engines support native JSON syntax; this includes MSSQL, PostgreSQL, SQLite, and MySQL. Furthemore, in the latest versions, all database engines enable JSON syntax by default, meaning it is prevalent in most database setups today.

While each database chose a different implementation and JSON parser, each supports a different range of JSON functions and operators. Also, they all support the JSON data type and basic JSON searches and modifications.

	JSON Support	Enabled by Default	Year JSON Added	JSON Parser Used	Functions \ Operators
PostgreSQL	Yes	Yes	v9.2 (2012)	Proprietary	json_object_keys() #- ?& @>
MySQL	Yes	Yes	v5.7.8 (2015)	RapidJSON	JSON_EXTRACT() JSON_QUOTE() JSON_DEPTH()
SQLite	Yes	Yes	v3.38.0 (2022)	Proprietary	json_quote() json_array_length() ->>
Microsoft SQL Server	Yes	Yes	SQL Server 2016	Proprietary	JSON_QUERY() JSON_PATH_EXISTS ()

A table showcasing the different JSON support of each major database.

However, even though all database engines added support for JSON, not all security tools added support for this “new” feature (which was added as early as 2012). This lack of support in security tools could introduce a mismatch in parsing primitives between the security tool (in our case, the WAF) and the actual database engines, and cause SQL syntax misidentification.

The New ‘ or ‘a’=’a

Using JSON syntax, it is possible to craft new SQLi payloads. These payloads, since they are not commonly known, could be used to fly under the radar and bypass many security tools. Using syntax from different database engines, we were able to compile the following list of true statements in SQL:

- **PostgreSQL**: `'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb` - Is the left JSON contained in the right one? True.
- **SQLite**: `'{"a":2,"c":[4,5,{"f":7}]}' -> '$.c[2].f' = 7` - Does the extracted value of this JSON equals 7? True.
- **MySQL**: `JSON_EXTRACT('{"id": 14, "name": "Aztalan"}', '$.name') = 'Aztalan'` - Does the extracted value of this JSON equals to ‘Aztalan’? True.

Armed With JSON Syntax

From our understanding of how a WAF could flag requests as malicious, we reached the conclusion that we need to find SQL syntax the WAF will not understand. If we could supply a SQLi payload that the WAF will not recognize as valid SQL, but the database engine will parse it, we could actually achieve the bypass.

As it turns out, JSON was exactly this mismatch between the WAF's parser and the database engine. When we passed valid SQL statements that used less prevalent JSON syntax, the WAF actually did not flag the request as malicious.

Here is a malicious SQLi payload, containing JSON syntax. As we can see, the WAF did not flag the request as malicious and did not drop it.

This simple JSON operator, `@>` in this case, which checks whether the right JSON is contained in the left one, above, threw the WAF into a loop and allowed us to supply malicious SQLi payloads, allowing us to bypass the WAF. By simply prepending simple JSON syntax to the start of the request, we were able to exfiltrate sensitive information using our SQLi vulnerability over the cloud!

```
GET /cn-rtr/audit/devices/sn/"}'  and
data @> '{"a":"a"}' union select ASCII(s.token) from unnest(string_to_array((select cookie from cookie limit 1 ),N
ULL)) s(token)--/state?sig=1&timeStamp=50 HTTP/1.1
```

Exploitation of an SQL Injection vulnerability over the cloud.

Dream Big: A Generic WAF Bypass

After demonstrating our bypass over Amazon AWS WAF, we wondered: “Maybe we have a bigger issue at hand?” The core issue of this bypass was a lack of conformance between the database engines and SQLi detection solutions; this is because JSON in SQL is not such a popular and well-known feature, and its syntax was not added to the WAF parser.

