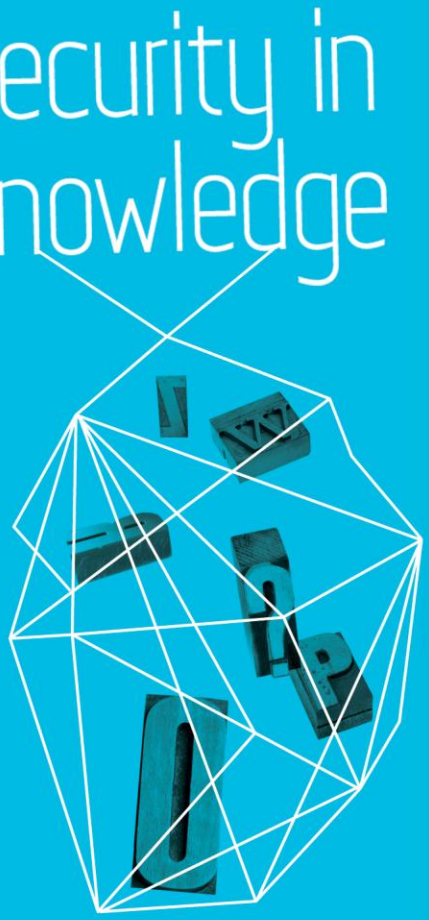
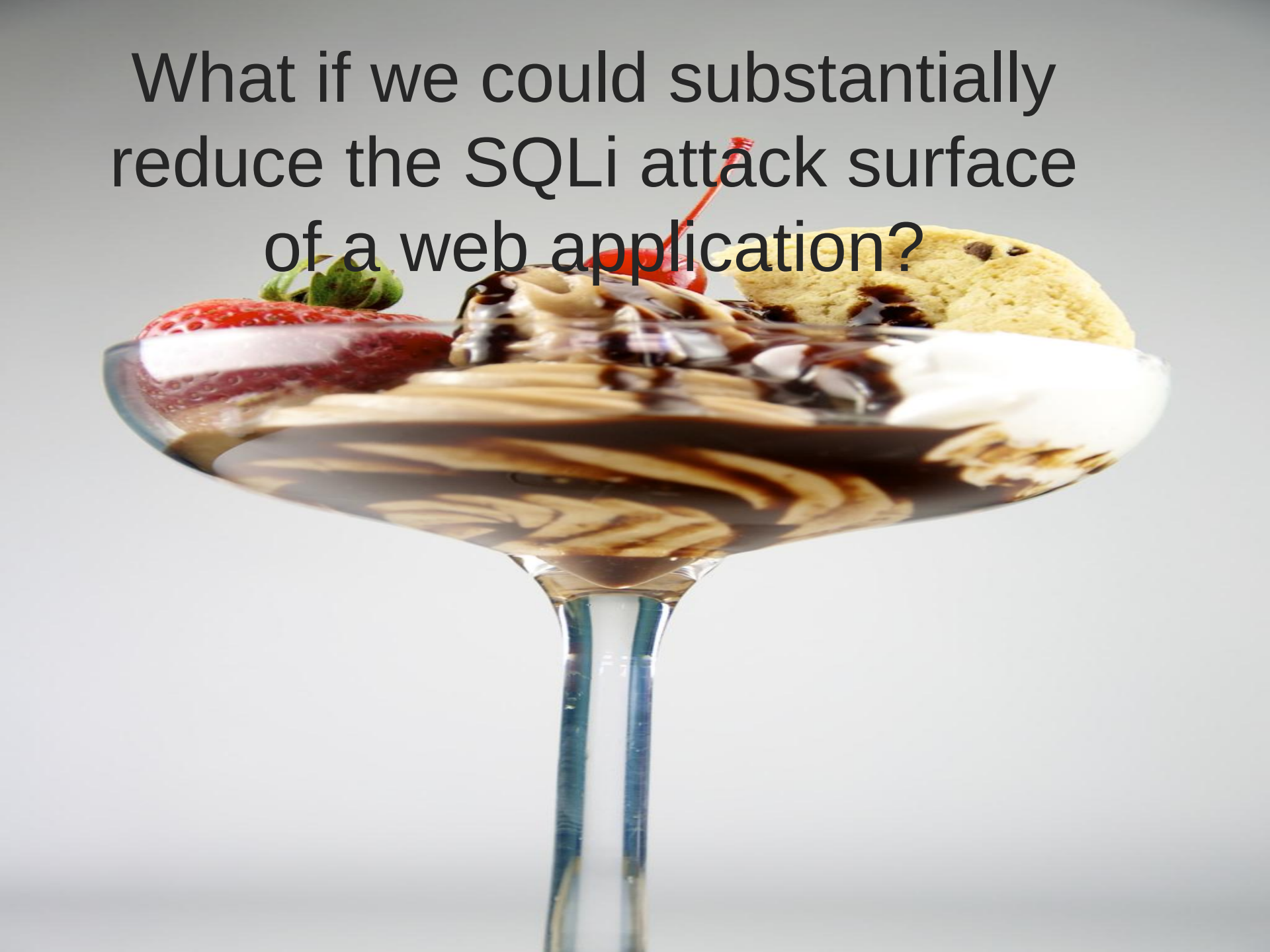
libinjection: New Directions in SQLi Detection



Nick Galbreath
IPONWEB

Session ID: ASEC-W23
Session Classification: Advanced

What if we could substantially reduce the SQLi attack surface of a web application?

