Exploiting Buffer Overflows: A C Program for Software Security Education

Mohamed Elwakil
United States Coast Guard Academy

Buffer overflow vulnerabilities remain a pervasive threat in software systems, yet students often only encounter them in theory. This experience report presents LoginApp, a small C application intentionally designed with a buffer overflow flaw that allows an attacker to bypass authentication without a valid password. Used as a hands-on exercise in a junior-level software engineering course within a cybersecurity program, LoginApp was employed to bridge the gap between theoretical knowledge and practical exploitation. We describe the design of this instructional tool and its integration into the classroom, including a preliminary toy example that illustrates unsafe C functions, and detail how the buffer overflow exploit is constructed. We discuss the learning context and analyze the pedagogical outcomes – notably that only 1 of 47 students solved the challenge – to derive insights and identify improvements for future offerings. Our findings suggest that while LoginApp effectively engages students and instills a deeper appreciation of secure coding, proper scaffolding and safeguards are crucial to maximize learning and prevent misuse.

1. Introduction

Buffer overflow vulnerabilities remain among the most prevalent and dangerous software flaws—MITRE (2020) documents over 10,000 instances, nearly a quarter rated severe. While high-level languages like Java automatically prevent such errors, C and C++ place the burden on programmers, making them especially vulnerable.

Although buffer overflows are a standard topic in security education, students typically encounter them only through theory or code snippets, rarely through hands-on exploitation. This gap between abstract knowledge and practical skill leaves many unprepared for real-world threats (Zouahi & Talhi, 2023; Mirkovic & Peterson, 2014).

To address this, we introduce LoginApp, a deliberately vulnerable C program that simulates a login system and can be compromised via a buffer overflow to bypass authentication. Designed as a turnkey classroom

module, LoginApp enables students to experience firsthand how a small coding flaw can lead to a critical breach. By engaging in a controlled exploit, learners move beyond passive understanding toward active security reasoning—an approach supported by research on experiential cybersecurity education (e.g., CTF-style labs).

This paper presents the design, classroom deployment, and pedagogical insights from using LoginApp, along with reflections on its effectiveness, limitations, and future enhancements.

2. Literature Review and Related Work

Effective cybersecurity education requires hands-on experience, as passive lectures often fail to equip students with the practical skills needed to address real-world threats (Alnajim et al., 2023; Ramezanian & Niemi, 2024). Buffer overflows—among the "big three" software vulnerabilities (SANS Institute, 2006)—are a critical focus for secure coding instruction due to their prevalence and impact.

To help students grasp these low-level concepts, researchers have developed interactive tools. Zhang et al. (2020) introduced a web-based visualization that improves comprehension by letting students step through memory corruption. Resch (2023) used an ARM emulator and debugger to give students direct insight into how overflows alter program state. While effective, such approaches often require specialized tools or systems knowledge.

In contrast, Capture-the-Flag (CTF) challenges offer gamified, practical experience but are typically standalone puzzles lacking pedagogical scaffolding. *LoginApp* bridges this gap: it delivers a realistic, relatable exploit (authentication bypass via data-only overflow) using only standard C and a typical development environment. Unlike abstract visualizations or open-ended CTFs, *LoginApp* is designed as a structured, classroom-ready module that guides learners from vulnerability recognition to mitigation—making hands-on security accessible even in general software engineering courses. In the following sections, we detail how the *LoginApp* exercise was implemented in the classroom and evaluate its effectiveness, while addressing various recommendations and challenges highlighted in prior work. Our focus is on the pedagogical insights gained from this experience: what students learned, where they struggled, and how we as instructors adjusted our approach.

3. Educational Design and Methodology

Learning Objectives

Before deploying the exercise, we defined specific learning objectives to align with our course outcomes. By the end of the *LoginApp* module, students should be able to:

1. Explain buffer overflow basics: Define what a buffer overflow is and explain in general how an attacker can exploit such a flaw.
2. Identify common causes: Identify common situations in C/C++ code where buffer overflows may occur (especially the use of unsafe string handling functions).
3. Analyze a vulnerable program: Analyze the structure and workflow of the *LoginApp* program to understand exactly how vulnerability is introduced and exploited.
4. Propose secure alternatives: Propose or create more secure alternative implementations of the program to mitigate buffer overflow vulnerabilities.