However we thought that maybe this issue is not relevant for this WAF vendor alone, maybe other vendors have not added support for JSON syntax as well. So we took our vulnerable web application, and created a setup on most major WAF vendors. After a long few days, we discovered that JSON syntax could be used to bypass most vendors we checked:

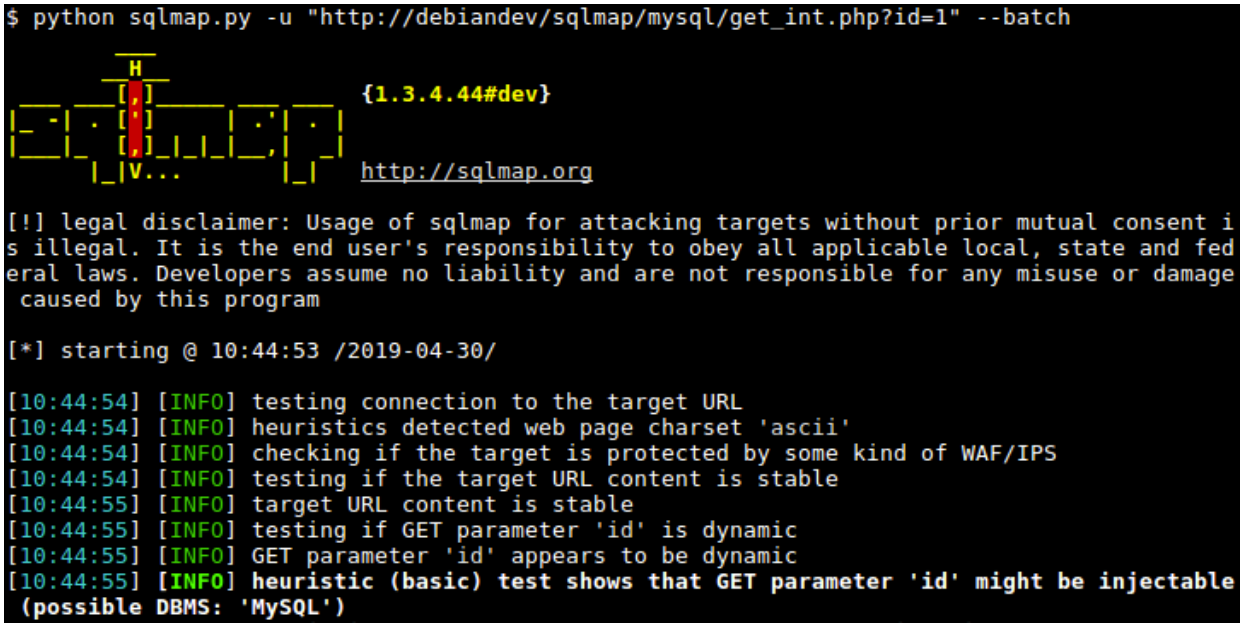
- Palo-Alto Next Generation Firewall
- F5 Big-IP
- Amazon AWS ELB
- Cloudflare
- Imperva

List of WAF vendors and products we managed to bypass using JSON syntax.

The fact we managed to bypass so many big WAF products, with limited if any changes to our payload meant we had a generic WAF bypass on our hands. This means that even without knowing exactly what WAF lies between us and our target, we can still exploit a SQL injection vulnerability, bypassing the WAF's protection.

Automating The Process

In order to showcase how big this WAF bypass is, we decided to add support for JSON syntax evasion techniques to the biggest open-source exploitation tool, SQLMap.

