ISSN: 2473-4901
v12 n6799

"Fish in a Barrel": Lowering the Barrier for Vulnerability Discovery in a Software Exploitation Course

Andrew Kramer
andrew.kramer@dsu.edu

Utsab Adhikari
utsab.adhikari1@trojans.dsu.edu

Dakota State University
Madison, SD

Abstract

Real-world vulnerability discovery is among the most motivating experiences for students learning about software exploitation, but the barrier to entry is high. One must select a promising target, build unfamiliar code, apply one or more analysis tools, and separate real defects from the noise before they ever see a crash. We provide an experience report on a classroom activity, carried out as part of a graduate-level software exploitation course, that lowers this barrier by solving the tedious steps ahead of time and allowing students to focus immediately on triage and verification. We lovingly refer to this as the "fish in a barrel" approach. Taking advantage of recent advancements in LLM-driven automation, we construct a pipeline to build and statically analyze a broad array of open-source GNU software packages with the Clang static analyzer (*scan-build*) and a set of security-focused checkers, resulting in thousands of candidate warnings which we aggregate into a searchable, sortable, attack-surface-oriented dashboard. Students used the dashboard to hunt for genuine, reachable memory safety bugs among the noise. In this paper we describe the pipeline, characterize the warning to verified-bug funnel, and detail two representative bugs discovered during the activity (one simple buffer overflow discovered by the instructor and used as a demonstration, and one subtle heap-corruption bug independently discovered by a student through the dashboard). We then address retrospective research questions and reflect on technical and pedagogical considerations of the activity. The approach is inexpensive, easy to reproduce, and generalizes to nearly any corpus of open-source software.

Keywords: vulnerability discovery, static analysis, cybersecurity education

"Fish in a Barrel": Lowering the Barrier for Vulnerability Discovery in a Software Exploitation Course

Andrew Kramer and Utsab Adhikari

1. INTRODUCTION

A commonly stated goal for students studying software exploitation is to identify a real, previously unknown defect in a real software project (a "zero day") and demonstrate its security impact by writing an exploit. In our experience teaching CSC-748 (Software Exploitation) at Dakota State University, the single largest obstacle to that goal is not the exploitation itself, but the vulnerability discovery work that precedes it.

The application of modern static analysis tools against readily available open-source code is a low barrier-of-entry place for students to start, but even here, barriers exist. Before a student can practice the skills the course teaches against a real target, they must clear an array of hurdles, such as selecting a plausible target, learning an unfamiliar code-base, applying one or more complex analysis tools, and wading through a large volume of noisy, low-value tool output in search of the rare warning that corresponds to a real bug. Each of these steps is a place to get stuck and become discouraged, eroding the motivation that hands-on discovery is supposed to create.

In this experience report, we describe a classroom activity that deliberately removes that up-front friction so that students can skip right to the more exciting, rewarding work of analyzing potential bugs, reasoning about reachability, and verifying exploitability. In other words, instead of simply teaching our students to fish and then sending them out into the ocean, we fill a proverbial barrel full of fish and place it before them, providing a gentler, richer, more rewarding environment to hunt down their first bugs. Specifically, we pre-built and statically analyzed a large corpus of GNU software using the Clang static analyzer (*scan-build*), then presented the results to students via a triage-oriented web dashboard that students can easily browse, sort, and drill into. The tedious work of assembling the "barrel" was automated with substantial help from the popular LLM automation tool Claude Code, but the intellectually valuable work of distinguishing real, reachable memory safety defects from false positives was left to the students.

This paper represents an experience report documenting that process. We make four contributions: **(1)** a reproducible, LLM-assisted pipeline that builds and runs the Clang static analyzer (*scan-build*) across a large collection of GNU open-source packages; **(2)** an attack-surface-oriented triage dashboard that helped us organize 55,775 raw warnings to a browsable, security-impact prioritized set; **(3)** two examples of real, verified, memory-safety bugs found using the system (one identified by the instructor and one identified by a student); and **(4)** a reflection on the pedagogical advantages of supplying students with pre-staged vulnerability candidates.

