Exposition: What is SQL injection?

SQL (Structured Query Language) is a declarative language used for interacting with databases; it is very popularly used in web applications.[1] Databases are filled with two-dimensional tables, which in turn contain a series of entries (organized by row) populated with fields (enumerated over columns). SQL may be used to manage relational databases, where different tables have relations to other tables via shared data, and in non-relational databases, which use other means to access stored data.[2 – 4]

In SQL (and database management in general), there are several basic operations of interest, which may be summarized in the acronym CRUD: [4, 5]

- Create: make new data-related objects
- Read: retrieve the data
- Update: modify the data/objects
- Delete: Remove data

To accomplish these operations, SQL codes comprise series of expressions indicating desired actions. Each expression is a complete statement: an action to be performed on some set of objects. Expressions are composed of clauses, which parameterize the expression and determine the type of result delivered, if any. Some basic keywords are:

- `SELECT`: retrieve a result from an expression
- `DELETE`: remove parts of tables
- `DROP`: erase entire tables, or other objects
- `UPDATE`: change values in a table
- `INSERT`: put new values into a table
- `CREATE`: make a table or other object
- `NULL`: a special keyword indicating that a particular data entry is absent

These usages are representative of basic/core functionality, but are not exhaustive. It should be noted that different operations performing different functions on data or database objects (read, write, alter, remove), require different kinds of privileges or permissions, which define what operations different users are allowed to perform.[6, 7]

Consider a table named `Users` in a database for a web service that requires users to log in. The table stores information on user names in the <`first_name`> and <`last_name`> columns, unique id numbers associated with users <`id`>, and other statistics of interest <e.g. `join_date` = date the user signed up>. The table might look like this:

id	first_name	last_name	join_date	last_login
1	John	Doe	1-20-19	4-21-20
2	Jane	Doe	3-5-17	4-22-20
3	James	Smith	4-3-12	4-23-20
4	Jennifer	Smith	6-10-11	4-24-20

Here are some example commands we could use to retrieve and filter data from this table:

- < `SELECT * FROM Users;` >: get all rows and columns from this table
- < `SELECT first_name, last_name FROM Users;` >: get all rows but only the < `first_name, last_name` > columns
- < `SELECT * FROM Users WHERE last_name = 'Doe';` >: get all columns, but only rows where the last name is 'Doe'
- <`INSERT INTO Users VALUES (5, 'John', 'Smith', '4-25-20', '4-25-20');`>: insert a new row into the table corresponding to a new user 'John Smith' who joined and logged in on 4/25/20

SQL variants generally allow for comments in two forms: text following the `--` symbol, or text between `/*` and `*/`:

- < `SELECT * FROM Users; -- this is a comment` >
- < `SELECT * FROM /* this is a comment */ Users;` >

In the first comment approach, everything after the `--` symbol is deemed a comment, and the symbol terminates the SQL command. In the second approach, only the <`/**/`> contained text is commented, and the comment can be embedded within an SQL command; this kind of comment is the same as in the C language. Note, some platforms support additional comment syntax/constructs, and some platforms allow for interpreting code inside enclosing comments as if it were not commented; see for example MySQL documentation at [8].

These commands are shown to highlight some important features about SQL. SQL is declarative—programmers specify what results they want, and the implementation details are left mostly abstracted away. Thus, notwithstanding the differences between different flavors of SQL, the structural form of SQL scripts will look very similar between applications, and the most basic keywords—`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `DROP`, `CREATE`—are common, if not ubiquitous. The language is relatively simple, and, as one article notes, not unlike everyday English.[9] This means that attempts to exploit vulnerabilities within SQL-related scripts need not be very complex.

Attack Mechanics

Background on SQL injection (SQLi) attacks

SQL is attacked by injection—the placement and execution of (malicious) code into a script intended to operate in ways that contradict the script's intended use.[10] SQLi may originate in data submitted through user input fields (http requests), cookies, or attacks on variables stored on web servers.[9, 11] The purposes of SQL injection (SQLi) vary widely; here are some examples: [10, 12 – 14, 59]

- Reconnaissance on database structure/contents
- Reconnaissance on information to use in other attacks
- Acquire sensitive/valuable data
- Credential theft
- Privilege Escalation/Manipulation
- Denial of service (DoS)
- Arbitrary command execution
- Malware infection/distribution
- Store malicious code to attack users who subsequently interact with the database
- Alter or destroy the structure of the database

The common root of SQLi attacks is that web services provide outlets for users to provide arbitrary, dynamic input (input that is not bounded to a pre-defined set of choices) that is used to execute commands; attackers can use this to change the nature of the commands executed by the SQL engine.

To demonstrate how attacks can be performed, I review some common attack types here. Where possible, I provide concrete examples of statements that result in injections. The set I review is not exhaustive, but it gives a sense of important and widespread forms of SQLi, and some insight into the variety of ways SQL queries can be manipulated maliciously.

In web services that reference an underlying database, it is very common to see statements such as follows. Let's say we have a page in a web service where users can query the database for people by name. The page has fields where users enter in the names of the people they are looking for, in particular first name and last name. Then we might see a statement like this somewhere in the page's source code:

```
query = "SELECT field1, field2 FROM table WHERE first_name = '"
        + user_input_first_name + "' AND last_name = '"
        + user_input_last_name  + "' ;"
```

The '+' symbol joins strings. The text colored in green and enclosed by double quotes are strings. The user entered a first name and last name into fields on the web page, and his input was captured as variables, marked in violet: < `user_input_first_name` , user_input_last_name >; these inputs are also treated as strings. The effect of the above statement < query = ... > is to join all the strings in order to form a query, which will be sent to the underlying database as an SQL command.

When writing out SQL queries, instead of writing out the web-page script that creates the SQL query, I will simply write out a symbolic representation of the resulting query, where user-supplied variables are captured between <> symbols:

```
SELECT field1, field2 FROM table WHERE first_name =
‘<user_input_first_name>’ AND last_name =
‘<user_input_last_name>’;
```

Notice two important syntactic elements of this query:

- User input is captured inside single quotes, so that SQL interprets them as strings
- The statement ends with a semicolon

These syntax rules are enforced for SQL statements; if they are violated, errors will occur. This is important when considering how to frame SQLi attacks.

Let’s say the SQL table in question has some sort of sensitive data—such as credit card numbers, or other secured or personally identifying information—that normally requires user-specific authentication to access. This sort of data is usually of interest to attackers. As our example query we will use this:

```
SELECT id, sens FROM table1 WHERE
uname = ‘<user_input_name>’ AND
password = ‘<user_input_password>’;
```

‘sens’ is a generic column name that represents the sensitive data of interest, and ‘id’ is the id associated with the user in question. table1 is the table where these columns are contained. The intended function of the query is to isolate the *single* row in table1 where both the uname and password columns match user input. Normal user input might produce a query such as this:

```
SELECT id, sens FROM table1 WHERE
uname = ‘John.Doe@123’ AND
password = ‘the*#_WoRd2PaSs^@_#*’;
```

With kudos to John Doe for making a relatively strong password.

SQLi Attack Types

Tautology [10]

Perhaps the simplest form of attack is a tautology, which is so-named because it involves injecting an SQL expression that is always true. Given our above model SQL query, let’s say that instead of a normal user name, the uname field is populated with the following: “’ OR 1=1;--”, and something arbitrary is used to populate the password field. Then the resultant query is:

```
SELECT id, sens FROM table1 WHERE
uname = ‘’ OR 1=1;
```

```
    --AND password = '<user_input_password>';
```

What does this accomplish? The last line is commented out, so it is not evaluated. Normally, the query selects rows where the condition following the `WHERE` keyword evaluates to true, which in normal usage should be only one row. But in this query, the condition is crafted differently. The first part, < `uname = ''` >, is presumably always false—no user name should be empty. However, the second part, < `1=1` >, is always true—meaning that any row passes this condition. Since these two expressions are joined with the `OR` keyword, (false or true) always evaluates to true, so all rows pass the joined condition. Thus, instead of returning the secured information for a single user, this query returns the data for every user in this table—very useful, say, for a thief looking to steal credit card information from a wide swathe of users.

Note that this attack can be used in other contexts—in general, whenever an application is configured to change its behavior when a given boolean condition evaluates to true, whether by returning additional data or executing other commands, a tautology attack may be relevant.[10]

Union [10, 15]
Let's say in our database we have another table, `table2`, which stores other sensitive information for all users under a column called `sens2`. Into the `uname` field we inject "`' UNION SELECT sens2, NULL, NULL from table2;--`". We get:

```
    SELECT id, sens FROM table1 WHERE uname = '' UNION
    SELECT sens2, NULL FROM table2;
    --AND password = '<user_input_password>';
```

What does this accomplish? The `UNION` keyword tells SQL to join the results of two `SELECT` queries into a single result table. Thus, we join these two query results:

- < `SELECT id, sens FROM table1 WHERE uname = ''` >, presumably an empty table
- < `SELECT sens2, NULL FROM table2` >, a new table containing three columns: one populated with all values of `sens2`, and another column filled with `NULL`. The reason we include a `NULL` column is that the `UNION` keyword requires at least that the two query results have the same number of columns; thus, we must populate an additional column in the second query to match the first query.[15]

As a result of this query, we now have unauthorized access to whatever information is provided in the `sens2` column in `table2`.

Privilege escalation [12]
There are many ways for gaining excessive privileges in a system, though sometimes it is as simple as guessing. In the aforementioned tautology example, the function of the baseline query was to retrieve information of interest from a table for a user. However, SQL statements are frequently used to perform other functions, such as login: given a username and password entered on a login page, the application queries a database to determine whether the user may access the service.[13] The query might function like this:

```
SELECT * FROM Users WHERE
uname = ‘<user_input_name>’ AND
password = ‘<user_input_password>’;
```

An attacker could attempt to exploit this mechanism in order to login to an account besides his own. Submitting `“admin’;--”` to the `uname` field may be worthwhile:

```
SELECT * FROM Users WHERE uname = ‘admin’;
--AND password = ‘<user_input_password>’;
```

If `‘admin’` is the name given to an administrator in the database, the attacker now has administrative access to the service.

Piggyback [10]
The examples reviewed thus far indicate that a dynamically constructed SQL query can allow for execution of different types of commands that alter its intended functionality. In different situations, it can be possible for a user to inject any kind of command. Appending arbitrary commands to the intended commands is known as piggybacking.

For example, say that an attacker’s goal was to cause damage to the web service by eliminating its data. Then in the `uname` field the attacker could enter the following: `“’; DELETE FROM table2; DELETE FROM table 1;--”`, and something arbitrary for `password`. Now the query becomes

```
SELECT id, sens FROM table1 WHERE uname = ‘’;
DELETE FROM table2; DELETE FROM table1;
--AND password = ‘<user_input_password>’;
```

Now all the data in `table1` and `table2` have been deleted.

Note, this attack would require that the user in question have the associated permissions/privileges to perform this attack. However, it is not uncommon for databases and enterprise computing environments in general to have vulnerable permission configurations, [16 – 18] and as such this kind of attack may succeed more often than desired. Also note that this kind of attack would not work on SQL platforms that do not allow for executing multiple semicolon-delimited statements in one batch, as is sometimes the case. [19]

Forced Error [10, 12, 20, 21]
Before having access to a database’s tables and the tables’ columns, one must know what the names of these objects are. There are different ways of achieving this—in particular, one may attempt to inject SQL code that calls inbuilt database functions to retrieve lists of available columns and tables—but if this is not an option, another approach is to submit an attack query with the goal of causing an error in the database. If the application is poorly configured, it will respond with a specific error message that names the objects involved in the query, thus revealing information such as the database’s contents and specific platform, and perhaps even the application-level language used as the layer above the database.

Blind [10, 12, 20, 22]

Evidently it is best for web applications to reveal as little information to users as necessary for ensuring proper functionality, and certainly to prevent users from learning about the application's design or the architecture in which it is embedded. Ideally, when data is not returned to the user as a result of a query and an error is not raised, the results of queries should not be known to users. When possible, the structure of queries should also be hidden from users. This is a means of ensuring that users do not have knowledge about the application's internals and have a higher barrier to manipulating its intended usage.

However, even when no information is supplied to users about the results of queries and errors are configured to be generic, SQL injection can still be performed. The tactic is to learn where the opportunity exists to perform an injection, even though one cannot have direct confirmation that an injection succeeds.[10, 12] In particular, an attacker might not know exactly what result is returned by a query, but he still may be aware of whether or not a query is executed by noticing changes in the html content showed by the service or in other aspects of the service's operation. For example, take the baseline query seen before:

```
SELECT * FROM Users WHERE
uname = '<user_input_name>' AND
password = '<user_input_password>';
```

Let's say this query is used on a webpage but hidden from a user, and an error in this query returns a non-specific message such as "*could not process request*." All the user sees is a login form. Additionally, upon successful login no data is returned to the user—the service simply logs the user in. An attacker would not know how to attack this without more information—but if the web page does not check user input, this information can be discovered. Halfond & Orso (2007) [22] provide an example of how to do this. One could submit two boolean queries by submitting two different values to the username field: `"valid_name' AND 1=0;--"` and `"valid_name' AND 1=1;--"`. The resultant query is:

```
SELECT * FROM table1 WHERE
uname = 'valid_name' AND 1=<COMP_VAL>;
--AND password = '<user_input_password>';
```

Where the placeholder `<COMP_VAL>` takes values of $\{0, 1\}$. Now if we assume that `valid_name` is indeed valid, it will not cause an error, and thus the only way an error arises is when the login attempt fails. If the application validates input, both statements should be rejected and raise generic errors, which yields nothing informative for the attacker. If not, then the first generated query still yields an error because `1=0` is false, causing the query (and thus the login) to fail; the second query, on the other hand, performs a successful login because `1=1` is true. Because of this, the attacker would know that the `uname` field permits injection, and he could proceed to submit other malicious queries via this field. [22]

Other blind injection mechanisms exist; a popular one is a timing attack. An attacker submits input that contains a call to inbuilt SQL routines (such as `SLEEP` or `WAITFOR DELAY`) [12, 20] that suspends activity for some specified amount of time, and waits to see if the application experiences a commensurate delay. [12] For example, on a webpage that normally responds very quickly, incorporating the command `SLEEP(5)` into a query would cause the webpage to pause for 5 seconds, thus alerting the attacker that he has uncovered a vulnerable injection point.

Second-order [10, 23, 24]
A second order SQLi attack is not much different than any other SQLi attack; the only difference is when it gets executed. Oftentimes, applications will reuse the results of earlier queries to perform subsequent queries. As one example, when a user logs into an application, the application may use cookies or some other identifier to associate the user with the particular session he has established with the application. Session identification of this nature is highly common among web applications, as it allows services to maintain a sense of state with regard to individual users even though the users may be using static (stateless) pages delivered by the service; this in turn can allow a user to login to a system once, and have the service remember he has already authenticated his session when he accesses other authentication-locked pages. [25] A session identifier can be stored in an SQL database. Let's say a user wants to obtain data from a field `sens` in `table`. Following an example from Oracle's documentation, [24] the process might look like this:

- When the user logs in, set an identifier for that user determined by the system:
  ```
  INSERT INTO OngoingSessions id, uname VALUES
  (<system_determined_id>, "<user_input_name>");
  ```

- To get the user name back, query the `OngoingSessions` table for the name and store it in a variable (the exact syntax is not so important)

- To retrieve the desired information, reuse this user name:
  ```
  EXEC("SELECT sens FROM table WHERE uname =
  '<retrieved_user_name>'");
  ```

The issue is that without proper safeguards, this construct can be exploited. For example, let's say an attacker submits a user name `"nonce' OR uname='John Doe;--"`. [24] The first query becomes:

```
INSERT INTO OngoingSessions id, uname VALUES
(<system_determined_id>, "nonce' OR uname='John Doe';--");
```

There is no error here—because safeguards are not in place, any text entered as a username is treated as valid and made into a string. The second query retrieves this string, and the third query uses it. Being that `EXEC` is used, this is the query that gets executed:

```
SELECT sens FROM table WHERE
uname = 'nonce' OR uname = 'John Doe';
```

So, foregoing whether `‘nonce’` is a valid user name, if there is a user named `‘John Doe’`, the attacker has just obtained his information without authorization. Notice—this particular works because `EXEC` is used in an unsafe way, showcasing again the need for verifying input. Without proper safeguards on input before it enters and exits the database, this kind of attack is notoriously difficult to guard against, [10, 11, 26, 27] and often data coming out of the database is treated as valid and not checked, [28] thus allowing second-order attacks to succeed. As with general SQLi attacks, there are a variety of functions that second-order SQLi can perform.[23] One canonical example uses second-order injection to reset an administrator’s password to an attacker’s choosing.[10]

--

Statistics

This brief section attempts to give some insight into the frequency of SQLi attacks in the wild. Note that this is not meant to provide perfectly accurate statistics, as each data source has its own particular sample, and it is not known how different samples overlap and which should be deemed representative. Still, the data reviewed show that SQLi is a highly active problem in the cybersecurity domain.

The reported frequency of SQLi attacks depends largely on the monitoring capacity of the source and the particular targets monitored; different sources report that out of all attacks, SQLi may represent a small minority to a plurality to a sizeable majority.[14, 29 – 39] In recent years, Akamai appears to have the strongest monitoring capabilities, with a counter-DDoS platform that currently "delivers more than 2 trillion Internet interaction" on a daily basis; [40] in 2015 they claimed to have "visibility into 15 – 30 percent of the world's web traffic." [13] They report attack numbers in terms of malicious requests.[13] According to their reports, in the first quarter of 2015 (2015 q1), there were 52.15M (million) observed SQLi attacks; this rose continuously through q4, when ~125M SQLi attacks were detected.[13, 41 – 43] In following years, SQLi became more popular, with hundreds of millions of SQLi attacks per quarter [44-47]; eventually, within 18-month periods the number of SQLi observed approached 3 billion. [39, 48, 49] A visit to Akamai's Web Attack Visualization page on April 15 of this year revealed that 194,059,850 SQLi attacks observed between Apr 5 and Apr 12 2020, constituting 96.78% of web attacks observed on their platform in that period.[40] Comparisons of the two most popular attacks observed by Akamai over different reports, SQLi and local file inclusion (LFI) shows that SQLi has come to dominate web application attacks:

% of web app attacks	Time	2015 q1	2015 q2*	2016 q1	2016 q2	2016 q3	2016 q4	2017 q1	2017 q2	2017 q3	2017 q4
Type											
LFI		66	18	35	44	40	39	39	33	38	36
SQLi		29	26	47	44	49	44	44	51	47	50

% of web app attacks due to SQLi in quarters with available data. * 2015 q2: the percentages are low due to the introduction of the novel Shellshock attack, which usurped the other two attack's typical dominance in that quarter. [41] Sources: [13, 41, 44 – 47, 50 – 53]

% of web app attacks	Time	Nov 2017 – Apr 2019 [49]	January 2018 – June 2019 [39]	Jan-Sept 2019 [48]
Type				
LFI		24	22	--
SQLi		66	70	~2/3 – 77%

Same as above table, but for different time periods. Notice how far SQLi has gotten from LFI in terms of share of web attacks. LFI data not available for the last time frame.

