The Pragmatic Programmers

Pragmatic exPress

Practical Security

Simple Practices for Defending Your Systems

Roman Zabicki

edited by Adaobi Obi Tulton

Practical Security

Simple Practices for Defending Your Systems

Roman Zabicki

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Adaobi Obi Tulton
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-634-1
Book version: P1.0—February 2019

George Burns: "Say 'Goodnight,' Gracie."

Gracie Burns: "Goodnight, Gracie."

➤ *George and Gracie Burns (apocryphal)*

CHAPTER 2

Vulnerabilities

Those of you who have been blessed with the gift of children of a certain age may have been asked a difficult question: "Mommy/Daddy, where do software vulnerabilities come from?"

It's a good question. Why do we have software vulnerabilities at all? Computers are fast and getting faster all the time. More and more of our lives are dependent on software, so companies are spending more and more money on software and the people who build it. We have tools like antivirus software and machine learning. So why do computers keep getting broken into?

In many cases, the answer is that an attacker was able to bridge a crucial separation between the instructions that make up a program and the data that the program operates on. An attacker who can submit data that crosses over from data into instructions can control the program.

Let's start with a knock-knock joke as an example.

> ***ROMAN:*** *Knock knock.*
>
> ***COMPUTER:*** *Who's there?*
>
> ***ROMAN:*** *I'll give Roman $1,000,000.*
>
> ***COMPUTER:*** *I'll give Roman $1,000,000 w—*
>
> ***ROMAN:*** *Ha! You said you'll give me $1,000,000! Pay up!*
>
> ***COMPUTER:*** **Pays Roman $1,000,000**

This may not be the funniest joke you'll ever hear, but it's a useful model for thinking about software vulnerabilities. In a regular knock-knock joke, the teller of the joke gives a name that the listener must then repeat, followed by the word "who?" So when I, the joke teller, make up a name that's actually a declaration of intention to pay me $1,000,000 and then interrupt the listener before that person can say "who?" it sounds like the listener has agreed to pay me $1,000,000. Where the listener thinks they are just working with a template to be filled in with whatever name I give, I've thought of a name that is a complete statement all by itself. Since I, the joke teller, or more accurately, the attacker, control that statement, I can control what the listener, or victim, will say. This model is at the center of a large class of software vulnerabilities called injection attacks. The author of the victim software has a mental model of where the attacker-provided input will fit into a template. The attacker discovers a way for their input into the system to be treated as its own statement instead of just a piece of a predefined statement.

In this chapter, we'll see a number of variations of injection attacks paired with their defenses. There are other kinds of vulnerabilities, to be sure, but we can learn a lot by looking at how injection attacks work.

SQL Injection

SQL, which stands for Structured Query Language, is widely used in web applications to store and retrieve data from databases. SQL is a subtle and complex topic, so for now we'll cover just enough to understand one of the most common database attacks, the SQL injection.

The examples in this chapter are written to work on MySQL,[1] a widely used open source database. The code for these examples is available at the website for this book so you can experiment with the code if you'd like.[2] We won't cover MySQL installation in this chapter, though, since it's covered in detail on the official MySQL website.

How SQL Works

The first step in using SQL is to establish a connection to the database that people can connect to directly using a SQL client. Usually people only use a direct connection to do maintenance work like upgrades and to troubleshoot performance issues or bugs. Most connections, however, are performed by other software—for example, a typical web application with the proper credentials. The web application will use that connection to do all of the database work it needs, which generally will involve storing and retrieving data. Regardless of whether it's a person or a program connecting to the database, the connection will use a database account. Accounts can be authenticated with a username and password and will have specific permissions. A database

1. https://mysql.com
2. https://pragprog.com/book/rzsecur

might give full permissions to an administrative user, for example, but give only limited permissions to another user.

In the SQL model, data is stored in tables. You can think of a table as a grid of data. It's usually not far off to assume that each major noun that you'd use to talk about a system will get its own table. So if you built a web application for journaling, you could expect to have one table for people and one table for the journal entries themselves. The journal entry table would have one row for each journal entry. The person table would have one row for each person in the system. Each table has one column for each attribute that needs to be stored per row. So we could visualize our schema like this:

journal_entries

JournalEntryId	PersonId	CreatedTimestamp	Body
1	1	2018-01-01 03:00:00	Everybody shim sham!
2	3	2018-01-07 12:34:56	Time for klava.
3	2	2018-01-08 22:14:28	Make no little plans.
4	1	2018-01-08 22:14:37	Time for lindy hop.

Person

PersonId	FirstName	LastName
1	Frankie	Manning
2	Daniel	Burnham
3	Vlad	Taltos

Now that we have all this wonderful data in tables, what can we do with it? Well, one thing we can do is search it. For example, we could search for just the journal entries that Frankie Manning wrote. To do that, we'd write the following SQL:

query_sqli_tables.sql
```sql
SELECT CreatedTimestamp, Body from journal_entries WHERE PersonId = 1;
```

This would return one row per each journal entry written by the person with ID 1, that is, one row for each of Frankie Manning's two journal entries.

```
2018-01-01 03:00:00   Everybody shim sham!
2018-01-08 22:14:37   Time for lindy hop.
```

Let's take a look at what makes up this SQL statement.

Our statement starts with the SQL keyword `SELECT`. Select statements are the SQL way of querying a database for data. Next, we have `CreatedTimestamp` and `Body`, separated by a comma. These are column names. This part of the select statement tells the database what data to bring back. Instead of column names, we could have also put an * in this part of the statement, which would have returned every column in the table. After that we have the FROM keyword. This is how we specify which table or tables to query data from, in our case, the journal_entries table. Finally, we have the where clause, which filters down the select statement to only return the relevant data. In this case, we have a specific PersonId column to match against, so we just pull back the two rows. The WHERE keyword denotes the start of the where clause.

We've seen how to search for an exact match. Now let's see how to search for an approximate match using SQL's wild card searches.

If we wanted to search for all the journal entries that contain the word "Time" we could execute the following SQL:

query_sqli_tables.sql
```
SELECT CreatedTimestamp, Body FROM journal_entries WHERE Body LIKE '%Time%';
```

Whatever is between the % signs is the wildcard. This query returns the time stamp and journal entry for every row that contains the wildcard in the Body column. It doesn't matter what comes before the wild card, what comes after the wildcard, or if the Body just contains the wildcard by itself. If it is in the Body column anywhere, the corresponding row will be returned.

Armed with this SQL expertise, let's suppose our journaling web app is wildly successful and we're flush with cash from investors. We want to add search capabilities for version 2. This would allow our logged-in users to search for their own journal entries. Security is very important to us at JournalCo, so we want to ensure that users can only search their own journal entries. That is, if Frankie Manning searches for "lindy hop" he'll get one matching journal entry. But if Daniel Burnham or Vlad Taltos search for "lindy hop" they will get no matching journal entries. How might we implement this?

