**Rust System Programming Vulnerabilities Detection Using Dynamic Debugging**

**Architecture Overview**

The project is structured as a **Rust-based dynamic analysis tool** with both a command-line interface (CLI) and a graphical user interface (GUI). It is designed to detect common system programming vulnerabilities by running target programs under instrumented environments and analyzing their behavior at runtime. The architecture is composed of several components:

* **Core Analysis Engine (Library Crate):** Implements the logic to launch target binaries with dynamic instrumentation (sanitizers or external tools), monitor their execution, and collect any error reports. This core is used by both the CLI and GUI. It supports detecting memory errors and concurrency issues by leveraging tools like AddressSanitizer and ThreadSanitizer (when available) or equivalent instrumentation.
* **CLI Frontend (Binary Crate):** A console application that parses user inputs (target binary path, options for types of checks to perform, output format, etc.), invokes the core analysis engine, and prints or saves a detailed vulnerability report. This allows automation and scripting usage.
* **GUI Frontend (Binary Crate):** A desktop application for interactive use. It provides a user-friendly interface to configure analysis (select a binary, enable/disable specific checks) and displays the results in an organized format (tables or lists of found issues with details). The GUI uses a Rust UI library that works on Windows (for example, using **egui** for an immediate-mode GUI ([egui: an easy-to-use immediate mode GUI in Rust that runs ... - GitHub](https://github.com/emilk/egui#:~:text=egui%3A%20an%20easy,and%20in%20your%20favorite))) to ensure easy cross-platform compatibility.
* **Reporting Module:** Part of the core engine responsible for formatting the collected vulnerability data into comprehensive reports (e.g., JSON for machine-readable output and HTML for human-readable reports). This module collates information such as the type of vulnerability, memory addresses or variable names involved, stack traces, and source code locations (when available) for each detected issue.
* **Instrumentation Integration:** The engine interacts with dynamic debugging tools. It can either directly leverage compiler-based sanitizers (if the target program is compiled with sanitizer support) or employ binary instrumentation tools for targets without compile-time instrumentation. This dual approach ensures it can handle both **Rust and C/C++ binaries** as required.

Overall, the tool orchestrates an instrumented execution of the target program, intercepts any runtime error reports (such as sanitizer findings or custom trap detections), and then aggregates those findings into a user-friendly report. The design emphasizes modularity: the CLI and GUI share the same backend logic, and the system can be extended to support new types of analyses or additional platforms. The focus in this project is Windows compatibility, so all components are tested on Windows 10/11 and integrated with Visual Studio Code for development convenience.

**Supported Vulnerabilities and Detection Techniques**

This tool targets a range of **common vulnerabilities in low-level programming**. Specifically, it can detect:

* **Buffer Overflows/Out-of-Bounds Accesses:** Errors where a program reads or writes outside the bounds of an allocated buffer (heap, stack, or global memory). These are caught via instrumentation that monitors memory accesses. For example, AddressSanitizer can detect out-of-bounds accesses to heap, stack, and global memory ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=AddressSanitizer%20is%20a%20memory%20error,the%20following%20types%20of%20bugs)). Our tool uses such instrumentation to flag buffer overflows.
* **Use-After-Free (Dangling Pointer Access):** Cases where memory is freed but later accessed. Memory sanitizers (ASan or dynamic tools) mark freed memory and detect any subsequent access as an error ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=,free%2C%20invalid%20free)). The tool will report the location of the use-after-free and relevant stack traces.
* **Double Free/Invalid Free:** Attempting to free the same memory twice or freeing unallocated memory. These are also detected by AddressSanitizer’s runtime or by intercepting heap calls; the tool will categorize and report double-free issues ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=,Memory%20leaks)).
* **Memory Leaks:** Situations where allocated memory is never freed, leading to resource leaks. LeakSanitizer (LSan) is integrated with AddressSanitizer to detect leaks ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=,Memory%20leaks)). On Windows, our tool can invoke leak detection at program end and include any leaks (with allocation stack traces) in the report. If using an external tool, Dr. Memory’s leak checking facility can identify leaks even without compile-time instrumentation ([Home](https://drmemory.org/#:~:text=Dr,reserved%20thread%20local%20storage%20slots)).
* **Data Races (Race Conditions):** Concurrency bugs where two threads access the same memory without proper synchronization, and at least one access is a write. The tool uses ThreadSanitizer when available (requires compilation with thread instrumentation) which is a fast data race detector ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=ThreadSanitizer)). Since TSan is not natively available on Windows MSVC-compiled binaries, the tool can alternatively utilize a dynamic instrumentation approach on Windows. For example, it can integrate **DRace**, a data race detector built on DynamoRIO, to catch races in Windows binaries without special compilation ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=Image%3A%20REUSE%20status%20Image%3A%20CII,Best%20Practices)). This allows detecting race conditions in C/C++ or Rust programs that use threads (so long as they use standard synchronization primitives).
* **Other Memory Errors:** Accesses to uninitialized memory, writing to unaddressable memory regions, or misuse of certain APIs. If using Dr. Memory for analysis, it can catch uses of uninitialized memory and other memory API errors ([Home](https://drmemory.org/#:~:text=Dr,reserved%20thread%20local%20storage%20slots)). UndefinedBehaviorSanitizer (UBSan) is not explicitly mentioned in the requirements, but the design could be extended to include UB checks as well.

**Detection Techniques:** The tool primarily relies on dynamic analysis – running the program with instrumentation:

* For memory errors (buffer overflow, use-after-free, etc.), **AddressSanitizer (ASan)** is used when possible. ASan consists of a compiler instrumentation module and a runtime library ([Performance of Valgrind Memcheck and DBILL Address Sanitizer on ...](https://www.researchgate.net/figure/Performance-of-Valgrind-Memcheck-and-DBILL-Address-Sanitizer-on-SPEC-CINT2006-reference_fig4_262175489#:~:text=,art%20DBI)). The target program needs to be compiled with sanitizer flags (-fsanitize=address for C/C++ or Rust’s -Zsanitizer=address on nightly) so that it includes the necessary checks. AddressSanitizer can detect out-of-bounds, use-after-free, double free, and more with low overhead ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=AddressSanitizer%20is%20a%20memory%20error,the%20following%20types%20of%20bugs)). If the target binary is compiled with ASan, our tool will simply run it with the appropriate environment settings and parse the sanitizer’s output.
* If the target binary was **not compiled with ASan** (especially likely for third-party C/C++ software or Rust stable code), the tool can fall back to **binary instrumentation**. We integrate **Dr. Memory**, which dynamically instruments the binary at runtime (using the DynamoRIO framework) and reports memory errors *without requiring recompilation* ([Home](https://drmemory.org/#:~:text=thread%20local%20storage%20slots)). Dr. Memory will detect invalid memory accesses (out-of-bounds, use-after-free), double frees, uninitialized memory usage, and memory leaks ([Home](https://drmemory.org/#:~:text=Dr,reserved%20thread%20local%20storage%20slots)). This means even a stock Rust or C++ executable (compiled normally) can be checked for memory issues by our tool.
* For detecting **race conditions**, if the user can recompile the program (e.g., a Rust project or C++ project compiled with Clang), ThreadSanitizer can be enabled (-fsanitize=thread or -Zsanitizer=thread). ThreadSanitizer instruments memory accesses and synchronization to flag data races at runtime ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=To%20work%20correctly%20ThreadSanitizer%20needs,lead%20to%20false%20positive%20reports)). However, since not all Windows builds support TSan out-of-the-box, our tool also offers integration with **DRace** on Windows. DRace will attach to the running program and monitor thread interactions to detect data races without needing special compilation ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=Image%3A%20REUSE%20status%20Image%3A%20CII,Best%20Practices)). It uses DynamoRIO to intercept synchronization calls and memory accesses. This dual approach (TSan when possible, DRace otherwise) ensures race detection is covered on Windows.
* **Combination of Tools:** The user can specify which analyses to run. For example, a full run may involve both memory checking and race detection. The tool can sequentially (or concurrently) run the target under memory-error instrumentation and then under race-condition instrumentation, or use a single run if one tool covers both. (ASan/LSan cover memory errors and leaks, and a separate TSan or DRace run covers data races.)

