Google
Summer of Code

Extend FLOSS to use the rendering techniques pioneered by QUANTUMSTRAND

Lakshit Verma (vee1e)

March 30, 2026

Contents

Introduction

About Me

Name & GitHub Handle	Lakshit Verma (vee1e)
University	Manipal Institute of Technology
Program	B.Tech in Electrical Engineering (Minor in Computing)
Year	Third Year
Graduation	July 2027
Contact Information	+91 8448058867 / vermalucky2004@gmail.com
Location / Time Zone	Mumbai, India / IST (UTC+5:30)
Brief CV Link	**https://lverma.com/resume**

Background and Interests

What is your background and which current interests do you have?

I'm Lakshit Verma, an engineering student with a cumulative 2.5 years of experience in the cybersecurity field, mainly through playing Capture The Flag (CTF) tournaments in the reverse engineering and digital forensics domain with my academic team **Cryptonite**, ranked as the #3 Indian CTF team for two consecutive years. Currently I also serve as a member of the board and the Forensics domain head. I also possess an open source program background, having been selected for the highly selective **C4GT Dedicated Mentoring Program (DMP)** last year. It's format is similar to GSoC, occurring over the summer from July to August, a mid-term and end-term evaluation, a mentor guiding you along and reviewing code, and a stipend equivalent to a small-medium GSoC project. You can see my project from that program **here**; it's a tool to convert XLSForm files to the organization's internal JSON format. I built the application from scratch, and worked on the Python/FastAPI backend, the Angular JS frontend and the CI/CD GitHub workflows for it.

I have been attended several on-site CTFs, hackathons and security conferences, and won several cybersecurity competitions such as the **Smart India Hackathon 2024**, India's largest hackathon, under a cybersecurity problem statement. In it, our team developed a novel parsing methodology to recover deleted data from Linux Btrfs and XFS filesystems, processing raw filesystem images from scratch (from the superblock to individual inodes) with no external libraries included.

I have a strong interest in the FLARE team's tooling for malware analysis and forensics as I've integrated them more in my workflows over the past months, and security tool development is a skill I've been building for the past year through OSS contributions and personal projects.

Do you have any background or experience in reverse engineering or malware analysis? If not, why are you interested in learning more about it?

I have a good amount of experience in reverse engineering, with knowledge and experience of all major decompilers such as Ghidra, IDA Pro, Binary Ninja and Cutter, and supporting tools such as GDB/pwndbg, Speakeasy and WinDbg.

Over the past few years I've designed multiple CTF challenges in the reverse engineering domain, presenting several obfuscation techniques such as a custom VM based obfuscation in niteCTF 2024 (**writeup link**) and function hashing and Rust malware in a 3-part incident response series in niteCTF 2025 (**writeup link**).

I also possess a good degree of knowledge in the domain of string obfuscation techniques such as runtime decryption, dynamic stack construction, and single and multi-byte XOR encryption, having dealt with the same through an untold amount of crackme's and reverse engineering challenges over the years.

As mentors and project coordinators, how can we best support you on your open source journey?

A mentor's job isn't to simply review code, its to look at the bigger picture of the project, guide the contributor in the right direction, and help them grow as a developer and OSS contributor: enough so that they can not only contribute to but maintain project(s) in the future as well.

My previous experience at C4GT, while insightful, was characterized by mentor apathy and the lack of clear communication and guidance, leading me to work more independently in it. Fortunately I've found the FLARE team and commmunity to be much more available and receptive to newer contributors like me.

Timely, meaningful interactions of mentor and contributor through every step of the way form the backbone of a successful GSoC project, and I hope to have that with you all this summer.

Code Contribution

I have actively contributed to the Mandiant organization over the past three months, with a total of **11 merged PRs** and **2 open PRs** across four repositories: **flare-floss**, **flare-floss-testfiles**, **macOS-UnifiedLogs** and **stringsifter**. One such contribution is this:

PR #1221: QS: Add Mach-O parsing and samples (`+566 -1`)

Wherein I added support for Mach-O structure parsing in QS, which was a major feature addition that resolved a nearly 3 year old issue (#**786**) and significantly enhanced the tool's capabilities.

Project Information

Project Abstract

QUANTUMSTRAND (QS) is an experiment within FLOSS augmenting traditional output of decoded and deobfuscated strings with context-aware rendering of string metadata to aid in malware analysis, such as the string's location in the binary section, its association with common libraries or APIs through the tagging system. It also contains a GUI viewer providing interactive filtering and visualization of the rendered string data.

With FLOSS being the main tool for string analysis, the end goal of the project is to select the best features from QS and integrate and merge them into the `master` branch to make them available to the wider reverse engineering community through FLOSS itself. Changes will be made both on the backend and frontend; a new GUI based on the QS viewer will be developed and integrated into the branch as well.

