Phishing Simulator Tool

Final Year Project Report

A Django-Based Phishing Simulation Platform for Security Awareness Training

Stephen Foley

South East Technological University

BSc (Hons) in Cybercrime & I.T Security

Supervisor Hisan Elshafi

April 2026

Table of Contents

Executive Summary

This report documents the design, development, and evaluation of the Phishing Simulator Tool, a Django-based phishing simulation platform built as a final year project for BSc (Hons) in Cybercrime & IT Security at South East Technological University.

The platform enables security researchers and organisations to run controlled phishing simulation campaigns for the purpose of measuring and improving employee security awareness. It supports the full campaign lifecycle — from email composition and dispatch to credential capture, risk scoring, and automated delivery of awareness training — all within an ethical framework aligned with GDPR requirements.

The system was built using Django 6, PostgreSQL, Redis, and Celery, deployed via Docker Compose as a fully containerised multi-service stack. Key capabilities include five realistic cloned login pages (Google, Microsoft 365, DocuSign, OneDrive, ServiceNow), two-stage login flows, federated sign-in simulation, randomised email scheduling, QR code campaign support, and a multi-factor risk scoring algorithm. Full participant anonymisation using auto-generated P-XXXXX identifiers ensures no personally identifiable information is exposed in the dashboard or reports.

All seven core project deliverables were achieved and verified by an automated test suite of 65 tests across 65 test cases covering models, utility functions, tracking views, and feature behaviour, all passing with a runtime of approximately three seconds.

Development involved resolving seven significant technical challenges, including HTTPS redirect loops in the test environment, Celery module import crashes, browser event propagation conflicts in two-stage login flows, and session-tracking

gaps for QR code access. Each issue was diagnosed, resolved, and documented as a lesson applied to future engineering work.

Key areas for future improvement include implementing a CI/CD pipeline, extending test coverage to admin views, adding mobile-responsive dashboard styling, and applying @login_required decorators to all dashboard views before any production deployment.

1. General Issues

This report discusses my experience developing my final-year project, a Phishing Simulation Tool. It is a Django-based phishing simulation platform designed to test and train users in recognising modern phishing techniques. I will talk about what went well, where it didn't, what I learned from this project, and what I would do differently if I were to start again. The goal of this report is to demonstrate that I planned a significant project using modern tools and that I can do so again.

1.1 Problems Encountered and how they were resolved

HTTPS redirect loop breaking the test suite:

The first time I ran the pytest suite, all the HTTP tests failed and returned 301 redirects. After a lot of debugging, I realised that this was because SECURE_SSL_REDIRECT = TRUE was enabled. This meant that every test was silently redirected to HTTPS, and the POST data was dropped during the redirect.

To fix this issue, I introduced a dedicated testing module, test_settings.py, which inherits the main settings from my project but disables SECURE_SSL_REDIRECT, SESSION_COOKIE_SECURE, CSRF_COOKIE_SECURE, and HSTS. Then I reconfigured pytest to use this module. This allowed me to separate the test and production environments, enabling me to build a usable testing suite.

CSS Breaking:

While I was styling the dashboard and was happy with it, I turned off Django's Debugging. For some reason, this caused the CSS file and other static files to be dropped, and when trying to access them, a 404 error was returned.
Further investigation showed me there were two separate issues. First, Django's development server automatically serves files when DEBUG = True, but in production, it does not. To fix this, I had to download and enable WhiteNoise, which is a middleware file. This ensured that all static file requests were intercepted before they reached Django's routing layer. The second issue was that STATIC_URL was configured to 'static/', which used a relative path rather than the absolute path '/static/'. This resulted in broken URLs on nested pages, such as /dashboard/campaigns, which caused the browser to load the relative path rather than the absolute path. Neither issue was visible because they were handled automatically with DEBUG = True.

QR code landing pages returning 404:

When scanning the QR codes on a mobile device, it would not load. However, clicking the link in the email would.

This was because the landing_page view was configured to look up the active view in the Django session, where the tracking token had been stored when the user clicked the email link. The QR code scans didn't include session context, so when the session lookup function ran, it couldn't find any session, resulting in 404 responses.

