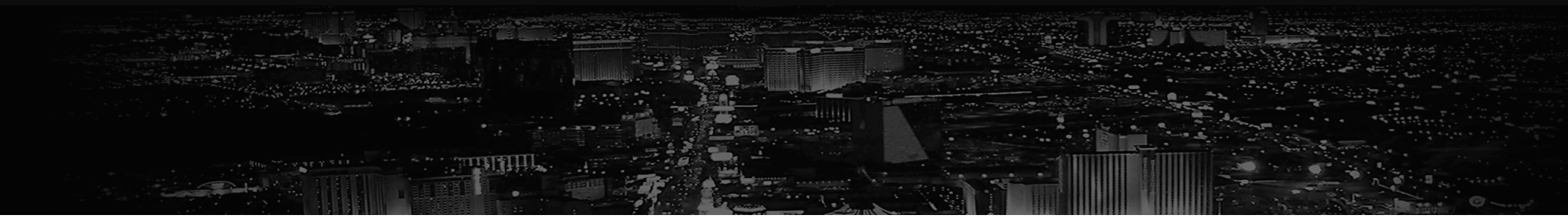
libinjection

a C library for SQLi detection and generation through lexical analysis of real world attacks

Nick Galbreath @ngalbreath nickg@client9.com

Wednesday July 25, 2012 2:45PM
Augustus I+II, Caesar's Palace, Las Vegas



The latest version of this presentation:

http://slidesha.re/0Bch5k

whoami

- Director of Engineering @ Etsy
- Enterprise, Fraud, Security, Email, Fun
- stringencoders:
  - C library for string processing
  - used by every ad server in the world
  - used in Chrome browser
  - http://code.google.com/p/stringencoders

The Next 14 Minutes

- Why is detecting SQLi hard
- The algorithm behind libinjection
- The results
- Next Steps

Detecting SQLi from User Input is a Hard Problem

It's Easy to Get Started with Regular Expressions!

`s/UNION\s+(ALL)?/i`

- At least two open source WAF use regular expressions.
- Failure cases in closed-source WAFs also indicate regexp.

SQL IS HUGE

- Turing Complete! (sorta)
- 1992 SQL Spec: `bit.ly/10fmhZ`
  - 625 pages of plain text
- 2003 SQL Spec: `bit.ly/0B5vfW`
  - 128 pages of pure BNF
- No one implements exactly
- Everyone has extensions, exceptions, bugs

It's more complicated than you think.

- Recursive commenting rules
- A single *number* can't be done in a single regexp.
- Really Loosely Typed
- String rules - OMFG. You think you know but you have no idea.
- Come see my talk at DEFCON this Friday at... 4:20 pm

RegExp Soup

```
(?:\)\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|load|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+returns)|(?:;\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|connection_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\((.*)\,(.*)\)))
(?:(union(.*)select(.*)from))
(?:^(-0000023456|4294967295|4294967296|2147483648|2147483647|0000012345|-2147483648|-2147483649|0000023456|2.225073858507200 7e-308|1e309)$)
```

Nick Galbreath

@ngalbreath

Guns and Butter

- In 2005, right here at Black Hat, Hanson and Patterson presented:
  Guns and Butter: Towards Formal Axioms of Validation (http://bit.ly/OBe7mJ)
- *…formally proved that for any regex validator, we could construct either a safe query which would be flagged as dangerous, or a dangerous query which would be flagged as correct.*
- *(summary from libdejector documentation)*

Existing WAFs

- Visual inspection shows bugs
- Don't see very much in testing
- Don't see much or any false positive testing
- Closed source WAF have zero accountability (e.g. there is no formal disclosure of what they detect or not, and how they do it)

CAN WE DO BETTER?

Nick Galbreath

@ngalbreath

libinjection

Key Insight

- A SQLi attack must be parsed as SQL with the original query.
- "Is it a SQLi attack?" becomes "Could it be a SQL snippet?"....

Only 3 Contexts

User input is only "injected" into SQL in three ways:

- "As-Is"
- Inside a single quoted string
- Inside a double quoted string

(I suppose another would be inside a comment, but we can't do everything)

Identification of SQL snippets without context is hard

- `1-917-660-3400` my phone number or an arithmetic expression?
- `@ngalbreath` my twitter account or a SQL variable?

