

University of Arkansas, Fayetteville

ScholarWorks@UARK

---

Computer Science and Computer Engineering  
Undergraduate Honors Theses

Computer Science and Computer Engineering

---

5-2022

## Demonstration of Cyberattacks and Mitigation of Vulnerabilities in a Webserver Interface for a Cybersecure Power Router

Benjamin Allen

Follow this and additional works at: <https://scholarworks.uark.edu/csceuht>



Part of the [Information Security Commons](#), and the [Programming Languages and Compilers Commons](#)

---

### Citation

Allen, B. (2022). Demonstration of Cyberattacks and Mitigation of Vulnerabilities in a Webserver Interface for a Cybersecure Power Router. *Computer Science and Computer Engineering Undergraduate Honors Theses* Retrieved from <https://scholarworks.uark.edu/csceuht/101>

This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact [scholar@uark.edu](mailto:scholar@uark.edu).

Demonstration of Cyberattacks and Mitigation of Vulnerabilities in a Webserver Interface for a  
Cybersecure Power Router

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering  
College of Engineering  
University of Arkansas  
Fayetteville, AR  
April, 2022

by

Benjamin E Allen

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Scope . . . . .	6
<b>2</b>	<b>Features and Development</b>	<b>6</b>
2.1	User Accounts . . . . .	7
2.2	Devices, Registers, and Values . . . . .	8
2.3	Device View and Memory View . . . . .	10
2.4	Frameworks and Libraries . . . . .	11
2.4.1	Django [7] . . . . .	11
2.4.2	SQL . . . . .	12
2.4.3	MinimalModbus [9] . . . . .	12
2.4.4	Django Two-Factor Authentication [10] . . . . .	13
2.4.5	Chart.js [11] . . . . .	13
<b>3</b>	<b>Attacks &amp; Mitigations</b>	<b>13</b>
3.1	Supply Chain Attack . . . . .	13
3.2	SQL Injection . . . . .	15
3.2.1	Sample attacks . . . . .	18
3.3	Cross-Site Scripting . . . . .	21
3.3.1	Sample attacks . . . . .	22
<b>4</b>	<b>General Security</b>	<b>24</b>
4.1	Multi-Factor Authentication . . . . .	24
4.2	Man-in-the-Middle . . . . .	25
<b>5</b>	<b>Conclusion</b>	<b>26</b>

# List of Figures

1	The login screen requires both a username and password. After entering the correct password the user is required to enter a 2FA code. . . . .	8
2	When using Django’s code-first database modeling, the framework is able to create an administration page that allows admin users to create and update database objects without knowledge of SQL. . . . .	9
3	The “data view” page, which allows users to view data received from the physical Modbus device. The chart in this image is empty since there was no physical device connected to communicate with. . . . .	10
4	The “memory view” page, which allows users to view data received from the physical Modbus device by viewing contiguous memory regions reported by the device. Here a sine wave was produced in hardware emulation and its data placed into memory for viewing. . . . .	11
5	Entering the SQL Injection payload. . . . .	19
6	The debugger paused at the point of receiving the POST request from the client. Note that the username field shown at the bottom of the screen matches the form, indicating that the server has received the input as written and mitigation of the attack must be server-side. . . . .	19
7	Verifying that the auth_user table still exists . . . . .	20
8	The XSS attack has failed, as the tag text is still visible and has not been parsed as HTML. . . . .	24

# 1 Introduction

## 1.1 Motivation

Grid-connected devices are increasingly outfitted with internet connectivity for the purpose of improved management of the United States’ electrical infrastructure. Colloquially known as the smart grid, the upgrade aims “to increase energy efficiency, reliability, and security; to transition to renewable sources of energy; to reduce greenhouse gas emissions; and to build a sustainable economy” [1]. However, these advanced networking capabilities come with costs, and the cost of network connectivity is exposure to cyberattacks.

Attacks on infrastructure are frequent. For recent examples, see the 2021 closure of the Colonial Pipeline by ransomware attackers [2], the 2015 cyberattack on Ukraine’s power grid which triggered a blackout [3], or the attempt to poison the water supply of Oldsmar, FL [4]. The attack in [2] halted the flow of refined gasoline and triggered a wave of panic buying until the flow was restored. The attack in Oldsmar was enabled by remote control systems installed on employee computers at the city’s water treatment plant. In the 2015 attack on the Ukraine power grid, the attack focused on compromising the industrial control systems and was initiated by a “phishing” campaign that relied on social engineering to gain access. These attacks are made possible by the networked infrastructure that is increasingly being used to manage the power grid; it follows that the security of this networked infrastructure is highly important.

Defense-in-depth is a critical strategy of building proper cybersecure systems. This approach works to limit the impact of security failures by ensuring that multiple layers of security features are in place to catch attacks. A system utilizing defense-in-depth does not rely on one security component to block all attacks, but overlapping components that can redundantly block attacks, mitigate attacks, or warn of potential breaches. Since many systems are themselves layered, defense-in-depth advises proper defenses at each layer, as well as building

defenses upon people, process, and technology [1] since individual elements can be defeated.

## 1.2 Scope

This undergraduate honors project is part of a defense-in-depth approach to designing a designing a cybersecure power router (CSPR) for use in smart grid infrastructure. In addition to the defenses built into the hardware itself, as well as that hardware's control system, a cybersecure management interface was necessary.

The task of this project was to build a webserver interface for interfacing with these CSPR devices and then subject that webserver to cyberattacks and verify its defenses against them. The different features of the webserver are discussed in section 2. These attacks make up the bulk of the project's effort, and are described in detail in section 3. SQL injection attacks, cross-site-scripting attacks, supply chain attacks, and man-in-the-middle attacks are all considered. Additionally, general security practices such as password requirements that work to avoid weak passwords and multi-factor authentication support are built into the webserver; these are discussed in section 4.