All detected issues are collected in a unified format internally, regardless of which underlying tool found them. Each issue record typically contains: the type of vulnerability, a description or error message, the memory address or variable involved (if applicable), and a stack trace or code location pointing to where the issue occurred. Where possible, the tool will symbolize addresses to source file and line (e.g., using debug symbols and the LLVM symbolizer for ASan output, or PDB symbols on Windows for Dr. Memory/DRace outputs ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=For%20best%20results%2C%20we%20recommend,level%20synchronization%20cannot%20be%20detected))). This makes the reports actionable for developers.

**CLI Component – Command Line Interface**

The CLI is implemented as a Rust binary (vuln\_detector\_cli) using a crate like **Clap** (Command Line Argument Parser) for user input parsing. This allows users to run analyses via terminal commands or integrate the tool into scripts/CI pipelines. Key features of the CLI:

* **Argument Parsing:** The CLI supports flags and options such as:
  + -t, --target <path> – the path to the target binary (Rust or C/C++ executable) to analyze.
  + -c, --checks <types> – which checks to perform (options could be memory, leak, race, or all). By default, all supported checks run.
  + -o, --output <report\_file> – optional path to save the report. If not provided, it may print a summary to stdout and still save a detailed report in a default location (like an outputs directory).
  + --format <json|html|text> – format of the output report. JSON is useful for machines, HTML for humans, text for quick viewing.
  + Other flags like --verbose for more logging, or --timeout <sec> to limit execution time of the target if needed (to handle programs that might hang).
* **Execution Flow:** After parsing arguments, the CLI uses the core analysis library to perform the requested checks. For example, if memory checks are requested, it will set up the instrumentation (possibly by setting environment variables for ASan or launching Dr. Memory). If using ASan, the CLI might set ASAN\_OPTIONS environment variables such as log\_path (to capture logs) and abort\_on\_error=0 (so it continues after finding one error) ([Address Sanitizer - How to set >1 ASAN\_OPTIONS? - Stack Overflow](https://stackoverflow.com/questions/77901419/address-sanitizer-how-to-set-1-asan-options#:~:text=Overflow%20stackoverflow,log_file%27%3Acontinue_on_error%3D1%20%C2%B7%20exchanged%20option)). If using Dr. Memory, the CLI constructs the command to run Dr. Memory’s drrun launcher with appropriate flags (e.g., -batch for non-interactive, -leaks to enable leak checking, etc., and the target command). Similarly for DRace, constructing the drrun -c drace-client.dll ... -- <exe> invocation per DRace’s documentation ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=Run%20the%20detector%20as%20follows)). All this is handled via Rust’s std::process::Command.
* **Capturing Output:** The CLI captures the stdout/stderr of the instrumented run. This can be done by piping the output of the Command. For example:
* use std::process::Command;
* let output = Command::new(drmemory\_path)
* .args(&["-batch", "--", target\_exe])
* .output()
* .expect("Failed to execute Dr. Memory");
* let stdout = String::from\_utf8\_lossy(&output.stdout);
* let stderr = String::from\_utf8\_lossy(&output.stderr);

The tool then passes these outputs to parser functions that look for known patterns indicating errors. For instance, AddressSanitizer errors have signatures like ERROR: AddressSanitizer: or WARNING: ThreadSanitizer: in their output. Dr. Memory outputs its findings at program exit with a summary of errors and details (it can produce reports of each error, often prefixed by Error #n: etc.).

* **Parsing and Aggregation:** The CLI (via the core library) parses the raw output to extract structured information. For ASan/TSan, we use the fact that their reports follow a standard format, including the type of error (e.g., "stack-buffer-overflow", "data race"), the memory address, and a stack trace ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20cargo%20run%20,rs%3A3%3A23)) ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20export%20RUSTFLAGS%3D,rs%3A5%3A18%20%28example%2B0x86cec%29)). We use regex or string search to identify each error block. Similarly, Dr. Memory’s output can be parsed by looking for keywords like "UNADDRESSABLE ACCESS" or "Invalid heap argument". Each found issue is converted into our internal VulnerabilityReport struct (with fields like issue\_type, description, backtrace, etc.).
* **Reporting:** After analysis, the CLI either prints a concise summary or indicates how many issues of each type were found. The full details are then written to the specified output format:
  + **JSON Report:** Using **Serde** to serialize the list of issue structs into JSON. The JSON might look like an array of objects, e.g.:
  + [
  + {
  + "type": "use-after-free",
  + "description": "Heap-use-after-free on address 0x60400000eff0",
  + "location": "src/main.rs:42:5",
  + "stack\_trace": [
  + "main::do\_something() at src/main.rs:42",
  + "main::inner() at src/main.rs:30",
  + "std::thread::spawn::{{closure}}() at ...",
  + "start\_thread in ntdll.dll"
  + ]
  + },
  + {
  + "type": "data-race",
  + "description": "Race condition on variable x",
  + "location": "example.cpp:88:7",
  + "stack\_trace": [ "...threads stacktrace..." ]
  + }
  + ]

Each entry provides the key info needed to understand and locate the bug.

* + **HTML Report:** The tool can use a templating crate (like **Tera** or **Askama**) to generate an HTML file embedding the results. For example, an HTML report may contain a table where each row is an issue, with columns for "Type", "Location", "Description". The stack trace can be included in a collapsible section. The HTML is styled for readability (e.g., highlighting error types in red). This report can be opened in a browser for inspection.
  + **Text Report:** Alternatively, a plain text or Markdown summary can be generated. This might list each issue in order with its details, which is useful for quick reading or including in documentation.
* **Exit Codes:** The CLI will return a non-zero exit code if any vulnerabilities were found (so it can fail a CI job, for example). Different categories might have different codes or it may simply be 1 if any issues, 0 if none.

