RESTing On Your Laurels Will Get You Pwned

By Abraham Kang, Dinis Cruz, and Alvaro Muñoz

Goals and Main Point

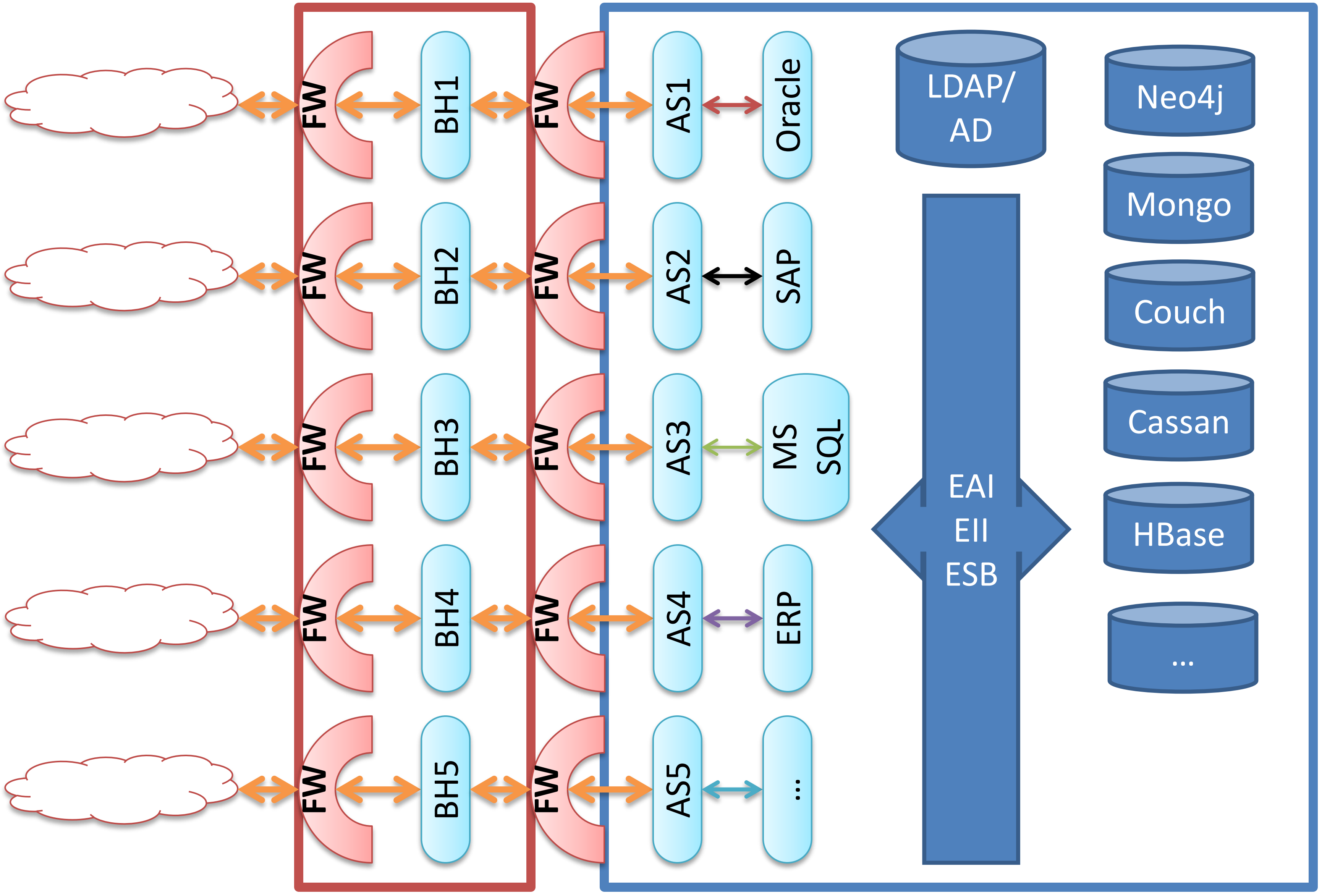
- Originally a 2 hour presentation so we will only be focusing on identifying remote code execution and data exfiltration vulnerabilities through REST APIs.
- Remember that a REST API is nothing more than a web application which follows a structured set of rules.
  - So all of the previous application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

- If you have both publically exposed and internal REST APIs then you probably have some remote code execution and data exfiltration issues.