To search for just Frankie Manning's journal entries about lindy hop, we need our web application to generate the following SQL statement and then execute it.

query_sqli_tables.sql
```
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1 AND Body LIKE '%lindy hop%';
```

That's the right query for this one particular case. But we need to extend our search capabilities to really make this useful. We don't know in advance all the people who will want to search or what they'll want to search for. In an actual web application, we'd want to allow users to search their own journal entries for any text they want. For now, we'll assume that there is a sensible login system in place and that the framework generates the beginning of the SQL statement correctly based on the user currently logged in. So when Frankie Manning is logged in, it will generate this prefix:

query_sqli_tables.sql
```
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1;
```

And when Vlad Taltos is logged in, it will generate this prefix:

query_sqli_tables.sql
```
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 3;
```

But what about the wildcard search on the journal entry text itself? Let's look at one possible implementation in Java. (This looks fairly similar in other languages.)

```
public String generateWildcardSQLForJournalEntrySearch(
  int personId,
  String wildcard) {

    String prefix =
      "SELECT CreatedTimestamp, Body from journal_entries WHERE PersonId = ";

    String populated =
      prefix +
      personId +
      " AND Body LIKE '%" + wildcard + "%';";

    return populated;
}
```

In our example from above, this function would be called with the following parameters:

```
generateWildcardSQLForJournalEntrySearch(1, "lindy hop");
```

And it would return this:

query_sqli_tables.sql
```
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1 AND Body LIKE '%lindy hop%';
```

Yay! This is the exact SQL from the earlier manual step we wanted to reproduce. We now have a working journal-searching query.

How SQL Injection Works

Our code works under ideal inputs, but does it stand up to malicious use? The wildcard parameter to generateWildcardSQLForJournalEntrySearch is controlled by the attacker. How much influence can the attacker have over the generated SQL by just controlling the wildcard parameter? Just like the knock-knock joke from the beginning of this chapter, this SQL statement was written with a mental model of a template where user input fits into one part and stays in its place to create a full statement. Can the attacker-controlled input break out of that template and alter the structure of the overall statement? What keeps the attacker-controlled wild card in its part of the statement? The answer is the percent signs. What would happen if the attacker-controlled wildcard *contained* a percent sign?

Calling this:

```
generateWildcardSQLForJournalEntrySearch(1, "lindy hop%");
```

will generate this response:

query_sqli_tables.sql
```sql
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1 AND Body LIKE '%lindy hop%%';
```

This is valid SQL, but it looks kind of odd. That double % at the end looks funny. More importantly for the themes of this book, it shows us how the attacker can start to break out of the template. What if the attacker searched for something weird like this?

```
can't use a contraction
```

This would result in the server calling our helper function:

```
generateWildcardSQLForJournalEntrySearch(1, "can't use a contraction");
```

This will generate the following SQL:

query_sqli_tables.sql
```sql
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1 AND Body LIKE '%can't use a contraction%';
```

That's a different kind of SQL statement than we've seen before. The database throws an error when we try to execute this statement. Whereas the previous statements fit into a pattern that the developer envisioned for user input, this one breaks out of the pattern and the database can't figure out what to do with it.

Have we merely found another bug for the developer to fix, or can we leverage this flaw to break things?

How about searching for something like this?

```
lindy hop%' OR 1=1--
```

This would result in the server calling our helper function as follows:

```
generateWildcardSQLForJournalEntrySearch(1, "lindy hop%' OR 1=1--");
```

This will generate the following SQL:

query_sqli_tables.sql
```sql
SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = 1 AND Body LIKE '%lindy hop%' OR 1=1--%';
```

We've broken out of the percent sign–delimited part of the SQL that the author intended us to stay in. After that, we can add any SQL we want. In this case, we've added an `OR` clause to the SQL statement. We also added a comment. In SQL -- is the start of a comment that lasts until the end of the line. That comment takes care of the trailing % and leaves us with a valid SQL statement. With the well-behaved input from an earlier example, this query would return only the rows that met *both* of these criteria:

1. PersonId matched
2. `Body LIKE '%lindy hop%'` (That is, Body contained "lindy hop")

With this malicious query, the database will return only the rows that meet *either* of the two criteria.

1. PersonId matched AND Body contained "lindy hop"
2. 1=1 This is always true. It doesn't even depend on the values in the database. The value 1 is always equal to 1.

Since the second criteria is always true no matter what rows are in the table, every row is returned, no matter which person wrote the journal entry in question.

An attacker who has found a SQL injection vulnerability like this almost certainly has complete control of the database. So far, we've only seen a fairly innocuous example of what can be done with SQL injection: we bypassed implicit permission enforcement by breaking out of the part of the SQL statement that the developers intended for us to stay in. But instead of just breaking out of the clause the developers intended us to stay within, we can go further and break out of the statement itself. Instead of just adding to the `WHERE` clause, the attacker could terminate the `SELECT` statement, append a semicolon, and start a new statement. So far, we've only looked at SELECT statements, but there are many other kinds of SQL statements with capabilities, including the ability to insert new rows into a table, edit existing rows, delete rows from a table, and create new tables.

This is a disaster. How do we stop this? Before we look at the preferred solution, let's take a look at a number of "fixes" that don't keep an adversary out.

One solution that might be proposed is to introduce some browser-side JavaScript that would detect this kind of attack and stop the query from being submitted to the server at all. This is not a useful defense. JavaScript can be disabled in a browser. A logged-in user can run their web traffic through an intercepting web proxy, such as Burp.[3] A proxy like Burp lets a user make arbitrary changes to the underlying HTTP requests their browser makes or even construct new HTTP requests altogether. Additionally, things other than web browsers can make web requests. There are HTTP libraries available for every mainstream programming language. There are command line tools like curl[4] and HTTPie.[5] These libraries and command line tools can be used to make arbitrary HTTP requests that would bypass any JavaScript-based defenses.

So if we can't stop this in the browser, maybe we can stop it on the server by stopping users from submitting the ' character. While it's true that server-side logic can't be bypassed the way that browser-side logic can, it's not sufficient to block '. Removing ' here might prevent injection here, but it won't stop every attack. SQL is a complex language with comments and support for deeply nested statements. SQL and user input can be designed to work together in many ways, so there are many ways malicious input could sneak in. Any attempt to find them all is likely to miss some. Even if you could find them all today, tomorrow's development efforts may introduce new interactions with new attack surfaces. Finally, sometimes people legitimately want to use contractions, refer to people with apostrophes in their last names, or discuss SQL injection attacks. Removing all apostrophes would hinder those conversations.