To fix this, I had to add a fallback lookup path. When no session token is present, it automatically queries the campaign model for the record whose landing_path matches the requested URL path. The cloned site is served directly via a separate function that neutralises form submissions and links, since there is no token to track them. This taught me the importance of considering all features before implementing one element of my projects going forward.

Form Submission Return JSON instead of Redirect:

After a user submitted credentials on a phishing page, they saw a raw JSON response {"status": "success"} rendered in their browser instead of being redirected to the training page.
The track__form_submission view was returning a JSON response. This is fine if the

platform is used via the API, but not via the browser submission method. A browser sending a POST request with an HTML form embedded expects a 302 HTTP redirect response, not the JSON payload. Because the response wasn't a redirect, it displayed the JSON response.

Change the values from JsonResponse to `HttpResponseRedirect(reverse('show_training',args=[token]))` resolved the issue for me. The same mismatched response was also found in the completed_training view, so I fixed that at the same time. This showed me that people using the views in the browser and API mode needed different responses.

Celery crashing all dashboards:

After implementing Celery support for randomised email scheduling, all dashboard routers began returning an HTTP 500 error whenever Redis was not running. This included trying to access the dashboard to view analytics.

The case was using a top-level module import `from .tasks import send_individual_email` in views_dashboard.py. Python executes top-level imports at module load time, so when Django starts and attempts to load the views_dashboard module, it tries to import the Celery task, which in turn tries to connect to Redis. When this connection failed, the import raised an error, causing the rest of the module to fail to load.

To fix this, I had to move the import into the specific function that needed it and also wrap it in a try/except block with a fallback to synchronous email sending. This was one of the more subtle bugs encountered during development: the error was thrown in the dashboard views, but the cause was in a different module.

Two-stage login pages are being submitted immediately:

The two-stage login flows for the Microsoft and Google clone pages were not functioning correctly. It was showing both the email and password fields on the landing pages, whereas the legitimate pages first ask for an email, then ask for a password. I fixed this issue, but then another issue presented itself, which was that clicking submit on the email form failed to load the password page. Instead, it immediately went to training.

The phishing injection JavaScript uses capture-phase event listeners on all button elements with `stopImmediatePropagation()` to ensure form submission is tracked. The capture-phase listeners fired before the bubble-phase listeners, and stopImmediatePropagation() prevents all subsequent event listeners from executing. This was intercepting the user clicking on the "Next" button before its on-click listener could execute.

The solution I came up with was to change the "Next" button from a button element to a styled <a> tag, which the script does not intercept. Only the submission button on the password page remained a normal button element. This showed me, at a detailed level, the difference between the capture and bubble phases, and gave me a much better understanding of how `stopImmediatePropagation()` works. This also provides an insight into how real phishing kits work.

Tracking returning 404 for invalid tokens:

While testing, I discovered that the email-tracking endpoint returned a 404 when an invalid or expired token was provided. Email clients that follow tracking pixel URLs would display a broken image instead of the expected image.

This would make it very obvious to participants that this was not a legitimate email and that they could identify it as part of the phishing simulation. To fix this, I implemented a way always to return a valid 1x1 transparent GIF, regardless of whether the token was found. This allowed it to fail silently if it wasn't found. This was discovered in testing and shows how it catches flaws before deploying the tool.

1.2 What was Achieved

My project delivers a fully functional phishing simulation platform that meets and exceeds the core deliverables defined in my project specification.

The following capabilities were implemented and verified through the automated test suite and real-world testing:

- End-to-end phishing campaign lifecycle: email composition, template variable substitution, campaign dispatch, credential capture and automatic redirect to awareness training.