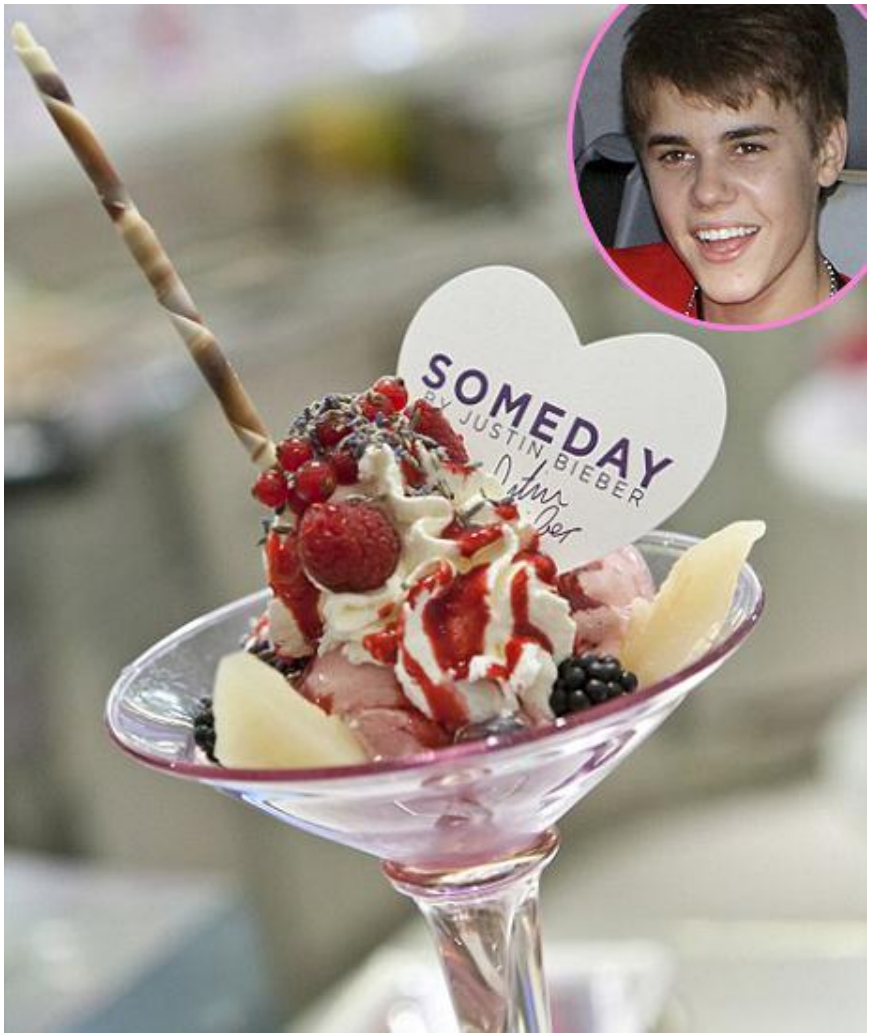


without new hardware or firewalls?

without application changes?


and comes with free SQLi attack monitoring?

Wild Speculation?



A Brief History of SQL

1970s: SQL Invented

- Hey that's 40 years ago.
- Why is it still around and so popular?
- Exercise for the reader: Pick any query you like, and write the equivalent in your favorite programming language.
- SQL is *scriptable data structures*.

1972: Oracle Releases the First Commercial Database

Also, Coppola releases The Godfather

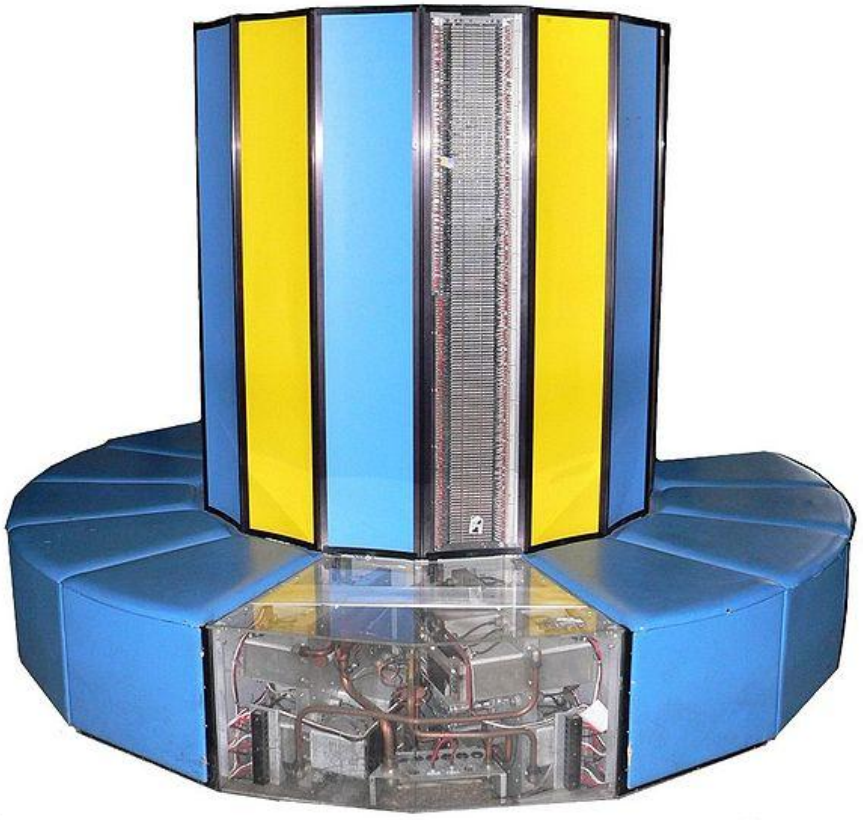
Remember the 80s?