Preventing SQL Injection with Prepared Statements

Prepared statements make up the core of our defense against SQL injection. These are sometimes referred to as parameterized queries. For our purposes,

3. https://portswigger.net
4. https://curl.haxx.se/
5. https://httpie.org/

we'll use the terms interchangeably. Prepared statements enforce the separation between templated SQL and user-supplied input. Instead of building up a SQL statement by concatenating strings and user-supplied input, prepared statements are constructed by using a parameterized SQL statement that has placeholder tokens (in most SQL dialects, this placeholder is a ?) and a list of the values that should be used for those parameters. The important difference with prepared statements in our vulnerable example above is that prepared statements *never concatenate the values and the SQL*. The separation is always maintained. Let's see an example in Java. As before, the concept is the same regardless of which language it's written in.

```
public PreparedStatement journalEntrySearch(
  Connection con,
  int personId,
  String wildcard) {

    String sql = "SELECT CreatedTimestamp, Body FROM journal_entries " +
      "WHERE PersonId = ? AND Body LIKE ?"
    PreparedStatement search = con.PrepareStatement(sql);
    search.setInt(1, personId);
    search.setString(2, "%" + wildcard + "%");

    return search;
}
```

With a function like this, even if the attacker tries to use % signs to escape out, they can't because the attacker-controlled wildcard parameter isn't concatenated into the SQL expression. Instead, it's passed to a call to setString, and the database will keep it separated.

When reading code and looking for SQL injection, keep in mind that concatenation can look different in different languages. The examples above used +, but string interpolation can also open the door to SQL injection when it's used with user-supplied data, as in the following example in Python.

```
def generate_sql(person_id, wildcard):
  #This is just as vulnerable as the original
  #Java code even though there's no +
  return "SELECT CreatedTimestamp, Body
FROM journal_entries
WHERE PersonId = {0} AND Body LIKE {1}".format(person_id, wildcard)
```

Correct use of prepared statements should be the preferred way to prevent SQL injection. It's possible to misuse a prepared statement and undo the protection it can bring, however. Suppose we defined journalEntrySearch as follows:

```
public PreparedStatement journalEntrySearch(
  Connection con,
  int personId,
  String wildcard) {
    String sql = "SELECT CreatedTimestamp, Body FROM journal_entries " +
      "WHERE PersonId = ? AND Body LIKE "
    PreparedStatement search =
     con.PrepareStatement(sql + "'%" + wildcard + "%'");
    search.setInt(1, personId);

    return search;
}
```

We can see that even though we're creating a prepared statement, we're using an attacker-controlled piece of data, wildcard, to construct the SQL for the prepared statement. This undoes the protection we hoped to gain. Hopefully a mistake like this would be caught before making it into production. Static analysis tools can be used to catch this kind of mistake during development.

Extending the Defense Beyond Prepared Statements

Prepared statements are great because they're nearly bulletproof. The downside is that not every part of a SQL statement can be parameterized. Table names, for instance, cannot be parameterized. There's no way to write a prepared statement like this:

```
public PreparedStatement journalEntrySearch(
    Connection con,
    String tableName,
    int personId,
    String wildcard) {
  String sql =
    "SELECT CreatedTimestamp, Body from ? WHERE PersonId = ? AND Body LIKE ?";

  PreparedStatement search = con.PrepareStatement(sql);
  search.setString(1, tableName);
  search.setInt(2, personId);
  search.setString(3, "%" + wildcard + "%");

  return search;
}
```

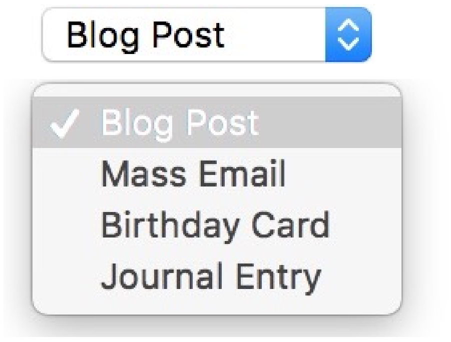
In our journal-keeping example, parameterizing the table name might sound a little silly. There are cases, however, where this level of flexibility would be useful. Suppose our journaling website takes off and we add support for blog posts, mass emails, and on-demand printing of birthday cards. We may find ourselves duplicating the search logic across tables for journal entries, blog posts, mass emails, and birthday cards. (Yes, there are ways to get rid of the duplication, but this is a security book, not a database book, so please indulge me.) If you find yourself in a situation where you can't protect yourself with prepared statements *and* concatenation is the only way to build the query you want, you'll need to check that the data you're concatenating is safe. One way to achieve this is to introduce a level of indirection so that the attacker picks an ID that corresponds to one option in a list of options but the attacker doesn't get to provide the table name itself.

Let's see this approach put to use in a slightly contrived example.

Our database has grown, and now we have BlogPost, MassEmail, and BirthdayCard tables in addition to the original JournalEntry table. All of them have a Body column that we want to search on. We want the user to be able to pick which table to search against using a drop-down list that is generated using a select tag in the HTML of our web page. It might look like this:

```
<select name="table" >
  <option value="BlogPost">Blog Post</option>
  <option value="MassEmail">Mass Email</option>
  <option value="BirthdayCard">Birthday Card</option>
  <option value="JournalEntry">Journal Entry</option>
</select>
```

If you need a refresher on HTML, the value is the literal text that the browser will send to the server if that option is selected. It's surrounded by double quotes in this case. The part between the > and the </option> is what's displayed in the browser. A browser might render this drop-down like this:



One way to make sure that the user-supplied data is legitimate is to maintain a mapping of IDs to table names on the server. This mapping would be used to generate a slightly different drop-down than what we showed before. Instead of having the browser send the server the table name to put into the SQL statement, the browser will send the ID of the table name to put into the SQL statement. This would be done with HTML similar to the following:

```
<select name="table" >
  <option value="1">Blog Post</option>
  <option value="2">Mass Email</option>
  <option value="3">Birthday Card</option>
  <option value="4">Journal Entry</option>
</select>
```

And a server-side mapping of IDs to table names similar to this:

Id	Table Name
1	BlogPost
2	MassEmail
3	BirthdayCard
4	journal_entries

This mapping could be maintained in a dedicated table, or it could be generated dynamically at start-up time and cached in memory. However the mapping is maintained; the server expects input that can be parsed as an integer, not a table name. So when the server parses this it will be readily apparent if it's not valid (either not a number or not a number that maps to anything.) Another benefit of this approach is that table names aren't exposed to the attacker. Obscurity does not provide security, but there's no need to shout our table structures from the rooftops, either. One final benefit to this approach is that any attempt by the attacker to try sending other values will stand out. If the server gets any value for `table` that's not one of the integers from 1 to 4, the server can log that and alert support staff. There's no reason that legitimate users going through the GUI would ever send any value other than 1, 2, 3, or 4. So if the server gets any other value, there is chicanery afoot. We'll see this pattern repeated throughout the book. First priority is to prevent an attack; second priority is to make it "noisy" for an attacker to probe our system.

