Exploiting Buffer Overflows: A C Program for Software Security Education

Mohamed Elwakil
United States Coast Guard Academy

Buffer-handling errors remain a central topic in secure coding, yet students often encounter them only through brief examples or static explanations. This paper presents LoginApp, a small C program intentionally designed to demonstrate how unsafe string handling can corrupt adjacent program state and undermine authentication. The paper describes the artifact's design goals, code structure, vulnerability mechanism, and instructional rationale, along with a preliminary toy example illustrating unsafe input handling in C. It also discusses secure coding remediation, including bounded input handling, safer library choices, and alternative implementation patterns that eliminate the vulnerability. The contribution of this paper is the artifact itself and an instructor-facing framework for using it in memory-safety instruction; the paper does not report student data, learning outcomes, or classroom study results.

1. Introduction

Buffer overflow vulnerabilities remain among the most prevalent and dangerous software flaws. For example, 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 gap, LoginApp uses a deliberately vulnerable login program to show how unsafe string handling can corrupt adjacent data and alter an authentication decision. The example is intentionally small: students can inspect the whole program, trace the vulnerable path, and then compare it with safer alternatives. That combination makes the paper

useful for discussing memory safety, authentication logic, and defensive redesign within one compact code base.

The sections that follow explain the program's design, walk through the overflow mechanism it demonstrates, and describe how instructors can use the example in secure-coding instruction.

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.

Capture-the-Flag exercises provide strong hands-on engagement, but they often prioritize open-ended problem solving over close analysis of one small code path. LoginApp serves a different instructional purpose. It focuses on a single authentication bypass caused by data-only memory corruption, uses standard C in a familiar development environment, and keeps the program small enough for line-by-line inspection. That narrower scope makes it easier to move from vulnerability recognition to explanation and then to remediation.

The next sections situate LoginApp as a teaching example, explain how the overflow works, and show how the same program can support discussion of mitigation and redesign.

3. Design Goals for LoginApp

Learning Objectives

The artifact was designed around a set of instructional objectives that can be adapted to local course outcomes. When the artifact is used instructionally, learners 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.

Taken together, these objectives move learners from definition to diagnosis to redesign. The point is not only to identify a memory-safety failure, but to explain why it occurs and how different implementation choices would prevent it.

Intended Instructional Setting

LoginApp is intended for upper-division undergraduate computing contexts in which students have prior exposure to C programming, basic software engineering concepts, and introductory discussion of memory-safety issues. The artifact is especially suited to instructional settings that aim to connect secure-coding principles with concrete examples of unsafe string handling, adjacent-memory corruption, and redesign of flawed implementations.

Suggested Instructional Sequence

One workable classroom sequence moves from concept review to a short precursor example, then to guided analysis of LoginApp, and finally to redesign. An instructor might begin with a brief review of fixed-size buffers, unsafe string routines, and the reason such flaws still matter even when

students have already seen them in lecture. That opening also sets the frame for the module: the exercise is about understanding the security consequences of routine coding decisions, not about treating exploitation as an end in itself.