## 2 Features and Development

Developing the webserver was one of the tasks undertaken as part of this project, but the primary task was still the security testing performed against the web application. The purpose of the webserver is to act as a management interface for smart-grid electronics communicating over the Modbus protocol. It can be used to monitor smart-grid devices in real-time. The administrator of the web application sets up particular Modbus registers (memory locations) to be monitored, and while the user is on that device's page the values in those Modbus registers will be read multiple times per minute. The server also includes a form to allow users to manage the smart-grid devices by updating values in Modbus registers. The webserver code can be found on the University of Arkansas GitLab instance, at the URL

provided in [6]. The security features mentioned here are discussed at more length in sections 3 and 4.

## **2.1 User Accounts**

The webserver supports user account creation with two-factor authentication security. User accounts are also subject to password requirements that disallow short passwords, passwords that are too repetitive, and passwords that match a list of common passwords maintained by Django. These follow the best practices outlined by the United States Cybersecurity & Infrastructure Security Agency [5]. After creation, user accounts are required to add a second factor. For development purposes this prints out access codes to the developer console, but integration with SMS-sending services is supported by the plugin.

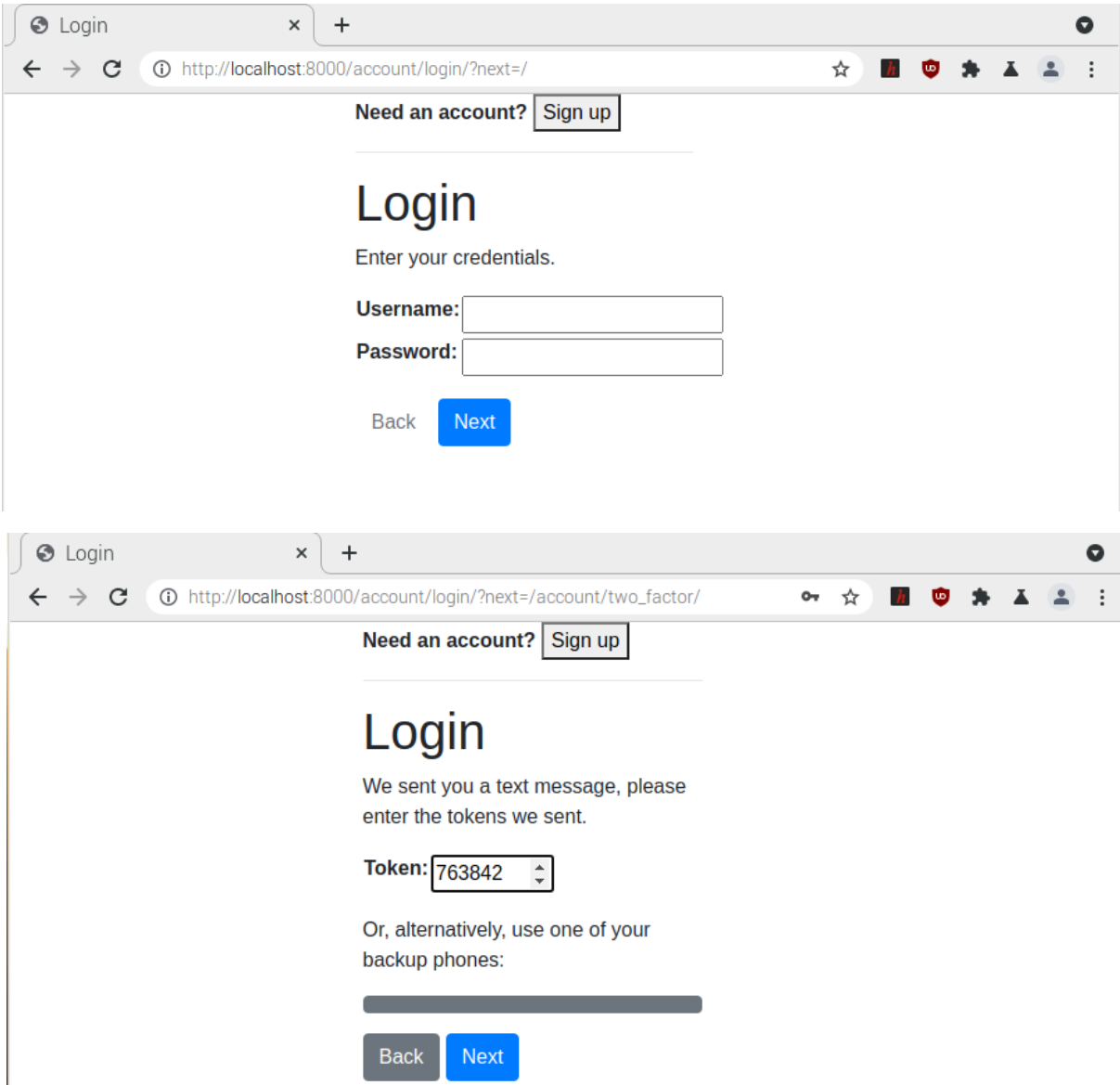


Figure 1: The login screen requires both a username and password. After entering the correct password the user is required to enter a 2FA code.

## 2.2 Devices, Registers, and Values

“Modbus devices” are physical hardware that the server communicates with over a serial connection via the Modbus RTU protocol. “Registers” are the memory locations that the webserver users are allowed to modify. “Values” are the memory locations that the webserver is allowed to query and display in its dynamic chart.



Users that are given administrative privileges can manage the Modbus devices, registers, and values that are defined in the database using the Django-provided admin panel. From here, administrators can create devices, add registers and values, or edit the memory locations referenced by these registers and values. The variables for device creation primarily relate to how the Modbus communication library should be configured to communicate with the physical hardware; some examples are the baud rate for communication and the number of stop bits to be expected. Registers and values have data for the memory address in question, as well as whether the data should be understood as an integer or a fixed-point decimal value, and, if so, how many decimal places it should have. All of this data is stored in a SQL database.