Layering Additional Defenses as a Mitigation Against Future Mistakes

Proper use of prepared statements is our primary defense against SQL injection. Prepared statements are great, but we have to remember to use them every time we write code that touches SQL; we're never "done" with applying this defense. And if we're building complex, dynamic SQL statements with user input in parts of the SQL that aren't parameterizable, we need to exercise a great deal of caution in many places in the codebase. If we're sloppy in just one of those places, we can wind up leaving the door open to future SQL injection. It would be great if we could complete a one-time task that would protect us throughout future development. Unfortunately, we don't have anything quite that powerful, but proper use of database permissions can get us part of the way there. In theory, we could have a single database user for each table that we want to work with. In practice, this is unlikely to be effective except in very small applications. There are likely to be a large number of tables in an application. And some interactions involve using multiple tables in a single statement. If the number of tables doesn't get you, the number of combinations of tables will.

While it isn't worthwhile to introduce a dedicated database account for every table, it can be worthwhile to introduce them for particularly sensitive tables, such as audit tables or tables that contain passwords. It would be unfortunate if SQL injection in some random part of your application allowed an attacker to change admin passwords or cover their tracks by deleting audit records.

Putting It All Together for a Robust Defense

Adding database permissions to widespread use of stored procedures leaves us with a layered defense that can serve as a model for how we want to defend other parts of our system. We start by defending as much as we can with a nearly bulletproof defense like prepared statements. We then expand the scope of our defense with ongoing diligent development. Finally, we minimize the impact of development mistakes with the one-time application of a broadly effective defense like database permissions. We also set up our system so that attacks will be noisy.

Noisiness here means that attempts to carry out these attacks can be made to stand out. When we build alerting into our system, we can't allow many false positives because that won't scale, will burn out employees, and will lower urgency around responding to alerts. The alerts we've discussed should never happen under well-meaning use of the system, so if we detect these attacks, we have a high-quality indication that an attack is underway. With built-in alerting, the system can notify support staff and possibly take defensive action, such as locking accounts.

This defense requires a lot of ongoing diligence during development. The problem is that diligence is scarce. So if we can't easily increase the amount of diligence we'll be able to bring to bear, let's try to minimize the number of places where we need to use diligence. It's a good idea to introduce some kind of shared framework code to minimize the number of places where diligence is required. Make it easy for application developers to do the right thing and make it clear which parts of the code should access the database and which shouldn't. Don't overlook the importance of examples. Future developers who haven't joined your team yet will draw heavily on the code they've inherited when they write code. Make it easy for them to find good examples in your codebase.

We started the chapter with an explanation of software vulnerabilities by way of a knock-knock joke. Now that we've taken a good look at SQL injection, let's reward ourselves with a software vulnerability joke that's actually funny. Check out Bobby Tables by Randall Munroe.[6]

Cross-Site Scripting (XSS)

We've seen the knock-knock joke principle applied to SQL (SQL injection). Let's take a look at attacks using that same principle when applied to the HTML and JavaScript in a web page. We call this attack cross-site scripting (or XSS for short) if the attack injects JavaScript. We call it DOM injection if it injects regular HTML.

Let's continue with the example from earlier in the chapter of a blogging site. One of the most basic requirements is for anyone using the site to be able to read posts written by other users. Suppose a reader writes a blog post such as this:

> Dear Diary, Today I read the most wonderful book, *Practical Security*.

The reader would expect to be able to see this blog post in their browser. But what if instead of a heartwarming blog post like the one above, an attacker wrote this:

> Dear Diary, <script>alert('Look! A pop-up!")</script>

In a naive web application, the contents of this blog post would be concatenated directly into the HTML that makes up the page. So when another user loads this page, part of the HTML that will be loaded by the browser will include this script tag and the browser will dutifully execute this JavaScript. This means that anyone who can author blog posts can author JavaScript that will execute in the browser of anyone else who visits the page. The example we've seen is harmless. But with a little imagination, we can think of more malicious payloads. Recall that JavaScript has the full ability to interact with all browser UI widgets such as buttons, links, text boxes, and radio buttons. Batching a few of these interactions together means that JavaScript can be written to do anything that the logged-in user can do. This includes things like authoring new blog posts, changing the password of the logged-in user, deleting posts, adding comments to other posts—anything that a logged-in user can do.

6. http://xkcd.com/327/

Dynamic data can also be inserted into HTML element attributes like this:

```
<img src="picture.jpg" alt="This alt text is supplied by the user." />
```

Here we have an img tag with alt text that's supplied by the user. The alt text could be supplied in the query string of the page that loads the img, or it could be read out of the database. In a naive web application, an alt text of this:

```
blah" onload="alert('Hello from alt text!');
```

would turn into this:

```
<img src="picture.jpg" alt="blah" onload="alert('Hello from alt text!');" />
```

Note that this payload contains the opening double quote for the onload attribute, not the closing one. It relies on the double quote that was intended to close the alt text attribute. This keeps the double quotes balanced and results in valid markup.

The most interesting thing about dynamic data in HTML attributes like alt is that it can lead to XSS without using < or > characters. This is another reason that the primary defense against XSS is HTML encoding, not stripping out suspicious characters.

To illustrate how this vulnerability can be exploited, let's look at what would happen with alt text like this:

```
blah" onload="document.getElementById('submitbutton').click();
```

If that were loaded naively into the alt text above, we'd have HTML like this:

```
<img src="picture.jpg" alt="blah"
onload="document.getElementById('submitbutton').click();" />
```

If this were placed into a page with a button with the ID submitbutton, then this JavaScript will click that button when the image loads. From here, you can see how this approach could be extended to script arbitrary interactions with a web page.

For an interesting case study of what XSS can do, consider the case of the Samy worm.[7] Samy Kamkar, the author of the worm, introduced a little bit of JavaScript onto his home page on Myspace. When a logged-in victim visited Samy's page, Samy's JavaScript would execute in the victim's browser. This JavaScript would programmatically click all the buttons that were required to add Samy as a friend and copy itself onto the victim's home page. Then,

7. https://en.wikipedia.org/wiki/Samy_(computer_worm)

when yet another victim visited the first victim's home page, they too would add Samy as a friend and copy the JavaScript onto their home page. This worm quickly went viral and in less than a day more than one million friends had been added to Samy's account.

The beauty of the XSS attack is all the malicious code executes in the victim's web browser. Every click and key press originate from the victim's machine, so network logs and access logs all show traffic from the victim's logged-in machine.

Now let's consider how we can defend against this. A frequently suggested defense that doesn't work is to strip out < and > characters. One problem with this defense is that sometimes people need to discuss dangerous inputs. Readers of this book, for example, may want to discuss XSS payloads on a web-based forum. Attempts to strip out < and > would stop these conversations. Also, we'll see that not every XSS attack needs < or >.