Before introducing LoginApp itself, instructors may find it useful to present a small preliminary example that isolates the core memory-safety issue in a form that is easy to inspect. As shown in Figure 1, a compact toy program with two adjacent static character arrays can serve this purpose well. An instructor can invite prediction of the output for a particular pair of inputs, then reveal that the actual output differs because one input has overflowed its destination buffer and altered neighboring memory.

```
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

This intermediate example helps make visible a behavior that is otherwise difficult to grasp from abstract explanation alone. It also creates a natural opening for discussion of the distinction between bounded and unbounded input operations in C, and for a short comparison of unsafe functions such as gets, strcpy, or unrestricted scanf("%s") with safer patterns such as fgets, width-limited scans, explicit length checks, and careful null termination.

After that groundwork, instructors can introduce LoginApp as a small authentication program whose failure mode can be studied in detail. A useful starting point is the program's intended behavior: how it reads a username, retrieves a stored credential value, accepts a password, computes a comparison value, and decides whether access should be granted. Starting with the normal workflow helps students identify what the code is trying to do before they analyze how unsafe input changes the result.

Once students understand the intended workflow, attention can shift to the vulnerable parts of the implementation. Instructors can direct them to the buffer declarations, the sequence of string operations, and the intermediate variables that store user input and comparison values. The crucial point is that the security failure comes from ordinary program state being corrupted, not from control-flow hijacking. That distinction makes the

example especially useful in courses where the goal is to connect memory errors to application logic without introducing a full code-injection unit. One way to organize the core activity is to ask learners to reason through several questions in sequence. What assumptions does the program make about input size? Which operations trust those assumptions without enforcing them? Which variables are security relevant? And if a write extends beyond the intended destination buffer, which neighboring values become vulnerable to corruption? Approached this way, the exercise becomes an analysis of software design fragility as much as a demonstration of low-level exploitation. Instructors may choose to keep this phase open-ended, or they may provide targeted prompts that narrow attention to specific lines of code, specific buffer sizes, or the consequences of particular input patterns. Either approach can support the same instructional objective, provided the emphasis remains on understanding how unchecked writes alter program state.

LoginApp is also practical because the full program remains small enough to inspect without losing sight of the larger argument. Students can read the source, relate buffer placement to program behavior, and then test how safer input handling changes the result. That compact scale makes the example reusable across lecture, lab, walkthrough, and redesign activities without requiring a larger software system to explain the same point.

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 4 bytes long in this design, meaning it can hold at most a 3-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. For illustration, consider the example 'bad' as the attacker-chosen input and let its corresponding derived value be denoted as 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 the 4-byte password buffer. To accomplish (b), the bytes beyond the first four positions, which 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.

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

Figure 2: Exploit File

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 (4 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.

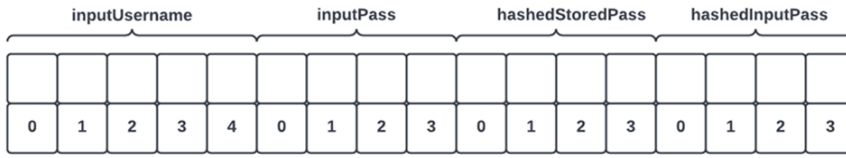

Figure 3: Memory layout after creating variables

- **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.

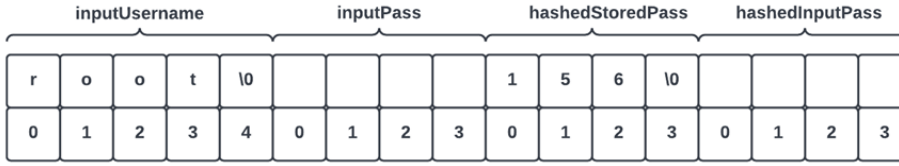

Figure 4: Memory layout after hashing the entered username

- **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.

This exploit does not crash the program or overwrite a return address. The input is sized to alter one adjacent value and no more, which keeps attention on data corruption rather than on DEP, ASLR, or other defenses tied to control-flow attacks.

Vulnerable Functions and Safer Alternatives

The vulnerable behavior depends on two unchecked operations: copying the stored hash with strcpy and reading the password with an unbounded call such as scanf("%s", password) or gets(password). Those routines do not enforce destination size, so they can write past the end of a fixed buffer when the input is too long (Kak, 2020). That choice is pedagogically useful because students can see exactly how an unsafe read and an unsafe copy create the conditions for the bypass. A redesigned

version of the same program can then illustrate bounded input, safer copy operations, explicit length checks, and less fragile handling of intermediate values. 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 4 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.

Comparison to More Severe Exploits