These objectives map to increasing levels of understanding—from basic knowledge and comprehension (objectives 1 and 2) to application and evaluation (objective 3), and finally to the synthesis of improved solutions (objective 4). In other words, the exercise was intended not only to illustrate the concept of buffer overflow but also to engage students in critical thinking about how to prevent such vulnerabilities.

Classroom Context

The exercise was deployed in a junior-level Software Engineering course that is part of an undergraduate cybersecurity degree program. The 47 students in the class had a solid foundation in general programming, having taken introductory courses in Python and C/C++. Most had also taken operating systems and databases courses. However, none (aside from one exceptional student discussed later) had prior hands-on experience with actually exploiting vulnerabilities. The course primarily focused on software development practices, so this security module was a special segment designed to raise awareness of secure coding. We scheduled the *LoginApp* exercise for the middle of the semester, after covering basic secure coding principles in lectures. By that point, students had been

warned about dangerous library functions (like using *gets()* or unchecked *strcpy*), but they had not yet seen a full exploit in practice.

Procedure

The module spanned a single class period and included a follow-up assignment. We began with a brief lecture reviewing buffer overflow basics and the importance of bounds checking. To ensure students understood the low-level behavior of strings in C, we presented a toy example before introducing the main tool. This toy program in Figure 1 has two static character arrays and deliberately used unsafe input functions, resulting in an unexpected output. Specifically, the program asks for two inputs.

```
char first[3];
char last[3];
char full[6];

printf("Please type your first name ");
scanf("%s", first);
printf("Please type your last name ");
scanf("%s", last);

strcpy (full, first);
strcat (full, " ");
strcat (full, last);

printf("Your full name is '%s'", full);
```

Figure 1: Toy program that is susceptible to buffer overflows due to unsafe string functions

When a user enters *"AB"* for the first input and *"DEF"* for the second, most students predict the output will simply concatenate into *AB DEF*. In reality, due to a buffer overflow in how the program stores these inputs, the actual output was *▽*. This surprise outcome immediately caught the students' attention and vividly demonstrated how writing past a buffer's end can alter the data in adjacent memory locations. We discussed why this happened, highlighting the distinction between safe vs. unsafe string functions in C. For instance, using *gets()* or *scanf("%s")* without length limits can overflow a buffer, whereas functions like *fgets()* or *scanf("%5s")* (with an appropriate field width) could prevent the overflow.

After this demonstration, we introduced the *LoginApp* tool—the main exercise. We explained that *LoginApp* is a simple authentication program written in C, and we walked through its intended functionality (without yet revealing the vulnerability).

Once students understood how *LoginApp* should work normally, we shifted to the security challenge: Could they log in without knowing the correct password? We told them to assume the role of an attacker who has the program's source code and at least one valid username. This models a

situation where the source might be leaked, or the attacker has reverse-engineered the binary. The task was phrased openly: *"Find a way to bypass the authentication — i.e., make the program grant access without entering the real password."* We clarified that this likely involves exploiting a bug in the code. Importantly, we did not explicitly instruct them to perform a buffer overflow; rather, we let them reason it out from the hints (e.g., the presence of *strcpy*, fixed-size arrays, the suspicious use of *scanf* with **%s**, and the prior toy example demonstrating overflow). Students were allowed to work in pairs and use any tools at their disposal (code editors, compilers, even debuggers) to experiment with the program. The challenge was allotted roughly 20–25 minutes of class time.

During this hands-on period, the instructor walked around to observe and give light hints if students were completely stuck. We deliberately did not provide a step-by-step guide initially, as the intent was to let them engage in some adversarial thinking on their own.

