Defining Code-injection Attacks

Discussion on prepared statements

1. SQL statements having "holes". The holes have types, and their values are sanitized according to that type.

2. Does not preclude injection attacks, e.g. `"SELECT * FROM User WHERE Name=" + name` is a valid prepared statement.

Discussion on Theorem 9

The theorem basically says that if an SQL generation function copies user input i_m verbatim to the output SQL statement, and there exists an input that results in a syntactically well-formed output, then there exists an input with which the generation function exhibits copy-based $CIAO$.

The proof branches on whether the existing input v_m exhibits a $CIAO$ already, if it does there is the required input.

If it doesn't, then by definition of $CIAO$, v_m must be a substring of an SQL value. The proof then proceeds by case analysis on the type of value v_m and constructs a_m such that replacing v_m by a_m results in $CIAO$.

The proof given in the paper only considers SQL, however, it claims that proof techniques are general. *Can we design similar proof for XSS ?*

Discussion on stack smashing

How applicable is this paper's definition to stack smashing based or ROP based injection attacks ?

Suppose we could parse machine state including return address on the stack after every instruction, would stack smashing constitute a parsing error ?

For ROP, does jumping to the middle of an instruction constitutes parse error in the machine state ?

For attacks when the attacker does not inject code but provides some string that is `system` by the program, it's not code injection.

Other random discussion

Static taint analysis is necessarily conservative, and hence results in false positive. This paper is mainly concerned with *defining* injection attacks, in a way that there are no false positives and false negatives in the definition itself. The enforcement mechanism could err on either side though.

The paper defines values using language grammar, rather than through reduction semantics.

Diglossia: Detecting Code Injection Attacks with Precision and Efficiency

Discussion on implementation

When constructing the SQL query string, do a shadow set of operations that are similar to the actual operations, except the constant characters in the actual language are mapped to a random (and distinct) character set. For example, they choose Korean alphabet as the shadow language.

The user supplied inputs used in constructing the query are left as is. So, in a way trusted strings are mapped whereas untrusted strings are left as is.

Just before sending the query to underlying database, check that:

1. The syntactic structure of shadow query is same as actual query.
2. The code terminals in the query come entirely from the program, and not even a single character from the user input (*enforcing the code injection definition from previous paper*).

The shadow alphabet must be distinct from the language (e.g. SQL) alphabet, else attacker has possibility of crafting inputs.

Does it matter if the attacker knows the shadow alphabet ?

Discussion on limitations

The implementation handles only strings and not integers. As a result, the `Char(40)` attack is undetected. The paper says it's not an injection attack from their view-point. It was discussed in the class that since they do not define what bad things are, their claim is sort of hanging in the air.