In these reports and other sources [30-32], the frequency of SQL attacks has swung up and down; many factors, such as advances in techniques on both the malicious and protective fronts, could account for this. However, the general trend is one of immense growth, signs of which have been apparent for over a decade. In the months of 2008 before June, IBM reported blocking 5,000 attacks daily; by the latter months of the year, this had risen to 450,000 attacks per day.[54] In 2014, the Sucuri website security service reported observing 50,000+ SQLi attacks per day.[55] A summary of a WatchGuard report tells that SQLi attacks jumped by an absurd 8000% from q4 2018 to q4 2019.[56] A common theme among these reports is that SQL—which was never the most difficult attack to perform in the first place—has become more easily automated than ever before, enabling attacks on scales previously unknown.[32, 41, 55 – 57] In some industries, automated attacks have become more prevalent than manual attacks.[58] Indeed, in an analysis of SQLi attacks by Imperva, the fewer than 1/3 of IP addresses that attempted multi-day SQLi attacks constituted over 4/5 of requests associated with SQLi.[59] This and other reports that show imbalance in the distribution of SQLi attacks across time and location suggest that automated tools are making it easy for a small share of sources to mount large-scale attacks. [59, 60] A report from Alert Logic in 2018 warns that due to automation, organizations of all sizes and functions can expect to be attacked regularly: "attack automation and 'spray and pray' techniques are aiming at everything with an IP address."[57] It is worth noting that in general, most attacks do not appear to succeed, perhaps because they are aiming blindly; a recent report from Contrast Labs told that 99% of attempted attacks missed vulnerable code.[61]

The targets and functions of SQLi may vary, with different reports disagreeing on exactly which sectors sustain the heaviest attack load. Akamai consistently mentions retail as a prime SQLi target [13, 41, 42, 44], as do some other reports [30, 57]. Healthcare is also mentioned as a significant target for SQLi.[29-32] These industries are particularly apt to being attacked due to the volume and value of the data that can be exposed. Perhaps for political reasons, government sites also are popular attacks targets. [58] Two reports from Positive Technologies (2018, 2019) state that SQLi is consistently the most popular attack vector against IT services; [38, 62] in other industries in 2018, its popularity among other attack vectors was observed to fluctuate throughout the year. [62] Though SQLi appears to be less common than other attacks against the financial sector, [38, 49, 62] some financial organizations have also been attacked via SQLi with great damages involved. SQLi may also play a significant role in attacks on infrastructure and transportation centered services. [29, 35] Unfortunately, none of these industries has a reputation for being particularly secure against the different vulnerabilities that exist.[173] In general, data breaches have been on the rise in recent years, due in no small part to the widespread lack of basic good security practices (e.g. strong password guarding, privileges) [63], as well as classic problems such as insider involvement and physical theft. [32] Different estimates suggest that anywhere from 40% to over 80% of breaches involve SQLi. [64 – 66] Imperva states that between 2005 and Sept. 2011, at least 262 million records were compromised due to SQLi, out of ~312 million compromised by hackers.[66]

The increasing popularity of SQLi attacks over time seems to run counter to reports that cite SQLi vulnerabilities as occupying a relatively low portion vulnerability disclosures in surveyed web applications. [67-70] CVE reports SQL as only 6.4% of vulnerabilities out of all listed on its site[71] One word of caution is that CVE is restricted to publically disclosed vulnerabilities that are reported directly to them, [72, 73] so a number of vulnerabilities are missing [73, 74] (open source projects may be particular underrepresented—one writer calls the coverage “dismal” [74]). But even if the statistics from such sources are reliable enough, the threat posed by SQLi may still remain as high as ever. The lack of new vulnerability disclosures does not affect the abundance of old software running in production (or soon to be deployed) with vulnerabilities, nor the fact that patches are often relatively slow to make their way into deployment environments. Vulnerabilities (critical ones included) may drift in the dark for years on websites or in existing code based before being exploited. [31, 61] This means that even as the number of vulnerabilities in production declines, the danger presented by existing vulnerabilities can rise, [75] especially as automated attacks begin to roam freely around the web. It is possible, as some authors suggest, that successful SQLi exploitation is on the decline [3], but representative data towards this end is difficult to find. For the time being, SQL injection has been and remains a very frequent, highly automatable, and well known attack vector, which if successful can have severe consequences; we cannot yet identify it as a thing of the past.

--

Notable Instances

In this section I review a handful of cases where SQLi has played a role in a notable data breach. The aim is to demonstrate that, despite its relative simplicity, SQLi is used in significant real-world attacks, with potentially costly effects. I also want to draw essential lessons out of these cases that can inform future business insight on cybersecurity.

TalkTalk, October 2015

On October 21, 2015, not a year after a previous breach, UK-based ISP/Telecom provider TalkTalk received word from a security researcher about a vulnerability in their web service. The same day users began to report technical issues, and the company took its website down. Two days later, the company notified customers that a breach had occurred, and that the company had received a ransom demand.[76] The breach was relatively mild in terms of the scale that data breaches can reach, but still significant; it compromised PII from nearly 157,000 accounts, including sensitive banking details from over 15,000 accounts.[77]

The attack was estimated to cost the company anywhere from £30 – £77M (~ $37-$95M). Familiar (missing) security themes emerged from subsequent reports:

- The attack vector was SQLi, a relatively simple attack. The culprits were eventually found to be a group of teenagers and young adults—the attack was not so complex to require professional hacking.[76]
- The company was negligent about security. It had not patched relevant software for three years.[78] Several months prior to the breach, a customer who was contacted by somebody impersonating a TalkTalk employee had warned the company that an attack might be mounting, but he was mostly ignored.[76] As it turns out, two prior SQLi attacks had occurred in July and September, but the company was unaware because it was not monitoring its systems carefully enough.[77]
- The company was unequipped with basic security knowledge. When the breach did occur, the company did not seem to understand the nature of the attack and misreported it.[76]

Sony, May-June 2011

SQLi attacks were the source of a data breach in Sony's systems that attacked online music platforms in Japan and Greece [79] and purportedly compromise information for over a million accounts on the Sony Pictures platform [80, 81]. In a year when Sony had suffered a number of cyber attacks, the hackers' stated aim was simply to reveal how embarrassingly vulnerable Sony had left itself.[79, 80] And, to the hackers' credit, the security details were embarrassing. Sony had stored passwords in plain text; [81, 82] when a Yahoo service was hacked with another SQLi attack later in the year, releasing information for nearly half a million accounts, it was found that 59% of people with accounts for both Sony and Yahoo had the same password for both services.[83]

Gaming & Tech Industries/Army hacks January 2011 – April 2014

In 2014, four conspirators were charged for hacking into the systems of Microsoft, Epic Games, Valve Corporation, Zombie Studios, and the U.S. Army. Their payload was unreleased software and code, [84] which amounted to an estimated $100-$200M in intellectual property [85]. Though different attack vectors were utilized, SQLi was one of the methods used to gain access to the companies' networks, along with the use of company login credentials.[85] That such high-status organizations were able to be hacked through methods as simple as SQLi, and that it took over three years for prosecutions to ensue, raises strong concerns for web security.

RockYou, December 2009

RockYou was a social game site [86] ad network company [87], and widget maker that achieved relatively high prominence between its founding in 2005 and following years when it attained high performance in marketing products on social media sites such as Facebook [88]. Unfortunately for the company, it was hit with a massive data breach in 2009, which was caused by none other than a "trivial" SQL injection.[89] Data for 32 million users' accounts, including credentials for third-party services, were stored in plain text, all of which was made available when the injection occurred.[90] RockYou did not enforce strong password policies—it actually barred punctuation from passwords—and the passwords used were by and large very weak.[89, 91] The company's response was just about as bad as its security measures: it delayed in notifying users, took no corrective action (such as requiring users to make new passwords), provided users with poor or misleading descriptions about how their data were being handled, and falsely stated that only some accounts were hacked when in fact the entire user base had been exposed.[86, 89, 90] Thankfully the company is no longer conducting such insecure practices: it is no longer in business.[87]

The NY Times published an article after the breach occurred, in which it claimed to interview the attacker. The hacker claimed that he worked in a security-related position, routinely attempting to hack web services in an attempt to send a loud message to companies: "Take better care of your customers' data." [90] If we are to believe the account, then he has a good reason for his habit: some third of the services he breached stored sensitive information in plain text, prompting strong concern that companies were not taking cybersecurity seriously. [90]

Heartland Payment Systems et al., 2005-12

For several years vulnerable code allowing an SQL injection was present in Heartland Payment Systems' site—which failed to reveal itself in annual internal audits as well as ongoing system-monitoring procedures—which was finally exploited toward the end of 2007. [92] Through a combination of other tactics, the hackers responsible were able to lurk around the company's systems without any alerts being raised until October 2008; it was not until a few months later that they learned the source to be SQLi. [93] Using tactics to capture unencrypted data while being transferred throughout the organization, they compromised about 134M credit and debit cards, making it the largest breach recorded for its time. [94] The cost to Heartland was great: it lost half of its market value shortly after the breach [92], was barred from processing card payments for some months, and paid an estimated $145M to compensate for the ensuing fraud [94]; the overall estimated cost of this event was $200M [95].

The breach was part of a larger attack including four other prominent companies: Hannaford Brothers, 7-Eleven, and two unidentified large retailers. As with Heartland, SQL injections permitted intrusion into the two retail companies.[96] If this was not enough, some of the criminals involved in these attacks continued to conspire in hacking into the systems of major institutions, including NASDAQ, JetBlue, Dow Jones, Global Payment Systems, and others; this went on from at least 2005-2012, and some companies had malware in their systems for over a year.[95, 97] In total, 17 corporations were attacked, yielding damages in the hundreds of millions of dollars and compromising at least 160M cards.[95] The latest sentences were delivered in 2018 on arrests made in 2012, and it was partially through good fortune that the criminals were caught—they actively posted on social media and left their phone GPS functionality on, making it easier to track them.[98, 99]

Now to be sure, SQL injection was not the only technique used in this conspiracy, and the hackers were highly proficient and experienced;[97] some even visited retail stores in person to conduct reconnaissance on the technology that they would later hack.[95] However, SQLi is explicitly mentioned as the intrusive gateway into 12 out of the 17 organizations involved.[95] Although more advanced malware and tactics were used to steal information and conceal activities after initial system access,[92, 95, 99] largely permitted by the lack of encryption on data being transferred internally within the organization,[92] it was the abashedly simple vulnerability that opened the gates. It states a strong message that hackers with sufficient ability to go unnoticed for months to over a year lurking throughout organization's private servers utilized SQLi as the intrusion method for the majority of the targeted organizations: clearly it can be a highly effective mean for accessing valuable payloads.

This case is interesting not only for its scope and the foundational role that SQLi played, but also because of Heartland's security focus. Cybersecurity was purportedly among the top concerns at Heartland when the breach occurred, and indeed the company had been determined to comply with PCI-DSS before the breach (it was deemed non-compliant afterwards).[92, 94] One source states that the SQL injection vulnerability that allowed the hackers to penetrate Heartland's systems was old news by the time the breach happened—security researchers are said to have reported it to retail organizations in years prior [94]—though I have not yet confirmed this with other sources.

--

Coding Countermeasures

Here I discuss countermeasures that may be taken against SQLi at the code level or in the configuration of the servers running these codes. I will also discuss relevant caveats and issues concerning these approaches.

Simple input verification

An SQLi attack is essentially a deviation in expected input that results in a deviation from expected command execution. Thus the first step in preventing SQLi is ensuring that user input does not deviate from our expectations.

Whether by web-application firewalls (WAF), automated intrusion detection systems (IDS), or instructions hard-coded into an application, web services will often filter incoming input and/or the resultant queries before allowing query execution.[12, 100] They may check incoming requests against a blacklist do not match certain known attack signatures, such as containing SQL keywords, before allowing them to be processed. Whitelisting inverts the logic and filters user input against a set of allowed inputs, rejecting nonconformance; when feasible, this approach is generally much more effective in preventing attacks (reducing false negatives).[12, 101]

For queries that perform operations with native types, such as integers, a very simple verification is simply to check that the provided input is indeed compatible with the required type. For example, if a parameter is meant to receive an integer, it should be checked that it is an integer before being passed to a query. In this way, a value such as `'DROP TABLE Users; --'` could not be provided to this field, because it is not compatible with the integer type.

As it is, the absence of simple checks of this nature appears to be responsible for the majority of injection vulnerabilities, such as SQLi and cross-site scripting (XSS) within web applications. In two studies [102, 103], authors found that simple type-checks would remove 2/3 or more of vulnerabilities in analyzed web applications. The authors also note that such primitive type checking is inherently provided in strongly typed languages, which require objects to have a declared type upon creation; weakly typed languages do not offer this without explicit checks being called.

For more complex types or patterns, which often get converted to strings, more complex checks are required, which can be error-prone. Fortunately, there are many existing web application frameworks to verify that input matches the expected form for types such as email addresses, dates, URLs, phone numbers, and others. Unfortunately, these routines are not equally prevalent across all languages and web frameworks, and even when they are present they are not always used. For example, although in Scholte et al.'s (2012) [103] analysis, only 37 out of 78 open-source frameworks provided support for validating some kind of complex inputs. This might explain why "structured text" accounted for 24% of

SQLi vulnerabilities. However, most of these frameworks (21) applied to the PHP language (very frequently used for developing web applications), and yet of the applications analyzed, PHP was disproportionately affected by SQLi vulnerability: 52% of applications were PHP, and they contained 80% of SQLi vulnerabilities across all applications. Assuming that the aforementioned frameworks were available when these PHP scripts were written, this suggests that security-minded libraries are not sufficiently employed.

Signatures & obfuscation: shortcomings of input verification

If blacklisting routines are not properly implemented or rely on an incomplete set of attack signatures, SQL injections can still slide by.[100] For example, let's say that one attack signature is `UNION SELECT`: the anti-intrusion system rules that any request containing this string should be denied. But what about the same words, separated by two spaces, or a tab, or any combination of whitespace characters? If the system is not configured to handle these different cases, the attack can slip through, and the database will interpret it the same way no matter what combination of whitespaces is submitted. Another subversion technique is to embed comment characters, e.g. `UNION /**/ SELECT` or `UN/**/ION /**/ SELE/**/CT`. Tautologies similarly have an array of signature evasion methods, as do other commands. [100]

A general method of hiding SQL commands is to convert them into an alternate encoding, such as hexadecimal, and submit them in a query that reconverts them into SQL commands via a call that transforms them from this alternate encoding back into plain text, such as `CAST` or `CHAR`. [10, 104] For example, if an input check attempts to verify that certain key SQL statements, such as {`DROP`, `UNION`, `SET`, `DELETE FROM`, `UPDATE`, …}, but lacks certain SQL functions, one might try submitting something like this into a field (the syntax may vary across different variants of SQL):[10]

```
EXEC(CHAR(0x44454c4554452046524f4d2055736572733b));
```

The input in red is an integer encoded in the hexadecimal base, indicated by the leading `0x`. Upon being decoded into a string by the `CHAR` command, this is equivalent to submitting the statement

```
EXEC('DELETE FROM Users;');
```

Which clears the contents of the `Users` table. According to Halfond et al. (2006), due to the different ways different languages handle alternate encodings, this sort of attack can be difficult to detect.[10]

The difficulty of checking for the various evasion methods, which constantly evolve, is why blacklisting is generally not as effective a defense as whitelisting, particularly against knowledgeable attackers. [12, 100]

Incidentally, the `CAST` obfuscation method was used as the core of a malware-spreading bot in November 2007, which was improved and released again towards the end of the year into 2008. Different variants of this attack were early examples of bots exploiting search engines as vulnerability detectors.[105] Explicitly targeting servers running on platforms running Microsoft's SQL Server (MSSQL) and Microsoft's IIS web server, the bot submitted an SQL statement via a `GET` request under an alternate encoding to probe databases for user tables, select columns with type `VARCHAR`, and insert into those columns scripts pointing to malicious website domains that exploited different vulnerabilities in Microsoft systems.[104, 106, 107] Two malicious domains were registered on December 21 and 28 of 2007; by January 1, 2008, each of the malicious domains appeared to have compromised nearly 100,000 sites (fortunately, most sites seemed to rectify the issue rather quickly).[106, 108, 109] Now it should be noted that alongside `CAST`, this attack required storing the casted command in a variable via `DECLARE` and `SET` commands, and then calling the `EXEC` to execute this variable as a command.[104] These parts of the injection input were not inside the `CAST` statement and were separated by spaces (encoded in http requests at `%20` sequences [104]) and semicolons, so even basic signature detection should have stopped this attack. A number of websites were probably not checking input and leaving themselves wide open to injections.

Parameterized queries/prepared statements

The fundamental reason why SQLi works is that input text is transformed into an instruction set to be executed by an SQL backend. If user input were always treated as string literals, injected commands simply would not exist. Fortunately, this feature exists in the form of parameterized queries/prepared statements.[110] In a parameterized query, locations of user input are designated with placeholders, and the underlying SQL system creates an instruction set for how that query will be executed when inputs are provided. For example, in Java one would create such an object by using `?` symbols to denote locations of user input and next filling them in:[111]

```
String q =
   "SELECT id, sens FROM table1" +
   "WHERE uname = ? and password = ? ";

PreparedStatement p = connection.prepareStatement(q);

p.setString(1,'<user_input_name>')
p.setString(2,'<user_input_password >')
```

This ensures that user input is treated as strings (other methods exist to input other data types, e.g. `setInt` [111]—note that this happens to be an inbuilt form of type verification) and that SQLi attacks, such as the tautology seen before, fail:

```
SELECT id, sens FROM table1 WHERE
uname = "' OR 1=1" AND
password = '';
```

Presuming that this combination of user name and password do not exist, this attack yields nothing but a failed login.

The primary reason parameterized queries fail to prevent SQLi is because developers fail to use them rigorously; as one security commentator notes, "If websites were properly coded then SQL injection and XSS attacks would have disappeared long ago." [99] If this approach were employed ubiquitously on data originating from any source—user input, a database, other application components, etc.—before being submitted to SQL queries, direct SQLi would not happen.

Of course, using parameterized queries in only some cases can still allow injections. For example, if parameterized queries are used to handle direct user input, but not when executing commands based on data retrieved from a database, then second-order/persistent SQLi attacks are still possible. They must be used everywhere to guarantee their protective effect.[112] There are also possible fringe cases; e.g., as noted by Anley (2002), if attackers "can exert any influence over the non - data parts of the query string that is run," or if an attacker can exploit other implementation details of objects in the database, attacks can still skirt around use of parameterized queries.[113] These cases, however, are not the norm: SQLi is rather universally a result of user input being blindly trusted in cases where preventative measures (e.g. parameterized queries) are simple and effective.