- Networking sucked!
- TCP/IP not widespread
- Computers are fragile, expensive and slow
- Shoulder-pads

so what do you do?

Centralize

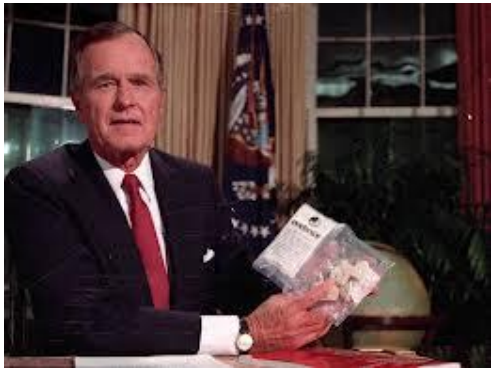
- Move computation as close as possible to the data
- Move to super-servers
- Have cheap/dumb clients



1988: Oracle V6
Introduction of PL/SQL

- The Database is now a full programming environment
- Unicode/UTF8 not standard
- Complexity explosion
- ... but most clients are private

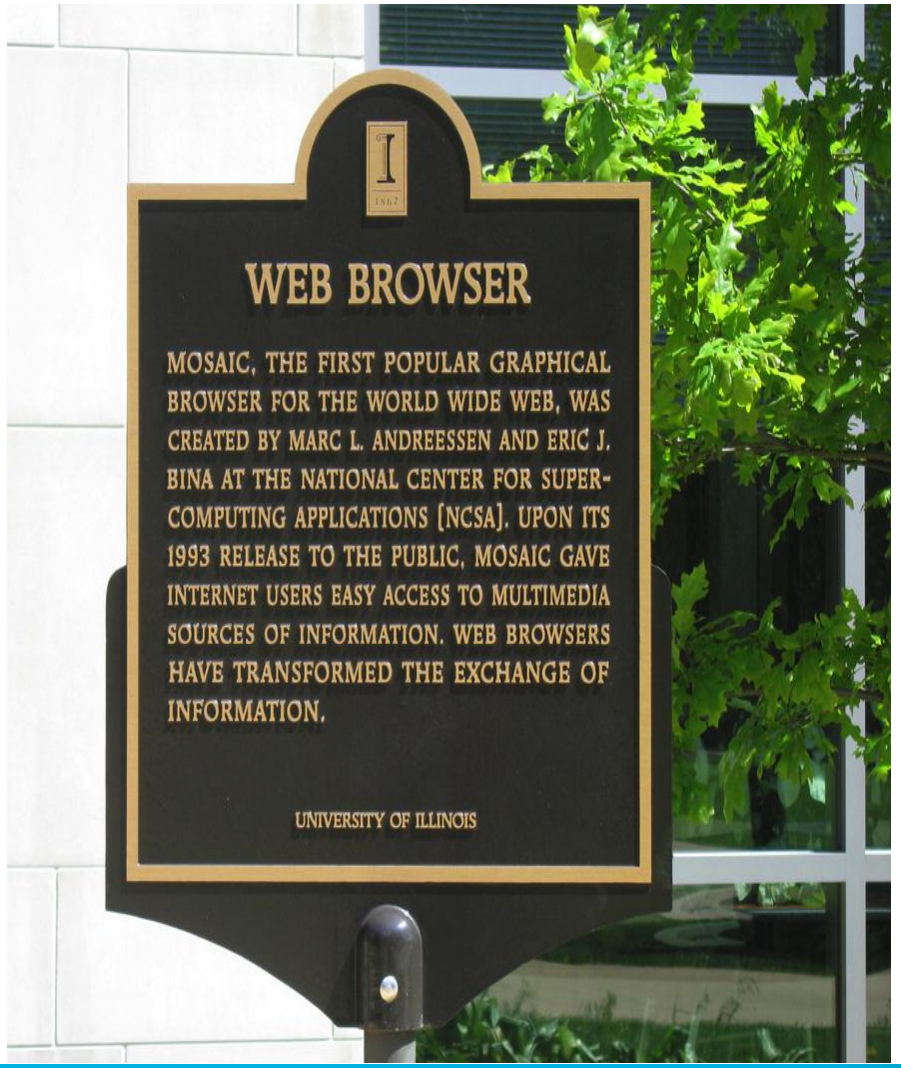
Also in 1988:
Crack Cocaine 'invented'



