AutoRand: Automatic Keyword Randomization to Prevent Injection Attacks

Jeff Perkins
MIT CSAIL
July 2016

Jordan Eikenberry, Daniel Willenson, Stelios Sidiriglou, and Martin Rinard of MIT/CSAIL

Alessandro Coglio of Kestrel Institute

SQL Injection

- Code:

```
String.format ("select … where user='%s'and
                passwd='%s'", user, passwd);
```

- Inputs:

```
user = John'or 1=1 --
passwd = xx
```

- Executed SQL:

```
select … where user='John'or 1=1 --'and
passwd='xx'
```

- Result: Attacker can access information without knowing the password

Isn’t this a Solved Problem?

- Prepared statements are easy to use and prevent injections
- But attacks still occur
  - Many documented attacks since May (Muslim Match dating site, Oracle eBusiness Suite, Drupal sites, and many more)
  - Chinese toy company VTech lost the personal data of over 4 million customers last November
  - Sony break-in alone cost $170 million
- First entry in the CWE/SANS top 25 list
- First entry in the OWASP top 10 list

Coding Solutions are Insufficient

- Require source code
- Require programmer time and understanding
- Can’t be utilized by end-users of the code
- Require Developers to be error free

AutoRand

- Operates on Java byte-code (no source required)
- Fully automatic
- Applicable to large real-world programs
- Eliminates injection attacks without false positives
- Low overhead

AutoRand Concept

- Similar to instruction set randomization
- Randomize SQL keywords (and operators/comment tokens) in the program and the SQL grammar
- Check SQL commands for valid (i.e., randomized) keywords

```
select<key> … where<key> user='John'
or 1=1 --'and<key> passwd='xx'
```

- The token `'or'` is not a valid operator and the command will fail the check. The `'--'` comment token is also invalid

Previous Work

- SQLRand introduced manual randomization
  - Developer
    - Finds each string containing SQL keywords
    - Runs the string through the SQL Rand tool
    - Copies the result back into the source
  - SQLRand proxy checks SQL commands for validity
- Not Automatic
- Randomized keywords may flow outside of SQL (files, error messages, etc)
  - Change the semantics of the program
  - Leak the random key

Automation is Challenging

- SQL keywords may appear in non-SQL contexts

```
String button = "select"
fis = new FileInputStream (button + ".jpg");
```

- The same constant may be used in both SQL and non-SQL contexts
- AutoRand must
  - Add the key to all SQL keywords
  - Propagate the key across string operations
  - Perform all string methods transparently (as if the key was not there)
  - Hide the key from all non-SQL operations (output, filenames, environment variables, reflection, etc.)

Augmented Strings

- Additional characters (payload) can be carried in strings transparently to the program
- Payload is propagated across all string operations
- Payload is identified by a complex random key
- Example operations
  - `equal("select<key>", "select") == true`
  - `concat ("or", " or<key>") == "or or<key>"`
  - `len ("select<key>") == 6`
  - `substr ("select<key> from", 8, 12) == "from"`

Augmented Strings in AutoRand

- Random key is placed after each SQL keyword (no other payload is needed)
- Random key is transparent to the program's normal operation
- SQL processing statements check for valid (randomized) keywords

Correctness

- Transparency

  A given state and string operation are transparent if running the operation in the state produces the same result as running the original operation in the de-randomized state

- Propagation

  A given operation satisfies propagation if each keyword that is propagated from its inputs to its output is consistently randomized.

Transparency

$$op(S) = r^{-1}(op'(r(S)))$$

Where

- r randomizes strings
- r^{-1} derandomizes strings
- $op(S)$ takes a string and yields a string on output
- $op'(S)$ AutoRand replacement operator for $op(S)$

Transparency: Operations on non-Strings

- r^{-1} is a nop on non-strings
- $op'(S) = op(S)$

```
len ("select") == len("select<key>") == 6
```

- Same results on same inputs

```
substr("select *", 6)
    = substr("select<key> *", 6) == " *"
```

Propagation

$$(K_r \in S) \wedge (K \in op(r^{-1}(S))) \leftrightarrow K_r \in op'(S)$$

Where

- *K_r is a randomized keyword*
- *K is the corresponding keyword*
- r randomizes strings
- r^{-1} derandomizes strings
- $op(S)$ takes a string and yields a string on output
- $op'(S)$ AutoRand replacement operator for $op(S)$

Randomization of Program Constants

- Tokenize each string constant
- Replace each SQL keyword with keyword<key>
- Key is 10 upper/lower case letters/digits
- 62^{10} possible combinations (~60 bits)
- Key size is easily configurable

Instrumentation Approach

- Replace calls to string methods with calls to the AutoRand library
- AutoRand library methods
  - Static call taking the receiver as the first argument (call stack is unchanged)
  - Adjust the arguments as necessary (e.g., derandomize)
  - Call the original method or re-implement.
- Instrument both the application and the Java libraries