**CLI Example Usage:** Suppose we have a sample vulnerable program vuln\_example.exe (compiled from C++ or Rust). Running the CLI might look like:

vuln\_detector\_cli.exe -t C:\path\to\vuln\_example.exe -c all -o report.json --format json

This would run all checks on the binary. During execution, the CLI might print:

[+] Running memory error analysis on vuln\_example.exe...

[!] Error detected: heap-buffer-overflow at 0x000001234560 (write of size 4)

[!] Error detected: memory leak (256 bytes lost)

[+] Running data race analysis...

[!] Warning: data race condition detected between threads T1 and T2 on address 0x0000012345A0

[+] Analysis complete. 3 issues found (1 overflow, 1 leak, 1 data race).

[+] Detailed report saved to report.json

The saved JSON report would contain the details of those 3 issues. In this example, the heap-buffer-overflow could correspond to a buffer overflow error that AddressSanitizer detected (with its description and stack trace) ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20cargo%20run%20,rs%3A3%3A23)), the leak might come from LeakSanitizer or Dr. Memory, and the data race from ThreadSanitizer or DRace.

This CLI component makes it straightforward to integrate the tool into development workflows, allowing quick checks of binaries for critical vulnerabilities.

**GUI Component – Graphical User Interface**

In addition to the CLI, the project includes a GUI application (vuln\_detector\_gui) for users who prefer a graphical interface or who want to continuously monitor a program in a user-friendly way. The GUI is built using a Rust GUI library that is cross-platform and works well on Windows. For example, we might use **egui** (via the eframe framework) which allows creating native Windows GUI apps in pure Rust ([egui: an easy-to-use immediate mode GUI in Rust that runs ... - GitHub](https://github.com/emilk/egui#:~:text=egui%3A%20an%20easy,and%20in%20your%20favorite)). Key aspects of the GUI:

* **Layout and Controls:** The GUI provides a window with sections to:
  + Select the target binary: a file picker or a text field to input the path of the program to analyze.
  + Choose analysis options: a set of checkboxes or toggles for "Memory Errors", "Memory Leaks", "Race Conditions". The user can enable or disable each category of bug to check for.
  + Start/Stop buttons: A "Run Analysis" button begins the analysis by calling into the same core engine the CLI uses. If the program execution is lengthy or if we allow running the target continuously, a "Stop" button could terminate the run.
  + Configuration options: Possibly allow setting timeouts, or the path to external tools (like if Dr. Memory is not bundled, let the user specify its path).
* **Executing Analysis:** When the user clicks "Run", the GUI spawns a background task (using Rust threads or async tasks) that performs the analysis. This prevents the UI from freezing. The core analysis library is invoked similarly to the CLI path. As the analysis progresses, the GUI can live-update with findings. For example, if an error is found, it can immediately show up in the GUI list with a label like "Use-After-Free detected in module XYZ".
* **Display of Results:** Results are presented in a structured form:
  + A table or list of issues found. Each entry shows at least the type (with an icon or color coding for severity), a brief description, and maybe the location (if available, file/line or module+address).
  + Selecting an issue (e.g., clicking on it) could open a detailed view that shows the full stack trace and memory address information. This detailed view can also provide suggestions or references (for example, if it's a use-after-free, explain that the memory was freed at a certain earlier point if known).
  + If no issues are found, the GUI can display a "No vulnerabilities detected in this run." message, perhaps in green.
* **Report Management:** The GUI can offer the ability to save the results to a file (JSON or HTML as in CLI) via a "Save Report" button or menu option. It might also show the report in-app if HTML (like render the HTML in a webview component) or allow copying the text.
* **Example GUI Workflow:** A user opens the GUI application. They select vuln\_example.exe and leave all checkboxes (Memory, Leaks, Races) checked. They click "Run Analysis". The GUI status bar might show "Running analysis, please wait..." During execution, if an issue is found, the GUI could list it immediately:

| **Type** | **Description** | **Location** |
| --- | --- | --- |
| Buffer Overflow | Heap buffer overflow (write of size 4) | vuln.cpp:120 |
| Memory Leak | 1 leak (256 bytes) not freed | N/A (see details) |
| Data Race | Data race on variable counter | threads.cpp:45 |

* After completion, the status bar updates to "Analysis complete. 3 issues found." The user can click on each issue to see more info. For example, clicking "Buffer Overflow" might expand to show the stack trace where the overflow occurred, similar to the ASan output (with function names and line numbers) ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20cargo%20run%20,rs%3A3%3A23)). The user can then choose "Save Report" and export an HTML report of these results.
* **GUI Implementation Details:** Using **egui/eframe**, the GUI code might look like:
* // Pseudocode sketch
* struct AppState {
* target\_path: String,
* check\_memory: bool,
* check\_race: bool,
* results: Vec<VulnReport>
* }
* impl eframe::App for AppState {
* fn update(&mut self, ctx: &egui::Context, frame: &mut eframe::Frame) {
* egui::CentralPanel::default().show(ctx, |ui| {
* ui.heading("Vulnerability Detector");
* ui.horizontal(|ui| {
* ui.label("Target:");
* if ui.text\_edit\_singleline(&mut self.target\_path).lost\_focus() {
* // handle file path input
* }
* if ui.button("Browse...").clicked() {
* // open file picker dialog (this might require native dialog integration)
* }
* });
* ui.checkbox(&mut self.check\_memory, "Memory Errors & Leaks");
* ui.checkbox(&mut self.check\_race, "Race Conditions");
* if ui.button("Run Analysis").clicked() {
* let target = self.target\_path.clone();
* let do\_mem = self.check\_memory;
* let do\_race = self.check\_race;
* // spawn a thread to run analysis
* std::thread::spawn(move || {
* let findings = core::analyze\_target(&target, do\_mem, do\_race);
* // After getting results, update UI state (using egui Context)
* // (In practice, we might use an async channel to send results back to the UI thread)
* });
* }
* ui.separator();
* // Display results if any
* for issue in &self.results {
* ui.collapsing(format!("{}: {}", issue.ty, issue.short\_desc()), |ui| {
* ui.label(issue.detail\_report());
* });
* }
* });
* }
* }

This simplified pseudo-code illustrates the structure: UI elements for input, a button to run, and a list of results. In a real implementation, we would handle threading carefully so that once the background analysis completes, we signal the GUI to refresh the results list (perhaps via a channel or by using egui's support for async). We might also disable the "Run" button while analysis is ongoing, and show a spinner or progress bar.

* **Integration with VS Code:** While the GUI is a standalone app, during development one can launch it from Visual Studio Code (e.g., using a launch configuration to run cargo run --bin vuln\_detector\_gui). The GUI crate will be part of the Cargo workspace, so building it in VS Code is as straightforward as building any Rust binary.

The GUI component makes the tool accessible to users who are less comfortable with command-line tools, and it provides a nicer way to visualize and browse through potentially long lists of vulnerabilities.