HTML Encoding

Before we look at its application for defense, let's take a look at how HTML encoding works. In the previous paragraph, we touched on an interesting problem in HTML. We use < and > to make HTML tags in our web pages. But what if HTML tags are what we want to talk about in the content of our web pages? At first glance, it would seem that we can't do that because writing about tags would insert tags into our HTML documents and the tags themselves wouldn't be displayed. Fortunately, HTML's authors thought of this and provided a mechanism for allowing discussions of HTML itself in HTML.

Most of the time, the content of an HTML document will consist of literal characters, which get rendered into exactly the characters that make up the source. So HTML markup like this:

```
<div>abcdefg</div>
```

gets rendered like this:

```
abcdefg
```

Each character inside the div gets rendered just as it appears in the source.

But there is another kind of character in HTML called a character reference.[8] Character references are rendered differently than they appear in source. Character references play two roles in HTML. One role is that they allow you to create content in non-Western languages even if you're using a Western

8. https://www.w3.org/TR/html5/syntax.html#character-references

keyboard. The second role is that they allow you to create content that displays key HTML characters like &, <, >, and " when rendered by a browser. This second role is exactly what we need to defend ourselves from HTML injection and XSS attacks.

HTML has two kinds of character references: named character references and numeric character references. All HTML character references start with an ampersand and end with a semicolon. Named character references will have a mnemonic in the middle. Numeric character references will have a unicode code point in the middle. The unicode code point can be represented in either hex or decimal. Named character references only exist for a set of the most commonly used characters. Numeric character references exist for each unicode character.

Any character can be encoded this way. Let's take a look at four examples. In this table, each row shows a rendered character in the leftmost column followed by three different ways of writing the character in the source of an HTML page.

Rendered Character	Named Character	Decimal Numeric Character	Hex Numeric Character
&	&amp;	&#38;	&#x26;
<	&lt;	&#60;	&#x3C;
>	&gt;	&#62;	&#x3E;
”	&quot;	&#34;	&#x22;

HTML Encoding as Defense

Now that we see how HTML encoding works, we can see how we can use this as a defense against HTML injection and XSS. Whenever we're building up HTML as part of our response to a web browser, if we ever concatenate in user-controlled data, we need to HTML-encode it first. That way, even if an attacker tries to sneak JavaScript into one of our responses, we'll encode it first and the browser will just display JavaScript source code to the user instead of executing attacker-controlled JavaScript.

The preferred defense is to use the encoding libraries that come with your web framework. That is, most web frameworks have built-in libraries that will HTML-encode user-supplied data like this:

```
<script>alert('Ha Ha!');</script>
```

into this:

```
`&lt;script&gt;alert('Ha Ha!');&lt;/script&gt;`
```

or this:

```
&#x3C;script&#x3E;alert(&#x27;Ha Ha!&#x27;);&#x3C;/script&#x3E;
```

As with the previous example, the solution here is to use our web framework's HTML encoding library. Proper encoding would result in markup like this:

```
<img src="picture.jpg" alt="blah&#x22;
onload=&#x22;$(&#x27;#submitbutton&#x27;).click();" />
```

The quotes are replaced by `&#x22;` so the onload is just part of the alt text instead of a new attribute. The HTML encoding prevents the attack.

Handling Attacker-Controlled Data in Other Contexts

Sometimes XSS payloads don't look much like textbook XSS payloads if they're built on top of JavaScript frameworks like AngularJS. For more details on Angular-specific attacks, see the excellent article "XSS without HTML: Client-Side Template Injection with AngularJS" by Gareth Heyes.[9] XSS by way of AngularJS expression injection doesn't need < or >, so traditional web framework escaping doesn't help. In general, you shouldn't need to allow dynamic content inside of a dom element that's decorated with the ng-app attribute. But if for some strange reason you do, be sure to encode the {{ and }} so that attackers can't inject an AngularJS expression.

In summary, the way to prevent XSS is to restrict user-controlled data in as few kinds of places as possible in a web page. Keep user-controlled input out of dom elements decorated with the `ng-app` attribute that marks the start of an Angular JS application. And keep user-controlled data out of JavaScript. If you can do this and keep user-controlled data between HTML tags, then you can definitely prevent XSS by making sure to HTML-encode all user-controlled data.

If you really can't get away without including dynamic data in other kinds of places in your markup (such as inside JavaScript,) consult the OWASP XSS prevention cheat sheet.[10] There are a lot of surprising gotchas to allowing dynamic data throughout your markup.

Cross-Site Request Forgery (XSRF)

If XSS is a case of a browser trusting JavaScript from the server too much, XSRF is a case of a server trusting a browser too much.

9. http://blog.portswigger.net/2016/01/xss-without-html-client-side-template.html
10. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Let's go back to our example of a blogging site. Somehow there must be a browser request that saves a blog post to the server. Suppose the blog posting request looks something like this:

```
POST /blog/create HTTP/1.1
Host: www.romansjournalingsite.com
Accept-Encoding: gzip, deflate
Accept: */*
Cookie: sessionid=Re9ljf4uObKk9mSFqBlusxamUKw
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Content-Length: 57

body=It+was+the+best+of+posts.+It+was+the+worst+of+posts.&submit=Publish
```

In a naive web application, that could be all it takes to publish to a hosted blog—a POST request with a logged-in sessionid cookie. Let's see how an attacker or an administrator of an evil website could use this for nefarious purposes.

Suppose I run a malicious website. I ostensibly serve up pictures of adorable kittens playing with yarn. But surreptitiously, I also serve up malicious content like this:

xsrf/kittens.html
```html
<!DOCTYPE html>
<html lang="en">
<body>
  <form action="http://romansjournalingsite.com/post/create" method="POST">
    <input
      name=body
      value="Arbitrary Attacker-Controlled Content. I love evilxsrf.com"/>
    <input type=submit id=submit name=submit value=Publish />');
  </form>

  <script>
   document.getElementById('submit').click();
  </script>
</body>
```

What does this do? It creates a form with the action we just saw when we looked at the romansjournalingsite.com request that creates a new blog post. Additionally, the form is prepopulated with content that will create a blog post that says Arbitrary Attacker-Controlled Content. I love evilxsrf.com. Finally, it has JavaScript that automatically submits this form as soon as the page is loaded. This payload will cause a modern browser to make a POST request that looks like this:

```
POST /post/create HTTP/1.1
Host: romansjournalingsite.com
Content-Length: 74
Cache-Control: max-age=0
Origin: http://evilxsrf.com
Content-Type: application/x-www-form-urlencoded
Referer: http://evilxsrf.com/kittens.html
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: sessionid=1234567890abcdef
Connection: close

body=Arbitrary+Attacker-Controlled+Content.+I+love+evilxsrf.com&submit=Publish
```

This looks just like the legitimate request! As far as romansjournalingsite.com is concerned, it *is* a legitimate request: it has a valid sessionid cookie. The admins of evilxsrf.com can use this to control the romansjournalingsite.com account of anyone who's logged in to romansjournalingsite.com and visits evilxsrf.com.

