Advanced Network Security

DNS Security

Thomas Martin

t.martin@mmu.ac.uk

April 17, 2018

Outline

Outline

SQL

Many websites use databases to store user information (web servers being essentially stateless). SQL (Structured Query Language) is a way to input and extract values from a database common across different platforms (Oracle, MS, MySQL). The basic commands are:

- `SELECT columns FROM table WHERE condition ORDER BY column;`
- `INSERT INTO table (col1, col2, ...) VALUES (1, "abc", ...);`
- `UPDATE table SET col1 = "value" WHERE condition;`
- `DELETE FROM table WHERE condition;`

A site will often take user input as one of the variables in the above commands to update values (e.g. a user posting a message to a blog) or perform calculations on them (e.g. checking authentication).

SQL Injection

SQL injection is the process of adding SQL statements in user input. SQL injection occurs when developers dynamically build SQL statements by using user input. The hacker can add their own commands to the SQL statement via the user input, thereby performing operations that were not intended by the developer. Hackers can exploit this vulnerability in the following ways:

- *Probing databases:* Hackers can use the default error messages returned to investigate the design of a database.
- *Bypassing authorization:* Hackers can modify authorization-based SQL statements to gain entry to a database.
- *Executing multiple SQL statements:* Hackers can append additional SQL statements to the ones written by the developer.

SQL Injection

Expected

username: abc
password: test123
When submitted, the SQL query will be built up as:

```
SELECT * FROM users WHERE username='abc' AND password
= 'test123';
```

Unexpected

username: abc'; --'
password:
The following is the query sent onto the DB:

```
SELECT * FROM users WHERE username='abc'; --'' AND
password='';
```

SQL Injection

Expected

```
username:  doug
password:  p@$$w0rd
SELECT COUNT(*) FROM Users WHERE username='doug' and
password='p@$$w0rd'
```

Unexpected

```
username:  '  OR 1=1 --
password:
SELECT COUNT(*) FROM Users WHERE username='' OR 1=1 --
and password=''
```

SQL Code

```
<?php
session_start();
if(isset($_POST['login'])){
$username = $_POST['user']; $password = $_POST['pswd'];

$query = "SELECT * FROM login WHERE user = '$username'
AND password = '$password'";
$result = mysql_query($query);

mysql_close($conn);

if ( mysql_num_rows($result) ) {
    $_SESSION['phplogin'] = true;
     header('Location: congrats.php'); exit;
} else {
// Error Message ...
```

DROP TABLES

By entering the appropriate value in the form, an attacker could make the web server execute the following:

```
SELECT * FROM users WHERE username='’; DROP TABLE users;
-- ‘ AND password=‘asdf”;
```

An extra command is injected that will delete the whole table (or do anything else the attacker wants). SQL injections can make any changes to the database the web server can.

SQL Causes

How do attackers know to use SQL Injections?

- Insider Information
- Trial and Error - Error message often reveal too much
- Malicious user can force an error to discover information about the database

How can SQL Injections be prevented? Strong validation at server side for user input.
Data validation strategies

- Accept Only Known Valid Data
- Reject Known Bad Data
- Sanitize Bad Data

All the methods must check Data Type, Syntax, Length

SQL Prevention

Most programming languages provide specific mechanisms to prevent SQL Injection, namely parameterized queries.

```
$sql = "SELECT * FROM User WHERE User = ?  and Password = ?"
$params = array($_POST['Username'], $_POST['Password']);
$stmt = sqlsrv_query($connection, $sql, $params);
```

SQL Injection attacks can be easily defeated with simple programming changes, however, developers must be disciplined enough to apply the necessary methods to every web accessible procedure and function.

Every dynamic SQL statement must be protected. A single unprotected SQL statement can result in comprising of the application, data, or database server.

Outline

DNS

DNS is an incredibly important protocol for the correct working of the Internet. However, there are a number of vulnerabilities that have been exploited in the past and may cause more problems in the future.

At the very least, the correct functioning of DNS requires that the user be able to trust their local device. The hosts file[1] contains a list of hard-coded IP-domain name mappings that are always checked first. Alternatively, an attacker could configure the device to only query attacker-controlled DNS servers.

[1] `C:\Windows\System32\drivers\etc\hosts` in Windows or `/etc/hosts` on Linux

DNS Cache Poisoning

Compromising the DNS of an end-user device will allow the attacker to redirect all their traffic to malicious sites. The user is unlikely to notice, e.g. the correct URL will appear in the browser. However, the impact is still limited to a single user. DNS attacks on servers have the potential to impact many users.

DNS servers on the Internet need to be simple, and need to resolve any queries they receive. One way they can be attacked is with Cache Poisoning[2]. The attacker causes false records to be stored, that are then circulated to other users.

[2] https://en.wikipedia.org/wiki/DNS_spoofing

DNS Cache Poisoning

Example: The attacker forces the target server to lookup the address for `subdomain.attacker.example.` (which will result in a query going to the attacker controlled name server). The response is:

```
Answer:
(no response)

Authority section:
attacker.example. 3600 IN NS ns.target.example.

Additional section:
ns.target.example. IN A w.x.y.z
```

BIND, the most popular DNS server software, has been updated to ignore unrelated responses (version 9.5.0-P1 and above).

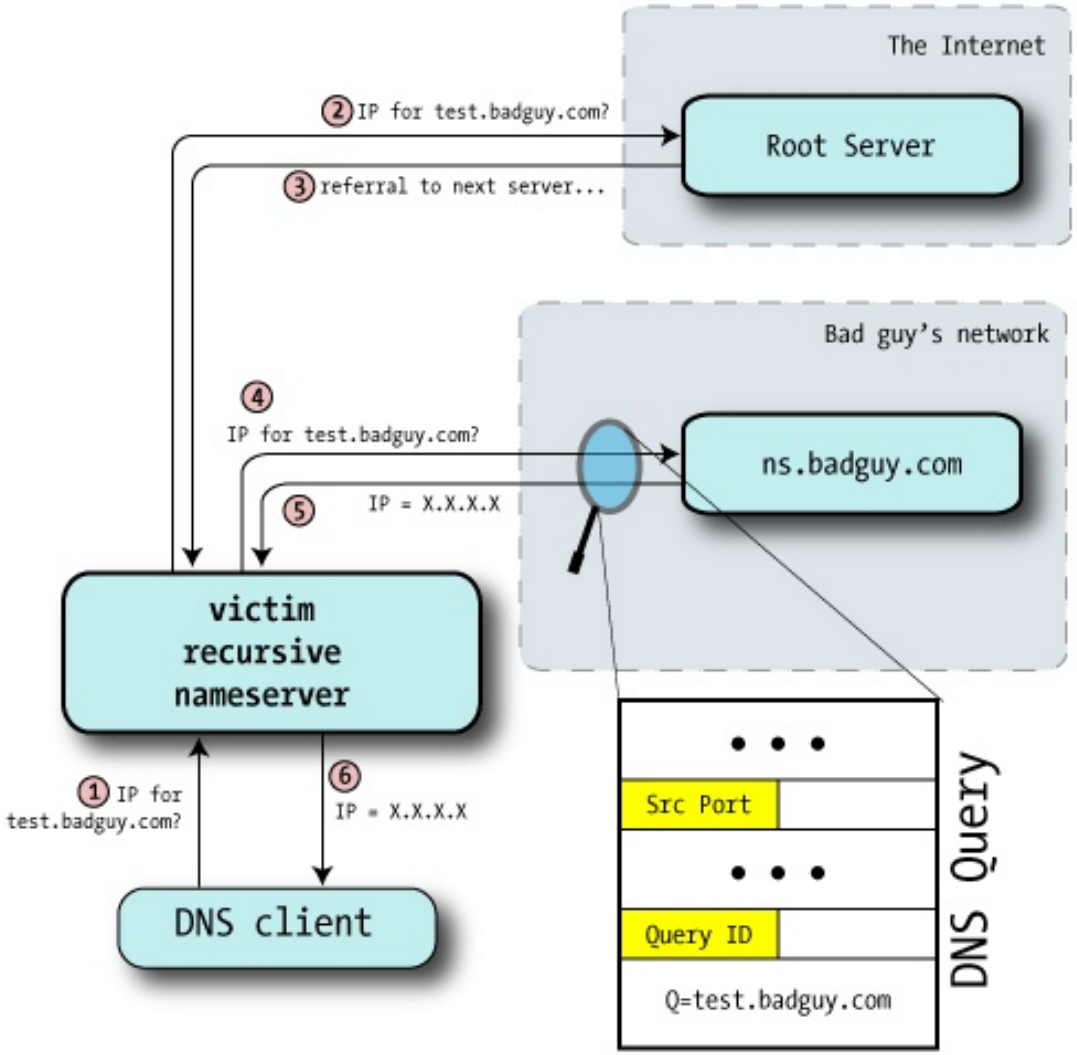
DNS Cache Poisoning

Even if a server discards unrelated responses, there is still the possibility that an attacker could spoof a reply for a query sent to another (legitimate) server. Remember, DNS uses UDP so forging responses is very feasible.