Data Collection: Our evaluation of the exercise's impact was primarily qualitative. After the challenge, we held a debriefing where we revealed the solution and dissected the exploit's mechanics. We then gathered student feedback in two ways: (1) an interactive discussion in class about how they approached the problem, how they felt, and what they learned, and (2) an anonymous short survey afterward with a few questions about the experience (e.g., *What was most surprising? How difficult was it? Suggestions for improvement?*). We did not administer a formal pre-test/post-test on buffer overflow knowledge, but we did ask students to reflect on whether their understanding of buffer overflows had changed as a result of the exercise. Additionally, we noted the number of students (or teams) that successfully exploited the program within the allotted time (which, as we will discuss, was essentially just one individual).

4. The *LoginApp* Vulnerability and Exploit

Implementation Details

The core of *LoginApp* is implemented in C and consists of a simple *main* function orchestrating the input/output and a few helper functions (for user lookup and hashing). Key variables (allocated on the stack) include a list of valid usernames and their password hashes (e.g., stored in an array of structs or parallel arrays), as well as several fixed-size buffers to hold

user-supplied and processed data. In particular, the program defines something akin to:

```
char username[5];
char password[4];
char hashedStoredPass[4];
char hashedInputPass[4];
```

The program's intended flow is as follows: it prompts for a username and reads it into the *username* buffer. Next, it checks this username against the list of authorized users. If the username is found, the program uses *strcpy* to copy the corresponding stored hash from the "database" into the *hashedStoredPass* buffer. (If not found, it prints an error and exits.) After fetching the stored password hash, the program prompts the user for their password. It then reads the password from the input buffer into the *password* buffer (using a standard unsafe input function like *scanf("%s", password)* or *gets(password)* in the intentionally flawed version). The program computes the hash of the entered password, producing a 4-character hexadecimal string, and stores it in *hashedInputPass*. Finally, it uses *strcmp* (or equivalent) to compare *hashedInputPass* with *hashedStoredPass*. If they match, the login is successful and an access-granted message is displayed; if not, it reports a login failure.

Under normal conditions (with a correct username/password), this sequence works as intended. But there is an inherent weakness: the fixed-size buffers and unsafe functions leave the door open to buffer overflows. Specifically, the *password* buffer is only 8 bytes long in this design, meaning it can hold at most a 7-character string plus the null terminator. If a user enters a longer password, it will overflow into adjacent stack memory. In C, local variables are typically laid out contiguously in memory, in the order they are declared (though this is not strictly guaranteed, compilers usually arrange them sequentially). In our case, the memory for these buffers is arranged as: *[username] [password] [hashedStoredPass] [hashedInputPass]* on the stack. We intentionally wrote the code such that the *hashedStoredPass* buffer is placed next to the *password* buffer in memory. That way, an overflow of *password* could overwrite data in *hashedStoredPass*.

Nature of the Vulnerability

The bug in *LoginApp* is a classic stack-based buffer overflow. Uniquely, it does not overwrite a return address or function pointer; instead, it overwrites an adjacent data buffer (a non-control data attack). This is often referred to as a *data-only* attack – the overflow corrupts application data (in this case, a password hash) to manipulate program logic, rather than hijacking the program's instruction flow. Data-only buffer overflow exploits are highly relevant, as many real attacks focus on modifying security-critical variables (e.g., flags, credentials) in memory when modern defenses block control-flow hijacking. Our scenario demonstrates that even without injecting shellcode or altering a return pointer, a buffer overflow can cause an unauthorized privilege escalation – here, logging in without the real password.

Crafting the Exploit Input

To exploit the vulnerability, an attacker needs to craft input that will spill over the *password* buffer and *tamper with hashedStoredPass*. The approach is as follows:

1. **Choose a target username:** First, the attacker picks an existing username.
2. **Prepare a fake password and its hash:** The attacker wants to fool the program into thinking the correct password was entered. To do this, they decide on a fake password – any string of their choice – and compute that string's hash. In our class, we used the example *"bad"* as the fake password (3 characters) and computed its hash. Let's call this hash H("bad").
3. **Construct the overflow string:** The input that the attacker will supply as the *password* needs to achieve two things: (a) overflow the *password* buffer into *hashedStoredPass*, and (b) plant the chosen hash value into *hashedStoredPass*. To accomplish (a), the input must exceed 7 characters (the capacity of *password* minus 1 for the null terminator). To accomplish (b), the content beyond the 7th character (which will overflow into the next buffer) should correspond to the bytes of H("bad"). There is one catch: when reading the password, functions like *scanf("%s")* or *gets* treat the input as a C-string, meaning they will stop reading at a newline and also end the string with a *'\0'* byte. That null terminator is crucial – it

will be written into memory and can terminate a string if it appears in the middle of it.