The research artifacts are published online under the GNU General Public License (GPL). Details are provided in Section 8 below.

2. BACKGROUND AND RELATED WORK

Static Analysis for Bug Discovery

Static analysis tools inspect source code without executing it and identify features that may be defects. Their immense value, as well as their propensity for generating false positives, are both well documented. Bessey et al. (2010) describe the commercial application of static analysis to real-world code at scale and demonstrate its impact. Johnson et al. (2013) find that developers frequently abandon static analysis when output volume and false positives overwhelm the signal, frustrating their efforts. The Clang Static Analyzer and its associated *scan-build* utility (LLVM Project, 2026) provide a free, open-source, checker-based analysis engine that is relatively simple to wrap around an existing build system,

making it well suited to a teaching context. Beyond its default checks, the analyzer ships a set of *alpha*-stage security-relevant checkers (bounds checking, taint tracking, use-after-free detection, and insecure API usage) that are available but off by default. We enable these deliberately in order to increase the likelihood of discovering new, yet unknown bugs.

Vulnerability Discovery as Pedagogy

Hands-on, realistic offensive and defensive exercises are a long-standing tradition in computer security education. Bratus (2007) argues that offensive, hands-on practice teaches a way of reasoning about systems that defensive coursework alone cannot. Capture-the-flag (CTF) competitions put this concept into practice, and long-running events have been studied as educational tools for years (Vigna et al., 2014). A problem frequently identified in this literature is the gap between curated, contrived challenges (which are approachable, but artificial) and real-world targets (which are authentic, but with a far greater barrier to entry). Our classroom activity sits somewhere in between these ends of the spectrum. The targets are real, open-source GNU software projects, but the vulnerability discovery workflow is aided and bounded so that students are more likely to find a genuine bug more easily.

LLM-Assisted Tooling

Large language models have recently seen massive success in tasks related to code comprehension and vulnerability identification. Google's *Big Sleep* agent used an LLM to catch a previously unknown, exploitable memory-safety flaw in widely used real-world code (Big Sleep team, 2024). Heelan (2025) used OpenAI's *o3* model to uncover CVE-2025-37899, a remote zero-day vulnerability in the Linux kernel. Anthropic's evaluation of its new *Mythos* model reports rapidly improving performance on offensive cybersecurity tasks (Carlini et al., 2026). Compared to that frontier of autonomous bug discovery, our use of an LLM is indirect rather than direct, and pragmatic rather than novel, but still leans on the massive performance gains that have been recently realized in the research space. In our case, the LLM simply performed the heavy lifting of building many packages across a diverse range of build systems and generating the triage site, tasks whose scale would otherwise have made the exercise impractical in this context.

3. RESEARCH QUESTIONS

This activity was conceived as teaching practice, not as an official research study. Thus, we articulate the following research questions retrospectively.

RQ1 (Feasibility)
Can an LLM-driven pipeline build and statically analyze a large corpus of C/C++ software (approximately 200 GNU packages spanning various build systems) at a time and effort cost low enough for a single instructor to prepare as a course activity?

RQ2 (Signal to noise)
Can a corpus of tens of thousands of raw *scan-build* warnings be organized and presented as a tractable set of candidate memory-safety bugs for students to examine, by grouping them into sortable, filterable categories such as attack surface, reachability path length, or severity?

RQ3 (Barrier to entry)
Does supplying a pre-computed, browsable "barrel" of candidate bugs lower the barrier enough that students in a graduate-level software exploitation course can discover and verify real software defects?

RQ4 (Engagement)
Does the approach enhance student engagement with the material? We did not run a controlled comparison but report qualitative classroom observations and a case study as example.

4. METHODOLOGY

At a high-level, our methodology was to assemble a corpus of open-source software, generate a generic harness to build it, incorporate static analysis into the build process, generate a searchable triage dashboard from the static analysis findings, and present that triage dashboard to students for them to explore. This is shown in Figure 1.