The rule is: *first good answer wins*. Any subsequent replies are dropped (even if they are completely valid). An attacker does need to correctly match the UDP source port[3], the Question section, and the Query ID (two bytes).

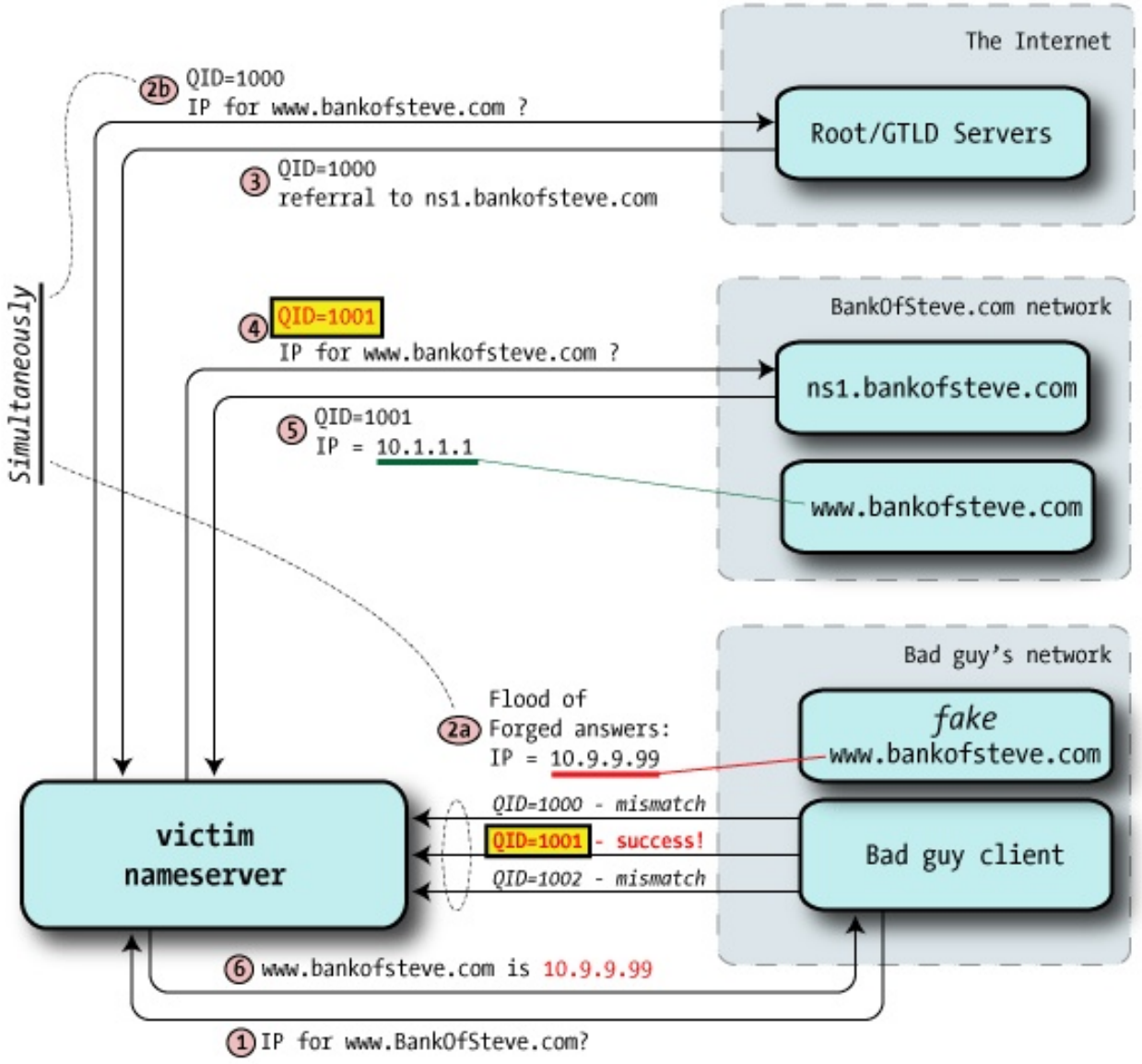
[3] Two bytes, but some servers reuse the same source port for all queries from a given client

DNS Spoof - Get Port and ID[4]



[4] Images courtesy of http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html

DNS Spoof - Sequential Query IDs

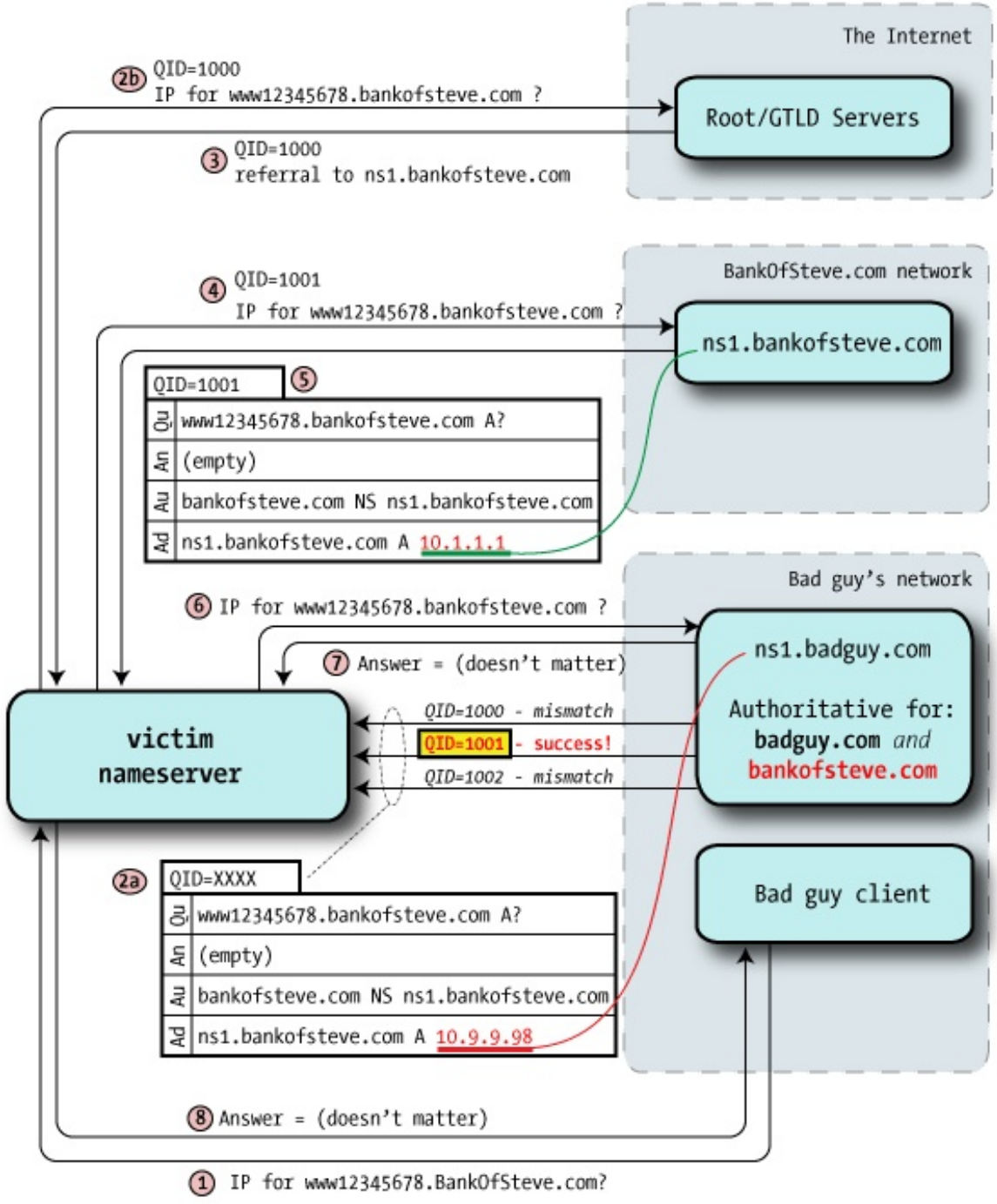


DNS Spoof

For this attack to work:

- The name cannot already be in the cache
- The bad guy has to guess the Query ID
- The bad guy has to be faster than the real nameserver

However, the attacker can also poison the target nameserver by matching any of the earlier replies. The higher up the DNS hierarchy the attack, the more damage it can do.


The Internet
Root/GTLD Servers
2b QID=1000
IP for www12345678.bankofsteve.com ?
3 QID=1000
referral to ns1.bankofsteve.com
BankOfSteve.com network
ns1.bankofsteve.com
4 QID=1001
IP for www12345678.bankofsteve.com ?
QID=1001
5
Qu www12345678.bankofsteve.com A?
An (empty)
Au bankofsteve.com NS ns1.bankofsteve.com
Ad ns1.bankofsteve.com A 10.1.1.1
Bad guy's network
ns1.badguy.com
Authoritative for:
badguy.com and
bankofsteve.com
6 IP for www12345678.bankofsteve.com ?
7 Answer = (doesn't matter)
victim
nameserver
QID=1000 - mismatch
QID=1001 - success!
QID=1002 - mismatch
Bad guy client
2a QID=XXXX
Qu www12345678.bankofsteve.com A?
An (empty)
Au bankofsteve.com NS ns1.bankofsteve.com
Ad ns1.bankofsteve.com A 10.9.9.98
8 Answer = (doesn't matter)
1 IP for www12345678.BankOfSteve.com?

Birthday Attack

The attack is trivial if the DNS server has sequential ID and fixed source port numbers. Any predictable pattern in the ID's can make poisoning possible. The attack is even more likely to succeed if the server generates multiple recursive queries in response to multiple requests for the same address[5].

The probability that one of 20 spoofed responses will match the Query ID is small (0.03%). However, the odds are much higher if you just need one of the 20 spoofed responses to match any of 20 Query ID's (0.3%). When you raise the number to 300, the probability is around 50%. At 700, it is almost 100%.

[5] https://www.secureworks.com/blog/dns-cache-poisoning

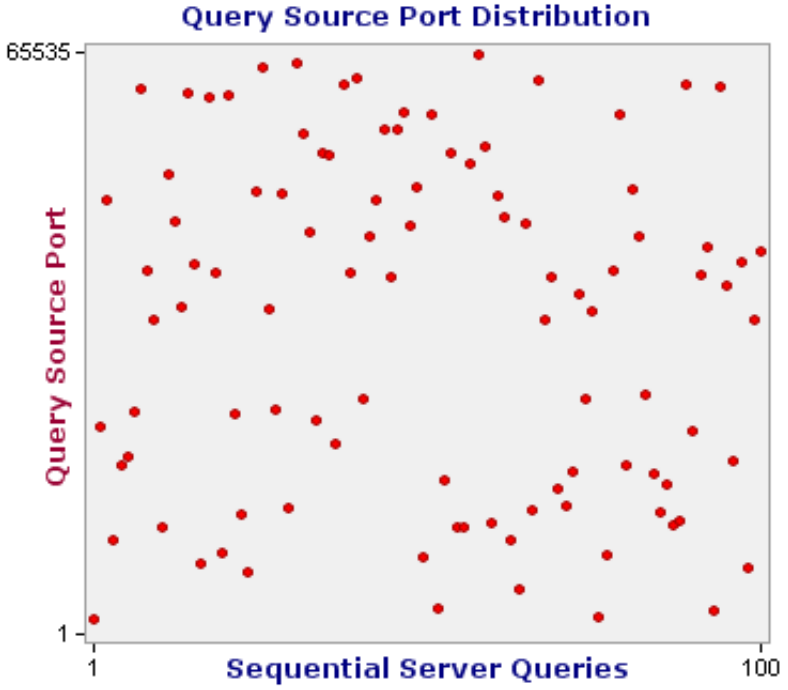
Mitigation

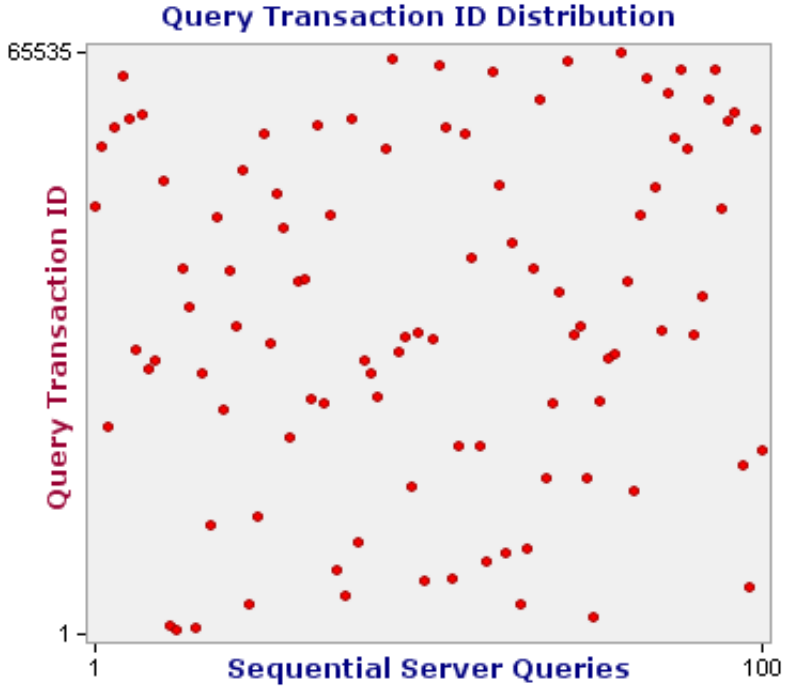
Some precautions have already been mentioned: ignoring unrelated records in responses, limiting traffic generated in response to multiple similar requests. But the main mitigation for these attacks is in having unpredictable Query ID's and source port numbers.

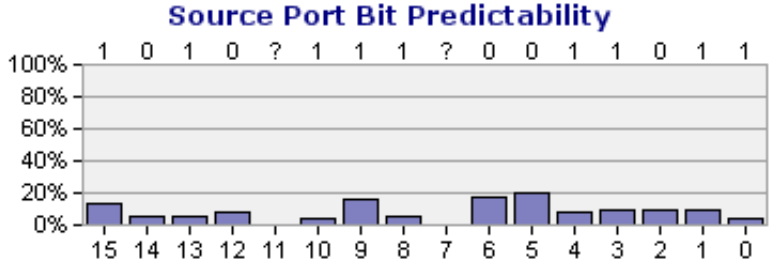
There is a freely available DNS Nameserver Spoofability Test[6] available that will perform some statistical analysis on these parameters for the locally configured DNS servers.