And then the 1990s

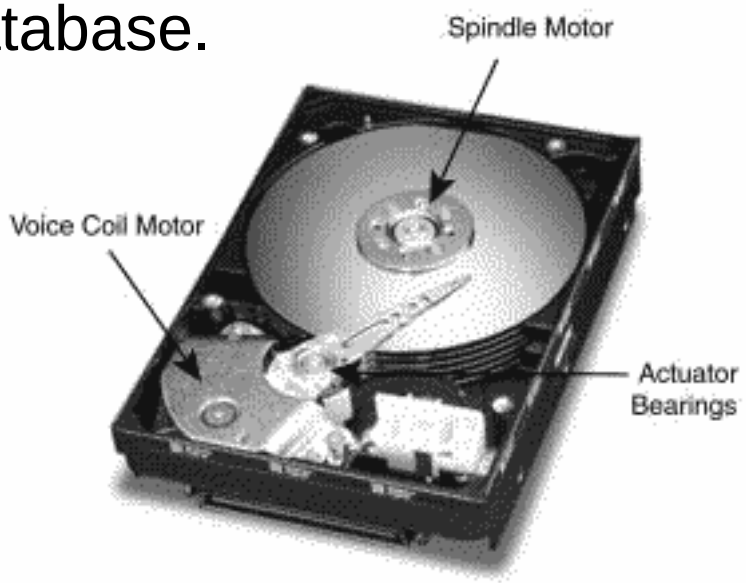
- TCP/IP
- Cheap CPUs
- Web Browsers
- Attaching databases to public networks

This is why most of us are here today.



2000+ Web Scale

- Discovery that data problems are a lot more painful than CPU problems.
- Turns out disk drives are mechanical
- If your database maxes out, you have big problems... so move everything out of the database.
- Complete reversal in strategy



SQL isn't going anywhere

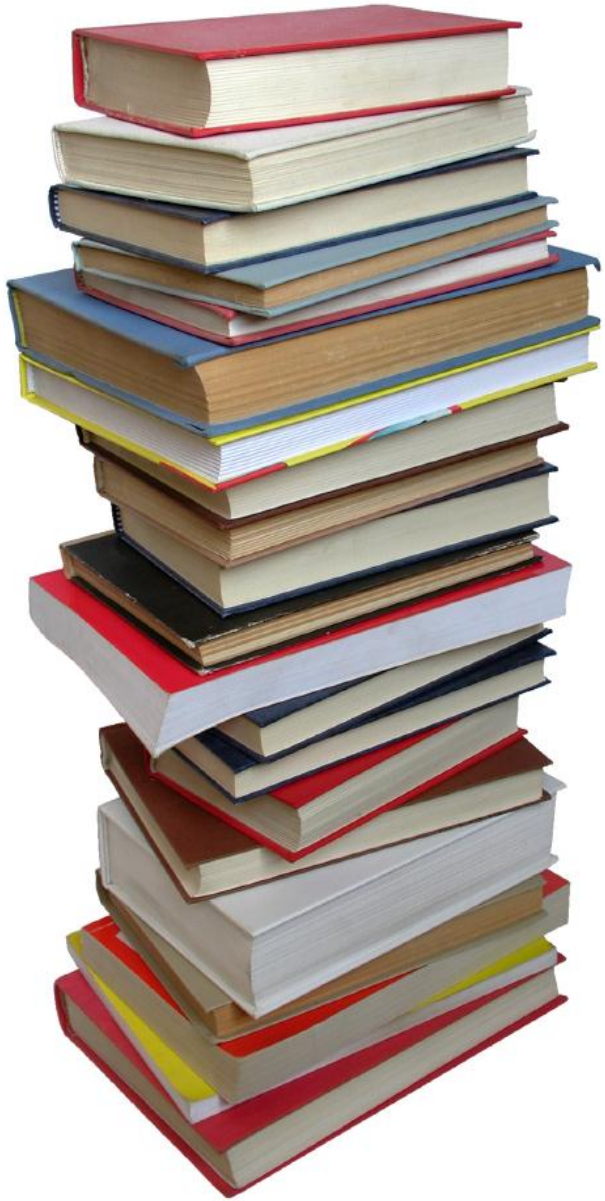
- For front-ends, general trend is federating data across cheap machines, using stripped down SQL
- SQL-like languages used by Amazon, Google and others.
- Still great for analytics and reporting on the back end, and generic data storage.
- But we stuck with legacy of the past.