SQL API Calls

- Intercept calls to the Java SQL interface
- Tokenize the SQL statement
- All tokens that are SQL keywords are checked for the random key
- If any keywords are invalid, an error is thrown
- Otherwise, all of the random keys are removed, and the SQL command executed in the normal fashion

String Manipulations

- Many operations need to be adjusted to achieve transparency and propagation
- Operation categories
  - Observer methods
  - Complete string methods
  - Partial string methods
  - Character methods
  - Misc methods [see paper for details]

Classification by Method

Category	Methods
Complete	<init>, append, appendCP, concat, copyValueOf, toString, valueOf
Observer	compareTo*, contains, contentEquals, endsWith, equals*, hashcode, indexOf, isEmpty, lastIndexOf, length, matches, offsetByCPs, regionMatches, startsWith
Partial	delete*, format, insert, replace*, setCharAt, toUpperCase, trim
Character	charAt, codePoint*, getBytes, getChars, toCharArray
Miscellaneous	Capacity, ensureCapacity, intern, reverse, trimToSize

- String, StringBuffer, and StringBuilder calls
- Similar calls (indicated with *) and calls that differ only in their arguments are grouped together
- CodePoint is abbreviated as CP

Observer Methods

- Do not create or modify strings
- AutoRand derandomizes each string argument and applies the original methods

```
AutoRand.length (String s) {
  return derandomize(s).length();
}
```

- Propagation is not an issue (since strings are not created or modified)
- Transparency is guaranteed since the operation uses the derandomized string

Complete String Methods

- Operate on entire strings, not portions of them
- Entire content of string is propagated (including keys) guaranteeing propagation and transparency
- AutoRand leaves these calls unmodified

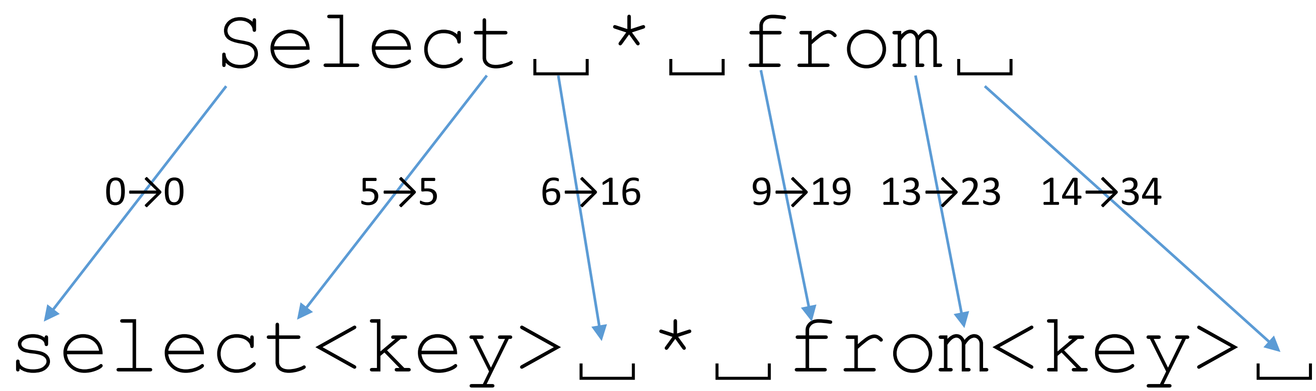
Partial String Methods

- Methods that operate on pieces of a string
- Pieces can be specified by indices or substring matches
    - Matches can be turned into indices
- AutoRand supports these operations by mapping the operation from the original (derandomized) string to the randomized string

Index Map

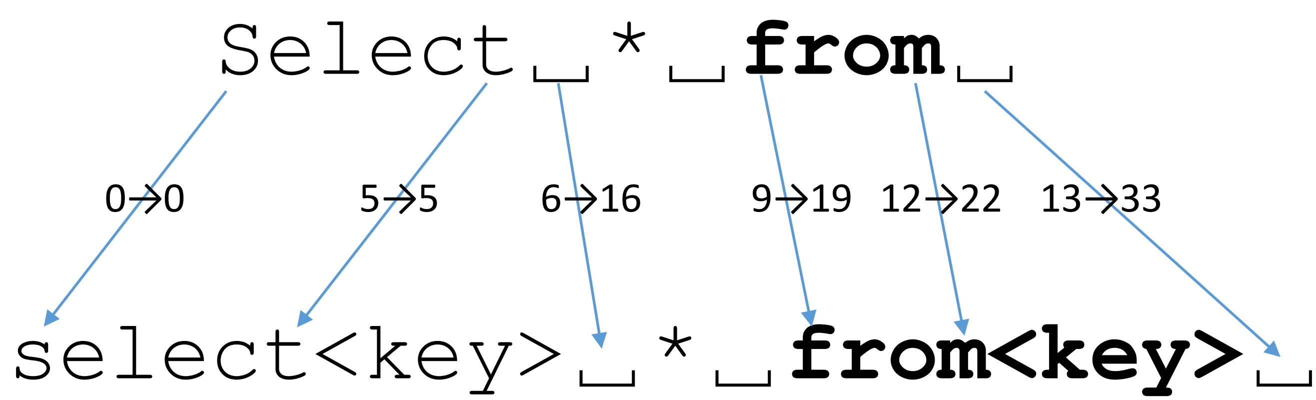
- Map from character indices in derandomized string to randomized string
- Allows any operation over indices to be applied to the randomized string.
- Key operations are substring, delete and insert.
  - All string operations can be built from these
- Guarantees that keys are always included entirely or not at all (with their proceeding character)