Just like we saw with XSS requests, this attack forges legitimate requests. They have a legitimate session ID, so the server will treat it as a legitimate request. The POST request wasn't generated because the user wanted to make that post, but the blog site can't tell that. The browser's same-origin policy (SOP) won't help here either. SOP only says that JavaScript from one site can't see responses sent back from other sites. But this attack doesn't require the JavaScript from the malicious site to see a response from the blogging site. This attack only requires that the JavaScript from the malicious site be able to POST a request to the blog site, which it can.

Romansjournalingsite.com needs to differentiate valid requests intentionally made by legitimite users from those that were made by malicious websites. Romansjournalingsite.com can't rely on session IDs or cookies, as we've seen. Romansjournalingsite.com needs to submit a little bit of secret data with every state-modifying request that only romansjournalingsite.com knows. This secret can't be stored in a cookie, though, because cookie values will be sent regardless of whether the request was initiated by a logged-in user or a malicious site operator. This leaves the form itself.

The defense against this is the XSRF hidden form input. When a user logs into the blog site, the blog site should set a large (say 128 bits, base64–encoded) cookie valid only for the duration of the session. Every page that contains a form that will POST back to the blog server needs to put that same value into a hidden form input. Only the blog site and the browser have this value. So when the user submits a valid POST to the blog site, this request will contain the anti-XSRF value, like this:

```
POST / HTTP/1.1
Host: localhost:1234
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: antixsrf=XKRYopsd8jXj5DqgfNpHmA
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Content-Length: 94

body=It+was+the+best+of+posts.+It+was+the+worst+of+posts.&submit=Publish&
antixsrf=XKRYopsd8jXj5DqgfNpHmA
```

So the form contains a hidden input whose value matches the xsrf cookie. If this form had been constructed by a malicious third-party website, the request would have looked like this:

```
POST / HTTP/1.1
Host: localhost:1234
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: antixsrf=XKRYopsd8jXj5DqgfNpHmA
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Content-Length: 72

body=It+was+the+best+of+posts.+It+was+the+worst+of+posts.&submit=Publish
```

That is, the POST request would have had the anti-XSRF cookie, because all requests to the blog site will contain the cookie. But the malicious website wouldn't be able to guess the value of the anti-XSRF cookie, and so it would not be able to replicate that value in the body of the request. If the server doesn't see the value in both places, the server can reject the request as a fake. At a minimum, requests like this should be denied and logged for later review.

Most modern web frameworks have defenses against XSRF. They may require additional development effort, however. So be sure to learn what your web development framework provides and use it on all state-modifying requests in your application. For example, there is good documentation on built-in XSRF defenses for ASP.NET,[11] Django,[12] and Ruby on Rails.[13]

11. https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages
12. https://docs.djangoproject.com/en/2.0/ref/csrf/
13. http://guides.rubyonrails.org/security.html

There’s an important caveat to XSRF defense. If the page is vulnerable to cross-site scripting (XSS), then the XSRF defenses will be bypassable. So it’s important to make sure that our web applications are not vulnerable to XSS. To see why XSS can bypass XSRF defenses, let’s think back to how XSRF defenses work. They add a little bit of secret information that wouldn’t be exposed or sendable from other websites. Typically, this is done with a hidden form input. If there’s XSS on a page with a hidden form input, malicious JavaScript can be injected into the page to send a request with the XSRF defense value.

XSRF Prevention with SameSite

We now have a very strong defense against XSRF—using an anti-XSRF hidden form input on all state-modifying requests. But that defense requires ongoing diligence. We’re never done applying it. We need to reapply this defense every time we add a new state-modifying request to our web application (which will happen pretty often during active development of a web application). It would be nice if we could layer on a one-time effort to help lessen the impact if we ever forget to be diligent in the future. That is the idea behind SameSite cookies.[14] Let’s take a look at this defense, how it helps, and what its limitations are.

Suppose we are building a web application that uses a cookie called SessionId to authenticate logged-in users. Normally, this cookie would be created by an HTTP response that includes a Set-Cookie header like this:

```
Set-Cookie: SessionId=sfVZ1yx68LD51I;
```

As we saw in the previous section on XSRF, if this cookie is the session cookie for our web application, it would then be sent on every request to our application, regardless of what site originated the request. Wouldn’t it be nice if we could tell browsers to only send that cookie for requests that originated from our site? That’s the idea behind SameSite.

With SameSite, the part of the response that sets the cookie would look like:

```
Set-Cookie: SessionId=sfVZ1yx68LD51I; SameSite=Strict;
```

or

```
Set-Cookie: SessionId=sfVZ1yx68LD51I; SameSite=Lax;
```

14. https://tools.ietf.org/html/draft-west-first-party-cookies-07

To understand this defense, we first need to understand a little bit of HTTP trivia. The HTTP specification defines "safe" and "unsafe" requests in Section 4.2.1 of RFC7231.[15] Safe requests use the GET, HEAD, OPTIONS, or TRACE methods. Unsafe requests use any of the other HTTP methods, including, most notably, POST requests. This distinction is there because the safe methods shouldn't change state on the server; only the unsafe ones should be used to modify state. It's the unsafe ones that we're concerned with when preventing XSRF.

Adding SameSite=Strict to a cookie definition tells a browser to never send that cookie for any request to our site unless the request originated from our site. That sounds good in practice, but it's often not what we want. With SameSite=Strict, other sites won't be able to link successfully to our web application because the initial request to our web application won't include the SessionId cookie since it originated from another site. If a user clicks on a link from another website to our web application, that first request would not include the SessionId cookie, so the user would probably be prompted to log in, even if they had already logged in. If you know that you don't need to support other sites linking to your web application, this would be an appropriate choice.

More often, what you want is to set SameSite=Lax. By doing this, all safe requests will send the cookie, even if the request originates from another site. This way, the initial link from an external site to our web application will send the SessionId cookie since clicking on the link will result in a GET request. But a malicious site that wants to exploit XSRF by constructing a form and getting a user to submit a POST request would fail because the SameSite attribute on the SessionId cookie wouldn't get sent because the request originated on another site.

So the addition of the SameSite attribute on our session cookie raises the bar for attackers in the event that we forget to implement XSRF defense in a new page in the future. Instead of being able to exploit the vulnerability by merely getting a logged-in user to browse to a website that the attacker controls, the attacker would need to find a DOM injection or XSS vulnerability in our web application in order to exploit the lack of XSRF defense.

Pretty cool. Now what are the limitations of this defense?