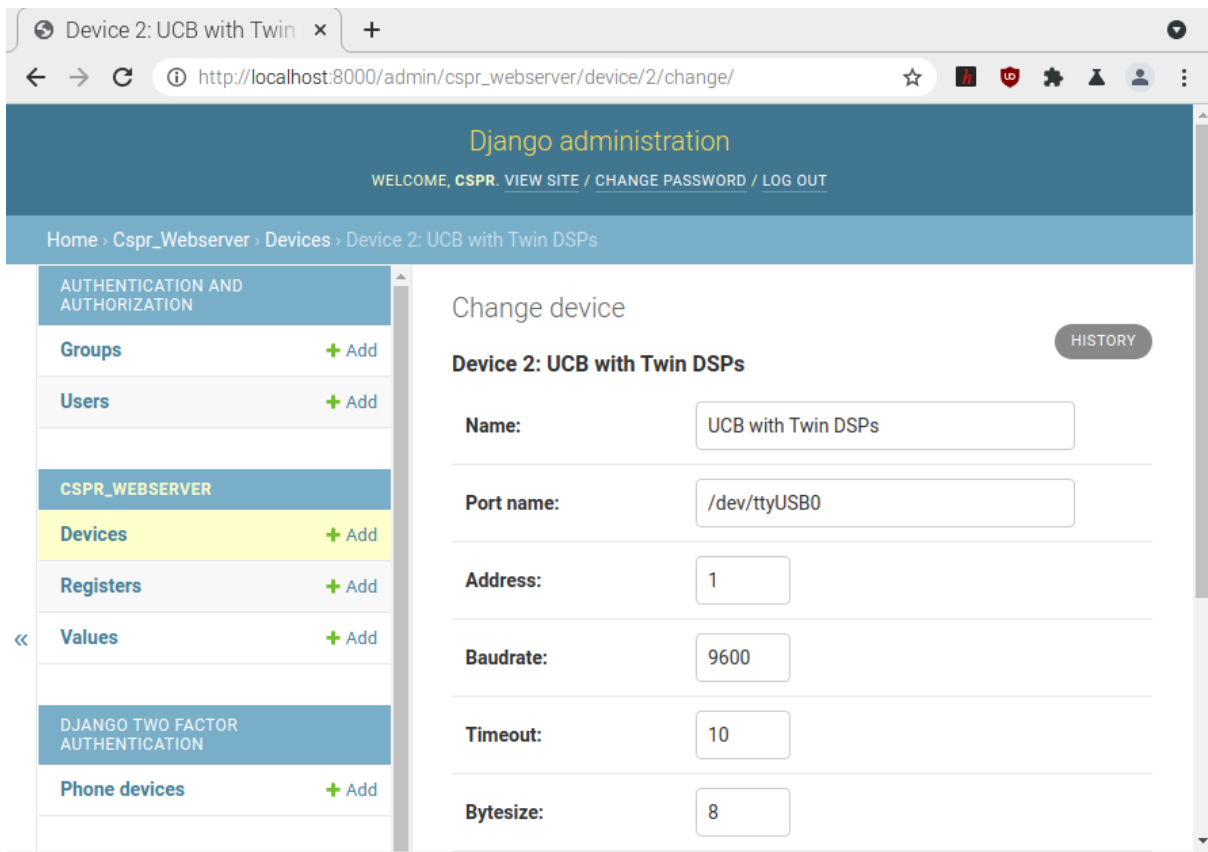


Figure 2: When using Django’s code-first database modeling, the framework is able to create an administration page that allows admin users to create and update database objects without knowledge of SQL.

## 2.3 Device View and Memory View

The server exposes a view for each device. When a user is routed to the view, the server initiates Modbus communication with the device and begins to fetch data for the Values. A form is generated which allows the user to input values for Registers and submit write commands to the Modbus device. A list of the values defined for that device is exposed, with buttons to add or remove them from the chart at the bottom of the page. The page runs a loop that asynchronously queries the server for updated readings from the Modbus device; when data is received it is stored in memory on the client’s machine. When the user clicks a button to add a value to the chart, a time series of that value’s data for each reading taken since the user arrived on the page is added to the chart, and the chart rescaled to fit that data.

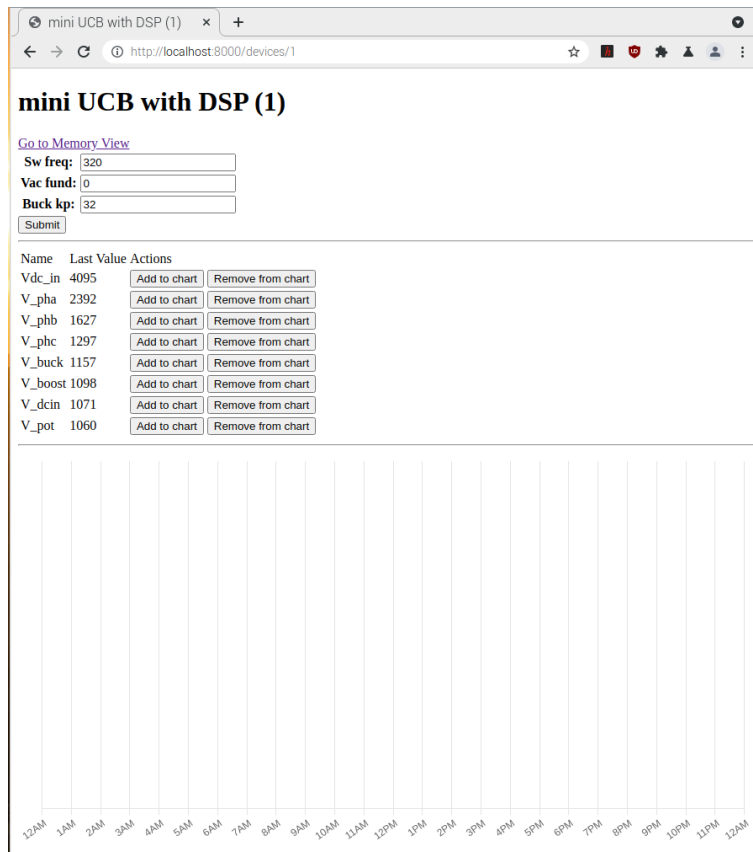


Figure 3: The “data view” page, which allows users to view data received from the physical Modbus device. The chart in this image is empty since there was no physical device connected to communicate with.

There is an alternate view, called the “memory view”. In this view the user can specify a memory address and a number of registers to read from the Modbus device. The data read in this way will be plotted on the chart at the bottom of the page. This was used in demonstrative sessions such as the one pictured in figure 4, where a sine waveform generated by emulation on the hardware was performed.

