All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

SQL Injection and Friends

WHAT IS IT?

Directory traversal is almost a ‘path injection’ attack. By controlling path construction, you’re able to walk up the filesystem tree and control where files are being read/written.

MECHANISM

As you may know, on most OS’s there are two special directories: **.** And **..**
The directory . is actually your current path, so /*foo*/bar/./ is really just /foo/bar/
The directory .. is your parent directory, so /*foo*/bar/../ is really /foo/

Take the following PHP example:

```php
<?php
echo file_get_contents('/var/www/sandbox/uploads/' . $_GET['file']);
?>
```

In this case, we have a very simple script: take the GET parameter ‘file’, append that to the path ‘/var/www/sandbox/uploads/’, then read that file and echo it to the page.
Nothing special, right? (Aside from the ease with which you could perform XSS if you control uploads, of course).

But consider the URL: http://badsite.org/get_upload.php?file=../../../../etc/passwd

You’d be reading:
/var/www/sandbox/uploads/../../../../etc/passwd
/var/www/sandbox/../../../etc/passwd
/var/www/../../etc/passwd
/var/../etc/passwd
/etc/password !

IMPACT

As you can see from the previous example, the trivial cases here are ridiculous high-impact: you can read any file on the server that the webserver process has permissions to read.

This could reveal code, database files, personal information, account details, etc etc.

Consider a hypothetical site that allows you to upload and share documents. We already know that fileserving scripts like the one we looked at could very well be vulnerable to directory traversal. What about uploads?
Well, HTTP multipart uploads have a filename in their headers. What if you sent this one?
../../../../../var/www/public/definitely_not_malicious.php

MITIGATION

This is truly one of the easiest vulnerabilities to mitigate, despite that it exists all over the place.
There are two ways to directly mitigate it:

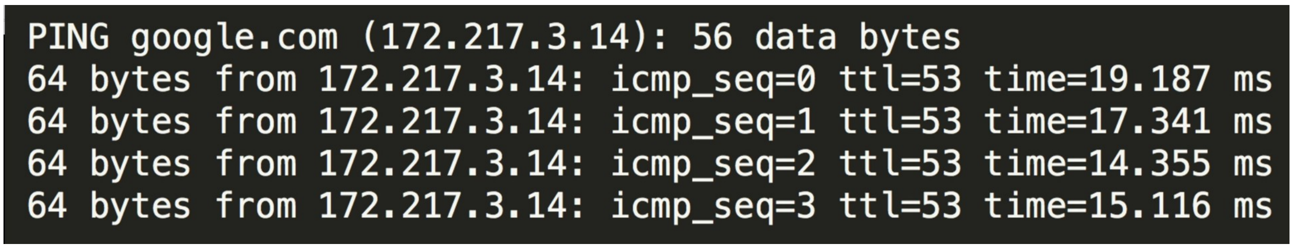
- Don’t allow path separators (/ and \ ) at all, if users shouldn’t be able to read/write/access files outside the path you’re specifying.
- Simply strip instances of ../ or ..\ from paths.

Or there’s the indirect – safer – approach:

Don’t allow user data to control paths whatsoever.
This is particularly relevant to file uploads paths. Generate a filename based on the extension/MIME-type of the file, a md5 of the contents, etc.

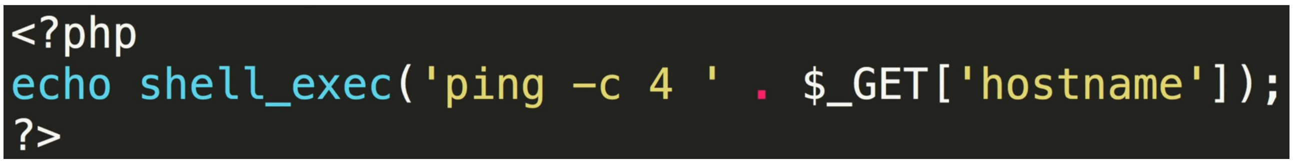
COMMAND INJECTION

You’re testing an appliance with a web interface for administration. As part of this interface, you have access a ping function, to test its ability to call out. You tell it ping google.com and you see:



```
PING google.com (172.217.3.14): 56 data bytes
64 bytes from 172.217.3.14: icmp_seq=0 ttl=53 time=19.187 ms
64 bytes from 172.217.3.14: icmp_seq=1 ttl=53 time=17.341 ms
64 bytes from 172.217.3.14: icmp_seq=2 ttl=53 time=14.355 ms
64 bytes from 172.217.3.14: icmp_seq=3 ttl=53 time=15.116 ms
```