Note, certain parts of queries cannot accept parameterization, such as names of columns or other objects; however, in such cases it is very simple for developers to verify that (1) the objects being queried are both valid names recognized by the database and (2) are accessible given the user's permissions.[110] We should also acknowledge that there are cases, most likely associated with older versions of different SQL variants, where a query under special circumstances could not be prepared in this manner.[21] However, these cases are likely far and few between.[112, 114] ^One report states, "until recently it was very challenging to bind parameters in SQL queries in PHP code." [115] However, I found a page on building prepared queries in PHP posted no later than 2005, and the syntax did not appear to be very complex; this was, however, particular to PostgreSQL.[116]^

Note also that even with full use of parameterized queries, a propagated attack could still occur if SQL-handled data were sent to a non-SQL component of an application, and that application component did not sanitize or validate incoming data before using it to execute commands. For example, one could pass a valid integer through an SQL interface, with the intent of using it to cause a buffer overflow in another application component. Thus, using parameterized queries in SQL in tandem with sanitization and verification routines distributed throughout an application's components is the strongest guarantee against SQL-derived web attacks.

Indeed, input sanitization should still be seen as the first and necessary line of defense against injections, because oftentimes sanitization alone is fully effective at stopping injections.[117] For example, if an input field only accepts integers, no injection could occur on this field if it were not allowed to receive any characters that were not numeric digits. Under the ideology of using the simplest effective approach, a prepared statement might be overkill in this sort of case. And as there are cases where prepared statements can be inefficient,[110] it is worthwhile to be conscientious of when sanitization would render prepared statements unnecessary.

Alongside parameterized queries, stored procedures are often promoted as a way to enforce security within SQL programs.[118, 119] Stored procedures are much like functions in that they take parameters and store execution instructions that can be reused. The advantage of this is that they can be used to automatically verify inputs before using them to dynamically construct queries.[21] In addition, they can be written in a way that makes them act like parameterized queries, in that they predefine the structure of a query and do not allow deviation from this structure.[21, 120, 121]

The caveat is that, since they are simply stored instructions specified by developers, they must be explicitly programmed in a secure way in order to guarantee secure execution. If programmed insecurely, they provide no security advantage. In particular, a stored procedure can still build a query dynamically and execute it without verifying its structure, allowing user input to be treated as code rather than confining it to be handled as data.[10, 21, 120, 122, 123] This is especially treacherous when stored procedures do not enforce type checks, which is the case in at least some systems (it may be the norm, but I am not aware)—in at least one case, a user took this feature for a language bug.[122] As Halfond et al. (2006) note, stored procedures may also be written in languages beyond SQL, allowing them to harbor other classes of vulnerabilities.[10] Thus, stored procedures can be useful for security, but only when used properly and with awareness of their composition.

Access control

Beyond filtering user input, it is crucial to follow least-privilege principles when granting database permissions to users. The practice of restricting the actions that different users may perform for different data and objects is broadly known as access control. The most basic paradigm is role-based access control (RBAC), a model where users are assigned to specific groups, and roles are assigned on the basis of group.[124, 125]

In terms of access control, an upside of stored procedures is that (in at least some systems) the permissions granted to a stored procedure are not automatically granted to whoever uses that procedure. They act as a command-specific permission token, allowing users to cause execution of specific commands with reserved privileges without them retaining those privileges when executing other commands, and without requiring administrators to manage permission details at the object level (the exact mechanics may vary by implementation).[119, 126] The other side of the coin is that without proper permission configuration, attackers can create stored procedures to execute arbitrary commands against the database.[10, 127]

SQL views offer another coding-based data encapsulation method. Views store results of queries for later access; they can be seen as an element of dynamic programming. They essentially are essentially read-only captures of query results and are created as such:

```
	CREATE VIEW TableView AS SELECT f1, f2, … FROM table;
```

However, they can also be used to restrict access to data. In particular, if users are supposed to access data from only part of a table, then a view may be created from a query on that part of the table, and the dynamically generated query can be run on that view instead of the original table. In this way, even if a user injects commands on that view, he cannot access the original table.[110] In this way, views can also be used as an obfuscation technique to prevent disclosure of sensitive data. In particular, let's say that strange circumstances require a database to store passwords or other sensitive information in plain text. If a user's request will interact with this data, then it is best to encapsulate the data in a view like this (as recommended by the OWASP security community [110]):

```
CREATE VIEW UsersView AS SELECT
hash_func(plaintext_password) FROM Users;
```

If a query on this view is subsequently injected, the attacker cannot retrieve the original passwords because he can only access the view; the hashed passwords are all he can see. In sensitive contexts, OWASP recommends all tables be guarded with views so that no underlying data is directly accessible.[128]

Taken altogether, the tailored use of parameterized queries and input verification, along with proper permission configuration, ensures that data is interpreted solely as data, and that the intended execution path cannot be altered; this is the most effective and efficient way to prevent injections.

--

Automated Countermeasures

Background on the literature

The literature has yielded a variety of methods for counteracting SQLi in web applications. These methods can be seen as pertaining to one or both of two categories: static analyzers and dynamic monitors.

Static analysis is a white-box procedure, meaning it utilizes knowledge about source code; its goal is to detect SQLi vulnerabilities in source code.[129] Pure static-analysis methods act like bug finders, but with security vulnerabilities in mind: they look for structures in source code that suggest a vulnerability may exist. Purely static methods are similar to dynamic signature-based intrusion detection in that utilize known patterns specific rules for finding vulnerabilities; they only look for that which they have been explicitly programmed to find. Although this is a reasonable approach for detecting known faults, there is never guarantee of a complete rule set, and unspecified vulnerabilities can slip by. For particular queries or functions called within a script, it may not even be possible to furnish a clear-cut rule for distinguishing a vulnerability or attack pattern from a valid one.[130, 131] It is also difficult for static methods to know the various ways a query's structure may be altered (even legitimately) at runtime.[130] However, static methods have the advantage of incurring no runtime overhead, since they do not operate in a runtime environment. Hence, even if they are not perfect for catching all vulnerable locations, they are still useful as tools to help programmers be aware of potential security faults and are worth using to automate the kinds of checks that people would perform—so long as they are used with understanding of their limitations. A greater issue arises if static analyzers generate excessive false positives—in this case they undo their usefulness because they distract programmers from the real vulnerabilities.[3, 132 – 134]

The reality is that SQLi vulnerabilities may lurk in code for a long time, if not indefinitely, and will not always be detected; so we must be prepared for SQLi attacks when they happen. Thus the next class of countermeasures does not aim to remove vulnerabilities from code or even identify whether an application is vulnerable—it assumes vulnerability and aims to prevent exploitation of it.[11] The logistics of such methods varies. Some dynamic methods do not consider source code at all; they are entirely black-box, and work by seeing how the target application responds to different kinds of queries (malicious and benign).

Dynamic methods may use the results of static analysis to inform dynamic prevention. They attempt to construct models of the form taken by valid SQL queries, and filter incoming runtime queries against these models. The models may be programmer-specified, or they may be determined dynamically by comparing the structure of SQL queries with respect to parameter-based structure alteration or parse-tree syntax. [10, 11, 132] A general issue associated with such methods is scalability and portability: they are likely to be language- and/or platform-specific, and they require source code access.[135]

Some methods are programmed statically but implemented dynamically: that is, they work by modifying a code base in some way so as to alter how it operates at runtime.[10] As we will see, this will typically involve verification or sanitization functions being systematically distributed throughout the code base so as to prevent the execution of statements that are deemed to be attacks. This may also involve tracking of input that cannot be trusted, due to being derived from user input.

Dynamic systems at the application level are similar to WAF in ways; a key difference is that WAF operate on entire HTTP requests, which incurs overhead.[135] WAF, at least historically, may have relied largely on simple signature-matching methods [100], the downsides of which have already been reviewed.

There is a lack of consensus as to whether static, white-box methods are more effective than dynamic methods—some authors favor static methods [136, 137], while others favor dynamic methods [138, 139]. Dynamic methods will require some degree of configuration to integrate the dynamic component into the target web service—this requirement can be minimal, or considerable. An additional concern for dynamic methods is the overhead that they impose; ideally the overheads are sufficiently small that end users will not be affected.

In this section I summarize articles detailing approaches to counteract SQLi that appear to work well, to not work well, or that I think merit review for some reason. My intent is to provide readers with a nuanced (though not absolutely complete) understanding of the reviewed articles, as well as their challenges, drawbacks, and advantages, without having to read the sources. Thus, the summaries are generally longer and more detailed than one might expect in a general survey. Of course, for additional details, the sources should be consulted. I group the methods under general paradigms to which they identify belonging; there are at least some paradigms of interest or significance not reviewed here.

Black box scanning/automatic penetration testing: the WAS standard

Methods in this group attempt to identify the kinds of vulnerabilities to which an application is susceptible, without tracking the particular source of these issues. They submit input to SQL queries, typically benign and malicious, to test for injection paths. They may have predefined attack patterns or rules for distorting valid queries to produce malicious queries, and their purpose is to attempt attacks and identify the ones that succeed. In order to submit queries, they must able to automatically crawl and extract the various pages comprising a web application, which may require some preconfiguration.[140 – 142]

Software of this sort has been frequently commercialized in the form of web application scanners (WAS). It is popular due to its automated, low-cost nature and its tech-stack-agnostic approach (black-box scanners don't need source code access, and don't even have to be integrated with the web application's technology platform).[143, 144] Khoury et al. (2011) also note that such software is incorporated under the umbrellas of different security standards, such as PCI DSS.[143]

Despite its prevalence, the literature suggests that this model of vulnerability detection generally performs rather poorly. Different studies published on the vulnerability detection rate of different kinds of vulnerabilities suggest that scanners may miss a large portion of vulnerable injection points,[142 – 144] and at least two articles reported detection rates of 0%.[144]

Antunes & Vieira (2009) [142] evaluate the performance of automated vulnerability analyzers—four penetration testing (black-box) products and three static analyzers (white-box)—and compare with the vulnerabilities found by a team of experts. Three of the pentest products were commercial, and one was developed in-house; one of the static analyzers was commercial, while the others were open-source. Altogether, the experts found 61 vulnerable SQL parameters (the metric used by the pentest software) and 28 vulnerable lines (the metric used by the static analysis software). The highest detection rate across the pentest products was 31 (51%), but that product also marked 5 false positives. The static analysis software attained greater coverage—one tool found all 28 vulnerable lines, and another found 23—but in exchange the false positive rates were higher (roughly 25% for all three analyzers). The authors caution that the results do not necessarily generalize, but still may offer insight into web-service scanning tools.

Fonseca et al. (2007) [141] take a unique method to testing web-scanning software: they actively inject faults into existing software. They review the performance of three common web scanners on two web applications, one of which they created themselves, and another from the web. Due to the size of the second web application, results were available for only the first. Their analytical method is as follows: for a given web app and scanner,

1. Scan the app for vulnerabilities before manipulating it;
2. Identify places for insertion of faults in the code of the web applications;
3. One fault at a time, start with fresh (non-tampered) code, corrupt the code to create the fault, and see if the scanner detects it.

Although their method holds promise, their data are incomplete and their presentation is confusing in places, making it difficult to extract results. Although they include two web apps in the study, they only had full results for the one that they created—the other one had too few data available to present any summary here. For the first app, of the 81 artificial vulnerabilities they injected, the three scanners collectively detected all vulnerabilities, but individually only found 41%, 25%, and 80% of them. [141]

The most (though not necessarily fully) generalizable study comes from **Vieira et al. (2009)** [145], who tasked four commercial web scanners (two of which were different versions of the same product) with evaluating 300 services randomly selected from a diverse set of 6180 openly available online services. The four scanners collectively identified 149 unique SQLi hotspots—many more than detections of other kinds of vulnerabilities, such as XPath or code injection—but the coverage distribution was highly skewed: 87% and 95% for two scanners, and under 20% for the other two. There are two reasons why these numbers are almost surely overestimates of the true coverage rates:

- These statistics include both confirmed findings and doubtful findings (cases of SQLi that the scanners detected but the authors were unable to verify).
- The authors did not have access to source code, meaning that the true number of vulnerabilities is likely underestimated.

It did not help that the two scanners with the highest coverage rates also had the highest false positive rates—40% and 37%, even without counting doubtful cases as false positives.[145]

Why do WAS generally perform poorly and often fail to attain high overlap on the vulnerabilities they find? For starters, Fonseca et al.'s article provides a hint. Although they do not disambiguate the distribution of SQLi vulnerabilities among the different fault types in their article, from the information they do provide, we can surmise that at least 48% of contrived SQLi vulnerabilities were attributable to a single fault type, denoted MFCE, "missing function call extended"—a type of error where a function is supposed to be called to process a variable but is not called, which can affect how the parameter is later used. [141] It is conceivable that this kind of vulnerability is inherently more likely to permit SQLi, as missing sanitization function calls are one of the fundamental reasons why SQLi succeeds. One may also wonder whether web scanners are configured to be most sensitive on this kind of vulnerability, thereby being likelier to miss other kinds. The three scanners collectively found 39 SQLi associated with MFCE faults; one scanner found 36 of them, another found 16, and the last found only 5. Likewise, other fault types demonstrated variability in detection rate by different scanners. [141] This suggests that different scanners are configured with different rules, which may be inadequate to cover all or even most injection points.

Attempting to answer the question of "*Why Johnny Can't Pentest*," **Doupé et al. (2010)** [143] designed a realistic, reasonably complex web application called WackoPicko. The application supports for authentication, uploading files, making purchases, commenting on images, performing search queries, a guestbook, a calendar that could generate endless new links dynamically, and administrative-only pages. The realistic nature of the service is meant to exercise web scanning services with the sorts of challenges they would face in real web services—crawling in and out of login portals, miscellaneous page interactions, and processing dynamic content such as JavaScript and Flash. The application totaled 16 vulnerabilities, 2 of which were related to SQL injection, and 11 scanners were tested on it. The scanners could be run without any input from people, with minimal input (providing a valid username/password for logging in), or they could be directed how to handle certain parts of pages in a manual operation mode.

Foregoing the disappointing vulnerability detection rates—which paled in comparison to the nearly perfect detection rate of a group of students "with average security skills"—the authors wanted to know what challenges the scanners encountered when crawling the services. They found the following:[143]

- Six scanners had problems of one sort or another parsing HTML.

- No scanners could upload a file automatically, and only two could do it after being shown how to do so in manual mode—some could not even upload files in manual mode.
- Using a separate tool (WIVET) that analyzes how well crawlers can handle links in various formats, including JavaScript, Flash, etc., 7 scanners handled 50% or less of JavaScript cases, and four scanners appeared not to parse JavaScript dynamically at all. No application passed WIVET's Flash test.
- Only five scanners could create an account, and only one could use this to exploit a vulnerability.
- Only two scanners could automatically create comments, though they did not create attack strings; when shown manually, only four scanners produced different comment strings than users, and only one formed a successful attack.
- One scanner got caught in an infinite loop trying to process the calendar, and a few others extracted hundreds or thousands of URLs from this feature before recognizing that it would lead nowhere.
- Many scanners had to make hundreds or thousands of vulnerable requests to vulnerable pages, suggesting they were working in a circular or imprecise (at least inefficient) nature.

Khoury et al. (2011) [144] revisited WackoPicko, along with 2 other simpler applications, probing for three commercial scanners' ability to exploit specifically second-order SQLi. They used only scanners where vendors explicitly claimed coverage of this kind of vulnerability. The scanners could be run in a mode to only look for second-order SQLi (persistent mode), only look for blind or reflected SQLi (B/R mode), a mode to look for all SQLi opportunities (Full-SQL mode), or a mode to look for any vulnerability type (full mode). Wireshark was used to monitor network traffic, and the number of unique stored attack codes was tracked.

The authors find that when configured to look only for persistent SQLi, the scanners did not know what to do—there was little to no traffic associated with pages storing and executing attack codes. There was more traffic on both page types when the scanning scope was expanded to B/R or Full-SQL mode. Furthermore, on the simplest application in the test suite—which required no authentication whatsoever—very few attack codes (6) were stored in both Full-SQL and B/R modes, and absolutely no traffic was observed on pages where stored code could have been executed. Similar to Doupé et al.'s results, the web scanners had trouble creating accounts; and when they did, they never used the accounts to trigger exploits. By and large they stored attack codes but did not summon these codes to be executed. [143, 144]

From both articles, the authors conclude that web scanners are not developed to handle the complexity of modern web applications, which are growing more complex with time. As per their findings, the applications need to be used wisely—users must know what sorts of input and configuration parameters (e.g. login directions) will help effectivize the software in question—and the software itself must be made more advanced.[143, 144] In particular, the applications need to be able to interpret applications as systems that can change state (e.g., logged in/not logged in), and keep track of the interactions they have had with this system. [144]

All in all, these articles tell that no WAS is in itself guaranteed to catch all (or even most) of injection fault points in an application; the black-box nature of such tools requires that users be very aware of how they behave and how they can be configured in order to be of any use.

Information flow

Any attack proceeds in stages: there are entry points and checkpoints en route to the payload. Any analysis of vulnerability within a system must consider the access points to that system, as these are the gateways to every other point of the attack chain. We also must be aware of where the vulnerable points in our program lie, and know what communication these regions maintain with the entry points—this lets us know what data we can trust and what must be verified before interacting with deeper or more secure layers of our system, such as the database.

This is fundamentally a problem of information flow across trust boundaries. Information flow is central to computer science, and has found numerous applications in security—generating filters on user input, detecting malware, and uncovering vulnerabilities in web applications—in particular injection vulnerabilities such as XSS and SQLi.[146] Two relevant problem domains are examining abstractly the possible paths through program execution, and monitoring data at the various points of execution to determine what can be passed forward and what cannot.[136, 140, 146]

A number of data properties can be monitored; for SQLi and similar vulnerabilities we want to know whether data is trustworthy (benign) or not (malicious). The process of determining this is known as taint analysis.[136] In a nutshell, taint analysis may involve these steps, statically or dynamically:[136, 139]

- Determine the source of untrustable/trustable data (*sources*);
- Assign taint information to data at a chosen granularity, marking untrustable data as tainted and trustable data as untainted;
- Propagate this information throughout control flow—i.e., track data that originates from other tainted data, and make sure it is marked as tainted;
    - In static taint analysis, the solution must be able to trace the target software’s possible execution paths;
- Assess the sensitive points where untrusted data must not be allowed to pass (*sinks*), and the possible pathways for tainted data to these locations;
- Identify policies that allow tainted data to be marked as safe (*declassification*);
- Handle untrusted data before it gets to any sink—either by sanitizing it, rejecting it, or interrupting execution.

The exact nature of how each step is executed depends on whether the analysis is run statically or dynamically. Additionally, each of the components mentioned above may have differing levels of granularity—e.g., whether any and all data in contact with tainted data should be marked as tainted, or whether there are more precise ways of telling when taint spreads throughout a program.