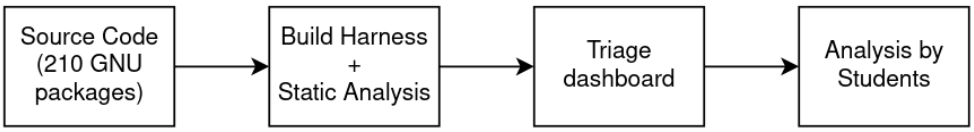

Figure 1: Research methodology.

Corpus Assembly

We chose to target only C/C++ code as this matched the established focus and goals of the course. GNU publishes a large corpus of C/C++ open-source software, making it a convenient choice of target(s). In total, we mirrored 210 C/C++ source packages from the GNU project FTP server, ranging from small utilities to large systems (*binutils*, *glibc*, *guile*, *ghostscript, etc*). We chose GNU software because it is freely available, realistic, security-relevant, uniformly licensed, and diverse.

Build Harness

One central obstacle is that the corpus of GNU projects does not share a single unified build system. We leaned heavily on LLM-assistance to solve this problem, generating a high-level harness to handle all the disparate build tasks. Our harness (*run_scan.sh*) classifies each project by build system and dispatches it to the appropriate recipe, including standard *autotools*, *autotools* requiring a bootstrap (*autoreconf*, *autogen.sh*, *bootstrap*, etc), *CMake*, *Meson*, plain *make*, and various bespoke configure scripts. The wrapper is deliberately non-destructive, meaning that rather than running *make clean* (destroying any past build progress), it simply bumps source-file modification times so objects rebuild under the analyzer without deleting artifacts. This also neutralizes autotools regeneration recipes that would otherwise run. Each project records a status (*BUILT_OK*, *BUILT_PARTIAL*, *CONFIGURE_FAILED*, etc.), making the run resumable. Preparing and debugging this harness across the full corpus is exactly the kind of labor that LLMs are ideal for automating (RQ1).

Checker Selection

We then wrapped each build in Clang *scan-build* (clang version 14.0.0) and enabled 25 explicit security-relevant checkers beyond the defaults, drawn from the *security.**, *alpha.security.**, *alpha.unix.**, *alpha.core.**, and *optin.portability.** families. These include checkers related to array-bounds violations, heap buffer size-overflow, heap use-after-free, taint propagation, insecure API usage, and CERT-derived checks. According to the National Security Agency (NSA) and Cybersecurity and Infrastructure Security Agency (CISA) these bug classes represent some of the most prevalent and high impact (NSA & CISA, 2025). These alpha-stage non-default checkers, although noisy, may surface additional interesting defects (Leong, 2022), providing students more opportunities to identify new and unique bugs.

Triage Dashboard

Once static analysis has completed, a generator script (*generate_index.py*) scrapes *scan-build*'s per-project HTML into an aggregated static webpage, with a sortable top-level index, a machine-readable *index.json*, and per-project pages linking back to the analyzer's interactive reports. Each warning is grouped by a keyword, chosen by the generator script, based heuristically on the checker name, including *critical-class* (use-after-free, out-of-bounds, double-free, taint sinks, *malloc* overflow, format string), *insecure-API* (unbounded *strcpy*, *sprintf*, *memcpy*, etc.), and others. The dashboard also groups projects by inferred real-world attack surface, also determined heuristically by the generator script, so

ISSN: 2473-4901
v12 n6799

a student can filter targets by likely security impact (e.g. network-reachable daemons) rather than from an unfiltered list (RQ2). It is worth emphasizing that these labels are determined heuristically based on checker names and LLM-derived project scope, and should not be interpreted as verified claims about bug validity, reachability, or security impact. These are simply tools to help filter out the noise. An image of the dashboard landing page is shown in Figure 2.