Causes of REST Vulnerabilities

- Location in the trusted network of your data center
- History of REST Implementations
- Self describing nature
- Input types and interfaces
- URLs to backend REST APIs are built with concatenation instead of URIBuilder (Prepared URI)
- Inbred Architecture
- Extensions in REST frameworks that enhance development of REST functionality
- Reliance on incorrectly implemented protocols (SAML, XML Signature, XML Encryption, etc.)
- Incorrect assumptions of application behavior

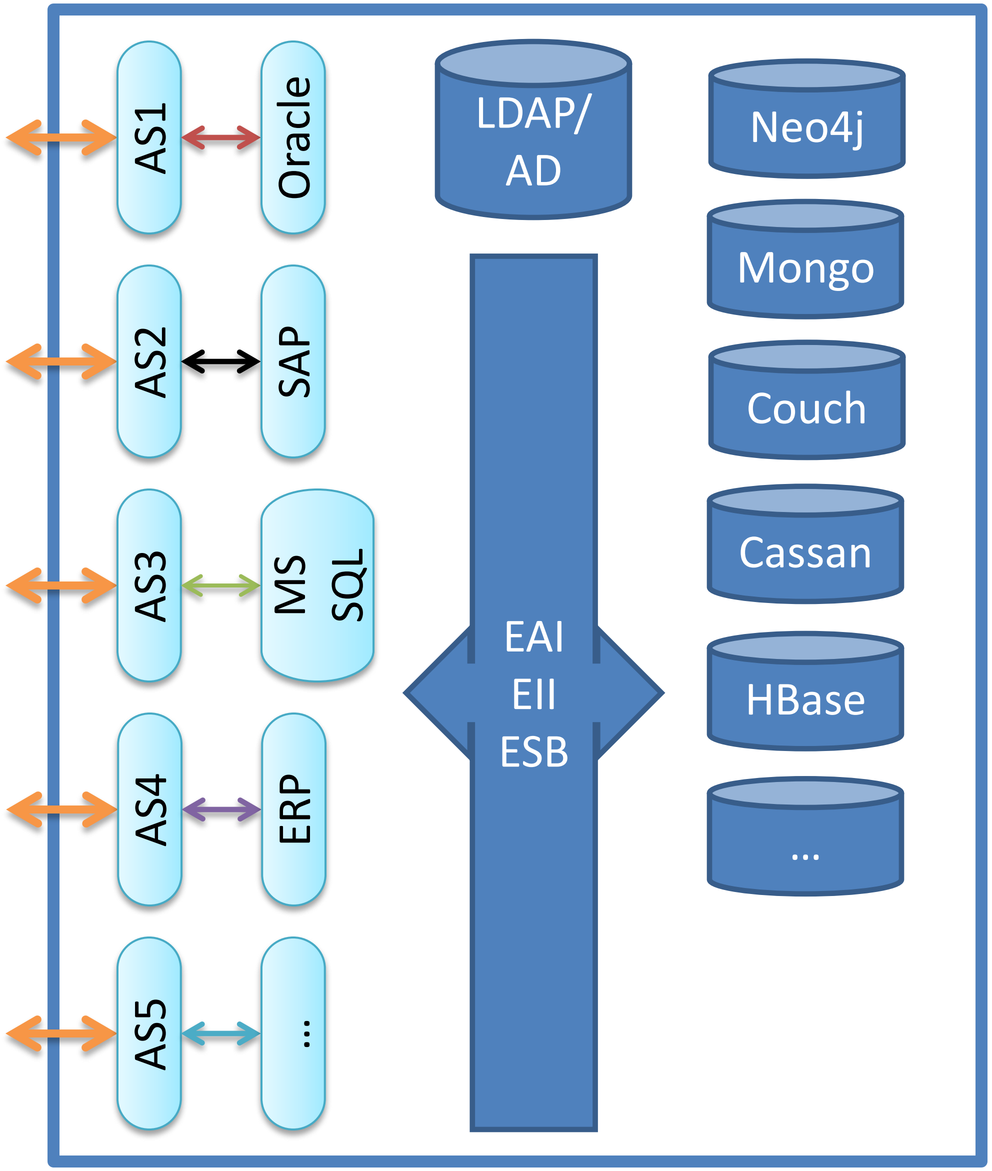
Application Architecture Background



⟷ Http Protocol (proprietary protocols are different colors)

