Forensic analysis of container checkpoints

Mentors: Radostin Stoyanov, Adrian Reber

Table of Contents

1. Abstract

The crit library in go-criu was created during GSoC 2022 to enable analysis of CRIU images with tools written in Go. It allows container management tools such as checkpointctl and Podman to provide capabilities similar to CRIT. The goal of this project is to extend this library with functionality for forensic analysis of container checkpoints to provide a better user experience. To effectively utilise this new feature, the checkpointctl CLI tool would also be extended to display information about the processes included in a container checkpoint and their runtime state (e.g. memory state, open files, sockets, etc).

2. Technical Details

> **NOTE**: There are two versions of CRIT, one implemented in Python and the other in Go. This proposal refers to the Go implementation of CRIT.

2.1. Forensic analysis with CRIT

In the context of container checkpointing, forensic analysis refers to the process of analysing the runtime state of processes running in a container. This analysis must also be done in a manner that maintains the integrity of the data. The CRIU team added C/R support to Kubernetes in 2022, with the primary goal of enabling forensic container checkpointing. This opens up a host of new possibilities for analysing container vulnerabilities. For example, by integrating the checkpointing functionality with an intrusion detection system, users would be able to automatically create checkpoints of containerised applications when signs of unauthorised access or suspicious behaviour are detected. These checkpoints can then be used to identify and study the actions performed by an attacker.

Currently, CRIT is the only tool available to inspect and analyse a container checkpoint. While CRIT allows decoding of the image files, it requires users to have in-depth understanding of the checkpointing process and image format used by CRIU. The user also must manually sift through each image file and consolidate the data in a format that can be used to meaningfully gather insights. The `crit explore` command provides organised views of some types of data, but it is inadequate for forensic analysis, and ill-suited for users who are not familiar with CRIU. In order to leverage the full power of container checkpointing, we require a powerful forensic analysis tool which works with CRIU image files and provide insights that can be used by security researchers.

2.2. Analysis tool features

In practice, forensic analysis is highly subjective - the exact process of uncovering security vulnerabilities is left to the creativity of the individual. But there are some battle-tested methods that depend on certain types of data to identify potential issues. Some common examples are:

- The process tree and the names of every process present
- The memory pages of a process
- Files opened by a process
- TCP/UDP sockets used by a process
- Changes to the filesystem

The tool that extends CRIT must support these features out of the box. A survey can also be conducted on popular security forums like r/cybersecurity to identify more features and incorporate them accordingly.

2.3. Protobuf issue with CRIT library

Currently, one major blocker is preventing the complete integration of CRIT with multiple other open-source projects - importing any of the packages provided by CRIT causes Go to statically link all 72 protobuf bindings, even if they are not used. This leads to blown-up binary sizes, and also increases bloat in the repository if the project vendors dependencies. The reason for this issue is that the library and CLI implementations of CRIT are tightly coupled. The functions for encoding and decoding use a default handler if the file does not have a special format (e.g. ghost files, pagemap, etc.). Consider the function used by `crit decode`:

```go
func (img *CriuImage) decodeDefault(
    f *os.File,
    decodeExtra func(*os.File, proto.Message, bool) (string, error),
    noPayload bool,
) error {
    sizeBuf := make([]byte, 4)
    for {
        if n, err := f.Read(sizeBuf); err != nil {
            if n == 0 && err == io.EOF {
                break
            }
            return err
        }
        // Create proto struct to hold payload
        payload, err := images.ProtoHandler(img.Magic)
                    //  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        if err != nil {
            return err
        }
        // ...
}
```

The annotated line is the source of all the unnecessary imports. `ProtoHandler()` is a generated function that simply switches the magic, and returns the appropriate protobuf struct. This leads to all the protobuf bindings being imported upon using any file in the library, as the image type must be resolved at runtime. This problem needs to be resolved in order to use the CRIT library as a dependency in any other project.

3. Implementation

3.1. Decoupling CRIT library and CLI

The decoupling process requires two major changes:

- Reworking the generation of protobuf bindings from the `.proto` files
- Refactoring the encode and decode handlers to receive the protobuf struct as a parameter

3.1.1. Generation of protobuf bindings

At the moment, the protobuf bindings for all the `.proto` files are being generated in a single `images` package, which also contains the handler for identifying the protobuf struct from the magic. The generation is being done by a Makefile target that appends the relevant flags to `protoc` and invokes it on every `.proto` file. This target is considerably complex, and would require significant effort to rework. Rather, a Python script to perform the generation offers far more flexibility and has the added benefit of being understandable at a quick glance.

The Python script will perform the following operations:

- Identify every `.proto` file in the `crit/images` directory.
- For each file, create a new directory with the same name.
- Append the `protoc` flag to add the name of the file as the Go package name.
- Generate the `.pb.go` file into the respective directories.

This ensures that the bindings for each type of image is in a separate package, requiring it to be explicitly imported. This eliminates the problem of importing all bindings even when only one type of image is being operated on.

3.1.2. Refactoring encode and decode handlers

The `decodeDefault()` and `encodeDefault()` functions must accept a `proto.Message` object, and use that instead of identifying the struct via `images.ProtoHandler()`. While using the library, it is the responsibility of the user to import the required protobuf bindings from the packages in `crit/images`, and pass the appropriate struct to these functions. The CLI will continue to use the handler to identify the struct required for the image type, ensuring that the CRIT binary is not affected by this refactor.