Here are some general challenges or considerations to address in taint analysis:

- *Control flow*. Programs of any measurable complexity can have (computationally) intractably many execution paths, and it is no simple task to automate the exploration or reduction of those paths (at least, in any efficient manner).[28, 139, 140] Implicit transfer of information (typically occurring when variables are related by condition rather than assignment between them) presents an extra challenge.[138, 139] Indeed, of 17 projects between 2005-2012 that applied dynamic tainting to runtime-embedded languages, only one tracked implicit information flow.[139, 147]
- *Sanitization*. There are some embedded sanitization in commonly used services and frameworks; however, application-specific sanitization procedures may still be required in many cases.[139]
- *Declassification*. While sanitization has some standardized support, thus far there is no standard way of marking tainted data as non-tainted, nor is their a consensus on whether this should or can be standardized.[139] It is a highly context dependent problem; automated attempts to declassify data may be too lenient.[136, 148]
- *Reducing false positives*. Imposing trust boundaries on data is necessary for security; but labeling everything as untrustworthy will prevent normal application usage. This is an active problem in taint analysis, both static [136] and dynamic [139]. This might especially be a problem in the context of implicit information flow, where variables hardcoded with benign values can be conditionally related to variables of untrusted origin.[138]

Here are some particular approaches presented in the literature:

Haldar et al. (2005) [136] provide a basic implementation of tainting for Java that handles different kinds of vulnerabilities and attacks, including SQLi. It applies to any version of the Java Virtual Machine (JVM) and considers only bytecode, not source code. Their methodology modifies the inbuilt String, StringBuffer, and StringBuilder classes to track taint at the string level by storing a taint flag with every string instance. Strings that come from external sources are tainted upon instantiation; everything else is not. In this way, developers do not have to track taint themselves. However, because developers cannot specify sanitization methods to be embedded in the class, they must apply those sanitization procedures manually. This feature, plus the lack of access to source code, means that automatic declassification cannot be done precisely; the authors assume that anytime a method is used that invokes a check or match operation on strings—e.g. a regex method—it properly applies sanitization, and the resultant string can be deemed safe. This is a substantial shortcoming, as it is well known that the checks and sanitizations that developers apply are often flawed or evadable.[10, 100] In addition, it is quite conceivable that simple string checks can be applied which have nothing to do with checking for safety. Although their method successfully stopped some attacks on an overly simple application suite—including an SQL injection—it needs more testing and refinement before being of substantial use in real production environments.[136] It is, for now, an initial proof of concept.

Xu et al. (2006) [138] present a program that modifies source C code in a way that allows taint information to be tracked at runtime so as to guard against various kinds of attacks, including SQLi. It can apply to C programs and to languages whose interpreters are written in C, such as PHP & Bash; and in the case when source code is not available or embedded functions are implemented in assembly, the authors provide means for issuing warnings and maintenance of taint information across such paths. The program marks functions that read data from external sources as untrusted, and allows users to enumerate security policies for specific functions which are automatically invoked to verify incoming data before the function is executed. The baseline policies are overly simple—e.g., that for SQLi simply checks for the use of special SQL characters in input strings, but does not consider keywords or other issues—it is presumed that more complex policies will be applied in forward usage.

The novelty of this approach is that it tracks taint at the byte level—each byte of memory has an associated taint entry in a corresponding taint array, which avoids complexities associated with handling taint at an object or variable level (e.g. memory corruption, type casting, and aliasing).[138] The authors provide a proof of concept by showing that their implementation is able to stop each of 9 exploits tied to CVE-listed vulnerabilities, among them an SQL injection, while operating on different (C-based) languages and large source files—but until adapted and applied in more realistic environments, it remains a proof of concept.

The authors note a number of advantages to their methodology—managing of byte-level taint information, ease of use, lack of need for end users to have deep knowledge of source code, and flexible policy specification. They mention that more realistic policy specifications are needed, but they focus on the contribution they make to automating the process of tainting data, and doing so with precision. My only caution would be not to underestimate the importance of this component, or the expertise it will require. Parsing and validating query structure is not a trivial task. Indeed, the authors propose an improvement to the aforementioned default SQL plan that involves creating a parse tree of an SQL query and propagating taint throughout it.[138] However, as other authors observe, this might introduce substantial complexity and require knowledge of the parsing structures of different database systems.[149]

Nguyen-Tuong et al. (2005) [137] develop a modified PHP interpret that inherently tracks taint at the character level. They specifically target SQL and XSS. It incorporates the standard tainting components: identifying untrusted sources, propagating taint, and verifying/sanitizing potentially tainted data before passing it to sinks, or otherwise rejecting it. However, their approach is unique in several ways. They are mindful of second-order XSS attacks, and so their taint analysis is applied to output HTML pages as well as incoming data. They take a conservative policy and automatically label data coming out of the database as tainted, relying on the precision of their taint tracking to untaint normal data being retrieved. They cite this as an advantage to overcome the “ad hoc filtering approaches” that can permit attacks. When it comes to SQL, the policy is simple, but targets the core features of an attack: mark a query as tainted if tainted characters are found either in keywords or in operator symbols (commas, parentheses, etc.); literals and primitive types are never marked tainted. The advantage of their approach, as they describe, is that developers need not remember to check the various sources of malicious commands—GET/POST request, variables, HTTP headers, cookies, etc.—it is handled automatically. Additionally, they report that preliminary investigations yielded overheads under 10%. The major shortcoming of this article is that the concept is never taken to proof—we do not get an empirical evaluation of how the interpreter fares against attack attempts.

Kieżun et al. (2008) [28] present ARDILLA, a PHP-targeted program designed to use taint analysis in conjunction with other methods to detect SQL and XSS vulnerabilities in programs. In particular, the method involves three stages:

1. Input generation/control flow mapping: ARDILLA attempts to iteratively generate inputs that automatically uncover new pathways for execution;
2. Taint propagation: the standard steps of identifying sources, propagating taint, and monitoring sinks
3. Input mutation/attack generation: ARDILLA alters benign statements to produce ones that may cause an attack, and submits these statements to see if they succeed in achieving an exploit. Parse trees are used to detect successful attacks.

ARDILLA offers the following features:

- It is white-box and does not modify existing code.
- Taint information is carried with data into and out of the database—data that is tainted is saved into the database with taint information included, so that when it is retrieved, the runtime system is alerted and prepared to prevent an attack. This, the authors propose, is a smarter approach than simply treating all data coming out of the database as trusted (as is commonly done) or untrusted (as done by some studies, e.g. Nguyen-Tuong et al. (2005) [137]).
- It does not require sanitization methods to be passed, though users can do so.

This is the first tainting article considered where a more realistic evaluation is provided. The authors evaluated ARDILLA on five open-source programs of varying complexity (~300 – 8200 lines of code), though they do not tell how they selected these apps. Across the apps, line coverage was low—on average, under 50%—and on a larger program not included in the study (~35000 lines of code), coverage was only 14%. The tool uncovered 366 SQLi sinks, but the input generator only uncovered pathways to 91 of these sinks, and only 76 had an attack conducted. Despite this, 23 previously unknown SQLi vectors (as well as a number of XSS vectors) were uncovered. The authors note this is a superset of the vulnerabilities uncovered by a popular WAS. There were no SQLi false positives (confirmed by manual inspection of all of ARDILLA's results), though there were some XSS false positives. [28] Despite the fact that there are obvious improvements to be made to the tool and more evaluation work to be done, the authors made a strong demonstration of its potential to detect vulnerabilities. Future work might evaluate the tool against a known set of vulnerabilities to perform true coverage analysis.

The last tainting project considered here is WASP, a dynamic tainting tool targeting exclusively SQLi attacks in Java applications, presented in **Halfond et al. (2006)** [148] ^Incidentally, I have found another page with a project entitled WASP dated from 2004, [150] predating the 2006 WASP article; the 2004 project is also a cybersecurity program meant to perform the same kind of functionality as Halfond et al.'s (2006) project, except it applied to SQLi and a range of other web-based attacks, and was implemented for PHP at the time with the intent to port to other languages, such as Python and Perl. I am not sure of how complete the tool was, or how it was implemented. The author has been involved in security for a number of years.[151] I am not sure what if any relation the projects have, but their parallels seem a strange coincidence.^. WASP consists of two components. The first involves the standard tainting procedures at the character level, but unlike in most other tainting studies [139] this tool implements positive tainting. Positive tainting marks everything as untrusted by default, and then proceeds to determine what can be explicitly verified as trustworthy; negative tainting marks data as trusted unless explicitly marked untrusted.[148] The authors believe this is a must because there are simply too many SQLi attack vectors for developers to reliably track. To overcome the apparent strictness of this approach, WASP allows developers to manually specify "trust policies" that demarcate trusted sources and the ways data from those sources may be used.

The second component is "syntax-aware evaluation." In short, similar to Nguyen-Tuong et al.'s approach [137], WASP only focuses on non-literal components of SQL queries when performing classification.[148] This is proposed as a more sensitive tactic than simply marking strings as untainted when passing through sanitizations, which still leaves opportunity for missed attacks. In particular, WASP dictates that non-literal strings contain only trusted characters—anything else is deemed an attack. As an additional measure, WASP uses a database parser to counteract attacks that attempt to use alternate encodings to subvert detection.

With the help of a graduate student with experience in SQLi, penetration testing, and web scanners, the authors prepared a sizeable set of queries to use in evaluating the tool. They applied WASP to a total of 7 services averaging ~9600 lines of code each; five were commercial and two were developed by students. They automated the process of injecting flaws and logging results, and did not test servlets that required complex interactions or unique session data, such as cookies. Without WASP, there were 28833 attack attempts, 12616 of which were successful. The authors also included 5309 "interesting" valid queries for submission—queries that contained SQL keywords, operators, and special characters, meant to demonstrate that WASP could handle tricky input that would trip up naïve filtration methods. When WASP was applied, every successful attack was caught (none were allowed to execute), and no valid queries were denied.[148]

The authors admit that there is more work to do, in particular putting the tool to work in a real environment where attacks are directed at websites from the wild, and investigating optimizations.[148] However, they appear to have a strong foundation, and WASP is advertised as a lightweight addition to existing code—the only change required is bytecode modification to support the custom string-operating components of WASP that the authors developed. Thus if similar tools could be applied to languages beyond Java, this would also greatly increase its usability.

Query data modeling

Another paradigm for evaluating SQL queries approaches the problem from a more information-theoretic perspective: what is the expected amount of information needed to convey a given query or query result, and can we detect injections as deviations from these expectations? Information theory is a broad domain with many applications and relations to other fields; in computer science, a foundational application is the study of the minimal amount of information needed to convey given information, or to convey a deterministic program that will generate this information.[152] Other applications of this domain to computing and cybersecurity include detecting malicious worms, measuring software complexity, enhancing data sharing/networking protocols, and others.[153]

I find this paradigm inherently interesting and, as an injection detection framework, unique. I also find it intuitive for the following reason. When I was reading articles about trying to detect SQLi by structure/parse analysis, I realized that this is a somewhat complex task and pondered the possibility of a simpler solution. One that came to mind was simply measuring the amount of information returned by queries compared with the expected return information size, leading me towards the information-theoretic approach.

Unfortunately, the two articles I have found pertinent to this approach are not purely information-theoretic approaches; though they invoke this paradigm in some way, in the details of their methodologies they fundamentally reduce to the paradigm of query structural analysis (the second one catastrophically so), which somewhat defeats the purpose of pursuing the information-theoretic route. I review these articles here.

Jang & Choi (2014) [154] use estimates of query result size generated from symbolic execution to dynamically prevent injections through Java applications. Symbolic execution involves mapping query elements to symbolic counterparts and representations of the operations that the query will perform, and then knowing how to interpret these symbols to understand what the result of the operations will be. [154] Estimating query result size is accomplished by gauging what fraction of rows from queried tables will satisfy given boolean conditions within queries (as parts of WHERE clauses), and aggregating these estimates to produce an estimate of how many rows will be returned by a given query. [154, 155] Incoming queries are translated into safe counterparts variable by variable. To make sure the variables are indeed safe, the detector identifies untrusted components by a process of character-level, positive taint analysis, akin to WASP [148]; untrusted elements are thus prevented from instantiating the query with SQL keywords or operators. After this, the original query and the derived safe query are compared on their symbolically estimated sizes. If the sizes are different, it means the forms of the queries are different, which implies that the original query was attempting to alter the command execution sequence—it was an attack. [154]

While the authors' method involves semantic analysis, they identify it as more precise than parse-tree analysis. Their reasoning is that malicious queries can match the structure of valid queries. To evaluate their method, they wrap their tools into an Injection Detector Library (IDL), which concomitantly symbolically reduces and sanitizes incoming queries and estimates their size, and test it on a baseline data set with ~16,000 attack queries, ~3,800 benign queries, and 30+ attack patterns represented (tautology, union, etc.) on seven web services (5 commercial, two student-made) comprising ~56,000 lines of code ^The authors mention that this is AMNESIA test bed, however, both the test beds from the original AMNESIA paper [131] and the expanded version listed online [156] have different query set sizes (7282 attacks/3500 benign queries in the 2005 paper, 41449 attacks/5602 benign from the online expanded version.). Thus it is uncertain exactly which queries were selected for this paper, or what the selection criteria were.^. Of the ~5800 (36%) of the attacks that succeeded without the IDL active, none succeeded with it active, while none of the benign queries were marked as attacks. In addition, from CWE/CVE data the authors collected 2,709 vulnerable parameters embedded in scripts across 7 real-world applications and submitted queries against them; none were missed, and no false positives were identified. The authors report that their new method outperforms several other prominent approaches from the literature on each of the apps in the test bed and collectively on the CVE-drawn vulnerabilities.[154]

Though the new method seems quite effective, by nature it cannot handle certain coding structures—e.g. complex queries, queries in loops, and "multiple table queries," such as JOIN statements—rendering it ill-equipped to handle a large portion of the SQL that exists in production. The authors' solution is to require developer intervention/code revision in these cases, but this defeats the purpose of having an automated method to detect injections. The fundamental issue is that their size estimation method requires semantic reduction of the query, and this inability to separate analytical concerns introduces complexity that ultimately reduces the efficacy of their method. This manifests in the method's success rates: for the AMNESIA test bed, developer intervention was required in nearly 100 cases. For the CWE/CVE dataset, 6 interventions were required—it is uncertain what proportion of the overall attack body this represents. This problem also surfaces computationally: summing the average overheads of the different components of the active IDL yields slightly over 100% overhead. [154]

A more subtle point related to the IDL's Java-centric focus is that it hinges on the fact that Java does not permit type-unsafe operations, which other languages (like C) will allow; this again limits the IDL ability to work across different environments.

No doubt, the IDL method proposed has potential to be a useful tool. However, its overhead may be too high, and the inherent limitations it has in handling complex queries ensures that in its presented form it cannot be a full defense against SQLi.

Shahriar & Zulkernine (2012) [153] use a process as follows to detect SQLi:

- Measure the entropy of each SQL query before deployment
- In runtime, for a given query, measure the entropy of the query that results from user input
- If baseline queries and input-derived queries have different entropy levels, the query is deemed an attack

To calculate the entropy of a query, the following procedure is used:

- Let the query be represented as a series of tokens $\{t_1, t_2, \ldots, t_n\}$. The authors do not explain how they tokenize queries, so I will assume that it uses whitespace delimitation to identify distinct characters/words and that it identifies special symbols as their own tokens.
- Let $P(t_i)$ be the probability of finding one of the tokens t_i in the query.
- The query's entropy is $-\sum_{i=1}^{n} P(t_i) \log P(t_i)$; i.e., multiply $P(t_i)$ by $\log P(t_i)$ for each token, and sum the results.

In the paper, the authors provide this particular example:

```
“SELECT id, level FROM tlogin
WHERE login = ‘” + sLogin + “’
AND password = ‘” + sPassword + “’;”;
```

From which a normal query might be:

```
SELECT id, level FROM tlogin
WHERE login = ‘login’
AND password = ‘pass’;
```

Commas are not counted as tokens, but other symbols (such as the = sign) are counted. There are 12 unique tokens: { `SELECT`, `WHERE`, `FROM`, `AND`, `sLogin=‘login’`, `sPassword=‘pass’`, `id`, `level`, `login`, `password`, `tlogin`, = }; the last one, the = sign, appears twice, so there are 13 tokens in total. The probability of encountering the = sign is thus 2/13, and the probability of encountering anything else is 1/13. We assume that the variables are replaced with benign input, such as ‘login’ and ‘pass’.

An attack query can be formed by setting <`sLogin`> to <`’ OR 1 = 1;--`>:

```
SELECT id, level FROM tlogin
WHERE login = ‘’ OR 1 = 1;
--AND password = ‘pass’;
```

Notice, there are now 10 distinct tokens: { `SELECT`, `WHERE`, `FROM`, `OR`, `id`, `level`, `login`, `tlogin`, `1`, = }, and now two tokens appear twice (‘1’ and ‘=’), resulting in 12 tokens overall. Since the probability distribution is different than in the base query, so the resultant entropy for this statement will be different. Note that the empty string ‘’ in the component <`login = ‘’`> is not counted as a token.

The authors test their method on three open-source PHP programs downloaded from sourceforge.net, each of which ranked in the top 5 of downloaded programs from that site. The numbers of attacks for these applications were respectively 128, 24, and 316. The authors generated the attack queries by the following procedure: [153]

- Within a given script, locate distinct occurrences of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` tokens within queries
- For each of these locations, create four attack queries by inserting four different malicious inputs into the given location: two union attacks, two tautology attacks

All attacks were caught. From the author's research into the erst-existing Open Sourced Vulnerability Database (OSVDB), their method was able to uncover 10 previously unknown vulnerabilities. [153]

The method seems interesting, and its appeal lies in it relative simplicity. However, the simplicity of it and of the author's evaluation of it is also its downfall. The authors do not test for false positives, apparently taking it for granted that false positives are not possible via this method (i.e., the entropy of a benign query cannot diverge from that of the respective unparameterized query). From what I can tell, it seems that as long as user input is not used to construct SQL code, but only to provide objects to pre-defined queries, then false positives should be impossible—providing that these objects will be treated as single tokens. However, if user input is used to dynamically construct SQL code—which is not a good idea in the first place—then benign queries can yield divergent entropies. It would have been preferable for this logic to be spelled out in the paper explicitly, but this may be a minor point.

A more significant concern is that there were relatively few applications and attacks tested, and the types of attacks submitted were only tautologies and union attacks. In terms of evaluation scope, we do not have strong guarantees of the method's robustness.

In fact, it is easy to find that the method is not robust, and that it is easy to bypass. I have come up with two variants on the example query provided in the article that would successfully mount attacks and go undetected by the method. Recall that this query was:

```
SELECT id, level FROM tlogin
WHERE login = < sLogin >
AND password = < sPassword >;
```

Recall that there are 12 unique tokens, one of which (the = sign) appears twice, so 13 tokens in total. Say that we set <`sLogin`> to <`nonce' OR 1 = 1.0;--`>; then the produced query is

```
SELECT id, level FROM tlogin
WHERE login = 'nonce' OR 1 = 1.0;
--AND password = 'pass';
```

The unique tokens are {`SELECT`, `WHERE`, `FROM`, `OR`, `sLogin='nonce'`, `id`, `level`, `login`, `tlogin`, `1`, `1.0`, `=`}; there are 12 unique tokens, one of which (the = sign) appears twice; thus the probability distribution is exactly the same as in the original query—this statement is an attack that would go undetected. Now one could notice that we rely on type-casting to evaluate <`1=1.0`> to true; one could argue that the method could easily be extended to check for duplicate numeric values in different types (*int* and *float* in this case). My counter would be twofold: (1) this check could lead to false positives in cases where duplicate numeric values arose as part of a valid query, though it depends on the details of how queries are tokenized, which the authors have not provided; and more importantly (2) this check is insufficient. Consider this other attack query that results when we set <`sLogin`> to <`nonce' OR id > 0;--`>:

```
SELECT id, level FROM tlogin
```

```
WHERE login = ‘nonce’ OR id > 0;
--AND password = ‘pass’;
```

The unique tokens are {SELECT, WHERE, FROM, OR, sLogin='nonce', id, level, login, tlogin, 0, >, =}; there are 12 unique tokens, one of which (the id token) appears twice; thus we again have the same probability distribution as in the base query, and the attack query does not rely on a mechanism with so simple a mitigation as type-casting.