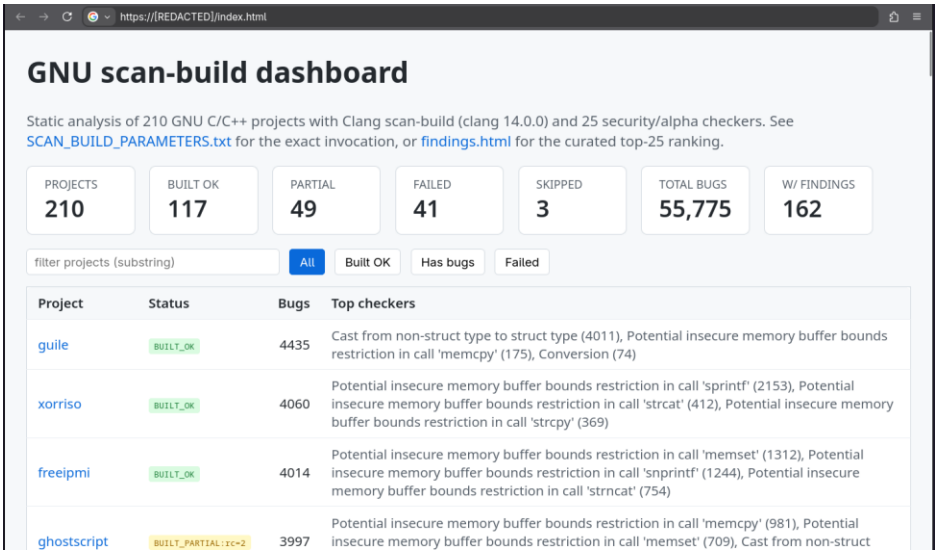

Figure 2: The triage dashboard landing page.

Classroom Integration

The triage dashboard was presented to a class of 32 graduate-level students (12 face-to-face synchronous, and 20 online asynchronous). It was presented during the fifteenth week of a sixteen-week semester, during a time which had been formally designated in the syllabus as open, flexible, study or exploration time. The triage dashboard was provided as an optional activity for students to engage with at their own pace and desire, rather than a required activity with formal evaluation. No grade, points, or other incentives were offered, beyond the student's own intrinsic motivation. We did not ask students to log the specific amount of time or effort spent on this activity.

Technical Resources

The technical work of assembling a corpus, building each project, running static analysis, and implementing the triage dashboard was performed on a desktop computer with the assistance of Claude Code. Specifically, the computer contained an AMD Ryzen 3700X (8-core, Zen 2) processor, 32GB of RAM, and a 1TB PCIe 3.0 SSD, running Ubuntu 22.04 LTS. Claude Code was configured to use Anthropic's Opus 4.7 model under the Pro subscription tier.

Although the precise amount of time and tokens required to produce this work were not recorded, we report the estimated burden based on session logs. All of the preparatory work was conducted in a single evening, during a work session of approximately two and a half (2.5) hours, using 19 LLM prompts. The initial source-code acquisition, build system design, and static analysis were performed in approximately one hour, using seven LLM prompts. Bug triage across all projects took approximately one hour and an additional seven LLM prompts. Construction of the triage dashboard consumed about one half (0.5) hour and five LLM prompts. We note however that this burden is tilted heavily towards up-front work. Now that tooling has been constructed, reproduction of this activity for future classes would require significantly less time and prompting.

The technical expertise required to perform this work was moderate. Prior understanding of the data set, static analysis tooling, the technical approach, and the end goal was necessary in order to author

ISSN: 2473-4901
v12 n6799

prompts which produced the artifacts, however specific technical experience using the tools was not. By leaning on the immense technical capabilities of modern LLMs, a researcher or educator with only high-level understanding should, in theory, be able to reproduce this work.

5. RESULTS

Build and Static Analysis Coverage

Of 210 projects in the corpus, 166 produced usable analysis output (fully or partially built) within a time and effort allotment commensurate with standard class preparation work. The remainder failed at configure or build, chiefly due to complex dependency issues or toolchain incompatibilities. Table 1 summarizes the run. We emphasize that our goal was not to spend whatever time and resources were necessary to achieve perfect results, but rather to determine what results were possible using the limited time and resources a teacher likely has available for normal class activity preparation. The fact that a single automated pass yields usable security analysis for a clear majority of a large, unmodified corpus is the core result. The "barrel" can be filled with "fish" quickly, cheaply, and easily (RQ1).

Metric	Count
Projects in tree	210
Built successfully (full analysis)	117
Built partial (partial analysis)	49
Total projects with usable analysis	166
Total projects with >= 1 bug warning	162
Total bug warnings	**55,775**
Warnings classified as critical severity	3,096
Warnings classified as insecure-API	34,275

Table 1: Run summary across the GNU software corpus. Values are derived from the analyzer output.