In describing the benefits of SameSite, we touched on the first limitation. SameSite doesn't protect you if your site is vulnerable to DOM injection or

15. https://tools.ietf.org/html/rfc7231#section-4.2.1

XSS. Put another way, if your site is dynamic enough to allow an attacker to submit HTML to be viewed by other users, then SameSite won't defend against XSRF attacks launched from within your own web application because all of the HTML and JavaScript is coming from the same site. It would still stop XSRF attacks initiated from other sites.

A second limitation is that the SameSite session cookie defense is dependent on support in the browser. Fortunately, it is widely supported.[16]

Finally, SameSite doesn't protect you if you allow safe methods to modify state. So if your website creates, updates, or deletes objects in your web application via safe methods, then SameSite will not protect you if you forget XSRF defenses.

Misconfiguration

> *Never attribute to malice that which is adequately explained by misconfiguration.*
> —Zabicki's Razor (with apologies to Hanlon)

Attackers are opportunistic. They won't bother with a sophisticated attack where a simple one will do, and seeking out and exploiting misconfigured systems is one of the simplest attacks there is.

We need to develop the capabilities for ongoing monitoring of our systems to make sure we haven't made the kinds of configuration mistakes that will open the door for easy attacks. The specifics of how you do this will vary significantly depending on exactly which technologies you use in your organization. We'll take a look at some of the most common misconfigurations and some tools to detect them. Even if you don't use these specific tools, these examples should give you an idea of the kinds of mistakes you'll want to be able to find.

Open S3 Buckets

Amazon[17] offers a popular storage service called Simple Storage Service, or S3 for short.[18] S3 is a large-scale key-value storage service that lets users store file-like "objects" inside of "buckets." A bucket can hold an arbitrary number of objects and an object can range in size up to 5TB. Behind the scenes, S3 is a highly durable storage service that automatically distributes data across multiple physical facilities. Amazon offers a lot of tools as well, including tools for big data analysis that integrate natively with S3.

16. https://caniuse.com/#search=samesite
17. https://aws.amazon.com
18. https://aws.amazon.com/s3/

It’s really neat. It also seems to be really easy to misconfigure.

A quick Google search for “S3 breach” will show many high-profile instances of misconfigured S3 buckets that left sensitive data open for the world to see. No need for fancy attacks or cryptographic breakthroughs if the data isn’t protected in the first place.

One particularly easy S3 mistake to make involves something called the Authenticated Users group. AWS permissions are based on group membership. So when setting up permissions, an administrator will typically create groups that represent the organization and assign permissions to those groups. The Authenticated Users group is a predefined group in AWS. It would be easy to look at the name and think that it describes the group of people that are authenticated users of one’s own organization. That is not what that group means, however. Anyone who is logged into AWS as a part of any organization is automatically a member of the Authenticated Users group. If we look at the relevant documentation we read this:[19]

> When you grant access to the Authenticated Users group, any AWS-authenticated user in the world can access your resource.

And just below that, we see another predefined group called the All Users group. Amazon’s documentation has this to say about the All Users group:

> Access permission to this group allows anyone in the world access to the resource. The requests can be signed (authenticated) or unsigned (anonymous).

So if you give the Authenticated Users group read access to your S3 buckets, you are giving read access to everyone in the world who has an AWS account. And any access you give to the All Users group is access you are also giving to everyone in the world, regardless of whether they have an AWS account or not.

The Authenticated Users and All Users group misconfigurations are a great example of the kind of misconfiguration we need to be able to detect. It’s easy to see how they could be misused. It’s easy to see the impact this could have. It’s easy to see how their misuse could happen at any point in the life of an online system. It’s easy to see how attackers could automate detection of this kind of misconfiguration and find this flaw in your system, even if they never had any reason to seek you out specifically.

A problem like this calls for automation. One tool that can help with finding misconfigurations like this is Scout2.[20] Scout2 is an open source tool designed

19. https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html
20. https://github.com/nccgroup/Scout2

to look for a wide range of AWS misconfigurations, not just overly permissive S3 buckets. Installation and usage is outlined on the GitHub page and is fairly straightforward. Scout2 works by using AWS credentials that you provide to query the extensive AWS APIs to find common misconfigurations. It then takes the results of these queries and creates a report that summarizes its findings. At the time of this writing, it's still under active development with new misconfiguration searches being added periodically.

It only takes a couple of minutes to install and run Scout2. So if you use AWS, it's probably worthwhile to run it right now and see if you have anything pressing to fix before continuing on with this chapter.

I'll wait.

Now that you've run Scout2, it's worthwhile to budget some time to automate the regular generation of a Scout2 report. Even if everything is locked down perfectly today, mistakes could be introduced tomorrow. And if bad things happen in the future, it can be helpful to look back on a record of when things changed.

Default Passwords

Default passwords are another kind of misconfiguration that saves attackers a lot of time and effort. They're easy to exploit and easy to detect—just the kind of thing that attackers love. So we need to find them first. We can leverage the network inventory work we did in chapter 1 to give us a starting point for where to look. We'll also want to include network infrastructure like switches. We'll want to pay particular attention to anything that's exposed to the internet.

As was the case with defenses against SQL injection, our defense against this kind of misconfiguration can be layered. The first layer of the defense is to add to our provisioning checklist to make sure to not use default passwords when provisioning new services. Beyond that, we can look into scanning our network for default passwords. This second layer is highly specific to your network. You won't have time to exhaustively scan everything on your network. You'll need to use your judgment on where to focus your efforts. You may get a good return on looking into crusty old infrastructure that doesn't have clear ownership. And don't overlook networked printers. Networked printers can have capabilities like emailing or connecting to an Active Directory server. If you can get administrative access to a printer by using default credentials, you may be able to see the email or Active Directory credentials that enable those capabilities.

Care in avoiding default passwords can open the door for helpful monitoring as well. If possible, alert on failed login attempts that tried using default credentials. Once you've configured your system with new, nondefault credentials, there will never be a legitimate login attempt that uses the default credentials. If you see that someone has attempted a login with default credentials, you have a high-quality signal that an attack is under way.

Credentials

Checking credentials into a public GitHub repo is a common mistake. In the eyes of an attacker, leaked credentials are just as good as default credentials.

Even if we only use private source control servers, we still don't want to check credentials into source control. We don't want to have to do a build in order to change credentials. Also, putting credentials into source control makes it hard to introduce tiers of access. For instance, you may not want junior team members or third-party contractors to be able to see or change production credentials. Some organizational models call for a separation between those who write code and those who have access to run or deploy that code in production.

If we're agreed that keeping passwords out of source control is a good idea, it's worthwhile to have an automated way to enforce this in case we forget. If you have sensitive data confined to a single configuration file, you may be able to use features of your source control system to keep that file from ever getting checked in, even accidentally. Many source control systems can be configured to not track specific files. If you're using Git, you can add your sensitive configuration files to your .gitignore file.[21] This will keep you and your team from being able to check in the sensitive files at all. Other source control systems may offer similar functionality as well.