This will involve preliminary research by users of both FLOSS and QS to determine the use cases and features to prioritize, the improvement of the QS codebase to complete some features and get it to a mergeable state, and the actual integration of the features into FLOSS itself.

Project Scope

In Scope

- Improve current QS features to be merge-ready into upstream `master`.
- Implement missing features in QS needed by FLOSS users and developers the most.
- Merge the essential features through Incremental Development.
- Build a GUI to view extracted FLOSS + QS enhanced strings.
- Document such that future developers can build upon the project post GSoC.

Not in Scope

- Large, sweeping changes to the QS codebase and their subsequent merging into FLOSS.

  This includes, but is not limited to: new language specific string extraction algorithms, mergers with other FLARE projects such as capa/capa-viewer, and large refactors of the codebase.

Detailed description

The project will be split into three distinct phases:

- **Improvement of current QS features:** I'm currently working on this with PRs merged such as #**1225**, #**1229** and #**1230**. Completion of issues, both smaller ones like #**771** and larger scale ones such as #**701** and tag database construction will be

the focus of this phase, to ensure the QS codebase is in a workable state and has all the necessary features for integration into FLOSS.

- **Integration of QS features into FLOSS:** This will be the main coding phase where we finalize the features through discussions with the community and the mentor to decide on the ones of priority, and then start slowly introducing them one by one, testing for issues, collecting feedback and repeating from there.
- **Creating an interactive GUI to display strings:** Extend the currently bare-bones GUI in `quantumstrand/qs-viewer` to a feature complete and FLOSS output interface with all the features malware analysts and the FLOSS community expect from such a tool.
- **Documentation:** Currently QS documentation is next to nonexistent, represented only by a few `readme.md` files scattered across root and in the database folders. This creates a serious problem if and when FLOSS adds new QS features. The technical documentation both will be worked on to ensure a smooth continuation of development of the features post-GSoC by other contributors.

Improvement of current QS features

Currently QS supports the following features:

- Display strings with appropriate tags next to them such as `#winapi` for Windows API functions, `#common` for common strings, etc. through an in-repo database.
- Render strings with their appropriate PE and Mach-O sections, such as `.rdata` and `__LINKEDIT`.
- Annotation of strings if their section is known, such as `import table`, `export table`, etc.
- Muting and filtering of strings based on tags.

All of which are what I would consider *highly useful* for malware string analysis in FLOSS. Please see the next section for more details on how they will be merged.

User Feedback

I asked Daniel, the **Malpedia FLOSSed** creator, for his thoughts on the GSoC project and what would be most useful in practice. His response highlighted a few key points:

> *Best to my knowledge, string parsing for Go/Rust/.NET has been in the main FLOSS branch for one or more minor versions. Section parsing and the simple decodings proposed by you are listed as features in the release log of the QS beta branch, so this may mostly be a port into production/main branch FLOSS. For memory-mapped and dumped malware, section recognition does not work as well, and some solid string validation/rating for junk filtering would be highly valuable.*

The section parsing issues are most relevant here: a sample from VT is generally properly compliant with the PE standard and has `FileAlignment`, but an executable dumped from memory uses `SectionAlignment` with slack space and heap data. Extra padding that tools like `binwalk` might include due to imperfect heuristics can also further hamper this.

The takeaway from this is that we should have a consideration for the source of a sample too and whether it was taken from a static environment (like a AD1 filesystem dump) or a live one (memory/crash dump). When section parsing makes its way to FLOSS we should conduct extensive testing using both kinds of executables to cover all edge cases.

New features to be added

- **Visualize the embedded string databases better (Medium Priority)**

  Refer to discussions in #**782** and #**784**. For analysts we can easily imagine a scenario where they see a string tagged and want to explore the tag itself. This is evidenced through a comment by **cxiao** where he says:

  > *I really like the tags for strings which appear in different databases (e.g. #msvc, #winapi, but I find myself almost immediately wanting to open up and peek inside the databases when I see that. One reason why I feel like I want to do this is because there may be ambiguity in the tag name — for example, it's not clear what the tag #common is referring to, and I feel like I would get a better idea of what kinds of strings are considered "common" if I could look inside the database and see what else is classified under that tag.*

  Further details are in the GUI viewer section.

- **Add ELF section parsing compatibility (Medium Priority)**

  From @FredCoast's comments in #**785** and checking out his **repository fork**, there has been significant progress on this done already, with just a few TODO's left such as implementing the `.dynamic` sections. FLOSS already supports PE and limited ELF binaries, so the porting of section layouts should not take significant effort to do.

- **Automatic tag database construction pipeline (Medium-High Priority)**

  The largest portion of the improvement stage, and greatly help in keeping #oss up-to-date so that tagging of library strings can work as intended without the risk of false negatives due to outdated DBs as evidenced in #**768**.

