Analyzing the Qubes code base

Candidate: Paras Chetal
B.Tech. Computer Science and Engineering,
IIT Roorkee
Email: paras.chetal@gmail.com
IRC Nick: feignix

Mentor: Jean-Philippe Ouellet

Introduction

The Qubes ecosystem comprises of many individual components and being a secure operating system, there is a need to ensure the security of each of these individual components and the interfaces between these components (a chain is only as strong as its weakest link). Manually analyzing each and every line of code and the system in its entirety is not feasible. Even then, bugs can easily escape typical code review.

I propose an automated solution to analyze the security of the Qubes code base on an ongoing basis, as the code evolves.

Project goals

This project will aim to lay the groundwork for automated analysis of the Qubes code base. This includes:

1. Static analysis: Creating a module for static analysis of the Qubes components.
2. Symbolic execution: Symbolically analyzing how untrusted data moves through the code.
3. Dynamic analysis: Integrating fuzzers with the components.

Implementation

Static analysis

General static analysis

Static analysis, among other things, will involve building the component code under a static analyzer(s), which will automatically find generic bugs in the code before packaging. Hopefully, we won't have many of these, but the inclusion of traditional static analysis is quite important for the robustness of our code. In order to implement this, the best way would be to integrate the qubes-builder with the static analyzers for the languages in which the code has been written for each component individually.

The following tools will come in handy while performing these static checks on the components before packaging them for the build process:

- Scan-build and Flawfinder (for the components coded in C):
  Clang's scan-build would enable the static analysis as a part of performing a regular build and packaging. This tool will perform these available checks on the component code: https://clang-analyzer.llvm.org/available_checks.html .
  Flawfinder will scan the C code and report any evident security weaknesses.
- Pylint and bandit (for the components coded in python):
  Bandit will be used to find common security issues in the python code.
  Pylint would perform these basic checks:
  http://pylint-messages.wikidot.com/all-codes.
- ShellCheck (for the components coded in bash): ShellCheck would be used for static checks on the bash based components. This tool will perform the following checks: https://github.com/koalaman/shellcheck/blob/master/README.md#gallery-of-bad-code on the code.

While the bugs which this generic static analysis reveals might not be high risk, they ought to be removed for keeping our code base secure and tidy. Many of the bugs might indicate how some implementation does work, but could be done in a better way. I plan to fix all of these bugs as I discover them.

The generic qubes-static-analysis module will have bash scripts which will allow the user to analyze any of the Qubes component (possibly specified through command line arguments) using the tools mentioned above. Integrating this with the build process would involve modifying the qubes-builder by adding some hook scripts to the qubes-builder/scripts directory along with some edits to the regular build scripts and configuration files to allow this analysis.

Custom static analysis

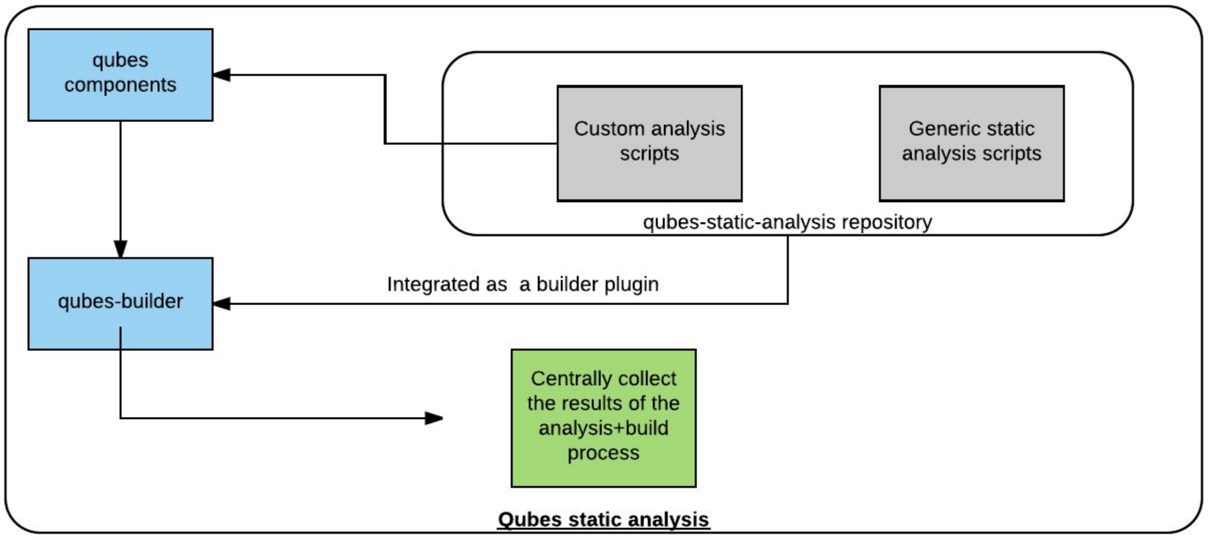
The custom static analysis part would involve analyzing how the untrusted_* values change and that there is no deviation from the security guidelines that the program is supposed to adhere to.