- Five realistic phishing templates with cloned login pages: Google sign-in, DocuSign, Microsoft 365 login, OneDrive File Share, ServiceNow, and it helpdesk.
- Two-stage login flows that mimic the user interface of the legitimate websites, including enter email -> enter password flows for Google and Microsoft login pages.
- Federated sign-in simulation: In the DocuSign cloned landing page, there is a sign-in with Microsoft button that redirects users to the fake Microsoft 365 login page while maintaining the tracking context of the original campaign sent.
- Randomised email scheduling using Celery and Redis, spreading sends across a configurable time window to prevent participants from detecting a pattern.
- Automated risk scoring algorithm which weighs click behaviour, credential submission rate, time to click, scenario difficulty and phishing report rate.
- Full participant anonymisation using auto-generated five-character codes in the format P-XXXXX. This ensures that no personally identifiable information is visible in the dashboard or admin interface.
- Informed consent management system with consent forms, confirmation pages and withdrawal functionality, aligned with GDPR requirements.
- Interactive awareness quiz with configurable questions, scoring and pass/fail thresholds.
- QR code phishing campaign support with device-specific redirect logic for mobile and desktop.
- CSV import and export for participant management and campaign results reporting. Researcher dashboard with real-time event feed, risk distribution charts and individual campaign metrics.
- Docker support via Docker Compose, enabling single command deployment of the full stack (PostgreSQL, Redis, Django, Celery worker, Celery beat).
- 65 automated tests covering features such as models, utility functions, tracking views, landing page routes and CSV importing.
- Comprehensive README to explain how to get started with the tool and how to use it.
- GNU General Public License V3

All seven core deliverables outlined in my project specification were achieved.

1.3 What wasn't achieved

Four items that I planned on implementing were not successfully integrated within the project timeline:

- Full test coverage: While I built 65 tests to cover the critical systems, some edge cases and Django Admin views remain untested. I wrote the tests towards the end of development rather than alongside each feature, which limited their scope.
- Mobile responsive dashboard: While the dashboard is fully functional on desktop, it doesn't work on mobile. A CSS media query pass would be needed before deploying to a mobile audience.
- Automated CI/CD pipeline. The project uses manual deployment by launching the quickstart and Docker files. A GitHub Actions workflow for automated test execution on each push was planned but not implemented.
- SMS phishing (smishing) capabilities in the simulator. At first, I listed this as an optional feature, but given the complexity of the core features, I decided not to implement it.

1.4 What was learned

This project considerably expanded my technical knowledge and practical skills across several areas:

Django in Production

The difference between running a Django application in development mode and one running behind a reverse proxy with DEBUG=false was the source of multiple bugs in my project. This project provided a detailed, practical understanding of WhiteNoise for static files, SECURE_PROXY_SSL_HEADER for reverse-proxy HTTPS handling, CSRF_TRUSTED_ORIGINS for cross-origin forms, and the importance of testing in a production configuration at each stage rather than all at the end.

Asynchronous Task Processing

Integrating Celery with Redis taught me about message brokers, asynchronous task execution, task scheduling with ETA parameters, and the challenges of designing graceful degradation when parts of the infrastructure are unavailable. I learned how top-level imports can cause modules to crash and how to implement imports properly going forward.

Browser Event Model and Injected JavaScript

The two-stage login page bug required me to learn in depth about capture-phase versus bubble-phase event propagation, the behaviour of stopImmediatePropagation(), and how to effectively design HTML elements that coexist with the injected JavaScript they were not designed to interact with. The knowledge from solving this bugs applies to real phishing kits and how browser extensions operate.

Ethical Security Research Practice

Building a tool that simulates real attacks requires constant evaluation of ethical boundaries, such as ensuring informed consent, anonymising participant data, never storing credentials entered, and providing immediate educational feedback after every simulated compromise. These constraints shaped my systems architecture in good ways; the anonymisation system, the consent management views, and the training logic all exist because of ethical requirements. This demonstrates how ethics in security research translates into designing decisions.

Testing Strategy and Test-Driven Thinking

Writing the test suite after most of the development was complete taught me that this was a mistake. If I had written the tests at each stage, it would have been easier to catch the bugs. Several issues were revealed during my tests that weren't found in manual testing; some of these were tracking-pixel design flaws and edge cases caught in the CSV import logic. While ultimately all 65 tests pass, this experience demonstrates the practical value of writing tests alongside features.