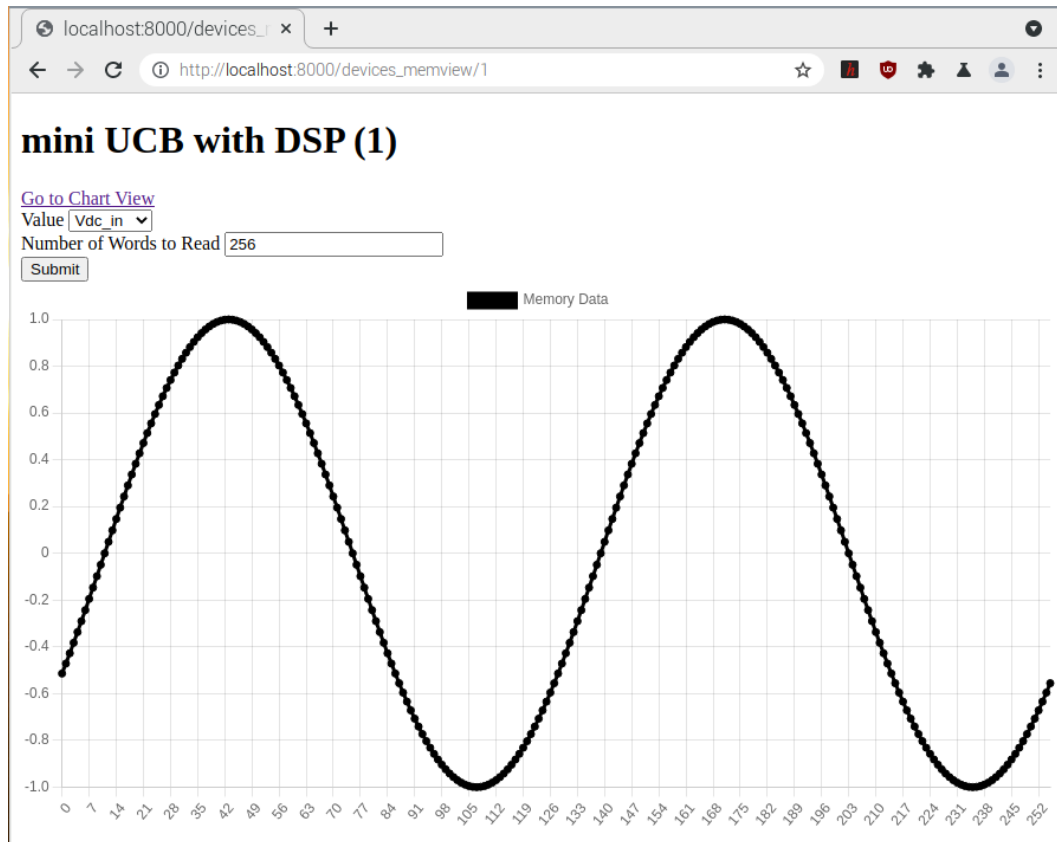


Figure 4: The “memory view” page, which allows users to view data received from the physical Modbus device by viewing contiguous memory regions reported by the device. Here a sine wave was produced in hardware emulation and its data placed into memory for viewing.

## 2.4 Frameworks and Libraries

### 2.4.1 Django [7]

Django is a framework for developing web applications in Python. It focuses on rapid execution of app ideas and adheres to Python’s “batteries included” philosophy. Django

provided the webservice capabilities and routing systems, allowing my development work to be limited to application logic, views, and communication with Modbus devices. Django also provided a suite of secure-by-default subsystems that were critical to implementing this server in a cyberattack-resistant manner. These secure defaults are discussed more in section 3.

### **2.4.2 SQL**

A SQL server is used to store persistent data about configured devices. Interface to the server is provided by Django's object-relational model (ORM), which enables code-first definitions of database constructs and allows for easy upgrades via migrations. Models are first defined as Python classes, provided with database-specific attributes (data type, primary key, etc.), then Django uses this information to build or update tables within the SQL database. New rows, updates, or deletions are all passed through the ORM, enabling Django's secure defaults to protect the database from SQL injection attacks. While the server is database-agnostic (requiring only that a Django plugin exists to support your database provider of choice), the development version of the website used MariaDB as the database provider [8].

### **2.4.3 MinimalModbus [9]**

MinimalModbus is a python library that facilitates communication with electronic devices implementing the Modbus standard. The library manages the construction of modbus data packets and the serial communication necessary to transmit requests and responses. Once properly configured, use of this library enabled me to focus on the high-level logic of Modbus communication. Modbus registers are read from and written to synchronously, which integrates well with Django's default setup for synchronous communication.

#### **2.4.4 Django Two-Factor Authentication [10]**

This library adds two-factor authentication support to the Django user accounts module. Two-factor authentication is discussed in more detail in section 4. In this project a second factor is required to log in once the user has enabled two-factor authentication, and each view exposed by the server (excepting the login view and the view to set up the second factor) require the user to have logged in with a two-factor protected account.

#### **2.4.5 Chart.js [11]**

Chart.js is a javascript module that enables the production of dynamic charts on webpages. The library provides a configurable HTML5 canvas element that is used in this project to generate a chart of data received from Modbus communications with a device. These updates are performed in real time. The logic to upkeep the dataset was implemented as part of this project, but chart.js handles all features related to display.

## **3 Attacks & Mitigations**

### **3.1 Supply Chain Attack**

Instead of directly targeting the application to be compromised, the attacker in a supply chain attack compromises some part of that application's dependencies. This definition is broad, and attacks that are classified under it include

- A hacker compromises an application provided to internet users for download. The compromised application downloads or executes malware upon its installation, but the original creator of the application does not know this.
- A rogue employee compromises the software produced by their company to extract sensitive information from one of that company's clients. The compromised software is a dependency of the real target and is not the target itself.

- An attacker compromises an open-source library that is a dependency of several larger projects.