Existing SQL Parsers

- Only parse their flavor of SQL
- Not well designed to handle snippets
- Hard to extend
- Worried about correctness

... so I wrote my own!

Tokenization

- Converts input into a stream of tokens
- Uses "master list" of keywords and functions across all databases.
- Handles comments, string, literals, weirdos.

```
select 1 /*!00000AND 2>1*/
```

```
[('k', 'SELECT'), // keyword
 ('1', '1'),      // number
 ('o', 'AND'),    // operator
 ('1', '2'),      // number
 ('o', '>'),      // operator
 ('1', '1')]      // number
```

Meet the Tokens

- none/name
- variable
- string
- regular operator
- unknown
- number
- comment
- keyword
- group-like operation
- union-like operator
- logical operator
- function
- comma
- semi-colon
- left parens
- right parens

Merging, Specialization, Disambiguation

- "IS", "NOT" ==> "IS NOT" (single op)
- "NATURAL", "JOIN" => "NATURAL JOIN"
- "+" operator -> "+", "unary operator"
- COS, function, 1, number ==>
  COS, not a function... not followed by (

Folding

- This step actually isn't needed to detect, but is needed to reduce false positives.
- Converts simple arithmetic expressions into a single value (don't try to evaluate them).
- 1-917-660-3400 -> "1"

Knows nothing about SQLi

- So far this is purely a parsing problem.
- Knows nothing about SQLi (which is evolving)
- Can be 100% tested against any SQL input (not SQLi) for correctness.

Fingerprints

- The token types of a user input form a hash or a fingerprint.
  - `select 1 /*!00000AND 2>1*/`
  - `K10101`
- Now let's generate fingerprints from Real World Data.
- Can we distinguish between SQLi and benign input?

Training on SQLi

- Parse known SQLi attacks from
  - SQLi vulnerability scanners
  - Published reports
  - SQLI How-Tos
- > 32,000 total

Training on real Input

- 100s of Millions of user inputs from Etsy's log were also parsed.
- Large enough to get a good sample (Top 50 USA site)
- Old enough to have lots of odd ways of handling query string, etc
- Full text search with an diverse subject domain

How many tokens are needed to determine if a user input is SQLi or not?

5

No matter long the input is.

480 out of 1,048,576 are SQLi

Possible that more token types will be added to help reduce false positives.

Nick Galbreath

black hat USA 2012

@ngalbreath

The Library

- C, logic is under 1000 LOC
- No memory allocation
  (caller makes a copy of input)
- Fixed, stable memory usage
- No threads
- 100k query strings can be checked per second
- Could go even faster

Sample Usage

```
sfilter sf; // on stack, ~500 bytes
const char* ucg = "my user input";
bool issqli = is_sqli(&sf,
                      ucg, strlen(ucg));
// tada
```

metadata on input is in struct sfilter;

(names subject to change, cleanup)

Test Cases

- All input test cases available
- Including false positives found along the way
- Code coverage reports

Python Prototype

- Algorithm in python as well
- Not as up-to-date as the C version
- Working on it
- Runs under PyPy (and quite fast)

Make Existing Systems Work Better

- The Tokenizer could be ripped out, to make a "SQL normalizer/simplifier"
  - all white space normalized
  - all comments removed
  - all numbers in various flavors converted to "1"
  - all strings converted to a fixed value "foo"
- Makes existing regular expressions work better and detect more.

Great for Fuzzers

- The SQLi fingerprints are actually a great source of *templates* for fuzzers and SQLi generators
- Take fingerprint and turn it back into SQL

Available Now on GitHub

- https://github.com/client9/libinjection
- BSD License
  (only to track how this gets used)
- Use it.

Help!

- More SQLi test cases!
- More real-world test cases
- Missing some PGSQL / Oracle string insanity
- Need better understanding of non-ASCII usage
- Porting to other languages
  (it's not that hard).

More Analysis at DEFCON 20

New Techniques in SQLi Obfuscation

SQL never before used in SQLi

http://slidesha.re/MfOiNR

July 27, 2012 Friday, 4:20pm at the Rio

Nick Galbreath

@ngalbreath

https://github.com/client9/libinjection
http://slidesha.re/OBch5k
@ngalbreath
nickg@client9.com
nickg@etsy.com

Thanks for coming by!