Internal Network of a Data Center

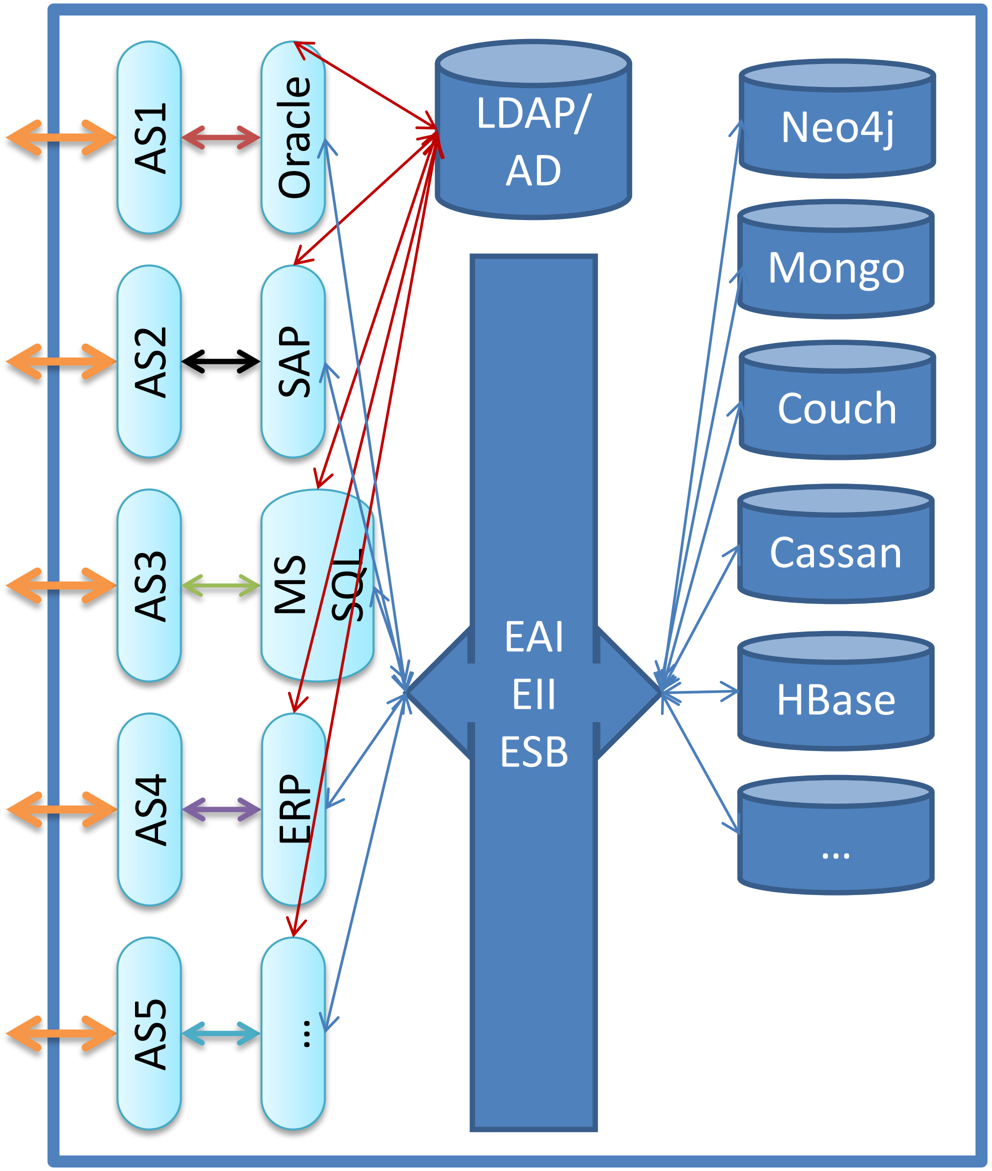
What are the characteristics of an Internal Network (BlueNet, GreenNet, Trusted Network)?