For the C based components, an excellent tool: Frama-C (along with its plugins) exists which I intend to use to perform this custom analysis. It would allow us to analyze the following:

1. Observing all the statements where the untrusted_* variables have been used, along with observing the untrusted values and how they will change at each point in the code. This will be accomplished through the Evolved Value Analysis and the Variable occurrence plugins of Frama-C.
2. Navigate the data flow of the program (we'll be mainly concerned with the untrusted data) using the Scope and Data flow browsing plugin.

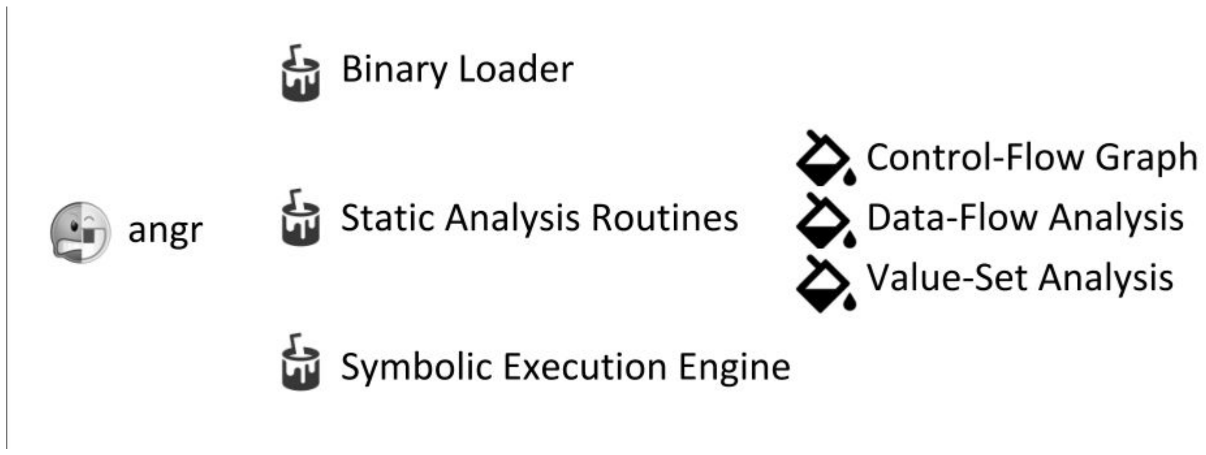
The analysis specification for Frama-C will be done in the ACSL language.

For the components which have not been coded in C, I am still looking for some good tools that allow such extensive custom static analysis. For now, I intend to explore pylint (possibly) for python and shellcheck (most likely not) for bash and see if they can be used here too.


qubes components
Custom analysis scripts
Generic static analysis scripts
qubes-static-analysis repository
Integrated as a builder plugin
qubes-builder
Centrally collect the results of the analysis+build process
Qubes static analysis

