

HANDS-ON WORKSHEET

Security at the Campus Festival

Five short experiments in authentication, software security, cryptography, network security, and anonymity/privacy.

The campus festival is coming up. The organisers have a guestbook, an online ticket service, and a collection of visitor data. Your job is to analyze the security of these features. Investigate what can go wrong and try out ways to improve things.

How this lab works

You do not need previous security knowledge or programming experience for the core tasks. Work in pairs if possible, take turns at the keyboard, and keep a short record of what you tried.

Each station has a **core challenge**, something concrete to **save or show**, and an optional **bonus**. If you already know the basics, move to the bonus as soon as you have demonstrated the core result. Hints are on the last page: use them whenever they help.

Station	Challenge	Core time
1	Guess the PIN	15 min
2	Make a guestbook interpret your input	15 min
3	Change an encrypted ticket order	15 min
4	Investigate a network recording	15 min
5	Re-identify visitors in a dataset	15 min

Allow about two hours including setup and discussion. Your tutor may select fewer stations for a shorter session. The stations are independent: you can start anywhere.

What you need

- A browser and a text editor; Wireshark for Station 4.
- The course material pack: a guestbook page, an encryption playground, two packet recordings, and two fictional visitor datasets.
- Optional for the bonuses: a programming language of your choice.

Keep a lab record

Save your successful inputs, changed files, commands, or screenshots in one folder. A short, reproducible demonstration is enough; you do not need to write an essay. If an attempt fails, keep it too: it may help you find the next step.

Scope of the experiments

Use only the supplied exercise pages, recordings, and fictional data. The attacks in this worksheet are intended for these practice materials. Ask your tutor if you are unsure which target belongs to the lab.

STATION 1 / AUTHENTICATION

Guess the PIN, then slow the guessing down

Goal: Automate a guessing attack against a simulated login and experiment with a defense.

Start here

Open the supplied PIN-login playground. It simulates a login protected by a four-digit PIN, using digits from 0 to 9. Leading zeros are allowed. The account and PIN are fictional. The playground includes a guess generator, an attempt counter, and a simulated clock. No programming is required for the core challenge.

Core challenge

1. Try a few PINs manually. Then configure the guess generator to try every possible four-digit PIN, starting with 0000. Run it until a login succeeds.
2. Record the number of attempts and the simulated time needed. Reset the experiment to use the same secret PIN again.
3. Enable a temporary account lockout after five failed attempts. Choose a lockout duration and repeat the attack. The simulated clock lets you advance time without waiting in real life.
4. Play the role of the legitimate user: deliberately enter five wrong PINs, then immediately enter the correct one. Observe whether you can still log in.
5. Adjust the defense and repeat both experiments. Try to slow down guessing while keeping accidental mistakes manageable.

Save or show

Keep the successful PIN and a small table comparing your experiments: defense settings, number of guesses processed, simulated time, and whether the legitimate user could log in immediately.

Bonus: implement the defense

Use the supplied starter code to implement the guessing loop or the login defense yourself. Preserve leading zeros when generating PINs. Add tests for a correct PIN, an incorrect PIN, the lockout threshold, and a successful login after the lockout expires.

Then simulate two clients attacking the same account. Check whether switching clients bypasses your limit. Also test whether repeated failed attempts can keep the legitimate user locked out.

Help and further exploration

- Four decimal digits give $10^4 = 10,000$ possible PINs. An automated attack can systematically enumerate them.
- A temporary lockout needs state: a failed-attempt counter and a time until which attempts are blocked.
- The [OWASP Authentication Cheat Sheet](#) covers login throttling, account lockout, and multi-factor authentication.

This playground deliberately uses a short PIN to make exhaustive guessing easy to observe. Its simulated controls are part of the exercise; a real service must enforce login protection on the server.

STATION 2 / SOFTWARE SECURITY

The guestbook follows your instructions

Goal: Make a page interpret input as code, then change it to treat that input as text.

Start here

Open `guestbook.html` in your browser. It is a deliberately vulnerable, local practice page. Submit a normal message and observe where it appears. You can inspect and edit the page with a text editor; reload the browser after saving changes.

Core challenge

1. Enter a message that appears as a large heading instead of ordinary text. Only use the guestbook input field for this step.
2. Find an input that makes the page open a JavaScript alert containing **Hello Security!**. You may use the hints to learn the required syntax.
3. Open the HTML file in your editor and find the marked line that displays the submitted message. Change it so that messages are displayed as plain text.
4. Reload the page and test three inputs: an ordinary message, your heading input, and your alert input. All three should now appear literally, without formatting or code execution.

Save or show

Keep your two successful attack inputs and the repaired HTML file. Demonstrate that a normal message still works and that both attack inputs are now displayed as text.

Bonus: allow some formatting safely

The organisers want to allow bold and italic messages. Create a second version that allows `<strong>` and `<em>` but rejects executable content. Use an established HTML sanitiser rather than building a filter from string replacements. Configure an explicit allowlist of the permitted tags and attributes.

Build a small set of test inputs: allowed formatting, an event-handler attribute, malformed markup, and a mixture of harmless and dangerous content. Check both that useful formatting survives and that code does not execute.

Help and further exploration

- [MDN: Cross-site scripting \(XSS\)](#) explains how input can become executable content.
- [MDN: textContent](#) describes how to insert plain text.
- [OWASP Juice Shop](#) offers further intentionally vulnerable challenges and guided tutorials.

STATION 3 / CRYPTOGRAPHY

Change an encrypted order

Goal: Modify a message without knowing its encryption key.