  Before this its imperative we migrate the database .gz files over to Git LFS as these builds will take up considerable space in Git history if treated as plain binary files.

  For this, I propose a recurring CI pipeline that can execute through a cronjob and build string databases for commonly encountered OSS libraries.

  Library Manifest: Currently there are 17 databases of library strings present and have not been updated since their initial commit. This will be expanded to include

at least the top 25 common libraries. The sorting process to find said top 25 libraries will include the processing of thousands of samples from sources like VT and vx-underground. For dynamically linked samples we can run a simple script for parsing the `DT_NEEDED` section; for statically linked ones we can implement FLIRT signature matching. This is assuming the worst case scenario that all or most samples have been stripped (a reasonable assumption to make if we're working with malware).

Build Matrix: `vcpkg` will create not just a single .lib file but multiple for different architectures and operating systems to ensure maximum coverage of all strings, including those that might show up on one architecture/OS but not another.

Basic structure of `build-db.yaml`:

```yaml
on:
  schedule:
    - cron: "0 5 * * 0"    # Runs weekly every Sunday 05:00 UTC
  workflow_dispatch:

jobs:
  build:
    name: ${{ matrix.os }} | ${{ matrix.triplet }} | ${{ matrix.compiler }}
    runs-on: ${{ matrix.os }}

    strategy:
      fail-fast: false
      matrix:
        include:
          - os: windows-latest
          - arch: x64
          - triplet: x64-windows-static
          - compiler: msvc143

          # --- Linux x64 ---
          . . .
```

```
vcpkg install ${{ matrix.lib }} --triplet ${{ matrix.triplet }}
```

jh extract and conversion to QS format: These will remain similar to the current workflow. To avoid wasting valuable GitHub CI time and resources, `jh_to_qs.py` will be audited for performance bottlenecks that might have been overlooked when it was written to run only once.

Dedupe and merge: This will ensure that redundant string write operations on the DB do not take place and that only new strings are added. This is also an unlikely

addition to impact performance significantly; local tests ran all of `openssl.jsonl.gz` in $\leq$ 10ms.

Publish Release: Finally the updated databases will be tagged and if significant additions happen, may trigger a new release to ensure users who download direct FLOSS binaries will have the same data as the ones who build from source.

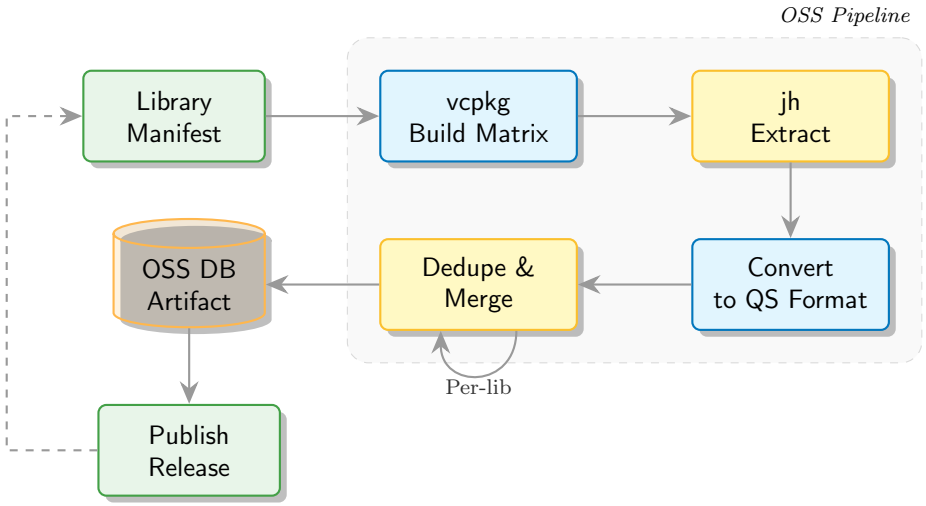


Figure 1: Automatic tag database construction pipeline for OSS libraries.

Deliverables: A merge ready QS codebase with both small UX improvements and a stable, efficient pipeline to update string tagging databases, specifcally the `#oss` database.

Integration of QS features into FLOSS

I propose a slice-based incremental method to slowly introduce the features without causing any breaking changes in `master`. We will create simple classes and methods initially, then slowly increase their function as the features become more expansive.

Slice 1

- Create `floss/qs_integration.py` for modularity.
- Add an additional argument, `-quantum (default: False)` to toggle QS techniques.
- Add helper functions to translate between a `ResultDocument` of QS and FLOSS.
- (PE only) map the FLOSS strings to sections.

This first slice will serve as the baseline for future work, and will undoubtedly introduce many bugs. Its vital that we test extensively here so that we don't face bigger challenges down the line.