The Warning-Bug Funnel

The corpus produced 55,775 total warnings across 162 projects. Of these, 34,275 are insecure-API warnings and 3,096 fall in the critical bug classes most associated with memory-corruption exploitability. The educational impact here is the shape of the reduction, shown in Figure 3. A student need not triage 55,775 items, but rather can start from the critical-class subset within a high-value attack surface, then narrow it to the handful of warnings that survive manual reachability analysis, or simply go looking for any specific bug class or pattern they wish right away. The dashboard exists to make each narrowing step as simple as a sort or a filter operation rather than a full research project on its own.

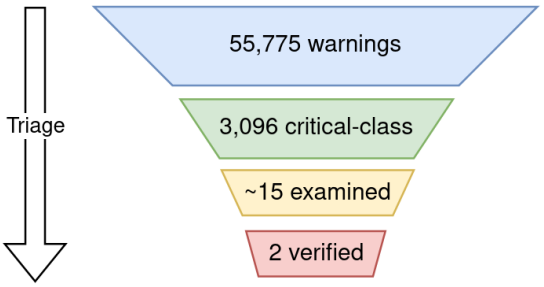

Figure 3: The triage funnel. Raw warnings reduce to bug-class heuristically, and then to a small number of confirmed, reachable bugs through manual verification.

Prioritization by Attack Surface

Table 2 lists the top ten projects with the most raw critical-class warnings. Table 3 narrows that focus to the top ten projects with network-reachable daemons and clients, the attack surface most interesting

and impactful to new learners. This allows students to quickly narrow and focus their attention on the bugs that most interest them.

Project	Critical	Total	Status
ghostscript	260	3,997	BUILT_PARTIAL
mifluz	136	2,153	BUILT_PARTIAL
gettext	105	1,399	BUILT_OK
bison	102	329	BUILT_OK
mailutils	93	1,106	BUILT_OK
xorriso	88	4,060	BUILT_OK
electric	80	3,050	BUILT_OK
cim	76	230	BUILT_PARTIAL
coreutils	64	517	BUILT_OK
groff	64	304	BUILT_OK

Table 2: Projects ranked by critical-class warning count.

Project	Bugs	Critical	Insecure API
mailutils	1,106	93	608
inetutils	672	51	428
mc	314	51	178
dico	478	50	270
gnubatch	1,098	42	690
lsh	210	42	79
screen	392	42	276
swbis	1,305	37	572
wget	349	28	302
pies	293	27	190

Table 3: Network-reachable daemons and clients, ordered by critical-class warnings, indicating potential high-impact starting points for triage.

6. CASE STUDIES

We present two verified memory-safety defects surfaced through the dashboard as examples, one used as an instructor demonstration, and one discovered independently by a student. Both illustrate that the "barrel" contains real "fish", and both discoveries were initiated from a *scan-build* warning in minutes rather than days.

Instructor Demonstration Bug

To model the workflow for the class, we followed the dashboard from an insecure-API warning to a confirmed overflow in the *gnushogi* program. Specifically, in *main.c* (Listing 1), the command-line option handler copies the *savefile* and *listfile* arguments directly into fixed-size global buffers with no bounds checks, a classic buffer overflow pattern.

ISSN: 2473-4901
v12 n6799

```
// from globals.c

char savefile[128];
char listfile[128];

// from main.c

case 'L':
    argc--;
    argv++;
    if (argc > 0)
        strcpy(listfile, argv[0]);  // may overflow!
    break;

case 's':
    argc--;
    argv++;
    if (argc > 0)
        strcpy(savefile, argv[0]);  // may overflow!
    break;
```

Listing 1: Code from *gnushogi*, showing two global array buffer overflow bugs.

Passing an argument 128 characters or longer to either of these parameters, as shown in Listing 2, results in a buffer overflow, which may result in an immediately observable *segmentation fault* error depending on the environment.

```
$ gnushogi -L $(python -c 'print("A"*4096)')
```

Listing 2: A long command line argument passed to the *gnushogi -L* (*listfile*) parameter, triggering the bug.