**Integration of Debugging/Instrumentation Tools**

A core aspect of this project is combining Rust code with established debugging/instrumentation tools:

* **AddressSanitizer Integration:** AddressSanitizer (ASan) is integrated by requiring the target program to be compiled with ASan support. For C/C++ binaries, this means using a compiler like clang or GCC with the -fsanitize=address flag and linking the ASan runtime. Notably, on Windows, using clang-cl under the MSVC environment is one way to get ASan support ([Google sanitizers | CLion Documentation](https://www.jetbrains.com/help/clion/google-sanitizers.html#:~:text=Sanitizers%20are%20implemented%20in%20Clang,are%20AddressSanitizer%2C%20ThreadSanitizer%2C%20and%20UndefinedBehaviorSanitizer)) (Microsoft’s Visual C++ compiler also added ASan support in recent versions, making ASan available in Visual Studio 2019+ ([AddressSanitizer on Windows - Victor Ciura [ ACCU 2021 ] - YouTube](https://www.youtube.com/watch?v=yJLyANPHNaA#:~:text=AddressSanitizer%20on%20Windows%20,is%20for%20MSVC%20projects))). For Rust binaries, ASan can be enabled on nightly Rust via RUSTFLAGS="-Z sanitizer=address" along with building the standard library with sanitizers ([GitHub - japaric/rust-san: How-to: Sanitize your Rust code!](https://github.com/japaric/rust-san#:~:text=You%20have%20to%20compile%20your,does%20the%20trick)). However, since binaries may not always be compiled with ASan, our tool checks for the presence of the ASan runtime in the target. If the target is ASan-instrumented (we might detect this by an indicator in the binary or simply by attempting to run and see if it reports sanitizer initialization), we then execute it normally and let it print errors. The tool sets environment variables such as ASAN\_OPTIONS to control the output:
  + We often use log\_path=... to have ASan write its reports to a file (especially if the output is very verbose) ([Address Sanitizer - How to set >1 ASAN\_OPTIONS? - Stack Overflow](https://stackoverflow.com/questions/77901419/address-sanitizer-how-to-set-1-asan-options#:~:text=Overflow%20stackoverflow,log_file%27%3Acontinue_on_error%3D1%20%C2%B7%20exchanged%20option)).
  + We set detect\_leaks=1 on Windows if leak detection is desired (since LeakSanitizer on Windows might be off by default depending on runtime support).
  + We might disable the default behavior of halting on the first error by setting abort\_on\_error=0 or halt\_on\_error=false, so that the program continues after detecting one issue, allowing the tool to capture multiple issues in one run ([[GIT pull] perf updates for 5.1 - Linux-Kernel](https://lkml.indiana.edu/hypermail/linux/kernel/1903.3/00330.html#:~:text=,pid%3E%27.%20%2B%20%2B)). (There is also ASAN\_OPTIONS=continue\_after\_error=1 in newer runtimes to gather more errors.)
  + The tool also ensures that the **LLVM symbolizer** is available in PATH so that ASan outputs are symbolized (file/line info). In a Windows environment, this might require having llvm-symbolizer.exe available (often bundled with LLVM or in Rust toolchain for MSVC).
* **ThreadSanitizer Integration:** If the target is compiled with -fsanitize=thread (or Rust’s -Z sanitizer=thread), we run it similarly. TSan will report data races to stderr. We may need to set TSAN\_OPTIONS if any custom behavior is desired (for example, controlling whether it reports the first race or all races, or suppressing known benign races via a suppression file). As noted, TSan requires that all code be instrumented to avoid false positives ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=To%20work%20correctly%20ThreadSanitizer%20needs,lead%20to%20false%20positive%20reports)), so it's primarily useful when we have control to recompile the target and its libraries. TSan is not supported on MSVC toolchain as of now, so this is more applicable if using clang on Windows or analyzing on Linux/macOS. For completeness, our project documentation would mention that data race detection on Windows without recompilation is handled by DRace instead of TSan.
* **Dr. Memory Integration:** Dr. Memory is used when the target cannot be recompiled with ASan/TSan. We include instructions or scripts to download Dr. Memory (which is LGPL licensed and can be freely used ([Home](https://drmemory.org/#:~:text=Downloading%20Dr))). The tool can either invoke Dr. Memory if it’s installed on the system, or we might bundle the Dr. Memory binaries in our project (perhaps under a tools/ directory). The core engine will construct a command like:
* drmemory.exe -batch -dr  # (drmemory flags)
* -- C:\path\to\target.exe [target arguments]

We use -batch to ensure it doesn’t open a GUI itself and returns after done, and possibly -disable\_async to make Dr. Memory run in a deterministic way. We definitely enable leak checking (which is on by default ([Dr. Memory Runtime Option Reference](https://drmemory.org/page_options.html#:~:text=%2A%20,be%20kept%20internally%2C%20while%20disabling))) unless the user turned it off. Dr. Memory will then run the program and upon completion output a summary of any errors found:

* + Errors such as invalid memory accesses (out-of-bounds or UAF) are reported with stack traces.
  + Leaks are reported with how many bytes were lost and at what allocation point (if it can determine).
  + The tool captures this output and parses it for inclusion in our report. For instance, Dr. Memory might output "Error #1: UNADDRESSABLE ACCESS: reading 0x00000010 at 0x00401234" followed by a call stack. We map that to a "buffer overflow" or "wild pointer dereference" in our terminology and include the call stack. Because Dr. Memory can instrument any **unmodified binary** on Windows, it greatly enhances our tool’s capability to work with arbitrary executables ([Home](https://drmemory.org/#:~:text=thread%20local%20storage%20slots)). It covers the same categories as ASan (and even some not covered by ASan, like uninitialized reads) with the trade-off of being slower.
* **DRace Integration:** For data races on arbitrary binaries, we integrate DRace (an open-source tool by Siemens). DRace uses the DynamoRIO dynamic instrumentation platform just like Dr. Memory, but specifically focuses on thread race detection ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=Image%3A%20REUSE%20status%20Image%3A%20CII,Best%20Practices)). We will include DRace’s drace-client.dll and configuration files, or guide users to download them. Running DRace is a bit involved; essentially one uses the DynamoRIO runner:
* drrun.exe -c drace-client.dll -- <target.exe> [args]

plus some DRace-specific options if needed (like suppression files for known false positives). Our tool will handle this internally. DRace will log any data races detected along with stack traces of the racing accesses. The output is captured and parsed similarly. DRace benefits from having debug symbols for better traces ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=For%20best%20results%2C%20we%20recommend,level%20synchronization%20cannot%20be%20detected)), so we document that providing PDB files for the target can improve race report quality. Because DRace does not require the program to be compiled with special flags, it nicely complements our approach to catch races even in third-party binaries.