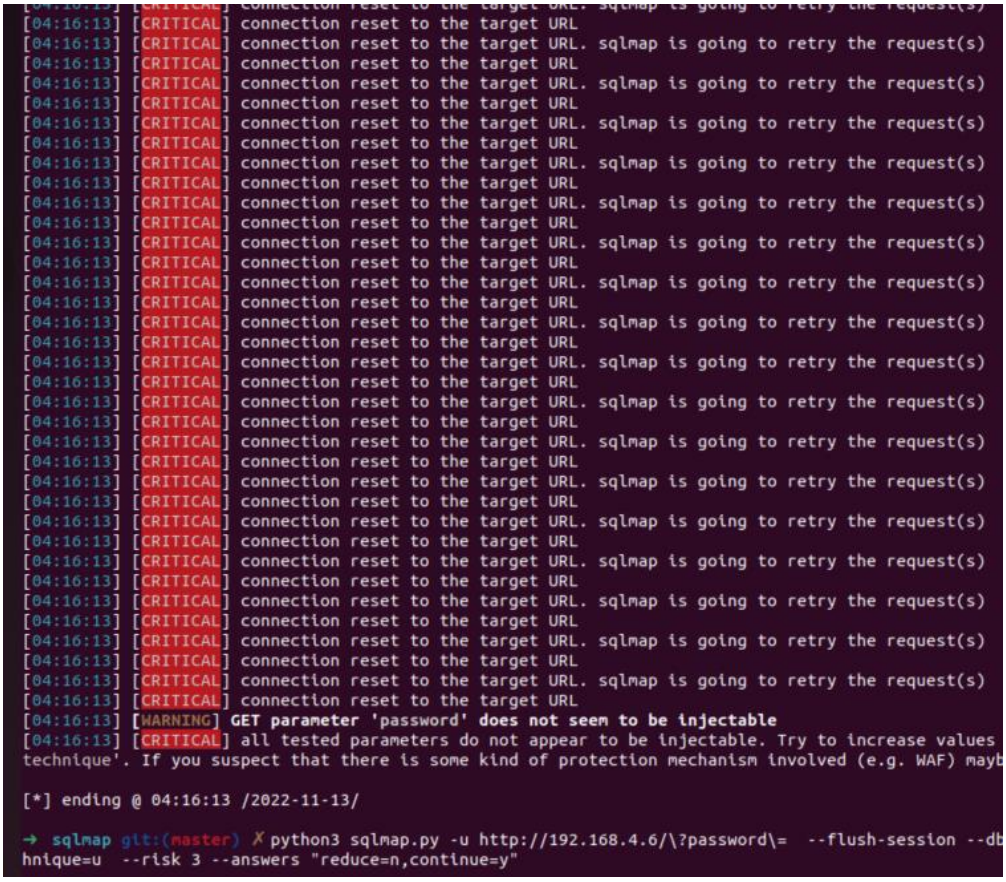


The SQLMap tool, allowing users to automate and attack targets.

SQLMap offers an automatic process of SQL injection exploitation, allowing users to scan entire sites for a vulnerability. After SQLMap identifies a SQL vulnerability, it offers the ability to both fingerprint the vulnerability type and to identify an exploitation technique best suited to this specific vulnerability.

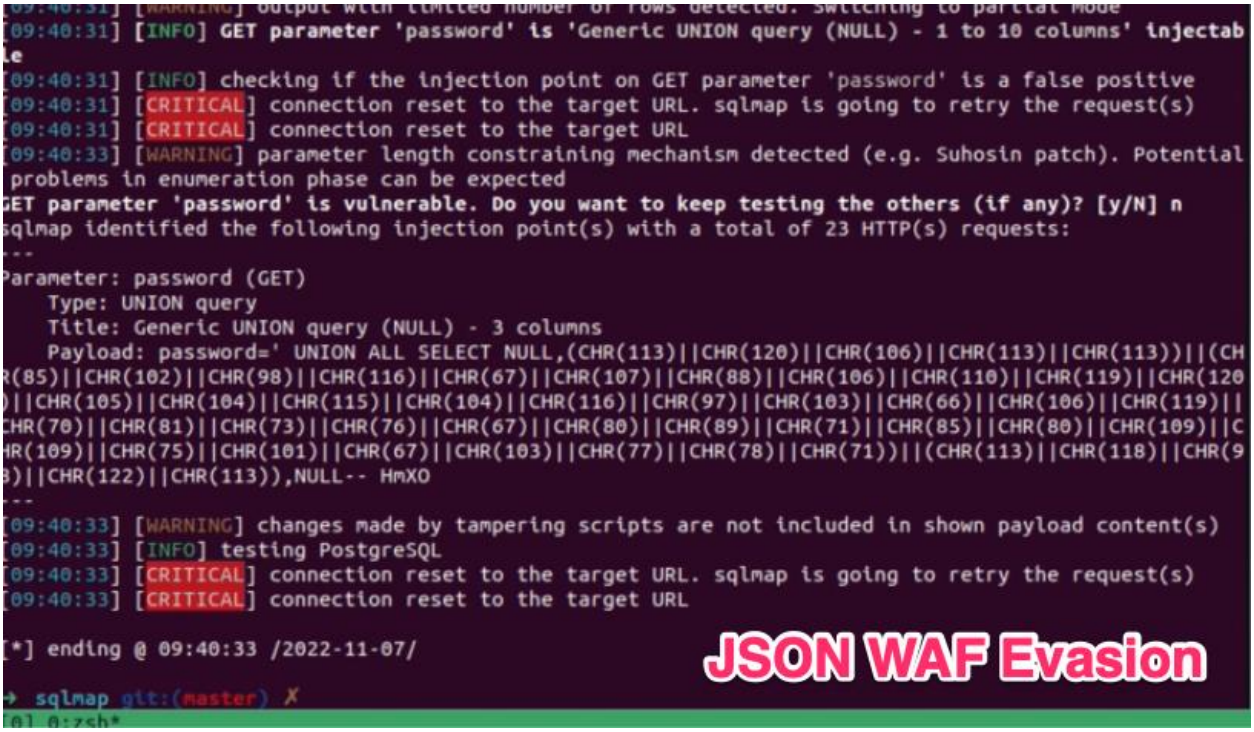
After correctly choosing the technique to exploit the vulnerability, SQLMap even offers users the ability to automatically dump information stored in the database, enumerate tables and databases, exfiltrate password hashes, and perform a few post-exploitation techniques.