Figure 2: Exploit File

As shown in Figure 2, the exploit takes advantage of this by structuring the input as: [FakePassword] [NULL] [FakePasswordHash]. In our example, this would be the bytes representing *"bad",* followed by a *'\0'* byte, and then the 4-byte hash H("bad"). When this sequence is provided as a single string input, what happens in memory is: - The *password* buffer (8 bytes) receives "bad" followed by the null terminator, then the beginning of the hash, thereby overflowing it. Specifically, the first four bytes stored in *password* will be *{'b','a','d','\0'}***.** The remaining bytes (the hash) will not fit in *password* and thus will overflow into the subsequent region of the stack, which is where *hashedStoredPass* resides. - As a result, after the input read, the *password* buffer contains the string *"bad"* (with an implicit terminator), and the *hashedStoredPass* buffer has been partially or wholly overwritten with the bytes of the hash of "bad". Since *hashedStoredPass* initially held the real password hash for root (let's call that H(real)), it is now overridden to hold H("bad"). Essentially, *hashedStoredPass* now equals H("bad") – the attacker's chosen hash – instead of the original H(real).

4. **Trigger the comparison:** After reading the input, *LoginApp* proceeds to hash the provided password (which it interprets as *"bad"* because it read up to the '\0'). It stores that result in *hashedInputPass*. So now *hashedInputPass* contains H("bad") as well. Finally, the program compares *hashedInputPass* to *hashedStoredPass*. Thanks to the overflow, both buffers now contain the identical hash value H("bad"). The comparison returns that they are equal, and the program mistakenly believes the correct password was entered, thereby granting access.

Memory Layout Explanation

It may be helpful to break down the memory changes in stages, referencing the earlier described buffer layout on the stack:

- **Initial state:** When the program starts and allocates the buffers, memory for *username*, *password*, *hashedStoredPass*, and *hashedInputPass* is reserved on the stack. Initially, these buffers contain indeterminate data (whatever values were on the stack, or zeros if the compiler zero-initialized them, though typically local char arrays are not zeroed). We depict this initial state as Figure 3, showing the four buffers in order.

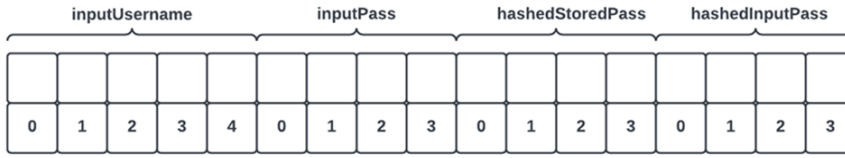

Figure 3: Memory layout after creating variables

- **After username input:** Suppose the user (attacker) inputs **"root"** as the username. The *username* buffer now holds *"root\0"* in memory. The program finds "root" in the user list and then copies her stored password hash (H(real)) into *hashedStoredPass* using *strcpy*. At this point, *hashedStoredPass* contains H(real) followed by its own **'\0'** terminator at the end of that string. The *password* and *hashedInputPass* buffers are still empty/unused at this moment. Figure 4 illustrates the stack after storing the hashed password, showing **username** filled and *hashedStoredPass* filled with H(real), while *password* and *hashedInputPass* remained untouched.)