* **Unified Handling:** Regardless of which tool is used under the hood, the core engine normalizes the outputs. For example, an AddressSanitizer report and a Dr. Memory report for a use-after-free might have slightly different wording, but the tool will interpret both and classify them as the same type of issue. This way, the reports to the user are consistent. We also keep references to the tool that detected it (perhaps in the detailed report, "Detected by ASan" or "Detected by Dr. Memory") for transparency.
* **Testing the Integration:** We include in the project some sample programs (source code in a examples/ directory):
  + A small Rust program with an intentional buffer overflow (using an unsafe block) to test ASan.
  + A C program with a memory leak and a use-after-free.
  + A multi-threaded C++ example with a data race. We provide precompiled binaries of these with and without sanitizers. This allows testing that our tool can detect the issues via different paths (with direct ASan vs with Dr. Memory/DRace). For instance, the Rust buffer overflow example, when compiled with ASan, should be caught by simply running it (ASan would output an error which we parse). If compiled normally, running under Dr. Memory should catch it as well. These examples serve as verification of each detection method.

In summary, the tool integrates industry-standard dynamic analysis engines (Google Sanitizers, DynamoRIO-based tools) into a coherent Rust application. This gives the benefit of powerful detection (memory errors with zero false positives ([AddressSanitizer | Microsoft Learn](https://learn.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170#:~:text=AddressSanitizer%20,bugs%20with%20zero%20false%20positives)), as AddressSanitizer is known to have no false positives and precise error reports) while providing a user-friendly interface and additional report generation on Windows, which is traditionally a challenging platform for such tools.

**Report Generation System**

The reporting subsystem takes the raw data of detected vulnerabilities and produces clear, detailed reports for developers. The system supports **JSON and HTML** output, as well as the interactive view in the GUI.

* **JSON Reports:** We use the serde and serde\_json crates to serialize the structured findings into JSON. This is straightforward since our internal VulnerabilityReport struct can derive Serialize. The JSON format was illustrated earlier – it's basically a list of issues. We ensure that even complex data like stack traces are included (likely as an array of strings, each string being one frame of the trace). JSON reports can be used by other tools, or even fed into IDEs or CI systems. For example, one could write a script to convert our JSON into GitHub annotations or similar for CI.
* **HTML Reports:** For HTML, we want a human-readable, possibly more nicely formatted output. We can use a template engine:
  + In our project, we include a template (e.g., report\_template.html) using **Tera**. This template might define an HTML file structure with placeholders for the title, summary, and list of issues. The Rust code loads the template and provides the data (perhaps converting the VulnerabilityReport list into a Tera context).
  + Alternatively, we could generate HTML manually using string concatenation, but using a template is cleaner and more maintainable.
  + The HTML report includes a summary at the top (e.g., "5 issues found: 2 buffer overflows, 1 use-after-free, 2 data races.") and then a section for each issue. Each issue could be in a <div> with a class indicating severity (so we can style, say, memory errors with one color and race conditions with another). We include the same details: type, description, location, and stack trace. The stack trace might be in a <pre> block to preserve formatting. If the issue involves a specific memory address, we include it; if it involves a variable or global, we include that name if available.
  + We also add links or references if possible – for example, if file paths are present and we know the source directory, the report could hyperlink file names to local file paths (this would rely on the user's environment, so this might be something we note but not implement by default).
  + An example snippet in the HTML report for a buffer overflow might look like:
  + <h3>Issue 1: Heap Buffer Overflow</h3>
  + <p>Description: Write of size 4 at 0x60400000fed0 overflows target buffer</p>
  + <p>Location: example.c:42 (in function <code>process\_data</code>)</p>
  + <details>
  + <summary>Stack Trace</summary>
  + <pre>
  + #0 process\_data (example.c:42)
  + #1 main (example.c:75)
  + #2 \_\_libc\_start\_main (libc.so.6+0x...)</pre>
  + </details>

And similarly for other issues. The use of <details> and <summary> tags (as shown) can make the stack trace collapsible for neatness.

* + We embed some basic CSS in the report to make it readable (e.g., .overflow { color: red; } .race { color: orange; } etc., and ensure the font is monospace for code).
* **Saving Reports:** The CLI directly writes the report files (using Rust’s file I/O). The GUI, if the user chooses to save, will trigger the same code and then possibly show a file save dialog.
* **Ensuring Detail:** The reports aim to be *detailed*. That means if an address is involved, we include it; if a variable name is known (like a global or static that ASan might name in the report), we include that; the exact error message from the sanitizer or tool is included as part of the description (this helps developers search the exact sanitizer output if needed). We also clearly indicate how many total issues and their classification in the report introduction.
* **Example Report Output:** Consider the sample run earlier that found a buffer overflow, a leak, and a race. In HTML, the report might start with:
* <h1>Vulnerability Report for vuln\_example.exe</h1>
* <p><strong>Summary:</strong> 3 issues found in vuln\_example.exe</p>
* <ul>
* <li>1 Buffer Overflow</li>
* <li>1 Memory Leak</li>
* <li>1 Data Race</li>
* </ul>
* <hr>
* <!-- then details for each issue... -->

In JSON, the top-level array would have three objects, each with "type": "buffer-overflow", "type": "memory-leak", etc., and details accordingly.

* **Extensibility:** The report system is designed such that adding a new kind of analyzer (say, an integration with an undefined-behavior sanitizer or a static analysis result) would just extend the VulnerabilityReport structure and the template accordingly. We isolate the formatting logic from the analysis logic – for instance, by collecting all results in memory first, then formatting to the desired output in one place.

By providing both JSON and HTML, we cover use cases where developers want to programmatically consume the results and cases where they want to read a nicely formatted report. The clarity of these reports is crucial: since this tool is meant to aid developers in finding and fixing bugs, the report should make it as easy as possible to pinpoint the error and understand its nature.

**Dependencies and Setup Instructions**

To implement the above functionalities, we utilize various dependencies (Rust crates and external tools). Below is a list of key dependencies and how to set up the project environment:

* **Rust Toolchain:** Ensure Rust is installed (we recommend the latest stable Rust via rustup). The project may use a Rust 2021 edition and requires nightly only if you plan to compile code with sanitizers; the tool itself is built with stable Rust.
* **Cargo Workspace:** The project is organized as a Cargo workspace with multiple crates:
  + core (library crate): in core/Cargo.toml
  + cli (binary crate): in cli/Cargo.toml
  + gui (binary crate): in gui/Cargo.toml
  + A top-level Cargo.toml that defines the workspace members.
* **Rust Crates (Dependencies):**
  + clap (or the newer clap v4 with derive macros) for parsing CLI arguments in the CLI crate.
  + serde and serde\_json for serializing reports to JSON in the core crate.
  + tera or askama for HTML templating in the core crate (for report generation).
  + log and env\_logger (optional) for logging internal info (we might log steps like "Running Dr. Memory..." or debug info).
  + egui and eframe for the GUI crate (this provides the immediate mode GUI framework that works on Windows, using winit under the hood for windowing).
  + Optionally rfd (Rust file dialogs) crate if we want a cross-platform file picker for the GUI "Browse" button (rfd can open the standard Windows file dialog).
  + anyhow or thiserror for error handling to manage results from running external processes.
  + regex for parsing output texts from tools like Dr. Memory/ASan if needed (though simple str::contains and splitting might suffice).
* **External Tools Dependencies:**
  + **LLVM/Clang**: If analyzing C/C++ code with sanitizers, the user should have clang available to compile with -fsanitize=address or thread. For Rust code with sanitizers, they need the nightly compiler and to enable the feature. (This is for instrumenting the target, not the tool itself.)
  + **AddressSanitizer Runtime**: On Windows, if using clang-cl or GCC, ensure the ASan runtime library is found. For clang, this is usually automatic. For GCC MinGW, one might need to install the libasan library. (If the target isn't compiled with ASan, this is not needed.)
  + **Dr. Memory**: Download the Dr. Memory package for Windows from the official site ([Home](https://drmemory.org/#:~:text=Downloading%20Dr)). We suggest using Dr. Memory version 2.3 or later (as of April 2025, the latest version is 2.3.20167 ([Home](https://drmemory.org/#:~:text=workflow%20as%20for%20DynamoRIO%20dynamorio,))). After downloading, extract it and note the path to drmemory.exe. Our tool can be configured to look for Dr. Memory in the %PATH% or a known installation directory. We might include a configuration in our Settings (for GUI) or a CLI flag like --drmemory-path if not in PATH.
  + **DynamoRIO/DRace**: DRace requires the DynamoRIO framework. DRace’s releases often bundle the necessary components. Download DRace from its GitHub releases and DynamoRIO 8.0 (weekly build as recommended) ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=When%20using%20the%20pre,only%20DynamoRIO%20is%20required)). Set an environment variable or configuration for the path to drrun.exe (DynamoRIO’s runner) and the drace-client.dll. Again, our tool can be configured to use these paths. *Alternatively*, to simplify user setup, we could package a minimal DynamoRIO and DRace along with our installer (if we make one), but in the initial project setup documentation, we instruct how to get them.
* **Building the Project:**
  + After cloning the repository, run cargo build --release to build all components. This will fetch all Rust crate dependencies automatically.
  + The GUI crate might require some extra linking on Windows (for example, egui/eframe will use Direct2D by default via wgpu or glow, but this is typically handled by the crate; just ensure you have a graphics card driver that supports at least OpenGL 3.3 or DirectX 11 for egui).
  + No special build script is required beyond Cargo’s usual process. We include a build.rs only if needed to maybe check platform or embed the HTML template into the binary for easier distribution.
* **Visual Studio Code Integration:** The project is fully compatible with VS Code:
  + We include a .vscode directory with recommended settings. For example, settings.json might enable features like "rust-analyzer cargo allFeatures": true if needed.
  + A tasks.json is provided to build the project (though cargo build is straightforward, this allows building via Ctrl+Shift+B). For example, a task to build the CLI and GUI:
  + {
  + "label": "Build All",
  + "type": "shell",
  + "command": "cargo build",
  + "problemMatcher": "$rustc"
  + }
  + We also provide a launch.json with configurations to run and debug:
    - One config for the CLI, e.g.:
    - {
    - "name": "Debug Vulnerability CLI",
    - "type": "cppvsdbg",
    - "request": "launch",
    - "program": "${workspaceFolder}/target/debug/vuln\_detector\_cli.exe",
    - "args": ["-t", "examples/overflow.exe", "-c", "all"],
    - "cwd": "${workspaceFolder}",
    - "stopAtEntry": false
    - }

(We use cppvsdbg here to leverage the C++ debugger which works for Rust in VSCode on Windows, or we could use the CodeLLDB extension which would be lldb type.)

* + - Another config for the GUI:
    - {
    - "name": "Run Vulnerability GUI",
    - "type": "cppvsdbg",
    - "request": "launch",
    - "program": "${workspaceFolder}/target/debug/vuln\_detector\_gui.exe",
    - "args": [],
    - "cwd": "${workspaceFolder}"
    - }

These allow pressing F5 in VSCode to launch the tool easily. (We document that the user should install the **Microsoft C++ VSCode debugger** or the **CodeLLDB** extension to debug Rust in VSCode.)

* + With these configurations, development and testing on Windows via VSCode are convenient. The developer can put breakpoints even in our Rust code (since the VSCode debugger or CodeLLDB can handle Rust debug symbols), which is helpful for extending the tool.
* **Running on Windows:** We clearly state that the tool should be run in a **Developer Command Prompt** or a VSCode terminal that has the necessary PATH for tools. For example, if using MSVC’s ASan, the environment needs to find asan\_runtime.dll; if using Dr. Memory, the PATH should include the Dr. Memory folder or we specify full paths.
* **Installation/Usage for End Users:** For someone who just wants to use the tool (not develop it), we would provide a pre-built binary release. That release would include:
  + vuln\_detector\_cli.exe
  + vuln\_detector\_gui.exe
  + Perhaps a subfolder tools/ with Dr. Memory and DRace binaries if redistribution is allowed (Dr. Memory is LGPL, which is okay if we dynamically call it; DRace is under an open-source license too). If not bundling, our release notes would point to where to get them.
  + A sample config.toml where the user can set paths to Dr. Memory/DRace if they installed them separately.
  + Documentation (this could be a README or a help flag in CLI) explaining usage.
* **Dependencies Summary:**
  + Rust crates: clap, serde, serde\_json, tera/askama, egui/eframe, possibly others (regex, log).
  + External: LLVM (optional, for compiling targets), Dr. Memory, DRace, Visual Studio Build Tools (if compiling C/C++ with MSVC ASan), or MinGW (for GCC with ASan).
  + These dependencies are chosen to ensure that the tool covers the full spectrum of required functionality (from parsing input, providing UI, to performing heavy-duty analysis via external libraries).

By carefully documenting these dependencies and providing easy setup scripts (like a scripts/setup\_env.ps1 that could check/install Dr. Memory), we ensure that developers can get the project up and running on Windows with minimal friction. Compatibility with Visual Studio Code means they can use a familiar environment to modify or extend the project.

**Example Usage and Demonstration**

To illustrate the capabilities of the tool, the project includes **sample vulnerable programs** and corresponding usage scenarios. Here we walk through an example to demonstrate how the tool detects issues in both Rust and C/C++ binaries.

**Example 1: Rust Program with Memory Bug**

Consider a Rust program (examples/buffer\_overflow.rs) that has an intentional buffer overflow using an unsafe block:

// buffer\_overflow.rs

fn main() {

let arr = [1, 2, 3, 4];

unsafe {

// Deliberately read out of bounds (index 4 is out of range 0-3)

let val = \*arr.as\_ptr().add(4);

println!("Read value: {}", val);

}

}

If we compile this without any special flags (just cargo build --release for example), the program will likely run and possibly print some arbitrary value or crash. Now, using our tool:

* **Using CLI:** We run vuln\_detector\_cli -t examples\buffer\_overflow.exe -c memory. The tool launches the binary under Dr. Memory (since the binary isn’t ASan-instrumented). Dr. Memory will catch the out-of-bounds read. The CLI output might show:
* [+] Running memory error analysis on buffer\_overflow.exe...
* [!] Error: Invalid memory access (heap out-of-bounds) detected at 0x0040A3B4
* -> Reading 4 bytes past a 16-byte array
* -> Location: buffer\_overflow.rs:5:18 (in main)
* [+] Analysis complete. 1 issue found.