Internal Network of a Data Center

What are the characteristics of an Internal Network (BlueNet, GreenNet, Trusted Network)?

- Connectivity Freedom (within the trusted network)
- Increased Physical Safe guards
- Hardened Systems at the OS level
- Shared Services and Infrastructure



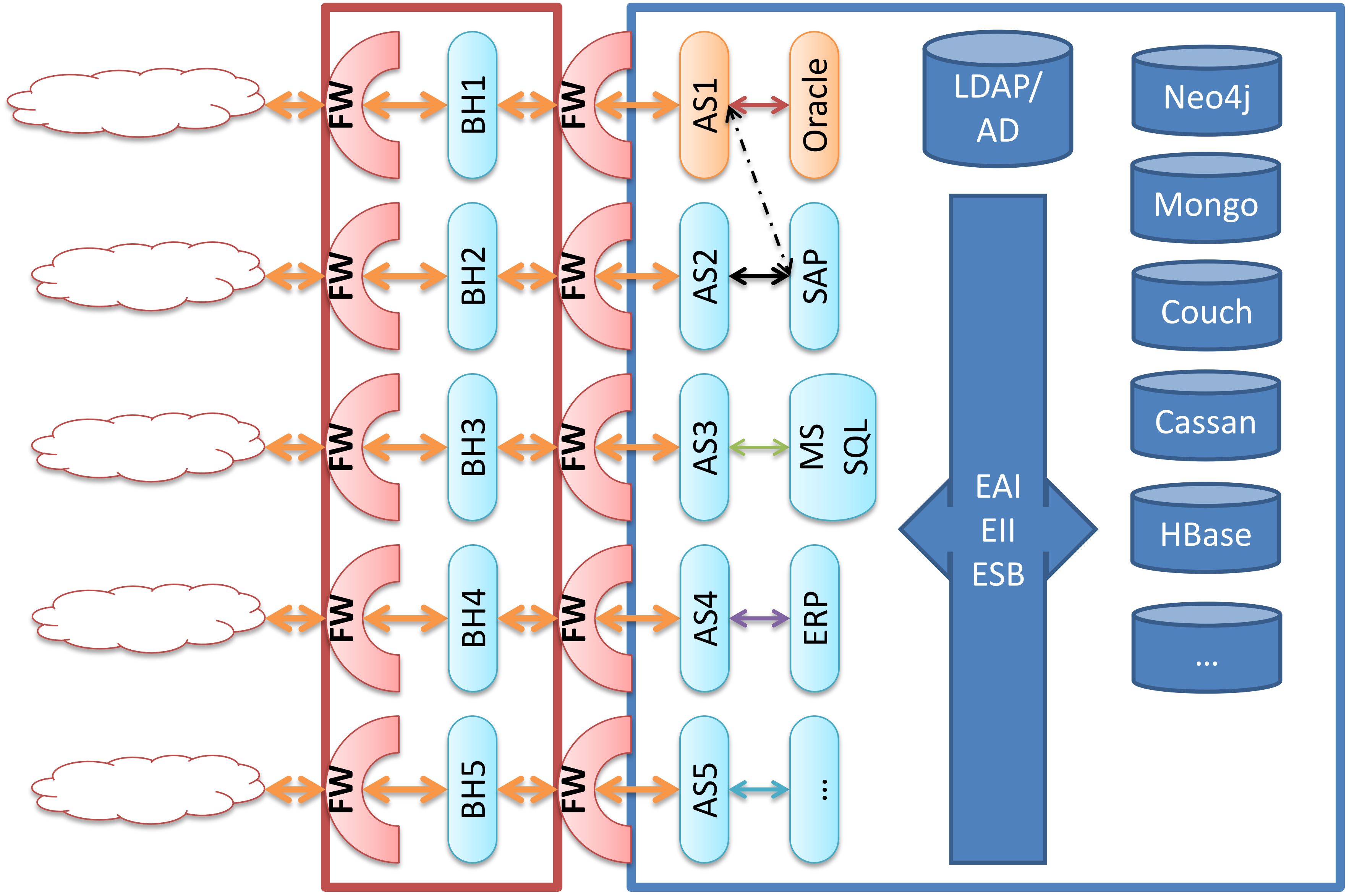
REST History

- Introduced to the world in a PHD dissertation by Roy Fielding in 2000.
