

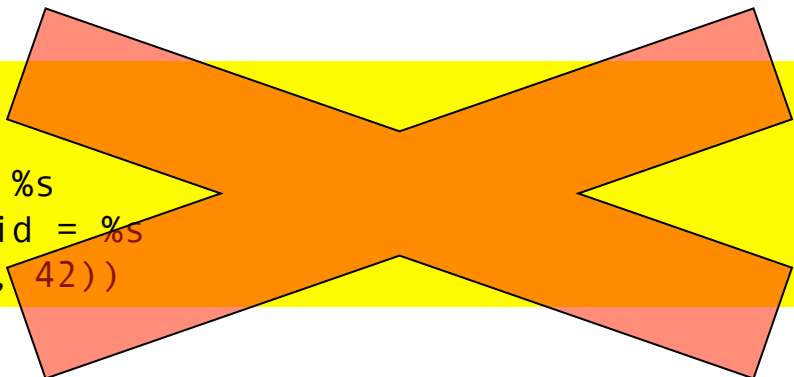
Passing parameters

- This is the tricky stuff.
- Psycopg2 provides specific mechanisms for passing parameters into an SQL statement
 - Especially important when involving modification statements
- You must think differently than usual with Python!
 - Usual process is to interpolate strings using the "%" operator
 - This is very, very bad practice in database programming.

Compare & contrast

```
cursor.execute("""  
update customers  
    set password = %s  
    where customerid = %s  
""", ["sekretWurd", 42])
```

```
cursor.execute("""  
update customers  
    set password = %s  
    where customerid = %s  
""") % ("sekretWurd", 42))
```



SQL injection attacks

- Problem:
 - Untrusted text from an application (i.e., web page) may be inserted without modification into a query
 - If carefully crafted, untrusted text could wreak havoc with security of our site.
- Although we may want to take extra steps...
 - ... using the "execute()" parameter passing mechanism will prevent arbitrary SQL from appearing in SQL statements
- Typical problem example: e-mail address

Problematic code...

- Note query is innocent on the face of it
 - Simple **select**
 - Fetches data from a table
 - **where** clause specifies row(s) when the contact attribute equals the value stored in email_address
 - email_address is a reference to a Python string
 - For our examples, we don't necessarily know how the string's value was set.

```
email_address = "finnthehuman@candykingdom.ooo"

cursor.execute("""
    select somefield1, somefield2, somefield3
    from really_important_table
    where contact = '%s'""" % email_address)
```

Attack (version 1)

- Changing where clause to a trivially true
- Will cause all rows in the table to be returned

```
email_address = "anything' or 'x'='x"

cursor.execute("""
    select somefield1, somefield2, somefield3
    from really_important_table
    where contact = '%s'""" % email_address)
```

```
select somefield1, somefield2, somefield3
from really_important_table
where contact = 'anything' or 'x'='x'
```

Attack (version 2)

- Trying to discover the names of attributes in tables
- In essence, guessing in a way that the query returns information that tells if the guess was right
- (Incorrect guess == syntax error)

```
email_address = "x' and email is NULL --"

cursor.execute("""
    select somefield1, somefield2, somefield3
    from really_important_table
    where contact = '%s'""" % email_address)
```

```
select somefield1, somefield2, somefield3
from really_important_table
where contact = 'x and email is NULL --'
```

Attack (version 3)