SQL's Dark Corners
bit.ly/UkQd7Z

SQL is Huge

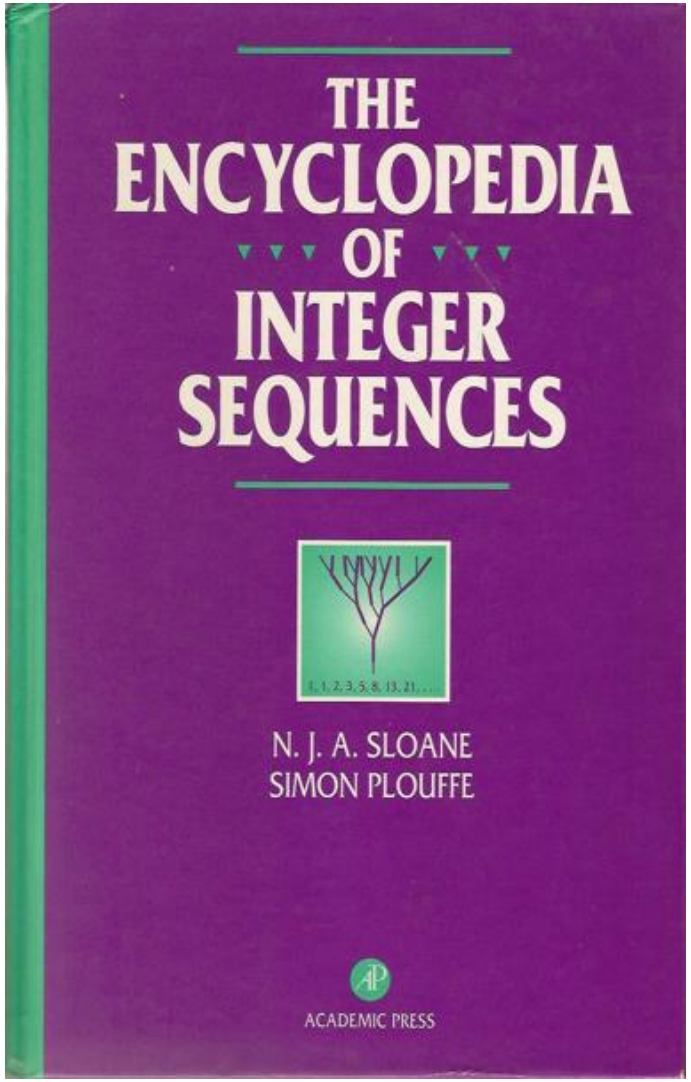
- 40+ years of built up crud
- 1992 spec is 625 pages of plain text
- 2003 spec is 128 pages of BNF
- No one is completely compliant
- Every one has special extensions

SQL is more complicated than you think....



SQL Integer Forms

- [0-9]+
- 0x[0-9a-fA-F]+ 0xDEADbeef
  MySQL, MSSQL
  0x is case sensitive
- 0x MSSQL only
- x'DEADbeef' PgSQL
- b'10101010' MySQL, PgSQL
- 0b010101 MySQL



SQL Floating Point Forms

- digits
- digits[.]F
- digits[.]digits
- digits[eE]digits
- digits[eE][+-]digits
- digits[.][eE]digits
- digits[.]digits[eE][+-]digits
- digits[.]digits[eE]digits
- [.]digits
- [.]digits[eE]digits
- [.]digits[eE][+-]digits
- "binary_float_infinity" (O)

http://bit.ly/Qp6KTu

Optional starts with [+-]
Optional ending with [dDfF] (Oracle)

SQL Money Literals

- MSSQL has a money type.
- -$45.12
- $123.0
- +$1,000,000.00 Commas ignored
- *Many* symbols are accepted for currency type

SQL Ridiculous Operators

- != not equals, standard
- <=> mysql
- <> mssql
- ^= oracle
- !>, !< not less than mssql
- /\ oracle
- !! factorial (pgsql)
- |/ sqaure root (pgsql)
- ||/ cube root (pgsql)
- ** exponents (oracle)
- # bitwise xor (pgsql conflicts with mysql comment)

SQL Strings, Charset & Comments

Such a tangled mess, I defer to my DEFCON 20 talk: http://client9.com/20120727/

SQLi Detection

Keyword Detection

```
s/UNION.ALL/i
```

- The dumbest possible regexp.
- I've used this regexp as a goof for a while, but *oddly works* well in detecting SQLi *scans*.
- Almost zero false positives