Notice that if this attack is truly to be a tautology, all the values of the id column should be greater than 0. If this is not the case, a tautology attack is still easy to perform: an attacker would just have to submit two queries: one with the condition (id > 0), and another with the condition (id < 1).

It is disappointing that the two simple tautology (or quasi-tautology) examples I devised are successful in subverting this method, especially given that the authors report having explicitly submitted tautology attacks against their method. But why do they work? The method’s underlying weakness is that it does not keep track of the distributions of particular tokens; it counts all tokens equally, so if one can come up with an attack query that has (1) the same number of distinct tokens as the base query and (2) the same count distribution among these tokens as the base query, then one can successfully sneak an attack by this method undetected. The tokens may be different if they satisfy these two properties, which means that different queries will be rendered identical in the view of the method presented—this is the fatal flaw and the attack vector.

Additionally, although entropies are measured, under the current implementation they may as well not be. Since entropy is only a product of the number of distinct tokens and the distribution of counts among these tokens, comparing two entropies for equivalence is identical to comparing the token distribution properties for equivalence. Thus, while this method claims to be information-theoretic, the connection is weak: the measurement of entropies bears no weight on the method’s results. We are instead left with a quasi-information-theoretic, quasi-structural approach; this ambiguity allows the aforementioned attack vector.

In summary, although this method can potentially be useful (it was able to detect previously unknown vulnerabilities), clear improvements are in order before it can be a sufficiently strong candidate for preventing injections. Whether these improvements would be simple or require an overhaul of the methodology is unknown.

Anomaly detection

Anomaly detection is another class of methods for automatically detecting and preventing SQLi attacks. In anomaly detection, the aim is to learn the features of normal queries and use this to filter new queries. As with a number of other dynamic detection methods, anomaly detectors take form in intrusion detection systems (IDS) that are situated so as to protect some component of the target web service. For SQLi, this system dwells between the web application and database. Benign queries may pass through this wall from application to database for processing; malicious ones ought to be rejected.

A potential advantage of this approach is that access to source code is unnecessary. Some anomaly detectors are also simple in that they can be trained externally and then integrated as an IDS into a server to be protected; others are integrated into the server as part of the learning process, after which they operate in a detection mode.

Anomaly detection is indirectly still a problem of comparing new query structure to expected query structure. The difference is that the process of mapping query structure is abstracted away and at least partially automated by machine-learning algorithms. As in machine learning at broad, this involves framing the problem in terms of probabilities: we get an estimate of the probability that a query belongs to a given class (benign or malicious), and on this basis decide which class classifies it. To arrive at this, we must first convert queries into collections of quantitative features that permit statistical modeling, and then apply a particular algorithm to distinguish queries in this feature space.[157] The performance of the algorithm may then be measured in terms of classic statistics such as accuracy, precision, recall, and F1-score.

Abstracting the process into a machine-learning problem removes certain complexities while adding others. To some extent, some structural parsing is needed to extract meaningful features from queries; however, deciding how to evaluate these structural decompositions in order to test for malice is left to the machine-learning algorithm. The primary complexity added by machine learning is machine learning itself: it is often an opaque method of solving problems, in that we can see that it attains success but cannot explain why exactly it works in some cases and not in others. All we know is that if the algorithm is fed an unrepresentative data set in its learning process, its subsequent predictions on new data will be skewed; this crucial dependence on the quality of algorithm inputs (which is not particular to merely this application) is cited as a caution against such methods in some reviews.[10]

Nonetheless, such methods hold clear promise, and for both their potential and uniqueness, I find them worth reviewing here. There is great variety to the machine-learning methods, input data type, and feature extraction methods used to construct anomaly alert systems. Here I will present a few that I found which were each unique and potentially useful. Each one focuses on the prevention of SQLi specifically. Although a detailed review of further articles would provide useful insights—and though my personal interest in machine learning might incline me to do so—explaining these methods satisfactorily requires a fair degree of detail, so I will limit the discussion to three articles.

Valeur et al. (2005) [157] develop an application-level IDS based on statistical modeling. It acts in three stages: training, threshold learning, and detection. Its components act as follows.

The parser generates a series of tokens from a query, distinguishing between constants and non-constants. It maintains a data-type attribute for tokens that represent the names of columns; there are inbuilt types, and users can also supply their own. The parser infers the type of constants from relations to other objects.

Feature extraction begins by creating a “skeleton query:” store constant tokens separately from the query, and in their place leave special placeholder tokens. The system tracks which script the query originates from. Each (skeleton-query, script) combination has an associated profile—a set of models that will correspond to each of the features selected. Once features are selected, statistical models (value distributions) are built upon them to determine the probabilistically expected distribution. The treatment of the query depends on the mode of operation:

- In training, models are continuously updated for each new query. A query must be benign to be processed during training.
- In threshold learning, each submitted query has an anomaly score calculated, and the maximum anomaly score is tracked for each model.
- In detection mode, if a query’s anomaly score is too high with respect to its respective model’s threshold, it is deemed anomalous.

The particular models used identify what the authors believe to be key attributes of data; they are taken from earlier work [158]. String length, character distributions, common filename suffixes/prefixes, and grammatical structure are modeled for strings. For other data types (including integers), a generic model is built to test whether the particular value is part of a commonly selected group of values of the same type, deviation from which suggests anomaly.

The authors train the system on 44,000+ queries and learn anomaly thresholds from ~14,000 queries. They evaluate their model on four different attack types directed against the (famously vulnerable) PHP-Nuke application. Some of the statistics presented in the results section are confusing—but from the authors’ comments and an external review [10], the method appears to be successful at detecting the attacks that the authors provided. Based on comments in another external review [135], the overhead is sufficiently small that it should not be noticed in typical usage.[157] The authors note that far more testing in production-like environments must be conducted to evaluate the model’s effectiveness.

One drawback encountered is that the system is over-sensitive to benign query attributes: in particular, several dozen queries were marked as anomalous because they were made in a different month then the single month when training was conducted. The authors mentioned that specifying a custom date data-type resolved this, and in fact state that end users would likely have to provide custom data types on every system implementing this method in order to avoid unneeded false alarms. Although the authors mention that the interface makes specifying new data types relatively simple, this is a downside to be reckoned with; given the great variety of applications and custom data types out there, how should users know when special types will unsuspectingly create false alarms? Would they have to hard-code an exception for every special type? Again, more empirical testing is needed to know the scope of this issue in deployment.

In summary, this method is an interesting proof of concept, which must be tested at scale.

Uwagbole et al. (2017) [9] present some enhancements to prior methods by invoking cloud technology and using advanced and practical machine-learning techniques. They use an extensive corpus of expected input strings, which they generate via a method detailed in another article of theirs. [129] Each input comes with a 0 or 1 label to annotate whether the input is malicious or not. The authors claim that unlike other methods, which often generate different strings that do not represent fundamentally different patterns, their method is designed to enumerate the various possible patterns to augment the potency of their machine-learning model.[9]

A special hashing procedure transforms the 725,000+ generated input strings into a binary matrix (filled only with 0's and 1's) with nearly 33,000 columns. These columns are the features for the machine-learning algorithm; the authors isolate the most significant 5,000 features for training. This feature matrix is submitted to a Support Vector Machine (SVM) for training and testing. When used for classification, SVM are fundamentally binary classifiers adept at dealing with high-dimensional datasets.[135] The authors employ this kernel method in their approach.

As this is a computationally intensive problem, scalability is a key concern; however, in the current approach training is done entirely in the cloud via Microsoft Azure, a marked advantage for reducing runtime and improving scalability.

Testing the system on a custom-built .NET application, the authors find that the model performs rather well on standard statistical measures (accuracy=.986, precision=.974, recall=.997, F1-score=.985). No explicit measure of overhead is provided, but the authors assure that this method scales well and brings SQLi countermeasures into the realm of "big data."[9]

To prove the effectiveness of this method in real environments, the authors should provide a bit more detail about certain aspects of their IDS. For example, whenever this system is deployed in a new environment, an administrator must "feed the data engineering or text pre-processing module with a new rule that matches the patterns present in the new data set;" [9] the authors do not describe how this is done, nor whether it is a simple task or one requiring special expertise. In addition, the authors do not describe the custom-made application they used to test their system—will this system fare as well in real environments? Perhaps so—we would hope so—but future studies must tell.

Kar et al. (2016) [135] also use SVM to probe queries for injections. However, their approach for selecting features is arguably much more intuitive and rooted in natural-language processing (NLP) techniques that have found widespread applications in various domains. The core of their method is the generation of token graphs: sequences of SQL tokens with some measure of relatedness between each token in a given query. In the general NLP sphere, a token is any unique, self-contained element of a text sequence. The definitions of tokens vary—but typically individual words or characters, spacedelimited text sequences, individual or joined numeric/symbolic characters, and other atomic textual elements comprise tokens.

To form these graphs, the first step is to replace all the components of inbound queries with a predefined set of tokens (words, in this case). This is accomplished by specifying a full mapping of every unique SQL query element to the encoded token. Foregoing full specification of their mapping procedure, here are some examples:

- Hexademical values map to *HEX*
- Integer values map to *INT*
- IP addresses map to *IPADDR*
- SQL Objects, such as databases, tables, and table columns, map to tokens of the pattern *SYS** or *USR**, depending on whether they were defined by the system or by users
- && maps to *AND*
- | | maps to *OR*

Some tokens—empty comments, empty parentheses, and backquotes—are removed, and whitespace is condensed so that only single spaces separate all elements. Besides eliminating unnecessary structural clutter, this helps to defend against attacks that attempt to avert detection by interspersing special characters amidst SQL keywords, as discussed earlier in this paper.[135] Note that overall this process reduces the rather large space of all possible words and syntactical elements that may comprise queries to a rather small set (size=686). This means that many query elements will overlap and map to the same end tokens, thus yielding a set of common baseline queries; this is useful for identifying underlying patterns that a machine-learning algorithm can employ.

One potential downside is that every object in the database must be known to the tokenizer. The authors create a separate automated process to retrieve this information in XML form. This in itself is not an issue; the problem is that databases are not static—their structure changes over time—so the corresponding XML catalog must be updated regularly. It is not clear what issues may arise when a query requests an object in the database if the XML file does not have a record of that object. The authors suggest periodically updating the XML schema every so often (i.e. by explicitly programming a chronic loop), but it would be better if an automatic update could be performed every time the state of the database changed. It is uncertain whether this article's implementation provided such a mechanism, or whether one could be provided in the future.

When queries are fully tokenized, a graph must be constructed and weights applied. Tokens are the vertices, and edges are formed between tokens based on proximity. Iterating over the sequence with a sliding window tracks how frequently certain tokens appear in proximity to one another, which derives the weights for the edges between those tokens. After all weights are calculated, the token graph and the sums of the weights of the edges connected to each vertex (vertex *degrees*) are submitted to the SVM.

In practice, more than one SVM can be used. If two SVMs are used, the combined system can decide to outright reject a query as malicious or accept it as benign if both SVMs agree; if they disagree, the query is sent to an administrator for review. If three SVMs are used, the decision of the joined system is computed by majority vote. Using varying numbers of SVM (up to three), different graph types (undirected/directed), and different weighting schemes, the authors come up with 10 different IDS systems (6 single SVM + 2 double SVM + 2 triple SVM). A set of ~9,500 unique token sequences, produced from pruning and preprocessing a collection of several million queries obtained from logs of real web applications, is used to train each system.

The results are very strong. Collectively, the different metrics (accuracy, recall, precision, F1-score) are almost always above 99%, and rather frequently above 99.5%; the false positive rate is under 1% in 8 runs, and under .5% in 5 runs. Analyses of overhead suggest that users would typically not notice any difference in query time with the IDS in place compared to without; testing with some web applications suggests an overhead of 2.5% of average web server response time—nonzero but quite low.

In addition to these results, the authors provide a number of benefits of their method that are not always found in combination elsewhere in the literature. In particular: [135]

- No modification of or access to source code is required.
- It is simple to integrate into existing servers.
- The method is not constrained by programming language and is easily portable to different SQL platforms.
- It can protect multiple websites in an application simultaneously.
- It detects a wide range of well-known attack types, including those involving stored procedures, which are notoriously difficult to counteract [10].

The authors note that although their method correctly identified all tautologies in their data sets, in theory there is no combinatorial limit to the ways a tautology may be constructed, so they believe their method only partially handles tautologies. This is a bit disappointing, given that this very potent attack is among the simplest of all, and that other methods can fully handle this attack type. In practice it might induce no extra disadvantage to implement a simple but robust tautology-checking algorithm alongside this in the IDS, but the authors do not evaluate this.

Kar et al.'s method shows substantial promise. Two significant but feasible improvements would be the aforementioned automatically triggered updates of XML database catalogs and handling of tautologies.

--

WRAPPING UP

In this section I offer my own thoughts and takeaways from the research I have conducted thus far.

For all of the automated measures and their respective paradigms surveyed in the previous section, the methods ultimately reduce to these problems:

- Determining what is trustable and what is not
- Determining when a query has valid structure and when not

It is not by chance that these two themes are universal. SQLi is by definition the divergence of an SQL command from its intended execution; SQLi detection is ultimately evaluation of the divergence of command execution, whether directly by structural decomposition, or indirectly by assuming that data labeled as untrusted will diverge from intended execution. Without a more thorough analysis, it is uncertain whether information-theoretic approaches could work around this by considering models of data flow independent of query analysis; however, current information-theoretic approaches do not attain such independence.

The question is then, what is the most effective and efficient way to do this—to ensure that a query runs in its intended structure, and that untrustworthy data cannot affect the execution path? You may notice that I have already answered this question earlier in the paper:

> *Taken altogether, the tailored use of parameterized queries and input verification, along with proper permission configuration, ensures that data is interpreted solely as data, and that the intended execution path cannot be altered; this is the most effective and efficient way to prevent injections.*

Ironically, the literature on SQLi aversion as a whole, while impressive and promising in many ways, seems in as many ways superfluous. The solutions presented earlier in this paper are simpler and more efficient, occur at the source, and have nearly complete chance of success when used correctly and ubiquitously; that they are not more utilized is an embarrassing state of affairs, which begs the question: why is it so?

Thoughts on the business/development paradigm

It is true that over time, more advanced SQLi attacks have been developed; however, by and large SQLi attempts are not complex in nature. An analysis of CVE-reported SQLi vulnerabilities from 2005-2009 suggested that the large majority of attacks do not attempt to use obfuscation methods such as embedded comments or alternate encoding of special quotes. [159] Other reports suggest that most SQLi attacks have basic aims and techniques [13, 59, 160]

- Blind injection, e.g. double conditionals and timing attacks

- Union attacks
- Using reserved keywords or functions to retrieve database information
- Probing specific database objects that are known to store user credentials

Each of these attacks is relatively simple to stop—but how? Developers must have at least some security knowledge and awareness, and they must incorporate this knowledge into the development process. These are ultimately business problems: [161]

- Do developers have relevant security experience? Is this a consideration in this hiring process? Are there ongoing training or learning outlets provided?
- Is security a priority in the design and initial implementation stages?
- Since mistakes and oversights are inevitable, is security auditing a regular part of the development process?

Transmission of good cybersecurity practice into code is deficient across organizations of all sizes.[162] While smaller projects may inherently face a greater challenge in this regard, the fact that larger organizations are also affected suggests that lack of resources is not solely responsible for the state of affairs. The more likely problem is that businesses are not devoting sufficient consideration to secure development, which could have a number of explanations; here are some possibilities:

- *Genuine lack of awareness*: organizations are not aware of the dangers posed by cyber attacks. Given the amount of press devoted to breaches and cybersecurity in recent years, this seems unlikely.
- *Apathy*: organizations are aware of cybersecurity threats but do not believe they will be affected. Alternatively, developers may be convinced (staunchly so) that they do not need to apply security measures, whether or not they fully understand the issues at hand. One does not need to look too hard to see how unproductive arguments can emerge out of this; e.g. [117].
- *Competing pressures*: organizations and developers are aware of cybersecurity threats and are not apathetic about them, but core business demands—producing functional, timely products—outweighs incentive for secure development.[161]
- *Lack of knowledge about how to prepare*: organizations might be devoted to implementing cyber countermeasures, but might be lost on how to do so.
- *Perceived lack of resources*: even if organizations know of some ways to guard against cyber attacks, they may believe that doing so will demand too much time or money, or will simply be too much of a disruption to their established development cycles, to implement.
- *Inefficiency*: even if all the above do not hold, and organizations are implementing measures to ensure secure development (or remediation of codes that were not developed securely), they simply may be doing so inefficiently, and leaving known vulnerabilities around for some time before fixing them.[173]

This reduces to three fundamental points of interest: awareness (what needs to be done), incentives (why it will be done), and practice (how it will be done).

In terms of awareness, organizations could contract with external experts/consultants, or organizations that offer consulting services, to provide developers with security training.[162] If this is perceived as too significant an expense, there are numerous low-cost or free resources available that will suffice. Search engines such as the general Google search and YouTube search are efficient and effective ways to find such resources, and it would not take much time for an organization to string together a set of resources from the web. For example, it took me little time to go to YouTube, search "secure programming series", and find a short playlist with common vulnerability prevention measures for PHP [163], one video in which discussed SQLi and demonstrated a use case of prepared queries [164].

Now granted, one cannot simply look up a quick video or link and expect to arrive at a comprehensive and informed security solution right away. However, upon finding such preliminary resources, which present the issue but do not explore it in depth, one could read further into other resources. Business heads might be surprised to discover how valuable a corpus they could build given a few weeks or months to develop training materials. If they were unable or unwilling to develop such materials themselves, one option is requiring developers to undergo external training, not necessarily by consulting for high fees with security organizations or consultants, but by researching externally prepared resources that have already been developed into coherent modules. This presupposes that such modules exist; I would be surprised if they did not exist already, especially given the prevalence of online learning platforms (according to a report by Veracode, online learning tends to yield measurable improvements for organizations in terms of the rate at which software vulnerabilities are corrected [115]). Of course, even if these materials exist, they may require improvement and continual updating as new security threats and defensive technologies emerge.