Figure 4: Memory layout after hashing the entered username

- **After password input (overflow):** The user then inputs the crafted second line as described above. The first part, "bad," is placed in the password buffer, and then the '\0' from the input is written, effectively terminating the password at 3 characters. The subsequent bytes (the hash of "bad") overflow out of the bounds of *password*. These overflow bytes sequentially overwrite the memory where *hashedStoredPass* is

stored. By the time the input is fully read, the original H(real) in *hashedStoredPass* has been completely replaced with H("bad"). Figure 5 illustrates this overflow, showing the region corresponding to *hashedStoredPass* being overwritten with the new values.

Figure 5: Memory layout after reading the manipulated password

- **After hashing the input:** Now the program calls the hash function on the *password* buffer. But the *password* buffer currently contains "*bad*" (since reading stopped at the null). The hash function computes H("bad") and stores the result in *hashedInputPass*. So *hashedInputPass* now also contains (H("bad")). Figure 6 shows both *hashedStoredPass* and *hashedInputPass* holding identical values at this point.

Figure 6: Memory layout after hashing the input password

- **Comparison:** Finally, *strcmp(hashedInputPass, hashedStoredPass)* is called and returns 0 (meaning the two strings are equal). Therefore, the program prints “Access granted” and proceeds as if the correct credentials were provided.

Throughout this exploit, it’s important to note that we did *not* crash the program or overwrite a return address. The overflow was carefully sized to only corrupt a specific piece of data and nothing further. This made the exploit more stable and easier to achieve under classroom conditions – students didn’t have to worry about bypassing advanced protections like DEP or ASLR or other modern OS defenses that come into play for code injection.

Vulnerable Functions and Safer Alternatives

It's worth highlighting the specific C library functions at fault, as this ties back to secure coding lessons. We deliberately used *strcpy* to copy the hash and an unbounded scanf/gets to read the password to emulate common mistakes. These functions are part of a notorious group of C functions that do not perform automatic buffer-boundary checking. Classic examples include *gets(), strcpy(), strcat(), sprintf(),* and their variants. All of these can write past the end of an array if the input or data to copy is larger than the destination (Kak, 2020). Secure coding standards (such as CERT) strongly advise against using these in favor of safer alternatives (e.g., fgets, strncpy, snprintf, etc.) or, at least, manually checking lengths. In our program, had we used *fgets(password, sizeof(password), stdin)* instead of *gets/scanf,* the overflow would not occur because the input would be truncated to 7 characters (leaving space for the terminator). Likewise, using *strncpy(hashedStoredPass, dbHash, sizeof(hashedStoredPass) - 1***)** and then explicitly adding a *'\0'* would prevent overflow when copying the stored hash. These are exactly the kinds of improvements we wanted students to consider for learning objective (4). After walking through the exploit, we explicitly discussed how to fix the code to eliminate the vulnerability – effectively turning the exploit into a lesson on defensive programming. Students readily identified that functions like *strcpy* and *gets* were risky; some suggested using *fgets* or *scanf* with width specifiers, others mentioned using *memcpy* with known lengths or simply using higher-level languages for such tasks. This discussion reinforced the practical importance of choosing safe functions and validating inputs.

Comparison to More Severe Exploits

One might point out that our example does not demonstrate the "most dangerous" aspect of buffer overflows, namely, arbitrary code execution (e.g., injecting shellcode or altering the return address to hijack control flow). This is true – our *LoginApp* exploit is intentionally a gentler introduction that demonstrates logical privilege escalation rather than a system takeover. We scoped it this way for our target audience and time frame. Executing code or spawning a shell via overflow typically requires more setup (controlling exact memory addresses, dealing with non-executable stack regions, etc.), which would be difficult for our students at their current level. Instead, we opted for an attack that manipulates program state in a meaningful way (bypassing a security check) without

digging into machine code or requiring debugger-intensive techniques. This approach still achieves the educational goal: students see that a buffer overflow can subvert security – in this case, an unauthorized user logs in – which is already an eye-opening outcome for them. That said, we do plan to extend the difficulty in future modules (as noted later in the Conclusion). After students digest this exercise, one could present a follow-up exercise that uses a buffer overflow to overwrite a return pointer and execute a payload, thus exposing them to the concepts of shellcode injection and return-oriented programming in a controlled manner. In fact, there are educational tools that visualize exactly that kind of attack (e.g., some of the advanced stack smashing visualizations mentioned in our related work). Our focus here, however, was foundational: to ensure students fundamentally understand what a buffer overflow is and why unchecked memory writes are dangerous, before tackling more complex exploits.