This is an intentionally simple example. Its real-world impact is limited (the input is a local command-line argument), but it is pedagogically ideal in that a novice can read the warning, read the two relevant lines of source, form a hypothesis, and confirm it with a single command. It also demonstrates a fact to students that they often doubt: that real bugs still exist and are genuinely within their reach to find. It serves as a source of excitement and motivation.

Identification of this issue was quick and cheap. Static analysis highlighted invocations of the strcpy() function (a classic vulnerability pattern), making them easy to inspect. Further filtering for invocations that did not contain string constants, and where the obvious *argv* variable was passed as an argument, narrowed the candidate pool. Ultimately, we examined fewer than 10 candidate bugs to identify this specific one.

Student-Discovered Bug

Following an insecure-API warning from the dashboard, one of the authors (then a student in the course) identified a heap buffer overflow in GNU Interactive Tools (*gnuit*). Its *gitfm* file manager lets users bind keys to commands in its configuration file, where each *[GITFM-Keys]* entry supplies a command *name* and a command *body*. The command *body* may also contain macros which are expanded when used. When a bound key is pressed but its body fails to expand, *gitfm* attempts to build an error message in a heap buffer using *sprintf()*, as shown in Listing 3. Unfortunately, that heap buffer is sized based on the wrong string. The heap buffer size was previously calculated using the *pressed key sequence*, rather than the *command name* it actually prints, a seemingly small programming typo with catastrophic results.

```
// from git.c
// (bound key pressed, command body invalid)

// sized from key seq rather than command name
msg = xmalloc(80 + strlen(ks->key_seq) + 1);

sprintf(msg, "%s: invalid command on key sequence %s !",
        command->name, command->sequence);
```

Listing 3: Code from *gitfm*, showing a heap buffer (*msg*) sized based on the wrong string.

After supplying an overly long *command name* (e.g. *AAA…*) and an invalid *command body* (e.g. *%j*) in the *gitfmrc* config file, pressing the associated hotkey results in heap buffer overflow. Listing 4 demonstrates triggering this bug using the *F1* hotkey.

```
// …/gnuitrc.common

[GITFM-Keys]
F1 = AAA …(~1,000 A's)… AAA; %j ;
```

Listing 4: Trigger for the *gitfm* heap overflow bug.

Although the student did not track the specific number of candidate bugs they traced from the dashboard before finding this specific bug, they estimate it to be five or six. Regardless, this is the existence proof RQ3 seeks. A student, new to vulnerability research, used the dashboard to find and verify a novel, non-trivial memory-safety bug.

7. DISCUSSION

Research Questions

RQ1 asked whether an LLM-driven pipeline could build and statically analyze a large corpus of C/C++ software at a time and effort cost low enough to be feasible for a single instructor. This question is answered affirmatively. The "barrel of fish" was constructed in under three hours and under 20 LLM prompts, using tools readily available to most university faculty.

RQ2 asked whether a corpus of tens of thousands of raw *scan-build* warnings can be organized and presented as a tractable set of candidate memory-safety bugs for students to examine. This question is also answered affirmatively. The initial set of 55,775 warnings was successfully distilled to two verified bugs.

RQ3 asked whether supplying a pre-computed, browsable "barrel" of candidate bugs lowered the barrier enough that students in a graduate-level software exploitation course could discover and verify real software defects. We answer this question affirmatively as well. A student was able to find a real memory safety bug using the triage dashboard.

RQ4 asked whether the approach enhanced student engagement with the material. Although we did not run a formal experiment, we demonstrate qualitatively that a student who engaged with this activity was able to discover a real memory safety bug, contributing to a sense of accomplishment in the subject.