Slice 2

- Add helpers to load tags and apply them to FLOSS `Tagger` $\rightarrow$ `QsTagResult` $\rightarrow$ `Merge`

`into ResultDocument`.

- Copy the CI ymls from quantumstrand to master.
- Import all current DBs (#expert, #oss, #common, #winapi, etc.)
- Add tag filters (mute/hide/highlight) as CLI arguments.

This will take care of all tagging features and will establish the CI pipeline working along FLOSS's `build-db.yml`.

Slice 3

- Port the `Structures` aspect of `compute_pe_layout` into `qs_integration.py`.
- Begin adding ELF parser from Fred's fork through the `elffile` library.
- Add `StaticString`-only support for Mach-O executables.
- Compile a diverse corpus of test executables with varying properties such as stripped symbols, memory dumped (see User Feedback section for why), broken/padded samples, unusual malware samples, etc.

This is a all-in-one slice that introduces lots of smaller new features into various parts of the codebase. The eventual result will be a generic function accepting multiple sources, such as `analyze_qs_layout(..., formats=("pe","elf","macho"))`

Slice 4

This is meant as a buffer slice to work on smaller features like:

- Discussing with the community and collecting general feedback of the previous slices.
- Finalizing whether to support more niche features such as shellcode support and language-specific tagging.
- Bug fixes, security patches, optimizations for previous code that was overlooked to just get it ported, etc.
- Begin writing documentation based on discussions with mentors (see the Documentation section for more details).

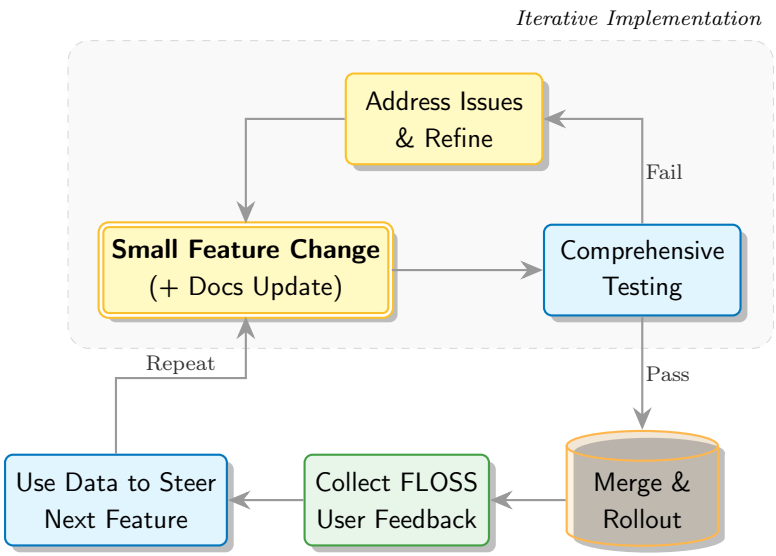


Figure 2: Slice-based incremental method and community feedback loop.

Merged FLOSS + QS JSON

To keep this explicit and testable, here is an example of the merged output. When the `quantum` flag is enabled, the output JSON will follow this shape, with existing FLOSS fields preserved and optional `qs_*` enrichments added. `qs_layout` stores structural hierarchy only; per-string QS metadata (`qs_tags`, `qs_section`, etc) is stored on each FLOSS string object.

```
{
  "analysis": {
    "enable_static_strings": true
  },
  "metadata": {
    "file_path": "dotnet-hello.exe",
    "qs_enabled": true
  },
  "strings": {
    "static_strings": [
      {
        "string": "!This program cannot be run in DOS mode.",
        "offset": 77,
        "encoding": "ASCII",
        "qs_tags": ["#common"],
        "qs_structure": "header",
        "qs_section": "header"
      },
      {
        "string": "WriteLine",
```

```
        "offset": 1036,
        "encoding": "ASCII",
        "qs_tags": ["#common", "#msvc"],
        "qs_structure": "",
        "qs_section": ".text"
      }
    ]
  },
  "qs_layout": {
    "name": "pe",
    "offset": 0,
    "length": 3584,
    "children": [
      {
        "name": "header",
        "offset": 0,
        "length": 512
      },
      {
        "name": ".text",
        "offset": 512,
        "length": 1024
      }
    ]
  }
}
```


In practice, this merged format means:

- Legacy FLOSS JSON tooling remains valid because the core roots are unchanged: `analysis`, `metadata`, `strings`.
- QS mode is indicated in metadata (e.g., `qs_enabled: true`).
- Tags/structure/section are cleanly attached to each string via optional `qs_tags`, `qs_structure`, and `qs_section`.
- `qs_layout` provides the recursive section/resource tree for grouping and navigation in the viewer.