5. Classroom Results and Student Feedback

The *LoginApp* challenge proved to be very challenging for the students. Out of 47 students (working in roughly 20 small groups), only one student successfully crafted a working exploit during the class session. This student had attended a separate Dynamic Application Security Testing (DAST) workshop a few weeks prior, where they learned about common vulnerabilities and tools. In other words, the only in-class success came from a student with some extracurricular training in finding and exploiting bugs. The other 46 students, despite some getting partially on the right track, did not manage to achieve the login bypass without significant hints or assistance.

During the debrief, we analyzed this outcome with the class. First, we congratulated the single solver and asked them to briefly describe their approach. Interestingly, that student used a debugger (gdb) to inspect memory and noticed the layout of variables, which led them to the idea of overflowing into the hash buffer. They then wrote a small Python script to generate the input file with the embedded null and hash, tested it, and found that it worked. This process was far beyond what most others attempted – it combined skills in debugging, understanding memory layout, and using external tools (like scripting or hex editing), which were not explicitly taught in this course.

For the majority of students, the thought process during the

exercise unfolded as follows (gleaned from their feedback and our observations): Many recognized that the presence of `strcpy` and the fixed buffer sizes suggested a buffer overflow. Some even hypothesized, "Maybe *we can overflow the password input."* However, they were unsure what the overflow would achieve or where to target it. A common misconception was: "Can we overflow the password buffer to overwrite the return address and skip the password check?" – a reasonable guess given what they know of classic exploits, but implementing it is complex and not feasible within the time or their skill level. They did not initially realize that a simpler target was at hand: the stored hash variable. The notion of inserting a null byte to truncate a string or to carefully overlay one buffer onto another was new to them. When we explained the actual solution step-by-step after the activity, we saw many "aha!" moments and also some frustration – *"I knew it was overflowing something, but I didn't think of that!"* was a common sentiment.

From a pedagogical perspective, the low success rate was not entirely surprising. This was the first time these students attempted to develop an exploit from scratch. In hindsight, expecting them to independently discover the exact technique in a short time was optimistic. The exercise as given was perhaps too open-ended for most. Several students commented, *"It was hard to even start – we've never done anything like this."* This suggests that additional scaffolding is needed if we want more students to arrive at the solution themselves. We discuss improvements below, but it's clear that without prior training in exploit development, most students struggled to make the imaginative leap from recognizing a bug to weaponizing it.

Despite the difficulty, informal feedback from students was overwhelmingly positive about the experience. In the surveys and class discussion, nearly all students said that seeing a buffer overflow in action left a strong impression. Many remarked along the lines of, *"I understood what a buffer overflow was before, but only now do I really appreciate how it can be used to break security."* The exercise succeeded in its primary goal: it raised awareness. Students were genuinely surprised that such a small program could be exploited so profoundly. One student wrote, *"Watching the login get bypassed felt like magic – scary magic. I won't forget why we need to check array bounds."* Another noted, *"It finally clicked for me why our professors kept telling us not to use `gets()`."* These comments indicate that even those who did not solve it gained a deeper understanding once

the solution was revealed and explained.