Benefits and Drawbacks of "Fish in a Barrel"

This activity removes several potential sources of friction for students: target selection, building unfamiliar code, tool setup, and warning triage. Proactively removing those pain-points makes real-world vulnerability hunting more approachable for new learners. This activity deliberately preserves the core skills the course is about: reading unfamiliar code, reasoning about whether a flagged construct is reachable from untrusted input, and demonstrating impact. This ensures students still have opportunities to practice the skills they've learned and feel a sense of accomplishment in their success, without entirely offloading the intellectual work to the instructor or an LLM. On the other hand, it is

worth considering whether other valuable learning opportunities are lost by cutting out the initial steps. Shortcuts allow students to reach the destination faster, but may result in missed opportunities along the way. A balance must be struck. We leave this as an open question for future research and future iterations of this activity.

The Role of the LLM

The LLM's chief contribution to this activity was scale rather than direct vulnerability identification. It made it feasible for one instructor to build and analyze a large software corpus and to generate a novel, usable triage interface within a short period of time, work that would otherwise be prohibitive to manually prepare for a single semester. The important security judgments (which warnings are real and reachable) remained a human activity, performed by the students. We believe this is the right workload allocation in an education setting. We automate filling the barrel, not catching the fish.

Transferability

Nothing in the pipeline is inherently specific to GNU software or to *scan-build*. The same approach could apply to any collection of open-source software and to other analyzers. An instructor could refresh the "barrel" each term, scope it to a specific domain (e.g. only network daemons), or fit it to a specific learning objective.

Limitations

Several limitations are worth noting. First, bug criticality labels were determined using a keyword heuristic on *scan-build* checker names and do not necessarily imply that the bug is real or reachable. These labels are simply a triage aid, and false positives are expected to be abundant, as literature predicts (Johnson et al., 2013). Second, the pipeline only analyzed the projects that either partially or successfully built. Only 166 of 210 projects yielded usable output, and partial builds analyzed only the software components that compiled, so coverage is known to be incomplete and non-uniform. Third, our evidence for RQ3 is simply an existence proof (two verified bugs, one instructor and one student), and our engagement evidence for RQ4 is observational. This activity was conducted during a single course offering, with a small cohort, and with no control group. We therefore make no quantitative claim about learning benefits over alternative approaches. The appropriate conclusion here is that the approach enabled students to discover real bugs, not that it is proven superior to alternatives. This is an experience report, not a formal study.

Responsible Disclosure

Both bugs discussed in this experience report (the overflow in *gnushogi* and the overflow in *gnuit gitfm*) were responsibly disclosed to the project developers more than 90 days prior to publication of this document.

The *gnushogi* bug was reported to the developers by email on July 10, 2026, but has not yet received a response. The most recent version of *gnushogi* was released February 17, 2014, meaning this project may simply be abandoned. Similarly, the *gitfm* bug was reported to the developers by email on July 9, 2026, but has also not yet received a response. The most recent version of *gnuit* (the project containing *gitfm*) was released February 23, 2009, meaning it too may simply be abandoned. At the time of writing we do not expect or anticipate that these bugs will receive CVEs.

Regardless, both bugs are extremely limited in security-impact because they only affect locally-run command line utilities, require highly specific user-input to trigger, and could not result in remote code execution or privilege escalation even if exploited. These are real memory-safety bugs, but arguably not "vulnerabilities".

The potential for abuse is nearly nonexistent, so we deem the risk of disclosure here to be negligible. In an abundance of caution, we opt not to publish exploitation details.

8. CONCLUSIONS

We set out to lower the barrier to entry for vulnerability discovery in a software exploitation course while leaving room for students to exercise the skills the course exists to teach. By automating the assembly and static analysis of a large corpus of real GNU software and presenting it through an organized triage dashboard, we let students spend their effort on reading code and verifying bugs rather than on tedious build systems and tool setup. The pipeline built and analyzed 166 projects and surfaced 55,775 candidate warnings. The dashboard reduced that volume to a workable funnel and the approach produced verified reachable memory-safety bugs, including one found independently by a student. The "fish in a barrel" framing captures the design intent, lowering the barrier for entry, while leaving the most valuable, rewarding parts for the students to solve on their own.

Future Work

Natural extensions of this activity might include: a controlled study of engagement and learning outcomes compared to unaided bug hunting, refreshing the corpus each term and scoping it to specific attack surfaces, or incorporating dynamic analysis (e.g. fuzzing) to pre-stage crashing inputs alongside static warnings. Future research could also study whether and how LLM assistance can help students with bug verification, without compromising the valuable learning opportunity that exists in that final stage of analysis.