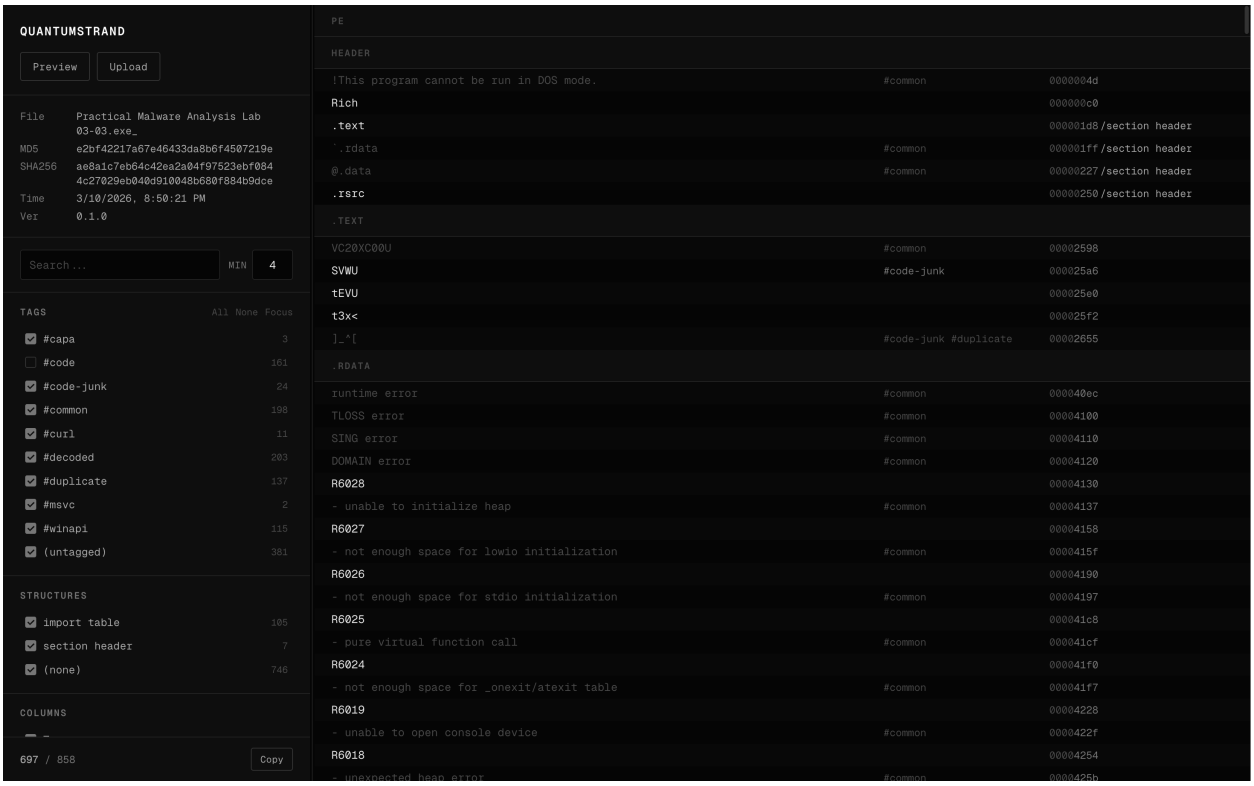
Final Directory Structure of FLOSS

```
flare-floss/
├── floss/
│   ├── main.py ..........................................adds the -quantum flag
│   ├── results.py .......................optional qs_* fields (backward compatible)
│   ├── qs_integration.py ..................new adapter entrypoints (layout + tags)
│   ├── qs_types.py ..................new lightweight QS-compatible dataclasses/types
│   ├── render/
│   │   └── json.py ..........unchanged behavior, includes optional qs fields when present
│   └── ...
├── floss-viewer/ ..................FLOSS string viewer GUI ported from qs-viewer
├── tests/
│   ├── test_qs_integration_cli.py ..............................flag behavior
│   ├── test_qs_layout_mapping.py ............................PE first, then ELF
│   ├── test_qs_tagging.py ....................winapi/common/OSS/expert tagging
│   ├── test_qs_visibility.py ........................... hide/mute/highlight rules
│   └── test_results_schema_compat.py ............old/new JSON load compatibility
├── doc/
│   └── quantumstrand.md ............................... architecture + rollout notes
└── .github/workflows/
    ├── (existing CI + QS integration test jobs)
    └── build-db.yml
```

Deliverables: A merged `master` branch integrating core QS features (section mapping, tagging, filtering) with existing FLOSS functionality, backed by comprehensive tests. It also includes a checked-in JSON Schema and compatibility tests ensuring seamless GUI ingestion of both legacy and QS-augmented formats.

Creating an interactive GUI to display strings

Currently the baseline GUI for QS which I've improved is below:



The link to my branch is here: **qs-viewer-enhance**. It contains a sidebar for tag filtering, sample metadata and other interactive options. The right 80% is purely for viewing strings, with the output translated 1-to-1 from the CLI QS output.

