NANYANG TECHNOLOGICAL UNIVERSITY

Tutorial 9: NoSQL Injection & Taint Analysis

presented by

Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

NoSQL Injection

```
db.users.find({username: username, password: password});
```

" … with MongoDB we are not building queries from strings, so traditional SQL injection attacks are not a problem. "

- MongoDB Developer FAQ

NoSQL Databases

- NoSQL: generic name for database systems that employ data structures different from those used in relational databases, e.g., key-value pairs
  - Makes 'simple' operations faster
  - Popular in 'Big Data' and modern web applications
- Many choice of NoSQL database systems around
- 'NoSQL ecosystem':
  - NoSQL databases – mongoDB, redis, memcached, etc.
  - Server-side runtime environments, e.g. NodeJS, PHP, Python, Ruby, accepting client inputs and preparing database queries
  - Frameworks, e.g., Mongoose between NodeJS and MongoDB, support modelling of data structures, etc.

https://www.crondose.com/2016/05/nosql-databases-developers-guide/

NoSQL vs. SQL


```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

NoSQL has No SQL Injection Problem!

- Does not build queries from strings; the way SQL injection was performed does not work here
- Queries are represented as BSON (Binary JSON) objects:

```
$collection->find(array(
    "username"=>$_GET['username'],
    "passwd"=>$_GET['passwd']));
```

- Still, plenty of opportunities for repeating old mistakes
- The attacker has to find a way of inserting malign objects into queries
- Examples focus on mongoDB; for similar attacks on other DBs, see https://www.owasp.org/images/e/ed/GOD16-NOSQL.pdf

https://www.idontplaydarts.com/2010/07/mongodb-is-vulnerable-to-sql-injection-in-php-at-least/

Example: NoSQLi in PHP

- A NoSQL query (to MongoDB) in PHP:

```
$collection->find(array("username" => $_GET['username'],
                        "passwd" => $_GET['passwd']));
```

- Instead of a string, attacker passes an object:

```
login.php?username=admin&passwd[$ne]=1
```

- Result:

```
$collection->find(array("username" => "admin",
                        "passwd" => array("$ne" => 1)));
```

- Returns array entry for user 'admin' as long as that user's password is not "1"
- Equivalent to SQL:

```
SELECT * FROM collection WHERE username="admin" AND
passwd != 1
```

https://www.owasp.org/images/e/ed/GOD16-NOSQL.pdf

MongoDB – Operators

Operators such as equals to and greater than are encoded as object structure

```
{"$gt":"1"}
```

Name	Description
`$eq`	Matches values that are equal to a specified value
`$ne`	Matches values that are not equal to a specified value
`$gt`	Matches values that are greater than a specified value
`$where`	Matches documents that satisfy a JavaScript expression
...	

Image courtesy of https://www.owasp.org/images/e/ed/GOD16-NOSQL.pdf

Query String Parsing

Query String	Resulting Object
`?param=foo`	`{"param": "foo"}`
`?param[]=foo&param[]=bar`	`{"param": ["foo", "bar"]}`
`?param[foo]=bar`	`{"param": {"foo" : "bar"}}`
`?param[foo][bar]=baz`	`{"param": {"foo" : {"bar" : "baz"}}}`
…	

In NodeJS and PHP

https://entwicklertag.de/karlsruhe/2017/sites/entwicklertag.de.karlsruhe.2017/files/folien/PresentationNoSQLiPatrick.pdf

Example: NoSQLi in ExpressJS

- NoSQL injection vulnerability with Express.js:

```
db.collection('users').find({
  "user": req.query.user, "password": req.query.password
});
```

- Query string parsing returns objects for arrays

```
?param[foo]=bar  →  {"param": {"foo" : "bar"}}
```

- Attack: send HTTP request

