This lab explores stack-based memory corruption behavior and common runtime mitigations (stack canaries, ASLR, and related defenses) in a closed, instrumented environment. The goal is to observe how mitigations affect exploitability, identify reliable runtime artifacts for detection, and produce defensive guidance for SOC/IR teams. Experiments were executed in isolated virtual machines with offline analysis tools. Findings include a catalogue of observable indicators, a short triage checklist to reproduce the analysis safely for defensive teams, and recommendations for SIEM monitoring and low-effort detection rules.

**Objectives**

1. Understand how stack canaries and address-space layout randomization (ASLR) affect exploit attempts in a test program.
2. Use static and dynamic analysis to identify the vulnerable function and runtime artifacts that indicate attempted exploitation.
3. Produce non-actionable detection artifacts, IOCs, and SOC-focused playbook fragments for monitoring and response.
4. Document the lab environment, methodology, and ethical/responsible usage constraints.

**Materials & Environment**

* Isolated analysis VM(s) with snapshot capability.
* Instrumentation tools used for analysis and observation: static analysis (Ghidra / objdump), dynamic debugging (GDB), and OS-level file/log monitoring.
* The lab script (aslr-analysis-tool.py) used to drive the instrumented test harness in the lab.

**Security note:** Experiments were conducted only on controlled artifacts in dedicated lab VMs, no external systems were targeted.

**Conceptual summary of the script**

The uploaded script is an analysis utility used in an instrumented lab to probe a test harness and observe mitigation effects. Conceptually it:

* Attempts to identify the presence and value of a stack protection (canary) by sending inputs and observing whether the process behaves differently when certain bytes are present or absent.
* Probes for a return-address / function address leak (ASLR information) indirectly by observing side effects logged or created by the test program.
* Performs arithmetic/address calculations (using known offsets) to map observed addresses to library/base addresses for analysis and reporting.
* Constructs and sends a final test payload in the isolated lab to evaluate whether the derived addresses and mitigations allow arbitrary action in the lab harness.

This summary describes the *purpose* and *observations* of the script, it does not provide operational instructions or attack steps.

**Methodology**

**Do not** execute the script or any exploit against systems you do not own or control. The following is an outline meant for instructor-guided lab replication.

1. Prepare an isolated VM snapshot with the test binary and monitoring hooks (file/log watchers, debugger).
2. Use static analysis tools (e.g., Ghidra, objdump) to locate candidate functions that handle user input and potential persistence/log points. Record those code locations for reference.
3. Instrument the binary in the isolated VM (run under debugger or with logging enabled) and execute controlled probes from the lab script to observe process behavior and any generated artifacts.
4. Capture relevant runtime artifacts (files created, log lines, memory traces) and collect them as evidence for detection analysis.
5. Use the observed outputs to confirm conceptual behavior (e.g., whether canaries are in effect, whether ASLR causes address variance), then stop further experimentation and analysis captured traces offline.
6. Summarize findings and translate them into SOC-friendly artifacts (IOCs, detection heuristics, triage steps).

**Observations**

Report these categories of observations:

* **Vulnerability discovery:** the test binary exhibits stack-buffer overwrite characteristics in a controlled environment, the vulnerable function and input vector are identified by static inspection and confirmed by behavioral traces under instrumentation.
* **Mitigation behavior:** presence of stack protectors (canaries) and ASLR was confirmed conceptually, attempts to overwrite control data produced detectable failure modes under the lab harness. Document how mitigation presence changed observable behavior (process termination patterns, error logs).
* **Side-channel / artifact signals:** specific runtime artifacts were observed that indicated probing activity: creation of a specific log file, distinct log entries, short-lived files, or error messages captured in the lab. These artifacts are the basis for SOC detection.
* **Address mapping:** derived base addresses and addresses of interest were computed inside the lab (kept internal to the lab report), these were used for analysis and demonstration only. Do **not** publish these addresses externally.

**Non-sensitive Indicators & IOCs**

List only benign labels and identifiers useful for detection, **do not** include offsets, addresses, or exploit bytes.

* Filenames observed in the lab when a probe triggered: anime\_log.txt (example test artifact), monitor for unexpected file creation under monitored application directories.
* Log entry patterns: distinct error/exception messages emitted on abnormal input handling.
* Unusual process exits patterns: repeated abnormal exits of the same process within a short timeframe (e.g., repeated child process crashes), treat as possible indicator.
* Memory trace markers: named strings or suspicious token markers detected in memory dumps (report names only, no binary data).

**Detection & SOC Recommendations**

Design SOC rules and monitoring that are low-effort but effective in practice.

**Suggested high-level primitive:**

* File creation watch: alert on the creation of unexpected files in application directories (e.g., \*/anime\_log.txt in the lab sample).
* Crash-rate alerting: alert on high crash frequency for critical processes within a short window.
* Log anomaly detection: fingerprint and alert on specific error message templates (avoid capturing addresses).
* Memory-sampling policy: schedule transient memory captures in isolated test environments for triage v correlate with static analysis findings.

**Playbook fragment**

1. On alert, snapshot the VM and preserve logs.
2. Acquire a non-executable memory capture for analysis.
3. Correlate logs with recent deployments/changes.
4. Escalate to engineering for code fixes, create detection rules for the log or file artefacts.

**9. Defensive recommendations**

* Harden input validation and apply secure coding practices at the code review stage.
* Ensure runtime crash signatures are centrally logged and retained long enough for triage.
* Maintain a secure CI/CD pipeline so test binaries are clearly separated from production artifacts.
* Instrument test builds with additional logging that deliberately avoids leaking addresses to logs.

**10. Ethical & Responsible Disclosure**

* All experiments were executed only within lab VMs and snapshots.
* The repository is for academic review and SOC training; all instructions are defensive and non-actionable.
* If any real product or service is suspected to be vulnerable in production, follow responsible disclosure channels to report to the vendor rather than publishing details.