In terms of the learning objectives we set, we observed the following outcomes:

- **Objectives (1) and (2)** were clearly achieved by most students through the combination of the lecture, toy example, and the attempt itself. They could define what a buffer overflow is and identify where it occurred in the code. Indeed, many pointed out (when asked) that the culprit was reading an overly long input into a too-small buffer (they zeroed in on the *scanf* on *password* or the *strcpy* usage). Thus, they learned to spot the dangerous pattern.
- **Objective (3)** (analyzing the vulnerable program's workflow) was *partially* achieved. Students did inspect the structure of *LoginApp*; in fact, understanding the normal workflow was necessary to even consider how to break it. We observed that while only one student fully exploited it, several groups correctly outlined the program's logic and pinpointed the variable relationships (e.g., recognizing that the password is hashed and compared to the stored hash). This shows they engaged analytically with the code and understood the intended behavior, even if they couldn't fully subvert it.
- **Objective (4)** (creating secure alternatives) was addressed after we demonstrated the exploit. We prompted the students, *"So how could the developer have prevented this attack?"* Immediately, multiple hands went up with suggestions: *"Use fgets instead of gets," "Make the password buffer bigger or check the input length," "Don't copy the hash into a buffer at all—just compare the strings directly,"* etc. They even discussed higher-level fixes, such as *"maybe double-check the input length before hashing or use safer languages."* This discussion indicated that students were thinking about defensive coding. We followed up by showing a revised version of *LoginApp* with proper fixes (indeed, using *fgets* and *strncmp* with explicit length checks prevented the overflow). Although we did this code fix as a walkthrough rather than as a separate assignment, it reinforced objective (4). In future offerings, we might actually have students implement the fixes themselves as a short follow-up exercise to cement this knowledge.

One notable aspect of the feedback was that engagement was high throughout. Even though many were frustrated by not solving it, they reported enjoying the challenge. It was a novel experience compared to typical programming assignments. The competitive element (knowing at least one person solved it) also sparked curiosity – several students stayed after class to ask the "winner" how they did it, and to try the exploit themselves once they knew the trick. This kind of active interest is exactly what we hoped to achieve: students caring about a low-level detail like buffer management because they saw its real impact on security.

From the instructor's viewpoint, the exercise was a success in terms of student motivation and in driving home the lesson, but it also highlighted areas to improve. The very low success rate is a double-edged sword. On one hand, even failure was instructive and perhaps humbling (several commented that they *"realized security is hard"*). On the other hand, if students never taste success in a hands-on activity, it could discourage them or make them feel such topics are beyond their capability. We want to adjust the difficulty so that more students can achieve at least partial success with some guidance.

6. Discussion

Pedagogical Implications

The fact that only 1 of 47 students solved the challenge unassisted highlights a key pedagogical insight: open-ended security tasks require careful scaffolding. While the exercise succeeded in raising awareness, most students lacked the foundational skills to independently bridge theory to exploit. To address this, we will restructure the activity into incremental steps, starting with a guided buffer overflow in a harmless toy program, progressing to memory inspection with a debugger (e.g., gdb), and culminating in the full exploit. This gradual release of responsibility aligns with best practices for teaching complex concepts.

The sole successful student had prior exposure to security tools through a DAST workshop, underscoring the value of basic tool literacy. Introducing brief, ethical training in debuggers or simple fuzzers can empower more students to systematically explore vulnerabilities—not to train hackers, but to build confidence in diagnosing and reasoning about low-level code.

Finally, while experiential learning significantly boosts retention

(Resch, 2023), its placement matters. A preparatory demo or lightweight lab earlier in the course—before the full challenge—would better prime students for this type of adversarial thinking.

Limitations of the Study

Our evaluation relied on informal feedback and observation rather than formal pre/post assessments or longitudinal tracking of coding practices, so we cannot quantify learning gains. While student comments and engagement were positive, future work should employ controlled studies with clear research questions—such as comparing hands-on vs. lecture-only instruction or measuring how scaffolding affects success rates—and include comparison groups or iterative refinements.

Additionally, our findings may not generalize beyond cybersecurity-focused students. The tool's effectiveness likely varies with audience background: general CS students might find it too unfamiliar, while advanced security students might find it too simple. Context and audience calibration are essential for broader adoption.