- Promoted the idea of using HTTP methods (PUT, POST, GET, DELETE) and the URL itself to communicate additional metadata as to the nature of an HTTP request.
  - PUT = Update
  - POST = Insert
  - GET = Select
  - DELETE = Delete
    - Allowed the mapping of DB interactions on top of self descriptive URLs

REST History (con't)

- 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

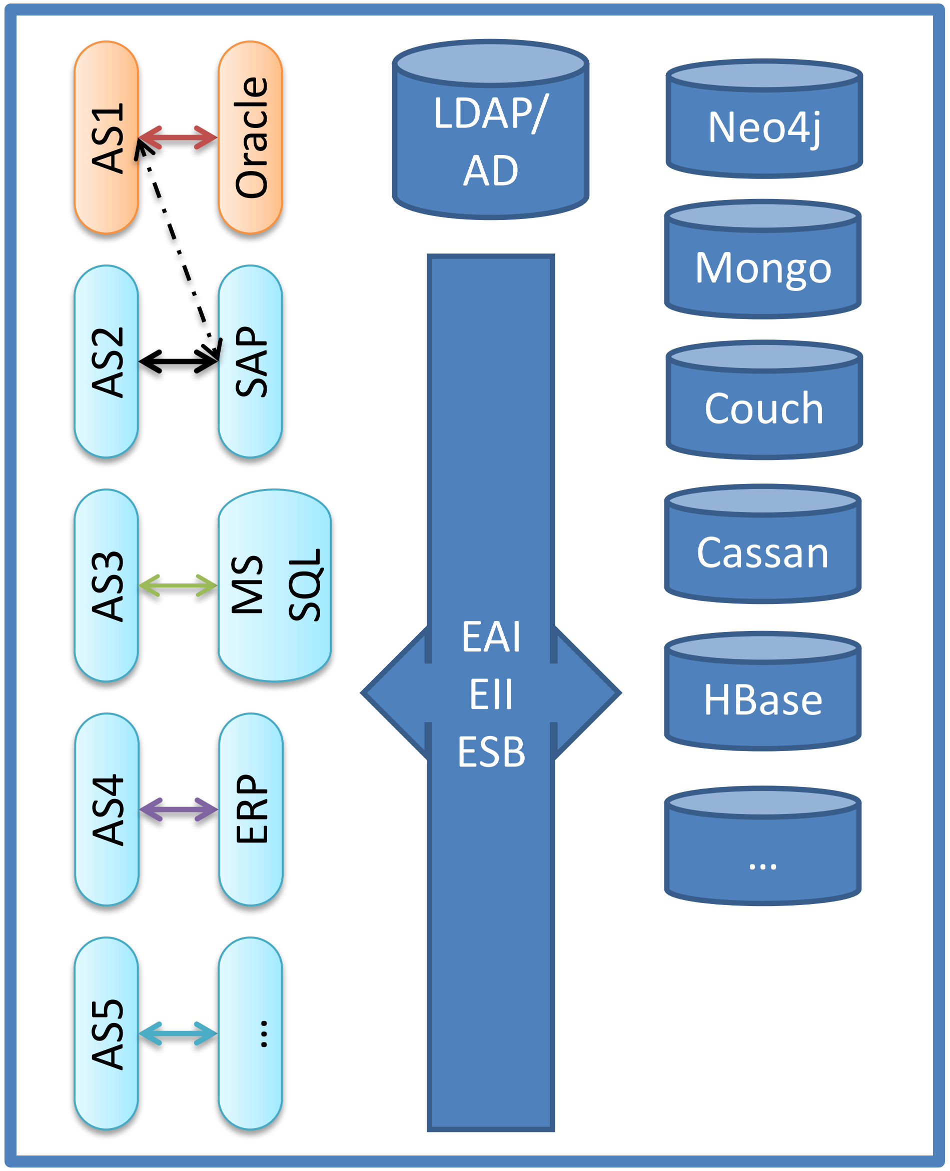
Attacking Backend Systems (Old School)