And a JSON or text report is produced indicating a buffer overflow. If we had compiled the Rust program with ASan (using nightly Rust with -Zsanitizer=address ([GitHub - japaric/rust-san: How-to: Sanitize your Rust code!](https://github.com/japaric/rust-san#:~:text=,gnu))), then running it under the tool would leverage ASan. AddressSanitizer would output an error like “ERROR: AddressSanitizer: stack-buffer-overflow on address 0x... at pc 0x... bp 0x... sp 0x...” along with the code location ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20cargo%20run%20,rs%3A3%3A23)). Our tool would parse that and report similarly that a stack buffer overflow was found at main in buffer\_overflow.rs. The stack trace from ASan (which would show the exact line of the overflow) is captured and included in the detailed report.

* **Using GUI:** If we open the GUI, select buffer\_overflow.exe, and run analysis (Memory Errors checkbox on, Race Conditions off), the GUI would show a table with one entry:
  + Type: **Buffer Overflow** (possibly marked with a red icon)
  + Description: "Read 4 bytes out of bounds from array arr"
  + Location: buffer\_overflow.rs:5:18 Clicking this entry would expand to show the stack trace (which in this case is trivial: just main). This confirms the tool caught the vulnerability. The GUI user can then save an HTML report. That HTML report might show the code snippet or mention the array name if our parser is smart enough (ASan sometimes indicates the variable name involved in an overflow ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=This%20frame%20has%201%20object,bytes%20around%20the%20buggy%20address)), e.g., it might say which variable’s memory was overflown).

This example shows that even Rust programs, which are normally memory-safe, can have vulnerabilities in unsafe code, and our tool is effective at catching them.

**Example 2: C++ Program with Use-After-Free and Leak**

Consider a C++ program (examples/use\_after\_free.cpp):

#include <iostream>

#include <cstring>

#include <thread>

int main() {

char\* data = new char[10];

std::strcpy(data, "hello");

delete[] data;

// Use-after-free

std::cout << data[0] << std::endl;

// Memory leak

char\* leak = new char[100];

(void)leak;

return 0;

}

This program frees data and then uses it, and also allocates leak without freeing it.

* If we compile this program normally (no sanitizers), running under our tool:
  + Dr. Memory will detect the use-after-free (as an "INVALID HEAP ARGUMENT" or "READ of freed memory" error) and the memory leak at program exit ([Home](https://drmemory.org/#:~:text=Dr,reserved%20thread%20local%20storage%20slots)).
  + CLI output might be:
  + [!] Error: Use-after-free detected at 0x00ABCDEF (read of freed memory)
  + -> Freed at main.cpp:8, used at main.cpp:9
  + [!] Warning: Memory leak detected – 100 bytes from allocation at main.cpp:11 were not freed

and these would be in the report as two issues. Dr. Memory provides the allocation site of leaked memory if it has debug info; our tool would report the leak size and location.

* + If we compile the program with AddressSanitizer (-fsanitize=address), then running it through our tool would yield ASan’s own reports: one for the use-after-free (with a message "heap-use-after-free" and a stack trace) ([GitHub - japaric/rust-san: How-to: Sanitize your Rust code!](https://github.com/japaric/rust-san#:~:text=AddressSanitizer)), and one for the memory leak (ASan prints leak summaries at program end if enabled). We would parse those. ASan’s use-after-free message includes the allocation and free context, which we can use to inform the report (e.g., "memory was freed here... and later used here").
  + The leak report from ASan/LSan would list the leaked block’s size and allocation trace. We include that in the report as well, marking it as a leak issue.
* In the GUI, after running on this C++ binary, you might see:
  + **Use-After-Free** – description “Read of data after it was freed (address 0x...)", location use\_after\_free.cpp:9. The detail view shows two stack traces: one for the allocation/free site and one for the usage site.
  + **Memory Leak** – description “100 bytes allocated at use\_after\_free.cpp:11 not freed”, location maybe use\_after\_free.cpp:11 (if we define location as the allocation site for leaks). The detail shows the allocation trace (which might just be main in this case) and indicates no free happened. The user could click each to see details. This demonstrates detection of both an *illegal memory access* and a *resource leak*.

This example underlines how the tool helps catch two common mistakes in C++.

**Example 3: Data Race in a Multithreaded Program**

Consider a multi-threaded example (examples/data\_race.cpp):

#include <thread>

#include <atomic>

int counter = 0;

std::atomic<bool> done(false);

void worker() {

// increment counter without locking

for(int i=0; i<1000; ++i) {

counter++;

}

done.store(true, std::memory\_order\_release);

}

int main() {

std::thread t(worker);

// Wait until thread signals done

while(!done.load(std::memory\_order\_acquire)) { /\* spin \*/ }

// Main thread also modifies counter concurrently without waiting

counter++;

t.join();

return 0;

}

Here, counter is a shared non-atomic variable being written by both the main thread and the worker thread without synchronization, causing a data race. done is atomic just to coordinate termination.

* Without any instrumentation, this program might run and finish without obvious issues, but it has a race (undefined behavior).
* Running under our tool:
  + If compiled with ThreadSanitizer (-fsanitize=thread and using clang on Windows or running it on Linux), TSan will almost certainly report a data race on counter ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20export%20RUSTFLAGS%3D,rs%3A5%3A18%20%28example%2B0x86cec%29)) ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=WARNING%3A%20ThreadSanitizer%3A%20data%20race%20,rs%3A5%3A18%20%28example%2B0x86cec%29)). Our tool would capture a **data race** issue. The report would show that two threads (T0 and T1) accessed counter concurrently, one write in worker and one write in main, without proper locking, referencing the code lines (TSan gives a report with both stack traces and marks one as the previous write and one as the conflicting access ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=WARNING%3A%20ThreadSanitizer%3A%20data%20race%20,rs%3A5%3A18%20%28example%2B0x86cec%29))).
  + On Windows, if we didn’t compile with TSan (perhaps using MSVC or a compiler that doesn’t support it directly), our tool uses DRace. DRace would observe the memory accesses to counter and report a race condition. In the output, DRace might say something like "Data race detected on address 0x... in module exe, thread 1 and thread 2, both writing" with call stack info. We parse that into a data race report. The location would likely be data\_race.cpp:... for both threads’ write operations.
* The CLI might output:
* [!] Warning: Data Race detected on variable 'counter'
* -> Two threads wrote to the same memory without synchronization (see report for stack traces)

In the JSON/HTML report, we’d have an issue entry:

{

"type": "data-race",

"description": "Concurrent write/write race on global variable 'counter'",

"location": "data\_race.cpp:5 (global counter definition)",

"thread1": { "thread\_id": 1, "stack": ["worker() at data\_race.cpp:8", "std::thread::\_Invoke..."] },

"thread2": { "thread\_id": 2, "stack": ["main at data\_race.cpp:17"] }

}

(We could structure it like this in JSON to clearly separate the two threads). In HTML, we would list both stacks.