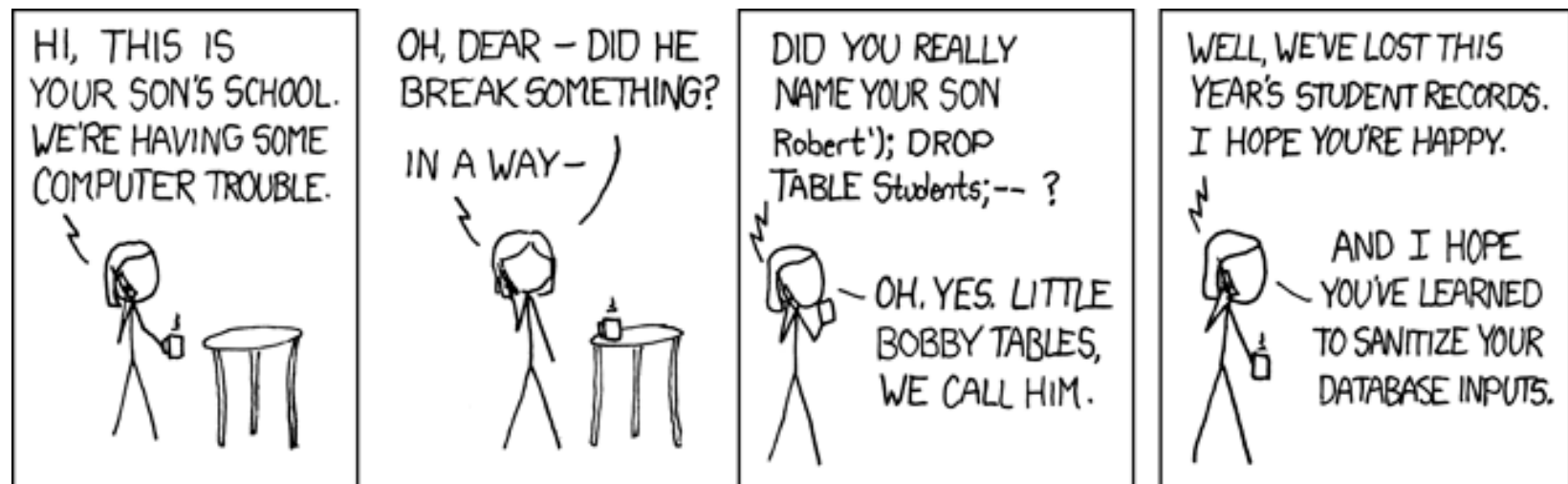
- Damage the database
 - Assumes the attacker knows the names of tables in the database
 - Drop table, view, index

```
email_address = "x'; drop table members --"
```

```
cursor.execute("""  
    select somefield1, somefield2, somefield3  
    from really_important_table  
    where contact = '%s'""" % email_address)
```

```
select somefield1, somefield2, somefield3  
from really_important_table  
where contact = 'x; drop table members --'
```

<http://xkcd.com/327/>



Passing parameters

- Notwithstanding the appropriate method for passing parameters, the details can be tricky.
- Some of this is due to Python semantics around tuples

```
cursor.execute("insert into foo values (%s)",
               "bar")           # won't work

cursor.execute("insert into foo values (%s)",
               ("bar"))         # won't work

cursor.execute("insert into foo values (%s)",
               ("bar",))        # does work as it creates a proper Python tuple

cursor.execute("insert into foo values (%s)",
               ["bar"])         # does work -- easiest is to use lists everywhere
```

Passing parameters

- Python dictionaries can be used
 - Becomes a way to refer to parameters in a keyword fashion.
 - A bit wordy, but it works nicely.
 - Note missing single quotes in the **execute** string

```
# one way (notice use of parentheses and curly braces)
#
cursor.execute("""
    update customers set password = %(pw)s where customerid = %(id)s
""", {'id':42, 'pw':"sekretWurd"})

# another way of accomplishing this kind of thing...
#
params = {'pr':2.99, 'br':'Blue'}
cursor.execute("""
    update sells set price = %(pr)s where beer = %(br)s
""", params)
```

Executemany

- Cursors give us access to the database table
- They also permit us to modify more than one row at a time
- The outermost list will cause **update** to be performed once for each innermost list
 - Innermost lists could also be tuples (modulo advice from a few slides back regarding passing a single parameter)

```
cursor.executemany("""
update customers
    set encrypted_password = %s
    where customerid = %s
""", [[["md5a32d09fcf5072cd0a805d7341792101e", 101],
        ["md5eb3eaa5317d676a51708b95a55e5357e", 102],
        ["md55b49ca8bb67e464746dbdbe011ef6e59", 103]]])
```

Working with types

- There are a range of types available in Postgresql for storing numeric-ish data
 - dates
 - decimals
- Dates can be stored as strings...
 - ... but we would be better off using a proper date type
- And decimals can be stored as reals...
 - ... yet there are representation issues for some decimal values
- Shown in what follows: How to accomplish this when executing SQL commands

Dates and decimals

```
from decimal import Decimal
from psycopg2 import Date
# ... stuff deleted
cursor.execute("""
insert into orders (orderdate, customerid,
                    netamount, tax, totalamount)
values (%s, %s, %s, %s, %s)""",
[Date(2013,11,30), 12345, Decimal("899.95"), 8.875, Decimal("979.82")])
```

```
from decimal import Decimal
from psycopg2 import Date
# ... stuff deleted
query_string = cursor.mogrify("""
                    insert into orders (orderdate, customerid,
                    netamount, tax, totalamount)
values (%s, %s, %s, %s, %s)""",
[Date(2013,11,30), 12345, Decimal("899.95"), 8.875, Decimal("979.82")])

print query_string
```

```
$ ./slide34.py
\ninsert into orders (orderdate, customerid,\n
    netamount, taks, totalamount)\nvalues
    ('2013-11-30'::date, 12345, 899.95, 8.875, 979.82)
```