⟷ Http Protocol (proprietary protocols are different colors)

Attacking An Internal Network (Old School)

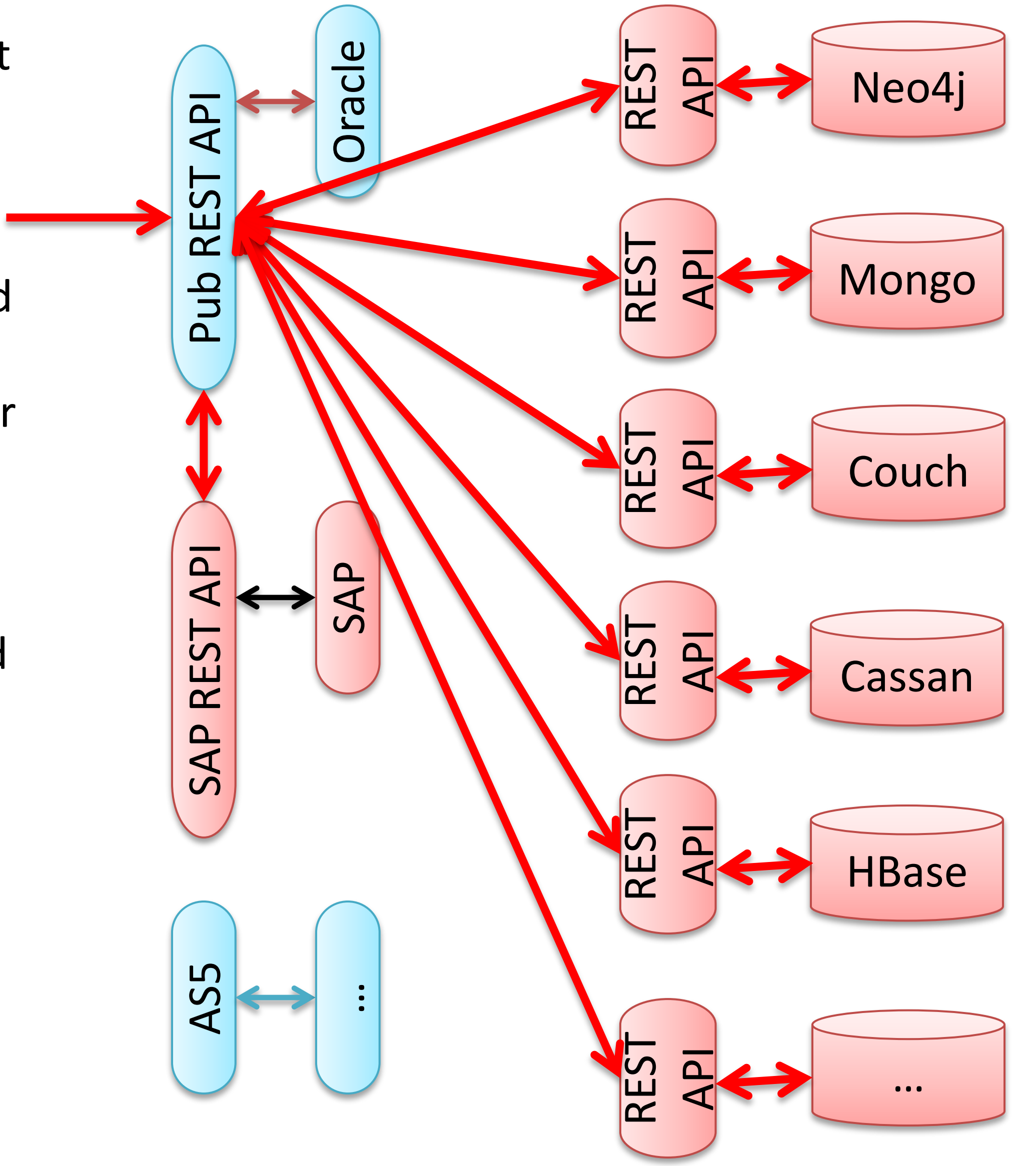
- Pwn the application server
- Figure out which systems are running on the internal network and target a data rich server. (Port Scanning and Fingerprinting)
- Install client protocol binaries to targeted system (in this case SAP client code) so you can connect to the system.
- Figure out the correct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