One might note that this example does not attempt to demonstrate arbitrary code execution, return-address overwrite, or other control-flow attacks often associated with classic stack-smashing demonstrations. That limitation is deliberate. LoginApp is designed as an introductory artifact centered on a data-only corruption scenario in which adjacent program state is modified without altering control flow. This narrower scope keeps the instructional emphasis on memory safety, unsafe assumptions about buffer boundaries, and the security significance of corrupting application data. More advanced topics such as shellcode injection, return-address overwrite, or return-oriented programming can be introduced later through separate artifacts if an instructor wishes to build a progression of increasing complexity.

5. Instructor Notes for Deployment

LoginApp works best as a guided code-analysis exercise rather than as a completely open-ended challenge. A useful sequence begins with a short example of unsafe string handling, moves into inspection of the program's buffer layout and control logic, and then explains how adjacent-memory corruption changes a security-relevant comparison. Framed this way, the activity rewards close reasoning about implementation choices instead of trial-and-error guessing. It also gives instructors room to pause at each stage and check whether students understand the intended behavior before introducing the failure mode. That pacing matters in courses where students can identify a buffer overflow in the abstract but still struggle to connect a local write past the end of a buffer to a later authentication decision.

It also helps to frame the module explicitly as an exercise in software assurance. Students should come away seeing that the vulnerability is produced by ordinary choices—a small fixed buffer, unchecked input, fragile copying of derived values, and a comparison that assumes nearby state is intact. That emphasis keeps the activity grounded in defensive programming even while it explains how the flaw can be abused. In practice, that framing can be established with a brief opening question: Which assumptions would have to remain true for this login routine to be trustworthy? Once the discussion starts there, the exploit no longer appears as a clever trick detached from engineering practice; it becomes evidence that a few routine implementation decisions can quietly undermine a security check.

Instructors can organize the activity in stages. First, students identify the intended authentication workflow. Next, they examine the assumptions built into the implementation: buffer sizes, string-copy operations, and trust in input length. Finally, they trace how one oversized write alters nearby data and changes the meaning of the final comparison. That progression keeps the example analytical rather than turning it into an isolated trick. It also creates natural checkpoints for discussion, short written reflections, or small-group work. For example, one stage might ask students to predict what values should be present in each buffer after ordinary execution, while a later stage asks how those expectations change once the password input exceeds the available space.

Visual support can make this example much easier to teach. Stack diagrams, annotated variable layouts, staged memory snapshots, or brief debugger views help students see a failure that otherwise remains abstract. That support matters because the program continues to run normally; the key change is corruption of a neighboring data buffer that later participates in a security check. In many classrooms, the decisive moment is not the exploit string itself but the point at which students can see the before-and-after state of the affected buffers. A single annotated memory snapshot may therefore be more useful than a longer verbal explanation, especially for learners who understand the source code but have difficulty imagining how the values are arranged in memory during execution.

Targeted prompts are also useful. Instructors can ask: What assumptions does this function make about input size? Which operations rely on those assumptions without enforcing them? Which variables become security-relevant later in the program? If a write passes the end of

one buffer, which neighboring values can it change? Questions like these steer the activity away from guessing an exploit string and toward systematic reasoning about implementation risk. They also help students practice the kind of review habits that transfer beyond this example. Instead of treating the module as a puzzle with one answer, instructors can use the prompts to model how a developer or reviewer would inspect any small authentication routine for fragile assumptions and unsafe handling of intermediate values.

The debrief should be a substantial part of the module rather than a short wrap-up. Reconstructing the exploit in order—the initial state, buffer placement, retrieval of the stored comparison value, oversized password input, overflow into adjacent memory, recomputation of the attacker-chosen value, and final comparison—helps students connect concepts they may otherwise hold separately, including null termination, bounded reads, stack adjacency, and authentication logic. A well-structured debrief also lets instructors revisit mistakes students made during prediction or tracing and turn those mistakes into part of the lesson. When learners can explain not only what happened but why their earlier assumption was wrong, the module does more than demonstrate a vulnerability; it strengthens their ability to reason about low-level behavior in later code.