Well, huh. That looks an awful lot like the output that the actual ping command gives you on a UNIX-like system.
What if they’re running the equivalent of this?



```php
<?php
echo shell_exec('ping -c 4 ' . $_GET['hostname']);
?>
```

Well, if that’s the case, then we could potentially run other commands!
So you try to ping the hostname google.com; echo test
And you see that it just silently gives back an empty page. Any use of ; or & does so.

Well, what about backticks? Quick reminder for those of you don’t spend much time at the shell, backticks(``) allow you yo embed a subcommand, whose output gets embedded into the original command.
E.g. ping `echo google.com` will work just like ping google.com

So you try exactly that string, and you see ping output just like you'd expect! From here, you own the system. It may take some work, but you could do anything at all, by embedding subcommands and watching how it affects the output of ping and the shell.

This isn't a hypothetical. This is a bug I found in the Yoggie Pico Pro security appliance, on release day.

You can read the full posting about the bug here, including a proof of concept that swapped the root password for a new one and opened up a SSH on a port of my choosing:
http://seclists.org/fulldisclosure/2007/Jul/20
This was in a brand new security appliance, not something saddled with legacy code and all that. This is by no means a rare case.

MITIGATION

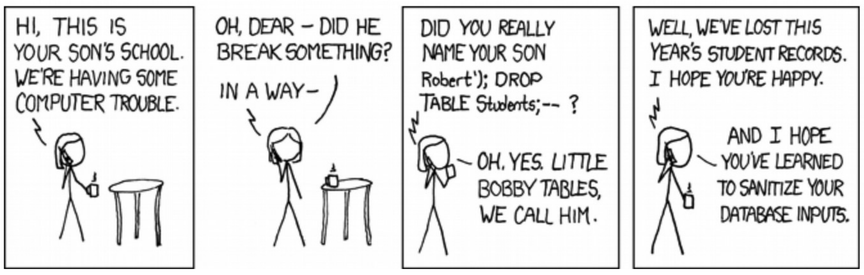
As with most of these sorts of injection bugs, the safest route is to just never embed user data into a command line at all. But if you must, then you should use shell escaping, e.g. escapeshellcmd() in PHP.
That function escapes **#&;`|*?~<>^()[]{}&\,\x0A, \xFF,** and any unbalanced quotes. Note that this doesn't prevent the use of spaces, so if the user input isn't quoted, it could very well add new arguments to the command!

PREVALENCE

It may seem like a rare case that user input would be used for execution of commands. In reality, it's astoundingly common, especially in enterprise-y backend code.
While definitely not indicative of the norm, I once at least one critical command injection. All of different customers doing totally different things.

INTRO

Most of you probably know SQL to some degree or another. In fact, you've probably even seen SQLi.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQL Refresher

For the purpose of this session, let's look at 3 types of SQL queries:

- SELECT some, columns, here FROM some_table WHERE some > comlumns AND here != 0;
- UPDATE some_table SET some=1, columns=2,here=3 WHERE id=5;
- INSERT INTO some_table (some, columns, here) VALUES(1,2,3);

SQL Injection

Applications build up SQL queries as strings, dispatch them to the database, then process the result set (if any).

It's in the query building that you run into issues.

Giving back to little Bobby tables, let's look at what that name may actually trigger.
With this code, we're building a query directly using user input. That makes it a perfect target for SQLi.

```
<?php
$name = $_GET['name'];
$results = mysql_query("SELECT age, grade, teacher FROM students WHERE (name = '$name')");
?>
```

So we put in Bobby's name, and we get the query: SELECT age, grade, teacher FROM students WHERE (name='Robert'); DROP TABLE Students;--')
Which is really three statements: A select, a drop and a comment.

This is really the most trivial case of SQL injection, and it's pretty clear what will happen when this payload executes: the SELECT happens, then the entire table gets DROPped from the system.