The burden of spreading cybersecurity awareness lies not only on organizations and developers, but also on everybody who serves as a source of knowledge to these entities. Wherever there is instruction or documentation on code, software, or anything thereof, security must be included, and not merely in passing. For example, the MySQL 8.0 documentation mentions in passing that stored procedures allow arbitrary execution of strings via the `EXECUTE` keyword, but does not mention the security implications of this,[123] despite maintaining stored procedures as a component of software where "security is paramount" [118]. Emphasis must be placed on the fact that developers cannot create secure code blindly: they have to understand how particular tools can be used insecurely and securely. This also applies to programming textbooks and other such condensed instructive resources: teaching how to accomplish certain actions via code is not enough when those actions could be done in a highly insecure way; a secure alternative should be presented, and the reasons for preferring such alternatives in certain circumstances should be highlighted. It is likely that many such resources today mention security at most in passing,[137] with the result that developers of web applications are trained in how to make productive but vulnerable applications.[157]

Other generic measures, such as regular cybersecurity conferences, perhaps tailored to specific industries, and development of security-aware curricula for formal schooling institutions, would also be useful—not only for organizations, but also for the public at large.

Organizations themselves must also be aware, not only of cybersecurity in general, but also acutely of the states of their own systems. Testing systems for security on a regular basis and minding the potential hazards of third-party software is a good start, especially given that many companies do not do this [161, 65], but this is insufficient. We saw that Heartland believed itself to be secure prior to its major breach—internal and external audits failed to prevent the breach, and PCI-DSS compliance was evidently an insufficient mark of the company's cybersecurity integrity. Perhaps the generic standards need to be clearer and stricter on what constitutes a reasonable level of security.

In terms of incentives, providing organizations with strong incentives to develop secure coding practices would lead them to in turn incentivize developers to code securely, independent of whether or not those developers were already motivated to do so (which is questionable). A large part of incentivizing developers, as noted by Wenham (2012), is not only requiring security training, but also instilling why security is so important in the development process.[162]

Unfortunately, as seen in practice, this incentive is not always enough for organizations. What sorts of incentives would be most effective here? This is an open question. Incentives could be positive or negative. Positive incentives come in the form of some reward, direct or indirect, for developing secure software, while negative incentives penalize organizations for sustaining insecure code.

For both kinds of incentive, there would have to be a standard way of measuring and ranking the severity of security vulnerabilities. This would require a taxonomy of different kinds of vulnerabilities (e.g. SQLi, XSS, etc.), the different severity levels associated with those vulnerabilities, and the different dimensions of vulnerability that exist (e.g., threats to CIA – confidentiality, integrity, availability –, effort required to exploit, and possibly other dimensions). Such taxonomies already exist (e.g. CVSS [165]), but whether they are at a status to be used for generalized rankings and high-stakes applications is uncertain. There would be a lot of work to do in creating such a set of standards, and convincing industry and legislators of the usefulness of such a measure might be a non-trivial endeavor; yet it should be put on the table for broad consideration, in the event that it turns out to be useful.

For financial reasons, organizations already have incentive to safeguard their own data and intellectual property, so the strongest incentives should be centered on protection of user data. Breaches are of especial concern: current penalties involving breaches are too imbalanced, leaning on stronger consequences for smaller organizations and not for larger organizations. In my opinion, the penalty for a breach should be more nuanced. In addition to auditing requirements following a breach, there should be detail-oriented fines. If the same baseline fine is applied to every organization for a breach, it will present a minor concern to organizations whose revenues far outweigh the fees. Raising the baseline fee substantially across the board would practically guarantee that any small organization suffering a breach became shutdown immediately; this is too forceful an approach. One solution is to fine organizations some predefined portion of their annual revenue, which put more pressure on larger corporations while also preventing damages from being overly destructive to smaller businesses. The details of the breach should also count: the more data breached, and the more sensitive this data is, the higher the fine should be. This would require formalizing the notion of sensitivity and the rules for translating quantity of data compromised to a penalization metric.

In terms of practice, the core concern is how to implement cybersecurity practices into existing development cycles. In at least some cases, it is inevitable that shifting to a secure development cycle will be disruptive to current practices, but that is a reality that businesses must face. Less disruptive changes, while preferable from a business standpoint, may be less effective. For example, one approach would be to have bifurcated software development: in each iteration of code development that yields a new piece of working software, either the same team that developed that code or another team audits it for security, patching faults where located. Although this could potentially be effective, it has drawbacks. It is inefficient: although in some cases patching security faults might be as simple as replacing an unsafe query with a parameterized one, in general some flaws might require drastic reworking to solve. In addition, in more complex software projects, relatively simple faults might be overlooked if too many accumulate, and even one small fault missed (such as a query that was mistakenly not parameterized) can render other fixes ineffective[112].

It would be better to design for security at the beginning, so that code was written securely the first time around, without the need for spending resources on fixing it later. Second, this approach makes it easy for security to become an afterthought, especially when deadlines are tight and there is pressure to produce working software quickly. Designing for security from the start would ensure that security were a concern upfront. They key is convincing organizations that they will, in the longer run, save time and effort by preventing security faults at the start.

A generic paradigm for preventing SQLi specifically is changing the APIs used to configure web applications, so that sensitive functions that query and execute commands within databases are not made public and cannot be tampered with through standard injection.[10] Some success has been observed thus far in reducing injection vulnerabilities in this way, but this is interpreted as more of a serendipitous consequence than a deliberate guard against attacks.[61] Insofar as this is the case, it is left to chance whether this paradigm becomes a broad mechanism for reducing this kind of vulnerability. Halfond et al. (2006) also observe that, notwithstanding the practical challenges of this approach, this approach is primarily intended for new software projects, not systems that already exist.[10]

The research paradigm

Research into new protection mechanisms is vital if for no other reason than research itself; we never know what new research will uncover, or what key discoveries lie around the corner. Although SQLi vulnerabilities admit relatively simple solutions in the overwhelming majority (if not totality) of cases, the practical reality is that not all organizations will be willing or able to employ those solutions now or in the near future, and thus active detection systems may be warranted. That being said, in exploring the literature for this paper, there were some issues that I found.

The first and foremost issue is that, despite the significant amount of research that has been done on SQLi since the early 2000s, the problem has not gone away, despite the fact that a number of solutions (most of the ones I have reviewed) claimed to be highly effective in suppressing injection. Yet many of these articles, including highly cited ones, were published years before some of the worst cases of SQLi-caused breaches occurred. Clearly there is a disconnect somewhere—where? Are the methods really so effective, but have not been adopted? Were the authors' evaluations of their methods too limited or optimistic?

The answer probably lies somewhere in a combination of these. One of the problems I have isolated in the literature is simply that there is no clear answer—there is no certain explanation for what has happened to these methods over the years, and whether they have become incorporated into existing systems or have disappeared into obscurity. Articles are published with limited to no follow-up, and subsequently only ever appear in retrospective reviews. Take as an example WASP by Halfond et al. (2006). Halfond, who was involved with two of the most highly cited counter-SQLi projects I have found (WASP and AMNESIA), has not done anything with SQLi or web security in quite a while [166, 167]; the second author, Orso, who was also involved with AMNESIA, hasn't touched SQL in publications since the second WASP paper in 2008 [168] Manolios, the third author on the WASP paper similarly has not done anything else related since that paper. [169] This is by no means a criticism of these authors; rather it is meant to represent the common state of affairs, that projects are presented and then appear to be dropped after a relatively short time frame without ongoing development or integration into other services, even when the authors express interest in continuing the work.

In general, this is a problem that I have seen acknowledged in only one article I have thus far encountered [140]: research continuously yields new methods but does not attempt to improve upon old methods or integrate them into current solutions. The literature does not yield consensus on what is effective and what is not; authors highlight the strengths of their own methods and the shortcomings of others, and the reader is left wondering what really works and what does not.

Much of the problem lies in the fact that there is no standard basis for evaluating the various methods in the literature: most articles have to generate their own datasets for testing because there is no corresponding, universally accepted standard in the literature.[9] I have not identified reports that attempt to compare the methods in real-world environments, detailing performance and implementation issues, though such an evaluation would be incredibly useful for anybody trying to make sense of the literature. As one review notes about the tainting literature, "While the papers mentioned above provide useful guidance… consistent and comprehensive reports describing performance of runtime tainting are hard to come by." [139] Researchers should strive to make their work available, implementable, reviewable, and augmentable; the literature is very imbalanced in this regard.

The hint I have found thus far of the literature's methods making headway into practice is a 2014 survey conducted by the Ponemon Institute.[65] A number of findings emerged from the study, perhaps most interesting among them that from the 16,500+ surveys sent out to individuals working in IT/IT security at various levels, only about 700 responded—research on the cybersecurity community cannot proceed if the community is not engaged or interested in taking part. ~92% of the respondents were in supervisory, managerial, or executive positions, suggesting that these responses offer insight into how business heads perceive security.[65] A majority of respondents/respective organizations:

- Had an SQLi attack within the year prior that evaded in place defense mechanisms (65%);
- If attacked, took a month or more to discover the attack (60%);
- Were unfamiliar with or had no knowledge of tactics used to evade such mechanisms (61%);
- Agree that understanding the underlying causes of attacks "strengthens [their] organization's readiness to mitigate future attacks;" (64%)
- Report the threat faced by SQLi as severe (7 or higher on a 1-10 scale) (72%)
- Do not audit third-party software for SQLi vulnerabilities (52%);
- Were familiar or very familiar with "behavioral analysis" IDS systems (90%), saw these favorably or very favorably (88%), and were planning to or already had implemented such a system (60%) on the client side or somewhere on server side (either on network perimeter or in front of a database)

A minority of respondents:

- Agree that "technologies are in place to quickly detect a SQL injection attack" (34%)
- Agree that IT personnel had "the skills, knowledge and expertise to quickly detect a SQL injection attack" (31%)
- "Scan for active databases" on a weekly or more frequent basis (38%)

Whether or not these findings generalize to the broader population is questionable, given the small response rate. Still, this sample exhibits some concerning trends. Respondents by and large appreciate the threat faced by SQLi and acknowledge that their organizations have been susceptible to these attacks, yet they did not take basic security measures such as regularly checking on the state of their tech stack and databases. And while nearly 2/3 believe that understanding the attack at its root would help to counteract it, a substantial minority (36%) do not. Although many value understanding SQLi at the source, nearly as many do not understand the mechanisms that attackers take to skirt defense mechanisms, and they do not believe their IT departments have such competency either. Some of this might be a product of the fact that respondents were almost exclusively in supervisory positions or above; it would be helpful to see how developers respond to this question set.

One of the most interesting findings here is that the openness to implementing an IDS was overall significantly higher than the priority placed on understanding the nature of SQLi, even though both were majority stances. It seems a number of organizations would rather farm off the technical details of the attack to somebody else, who could develop an automated system to handle the problem for them—a conundrum, if we are to solve the problems at the source.

New development paradigm

In general, the countermeasures identified in the literature are founded on the premise that developers will invariably leave software full of security holes, and that the only solution is one that removes the need for their security experience from the equation—whether by automated tools or by alternate programming paradigms (e.g. [137, 157]). The aforementioned conundrum in the Ponemon survey suggests that from a practical perspective, a substantial number of organizations see the matter similarly. As a result, many portray the development process as unsalvageably plagued by security failures, proposing that the mitigation must be external to the development process. An article by Viega et al. (2001) [170] states this quite boldly:

> "*The amount of expert knowledge necessary to secure source code should be minimized. It should not be easy for a developer to accidentally introduce security problems into a program just because the language and the concepts of secure programming haven't been mastered. Common language pitfalls should be averted, and the programmer should be protected from common classes of mistakes that are not language specific.*" [170]

Certain parts of this proposition are, at least teleologically or philosophically, agreeable or encouraging; in an ideal world, it would not be easy for security flaws to arise, and developers would be protected from. And maybe in an ideal world, people would not make mistakes, security flaws would not exist, and nobody would be looking to exploit them anyway. Nonetheless, the world is not ideal: mistakes are a part of the learning process, they are bound to happen (even though they can be costly), and they may become security faults. The question is, is it possible to fulfill the above proposal—to prevent developers from introducing security faults *without* them having experience with security concepts, and preventing them from errors beyond the ones particular to the coding language used? That is, can we write a language so as not to be vulnerable?

Let's consider how we might do this in SQL, confining our examination to the scope of interacting with the database via queries—never mind for now the other relevant security dimensions associated with SQL deployment. Make the language or programming paradigm restrict certain actions outright, like using unprepared or unsanitized queries. But how would this be enforced?

- One option is to have an inbuilt class that captures user input and requires sanitization on this input before incorporating it into SQL queries, and enforce usage of this mechanism on any query being submitted. Whether this is a reasonable design approach is debatable. Developers could override a ``` `sanitize` ``` method to specify particular sanitization actions for particular applications. If the default were to remove special SQL characters, this might be acceptable in many cases, but not all; it would still require the developers to understand how their code is applied. Developers could also simply pull a non-implement: if the default does nothing, they could simply refrain from overriding it, or override it with a method that does nothing, defeating the purpose.

- Another option is to have a database interface that does not permit unparameterized queries. As it is arguably unreasonable to enforce parameterize queries in cases where simple sanitization would suffice, this approach is unnecessarily inefficient.
- A compromise is to balance these two solutions, and have a database interface that requires either sanitization or parameterization on any submitted queries. This is not an unreasonable solution, though there are two points to mention. First, we would have to rely on developers not to simply fall back on non-implements. Second, developers would still have to know when to apply each case (sanitization/parameterization), which requires them to know the relevant security and efficiency implications of both.

We see, even in this simple case, the most reasonable solution requires developers to have security awareness. The solutions that attempt to abstract security away impose inefficiencies or insensible conditions, and the first one has an easy workaround that defeats the purpose.

To more generally demonstrate the issue, how would we go about abstracting security away from a language in the generic case? A language can be built within a spectrum of security (the resistance of the language to faults) and functionality (the range of tasks that the language may perform, with consideration of how feasible it is in practice to do each task)—each can range from low to high, and a language may fall anywhere within this space. At the extremes, there are four combinations:

- *Minimal functionality, minimal security*: nobody would want a language in this state.
- *Maximal functionality, low security*: this might be productive from a business standpoint, but it worrisome from the security standpoint. This is the problem we are trying to solve.
- *Maximal functionality, maximal security*. A language like this would have to allow all possible functions (i.e., generic tasks) relevant to a given domain, while ensuring that none of those functions could be performed in an insecure way. To accomplish this, there are several options:
    - Abstract and enumerate all the security-related circumstances where the language would be applied—how? We would have to consider a priori every possible function that could be performed with the language—that is, the permutation of all executable commands, all objects to which these commands could be applied, and all possible or relevant combinations/chains of these commands. After this, each command combination that contained a security flaw (if it were possible to identify this a priori) could be rejected, leaving only secure ones left. This is combinatorically infeasible.
    - Develop an API over a language to achieve the same thing: this is another way to attempt the same thing, and so suffers from the same drawbacks.

    - Permit users to do anything, but embed in the language automatic checks on every action taken by the user. This is not unheard of—dynamically/weakly types languages do this to ensure that type-illegal operations are not performed. But such checks must be severely limited in scope, otherwise the language could not run efficiently. Thus, in practice this could not provide a comprehensive security solution.
    - Attempt to know the consequences of any generic command sequence before execution, so as to block insecure commands: beware the *Entscheidungsproblem*. [171]
- *Minimal functionality, maximal security*: in pursuit of security at all costs, you could restrict a language to a minimal set of secure operations; this would still require proof that any given operation or combination thus cannot result in a security flaw. Aside from the difficulty of proving the robustness of this operation subset, such an approach would be too restrictive to permit the variety of control flows and commands that web applications require, or make these too difficult to achieve in practice.

To accomplish Viega et al.'s (2001) aim, we have would to select an option with maximal security. However, I have enumerated the extremes to show the shortcomings of each—none are acceptable in practice. The solution is to compromise: allow enough functionality in a language for it to be useful in its intended contexts; but provide ways to execute commands securely, whether inherent in the language, or by requiring developers to use the lowest-level tools that the language provides to do so.

The takeaway is that Viega et al.'s proposition is reasonable only within limits: we cannot totally abstract security from a language, unless we are willing to sacrifice at least some functionality and flexibility of the language. The more we abstract away, the greater the sacrifice.

In practice, this tradeoff happens all the time, and to some extent it is justifiable. A simple example is a comparison between languages such as Python and C. Python does not allow everything that C does; e.g., in C one can index an array beyond its bounds and thus read/write data outside of the array location, but Python's native array-like objects raise an error on any attempt to do this. This makes it more difficult to cause a buffer overflow in Python (one would have to exploit an implementation detail in the underlying C interface of Python, or in a third-party library used in Python that offers a window into a bounds-insecure language such as C), whereas in C it is easy to do so unless developers ensure that arrays are not indexed out of bounds.

To play devil's advocate—if C is more vulnerable, why use it? Because C provides functionality that is useful in many circumstances that Python does not provide. But then if C can do everything Python can do and then more, why use Python? Because Python alleviates developers of having to consider many low-level details (some of which are security-related) that C necessitates, which they arguably should not have to care about if they don't require such functionality.

Python and C provide a concrete example for an abstraction that holds across any language and development paradigm; the underlying tension is that between liberty or free range on the one hand and security or restriction on the other. More free range provides for greater usability but also more room for vulnerabilities to be created; the less freedom allowed, the easier to suppress such vulnerabilities, but the less generalizable or flexible the language becomes. The argument against overly restrictive languages is that programming is meant to be abstract and widely applicable. But if developers are given appreciable freedom—which I believe necessary for productivity—they must be trusted to use it responsibly, developing and employing security knowhow if their roles directly entail security consequences.

This is a subtle point, but may require a profound change in modern industry's perspective on development and security: coding flair is not enough for developers when they are immersed in security-sensitive contexts: they must be knowledgeable about security and about the relevant consequences of their implementation choices. In my mind this is not a drastic requirement, given that this view already holds in a number of other development contexts. Programmers getting into UX are often sought to have UX experience beyond experience in whatever languages are used; developers working in data science or machine learning contexts cannot simply rely on coding knowledge to be productive, but must immerse themselves in the (broad) other domains that comprise that field. Why should security be any different, or follow a lesser standard?

I would go further to say that so long as we fail to adopt this way of thinking, we will continue to ensure that developers be deficient in security knowledge. We attempt to abstract security knowledge away from them, while demanding that they produce secure applications, and then we wonder why they fail to do so. We are setting development up for failure by imposing such contradictions on it.

Moreover, I believe it will be more expensive (at least inefficient) for organizations to maintain security in this manner: for instead of establishing a tradition of security knowledge internally, which employees can propagate to subsequent hires, organizations that do not build a strong security foundation will constantly have to outsource their security concerns and rebuild the basic foundations of good security practices. This brings up another concern I have noticed in perusing the web—a number of organizations promoting security solutions seem very eager to market their products as way to counteract SQLi (as well as other flaws), yet conveniently fail to even mention solutions such as parameterized queries, much less state how effective such simple solutions can be. Akamai deserves mention as one source that consistently mentions the significance of basic solutions such parameterized queries and input sanitization in defending against SQLi. [41 – 44, 172]

To be sure, it is not simply the case that developers in security-sensitive contexts always overlook security or have no security awareness; however, their attempts to mitigate vulnerabilities are not always insufficient. For example, one Akamai report notes that developers often took measures against certain kinds of attacks (e.g. LFI and XSS), but these measures were composed with varying levels of sophistication and were not applied across the board, allowing a number of attacks to bypass them.[41]

SQLi is a relatively simple vulnerability with relatively simple but effective fixes. Other vulnerabilities are more complex in nature and may require more intricate solutions. But every problem is tractable, especially when we re proactive to prevent those problems in the first place. *Designing* a language to be both fully secure and functional may be combinatorically infeasible; but *using* a language wisely can surely provide for secure and functional code. Experience and knowledgeability, combined with vigilance, allow developers to cross the combinatoric chasm by enabling them to not have to search over all possible execution paths: they learn how to recognize patterns and antipatterns generically, and they become mindful of the vulnerabilities that pertain to different contexts (e.g., when users providing input without safeguards in place, their input can be interpreted as code rather than data). This in turn allows them to craft new codes in such a way so as to adhere to secure patterns and avoid the antipatterns. In this regard there is little practical substitute for awareness.