* In the GUI, after running, the results would show:
  + **Data Race** – description “Race condition on counter (two threads writing concurrently)”, location maybe data\_race.cpp:8 (where one write occurs, or just indicate "between worker() and main()"). The expanded detail shows thread 1’s stack and thread 2’s stack.

This example verifies the tool’s ability to catch concurrency issues that would be very hard to detect via normal testing.

**Validating on Windows in VS Code**

Throughout development, we used Visual Studio Code to compile and run these examples. Using the provided launch configurations, one can press F5 to run the GUI and test these examples, or run the CLI with an example path.

For instance, to test example 2 via VS Code:

* We set up a launch config for the CLI with args pointing to the built use\_after\_free.exe. After building that example, we hit F5 on "Debug Vulnerability CLI". We then watch the integrated terminal or debug console for the printed output and verify it matches expectations (use-after-free and leak reported). We can set breakpoints in our parsing code to see that it correctly identifies the patterns in Dr. Memory or ASan output.

This iterative process ensures the tool works as intended. We also include unit tests for some of the parser functions (for example, feeding in a known ASan report string to a function and asserting it produces the correct VulnerabilityReport struct).

**Build and Run Instructions (Windows + Visual Studio Code)**

Finally, here is a consolidated set of instructions to build and run the project in a Windows environment, assuming development is done in Visual Studio Code:

1. **Prerequisites Installation:**
   * Install Rust using [rustup](https://rustup.rs/). Ensure rustc --version and cargo --version work in a VS Code terminal. (Nightly toolchain is optional; for most use cases stable is fine.)
   * Install Visual Studio Code and the "Rust Analyzer" extension. Also install either the "C/C++" extension (for the MSVC debugger) or "CodeLLDB" for debugging support.
   * If you plan to test the sanitizers with C/C++ examples: Install LLVM/Clang for Windows (or Visual Studio Build Tools with C++ and enable AddressSanitizer). Also install Dr. Memory and DynamoRIO if you will test binaries without recompiling. (For quick start, you can skip this if you use our pre-instrumented examples.)
2. **Clone the Project:**
   * git clone https://github.com/yourusername/vuln-detector.git (replace with the actual repo URL). Then open this folder in Visual Studio Code.
3. **Build the Project:**
   * In VS Code, open the terminal (Ctrl+` ) and run cargo build. This will fetch dependencies and compile the core, CLI, and GUI. Alternatively, use the build task (Terminal > Run Build Task > Build All).
   * If the build is successful, you should have vuln\_detector\_cli.exe and vuln\_detector\_gui.exe in the target/debug directory.
4. **Configure External Tools (if needed):**
   * Make sure Dr. Memory’s path is added to your system PATH if you will use it. For example, if you installed it to C:\DrMemory, add that to PATH so drmemory.exe is found. Otherwise, edit the configuration file (e.g., in core/config.toml) to set drmemory\_path = "C:/DrMemory/bin/drmemory.exe" and similar for DRace if not using PATH.
   * (If using clang for C++ code, ensure llvm-symbolizer.exe is in PATH for better symbol resolution in ASan reports.)
5. **Run CLI Analysis:**
   * You can directly use the terminal: cargo run --bin vuln\_detector\_cli -- -t path\to\program.exe -c all -o output.json.
   * Or use the VS Code debugger: select "Debug Vulnerability CLI" and hit F5 (make sure to edit the args in launch.json to point to your target program). The program will run and you’ll see output in the debug console. After it finishes, open the output.json (or specified file) to see the results.
6. **Run GUI Analysis:**
   * Use cargo run --bin vuln\_detector\_gui to launch the GUI. Or use the "Run Vulnerability GUI" config in VS Code. A window should appear. Use the interface to select a program and start analysis.
   * Try it on one of the provided examples in the examples/ folder. For instance, compile the C++ example with cl /Zi use\_after\_free.cpp (for debug info) and then select that binary in the GUI.
   * View the results and experiment with saving a report.
7. **Interpret Results:**
   * Open the generated HTML report in a browser to verify it looks correct. The JSON can be opened in VS Code to verify the structure.
   * If using VS Code, you might also leverage the Problems panel: we could consider (in future) integrating with VS Code's diagnostics such that found issues appear as problems with file/line references, but that is optional.

By following these steps, a developer or user on Windows can build the project and use the vulnerability detector on their own binaries. The Visual Studio Code environment, with the provided config, streamlines development (e.g., one can modify how we parse outputs and quickly re-run on the examples to see if it still works).

**References:**

* Rust Unstable Book – Sanitizers: Details the types of bugs AddressSanitizer and ThreadSanitizer can detect ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=AddressSanitizer%20is%20a%20memory%20error,the%20following%20types%20of%20bugs)) ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20export%20RUSTFLAGS%3D,rs%3A5%3A18%20%28example%2B0x86cec%29)). This guided our design for which vulnerabilities to target.
* Dr. Memory Official Documentation: Confirms that Dr. Memory can detect out-of-bounds, use-after-free, uninitialized reads, and memory leaks on unmodified binaries ([Home](https://drmemory.org/#:~:text=Dr,reserved%20thread%20local%20storage%20slots)), which is crucial for handling non-instrumented programs.
* DRace (Siemens): Demonstrates a method for data race detection on Windows via dynamic instrumentation ([GitHub - siemens/drace: Data-race detector for windows applications - built on top of DynamoRIO](https://github.com/siemens/drace#:~:text=Image%3A%20REUSE%20status%20Image%3A%20CII,Best%20Practices)), allowing our tool to catch race conditions even when ThreadSanitizer is not available.
* JetBrains/CLion Sanitizers Documentation: Notes that AddressSanitizer is usable on Windows 10 with clang-cl under MSVC ([Google sanitizers | CLion Documentation](https://www.jetbrains.com/help/clion/google-sanitizers.html#:~:text=Sanitizers%20are%20implemented%20in%20Clang,are%20AddressSanitizer%2C%20ThreadSanitizer%2C%20and%20UndefinedBehaviorSanitizer)), reinforcing that our tool can rely on ASan on Windows for instrumented builds.
* Example sanitizer outputs from official sources were used to ensure our parsing covers all necessary information (e.g., stack traces in ASan reports ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=%24%20cargo%20run%20,rs%3A3%3A23)) and data race report format from TSan ([sanitizer - The Rust Unstable Book](https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html#:~:text=WARNING%3A%20ThreadSanitizer%3A%20data%20race%20,rs%3A5%3A18%20%28example%2B0x86cec%29))).