Improvements and Future Directions

Based on the feedback and our observations, we plan several enhancements for the next iteration of this module:

- **Guided Lab Format:** Convert the challenge into a semi-guided lab exercise. For example, provide a worksheet or structured handout that first asks them to run the program with normal input and observe the output, then to experiment with a long input and note what happens (likely causing a crash or odd behavior), then perhaps hint at examining the memory layout (e.g., drawing out where each buffer might be), etc., culminating in the full exploit. This way, even if they don't arrive at the final answer entirely on their own, they will have walked through the critical thinking process step by step and seen the key phenomena.
- **Use of Debugging Tools:** Introduce a short tutorial on using a debugger (like gdb) or a memory visualization tool as part of this exercise. We could show them how to set a breakpoint right after the password read and inspect memory addresses or the stack. Seeing the two hash variables in memory and how one might overflow into the other could prompt the realization of making

them equal. Essentially, teach a mini-skills lesson within the exercise.

- **Alternate Scenarios:** Develop additional scenarios as follow-ups. For example, modify *LoginApp* to be vulnerable in a slightly different way (maybe a heap overflow or an off-by-one error) and ask students to exploit that. This would broaden their skill set and keep the activity fresh for those who might take more advanced courses. We already mentioned plans for eventually showing more dangerous cases like return address overwrites in a future module. The idea is to provide a progression of difficulty: *LoginApp* being level 1, and later modules introducing levels 2 and 3 of exploit complexity.

7. Conclusion and Future Work

LoginApp successfully bridges theory and practice by letting students experience a real buffer overflow exploit in a controlled setting. Despite a low success rate (1 of 47), the exercise deepened students' understanding of memory safety and motivated secure coding—confirming the value of hands-on learning (Alnajim et al., 2023).

Future iterations will add scaffoldings such as guided labs and introductory debugger training to improve accessibility and success rates. We also plan to formally assess learning gains through pre-/post-quizzes and code analysis.

Long-term, LoginApp will be the first in a series of progressively complex vulnerability modules (e.g., stack smashing, ROP), forming a curriculum-integrated "mini-CTF" suite. Our goal is to equip students not just with knowledge, but with the mindset to prevent vulnerabilities in real-world software.

References

Alnajim, A. M., Habib, S., Islam, M., AlRawashdeh, H. S., & Wasim, M. (2023). *Exploring cybersecurity education and training techniques: A comprehensive review of traditional, virtual reality, and augmented reality approaches*. Symmetry, 15(12), 2175. https://doi.org/10.3390/sym15122175

Kak, A. (2020). *Lecture 21: Buffer Overflow Attack*. In Lecture Notes on

"Computer and Network Security". Purdue University. *(Available online at engineering.purdue.edu/kak/compsec lecture notes.)*

Mirkovic, J., & Peterson, P. A. H. (2014). *Class Capture-the-Flag Exercises*. In Proceedings of the 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE '14). USENIX Association.

MITRE. (2020). *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*. MITRE CWE Database. Retrieved from https://cwe.mitre.org/data/definitions/120.html

Ramezanian, S., & Niemi, V. (2024). *Cybersecurity Education in Universities: A Comprehensive Guide to Curriculum Development*. IEEE Access, 12, 61741–61766. https://doi.org/10.1109/ACCESS.2024.3392970

Resch, C. (2023). *Giving Students a View of Buffer Overflow*. Engaging Learning Lab, University of Florida. *(Experience report and assignment materials, Fall 2023.)*

SANS Institute. (2006). Common Programming Errors and Vulnerabilities (White paper). *Retrieved from SANS.org (slide: "Big Three – 85% of vulnerabilities").*

Zhang, J., Yuan, X., Johnson, J., Xu, J., & Vanamala, M. (2020). *Developing and Assessing a Web-Based Interactive Visualization Tool to Teach Buffer Overflow Concepts*. In Proceedings of the 2020 IEEE Frontiers in Education Conference (FIE) (pp. 1–7). IEEE. https://doi.org/10.1109/FIE44824.2020.9274239

Zouahi, H., & Talhi, C. (2023). *Gamifying Cybersecurity Education: A CTF-based Approach to Engaging Students in Software Security Laboratories*. In Proceedings of the Canadian Engineering Education Association (CEEA 2023). (Paper presented at CEEA-ACEG 2023, June 2023, Montreal, QC).