There are a host of other concerns that I cannot address here, for lack of both space and awareness. But I believe the changes I have thus far recommended would go a long way.

--

References

1. Buckler, C. (2018, Sept 18). SQL vs NoSQL: The Differences. *Sitepoint*. https://www.sitepoint.com/sql-vs-nosql-differences/ (Accessed Apr 26, 2020)
2. McDonald, D. (2008). Introduction to Database Management Systems. *CIS 3730 - Database Management Systems*. http://www3.cis.gsu.edu/dmcdonald/cis3730/SQL.pdf (Accessed Apr 26, 2020)
3. 1506.04082.pdf:::Ron, A., Shulman-Peleg, A., & Bronshtein, E. (2015). No sql, no injection? examining nosql security. *arXiv preprint arXiv:1506.04082*. https://arxiv.org/pdf/1506.04082.pdf (Accessed Apr 26, 2020)
4. Georgetown University (2020). XBUS-493 SQL Fundamentals. *Georgetown University Course Search*. https://portal.scs.georgetown.edu/search/publicCourseSearchDetails.do?method=load&courseId=18412833 (Accessed Apr 26, 2020)
5. Team, dbF. (2019, Nov 28). How to generate and use CRUD stored procedures in SQL Server. *devart: HOW TO*. https://blog.devart.com/how-to-generate-and-use-crud-stored-procedures.html (Accessed Apr 26, 2020)
6. Microsoft/various (2019, July 11). Database-Level Roles. *Microsoft SQL Documentation*. https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/database-level-roles?view=sql-server-ver15 (Accessed Apr 26, 2020)
7. Oracle (2002). 25: Managing User Privileges and Roles. *Oracle9i Database Administrator's Guide Release 2 (9.2)*. https://docs.oracle.com/cd/A97630_01/server.920/a96521/privs.htm (Accessed Apr 26, 2020)
8. Oracle (2020). 9.6 Comment Syntax. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/comments.html (Accessed Apr 26, 2020)
9. Uwagbole, S. O., Buchanan, W. J., & Fan, L. (2017, May). Applied machine learning predictive analytics to SQL injection attack detection and prevention. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (pp. 1087-1090). IEEE.
10. Halfond, W. G., Viegas, J., & Orso, A. (2006, March). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*(Vol. 1, pp. 13-15). IEEE.
11. Deepa, G., & Thilagam, P. S. (2016). Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, *74*, 160-180.
12. Cisco Security (2014, Nov 10). Understanding SQL Injection. *Cisco Security: Resources*. https://tools.cisco.com/security/center/resources/sql_injection (Accessed Apr 26, 2020)
13. Akamai (2015). The State of the Internet [security] / Q1 2015. *Akamai SOTI: Vol. 2 No. 1*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/2015-q1-internet-security-report.pdf (Accessed Apr 26, 2020)
14. Sucuri (2020). 2019 Website Threat Research Report. https://sucuri.net/wp-content/uploads/2020/01/20-sucuri-2019-hacked-report-1.pdf (Accessed Apr 26, 2020)

15. Portswigger (n.d.). SQL injection UNION attacks. *Web Security Academy: SQL injection*. https://portswigger.net/web-security/sql-injection/union-attacks (Accessed Apr 26, 2020)
16. DarkReading. (2010, Nov 1). The 10 Most Common Database Vulnerabilities. *Vulnerabilities/Threats*. https://www.darkreading.com/vulnerabilities---threats/the-10-most-common-database-vulnerabilities/d/d-id/1134676 (Accessed Apr 26, 2020)
17. NCCGroup (2014, July 21). Top 10 Common Database Security Issues. *NCCGroup blog*. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/july/top-10-common-database-security-issues/ (Accessed Apr 26, 2020)
18. Columbus, L. (2019, Feb 26). 74% Of Data Breaches Start With Privileged Credential Abuse. *Forbes*. https://www.forbes.com/sites/louiscolumbus/2019/02/26/74-of-data-breaches-start-with-privileged-credential-abuse/#6bb57a33ce45 (Accessed Apr 26, 2020)
19. OWASP (2020). SQL Injection. *OWASP www-community: attacks*. https://owasp.org/www-community/attacks/SQL_Injection (Accessed Apr 26, 2020)
20. Imperva (2011). Hacker Intelligence Summary Report – An Anatomy of a SQL Injection Attack. Report 4 in series *Hacker Intelligence Initiative*. https://www.imperva.com/docs/HII_An_Anatomy_of_a_SQL_Injection_Attack_SQLi.pdf (Accessed Apr 26, 2020)
21. Mackay, C. A. (2005, Jan 23). SQL Injection Attacks and Some Tips on How to Prevent Them. *CodeProject*. https://www.codeproject.com/Articles/9378/SQL-Injection-Attacks-and-Some-Tips-on-How-to-Prev (Accessed Apr 26, 2020)
22. Halfond, W. G., & Orso, A. (2007). Detection and prevention of SQL injection attacks. In *Malware Detection*(pp. 85-109). Springer, Boston, MA.
23. Portswigger (n.d.). SQL injection (second order). *Support Center: Issue Definitions*. https://portswigger.net/kb/issues/00100210_sql-injection-second-order (Accessed Apr 26, 2020)
24. Oracle (n.d.). Examples of Second Order SQL Injection Attack. *Tutorials: SQL Injection*. https://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_attacks2.htm
25. Ollmannn, G. (2001). Web Based Session Management: Best practices in managing HTTP-based client sessions. *Web Based Session Management: Whitepapers*. http://www.technicalinfo.net/papers/WebBasedSessionManagement.html (Accessed Apr 26, 2020)
26. Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. *Acm Sigplan Notices*, *41*(1), 372-382.
27. Sekar, R. (2009, February). An Efficient Black-box Technique for Defeating Web Application Attacks. In *NDSS*.
28. Kieyzun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009, May). Automatic creation of SQL injection and cross-site scripting attacks. In *2009 IEEE 31st international conference on software engineering* (pp. 199-209). IEEE.
29. Positive Technologies. (2017, Sept 14). *Web Application Attack Statistics: Q2 2017*. *Positive Technologies*. http://blog.ptsecurity.com/2017/09/web-application-attack-statistics-q2.html (Accessed Apr 26, 2020)
30. European Network and Information Security Agency (2012). ENISA Threat Landscape: Responding to the Evolving Threat Environment. https://www.enisa.europa.eu/publications/ENISA_Threat_Landscape (Accessed Apr 26, 2020)

31. European Network and Information Security Agency (2014). ENISA Threat Landscape 2014: Overview of current and emerging cyber-threats. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2014 (Accessed Apr 26, 2020)
32. European Network and Information Security Agency (2016). ENISA Threat Landscape 2015. https://www.enisa.europa.eu/publications/etl2015 (Accessed Apr 26, 2020)
33. European Network and Information Security Agency (2019). ENISA Threat Landscape Report 2018: 15 Top Cyberthreats and Trends. https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2018 (Accessed Apr 26, 2020)
34. Bradley, T. (2017, Aug 31). Tried and True - SQL Injection Still a Leading Method of Cyber Attack. *Alert Logic blog*. https://blog.alertlogic.com/blog/tried-and-true-sql-injection-still-a-leading-method-of-cyber-attack/ (Accessed Apr 26, 2020)
35. Penta. (2017, May 26). Web Application Threat Trend (WATT) Report Released from 2016 by Penta Security Systems. *Penta Security: Press Releases*. https://www.pentasecurity.com/press-releases/web-application-threat-trend-watt-report-released-2016-penta-security-systems/ (Accessed Apr 26, 2020)
36. Watson, K. (2019, Dec 16). November 2019 AppSec Intelligence Report. *Security Boulevard Security Bloggers Network*. https://securityboulevard.com/2019/12/november-2019-appsec-intelligence-report/ (Accessed Apr 26, 2020)
37. Yimin. (2018, Apr 25). 2017 DDoS and Web Application Attack Landscape. *NSFOCUS blog*. https://blog.nsfocusglobal.com/threats/vulnerability-analysis/2017-ddos-and-web-application-attack-landscape/ (Accessed Apr 26, 2020)
38. Positive Technologies (2019). Attacks On Web Applications: 2018 In Review. https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-application-attacks-2019-eng.pdf (Accessed Apr 26, 2020)
39. Akamai (2019). The State of the Internet [security] / 2019: Media Under Assault. *Akamai SOTI: Vol. 5 Special Ed*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-media-under-assault-report-2019.pdf (Accessed Apr 26, 2020)
40. Akamai. (2020, April). Web Attack Visualization. *Our Thinking: State of the Internet*. https://www.akamai.com/us/en/resources/our-thinking/state-of-the-internet-report/web-attack-visualization.jsp (Accessed Apr 26, 2020)
41. Akamai (2015). The State of the Internet [security] / Q2 2015. *Akamai SOTI: Vol. 2 No. 2*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/2015-q2-internet-security-report.pdf (Accessed Apr 26, 2020)
42. Akamai (2015). The State of the Internet [security] / Q3 2015. *Akamai SOTI: Vol. 2 No. 3*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/2015-q3-internet-security-report.pdf (Accessed Apr 26, 2020)
43. Akamai (2015). The State of the Internet [security] / Q4 2015. *Akamai SOTI: Vol. 2 No. 4*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/2015-q4-internet-security-report.pdf (Accessed Apr 26, 2020)
44. Akamai (2016). The State of the Internet [security] / Q1 2016. *Akamai SOTI: Vol. 3 No. 1*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q1-2016-internet-security-report.pdf (Accessed Apr 26, 2020)

45. Akamai (2016). The State of the Internet [security] / Q2 2016. *Akamai SOTI: Vol. 3 No. 2*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q2-2016-internet-security-report.pdf (Accessed Apr 26, 2020)
46. Akamai (2016). The State of the Internet [security] / Q3 2016. *Akamai SOTI: Vol. 3 No. 3*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q3-2016-internet-security-report.pdf (Accessed Apr 26, 2020)
47. Akamai (2016). The State of the Internet [security] / Q4 2016. *Akamai SOTI: Vol. 3 No. 4*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q4-2016-internet-security-report.pdf (Accessed Apr 26, 2020)
48. Akamai (2019). The State of the Internet [security] / 2019: A Year In Review. *Akamai SOTI: Vol. 5 Is. 6*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-a-year-in-review-report-2019.pdf (Accessed Apr 26, 2020)
49. Akamai (2019). The State of the Internet [security] / 2019: Financial Services Attack Economy. *Akamai SOTI: Vol. 5 Is. 4*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-financial-services-attack-economy-report-2019.pdf (Accessed Apr 26, 2020)
50. Akamai (2017). The State of the Internet [security] / Q1 2017. *Akamai SOTI: Vol. 4 No. 1*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q1-2017-internet-security-report.pdf (Accessed Apr 26, 2020)
51. Akamai (2017). The State of the Internet [security] / Q2 2017. *Akamai SOTI: Vol. 4 No. 2*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q2-2017-internet-security-report.pdf (Accessed Apr 26, 2020)
52. Akamai (2017). The State of the Internet [security] / Q3 2017. *Akamai SOTI: Vol. 4 No. 3*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q3-2017-internet-security-report.pdf (Accessed Apr 26, 2020)
53. Akamai (2017). The State of the Internet [security] / Q4 2017. *Akamai SOTI: Vol. 4 No. 4*. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q4-2017-internet-security-report.pdf (Accessed Apr 26, 2020)
54. Acohido, B. (2009, Mar 16). Website-infecting SQL injection attacks hit 450,000 a day. *abcNews Technology*. https://abcnews.go.com/Technology/story?id=7098329&page=1
55. Cid, D. (2014, Oct 8). Website Attacks – SQL Injection And The Threat They Present. *Sucuri blog*. https://blog.sucuri.net/2014/10/website-attacks-sql-injection-and-the-threat-they-present.html (Accessed Apr 26, 2020)
56. Gopalakrishnan, C. (2020, Mar 26). Zero-day malware, SQL injections rise in Q4 2019. *SC Media: The Cyber-Security source*. https://www.scmagazineuk.com/zero-day-malware-sql-injections-rise-q4-2019/article/1678381 (Accessed Apr 26, 2020)
57. CriticalWatchReport_18.pdf:::Alert Logic (2018). The State of Threat Detection 2018. *Critical Watch Report* series. https://www.alertlogic.com/assets/critcal-watch-report/CriticalWatchReport_18.pdf (Accessed Apr 26, 2020)
58. Positive Technologies (2017). Web Application Attack Trends 2017. https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-Application-Attack-Trends-2017-eng.pdf (Accessed Apr 26, 2020)
59. Nakar, O., & Azaria, J. (2019, June 13). SQL Injection Attacks: So Old, but Still So Relevant. Here's Why (Charts): Imperva. https://www.imperva.com/blog/sql-injection-attacks-so-old-but-still-so-relevant-heres-why-charts/ (Accessed Apr 26, 2020)

60. Imperva (2011). Imperva's Web Application Attack Report. *Web Application Attack Report, Ed. 1*. https://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed1.pdf (Accessed Apr 26, 2020)
61. Watson, K., (2020, Apr 10). What Vulnerabilities and Attacks Matter? Insights from Contrast Labs' AppSec Intelligence Report. *Security Boulevard*. https://securityboulevard.com/2020/04/what-vulnerabilities-and-attacks-matter-insights-from-contrast-labs-appsec-intelligence-report/ (Accessed Apr 26, 2020)
62. Positive Technologies (2017). Web Application Attack Statistics: 2017 In Review. https://www.ptsecurity.com/upload/corporate/ww-en/analytics/Web-application-attacks-2018-eng.pdf (Accessed Apr 26, 2020)
63. European Network and Information Security Agency (2017). ENISA Threat Landscape 2015: 15 Top Cyber-Threats and Trends. https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2016 (Accessed Apr 26, 2020)
64. Danchev, D. (2010, Feb 9). Reports: SQL injection attacks and malware led to most data breaches. *ZDNet*. https://www.zdnet.com/article/reports-sql-injection-attacks-and-malware-led-to-most-data-breaches/ (Accessed Apr 26, 2020)
65. Ponemon Institute (2014). The SQL Injection Threat Study. https://www.ponemon.org/local/upload/file/DB%20Networks%20Research%20Report%20FINAL5.pdf (Accessed Apr 26, 2020)
66. Tuby, S. (2011, Sept 20). SQL Injection: By The Numbers. *Imperva blog*. https://www.imperva.com/blog/sql-injection-by-the-numbers/ (Accessed Apr 26, 2020)
67. Avital, N. (2019, Jan 9). The State of Web Application Vulnerabilities in 2018. *Imperva blog*. https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/ (Accessed Apr 26, 2020)
68. Bekerman, D., & Yerushalmi, S. (2020, Jan 23). The State of Vulnerabilities in 2019. *Imperva blog*. https://www.imperva.com/blog/the-state-of-vulnerabilities-in-2019/ (Accessed Apr 26, 2020)
69. Common Weakness Enumeration. (2019, Sept 18). 2019 CWE Top 25 Most Dangerous Software Errors. *CWE Top 25*. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (Accessed Apr 26, 2020)
70. Vijayan, J. (2019, Nov 27). SQL Injection Errors No Longer the Top Software Security Issue. *Vulnerabilities/Threats*. https://www.darkreading.com/vulnerabilities---threats/sql-injection-errors-no-longer-the-top-software-security-issue/d/d-id/1336481 (Accessed Apr 26, 2020)
71. Özkan, S./MITRE corporation (2020). Vulnerabilities By Type. *CVE details*. https://www.cvedetails.com/vulnerabilities-by-types.php (Accessed Apr 26, 2020)
72. MITRE corporation (2020, Mar 23). Frequently Asked Questions. *MITRE: Common Vulnerabilities and Exposures*. https://cve.mitre.org/about/faqs.html (Accessed Apr 26, 2020)
73. Vijayan, J. (2019, Nov 25). Most Organizations Have Incomplete Vulnerability Information. *Vulnerabilities/Threats*. https://www.darkreading.com/vulnerabilities---threats/most-organizations-have-incomplete-vulnerability-information/d/d-id/1336460 (Accessed Apr 26, 2020)

74. jerichoattrition (2019, June 30). WhiteSource on 'Open Source Vulnerability Databases' – Errata. *OSVDB blog*. https://blog.osvdb.org/2019/06/30/whitesource-on-open-source-vulnerability-databases-errata/ (Accessed Apr 26, 2020)
75. Bannister, A. (2020, Mar 19). Vulnerabilities in web and app frameworks fall, but weaponization rate jumps – study. *The Daily Swig*. https://portswigger.net/daily-swig/vulnerabilities-in-web-and-app-frameworks-fall-but-weaponization-rate-jumps-study (Accessed Apr 26, 2020)
76. Bisson, D. (2015, Nov 3). The TalkTalk Breach: Timeline of a Hack. *Tripwire: The State of Security*. https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/the-talktalk-breach-timeline-of-a-hack/ (Accessed Apr 26, 2020)
77. UK Information Commissioner's Office (2016, Nov 1). TalkTalk cyber attack – how the ICO's investigation unfolded. *ICO*. https://ico.org.uk/about-the-ico/news-and-events/talktalk-cyber-attack-how-the-ico-investigation-unfolded/ (Accessed Apr 26, 2020)
78. Cluley, G., (2018, Nov 20). Two friends jailed for TalkTalk hack plot. *Security Boulevard*. https://securityboulevard.com/2018/11/two-friends-jailed-for-talktalk-hack-plot/ (Accessed Apr 26, 2020)
79. Rashid, F. Y. (2011, May 24). Sony Woes Continue With SQL Injection Attacks. *eWeek blog*. https://www.eweek.com/blogs/security-watch/sony-woes-continue-with-sql-injection-attacks (Accessed Apr 26, 2020)
80. Martin, A. (2011, Sept 22). LulzSec's Sony Hack Really Was as Simple as It Claimed. *The Atlantic: Technology*. https://www.theatlantic.com/technology/archive/2011/09/lulzsecs-sony-hack-really-was-simple-it-claimed/335527/ (Accessed Apr 26, 2020)
81. Hoffman, S. (2011, June 3). Sony Web Site Hack Compromises 1 Million Accounts. *CRN*. https://www.crn.com/news/security/229900144/sony-web-site-hack-compromises-1-million-accounts.htm (Accessed Apr 26, 2020)
82. Troy Hunt. (2011, June 6). A brief Sony password analysis. *troyhunt.com*. https://www.troyhunt.com/brief-sony-password-analysis/ (Accessed Apr 26, 2020)
83. Troy Hunt (n.d.). Pwned websites. *';--have i been pwned?*. https://haveibeenpwned.com/PwnedWebsites (Accessed Apr 26, 2020)
84. Office of Public Affairs (2014, Sept 30). Four Members of International Computer Hacking Ring Indicted for Stealing Gaming Technology, Apache Helicopter Training Software. *The United States Department of Justice*. https://www.justice.gov/opa/pr/four-members-international-computer-hacking-ring-indicted-stealing-gaming-technology-apache (Accessed Apr 26, 2020)
85. United States of America v. Leroux. Criminal Action No. 13-78-GMS. United States District Court for the District Of Delaware. 2015. Published by U.S. Dept. of Justice at https://www.justice.gov/file/318656/download (Accessed Apr 26, 2020)
86. Federal Trade Commission (2012, Mar 27). FTC Charges That Security Flaws in RockYou Game Site Exposed 32 Million Email Addresses and Passwords. *Federal Trade Commission*. https://www.ftc.gov/news-events/press-releases/2012/03/ftc-charges-security-flaws-rockyou-game-site-exposed-32-million (Accessed Apr 26, 2020)
87. Sluis, S. (2019, Feb 26). Flight To Quality Bankrupts RockYou. *adexchanger*. https://www.adexchanger.com/online-advertising/flight-to-quality-bankrupts-rockyou/ (Accessed Apr 26, 2020)