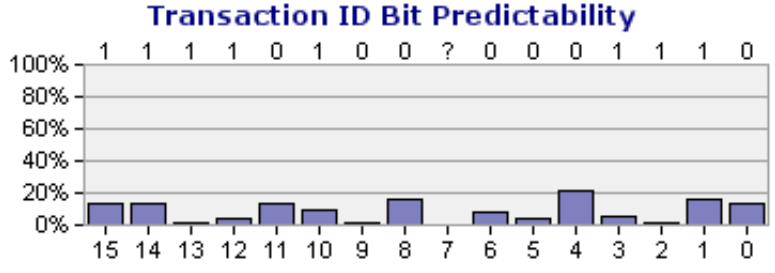
[6] https://www.grc.com/dns/dns.htm

Scatter Charts - Good Examples









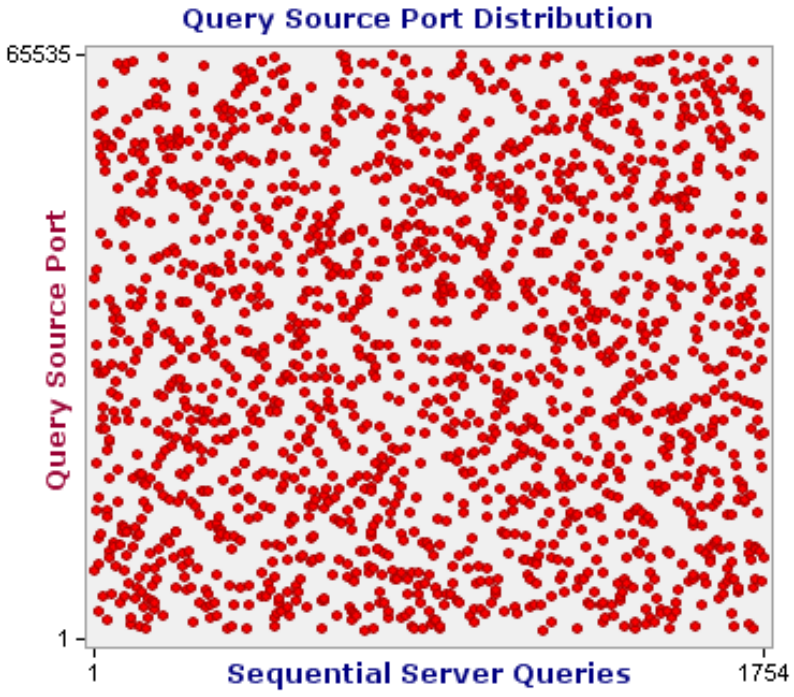
Query Source Port Analysis (worst case)

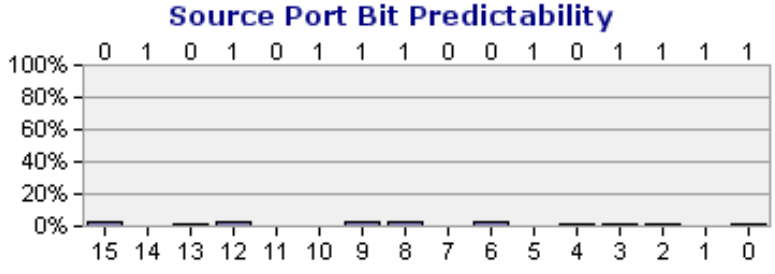
Max Entropy: 15.96 **Excellent**		Dir Bias: 3.03% **Excellent**	
Lost Entropy: 0 **Excellent**		Stuck Bits: 0 **Excellent**	

Query Transaction ID Analysis (worst case)

Max Entropy: 15.99 **Excellent**		Dir Bias: 3.03% **Excellent**	
Lost Entropy: 0 **Excellent**		Stuck Bits: 0 **Excellent**	

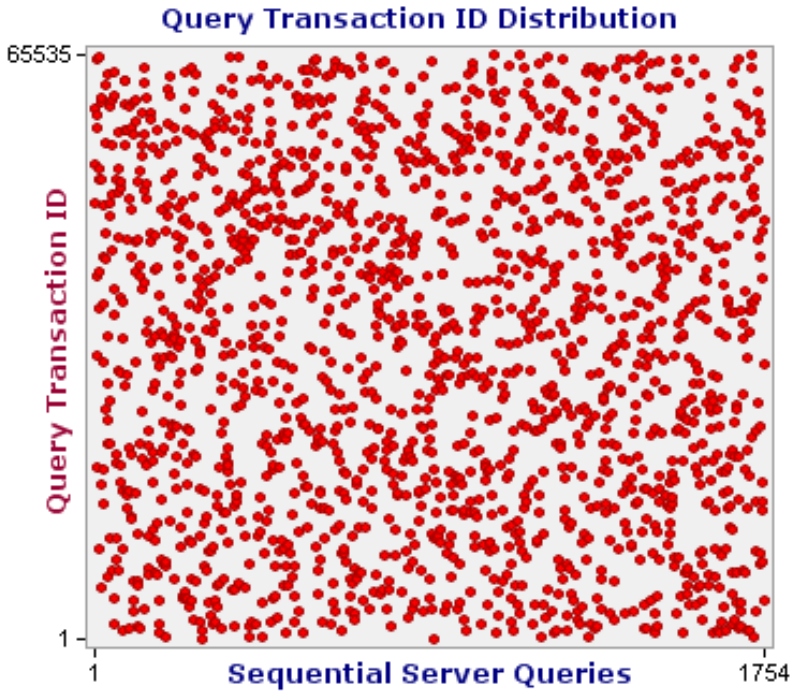
Scatter Charts - Good Examples

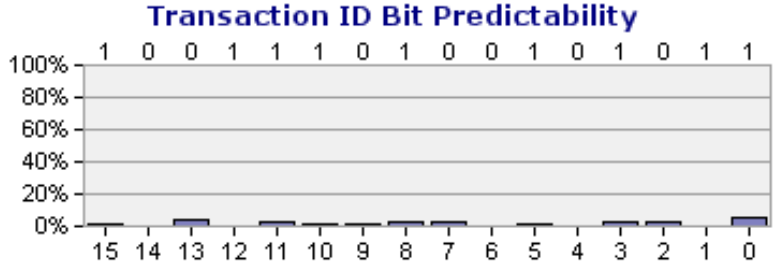




Query Source Port Analysis (worst case)

Max Entropy:	15.98	Excellent	Dir Bias:	1.43%	Excellent
Lost Entropy:	0.02	Excellent	Stuck Bits:	0	Excellent