Static and dynamic symbolic execution

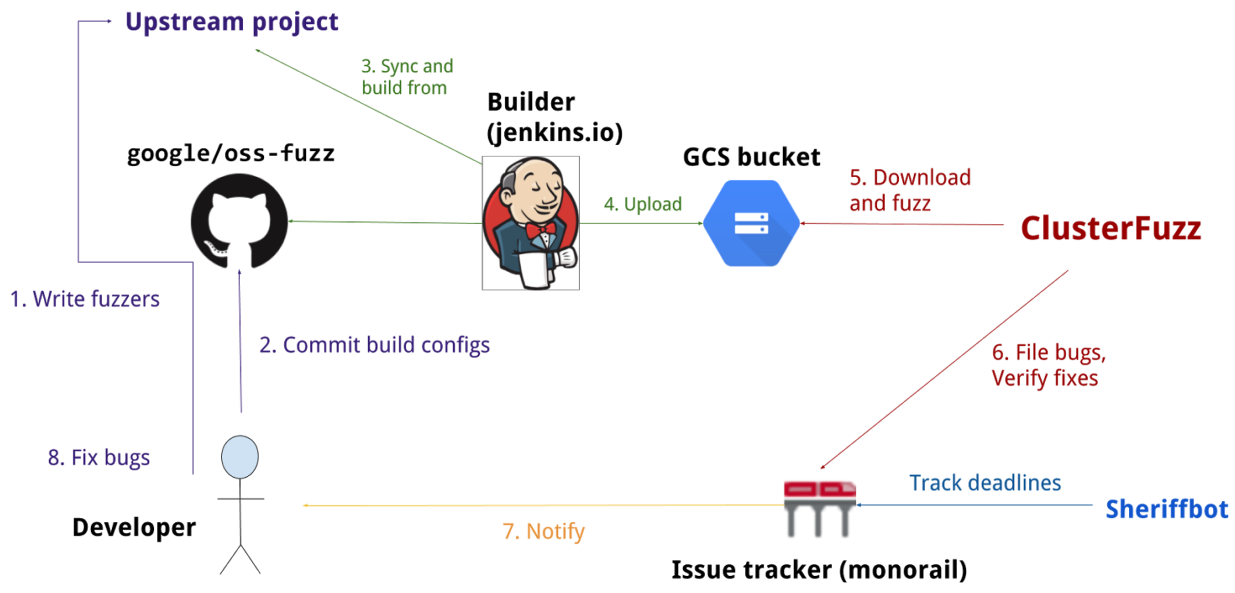
Along with the custom untrusted_* static analysis, using symbolic execution engines to analyze which code paths the untrusted data goes through will add a lot of value to the untrusted data analysis. This would involve initializing the untrusted_* variables with symbolic values, studying the possible paths that they will follow based on the constraints on them and concretizing what needs to be fixed as the analysis proceeds. I plan to implement this module using shellphish's angr. Angr will perform the static and dynamic symbolic analysis for each of the component's binary executable files. While the previous tools mentioned (Frama-C etc.) are program analyzers and will perform the analysis before building (or as a part of the build process), angr will analyze the already built binary executables corresponding to the component.



The symbolic analysis within a component itself will be quite useful, and it (along with the custom static analysis passes) would help us enumerate and prioritize the choice of fuzz targets which will be used during dynamic analysis.

Dynamic analysis

The goal of the dynamic analysis module will be to integrate the Qubes components for continuous fuzzing under Google's oss-fuzz. A major advantage of this would be leveraging the distributed fuzzing execution environment provided through cluster-fuzz.



As a developer, this integration would involve the following steps:

- Creating fuzz-targets: This is an essential step and needs to be done thoroughly for effective fuzzing. The resultant targets can be used with libFuzzer (supported by oss-fuzz) (or afl or radamsa). I'll create fuzz-targets for all the interfaces and components which will be prioritized based on the results of static and symbolic analysis performed on the Qubes components earlier. The actual components which need to be fuzzed are known (like qrexec, qubesdb, gui etc.), but the static and symbolic analysis results will help identify which particular interfaces of these components are security critical and help create better fuzz-targets and harness code. Also, we are not looking for only traditional memory corruption based bugs. For example, we are also looking for filesystem modifications outside of the supposed hierarchy which might happen in some components (like qfilecopy). Details like this will need to be studied and integrated while writing the harness code for the fuzz-targets.

- Integrating the fuzz-target with oss-fuzz: Google recommends some ideal integration steps. I will try to stick to their recommended steps as much as possible for effective integration. The integration would also involve writing build scripts for the fuzzing environment. The caveat here is that the runtime dependencies installed through build scripts are not available where the fuzzers actually run. To be able to use these dependencies, we would need to either install them via Dockerfile and statically link them, or we would have to build the dependencies statically in the build script. Also, I would leverage the sanitizers which google offers while building.

- Applying to google to get accepted: Google does not accept all open source projects to oss-fuzz. However, since Qubes has a significant (and growing) user base, we should hopefully get accepted. And since this is a part of google's own summer of code, they must consider Qubes to be a significant open source project. However, if by some chance, the project does not get accepted, we would still have made significant progress already by the previous two steps, and we could still run these fuzzers on any available machine(s).
- ClusterFuzz: Once accepted, the fuzz tests will run on ClusterFuzz. There would be no intervention from our side here, just regular monitoring as to whether the fuzzers are running properly. This will be done through its web interface. Whenever ClusterFuzz will detect any bug, the issue will be reported to the issue tracker and the Qubes developers will be CC'ed the bug report too. The issue will be made public 30 days after the fix is verified or 90 days after reporting.

So overall, my task would be to write the fuzzers for each Qubes component and corresponding interfaces, write build scripts/Dockerfiles based on oss-fuzz's integration instructions (and examples from other oss projects which have been so integrated), and attempting to fix the issues reported by ClusterFuzz. Of course, the entire Qubes developer community will help fixing the bugs.

Timeline

3rd April - 4th May (Homework period): I will get familiar with the usage of most of the tools that I intend to use for creating the analysis modules. Namely: scan-build, flawfinder, Frama-C, pylint ,bandit, shellcheck, angr, libFuzzer.