Multi-Service Architecture

Managing Django, PostgreSQL, Redis, a Celery beat, and a Celery Worker as a coordinated system, both manually via the terminal and automatically using Docker, provided me with practical experience that is not covered in my course modules. Understanding how all these services depend on one another for different elements and what actually happens if something is unavailable is knowledge that would be difficult to gain from coursework unless we were tasked with a project that required it.

1.5 What would I Do Differently

When I look back and reflect on my development process, I would perform the following changes if I were starting my project again:

- **Configure production settings from the get-go:** Multiple CSS and redirection bugs were caused by DEBUG = True being enabled. Setting up WhiteNoise, SECURE_PROXY_SSL_HEADER, ngrok, and a test settings file from the start of my project would have avoided days of debugging and allowed me to progress faster.
- **Write tests alongside features instead of at the end:** The tracking pixel problem was only caught during test writing. Writing tests for each stage immediately after I implemented it would have identified functional and design issues/flaws at the point where they would be easiest to fix, rather than when everything was implemented.
- **Design the cloned site injection system before making the individual pages:** The two-stage login flow broke the JavaScript injection because it was designed for simple pages that had one form for username and password. Had I known how the injection mechanism would work before building the pages, I would have used a styled <a> tag instead of a <button> element in my original design.
- **Adopt lazy imports from the start:** The Celery import causing the entire dashboard to be inaccessible was avoidable. If I had implemented a project convention of importing optional infrastructure within functions rather than at the module level, this bug could have been avoided entirely.

2. Technical Issues

2.1 Differences from earlier design and additional research required

The initial design I proposed in the project specification was much simpler: basic HTML email templates, a single generic landing page, and a simple click-and-open tracking log. During development, the scope of my project expanded considerably due to research into real-world phishing techniques and academic papers on the effectiveness of simulation.

The most significant design changes from my original specification are described below:

Two-Stage Login Flows

My original design used single-form login pages after researching how Google and Microsoft's actual sign-in interfaces work, which revealed I needed a two-step flow. The user enters their email on the first screen, clicks the "Next" button and then enters their password on the second screen. Using single-form pages makes it obvious to participants that the page is fake and outdated. Implementing the two-stage flow required designing a JavaScript state machine within the clone HTML to manage screen transitions, which unfortunately caused the submission interaction problem I described in Section 1.1.

Federated Sign-In Simulation

The federated sign-in flow was not in my original design. This came after I researched how threat actors target document signing platforms, with Docusign being the most commonly impersonated platform. After this, I decided to implement a federated login to make it feel more authentic by adding an option to sign in with Microsoft. This required creating a view called federated_landing to map provider names to landing paths and provide a mechanism for JavaScript injection, allowing users to click the fake Microsoft link and be directed to the fake Microsoft page while ensuring they are tracked under the original campaign.

Randomised Email Scheduling

The original design I proposed specified scheduled campaigns, but not delivery randomisation. Research into security team detection methods showed that the simultaneous delivery of all campaign emails is a detectable pattern, email server logs show a burst of outbound messages with identical structure and content, and many security operations centres flag this behaviour as a simulation. The Celery-backed randomised scheduling using `apply_async(eta=...)` was added to allow network admins to send emails in a campaign across a configurable time window, making the simulation more realistic.

Multi-Factor Risk Scoring

The original risk-scoring design I wanted to use was too simple: either a user clicked the link, or they didn't. Academic papers informed me of a more nuanced algorithm that incorporates time-to-click (where faster clicks indicate higher impulsivity), scenario difficulty (falling for a harder (more realistic) scenario is weighted more heavily), and positive behaviour such as reporting the email as phishing, which reduces a user's risk score.

Full Participant Anonymisation

After taking on feedback on my presentation, I dove deeper into GDPR requirements. After reviewing GDPR Article 5(1) (e) (storage limitation), which states that user information must be kept in an identifiable form for no longer than is necessary for the purposes for which it is processed. I decided to implement a system using auto-generated IDs in the form P-XXXXX, ensuring that nothing identifiable was retained. The email address and full name fields exist in the database for email dispatch but are never displayed in the dashboard, admin fields, or any report. This infrastructure decision was made after some development had taken place, but I was able to retrofit all existing dashboards and the admin panel.