The current features it supports are limited:

- Filtering of tags and tag categories ("All", "None", "Focus").
- Exact, linear searching of strings by content or length.
- Viewing of the sample metadata such as name, path, hashes, and date/time.
- Copying of selected and filtered strings to clipboard.

I propose a few crucial improvements and required features to add to this as it becomes FLOSS-native:

- Merge FLOSS and QS output schema. Refer to **Merged FLOSS + QS JSON** above.
- Then we add the string metadata shown in native FLOSS CLI output. This is a simplified example output FLOSS produces:

```
FLARE FLOSS RESULTS (version 3.1.1)
+-------------------------+-----------------------------+
| file path               | dotnet-hello.exe            |
```

```
| identified language    | dotnet (version unknown)  |
| static strings         | 57 (1028 characters)      |
+------------------------+---------------------------+
FLOSS Static Strings: ASCII and UTF-16 LE
Other FLOSS strings, such as stack, tight and decoded
```

Strings will be sorted section-wise as in the example, but with a clear indicator of whether they're static, stack, tight or decoded, and also language-based. The sidebar will expand to include FLOSS exclusive metadata such as identified language and number of extracted strings of each type.

- For QS specific enhancements, there are several good ideas already given in the **QS/ui** label. Some such as consolidation of adjacent tags have been accomplished in the CLI already and can be ported to the GUI.
- Implement other QoL enhancements such as custom tagging, selection of strings, and exporting the filtered/selected JSON.
- Add fuzzy search for the strings using an appropriate library such as Fuse.JS.
- Create features for tagging metadata. Hovering over a tag will give you more information pertaining to what the tag db actually does, what other strings it consists of, and in cases like #expert, CAPA-aware metadata such as the rule name and the ATT&CK technique associated with it. As the DBs are inside the repository itself, performance and latency won't be a major issue.
- After implementing these features, move to addressing performance bottlenecks. Loading a JSON with over 100,000 strings greatly impacts the performance in load times, filtering and general scrolling and viewing. For this we'll implement optimizations such as lazy loading and virtual scrolling.
- Deploy this to a live website so that a user doesn't need to install the entire FLOSS suite to just view data that he already has. Since this consists of only a frontend, GitHub pages is the most suitable option.

The current `qs-viewer` is a React app with Vite as a bundler. I will keep the current stack as-is, but move the runtime from NPM to Bun for faster builds.

As GUIs are subjective by their very nature, this will require a great deal of feedback to get right and will inevitably lead to conflicts. Listening to all stakeholders (contributors, mentors, and community) is significant here.

This frontend is organized into three simple, modular layers, **Ingestion**, **Data** and **Presentation**, all in `floss-viewer/`.

The ingestion step loads the FLOSS JSON, validates it, and normalizes missing fields into one consistent `ViewerDocument` structure; this will be best split across modules such as

`ingest/loadFile.ts`, `validateFlossDocument.ts`, and `normalizeDocument.ts`.

From there, the main state, search and filter parts stay separate from rendering. The `Viewer` state holds search text, selected tags, structures, types, and layout toggles, while lazy loading and indexed lookups compute the active result set efficiently for large files.

The frontend consumes the output given by FLOSS JSON in `floss-viewer/` to render the sidebar, grouped string list, and export actions. This keeps loading, filtering, and rendering clearly separated and makes the viewer easier to extend without coupling UI components to FLOSS internals.

Unit and Integration Tests will be written to validate all features such as normalization, search, filter, and a complete pipeline from JSON to export/copy.