It's worthwhile to periodically look through our source code for credentials that have been checked in. This is especially important on a larger team. It's easy for decisions like not checking credentials into source control to not be communicated to the entire team, especially as the team grows over time.

The first tool you can use is just plain grep. It's worth doing a one-time manual search for words like the following:

- password
- cred
- token

```
grep -Ri password *
```

21. https://git-scm.com/docs/gitignore

These three words are worth scanning for, but you might get a lot of false positives. If you need to whittle down a lot of matches, you may want to filter these down. One way would be to look for occurrences that look like assignment statements in the programming language you use.

```
grep -Ri password * | grep '='
```

or

```
grep -Ri password * | grep ':' # if you use a lot of yaml or json
```

These next search patterns are very unlikely to have false positives. These are the beginnings of standard ssh private key files.

—–BEGIN RSA PRIVATE KEY—–

—–BEGIN OPENSSH PRIVATE KEY—–

—–BEGIN DSA PRIVATE KEY—–

—–BEGIN EC PRIVATE KEY—–

There are a number of tools that take this concept a little further. One example is TruffleHog.[22] One of the nice things about TruffleHog is that it understands Git. So point it at your Git repo and it will look for checked-in secrets on any check-in on any branch. The benefit is that it can find secrets even if they aren't in the latest branch. This catches the scenario where a developer accidentally checks in a password, realizes what they've done, then deletes the password on the next check-in. It's not enough to remove it from the latest branch, because as TruffleHog shows, an attacker with access to Git can go back through the check-in history and look for passwords that used to be checked in. TruffleHog has two modes of operation. One uses a configurable list of regex patterns, including the ssh private key patterns we looked at earlier. The second mode involves looking for high-entropy strings that "look" like checked-in certificates. The regex searches are faster and can be appropriate for adding into your build process.

Jenkins

If we use Jenkins,[23] we need to keep it patched, as we discussed back in Chapter 1, *Patching*, on page 1. But Jenkins has a common misconfiguration that merits special mention. Jenkins instances are often started with insecure settings that allow for unauthenticated execution of commands in a scripting

22. https://github.com/dxa4481/truffleHog
23. https://jenkins.io

language called Groovy.[24] Groovy scripts can execute arbitrary shell commands. So a common attack is to scan the network for misconfigured Jenkins servers, use the Groovy Scripting Console to dump passwords from the Jenkins server, then use those passwords to compromise other servers on the network. So make sure to lock down Jenkins so that it requires a login before allowing any of its functionality, especially the Groovy Scripting Console.

Public-Facing Servers

We're going to look at one last source of vulnerabilities in this chapter—long-forgotten public-facing servers. It's easy to forget to shut down public-facing servers that aren't used anymore. This mistake is even easier to make if you use a cloud-hosting service. We'll address the problem of forgotten or unpatched servers exposed to the internet similarly to the way we've addressed other vulnerabilities in this chapter. First, we'll do a one-time cleanup effort. Once we've addressed the problems of today, we'll add automation to make sure we don't reintroduce this problem again in the future.

Ideally, before you kick off a one-time cleanup effort, you already know exactly what servers you have exposed to the internet. Whether that's the case or not, it's worthwhile to examine your organization using a public tool. This can either serve as a first census or a double-check on your existing practices around maintaining an up-to-date inventory of your public-facing servers. The first time you do a check like this can be pretty eye-opening. You may be surprised to see how many public servers you actually maintain. You may also be surprised about how up-to-date the software running on those servers is. If these scans reveal version numbers of server-side software, be sure to google for CVEs for that software. We covered CVEs in *What Is a CVE?*, on page 3.

Two tools that are great for this are Shodan[25] and Censys.[26] Both Shodan and Censys continually scan the full IPv4 address space and provide queryable access to data about the servers they discover on the internet.

See what your organization has exposed to the internet. Hopefully there are no surprises there. Clean things up if there are. Then decide on a way to check automatically going forward. The amount of automation you will want to build out will be highly specific to your organization. If you only have a couple servers, maybe you can just manually look at your hosting service's

24. http://groovy-lang.org/
25. https://shodan.io
26. https://censys.io/

dashboard and eyeball it periodically. If you have a larger footprint or a larger organization with lots of people who can provision new servers, you'll probably want to put more effort into automating scans.

Suggested Reading

We've covered just some of the basics here. If you'd like to dig in deeper, I recommend reading the following:

- *The Art of Software Security Assessment [DMS06]* by Mark Dowd, John McDonald, and Justin Schuh
  - Read this for a more detailed look at a wide variety of coding mistakes that make software vulnerable to attack and their defenses.
- *The Web App Hacker's Handbook (2nd Edition) [SP11]* by Dafydd Stuttard and Marcus Pinto
  - This provides a more detailed look at web-specific attacks and their defenses.
- *The Hacker's Playbook (3rd Edition) [Kim18]* by Peter Kim
  - Kim gives you more ideas about where in your office network to look for vulnerabilities.
- "The Basics of Web Application Security" by Martin Fowler[27]
  - This article is a concise, general guide on how to write secure web-based software.
- Pushing Left, Like a Boss[28]
  - A great series of blog posts on how to adopt strong security practices earlier in the development cycle.
- DataSploit[29]
  - If you want to dig into what information is publicly available about the domains that your organization controls, take a look at DataSploit. This is an open source tool that queries a number of public repositories of information, including Shodan and Censys. It's designed to be used manually, but it also works nicely when called automatically. Scheduling it to run automatically on a regular basis and saving the output can give you a picture of how your public footprint has changed over time.

27. https://martinfowler.com/articles/web-security-basics.html
28. https://code.likeagirl.io/pushing-left-like-a-boss-part-1-80f1f007da95
29. https://github.com/DataSploit/datasploit

What’s Next?

As we look back on the vulnerabilities we covered in this chapter, we see two main classes of vulnerabilities. In the first, an attacker is able to inject code of their own choosing into the system. In the second, operators accidentally leave the system in an insecure state. Interestingly, the defense for both looks fairly similar. First we make a one-time effort to find the vulnerabilities and fix them. We then layer on automated defenses to prevent mistakes from reintroducing the vulnerability. As teams and systems grow larger and older, we want to have more than vigilance keeping us from introducing vulnerabilities into the system; we want the system to prevent vulnerabilities from being introduced.

In our next chapter, we’ll take a look at how we can use cryptography to secure the systems we build. We’ll also see how seemingly small mistakes can let an attacker break weak cryptography. Just like a seemingly small flaw in our SQL allowed an attacker to bypass permission checks earlier in this chapter, seemingly small flaws in cryptography can leave our systems unprotected.