5th May - 30th May (Community Bonding period): Based on my experience in the previous month, I will discuss with my mentor what all features of the various tools I could use to perform extensive analysis. I will thoroughly read the code of individual components as this will give me a better understanding of how I should analyze it later. I might also manually start analyzing the qubes' components during this duration itself.

31st May - 14th June (Coding period begins): I will work on the generic static analysis scripts. It would involve writing scripts which will use the tools I've mentioned before in the implementation to perform the analysis.

15th June - 16th June: I will integrate the generic static analysis scripts written for the components into the qubes-builder to allow analysis before building.

17th June - 24th June: I will start the work on custom analysis passes for C based components using Frama-C plugins + the ACSL specific language.

25th June - 26th June: I will document all the code written till now and make it presentable for the first evaluation.

26th June - 30th June (first evaluation): I will work on the feedback from my mentor and attempt to fix any issues and bugs in the code that may have been detected by the analysis modules till now.

1st July - 14th July: Continuing the work on custom analysis passes for C based components, I will finish it, and also attempt to write such custom passes for python and bash based components.

15th July - 23rd July: I will work on the symbolic analysis of the components using angr.

24th July - 28th July (second evaluation): I will work on the feedback from my mentor and attempt to fix any issues and bugs detected by the analysis modules till now.

29th July - 4th August: I will complete the symbolic analysis module and work on allowing collection of the results of both the custom static analysis and the symbolic execution as a part of the build process. Based on these results, I will prioritize the fuzz-targets for the dynamic analysis part.

5th August - 12th August: I will work on writing the harness code for each of those enumerated fuzz-targets.

13th August - 20th August: I will work on writing the build scripts for integration with oss-fuzz. I will also apply to Google's oss-fuzz on behalf of Qubes, to allow the fuzzers to run under it.

21st August - 29th August (Code submission): Hopefully, we would get accepted. If so, then all I'll have to do is to monitor ClusterFuzz's fuzzing results, and fix if something isn't working properly. If by some chance we are not accepted, I would still have completed writing the fuzzers which would run on any machine. I will work on documentation of the code written during the GSoC period and submit it for final evaluation.

30th August - 5th September (final evaluations): I will work on the feedback from my mentor and make improvements wherever I can.

6th September - … (Post GSoC period): I will continue actively contributing to Qubes after the GSoC period ends, fixing the bugs as they show up on oss-fuzz or through other analysis. During the GSoC period I would have laid the groundwork for analysis, now I'll streamline it to find as many bugs as possible and make specific improvements to each module.

About me

Blog: https://paraschetal.in

Github: https://github.com/paraschetal

Linkedin: https://www.linkedin.com/in/paraschetal

University: IIT Roorkee: https://iitr.ac.in/

I am a second year computer science undergraduate studying at IIT Roorkee. I am passionate about information security, developing tools related to security and I regularly participate in security CTFs and practice wargames.

Contact

Email: paras.chetal@gmail.com

IRC nick: feignix

Time Zone: UTC +5:30 (However, I have a messed up circadian rhythm so I'll be available at all times I'm not asleep)

Why me?

I feel I have sufficient motivation to learn and prerequisite knowledge in order to successfully complete this project. I have a some experience in static analysis, dynamic analysis and symbolic execution through my frequent participation in CTFs. Undertaking this project will further increase my familiarity with a variety of analysis tools and techniques. The main reason I took up this project was because it was an opportunity for me to work on analyzing a full-fledged operating system. I have been using the Qubes operating system as my primary OS for over a month now and I would love to improve it and become an active member of the community.

How do I plan to work?

My university's summer vacations are from 5th May - 17th July. I have no other commitments (as of now) during this duration (apart from participating in CTFs occasionally over the weekends) and will be able to devote 40-45 hours per week (approximately 8 hours per day). After 17th July, classes will commence at my university and I will be able to devote approximately 5 hours per day during weekdays. However, I'll make up for it by working extra hard during the weekends.

I will write regular reports on the progress made during the GSoC period on my blog and keep updating my progress on the qubes-devel mailing list as well.

References

- https://groups.google.com/forum/#!topic/qubes-devel/oQO9whS96fg
- https://www.qubes-os.org/doc/
- https://clang-analyzer.llvm.org/scan-build.html
- https://www.dwheeler.com/flawfinder/
- https://frama-c.com/download/frama-c-user-manual.pdf
- https://frama-c.com/download/acsl.pdf
- https://wiki.openstack.org/wiki/Security/Projects/Bandit
- https://pylint.readthedocs.io/en/latest/
- https://github.com/koalaman/shellcheck
- https://github.com/mre/awesome-static-analysis
- https://users.ece.cmu.edu/~aavgerin/papers/Oakland10.pdf
- https://docs.angr.io/
- http://llvm.org/docs/LibFuzzer.html
- https://github.com/google/oss-fuzz/tree/master/docs