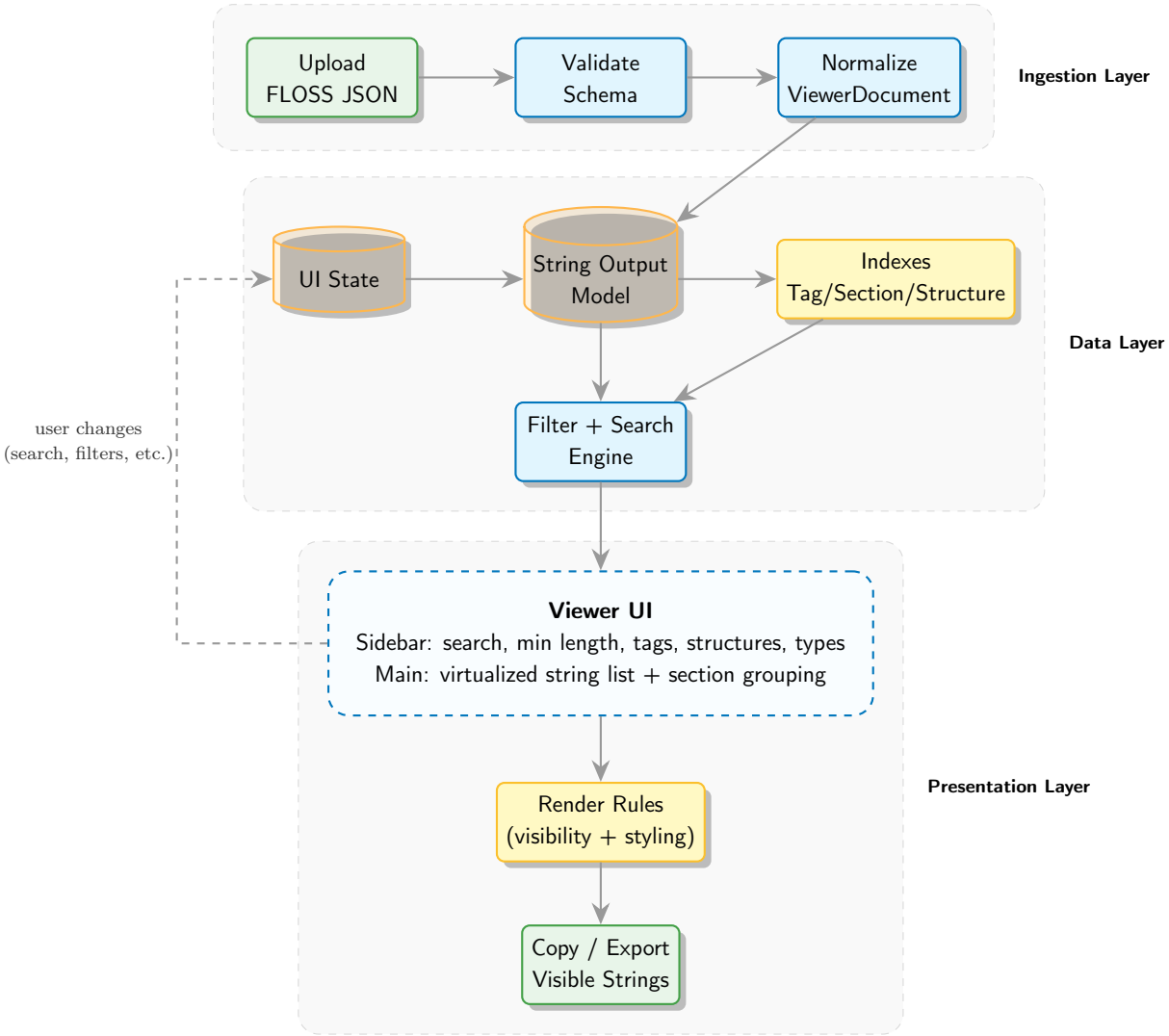


Figure 3: Frontend viewer architecture

```
floss-viewer/
├── package.json ..................... frontend dependencies and scripts
├── vite.config.ts ..................... Vite config
├── index.html ..................... app entrypoint
├── src/
│   ├── main.tsx ..................... bootstrap
│   ├── App.tsx ..................... top-level app layout
│   ├── ingest/ ..................... file loading, validation, normalization
│   ├── model/ ..................... canonical viewer document types
│   ├── state/ ..................... viewer state and selectors
│   ├── query/ ..................... indexes, filtering, search
│   ├── components/ ..................... sidebar and viewer components
│   ├── render/ ..................... visibility and style rules
│   ├── export/ ..................... copy and export helpers
│   └── styles/ ..................... shared viewer styling
└── tests/ ..................... all testing infra
```

Figure 4: Planned high-level directory structure for the FLOSS viewer frontend.

Deliverables: An intuitive, full-featured and well designed deployed GUI interface that allows exploration of any aspect of the strings exposed by FLOSS and QS.

Documentation

The final but equally important part. For this we'll use a single file `doc/quantumstrand.md`.

This will include the following details

- The theory and philosophy behind QS and why it serves crucial part of malware string analysis, essentially the `theory.md`.
- Various features of QS, and what methods and APIs are exposed for it, and how to extend features or create new ones.
- Reports of the various libraries used and the algorithms used to process the data generated from them (particularly for string tagging).
- A technical description of the GUI along with screenshots demonstrating the features **(will go inside the main FLOSS `usage.md`)**

Parallel to this, I will also work on a blog post detailing my GSoC journey to give the perspective of a student QS developer and would serve as a good reference for future contributors.

Deliverables: A comprehensive technical document detailing the important parts needed for further development of the project.

Weekly timeline

The work is divided into four phases:

Phase 1 covers improving existing QS functionality and stabilizing missing pieces.

Phase 2 focuses on integrating those capabilities into FLOSS.

Phase 3 is dedicated to GUI development in `floss-viewer/`.