By Using Regular Expressions

Trying to catch more SQLi attacks leads to the question of is user input SQLi or not? Using regular expressions you end up with something like this:

```
(?:\)\s*when\s*\d+\s*then)|(?:"\s*(?:#|
|{))|(?:\/\*!\s?\d+)|(?:ch(?:a)?r\s*\(\s*\d)|(?:(?:(n?and|x?or|not)\s+|\||\|\||\&\&)\s*\w+\()(?:[\s()]case\s*\()|(?:\)\s*like\s*\()|(?:
having\s*[^\s]+\s*[^\w\s])|(?:if\s?\([\d\w]\s*[=<>~])(?:"\s*or\s*"?\d)|(?:\\x(?:23|27|3d))|(?:^.?"$)|(?:(?:^["\\]*(?:[\d"]+|[^"]+"))
+\s*(?:n?and|x?or|not|\||\|\||\&\&)\s*[\w"[+&!@(),.-])|(?:[^\w\s]\w+\s*[|-]\s*"\s*\w)|(?:@\w+\s+(and|or)\s*["\d]+)|(?:@[\w-
]+\s(and|or)\s*[^\w\s])|(?:[^\w\s:]\s*\d\W+[^\w\s]\s*".)|(?:\Winformation_schema|table_name\W)(?:"\s*\*.+(?:or|id)\W*"\d)|(?:\^")|(?
:^[\w\s"-
]+(?<=and\s)(?<=or\s)(?<=xor\s)(?<=nand\s)(?<=not\s)(?<=\||\|)(?<=\&\&)\w+\()|(?:"[\s\d]*[^\w\s]+\W*\d\W*.*["\d])|(?:"\s*[^\w\s?]+\s*
[^\w\s]+\s*")|(?:"\s*[^\w\s]+\s*[\W\d].*(?:#|--))|(?:".*\*\s*\d)|(?:"\s*or\s[^\d]+[\w-]+.*\d)|(?:[()*<>%+-][\w-
]+[^\w\s]+"[^,])(?:\d"\s+"\s+\d)|(?:^admin\s*"|(\/\*)+"+\s?(?:--|#|\/\*|{)?)|(?:"\s*or[\w\s-]+\s*[+<>=(),-
]\s*[\d"])|(?:"\s*[^\w\s]?=\s*")|(?:"\W*[+=]+\W*")|(?:"\s*[!=|][\d\s!=+-
]+.*["(].*$)|(?:"\s*[!=|][\d\s!=]+.*\d+$)|(?:"\s*like\W+[\w"(])|(?:\sis\s*0\W)|(?:where\s[\s\w\.,-
]+\s=)|(?:"[<>~]+")(?:union\s*(?:all|distinct|[(!@]*)?\s*([([*\s*select)|(?:\w+\s+like\s+\")|(?:like\s*"\%)|(?:"\s*like\W*["\d])|(?:
"\s*(?:n?and|x?or|not
|\|\||\&\&)\s+[\s\w]+=\s*\w+\s*having)|(?:"\s*\*\s*\w+\W+")|(?:"\s*[^?\w\s=.,;)(]+\s*[(@"]*\s*\w+\W+\w)|(?:select\s*[\[\]()\s\w\.,"-
]+from)|(?:find_in_set\s*\()(?:in\s*\(+\s*select)|(?:(?:n?and|x?or|not
|\|\||\&\&)\s+[\s\w+]+(?:regexp\s*\(|sounds\s+like\s*"|[=\d]+x))|("\s*\d\s*(?:--|#))|(?:"[%&<>^=]+\d\s*(=|or))|(?:"\W+[\w+-
]+\s*=\s*\d\W+")|(?:"\s*is\s*\d.+"?\w)|(?:"\|?[\w-
]{3,}[^\w\s.,]+")|(?:"\s*is\s*[\d.]+\s*\W.*")(?:[\d\W]\s+as\s*["\w]+\s*from)|(?:^[\W\d]+\s*(?:union|select|create|rename|truncate|lo
ad|alter|delete|update|insert|desc))|(?:(?:select|create|rename|truncate|load|alter|delete|update|insert|desc)\s+(?:(?:group_)concat
|char|load_file)\s?\(?)|(?:end\s*\);)|("\s+regexp\W)|(?:[\s(]load_file\s*\()(?:@.+=\s*\(\s*select)|(?:\d+\s*or\s*\d+\s*[\-
+])|(?:\/\w+;?\s+(?:having|and|or|select)\W)|(?:\d\s+group\s+by.+\()|(?:(?:;|#|--)\s*(?:drop|alter))|(?:(?:;|#|--
)\s*(?:update|insert)\s*\w{2,})|(?:[^\w]SET\s*@\w+)|(?:(?:n?and|x?or|not
|\|\||\&\&)[\s(]+\w+[\s)]*[!=+]+[\s\d]*["=()])(?:"\s+and\s*=\W)|(?:\(\s*select\s*\w+\s*\()|(?:\*\/from)|(?:\+\s*\d+\s*\+\s*@)|(?:\w"
\s*(?:[-
+=|@]+\s*)+[\d(])|(?:coalesce\s*\(|@@\w+\s*[^\w\s])|(?:\W!+"\w)|(?:";\s*(?:if|while|begin))|(?:"[\s\d]+=\s*\d)|(?:order\s+by\s+if\w*
\s*\()|(?:[\s(]+case\d*\W.+[tw]hen[\s(])(?:(select|;)\s+(?:benchmark|if|sleep)\s*?\(\s*\(?\s*\w+)(?:create\s+function\s+\w+\s+return
s)|(?:;\s*(?:select|create|rename|truncate|load|alter|delete|update|insert|desc)\s*[\[(]?\w{2,})(?:alter\s*\w+.*character\s+set\s+\w
+)|(";\s*waitfor\s+time\s+")|(?:";.*:\s*goto)(?:procedure\s+analyse\s*\()|(?:;\s*(declare|open)\s+[\w-
]+)|(?:create\s+(procedure|function)\s*\w+\s*\(\s*\)\s*-
)|(?:declare[^\w]+[@#]\s*\w+)|(exec\s*\(\s*@)(?:select\s*pg_sleep)|(?:waitfor\s*delay\s?"+\s?\d)|(?:;\s*shutdown\s*(?:;|--
|#|\/\*|{))(?:\sexec\s+xp_cmdshell)|(?:"\s*!\s*["\w])|(?:from\W+information_schema\W)|(?:(?:(?:current_)?user|database|schema|connec
tion_id)\s*\([^\)]*)|(?:";?\s*(?:select|union|having)\s*[^\s])|(?:\wiif\s*\()|(?:exec\s+master\.)|(?:union select
@)|(?:union[\w(\s]*select)|(?:select.*\w?user\()|(?:into[\s+]+(?:dump|out)file\s*")(?:merge.*using\s*\()|(execute\s*immediate\s*")|(
?:\W+\d*\s*having\s*[^\s\-])|(?:match\s*[\w(),+-]+\s*against\s*\()(?:,.*[)\da-
f"]"(?:".*"|\Z|[^"]+))|(?:\Wselect.+\W*from)|((?:select|create|rename|truncate|load|alter|delete|update|insert|desc)\s*\(\s*space\s*
\()(?:\[\$(?:ne|eq|lte?|gte?|n?in|mod|all|size|exists|type|slice|or)\])(?:(sleep\((\s*)(\d*)(\s*)\)|benchmark\((.*)\,(.*)\)))(?:(uni
on(.*)select(.*)from))(?:^(-0000023456|4294967295|4294967296|2147483648|2147483647|0000012345|-2147483648|-
2147483649|0000023456|2.2250738585072007e-308|1e309)$)
```