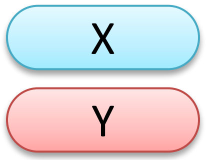


X	Non-compromised machine
Y	Compromised/Pwned machine

Attacking An Internal Network (REST style)

- Find an HTTP proxy in the publically exposed Application/REST API or get access to curl on a compromised system in the internal network
- Figure out which systems are running on the internal network and target a data rich server. (Port Scanning and Fingerprinting is easier because the REST protocol is self describing)
- Exfiltrate data from the REST interface of the backend system and pass the correct parameters by sniffing the network, reusing credentials, using default userids and passwords, bypassing authentication, reading server logs to find apiKeys, etc.





Non-compromised machine

Affected machine

REST is Self Describing

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

REST is Self Describing

- 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 the running HBase version:
  - http://host:port/version
- Find the nodes in the HBase Cluster:
  - http://host:port/status/cluster
- Find all the tables in the Hbase Cluster:
  - http://host:port/

  Returns: customer and profile
- 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 all databases in the Couch DB:
  - http://host:port/_all_dbs
- Find all the documents in the Couch DB:
  - http://host:port/{db_name}/_all_docs

Neo4j REST API

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

Mongo DB REST API

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

Cassandra.io REST API

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

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)

XXE Demo

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

How does the expect:// protocol work???

SSRF

- Anything which looks like a URI/URL in XML is a candidate for internal network port scanning or data exfiltration.
- WS-Addressing example:

<To xmlns=http://www.w3.org/2005/08/addressing>
http://MongoServer:8000</To>

*See: SSRF vs. Business-critical Applications Part 2: New Vectors and Connect-Back Attacks by Alexander Polyakov

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.
  - REST APIs which use XStream and XMLDecoder where found to have these vulnerabilities
- When xml is directly deserialized to ORM objects and persisted, an attacker could supply fields which are externally hidden but present in the database (i.e. role(s)) This usually occurs in the user or profile updating logic of a REST API.

XML Serialization Remote Code Execution – XStream (Demo)

- Alvaro Munoz figured this out

XML Serialization Remote Code Execution – XMLDecoder(Demo)

XML Serialization Mass Assignment (Demo)

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.