```
GET /login?user=alice&password[%24ne]=
```

  - `password[%24ne]=` becomes `password:{"$ne":null}`
  - `%24` is url-encoding of '$'

https://zanon.io/posts/nosql-injection-in-mongodb

NoSQL Injection – Defences

- Use frameworks or provided security APIs: e.g., secure BSON query assembly tool in MongoDB
- Sanitize inputs: e.g., mongo-sanitize strips out all keys starting with '$' (deactivate operators like `$eq`, `$ne`, `$ge`)
- Declare input type: set properties to be of type string; an object passed as input will be converted to a string
  - Works against first attack, e.g., this is safe:

```
$collection->find(array(
"username"=>(string)$_GET['username'],
"passwd"=>(string)$_GET['passwd']));
```

  - but not against `$where` attack
- Safe programming: never use `$where`; check inputs directly with operators like `$eq`, `$ne`, `$ge` – Avoid a layer of indirection!

https://zanon.io/posts/nosql-injection-in-mongodb

Mongo-sanitize in NodeJS

- This is safe:

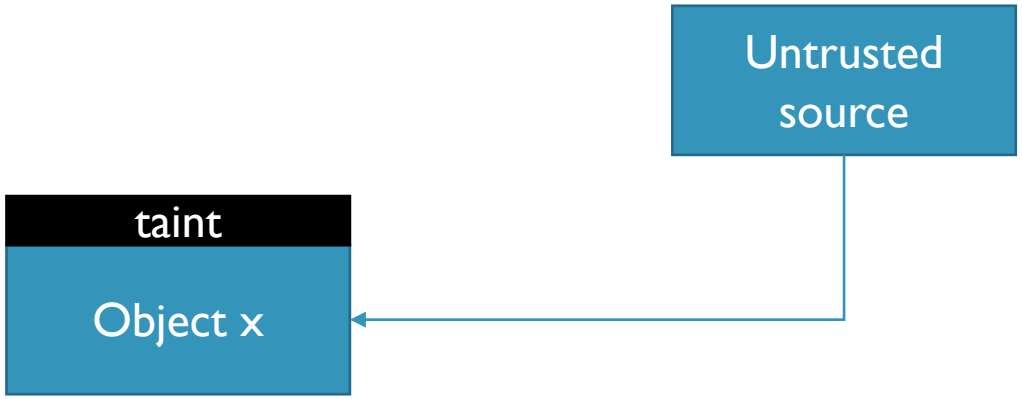
```
var sanitize = require('mongo-sanitize');

app.post('/user', function (req, res) {
 var query = {
      username: sanitize(req.body.username),
      password: sanitize(req.body.password)
  }
 db.collection('users').
      findOne(query, function (err, user) {
           console.log(user);
      });
});
```

Taint Analysis

Taint

- To "taint" user data is to insert some kind of tag or label for each object of the user data
- The tag allow us to track the influence of the tainted object along the execution of the program
- Two states of the tag:
  - T: tainted
  - N: not tainted



Source and Sink?

```
1 a = read();
2 c = read();
3 if (a.equals("hello")) {
4    b = a + "world";
5 } else {
6    a = sanitize(c);
7 }
8 query(c);
9 query(b);
```

Tainted sources
Lines 1, 2

Sensitive sinks
Lines 8, 9

Is this for integrity or confidentiality?

Dynamic Taint Analysis