3.2. Forensic analysis features

As all of the analysis features will require operating on multiple image files, it is ideal to extend `crit/explore.go` with these features. The file already contains the handlers for the existing features provided by `crit x`, and the new ones can be added to the same command and provided on the CLI too. Do note that all the data provided by CRIT comes directly from the checkpoint and is unfiltered. The client using the library (in our case `checkpointctl`) is responsible for implementing features to narrow down the data, like filtering by PID. Providing these features is a non-goal of CRIT.

3.2.1. Existing analysis features

- Viewing the process tree is available through `ExplorePs()`.
- The list of memory pages tagged with the corresponding PIDs is provided by `ExploreMems()`.
- The list of files opened can be fetched via `ExploreFds()`.
- The resident set size (the memory pages held in RAM) are available through `ExploreRss()`.

3.2.2. UNIX and network sockets

The following image files provide details regarding sockets:

- `unixsk.img`: UNIX sockets
- `tcp-stream.img`: TCP connections

These images can be decoded and the data consolidated into a single JSON object.

3.2.3. Filesystem changes

Container checkpoints provide the `rootfs-diff.tar` tarball along with the checkpoint image. This tarball contains all the files on the container's filesystem that were changed with respect to the base image. The list of files along with their MIME types can conveniently be processed into a JSON. Opening or viewing a file is a feature that must be handled by the client. Note that this feature is exclusive to container checkpoints and cannot be added to CRIT, as ordinary checkpoints do not generate the tarball. This feature must directly be added to checkpointctl.

3.3. Integration with checkpointctl

`checkpointctl show` provides the `--print-stats` flag to display the dump statistics of a checkpoint. In a similar fashion, the command can be extended with the following flags:

- `--ptree PID` : Print the process tree starting from the process. If no PID is passed, it prints the entire process tree from the root process that was checkpointed.
- `--mem-map PID` : Print the memory mappings of the process.
- `--files PID` : Print the files opened by the process.
- `--sockets PID` : Print the UNIX and network sockets opened by or bound to the process.
- `--fs-diff` : Print the files changed on the container filesystem.

All flags excluding `--ptree` must throw an error if a PID is not specified. As `checkpointctl show` implies printing, `--print-stats` can be renamed to `--stats` to make it more idiomatic.

4. Timeline

4.1. Before May 4

- Understand what is expected from a forensic analysis tool and plan any potential extra features accordingly.
- Work on a minimal set of test cases to prove the correctness of the features.

4.2. May 4 - May 28 (Community Bonding Period)

- Discuss the finer aspects of the solution and any potential challenges with my mentors.
- Begin writing the Python script to generate the protobuf bindings.

4.3. May 29 - July 14 (Phase I)

- **May 29 - June 4**: Complete the Python script to generate the bindings. Add tests to verify the generated bindings and the directory structure (A bonus would be to also port `magicgen.go` to Python, since Go is not the most ideal candidate for writing code generation scripts).
- **June 5 - June 11**: Begin refactoring the decode and encode handlers.
- **June 12 - June 18**: Complete the refactor and ensure the CLI and library are no longer coupled. Refactor the unit tests accordingly.
- **July 19 - June 25**: Extend `checkpointctl show` with the existing features provided by `crit explore`. Add appropriate tests along with the changes.
- **June 26 - July 2**: Begin implementing `crit show sk` for viewing UNIX and network sockets.
- **July 3 - July 9**: Complete implementation and add unit tests.
- **July 10 - July 14 (Phase I evaluation)**: Discuss progress with my mentors and any potential change of plan going forward.

4.4. July 14 - Aug 28 (Phase II)

- **July 14 - July 23**: Begin implementing `crit show fs-diff` for viewing the list of changed files.
- **July 24 - July 30**: Complete implementation and add unit tests.
- **July 31 - Aug 6**: Begin extending `checkpointctl show` with the newly added CRIT features. Discuss and confirm if opening or viewing individual files should be supported.
- **Aug 7 - Aug 13**: Complete adding the features and related tests. Update the READMEs of both projects.
- **Aug 14 - Aug 20**: Add documentation to both CRIT and checkpointctl for the new features.
- **Aug 21 - Aug 27**: Add the new features to the CRIU website along with examples for each.
- **Aug 28 - Sep 4 (Final evaluation)**: This serves as a buffer to accomodate any unexpected delays or emergencies.

4.5. After Aug 28

- Discuss the outcome of the project with my mentors and chart out a plan of action for future work.
- Engage with community members to get feedback on the project and discuss improvements or enhancements.

5. Personal information

Name: Prajwal S N
Email: prajwalnadig21@gmail.com
Mobile:
GitHub: snprajwal
LinkedIn: snprajwal
Location: Bengaluru, India
Timezone: GMT +0530

5.1. About me

I am a third year student pursuing my B.E. in computer science at Bengaluru, India. I currently work on static analysis for Rust and Go at DeepSource. I was a Google Summer of Code mentee with CRIU in 2022, where I designed and implemented the Go library for CRIT.

5.2. Contribution history

The details of my past work with go-criu along with the commit history can be found in my GSoC 2022 report.

5.3. Commitments during GSoC 2023

I will be dedicating 40 hours a week on average towards this project. My 6th semester classes will be ongoing during the program, but will not affect my work in any manner.