While SQLMap offers some WAF evasion techniques, we found that it is still easily identified by most modern WAFs, meaning that users cannot use it in cases where a WAF is present.



Trying to execute SQLMap on an application protected by a WAF. We can clearly see that even though the application is vulnerable, SQLMap does not deem it exploitable since most requests are dropped by the WAF.

Our goal was to bring this new technique into SQLMap, and to use JSON syntax in order to bypass WAFs. In order to do so, we injected payloads generated by SQLMap, adding randomly generated JSON syntax. Since every database engine implemented a different set of JSON functions and operators, we implemented a separate script for each database engine. Using our addition to SQLMap, we were able to bypass a well-known WAF and successfully exploit a vulnerable web application.



Running SQLMap using our script allowed SQLMap to successfully exploit the vulnerable web application and to bypass the WAF.

If you would like to use this script in order to test this bypass, simply clone the latest version of SQLMap from Github.

Conclusion

Team82's novel attack technique effectively bypasses the ability of a web application firewall to adequately detect SQL injection attacks. We did so through a complex journey that began with unrelated research that was being thwarted by a web application firewall, setting off a chain of events leading to our generic WAF bypass.

We discovered that the leading vendors' WAFs did not support JSON syntax in their SQL injection inspection process, allowing us to prepend JSON syntax to a SQL statement that blinded a WAF to the malicious code.

Team82 disclosed its findings to five of the leading WAF vendors, all of which have added JSON syntax support to their products. We believe that other vendors' products may be affected, and that reviews for JSON support should be carried out. Below are Amazon's and F5's acknowledgements and fixes, for example.

SQL database • `SQLi_BODY` • `SQLi_QUERYARGUMENTS` • `SQLi_COOKIE` • `SQLi_URIPATH` • `SQLiExtendedPatterns_BODY` • `SQLiExtendedPatterns_QUERYARGUMENTS`	Released version 2.0 of this rule group. Replaced the `URL_DECODE` text transformation with the double `URL_DECODE_UNI` text transformation and added the `COMPRESS_WHITE_SPACE` text transformation. Added more detection signatures to `SQLiExtendedPatterns_QUERYARGUMENTS`. Added JSON inspection to `SQLi_BODY`. Added the rule `SQLiExtendedPatterns_BODY`. Removed the rule `SQLi_URIPATH`.	2022-01-10

Part of the release notes of Amazon AWS ELB ruleset, adding support for JSON syntax in SQLi inspection and blocking this bypass.

The F5 Security Incident Response Team (**F5 SIRT**) is pleased to recognize the security researchers who have helped improve attack signatures for Advanced WAF/ASM/NGINX App Protect by finding and reporting ways to bypass certain attack signature checks. Each name listed represents an individual or company who has privately disclosed one or more bypass methods to us. The attack signature IDs listed are the attack signatures that F5 adds to or updates in the new attack signature update files based on the researcher's report.

2022 Acknowledgments

Name	Attack Signature Update Files	Attack Signature IDs
Noam Moshe of Claroty Research	ASM-SignatureFile_20220315_113554.im ASM-AttackSignatures_20220315_113554.im	200102058 200102059 200102060 200102061 200102062 200102063

Part of the F5 SIRT Security Acknowledgment of F5 BIG-IP, adding support for JSON syntax in SQLi inspection.

This is a dangerous bypass, especially as more organizations continue to migrate more business and functionality to the cloud. IoT and OT processes that are monitored and managed from the cloud may also be impacted by this issue, and organizations should ensure they're running updated versions of security tools in order to block these bypass attempts.