88. Nicole, K. (2007, May 29). RockYou Takes the Widget Lead on Facebook. *Mashable*. https://mashable.com/2007/05/29/rockyou-facebook/ (Accessed Apr 26, 2020)
89. Cubrilovic, N. (2009, Dec 15). RockYou Hack: From Bad To Worse. *TechCrunch*. https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/ (Accessed Apr 26, 2020)
90. O'dell J. (2009, Dec 16). RockYou Hacker: 30% of Sites Store Plain Text Passwords. *NyTimes: Technology*. https://archive.nytimes.com/www.nytimes.com/external/readwriteweb/2009/12/16/16readwriteweb-rockyou-hacker-30-of-sites-store-plain-text-13200.html (Accessed Apr 26, 2020)
91. Leyden, J. (2010, January 21). RockYou hack reveals easy-to-crack passwords. *The Register*. https://www.theregister.co.uk/2010/01/21/lame_passwords_exposed_by_rockyou_hack/ (Accessed Apr 26, 2020)
92. Cheney, J. S. (2010). Heartland Payment Systems: lessons learned from a data breach. *FRB of Philadelphia-Payment Cards Center Discussion Paper*, (10-1). https://www.phil.frb.org/-/media/consumer-finance-institute/payment-cards-center/publications/discussion-papers/2010/d-2010-january-heartland-payment-systems.pdf (Accessed Apr 26, 2020)
93. Gordover, M. (2015, Mar 19). Throwback Thursday: Lessons Learned from the 2008 Heartland Breach. *ObserveIT*. https://www.observeit.com/blog/throwback-thursday-lessons-learned-from-the-2008-heartland-breach/ (Accessed Apr 26, 2020)
94. Swinhoe, D. (2020, April 17). The 15 biggest data breaches of the 21st century. *CSO Online*. https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html (Accessed Apr 26, 2020)
95. United States of America v. Drinkman, Kalinin, Kotov, Rytikov, and Smilianets. Criminal Action No. 09-626 (JBS) (S-2); 18 U.S.C. §§ 371 , 1030, 1343, 1349, and 2. United States District Court for the District Of New jersey. 2018. Published by U.S. Dept. of Justice at https://www.justice.gov/usao-nj/file/770791/download (Accessed Apr 26, 2020)
96. Zetter, K. (2009, Dec 29). Albert Gonzalez Pleads Guilty in Heartland, 7-11 Breaches -- Updated. *Wired*. https://www.wired.com/2009/12/heartland-guilty-plea/ (Accessed Apr 26, 2020)
97. Office of Public Affairs (2018, Feb 15). Two Russian Nationals Sentenced to Prison for Massive Data Breach Conspiracy. *The United States Department of Justice*. https://www.justice.gov/opa/pr/two-russian-nationals-sentenced-prison-massive-data-breach-conspiracy (Accessed Apr 26, 2020)
98. Leyden, J. (2013, July 29). 'World's BIGGEST online fraud': Suspect's phone had 'location' switched on. *The Register*. https://www.theregister.co.uk/2013/07/29/how_russian_megahack_suspects_got_tracked/ (Accessed Apr 26, 2020)
99. Vaas, L. (2018, Feb 19). Hackers sentenced for SQL injections that cost $300 million. *Naked Security by Sophos*. https://nakedsecurity.sophos.com/2018/02/19/hackers-sentenced-for-sql-injections-that-cost-300-million/ (Accessed Apr 26, 2020)

100. Maor & Amichai (2004). *SQL Injection Signatures Evasion*. https://www.imperva.com/docs/IMPERVA_HII_SQL-Injection-Signatures-Evasion.pdf (Accessed Apr 26, 2020)
101. OWASP/various (2019, Mar 11). Input Validation Cheat Sheet. *OWASP Cheat Sheet Series on Github*. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Input_Validation_Cheat_Sheet.md (Accessed Apr 26, 2020)
102. Fonseca, J., Seixas, N., Vieira, M., & Madeira, H. (2013). Analysis of field data on web security vulnerabilities. *IEEE transactions on dependable and secure computing*, *11*(2), 89-100.
103. Scholte, T., Robertson, W., Balzarotti, D., & Kirda, E. (2012, March). An empirical analysis of input validation mechanisms in web applications and languages. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (pp. 1419-1426).
104. Zdrnja, B. (2008, Jan 9). Mass Exploits with SQL injection. *InfoSec Handlers Diary Blog*. https://isc.sans.edu/diary/Mass+exploits+with+SQL+Injection/3823 (Accessed Apr 26, 2020)
105. Maciejak, D., & Lovet, G. (2009, September). Botnet-powered SQL injection attacks: A deeper look within. In *Virus Bulletin Conference* (pp. 286-288). https://www.virusbulletin.com/uploads/pdf/conference_slides/2009/Maciejak-Lovet-VB2009.pdf (Accessed Apr 26, 2020)
106. tcsl (Roger) (2008, Jan 3). Neosploit January 2008. *explabs @ blogspot.com*. http://explabs.blogspot.com/2008/01/ (Accessed Apr 26, 2020)
107. SANS Institute (2008, Jan 8). Newsletters - Newsbites: Volume X - Issue #2. *Newsletters - Newsbites*. https://www.sans.org/newsletters/newsbites/x/2 (Accessed Apr 26, 2020)
108. SecurityFocus. (2008, Jan 10). SQL attack continues to infect Web sites. *SecurityFocus briefs*. https://www.securityfocus.com/brief/660 (Accessed Apr 26, 2020)
109. Fendley, S. (2008, Jan 4). Realplayer Vulnerability. *InfoSec Handlers Diary Blog*. https://isc.sans.edu/diary/Realplayer+Vulnerability/3810 (Accessed Apr 26, 2020)
110. OWASP/various. (2019, Aug 28). SQL Injection Prevention Cheat Sheet. *OWASP Cheat Sheet Series on Github*. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.md (Accessed Apr 26, 2020)
111. Oracle (n.d.). Using Prepared Statements. *Oracle Documentation: The Java™ Tutorials*. https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html (Accessed Apr 26, 2020)
112. Ray, D., & Ligatti, J. (2014, October). Defining injection attacks. In *International Conference on Information Security: 17th International Conference, Hong Kong, China, October 12-14, 2014, Proceedings* (pp. 425-441). Springer, Cham. https://www.google.com/books/edition/_/Mu8sBQAAQBAJ?hl=en (Accessed Apr 26, 2020)
113. Anley, C. (2002). Advanced SQL injection in SQL server applications. https://www.cgisecurity.com/lib/advanced_sql_injection.pdf (Accessed Apr 26, 2020)
114. Prakash, A. (n.d.). *Hack the world - Ethical Hacking*. https://www.google.com/books/edition/_/o62cCgAAQBAJ (Accessed Apr 26, 2020)

115. Veracode (2015). State of Software Security Report: focus on Application Development. Supplement to *State of Software Security Vol. 6*. https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-focus-on-application-development.pdf (Accessed Apr 26, 2020)
116. The PHP Group (2005, Apr 16). pg_prepare. *PHP Manual – PostgreSQL functions*. https://www.php.net/manual/en/function.pg-prepare.php (Accessed Apr 26, 2020)
117. Atwood, J. (2005, Apr 26). Give me parameterized SQL, or give me death. *Coding Horror*. https://blog.codinghorror.com/give-me-parameterized-sql-or-give-me-death/ (Accessed Apr 26, 2020)
118. Oracle (2020). 24.2 Using Stored Routines. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/stored-routines.html (Accessed Apr 26, 2020)
119. Microsoft/various (2017, Mar 14). Stored Procedures (Database Engine). *Microsoft SQL Documentation*. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-ver15 (Accessed Apr 26, 2020)
120. SuperUser Account (n.d.). SQL injection attack and prevention using stored procedure. *Canarys*. https://www.ecanarys.com/Blogs/ArticleID/112/SQL-injection-attack-and-prevention-using-stored-procedure (Accessed Apr 26, 2020)
121. Microsoft/various (2017, Mar 16). Parameters. *Microsoft SQL Documentation*. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/parameters?view=sql-server-ver15 (Accessed Apr 26, 2020)
122. Rahmanovic, T., Kravchuk, V., & Opisov, K. (2008, May 4). Bug #36496 – SQL injection in stored procedures - type checking non functioning. *MySQL bugs forum*. https://bugs.mysql.com/bug.php?id=36496 (Accessed Apr 26, 2020)
123. Oracle (2020). 24.8 Restrictions on Stored Programs. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html (Accessed Apr 26, 2020)
124. El Haourani, L., Elkalam, A. A., & Ouahman, A. A. (2018, October). Knowledge Based Access Control a model for security and privacy in the Big Data. In *Proceedings of the 3rd International Conference on Smart City Applications* (pp. 1-8).
125. Doan, D. (2016, June 15). MemSQL 5.1 Enhances Security for Real-Time Enterprises. *memSQL blog*. https://www.memsql.com/blog/rbac-security/ (Accessed Apr 26, 2020)
126. Snowflake (2020, April). Overview of Stored Procedures. *snowflake documentation*. https://docs.snowflake.com/en/sql-reference/stored-procedures-overview.html (Accessed Apr 26, 2020)
127. Oracle (2020). 24.6 Stored Object Access Control. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/stored-objects-security.html (Accessed Apr 26, 2020)
128. OWASP/various (2020, Feb 21). Database Security Cheat Sheet. *OWASP Cheat Sheet Series on Github*. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Database_Security_Cheat_Sheet.md (Accessed Apr 26, 2020)
129. Uwagbole, S. O., Buchanan, W. J., & Fan, L. (2017, September). An applied pattern-driven corpus to predictive analytics in mitigating SQL injection attack. In *2017 Seventh International Conference on Emerging Security Technologies (EST)* (pp. 12-17). IEEE.

130. Buehrer, G., Weide, B. W., & Sivilotti, P. A. (2005, September). Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware* (pp. 106-113).
131. Halfond, W. G., & Orso, A. (2005, November). AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 174-183).
132. Antunes, N., Laranjeiro, N., Vieira, M., & Madeira, H. (2009, September). Effective detection of SQL/XPath injection vulnerabilities in web services. In *2009 IEEE International Conference on Services Computing* (pp. 260-267). IEEE.
133. Shahriar, H., & Zulkernine, M. (2012). Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, *44*(3), 1-46.
134. Baca, D., Petersen, K., Carlsson, B., & Lundberg, L. (2009, March). Static code analysis to detect software security vulnerabilities-does experience matter?. In *2009 International Conference on Availability, Reliability and Security* (pp. 804-810). IEEE.
135. Kar, D., Panigrahi, S., & Sundararajan, S. (2016). SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM. *Computers & Security*, *60*, 206-225.
136. Haldar, V., Chandra, D., & Franz, M. (2005, December). Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference (ACSAC'05)*(pp. 9-pp). IEEE.
137. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., & Evans, D. (2005, May). Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference* (pp. 295-307). Springer, Boston, MA.
138. Xu, W., Bhatkar, S., & Sekar, R. (2006, August). Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium* (pp. 121-136).
139. Livshits, B. (2012). Dynamic taint tracking in managed runtimes. *Microsoft Research Technical Report*. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-6.pdf (Accessed Apr 26, 2020)
140. Shar, L. K., & Tan, H. B. K. (2012). Defeating SQL injection. *Computer*, *46*(3), 69-77.
141. Fonseca, J., Vieira, M., & Madeira, H. (2007, December). Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *13th Pacific Rim international symposium on dependable computing (PRDC 2007)* (pp. 365-372). IEEE.
142. Antunes, N., & Vieira, M. (2009, November). Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing* (pp. 301-306). IEEE.
143. Doupé, A., Cova, M., & Vigna, G. (2010, July). Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 111-131). Springer, Berlin, Heidelberg.
144. Khoury, N., Zavarsky, P., Lindskog, D., & Ruhl, R. (2011, October). An analysis of black-box web application security scanners against stored SQL injection. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing* (pp. 1095-1101). IEEE.

145. Vieira, M., Antunes, N., & Madeira, H. (2009, June). Using web security scanners to detect vulnerabilities in web services. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks* (pp. 566-571). IEEE.
146. Schwartz, E. J., Avgerinos, T., & Brumley, D. (2010, May). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy* (pp. 317-331). IEEE.
147. Various (2016, June 20). What is managed code? *Microsoft .NET Documentation*. https://docs.microsoft.com/en-us/dotnet/standard/managed-code (Accessed Apr 26, 2020)
148. Halfond, W. G., Orso, A., & Manolios, P. (2006, November). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (pp. 175-185).
149. Lee, I., Jeong, S., Yeo, S., & Moon, J. (2012). A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, *55*(1-2), 58-68.
150. Coret, J. A. (2004, Sep 25). The WASP Project. *Savannah non-GNU*. http://www.nongnu.org/wasp/ (Accessed Apr 26, 2020)
151. Joxean Koret (updated 2018). Latest Blog entries. *joxeankoret.com*. http://joxeankoret.com/ (Accessed Apr 26, 2020)
152. Cover, T. M., & Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons. Preview at https://www.google.com/books/edition/_/VWq5GG6ycxMC?hl=en (Accessed Apr 26, 2020)
153. Shahriar, H., & Zulkernine, M. (2012, October). Information-theoretic detection of sql injection attacks. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering* (pp. 40-47). IEEE.
154. Jang, Y. S., & Choi, J. Y. (2014). Detecting SQL injection attacks using query result size. *Computers & Security*, *44*, 104-118.
155. Breitling, W. (2003, February). Fallacies of the cost based optimizer. In *Hotsos Symposium on Oracle Performance, Dallas, Texas*.
156. Halfond, W. G. J. (n.d.). SQL Injection Application Testbed. *William G.J. Halfond's pages at University of Southern California*. https://viterbi-web.usc.edu/~halfond/testbed.html (Accessed Apr 26, 2020)
157. Valeur, F., Mutz, D., & Vigna, G. (2005, July). A learning-based approach to the detection of SQL attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 123-140). Springer, Berlin, Heidelberg.
158. Kruegel, C., Mutz, D., Valeur, F., & Vigna, G. (2003, October). On the detection of anomalous system call arguments. In *European Symposium on Research in Computer Security* (pp. 326-343). Springer, Berlin, Heidelberg.
159. Scholte, T., Balzarotti, D., & Kirda, E. (2011, February). Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *International Conference on Financial Cryptography and Data Security*(pp. 284-298). Springer, Berlin, Heidelberg.
160. Check Point (2015, May 7). The Latest SQL Injection Trends. *Check Point Software Technologies Ltd. blog*. https://blog.checkpoint.com/2015/05/07/latest-sql-injection-trends/ (Accessed Apr 26, 2020)

161. Veracode (2016). How Do Vulnerabilities Get Into Software?. https://www.veracode.com/sites/default/files/Resources/Whitepapers/how-vulnerabilities-get-into-software-veracode.pdf (Accessed Apr 26, 2020)
162. Wenham, P. (2012, Sep 7). Security Think Tank: No quick fix to SQLi attacks. *ComputerWeekly*. https://www.computerweekly.com/opinion/Security-Think-Tank-No-quick-fix-to-SQLi-attacks (Accessed Apr 26, 2020)
163. CodeCourse (2015, Oct 6). PHP Security (playlist). *YouTube*. https://www.youtube.com/playlist?list=PLfdtiltiRHWFsPxAGO-SVPGhCbCwKWF_N (Accessed Apr 26, 2020)
164. CodeCourse (2015, Oct 14). PHP Security: SQL Injection. *PHP Security playlist by CodeCourse on YouTube*. https://www.youtube.com/watch?v=cgwWpd4SqIM&list=PLfdtiltiRHWFsPxAGO-SVPGhCbCwKWF_N&index=10&t=0s (Accessed Apr 26, 2020)
165. First. Common Vulnerability Scoring System SIG. (n.d.). *First CVSS*. https://www.first.org/cvss/ (Accessed Apr 26, 2020)
166. Halfond, W. G. J. (n.d.). Web Security. *William G.J. Halfond's pages at University of Southern California*. https://viterbi-web.usc.edu/~halfond/security.html (Accessed Apr 26, 2020)
167. Halfond, W. G. J. (n.d.). List of Publications. *William G.J. Halfond's pages at University of Southern California*. https://viterbi-web.usc.edu/~halfond/publications.html (Accessed Apr 26, 2020)
168. Orso, A. (2020). Alex Orso - Peer-reviewed Publications. Alessandro (Alex) Orso's pages at Georgia Tech College of Computing. https://www.cc.gatech.edu/home/orso/papers/index.html (Accessed Apr 26, 2020)
169. Manolios, P. (2019). Research. *Panagiotis (Pete) Manolios' pages at Northeastern University's Khoury College of Computer Sciences*. http://www.ccs.neu.edu/~pete/research.html
170. Viega, J., Bloch, J. T., & Chandra, P. (2001). Applying aspect-oriented programming to security. *Cutter IT Journal*, *14*(2), 31-39.
171. Wikipedia contributors (2020, Apr 21). *Entscheidungsproblem*. *Wikipedia*. https://en.wikipedia.org/wiki/Entscheidungsproblem (Accessed Apr 26, 2020)
172. Akamai SIRT Alerts (2016, Jan 12). How Web Applications Become Seo Pawns. *The Akamai Blog*. https://blogs.akamai.com/2016/01/how-web-applications-become-seo-pawns.html (Accessed Apr 26, 2020)
173. Veracode (2015). State of Software Security Report: Focus on Industry Verticals. *State of Software Security, Vol. 6*. https://www.veracode.com/sites/default/files/Resources/Reports/state-software-security-report-june-2015-report.pdf (Accessed Apr 26, 2020)