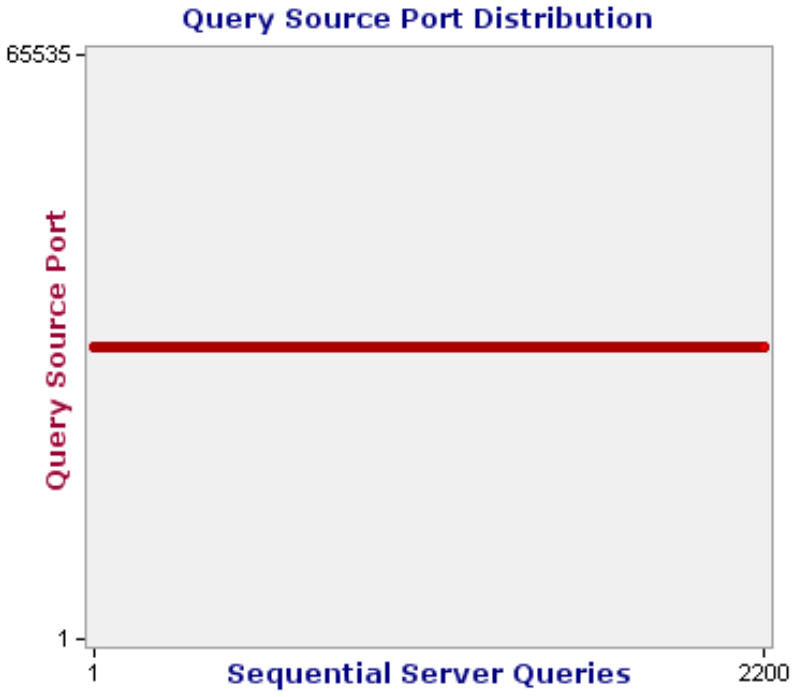


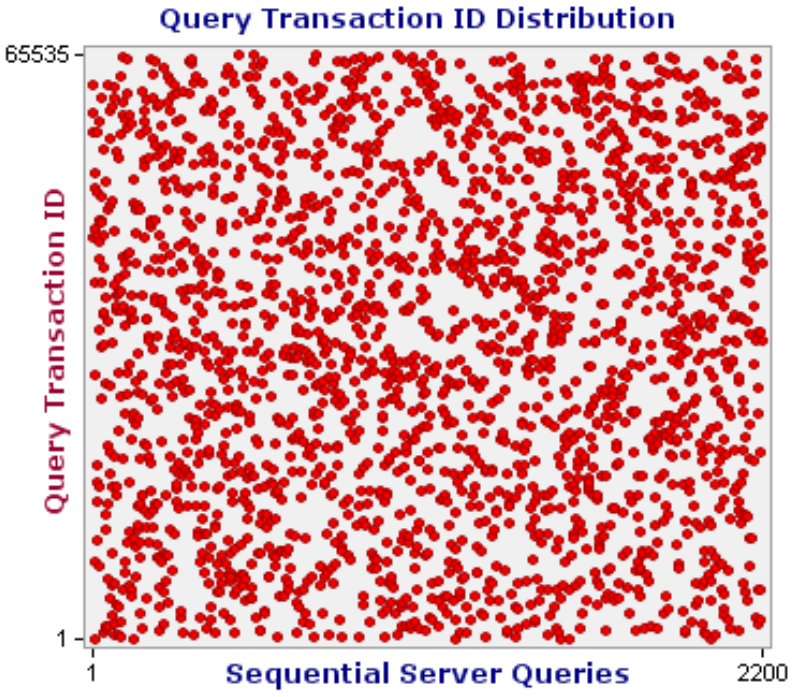


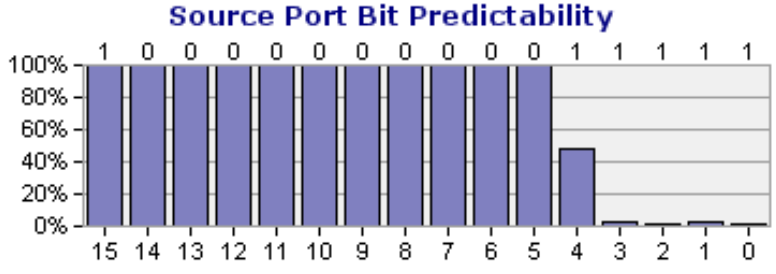
Query Transaction ID Analysis (worst case)

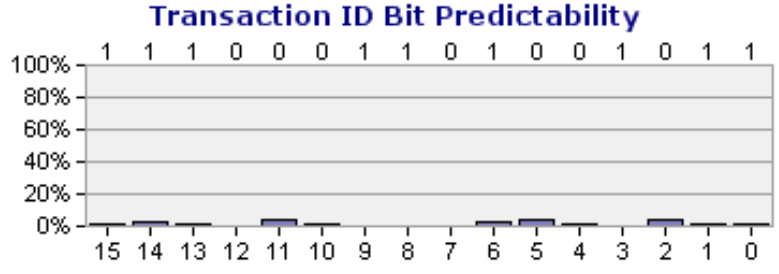
Max Entropy:	16	Excellent	Dir Bias:	0.63%	Excellent
Lost Entropy:	0.03	Excellent	Stuck Bits:	0	Excellent

Scatter Charts - Bad Examples









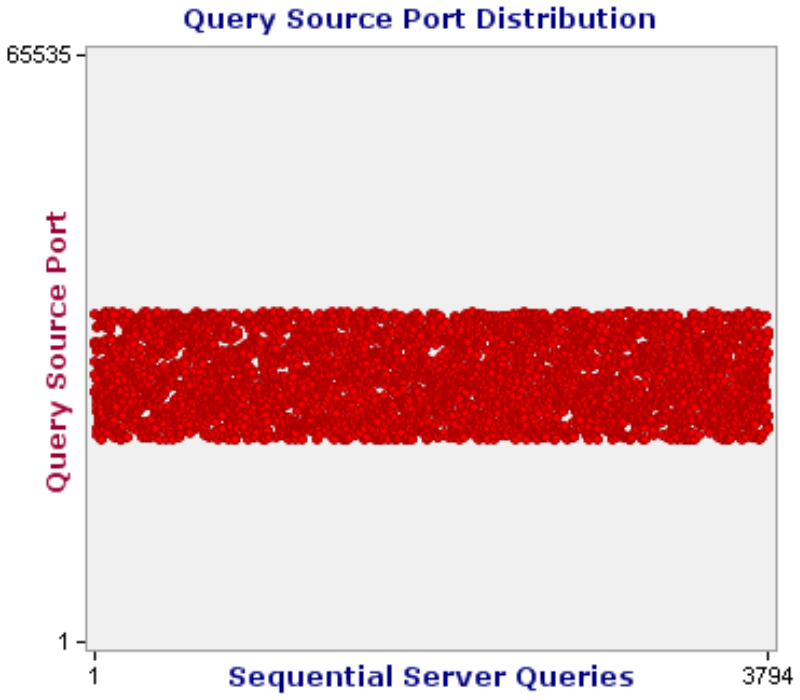
Query Source Port Analysis (worst case)

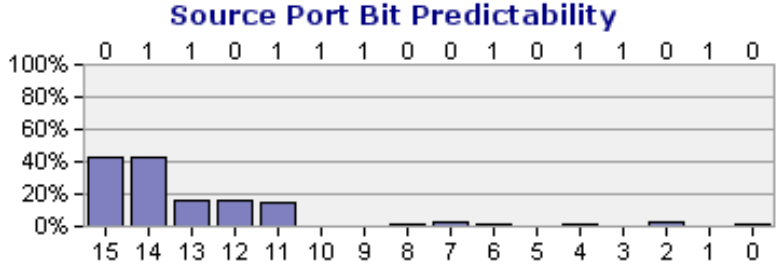
Max Entropy:	3.91 **Very Bad**	Dir Bias:	0.23% **Excellent**
Lost Entropy:	7.1 **Very Bad**	Stuck Bits:	11 **Very Bad**

Query Transaction ID Analysis (worst case)

Max Entropy:	16 **Excellent**	Dir Bias:	0.14% **Excellent**
Lost Entropy:	0.03 **Excellent**	Stuck Bits:	0 **Excellent**

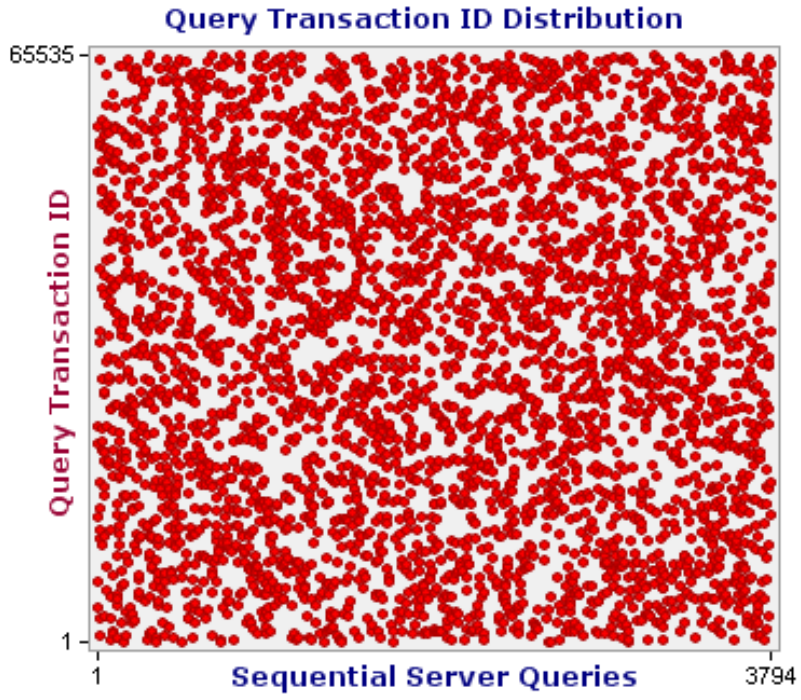
Scatter Charts - Bad Examples

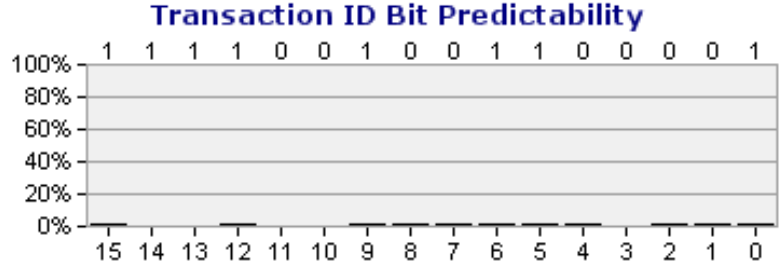




Query Source Port Analysis (worst case)