libinjection

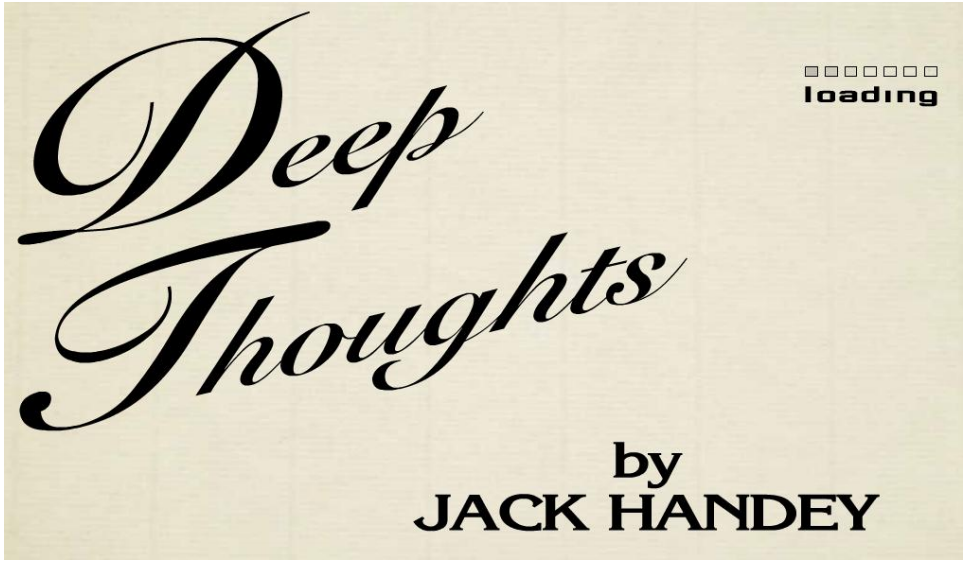
First presented at
Black Hat USA 2012
http://client9.com/20120725

iSEC Partners party at Bellagio

libinjection

- Takes input and create tokens as if it SQL
- Compares first 5 tokens to
  "things that match SQLi"
- 50k+ SQLi samples, some from wild, some hand made, some from scans
- C, 100k checks per second
- Open Source, BSD License
- http://client9.com/libinjection

Why do we have UNION at all?



http://www.deepthoughtsbyjackhandey.com

and what else do I, the developer, *never* use, but *is commonly used* by SQLi attackers?

Let's use libinjection to find features of SQL used in SQLi!

- That's what I wrote in the abstract.
- Turns out to be not necessary
- Used the 50,000+ SQLi samples library from libinjection and ...
- ... the Awesome Power of Grep
  (well ... python regexp actually)

A Highly Unscientific collection of 50,000+ SQLi attacks collected from actual attacks, scanners, how to guides, etc. And doesn't take into account:

- Frequency
- Severity
- Uniqueness
- Actual successful attacks versus probes
- Doesn't look at exfiltration techniques

SQL used in SQLi

`union`	75%
comments (any type)	70%
`concat`, etc	23%
hex number literals	22%
subselects	22%
`chr(),char()`	6%
`aes, hex, compress`	4%
SQL variables	2%

95% reduction in SQLi

By eliminating the following in SQL:

- *unions*
- *comments*
- *subselects*

More than 95% of all SQLi samples were prevented.