Index Map Example



SubString

- Substring from start index (inclusive) to the end index (exclusive)
- Substring (9,13) → Substring (19,33)



Insert and Delete

- Similar to substring
- Delete from start (inclusive) to end (exclusive)
- Insert string before specified index
- Inserts cannot occur in the middle of a key or between a keyword and its key.

Complex Methods: Example Implementation of Replace

```
replace (String s, String target, String repl) {
  StringBuffer sb = new StringBuffer();
  int start = 0;
  int offset = s.indexOf(target);
  while (offset != -1) {
    sb.append (s.substr (start, offset);
    sb.append (repl);
    start = offset + target.length();
    offset = s.indexOf (target, start);
  }
  sb.append (s.substr (start));
}
```

Character Methods

- Convert (portions of) strings to characters, bytes code points or arrays thereof
- AutoRand derandomizes before conversion preserving transparency
- Since the result is not a string, keys are not propagated.

Are Character Methods a Problem?

- In order for propagation to be an issue, the following must occur

$$\text{String(w/keys)} \rightarrow \text{chars/bytes} \rightarrow \text{String} \rightarrow \text{SQL}$$

- Seemingly there is little reason to manipulate program constants in this fashion
- We evaluated this conservatively in our evaluation programs

$$\text{String(w/keys)} \rightarrow \text{chars/bytes} \rightarrow \text{String}$$

Evaluation Programs

Program	Lines of Code	Description
Ant	256K	A Java build system
Barcode4J	28K	A barcode generator
FindBugs	208K	A bug finder
FTPS	40K	An FTP Server
HTMLCleaner	9K	An HTML formatter
JMeter	178K	A performance measurement tool
PMD	110K	A source code analyzer
SchemaSpy	16K	A database inspecting tool

Character Method Calls by Application

Character Call	Applications	Purpose
getBytes	Ant and FTPS	Prepare for stream output
getChars	JMeter (one class)	XML output
toCharArray	JMeter (one class)	XML output
charAt	7 of 8 (12 call sites)	String queries
codePointAt	none	Similar to charAt

- Use is rare in general
- Output is not an issue as the string would be derandomized for output anyway
- String Queries
  - charAt() method is sometimes used to process a string char by char
  - Strings can be built on the information found
  - In each case, however, strings are build based on the indices found in the original string and not from the individual characters

Evaluation

- Independent T&E team developed tests
- Injected Vulnerabilities
  - Real world programs (16K to 256K lines of code)
  - SQL vulnerabilities added to various locations in program
- Smaller focused test programs with vulnerabilities
- Tested with benign and attack inputs
- Correct behavior with benign inputs was checked

Results on Injected Vulnerabilities

- 13 attack variants
- 1 to 43 unique injection locations per program
- MySQL, Postgres, Microsoft SQLServer databases
- 289 distinct test cases (base program * variant * injection location
- 2 benign and 5 attack inputs for each test case
- All attacks were detected with no false positives (or change in behavior)

Focused Test Results

- 17 test programs written by T&E team.
- MySQL, Hibernate and Postgres databases
- Attack types included string tautology, adding syntax to primitive types, and adding comments to primitive types.
- All attacks were detected with no false positives (or change in behavior)

Overhead

- Evaluation Programs
  - One benign test case from each program/variant combination
  - 23 total test cases
  - Overhead ranged from 0% to 15% with an average of 4.9%
- OpenCMS (content management system, 100K LOC)
  - Site content stored in database
  - Captured and replayed 1000 interactions
  - 4.5% overhead

Related Work

- Manual Prevention Error-free coding practices
- SQLRand
- Parse tree structures
  - Attacks modify query structure
- Static Analysis
  - Static data flows
- Dynamic taint tracking (WASP)

Java Taint Tracking: WASP

- Tracks trusted (not untrusted data)
- Instruments applications to use its MetaString library
- Does not instrument Java libraries
  - Loses taint on any string created in the Java libraries
  - Pattern, Matcher, Formatter, etc
- Evaluated only on smaller applications (17K LOC or less)
- 6.1% overhead with no coverage of the Java libraries

AutoRand can Provide Real-World Protection

- Fully automatic
- Source code is not required
- Low overhead (4.9%)
- Verified on large complex real-world programs