Artifact

The build-and-analyze harness (*run_scan.sh*), the dashboard generator (*generate_index.py*), the checker configuration, and the machine-readable results (*index.json*, *findings.md*) are available at https://github.com/Rewzilla/scan-build-dashboard. To keep the repository small, it provides only the scripts, configuration, and aggregated results with instructions to reproduce the analysis, but not the GNU source corpus or the per-project HTML reports.

9. ACKNOWLEDGEMENTS

We thank the students of CSC-748: Software Exploitation (Spring 2026) for their enthusiastic engagement with this activity. Additionally, we thank Dr. Shawn Zwach for reviewing the final manuscript prior to publication.

Author Contributions

Andrew Kramer taught the course, designed and built the static-analysis pipeline and triage dashboard, and primarily wrote the experience report. Utsab Adhikari, then a student in the course, independently discovered and verified the *gitfm* heap overflow bug and reviewed the manuscript.

AI Usage Statement

Generative AI (specifically Claude Code) was used to build the research artifacts as described above, as well as to aid in the initial manuscript outline. All written work is the product of the human authors, who have fully reviewed, edited, and maintain responsibility for it.

10. REFERENCES

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., & Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 66-75. doi:10.1145/1646353.1646374

Big Sleep team. (2024). From Naptime to Big Sleep: Using large language models to catch vulnerabilities in real-world code. *Project Zero*. https://projectzero.google/2024/10/from-naptime-to-big-sleep.html (Accessed 2026)

Bratus, S. (2007). What hackers learn that the rest of us don't: Notes on hacker curriculum. *IEEE Security & Privacy*, 5(4), 72-75. https://doi.org/10.1109/MSP.2007.101

Carlini, N., Cheng, N., Lucas, K., Moore, M., Nasr, M., Prabhushankar, V., Xiao, W., Angulu, H., Ben Asher, E., Bow, J., Bradwell, K., Buchanan, B., Forsythe, D., Freeman, D., Gaynor, A., Ge, X., Graham, L., Guru, K., Lakhani, H., McNiece, M., Mehrara, M., Nichol, R., Pirzada, A., Porter, S., Terzis, A., & Troy, K. (2026). Assessing Claude Mythos Preview's cybersecurity capabilities. *Anthropic*. https://www.anthropic.com/research/mythos-preview

Heelan, S. (2025). How I used o3 to find CVE-2025-37899, a remote zeroday vulnerability in the Linux kernel's SMB implementation. *Sean Heelan's Blog*. https://sean.heelan.io/2025/05/22/how-i-used-o3-to-find-cve-2025-37899-a-remote-zeroday-vulnerability-in-the-linux-kernels-smb-implementation/ (Accessed 2026)

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)* (pp. 672-681). IEEE. https://doi.org/10.1109/ICSE.2013.6606613

Leong, L. (2022). MindShaRE: When MySQL Cluster encounters taint analysis. *Zero Day Initiative*. https://www.zerodayinitiative.com/blog/2022/2/10/mindshare-when-mysql-cluster-encounters-taint-analysis

LLVM Project. (2026). *Clang static analyzer*. https://clang-analyzer.llvm.org/ (Accessed 2026)

National Security Agency (NSA), & Cybersecurity and Infrastructure Security Agency (CISA). (2025). Memory safe languages: Reducing vulnerabilities in modern software development (Report No. U/OO/172709-25; PP-25-2574). https://media.defense.gov/2025/Jun/23/2003742198/-1/-1/0/CSI_MEMORY_SAFE_LANGUAGES_REDUCING_VULNERABILITIES_IN_MODERN_SOFTWARE_DEVELOPMENT.PDF

Vigna, G., Borgolte, K., Corbetta, J., Doupe, A., Fratantonio, Y., Invernizzi, L., Kirat, D., & Shoshitaishvili, Y. (2014). Ten years of iCTF: The good, the bad, and the ugly. *USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE)*.