Additional Technical Research

Additional research was required in the following areas that I didn't anticipate in my original design:

Django middleware order and the interaction between SecurityMiddleware, WhiteNoise and CorsMiddleware.

Celery task design patterns for graceful degradation with a running broker.

Browser event capture versus bubble phase propagation.

2.2 Difficulties in Implementing the Project

Landing Page HTML Injection System

The core mechanism for serving clone phishing pages required injecting form action URLs, CSRF tokens and JavaScript event handlers into arbitrary third-party HTML. This injection had to work across five different page structures, each with different form layouts, button types and inline JavaScript. The approach I used was to use regular expressions to find and modify form opening tags, which was fragile. Each new cloned page revealed new cases: forms with existing action attributes, forms without explicit method attributes, buttons that used onclick handlers rather than form submit events, and pages whose own JavaScript overwrote the form action attribute after page load. The solution required multiple layers of redundancy, HTML attribute injection, a JavaScript re-assertion of the form action on DOMContentLoaded and capture-phase event listeners on all button elements.

Session-Based Tracking Across Page Transitions

Maintaining tracking context across multiple HTTP requests required careful session management. The phishing flow needs at least three requests: the click-tracking redirect (which sets the session token), the landing page load (which reads the session token), and the form submission. Adding QR code access (no prior session), federated login navigation (changes to a different URL while keeping the same session), and the possibility of participants opening the email on a different device added a lot of complexity, requiring separate code parts for each scenario.

Realistic Page Reproduction

Producing cloned login pages that are convincing enough for a meaningful simulation required significant research into the visual design systems of major service providers. Small details such as border-radius values and the exact colours of logos on pages, all of which determined whether the page looked authentic or was

obviously fake. Pages that were clearly distinguishable from their real counterparts would undermine the simulation's validity.

Celery Integration with Graceful Fallback

The requirement to support environments both with and without Redis and Celery running — to allow researchers to use the tool on machines without Docker or a Redis installation — necessitated try/except fallback logic throughout the codebase wherever Celery tasks were called. Getting this pattern right without making the calling code unreadable required establishing a consistent convention of lazy imports and a clear fallback hierarchy (attempt async dispatch, fall back to synchronous dispatch, catch and log exceptions without crashing).

2.3 Issues with Tools and Software

Tool Name/ **Version**	**Issue Encountered**
Django 6.0.1	As a very recent major release at the time of development, several third-party packages had not yet published compatible versions. django-celery-beat required specific version pinning to avoid import errors. Django 6's changes to the default middleware ordering also required reviewing the WhiteNoise documentation to find the correct insertion point.
Celery 5.6.2 on Windows	Celery's default multiprocessing pool does not function reliably on Windows because Python spawns processes differently on that platform. The -P solo flag was required for the development worker, which restricts concurrency to a single process. This is adequate for a simulation tool but would not scale for production workloads.
ngrok (Free Tier)	The free tier of ngrok assigns a new random subdomain on every restart. Each restart required updating SITE_URL, ALLOWED_HOSTS, CSRF_TRUSTED_ORIGINS, and CORS_ALLOWED_ORIGINS in the .env file and restarting Django. This was a persistent friction point during iterative testing with external participants.
PostgreSQL via Docker on Windows	Running PostgreSQL in a Docker container on Windows Home occasionally caused permission errors with bind-mounted volumes. Switching from bind mounts to named Docker volumes resolved the issue, at the cost of making it slightly less straightforward to inspect the database files directly.
WhiteNoise Static Storage	The CompressedManifestStaticFilesStorage backend failed silently when it could not resolve all referenced static files in the manifest. The error only appeared at request time as a 500.