Phase 4 is reserved for documentation, cleanup, and final polish.

Period	Phase	Description	Deliverable
Community Bonding (May 1 – May 24)	Prep	Deepen FLOSS codebase knowledge, study QS and FLOSS string decoding algorithms in detail, align with mentors and the community on project priorities, and fix any remaining setup or CI issues in the repo by merging 1-2 PRs before coding starts.	Finalized project plan and ready-to-code development environment.
Week 1 (May 25 – May 31)	Phase 1	Start work on ELF and Mach-O section support in QS, focusing on layout handling and the initial parsing needed for both formats.	Initial ELF and Mach-O section support delivered with matching tests or validation samples.
Week 2 (Jun 1 – Jun 7)	Phase 1	Migrate the string databases to Git LFS and perform local testing of the library build pipelines to validate the planned OSS database workflow before CI rollout.	Git LFS migration completed and local pipeline test results documented.

Period	Phase	Description	Deliverable
Week 3 (Jun 8 – Jun 14)	Phase 1	Implement the first part of the CI pipeline for automated library builds, including workflow structure, build matrix setup, and artifact generation.	Initial CI workflow for library builds pushed and running.
Week 4 (Jun 15 – Jun 21)	Phase 1	Complete and refine the CI automation by adding database conversion, dedupe/merge logic, and the remaining release-oriented workflow pieces.	End-to-end CI-driven database pipeline ready for integration and review.
Week 5 (Jun 22 – Jun 28)	Phase 2	Complete Slice 1 and the first tagging step of Slice 2 by adding `qs_integration.py`, the `-quantum` flag, result translation helpers, PE section mapping, and the initial tag merge path.	Baseline FLOSS integration with initial tagging support.
Week 6 (Jun 29 – Jul 5)	Phase 2	Import the current string databases, add CLI tag filters for mute/hide/highlight behavior, and finish the tagging feature set against FLOSS CI.	Complete tagging integration validated by `build-db.yml`.
Midterm Evaluation (July 10, 2026)			
Week 7 (Jul 6 – Jul 12)	Phase 2	Complete Slice 3 by adding structure support, beginning ELF parsing through `elffile`, adding scoped Mach-O support, and building a stronger executable test corpus.	Expanded multi-format integration with better structure coverage and tests.
Week 8 (Jul 13 – Jul 19)	Phase 2	Use this week to collect feedback, clean up rough edges, and make final decisions on what should or should not be included in scope.	Stable and reviewable QS integration phase.
Week 9 (Jul 20 – Jul 26)	Phase 3	Begin the GUI phase by setting up the viewer structure and migrating the frontend from QS to FLOSS output JSON schema.	Working frontend foundation with FLOSS JSON support.
Week 10 (Jul 27 – Aug 2)	Phase 3	Implement the main viewer interface, including sidebar controls, grouped rendering, virtualized string lists, and core search/filter behavior.	Usable viewer MVP with core interaction flows.
Week 11 (Aug 3 – Aug 9)	Phase 3	Finish advanced GUI features such as fuzzy search, richer tag metadata interaction, copy/export actions, and performance tuning for large JSON inputs.	Feature-complete GUI ready for final polish.
Week 12 (Aug 10 – Aug 16)	Phase 4	Complete `doc/quantumstrand.md`, usage-oriented documentation, screenshots, final cleanup, and any remaining polish before submission.	Final documentation and submission-ready project state.
Final Submission (August 24, 2026)			

Future Plans

If I am selected for the program, I would like to continue contributing to the FLOSS project post-GSoC by focusing on the long-term maintainability and expansion of FLOSS. My post-program roadmap includes creating comprehensive Doxygen documentation for the entire FLOSS codebase to streamline contributor onboarding, researching language-specific string extraction heuristics for niche languages like Nim and Zig to keep pace with evolving malware trends, and conducting a feasibility study into migrating the disassembly backend from Vivisect to alternatives such as Capstone or `idalib` for improved performance.

Availability

I am almost entirely free for the entire duration of the program, from Week 1 through the final evaluation period. I have no other academic, internship, or professional commitments during this time and can dedicate 20–35 hours per week consistently to this project, totaling approximately 350 hours across the coding period (12 weeks), which matches the expected project effort.

I have my semester final exams from May 2 – May 19 during the Community Bonding period, however I will remain active in the Org regardless.

My regular working window will be in the evening to late night, typically 18:00–00:00 IST (13:30–19:30 CET), giving me a good 5–6 hours a day for collaboration, implementation, and mentor feedback.

AI Disclosure

AI was used to proofread the text and fix grammatical errors and issues. It was also used in refining the diagrams and table formatting. Besides this, all text content and code is entirely human-typed, and I take full responsibility for any code I write. This is fully in line with Mandiant's **AI tooling guidance** policy.