98+% reduction in SQLi

- By eliminating the remaining 'unusual' SQL, more than 98% of SQLi samples were prevented or rejected.
- Remaining 2% of SQLi attacks are all equivalent of

  1 OR 1=1

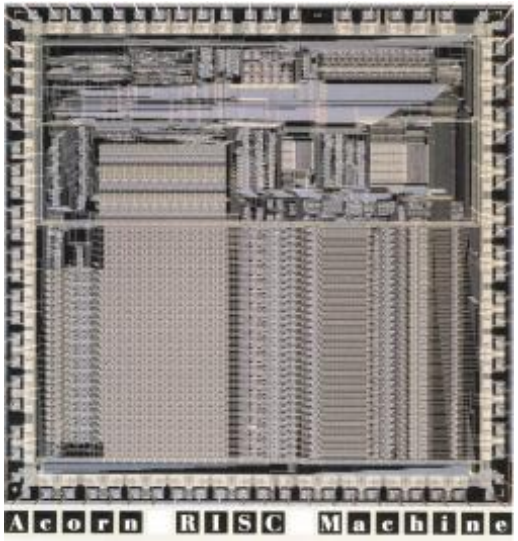
- Those remaining SQLi probes are mostly annoying, and mostly harmless.

Introducing SQL-RISC

I call this "simplified SQL" – SQL that limits SQLi damage -- "SQL-RISC" in honor of RISC computing.

It also sounds cool.

http://en.wikipedia.org/wiki/Reduced_instruction_set_computing



Feasibility:

Can public applications run using SQL-RISC?

Can we replace unions?

- Strongly suspect most? many? websites do not use unions.
- All unions can be rewritten into two queries.
  Minor overhead cost.
- Applications can create 'views' if absolutely required.

Can we remove SQL comments?

- Uhh, I have a hard time getting developers to write *any* comments, let alone comments in SQL.
- Or just allow /* */ only at *start* of query
  (some ORMs generate a comment on where the query is coming from)

Can we replace SQL subselects?

- *All* subselects *can* be re-written as joins , with almost no application level code changes needed.
- SQLi that uses subselects *can not* be rewritten as joins (except in rare cases)

Can we replace SQL string functions?

- Including: substring, concatenation, encoding, encrypting, char functions, replacements
- Easy to move into application logic
- Suspect many web apps don't use these.

Other SQL Functions

- sleep/waitfor/shutdown 3.2%
- in boolean mode 1.7%
- drop|create|replace 1.58%
- rand() 1.359%
- dbms_ 1.0%
- convert() 0.929%
- (updatexml|xmltype) 0.866%
- generate_series() 0.86%
- randomblob() 0.78%
- waitfor 0.46%
- extractvalue() 0.46%
- begin 0.45%
- load_file() 0.29%
- ascii() 0.23%
- nvarchar() 0.11%
- as binary() 0.1%
- into outfile 6 0.01%

Having a hard time seeing any need for them in public applications.

Again, I was going to analyze "real world" benign SQL, but i've never used any of these functions, ever, *on web applications.*

Enterprise Apps

- If SQL-RISC were implemented as a *separate client*, then
  - public apps could use SQL-RISC
  - internal apps could use regular SQL, and use all functions if required.
- This would make adoption much easier.

SQL-RISC Benefits

Fixing SQLi the Old Way

- Ensuring that every user input is properly validated is *intractable*
  (true, some frameworks help, but ....
  only if you are using them)
- Parameterized queries *helps* but some common SQL expressions *cannot* be expressed with parameterization (e.g. IN lists)
- Auditing is very slow
- Every code change may introduce new problem.

Fixing SQLi using SQL-RISC

- Using SQL-RISC *may* some require some application changes, however, this is **a *greppable finite problem***.
- Feasibility, conversion & testing can be done *before* deployment of SQL-RISC.
- Once complete, the entire application, current and future is protected.

Auto-Detecting Attacks

By using SQL-RISC, critical SQLi features are de-activated.

SQLi *Attacks*
turn into
SQL *Syntax Errors*

SQL Syntax Errors are Easy to Monitor

- SQL Syntax errors are annoying but harmless, and are put into logs (database and/or application)
- Trivial to monitor using existing tools
  (e.g. grep 'syntax error' *.log)
- *Now you know where input validation isn't being done*

Next Steps

Proof of Concept Patch

- Making a source code patch to deactivating functions and features should be a relatively simple task.
- *May* be possible to produce a binary patch:

```
perl -p -i -e mysqld
    's/UNION/blah/g'
```

Access Control

- However, best done via access control.
- Different clients could enable or disable SQL functions and features depending on need.
- This is a more complicated patch!

Closed-Source Database Vendors

- I challenge you!
- Provide controls for 'advanced' SQL features or provide a simplified parser option.

ORACLE®

Let's Eat!

- Help wanted!
- contact me nickg@client9.com
- libinjection home page: http://client9.com/libinjection
- these slides: http://client9.com/20130227/