With respect to the CSPR webserver, our concern is with the last of these. The project utilizes the Django framework as well as several open-source JavaScript modules; any of these are potential points of weakness. Supply chain attacks of this variety are not theoretical — they have actually occurred. Recently the developer of two open-source packages, `colors.js` and `faker.js`, sabotaged these projects by pushing breaking changes to the repositories used to serve them to thousands of dependents. Many projects that had these libraries as transitive dependencies, often several layers deep, found themselves victim to this self-sabotage [12]. Another instance is the well-known SolarWinds attack, in which hackers breached the network of IT firm SolarWinds and caused the updates issued by SolarWinds to include compromised code. The perpetrators are believed to have used this to focus on a small number of high value targets, even though many were compromised [13].

Finally, in July of 2021, Cloudflare described a vulnerability found within their `cdnjs` product which they had patched and believed to not be executed. `cdnjs` "provides JavaScript, CSS, images, and font assets for websites" with the benefits that 1) visitors to the website need not re-download script files if another website they visited has already referenced them, and 2) server owners do not need to host the files themselves [14]. The vulnerability allowed remote code execution and the ability to modify assets, potentially forcing `cdnjs` to serve compromised JavaScript files. Cloudflare believes that this vulnerability was not exploited before it was patched, but it's possible that future vulnerabilities of similar scope will be found in any web-based package provider. However, the first redeveloped version of the CSPR webserver utilized `cdnjs` to deliver JavaScript libraries for data visualization, namely the `chart.js`, `moment.js`, and `chartjs-adapter-moment` packages. When the user landed on the device management page these scripts would be downloaded from `cdnjs` and, if compromised, run compromised code in the user's browser session. Note that while this vulnerability exists,

it's not possible to demonstrate an exploit on the webserver due to the requirement to take control of a package provider like cdnjs, which is illegal. Nevertheless, this vulnerability required an immediate fix.

Because the number of scripts used was small and the size of the files themselves were small enough to avoid causing hosting problems, the webserver pages were redesigned to serve a copy of stable versions of these scripts. The dependency on cdnjs was completely removed and the reliance on an auto-updating set of dependencies was severed, mitigating the vulnerability to supply chain attacks on the CSPR webserver. The updates required configuring the Apache service hosting the webserver to also serve static files from a static directory. The files are included in the code repository and managed by the Django framework.

## 3.2 SQL Injection

SQL injection is a class of attacks that exploit poor input-sanitization procedures to interact with a data store in unintended ways. A SQL injection exploit may allow an attacker to read data from the database arbitrarily, update entries in the database (possibly giving a user controlled by the attacker more access than they are otherwise authorized), delete data in the database, or change the schema of stored data in service of future exploits [15]. Since these attacks can serve to escalate privileges they are a high-value exploit sought after by malicious actors. Clarke notes that while web applications are not uniquely vulnerable and any application that receives untrusted user input has a chance of compromise, web applications are targeted most frequently due to their oftentimes database-driven nature. The inputs that the user enters will cause interactions with the database, so there are ripe opportunities for exploit [15].

Galluccio et. al [16] describe how these vulnerabilities arise: Imagine a web form in which a user enters a string to search for other users registered with the application (perhaps for the purposes of sharing a document). The web application searches for usernames matching

that substring in the database and returns a list of all partial matches, which is then placed into a dropdown list on screen to allow the user to make their selections. Under intended operation, the webserver runs a query, shown below, that is based on the user's input.

```
SELECT username
FROM users
WHERE username LIKE '<input>%'
```

The webserver replaces the <input> text with the entry entered by the user. However, if the server uses simple text-replacement to construct the query, it is vulnerable to a maliciously-engineered query. Suppose that an attacker wishes to retrieve user emails for a spamming operation. They input ' UNION SELECT email FROM users; --'. The server will perform text-replacement and execute the following query:

```
SELECT username
FROM users
WHERE username LIKE '' UNION SELECT email FROM users; —%'
```

This is most clearly reformatted as

```
SELECT username
FROM users
WHERE username LIKE ''

UNION

SELECT email
FROM users;
—%'
```

With this formatting it is clear that two queries are being executed. The first query is



“intended” by the web application programmer and is simply searching for users with no username. This query’s results are unioned with another query that reads user emails from the database. This malicious query thus returns all available user emails, and the method could be extended to determine which users are administrators, or to extract passwords or password hashes for later cracking.

Instead of searching for data, an attacker can also destroy information. With the input `' ; DROP TABLE users; --` the attacker can cause the entire users table to be destroyed, necessitating restoration from backup. Alternatively the attacker could engineer the takeover of an existing user account by changing the email associated or deleting that user and recreating it.

Galluccio et. al. demonstrate several methods of defense against SQL injection. These defenses are often mixed and matched as part of a defense-in-depth strategy. *Input validation* is performed by inspecting the user input before a query is constructed. This process can involve denying input that contains any “special” characters that are necessary for a SQL injection attack, such as `'` or `-`, or only accepting input if it matches a certain pattern (and that particular pattern only contains safe characters). *Parameterized queries* are another approach. Instead of constructing strings dynamically as shown in the previous examples, placeholder variables are used while building the SQL statement. Data is then bound to those variables, which properly escapes the input data and constructs a safe string. If the example above had used parameterized queries, the database would have correctly searched for a user named `' ; DROP TABLE users; --` instead.

According to the Django documentation, parameterized queries are used when interfacing with a SQL database from the Django framework and its Object-Relational Mapping [17]. The documentation explains that “since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.” Because the updated CSPR webserver performs all database access using Django models, the inputs are safely escaped

and the system is not vulnerable to SQL injection.

### 3.2.1 Sample attacks

This section describes attacks run on the webserver that are SQL injection attempts. In each attack an attached debugger was used to stop the program and inspect memory.

1. This attack was performed on the login form, with the goal to delete the users table from the database. The text `' ; DROP TABLE auth_user; --` was input to the login form in the username field and a garbage password was entered.

Result: The users table was not deleted. This was verified by inspecting the database and confirming that no data deletion occurred.