MITIGATION

Mitigation with SQL Injection is, generally speaking, fairly easy and you have a few options:

1. Ensure that all strings are properly quoted and run through the appropriate escaping function, e.g. mysql_real_scape_string() in PHP
2. Use parameterized queries
3. Use an ORM for data access instead of direct queries

So if it's as simple as escaping some strings or using parameterized queries or an ORM, why are SQLi bugs still everywhere?
Most web apps will perform dozens – if not hundreds or thousands – of queries. If you're handling these all manually, the odds of getting every single instance correct is slim. And if you have a single bug, that's enough to destroy/exfiltrate data, or even take control of the system.

DETECTION

The most common SQLi you’ll find is in the conditions of a SELECT, so the simplest way to detect the presence of such bug is through the use of two payloads:

- ‘ OR 1=’1 – This returns all rows(constant true)
- ‘ AND 0=’1 – This return no rows(constant false)

These simple payloads can easily identify SQLi

EXFILTRATION

Often, you goal with SQLi is to get data out of the system. There are a multitude of ways to do this, but the simplest is generally a UNION.
Take this query: SELECT foo, bar, baz FROM some_table WHERE foo=’some input’;
That returns 3 columns of data.

So knowing that we have 3 columns of data, we could try a payload such to create a query like this: SELECT foo, bar, baz FROM some_table WHERE foo=’1’ UNION SELECT 1, 2, 3; --’;

This will return an extra row containing the values (1,2,3);
We can use this technique to select data from other tables too, as long as we match column counts.

BLIND SQL INJECTION

We just covered the simple case of SQLi, but what is the ‘blind’ part?
Blind SQLi is when your input is being inserted into a query, but you can’t directly see the results of the query.
For instance, a login page might contain blind SQLi, in that you can only get back whether or not a login has succeeded.

TYPES

There are two types of blind SQLi:

1. Oracles – Where you’re able to get back a binary condition; the query succeeded/returned results or not
2. Truly blind – You see no difference whether the query failed or not

ORACLES

The login page scenario mentioned is a good example of a blind SQLi oracle: the feedback you get is either a successful authentication or a failed login.

TRULY BLIND

Truly blind SQLi is actually fairly rare, but understating how to exploit it is critical. It's been the key of many game-over attacks I've performed in client testing in the past.
A good example of this is a facility that logs web requests to the database as you interact with it. In this case, you'll never see the results of the query whatsoever, nor will its failure impact your use of the application.

EXPLOITING ORACLES

Oracles allow you to answer a question with a true or false. Meaning that you can exfiltrate data once bit at a time.
For instance, you could read the administrator password for a site, bit by bit, to reconstruct it.
You'll generally write a script to do this, rather than performing the attack by hand.

TRULY BLIND → ORACLE

We can't see the query results, but we can see at least one side-effect of the query: how long it takes to execute.
In our log saver example, those INSERTs would most likely be instantaneous in most cases.
But if it's MSSQL server, we can introduce a delay:
By introducing a conditional delay, we can make this blind SQLi into an oracle: 10 second delay if 1, no delay if 0.

Then we can exploit it identically to a standard oracle type, just watching time instead of differences on the page.

DETECTING DATABASE

Often, SQL syntax or functionality differs dramatically across different database engines.
For instance, MSSQL has WAITFOR DELAY as mentioned, but with MSSQL you'd use the BENCHMARK() function to slow down the query and induce a delay.
So being able to identify which DB the application is using is critical to easy exploination.

There are a couple tricks here:

- /*! comment here */ – This looks like a normal comment to most DB's, but MySQL will include the contents of the comment inline,if it has an exclamation point at the beginning
- WAITFOR DELAY will work on MSSQL and fail elsewhere
- UTL_INADDR.get_host_address('google.com') will do a DNS request on Oracle.

REVIEW

- Directory traversal
  - User input is used to build paths
  - Leads to arbitrary file reads/writes in many cases
  - The directory .. is always the parent
- Command injection
  - User input gets injected into a command line that’s being executed
  - Often allows complete system compromise
  - Backticks, semicolons, pipes, and ampersands are your friends here
- SQL Injection
  - User data is put into SQL queries unescaped
  - Can allow destruction and exfiltration of data
    - In rare cases, it can even allow filesystem access or code execution
  - Most typical is that you can see the results of your queries
  - When you can’t see them, you have blind SQLi
- Blind SQLi
  - You can get data out one bit at a time
  - Often slow but always dangerous