	Switching to CompressedStaticFilesStorage (without manifest) resolved the issue but removed cache-busting filename hashing.
pytest-django	Initially, all tests failed due to SSL redirect errors because the test runner loaded the main settings module, which set SECURE_SSL_REDIRECT = True. A dedicated phishingsim/test_settings.py module that inherits from the main settings but disables production-only security flags was required, along with a pytest.ini file specifying DJANGO_SETTINGS_MODULE = phishingsim.test_settings.

2.4 Testing Used to Assess Reliability

The project uses a pytest-based automated test suite comprising 65 tests organised across 15 test classes. All tests pass with a runtime of approximately three seconds. The test suite is divided into four areas:

Model Tests 14 tests

These tests verify the behaviour of the core database models. They confirm that Target objects automatically generate a unique, anonymous ID on creation, that the ID is not overwritten on subsequent saves, that risk scores increase correctly with click and submission events, that the risk-reduction-based reporting works as specified, and that the SimulatorConfig singleton pattern returns the same record on repeated calls.

Utility Function Tests: 15 tests.

These tests verify standalone utility functions independently of the HTTP layer. They confirm that generate_tracking_token produces 32-character lowercase hexadecimal strings and that 100 consecutive tokens are all unique. They verify that render_template_variables correctly substitutes all supported placeholders (name, first_name, company, email, and others) and leaves unknown placeholders unchanged. They verify that calculate_randomized_send_time returns a datetime that is later than the base time, within the specified window, and at least sixty seconds in the future. They also verify that calculate_risk_score returns zero for participants with no interaction history, increases for faster clicks and higher difficulty scenarios, and always returns a value within the 0–100 range.

Tracking View Tests — 24 tests

These are integration tests that exercise the full HTTP request/response cycle for all tracking endpoints through the Django test client. They confirm that track_open always returns a GIF with a 200 status code, even for invalid tokens; that track_click

sets the phishing_token session key, increments the target click counter, and creates an EventLog record; that track_form_submission records the submission, increments the target submission counter, redirects to the training page, and does not double-count if called twice; and that complete_training sets the training_completed flag and returns a 200 response. Invalid tokens are verified to return 404 for all endpoints except the tracking pixel.

Feature Tests 12 tests

These tests verify higher-level feature behaviour. The landing page tests confirm that serving a cloned site requires an active session token, that the served HTML contains the track/submit injection, and that direct QR code access works without a session. The federated login tests confirm that accessing the Microsoft federated URL without a session returns 404, that it serves the Microsoft page with an active session, and that an unknown provider name returns 404. The CSV import tests cover target creation from a valid CSV file, skipping rows with missing email addresses, updating existing targets rather than duplicating them, assigning imported targets to a specified group, rejecting non-CSV file types, and verifying that each imported participant receives a unique anonymous code.

The test infrastructure includes a pytest.ini configuration file specifying DJANGO_SETTINGS_MODULE = phishingsim.test_settings, a test_settings.py module that disables SECURE_SSL_REDIRECT and uses Django's fast MD5 password hasher to accelerate test runs, and a set of factory helper functions (make_template, make_target, make_campaign, make_send) that create standardised test fixtures with minimal boilerplate.

3. Repository and Code Review

The project repository contains no unnecessary files. The .gitignore correctly excludes compiled Python files (e.g.,__pycache__ and *.pyc), virtual environment directories (.venv), environment configuration (.env), media uploads, generated static files (staticfiles/), SQLite database files, and log files. All sensitive credentials are loaded from environment variables via python-dotenv; no secrets are committed.

The .env.example template is committed and provides clear documentation of every required environment variable. The LICENSE file correctly applies GNU General Public License v3, and the README.md provides comprehensive setup and deployment instructions for both Windows and Linux/macOS.

The migrations directory contains seven sequential migration files representing the complete database schema evolution. Each migration has a clear dependency chain. No migration files are missing or duplicated.

The following note is relevant for submission and deployment. The REST API is currently configured with rest_framework.permissions.AllowAny, which is appropriate for a lab environment, but must be changed before any production deployment. This is documented in the README under Security Notes. Additionally,

the dashboard views in views_dashboard.py do not yet carry @login_required decorators; this is the primary outstanding security item for production readiness.