Start here

Open the supplied encryption playground. It shows the original order `tickets=1`, its byte values, and its encrypted form. The secret key is hidden. You can edit the encrypted bytes and ask the receiver to decrypt the result.

This experiment uses a simplified construction: each message byte is combined with a secret key byte using XOR. XOR acts on individual bits: equal bits give 0; different bits give 1. Applying the same XOR value twice undoes the change. The playground includes a calculator.

Core challenge

1. Locate the byte for the character `1` in the original message. Remember that the character `1` is represented by a byte value; it is not the number 1.
2. Change only the corresponding encrypted byte so that the receiver reads `tickets=9`. Do not change the original message or reveal the key.
3. Repeat the experiment with another single-digit ticket count. Keep a record of the original and modified encrypted byte values.
4. Switch to the supplied authenticated-encryption version. Change an encrypted byte and submit it. Confirm that the receiver rejects the modified message rather than accepting a changed order.

Save or show

Demonstrate one accepted, modified order in the first version and one rejected modification in the authenticated version. Save the byte changes needed to reproduce your first result within the same experiment.

Bonus: automate and investigate key reuse

Write a function that changes a known part of an encrypted message into a chosen replacement of the same length. It should accept the ciphertext, the offset, the known text, and the replacement text. Test it on at least two different replacements.

Then use the bonus dataset: two messages were encrypted with the same key stream, and one complete plaintext is known. Recover the other message wherever the known plaintext provides enough bytes. The playground keeps a key fixed within one experiment; accidental reuse across different messages is a separate mistake explored in this bonus.

Help and further exploration

[CyberChef](#) provides XOR and byte-conversion operations. Choose byte encodings explicitly when using it, especially when copying hexadecimal values.

The construction in the first part is a teaching model without an authenticity check. The second part demonstrates why real applications need authenticated encryption, not just hidden message contents.

STATION 4 / NETWORK SECURITY

Read the network recording

Goal: Recover visible application data and inspect what remains observable with HTTPS.

Start here

Open `festival-http.pcapng` in Wireshark. This is a prepared recording of fictional traffic; you do not need to capture your own network. Each row is a packet. Selecting a row reveals more details below it.

Core challenge

1. Find the HTTP request that sends a test access code to the festival portal. Recover the code and the requested path.
2. Reconstruct the request with *Follow TCP Stream*. Save the relevant excerpt and note a packet number that helps someone else find it.
3. Open `festival-https.pcapng`, which records a comparable interaction protected with HTTPS. No TLS session keys are supplied for the core task.
4. Record three things you can directly observe, such as a destination IP address, a timestamp, or an individual packet's length. For each observation, include a packet number and the field you used. If a hostname is visible, show where; do not assume that it must be visible.

Save or show

Keep the recovered HTTP code and request excerpt, followed by three observations from the HTTPS recording with supporting packet numbers. Separate directly observed values from guesses about what the user was doing.

Bonus A: automate the investigation

Use `tshark`, Wireshark's command-line tool, or a packet-processing library to produce a list of connections and the total captured bytes in each. State how you identify a connection and which length field you sum. Compare one result with Wireshark's conversation statistics.

Bonus B: decrypt with session secrets

Load the supplied bonus TLS key log into Wireshark and inspect the HTTPS recording again. Recover the application request. Save the relevant excerpt. The key log contains session secrets intentionally provided for this exercise; possessing a packet recording alone does not provide these secrets.

Help and further exploration

- Start with display filters such as `http` and `tls`. Clear a filter to see all packets again.
- Right-click a relevant TCP packet and select *Follow → TCP Stream*.
- The [Wireshark User's Guide](#) covers filtering, stream reconstruction, conversation statistics, and TLS key logs.

STATION 5 / PRIVACY

No names, no problem?

Goal: Link records to people, then release less revealing data.

Start here

Open the supplied visitor table and public-post table. All people and records are fictional. The visitor table contains stable visitor IDs, events, and precise visit times, but no names. The posts contain names and clues about events and times.

You can solve the core task by reading, sorting, and marking the tables. No programming is required. Treat the supplied posts as accurate within this fictional exercise.

Core challenge

1. Combine clues from both tables to associate at least three visitor IDs with named people. Use more than one visit where necessary.
2. For each association, mark the supporting row IDs in both tables. Check whether another visitor also matches the same clues. Keep ambiguous matches separate from unique matches.
3. Create a revised publication that makes your previous unique associations impossible using the supplied posts, while still allowing the organisers to determine visitor counts for each event. You may remove fields, make values less precise, or publish an aggregate table instead of individual records.
4. Repeat your matching attempt against the revised publication. Check that the event counts are still correct.

Save or show

Keep three associations with their supporting rows and your revised publication. Demonstrate both that the original links no longer work uniquely and that the event visitor counts remain available.

Bonus: scale up the linkage attack

Use the larger bonus dataset with Python, SQL, or another tool. For each person's public clues, compute the set of matching visitor IDs. Report how many people have zero, one, or multiple candidates.

Repeat the analysis after rounding visit times to 5, 15, and 60 minutes. Apply a consistent matching rule to the clues and visit records. Compare how many unique matches remain. Investigate whether combining several visits can still single someone out even when each visit alone is common.

Help and further exploration

- Begin with an unusual event or time, then use another clue to narrow the candidates.
- The EFF's [introduction to metadata](#) explains why information surrounding an activity can reveal a great deal.

Stopping this particular linkage attack is not a proof of anonymity against all possible additional information. Your result is an experiment with the supplied data and clues.