Nulls, booleans

```
query_string = cursor.mogrify("select %s", [None])  
print query_string
```

```
'select NULL'
```

```
cursor.execute("select null")  
result = cursor.fetchone()  
print result
```

```
(None,)
```

```
query_string = cursor.mogrify("select %s, %s", [True, False])  
print query_string
```

```
'select true, false'
```

Binary data

```
from psycopg2 import Binary
query_string = cursor.mogrify("select %s", [Binary("foo")])
print query_string
```

```
"select E'\\x6666f6f'::bytea"
```

```
query_string = cursor.mogrify("select %s", [buffer("foo")])
print query_string
```

```
"select E'\\x6666f6f'::bytea"
```

```
query_string = cursor.mogrify("select %s",
    [bytearray.fromhex(u"deadbeef")])
print query_string
```

```
"select E'\\xdeadbeef'::bytea"
```

A bit more about dates

```
import datetime

# ... stuff deleted ...
query_string = cursor.mogrify("select %s, %s, %s, %s",
    [datetime.date(2013, 11, 30),
      datetime.time(11, 0, 0),
      datetime.datetime(2013, 11, 30, 11, 0, 0),
      datetime.timedelta(minutes=90)])

print query_string
```

```
$ ./slide42.py
select '2013-11-30'::date, '11:00:00'::time, '2013-11-30T11:00:00'::timestamp, '0
days 5400.000000 seconds'::interval
```


Getting tuples from Python into SQL

- The SQL **in** operator takes a set as an argument
- This set can be expressed as a comma-delimited sequence of values
- Looks a lot like the tuple notation in Python

```
data = (1, 2, 3)

query_string = cursor.mogrify("select * from customers where customerid in %s",
                               [data])

print query_string
```

```
$ ./slide43.py
select * from customers where customerid in (1, 2, 3)
```

Transactions in psycopg2

- The opening of a connection is an implicit transaction **begin**
- A connection is therefore responsible for terminating the transaction
 - commit() method
 - rollback() method
- However, there is some implicit behavior
 - Should any cursor command fail, the transaction will be aborted.
 - Connections can be set to autocommit mode (all commands are immediately executed and no rollback is possible)

```
import psycopg2.extensions

dbconn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

cursor = dbconn.cursor()
cursor.execute("<... some modification SQL statement ...>")
```

Isolation levels

- Example from the previous slide hinted at the way isolation levels may be set
 - ISOLATION_LEVEL_READ_UNCOMMITTED
 - ISOLATION_LEVEL_READ_COMMITTED
 - ISOLATION_LEVEL_REPEATABLE_READ
 - ISOLATION_LEVEL_SERIALIZABLE

```
import psycopg2.extensions

dbconn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE)

cursor = dbconn.cursor()
cursor.execute("...")
....
dbconn.commit()
```

Exceptions

- The DB API 2.0 standard for Python requires a specific set of exceptions/errors
- The hierarchy is shown on the right
- Warnings:
 - Raised for important warnings (e.g., data truncations during inserts)
- Errors:
 - These do indeed "stop the clock"
 - Cover a range of errors, some internal to DBMS, others external

```
StandardError
|__Warning
|__Error
    |__InterfaceError
    |__DatabaseError
        |__DataError
        |__OperationalError
        |__IntegrityError
        |__InternalError
        |__ProgrammingError
        |__NotSupportedError
```

Exceptions / errors

```
import psycopg2.errorcodes

try:
    cursor.execute("boom")
    cursor.execute("select 3/0")
except psycopg2.DataError, de:
    print "DEBUG data error"
except psycopg2.ProgrammingError, pe:
    print "DEBUG programming error"
except psycopg2.IntegrityError, ie:
    print "DEBUG foreign-key check probably failed"
except Exception, e:
    if e.pgcode == psycopg2.errorcodes.SYNTAX_ERROR:
        print "DEBUG syntax error"
    else:
        print "DEBUG something else"

# Have a look at http://initd.org/psycopg/docs/module.html#exceptions
```

Connection pooling

- Connections are expensive
 - Consume space on the server
 - Also consumes time for setup and teardown
- Therefore creating a new connection for every user request can be wasteful (i.e., slows down system)
- Alternative:
 - Create a pool of connections
 - When user request needs connection, use one from the pool.
 - When user is finished, return connection to pool.
 - This reduces the overhead associated with connecting to a database server each individual service request

Colophon

- Some slide material is by Peter Eisentraut from his talk at PostgreSQL Conference East 2011
 - Original presentation slides at www.slideshare.net/petereisentraut/programming-with-python-and-postgresql
- Psycopg2 docs:
 - Includes overview and another tutorial-ish page
 - <http://initd.org/psycopg/docs/>