Max Entropy:	13.81	Moderate	Dir Bias:	1.87%	Excellent
Lost Entropy:	0.16	Good	Stuck Bits:	0	Excellent

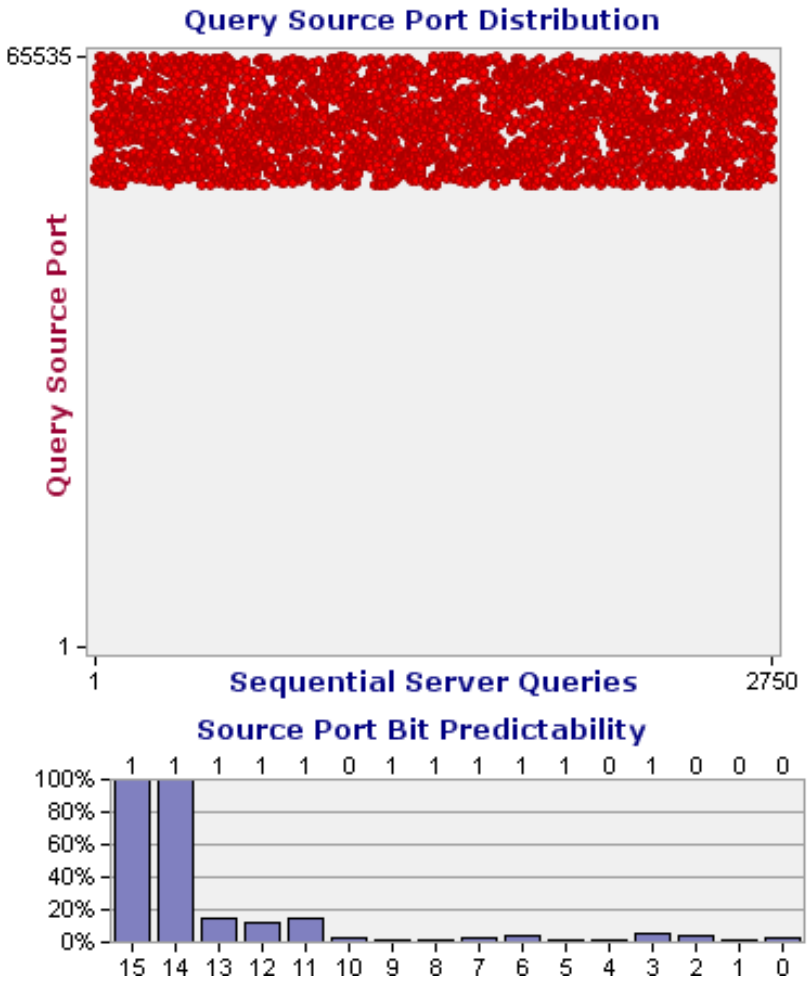


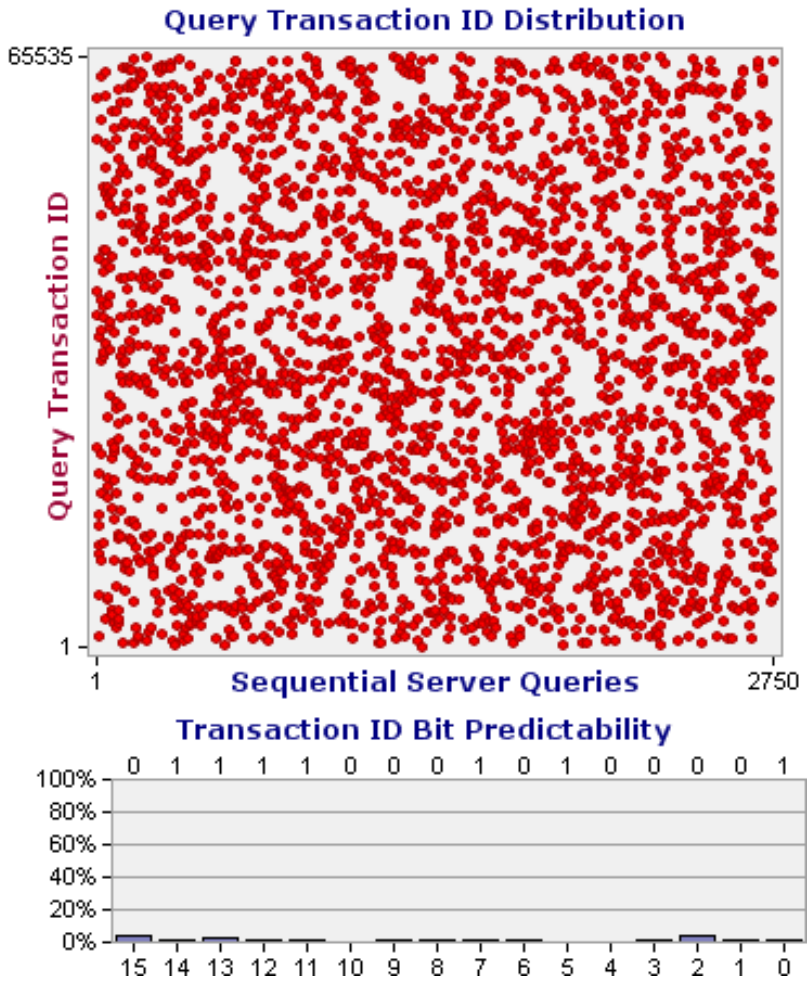


Query Transaction ID Analysis (worst case)

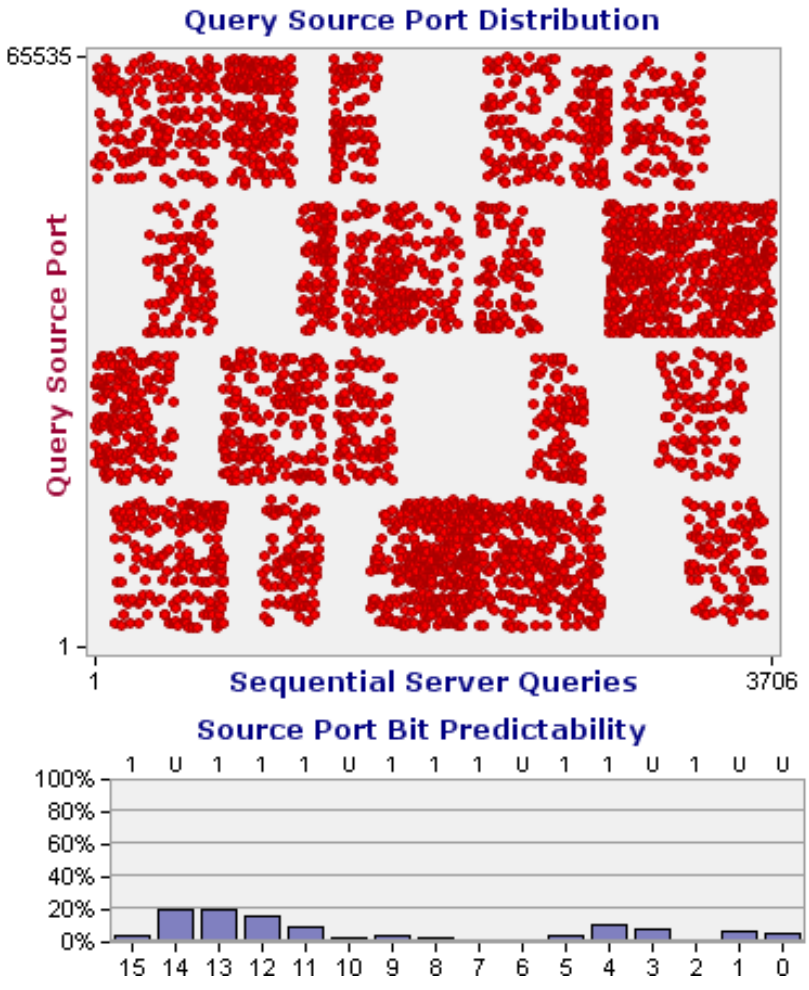
Max Entropy:	16	Excellent	Dir Bias:	0.4%	Excellent
Lost Entropy:	0.04	Excellent	Stuck Bits:	0	Excellent