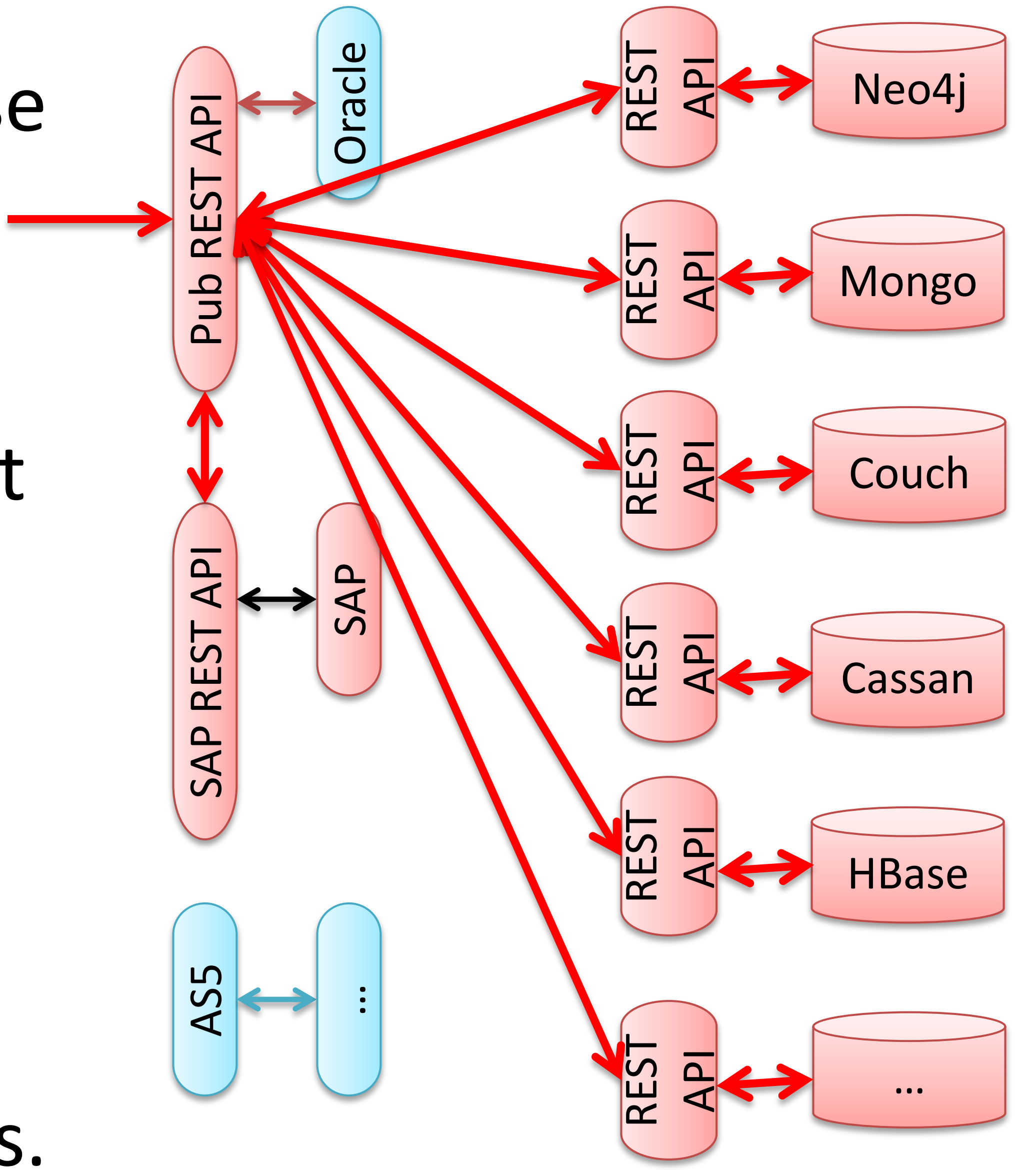
Extended HPPP (HTTP Path & Parameter Pollution)

- 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 as well as 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

Extended HPPP (Demo)

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

- Turns remote code execution 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.

Rest Extensions Remote Code Execution(Demo)

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

- Guidance related to REST implementation of security take adhering to REST principles over security
- REST provides for dynamic URLs and dynamic resource allocation

Incorrect assumptions of REST application behavior (Example 1)

- According to many REST authentication guides on the Internet, an “apiKey” passed as a GET parameter is the best way to keep track of authenticated users with stateless sessions.

Incorrect assumptions of application REST behavior (Example 1)

- According to many REST authentication guides on the Internet, an “apiKey” passed as a GET parameter is the best way to keep track of authenticated users with stateless sessions.

- But HTTP GET Parameters are usually exposed in proxy logs, browser histories, and HTTP server logs.

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 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
- The POST the following:

  POST http://svr.com:27080/test/_cmd
  - With the following body:

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

REST Attacking Summary

- Attack serialization in the exposed XML/JSON interfaces to execute remote code
- Attack the proxied requests to backend systems using Extended HPPP
- Use XXE/SSRF to read local config files, execute arbitrary code, or port scan and attack other internal REST exposed applications
- Look for other internal REST APIs through HATEOAS links in XML responses
- By-pass authentication

Questions

?