The steps to reproduce the attack are

- (a) With the project open, set a debugger in the file `django/contrib/auth/__init__.py` at the top of the `authenticate(request, **credentials)` function. In my project, set up following the install instructions included with the code, this file lives in the django subfolder of `~/venvs/cspr_env/lib/python3.7/site-packages`.
- (b) Run the project with debugger attached.
- (c) Navigate to `http://localhost:8000`. Logout of the CSPP website, if logged in.
- (d) Login to the MySQL instance. I run the following commands to login and verify that the table exists before the attack:

```
> sudo mysql -u cspr -p
> use cspr\_db;
> select * from auth\_user
```
- (e) On the CSPP website, enter any text you like into the password field, and enter

the text `' ; DROP TABLE auth_user; --` into the username field (see figure 5). Submit the form.

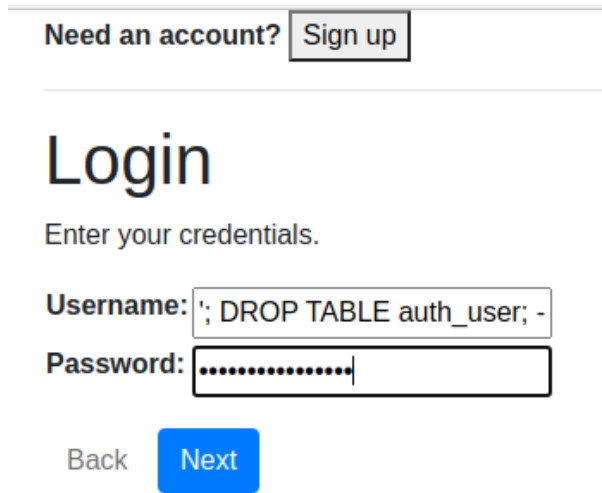
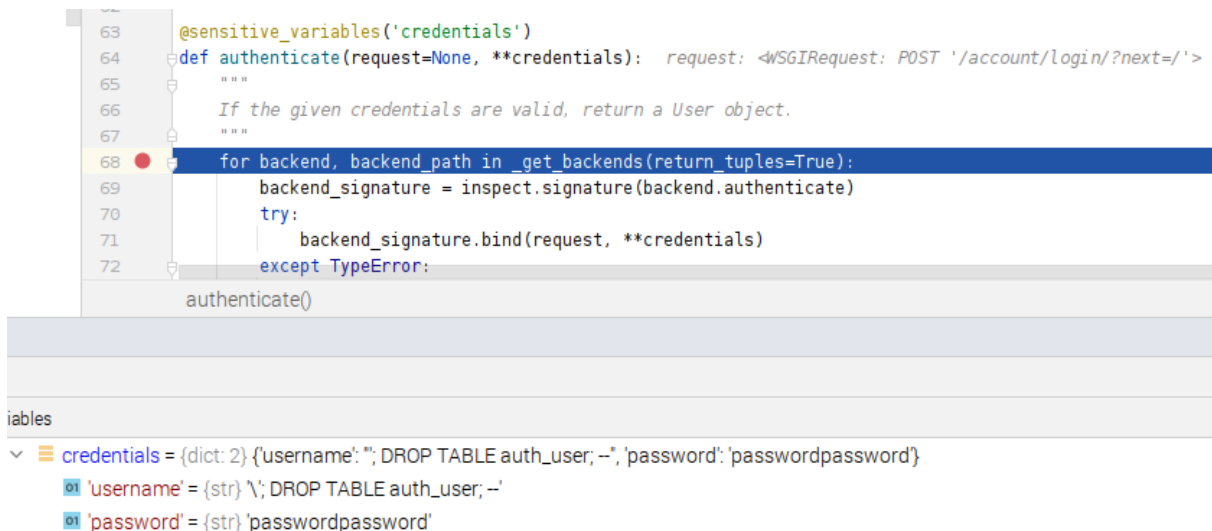


Figure 5: Entering the SQL Injection payload.

- (f) Your debugger should pause at the `authenticate` function where the breakpoint was set (see figure 6). From here it can be verified that the malicious input was received by the server and will be used to look up a user. Resume the program.



```
63 @sensitive_variables('credentials')
64 def authenticate(request=None, **credentials): request: <WSGIRequest: POST '/account/login/?next='>
65     """
66     If the given credentials are valid, return a User object.
67     """
68     for backend, backend_path in _get_backends(return_tuples=True):
69         backend_signature = inspect.signature(backend.authenticate)
70         try:
71             backend_signature.bind(request, **credentials)
72         except TypeError:
73             continue
74     return backend.authenticate(request, **credentials)
75
authenticate()

variables
credentials = {dict: 2} {username: '; DROP TABLE auth_user; --', password: 'passwordpassword'}
username = {str} \'; DROP TABLE auth_user; --
password = {str} 'passwordpassword'
```

Figure 6: The debugger paused at the point of receiving the POST request from the client. Note that the username field shown at the bottom of the screen matches the form, indicating that the server has received the input as written and mitigation of the attack must be server-side.

- (g) Verify after resuming the program that the login fails and the `auth_user` table still exists (see figure 7. The attack has failed.

```

MariaDB [csprr_db]> select * from auth_user;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | password | last_login | is_superuser | username | first_name | last_name | email | is_staff | is_active | date_joined |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | pbkdf2_sha256$268886539$e2121n10a3z6v02708wpdJ9m0m115fKv85PM31rWTF8p52QjYm01+yM0e= | 2021-10-26 02:07:02.705323 | 0 | temporary | | | | 0 | 1 | 2021-10-26 02:01:58.512615 |
| 2 | pbkdf2_sha256$268886539$eMq056rDQ1fKRbHc0jISv11s1fPm9eCw0tHvppc3xU0Z9v/0RmUJ33uARXSSXU= | 2022-02-21 21:09:10.599222 | 1 | csprr | | | | 1 | 1 | 2021-10-26 03:00:17.312775 |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

Figure 7: Verifying that the `auth_user` table still exists

2. This attack was performed on the login form, with the goal to gain access to another user without possessing that users password. A testing user was created named `sqltest2`. The text `sqltest2` was entered into the username field and the text `' AND 1=1; --` was entered into the password field.

Result: No users were returned by the query, which was verified by inspecting the immediate query results with an attached debugger.

To reproduce this attack after creating the `sqltest2` user:

- (a) Follow steps 1a, 1b, and 1c.
  - (b) On the CSPR website, enter the username `sqltest2` and the password `' AND 1=1; --`.
  - (c) Submit the form and verify with a debugger (similar to step 1f) that the input is received as written. Note that the login fails, and thus the attack has failed.
3. This attack was also performed on the login form, with the goal of updating an existing user to have administrative permissions. Using the same testing user as before, `sqltest2`, we attempt to login with a query that has the side effect of updating that user's admin flag to true. While logging in a query is appended that searches for the user and sets the `is_staff` to 1:

```
sqltest2'; UPDATE auth_user SET is_staff = 1 WHERE username = 'sqltest2'; --
```

Any text is allowed for the password.

To reproduce this attack:

- (a) Follow steps 1a, 1b, and 1c.
- (b) On the CSPR website, enter the username above and any password.
- (c) Submit the form and verify with a debugger (similar to step 1f) that the input is received as written. Continue execution and then inspect the database table. Note that the user's `is_staff` flag is still 0, indicating that the attack has failed. This failure can also be verified by logging in as the user and noting the lack of any links to the admin page.

### 3.3 Cross-Site Scripting

Cross-site scripting (abbreviated XSS) exploits unsanitized user input to change the structure of a web page or execute arbitrary code in the client browser [18]. The attack occurs when the site displays compromised text from a user. Because the user's input is part of the HTML content of the page, carefully crafted input can be rendered by the browser as if it was HTML text.

In a simple example, imagine a user profile page that includes a freeform text field for the user to write a description of themselves. The contents of that field are rendered inside a `<p>` tag for visitors to the profile page. If the user enters the text

Hello , `<b>world!</b>`

into the text field, the server will render the entered content inside of the `<p>` tags and the `<b>` tags will cause the content to be written in bold. This is due to the fact that the browser interprets the text as part of the HTML page.

More malicious variants of this attack exist. The Django security documentation describes how attackers can run scripts in the client's browser by storing the malicious script in database fields and then retrieving them for other users [17]. In the CSPR webserver defense against these attacks is provided both by the Django framework and by the schema of the database and models. Django escapes specific characters that must be used to perform an XSS attack, which works by converting characters according to the following scheme [19]:

- `<` and `>` are converted to `&lt;` and `&gt;`;
- `'` and `"` are converted to `&#x27;` and `&quot;`;
- `&` is converted to `&amp;`;

These changes protect from most attacks, but are still dependent on myself, the programmer, to avoid using the templating language in places where attackers can inject code without those characters. In some forms Django will also restrict character entry to certain allowed characters, like when creating users.

Another defense strategy is based on the database schema and model definitions. When entering values for registers on the device management page the server expects all inputs to be numbers. This requirement is enforced both on the front end and on the back end. The webpage shown to the user requires input to be numeric, but this can be defeated with client-side debugging tools. Therefore, the server also verifies that received data is numeric for these fields and returns an exception if the field is not numeric. Finally, the database's data types for certain fields also assist in enforcing data validity.

### 3.3.1 Sample attacks

1. I verified that the signup form does not allow users with the username `testuser<b>xss</b>` to be created. Django verifies the text on the server-side and refuses to create the user because it contains `<` and `>` in its username. To reproduce this attack, follow these

steps:

- (a) Run the project and navigate to the home page. If you are logged in, log out.
  - (b) Click on the signup button to go to the signup page.
  - (c) Enter the text `testuser<b>xss</b>` for the username field, and any password.
  - (d) Submit the form and notice that Django rejects the submission for having illegal characters in its username.
2. In this attack I edited an existing device's name in the database to demonstrate that Django properly escapes characters. A device had its name changed to `<i>Device</i>` by manipulating the database. Note that this change of data would require an attacker to first infiltrate the database, but it is useful as a demonstration of Django's anti-XSS features. To reproduce this attack:
- (a) Create a device using an admin user.
  - (b) Access the MySQL database with your admin user. On my testing machine, the command for this is `> sudo mariadb -u <username> -p`
  - (c) Use the database for the Webserver. `> use cspr_db`
  - (d) Run the following SQL command to change device names to `¡¿Device¿/i¿`. Note that this will update the name of every device, but this is sufficient for testing purposes.  
  
`> update devices set name = '<i>Device</i>';`
  - (e) Run the webserver, login, and navigate to a device page. The name of the device appears as written above, and not in italics, indicating that Django has detected this attempt at cross-site scripting and properly escaped the characters to foil it.



Figure 8: The XSS attack has failed, as the tag text is still visible and has not been parsed as HTML.

For another example, use the text `<script>alert('XSS')</script>Device` instead, which will produce a pop-up dialog in the browser if it is successful. By extension, we can see how XSS attacks can enable the embedding of malicious scripts for the purpose of hijacking a client session, harvesting data, or escalating to another exploit.

## 4 General Security

### 4.1 Multi-Factor Authentication

In the most basic sense a multi-factor authentication (MFA) scheme requires the user attempting to access resources to present multiple proofs of identity; a real-life example might be the multitude of documents needed to obtain a passport. Each additional independent credential increases the difficulty of an account takeover. The Cybersecurity and Infrastructure Security Agency states [20]:

A typical MFA login would require the user to present some combination of the following:

- Something you know: like a password or Personal Identification Number (PIN);
- Something you have: like a smart card, mobile token, or hardware token; and,



- Some form of biometric factor (e.g., fingerprint, palm print, or voice recognition).

Multi-factor authentication was added to the CSPR webservice by using the plugin `django-two-factor-auth` [10]. During the signup process the user is able to create an account without providing a second factor, but when attempting to access any page the user must be authenticated with a second factor just as they must be logged in. If the user is not authenticated they are redirected to the plugin's provided page to enable a second factor. For development purposes, the plugin is currently configured to print the 2FA code to the developer console; configuration options exist to send the second-factor codes over SMS.