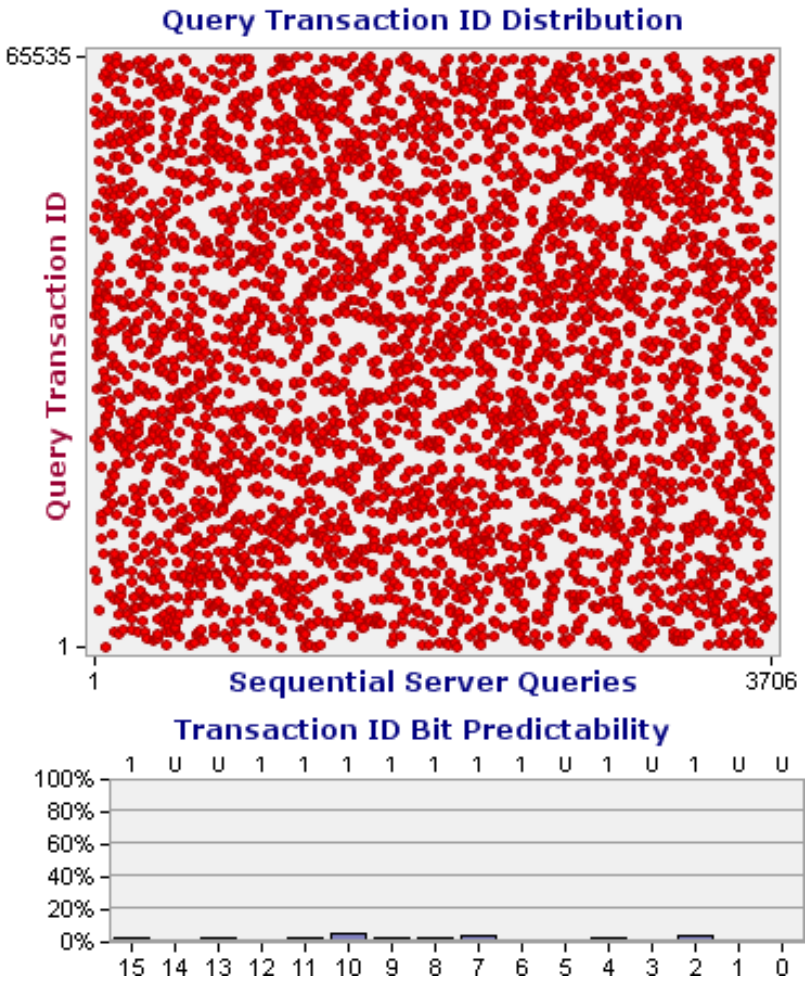
Scatter Charts - Bad Examples



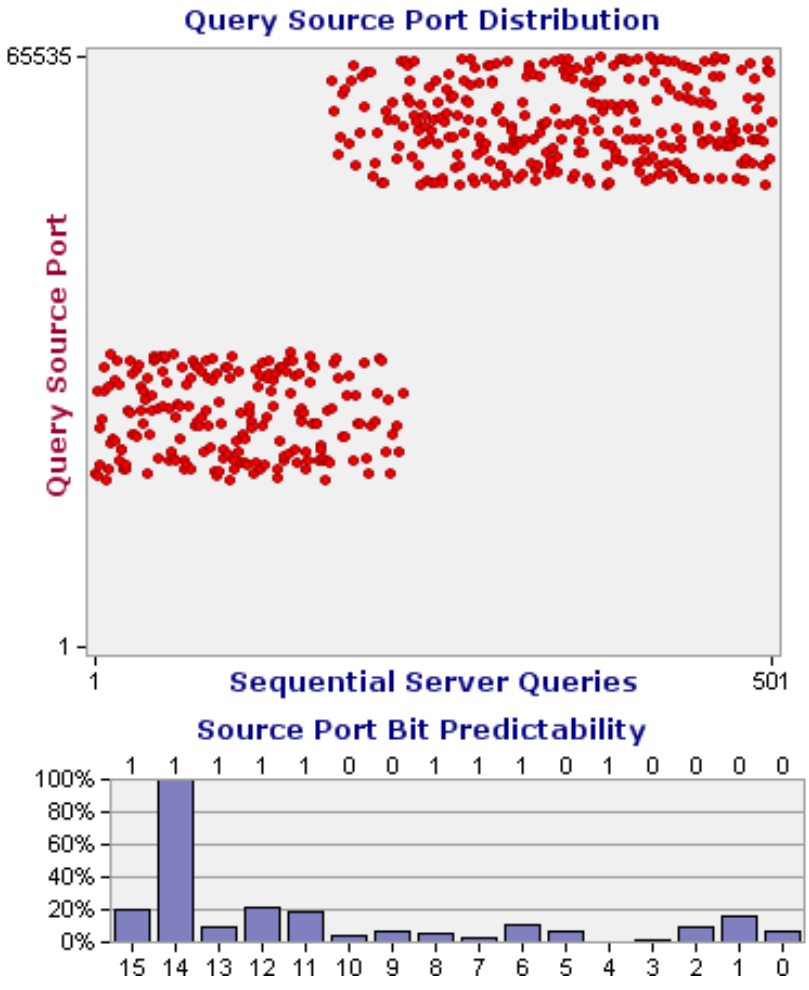


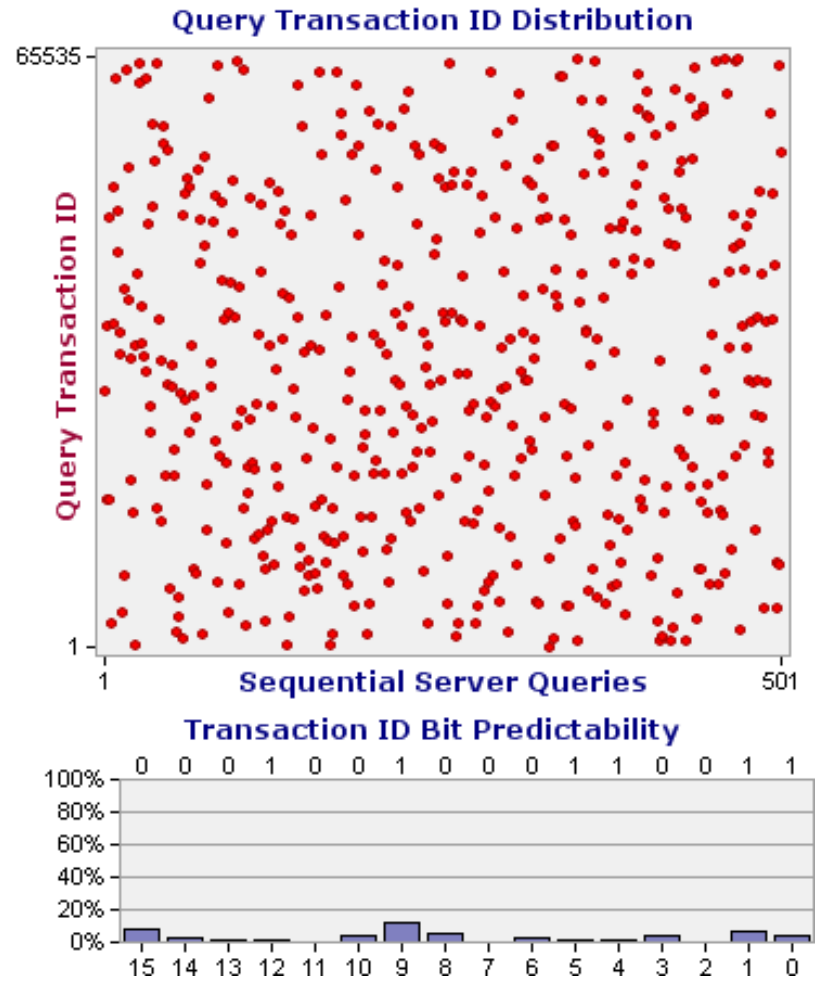
Scatter Charts - Bad Examples



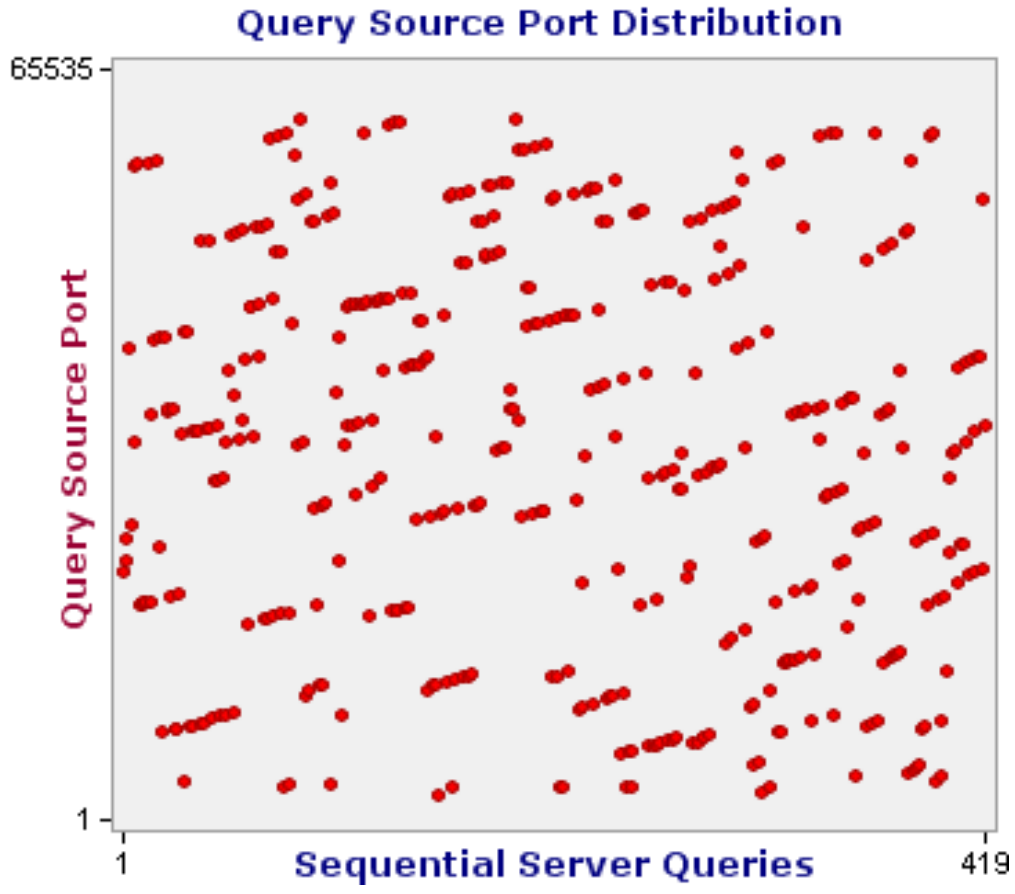


Scatter Charts - Bad Examples

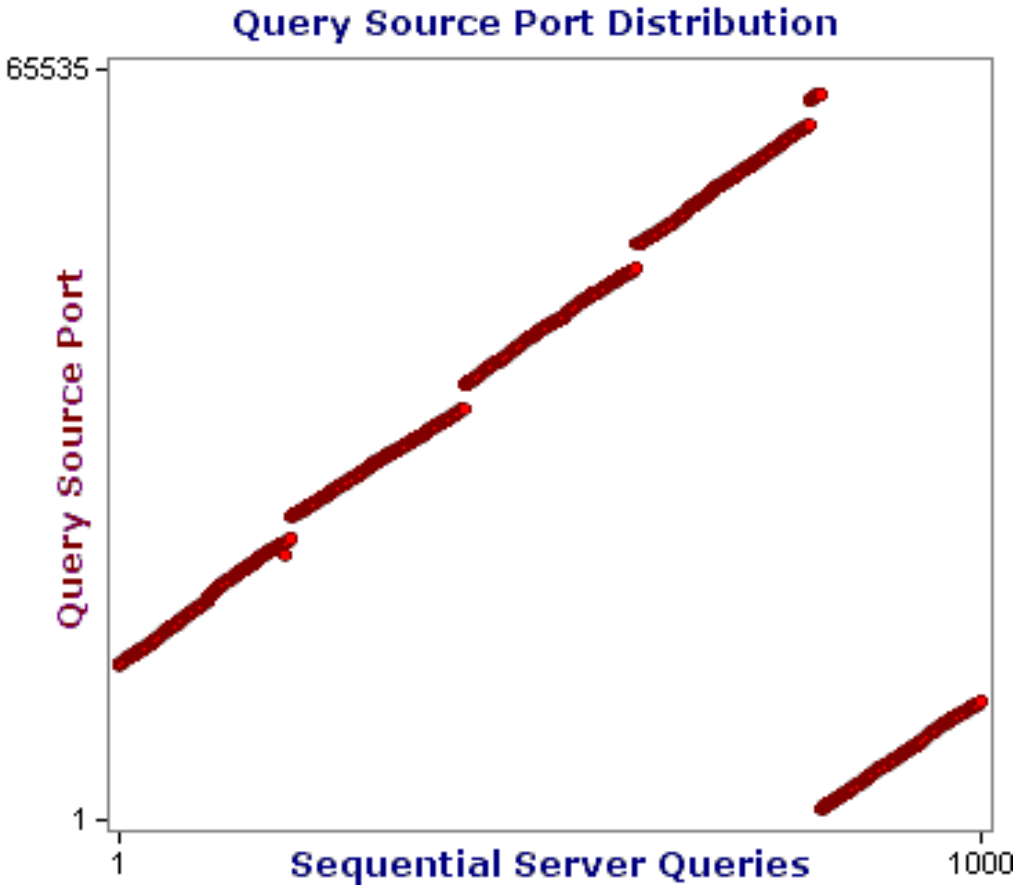




Scatter Charts - Bad Examples



Scatter Charts - Bad Examples



DNSSEC

There have been proposals for changes in the DNS system to allow clients to detect fraudulent records. The Domain Name System Security Extension[7] (DNSSEC) uses digital signatures to achieve this in a way that is backwards compatible.

The Resource Records for a domain are signed by the administrator, and these signatures (and the public key) are made available (DNSSEC defines several new record types: DNSKEY, RSIG, NSEC). They also need to get the public key signed by the private key of the parent zone. So long as each client trusts the root public key, any signed record can be verified.

[7] http://www.dnssec.net/

HSTS

DNS spoofing causes incorrect IP mappings to be accepted by the victim client. But if the client is browsing to a `HTTPS` site, there will still be an error because the attacker will not have the correct private key. Attackers get around this by simply blocking all attempts to create a secure connection and forcing the victim to use `HTTP`.

With HTTP Strict Transport Security, a server can notify a client that all future connections must be secure (for a period of time). A subsequent attack that includes a man-in-the-middle performing SSL/TLS stripping will result in an error. The major browsers all support HSTS.

Thank you

Relevant chapter:
McClure, Hacking Exposed 7, Chapter 10.

Any Questions?