All code follows consistent conventions: snake_case for functions and variables, docstrings on all model classes and utility functions, Google-style comments for complex logic blocks, and no hardcoded credentials anywhere in the codebase. The requirements.txt file pins all dependencies to specific versions for reproducibility.

4. Conclusion

PhishingSimulatorTool represents a substantial piece of software engineering work that combines several advanced topics: asynchronous task processing, session-based multi-step request flows, realistic HTML injection, ethical research design, and multi-service containerised deployment. The project delivered all seven core deliverables and two of three optional deliverables, backed by an automated test suite of 65 tests.

The problems encountered during development — the ngrok redirect loop, the CSS regression under DEBUG = False, the Celery import crash, the form submission response mismatch, the QR code session gap, the two-stage login event propagation conflict, and the tracking pixel information leak — were each non-trivial bugs that required genuine debugging skill and an understanding of how multiple system components interact. Each was resolved, and each provided a lesson that I can apply to future projects.

If there is a single overarching lesson from this project, it is the importance of testing in realistic conditions from the earliest stages of development. The majority of the most time-consuming bugs arose because the development environment (DEBUG = True, no Redis, no reverse proxy) silently masked problems that only appeared under production-like conditions. Future projects will establish a production-equivalent configuration from day one and treat the first passing test as the beginning of development, not its conclusion.

The finished platform is a credible research tool. It is capable of running multi-scenario phishing campaigns with realistic landing pages, tracking participant interactions at a fine-grained level, scoring and categorising participants by risk level, delivering immediate training after every simulated compromise, and generating anonymised reports suitable for academic analysis — all within a framework that respects participant consent and GDPR data handling requirements.

References

1. Canham, M., Posey, C., Strickland, D. and Constantino, M. (2021). Phishing for Long Tails: Examining Organisational Repeat Clickers and Protective Stewards. SAGE Open, 11(1). Available at: https://journals.sagepub.com/doi/full/10.1177/21582440219906 56 [Accessed 9 December 2025].
2. CREST (2024). Phishing your staff: A double-edged sword? Guide to ethical simulated phishing. Centre for Research and Education on Security and Trust. Available at: https://crestresearch.ac.uk/resources/phishing-your-staff/ [Accessed 9 December 2025].
3. Django Software Foundation (2024). Django documentation, version 6.0. Available at: https://docs.djangoproject.com/en/6.0/ [Accessed throughout 2025–2026].
4. European Data Protection Board (2024). Guidelines 1/2024 on processing of personal data based on legitimate interest (Article 6(1)(f) GDPR). Publications Office of the European Union. Available at: https://www.edpb.europa.eu/system/files/2024-10/edpb_guidelines_202401_legitimateinterest_en.pdf [Accessed 9 December 2025].
5. Hatfield, J.M. (2019). Virtuous human hacking: The ethics of social engineering in penetration-testing. Computers and Security, 83, pp. 354–366. Available at: https://www.sciencedirect.com/science/article/abs/pii/S016740481831174X [Accessed 9 December 2025].
6. IBM (2025). Cost of a Data Breach Report 2025. Available at: https://www.ibm.com/reports/data-breach [Accessed 2 December 2025].
7. NIH Office of Science Policy (2024). Informed Consent for Research Using Digital Health Technologies. U.S. Department of Health and Human Services. Available at: https://osp.od.nih.gov/wp-content/uploads/2024/05/DigitalHealthResource_Final.pdf [Accessed 9 December 2025].
8. Tóth, R., Dubniczky, R.A., Limonova, O. and Tihanyi, N. (2024). Sustaining Cyber Awareness: The Long-Term Impact of Continuous Phishing Training

and Emotional Triggers. arXiv Preprint, 2510.27298v1. Available at: https://arxiv.org/html/2510.27298v1 [Accessed 9 December 2025].

9. WhiteNoise Project (2024). WhiteNoise documentation: Radically simplified static file serving for Python web apps. Available at: https://whitenoise.readthedocs.io/en/stable/ [Accessed throughout 2025–2026].