## 4.2 Man-in-the-Middle

HTTP is vulnerable to man-in-the-middle (MITM) attacks, where an attacker impersonates the server and/or client while the real server and client transact information. Sitting in the middle of the two, the attacker has full access into the requests and responses and is free to edit data, save responses for later, or extract secret information. These attacks can result in total compromise of sensitive information, or enable the attacker to bypass security protocols by impersonating an authorized user

Transport Layer Security (TLS) is used to prevent this kind of attack; it is one of a kind of key exchange protocols that require the communicating to authenticate and is the dominant such protocol for the purpose of securing web traffic. Boyd et. al. outline how TLS differs from Secure Shell (SSH) and Internet Protocol Security (IPSec), but most critical to know is that web traffic secured using TLS is called HTTPS. There have been two versions of SSL and four versions of TLS, with most browsers and sites currently operating on TLS 1.2 and migrating to TLS 1.3 [21].

In an HTTPS context, the communication parties establish a TLS connection before sending HTTP data across the connection. In the handshake procedure the parties exchange cry-

tographic keys and thereafter can communicate with encrypted data. This prevents MITM attacks not by preventing an actor from intercepting the messages, but by preventing them from decoding any intercepted messages. TLS also has the property that the keys have forward secrecy, meaning that if, in the future, the key used by the server is compromised the old session data still cannot be recovered and an attacker must still own the client's key to decrypt the communicated data [21].

For this project, demonstrative HTTPS support was added to the webserver by configuring an instance of the Apache webserver to use a locally-generated certificate. In a production environment this would be switched for a certificate obtained from a certifying authority, such as Let's Encrypt (a service provided by the Internet Security Research Authority [22]).

## 5 Conclusion

Network-connected electric grid hardware promises to make the electric grid better-monitored and more efficient, and designing smart grid systems in a secure manner is vital to their continued operation. Since the electric grid is critical infrastructure, a defense-in-depth approach is necessary to secure the network-connected components of the smart grid. Thus, any interface to components of the grid system must also have defensive measures. The demonstrated attacks against the CSPR webserver serve to test its ability to defend against common intrusion attempts, and the general security work conforms to the defense-in-depth approach by making each attempted attack more costly.

The CSPR webserver utilizes the Django framework's approach to secure user input against SQL Injection and XSS attacks, and serves its own static files to greatly increase the difficulty of supply chain attacks.. With a security certificate and an HTTPS-enabled host, the web server can be assured that man-in-the-middle attacks are guarded against, and two-factor authentication provides additional layers of authentication to ensure that multi-

ple parts of a user’s identity must be compromised to hijack their account. These changes provide a strong foundation for cybersecurity and leave room for enhancements to follow. Possible enhancements include intrusion detection based on network activity analysis, or a dependency-verification system that monitors for vulnerabilities in the dependencies of the web application and notifies developers of potential vulnerabilities in their supply chain.

## References

- [1] V. Pillitteri and T. Brewer, “Guidelines for Smart Grid Cybersecurity,” September 2014.
- [2] D. E. Sanger, C. Krauss, and N. Perlroth, “Cyberattack Forces a Shutdown of a Top U.S. Pipeline,” *The New York Times*, May 2021. [Online]. Available: <https://www.nytimes.com/2021/05/08/us/politics/cyberattack-colonial-pipeline.html>
- [3] D. E. Sanger, February 2016.
- [4] F. Robles and N. Perlroth, “‘Dangerous Stuff’: Hackers Tried to Poison Water Supply of Florida Town,” February 2021. [Online]. Available: <https://www.nytimes.com/2021/02/08/us/oldsmar-florida-water-supply-hack.html>
- [5] “Choosing and Protecting Passwords,” Cybersecurity & Infrastructure Agency, 2022. [Online]. Available: <https://www.cisa.gov/uscert/ncas/tips/ST04-002>
- [6] B. Allen. CSPR Webserver. [Online]. Available: <https://git.uark.edu/bea005/cspr-webserver>
- [7] “Django,” Django Software Foundation, 2022. [Online]. Available: <https://www.djangoproject.com>
- [8] “Mariadb,” MariaDB Team, 2022. [Online]. Available: <https://mariadb.org>
- [9] J. Berg, “MinimalModbus Documentation,” 2022. [Online]. Available: <https://minimalmodbus.readthedocs.io/en/stable/>
- [10] “Django Two-Factor Authentication Documentation,” Jazzband, 2022. [Online]. Available: <https://django-two-factor-auth.readthedocs.io>
- [11] “Chart.js,” Chart.js Team, 2022. [Online]. Available: <https://www.chartjs.org>
- [12] E. Roth, “Open source developer corrupts widely-used libraries, affecting tons of projects,” *The Verge*, January 2022. [Online]. Available: <https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source->

libraries-projects-affected

- [13] “SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response,” Government Accountability Office, April 2021. [Online]. Available: <https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic>
- [14] J. Ganz, T. Calderon, and S. Sauleu, “Cloudflare’s Handling of an RCE Vulnerability in cdnjs,” Cloudflare, July 2021. [Online]. Available: <https://blog.cloudflare.com/cloudflares-handling-of-an-rce-vulnerability-in-cdnjs/>
- [15] J. Clarke, *SQL Injection Attacks and Defense*. Waltham, MA: Elsevier Science and Technology Books, 2012.
- [16] E. Galluccio, E. Caselli, and G. Lombardi, *SQL Injection Strategies : Practical Techniques to Secure Old Vulnerabilities Against Modern Attacks*. Birmingham, UK: Packt Publishing, 2020.
- [17] “Security in Django,” Django Software Foundation, February 2022. [Online]. Available: <https://docs.djangoproject.com/en/4.0/topics/security/>
- [18] M. Shema, *Hacking Web Apps : Detecting and Preventing Web Application Security Problems*. Waltham, MA: Elsevier Science and Technology Books, 2012.
- [19] “The Django template language,” Django Software Foundation, February 2022. [Online]. Available: <https://docs.djangoproject.com/en/4.0/ref/templates/language/automatic-html-escaping>
- [20] “Multi-Factor Authentication,” Cybersecurity & Infrastructure Security Agency, January 2022. [Online]. Available: <https://www.cisa.gov/publication/multi-factor-authentication-mfa>
- [21] C. Boyd, A. Mathuria, and D. Stebila, *Protocols for Authentication and Key Establishment*. Berlin, Germany: Springer-Verlag, 2020.
- [22] “Let’s Encrypt,” Internet Security Research Authority, 2022. [Online]. Available: <https://letsencrypt.org>