```
1 a = read();
2 c = read();
3 if (a.equals("hello")) {
4    b = a + "world";
5 } else {
6    a = sanitize(c);
7 }
8 query(c);
9 query(b);
```

Line	a		b		c	
	Value	Taint	Value	Taint	Value	Taint
1	"4067"	T	⊥	N	⊥	N
2	"4067"	T	⊥	N	"attack"	T
4						
6						
8						
9						

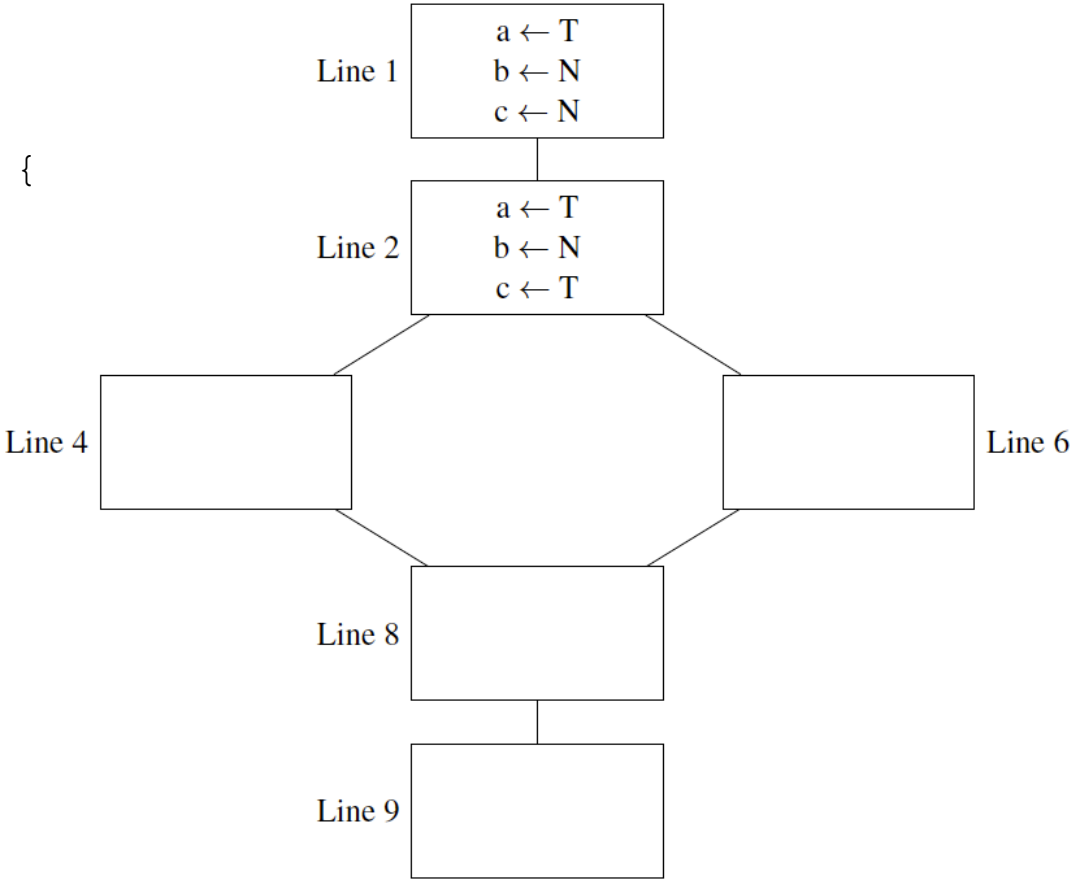
Dynamic Taint Analysis

```
1 a = read();
2 c = read();
3 if (a.equals("hello")) {
4    b = a + "world";
5 } else {
6    a = sanitize(c);
7 }
8 query(c);
9 query(b);
```

Line	a		b		c	
	Value	Taint	Value	Taint	Value	Taint
1	"4067"	T	⊥	N	⊥	N
2	"4067"	T	⊥	N	"attack"	T
4	-	-	-	-	-	-
6	S("attack")	N	⊥	N	"attack"	T
8	S("attack")	N	⊥	N	"attack"	T
9	S("attack")	N	⊥	N	"attack"	T

Static Path-Sensitive Taint Analysis

```
1 a = read();
2 c = read();
3 if (a.equals("hello")) {
4     b = a + "world";
5 } else {
6     a = sanitize(c);
7 }
8 query(c);
9 query(b);
```



Static Path-Sensitive Taint Analysis

```
1 a = read();
2 c = read();
3 if (a.equals("hello")) {
4     b = a + "world";
5 } else {
6     a = sanitize(c);
7 }
8 query(c);
9 query(b);
```