A redesign phase is equally important. Once the mechanism is clear, instructors can return to the same code and ask how it should be rewritten so that oversized input is bounded, intermediate values are handled safely, and sensitive comparisons no longer depend on fragile local string buffers. That shift turns the example from a demonstration of failure into an exercise in software quality and secure implementation. It also prevents the lesson from ending at the most dramatic point of the exploit. Students should leave the module having seen at least one credible defensive rewrite and having discussed why that rewrite changes the security properties of the program. Without that closing step, the exercise risks being remembered as an interesting bypass rather than as a case study in preventable design weakness.

Depending on course goals, instructors may also add a small tooling component. A short introduction to a debugger, memory visualization, or controlled scripting can make the program easier to inspect without changing the basic purpose of the exercise. The aim is not to turn the module into an advanced offensive lab, but to make invisible program state easier to examine. Even a brief demonstration of where the relevant buffers

sit in memory can reduce confusion and save class time later, especially in settings where students have read code before but have little experience examining runtime state. Tooling is therefore best treated as support for explanation rather than as a separate learning objective.

The same program can fit several instructional contexts without becoming a classroom-outcomes paper. In a software engineering course, instructors might emphasize poor API choices and redesign of the authentication workflow. In a systems or secure-coding course, more attention may go to stack layout and the difference between data-only corruption and control-flow hijacking. In a broader cybersecurity sequence, LoginApp can serve as an introductory example that prepares students for more complex memory-corruption cases later on. These variations matter because they allow the same artifact to support different emphases without requiring the paper to overclaim what it contributes. The code does not need to stand for every form of exploitation to be useful; it needs to support careful analysis of one concrete failure mode in a way that instructors can adapt to their own sequence.

Taken together, these deployment choices show where LoginApp is most effective: as a reusable module that links memory safety, authentication logic, and defensive programming through one inspectable example. Its value lies in making a low-level flaw visible without requiring the full complexity of code-injection or return-oriented-programming demonstrations. That is a modest contribution, but it is a practical one. Instructors often need examples that are small enough to teach in a limited amount of time yet rich enough to support code reading, memory reasoning, and redesign. LoginApp is strongest when it is presented in exactly that role.

6. Design Considerations, Constraints, and Future Refinements

Instructional Design Considerations

Because students must connect low-level memory behavior with application logic, many courses will benefit from scaffolding. Smaller precursor examples, guided memory inspection, and a deliberate debrief can make the exploit easier to follow, especially for learners with limited experience in debuggers or binary-level reasoning. The need for scaffolding does not weaken the artifact; rather, it clarifies the conditions under which the example is likely to work well. A program this small can still ask students

to coordinate several forms of reasoning at once, including string handling, runtime state, and authentication logic. Making that coordination explicit helps instructors decide how much preparation to provide. In practical terms, that preparation might include a short review of fixed-size arrays, a reminder about null termination, or a quick walkthrough of how local variables are typically arranged during execution. Those additions do not change the paper's main argument, but they make the teaching sequence easier to implement in courses where students are still building confidence with low-level program behavior.

Basic tool literacy can also help. A brief introduction to debuggers, memory visualization, or controlled scripting support may make it easier for students to inspect the relevant state and reason about the overflow. The goal is still conceptual clarity about software defects, not offensive training for its own sake. In some contexts, the most effective use of tools may be instructor-led rather than student-driven: a live demonstration of stack state before and after the vulnerable read may provide enough visibility for the class to follow the mechanism without requiring a separate technical lab on tooling. Even a single guided inspection of buffer contents can reduce confusion, because students can compare the program's intended logic with the altered state that results from the oversized input. That kind of demonstration strengthens the link between source code, memory state, and security consequence without requiring a major change in course design.

Placement within the course matters as well. A short demonstration or lightweight lab earlier in the term may prepare students to engage more productively with this example when the full module appears (Resch, 2023). Conversely, if the artifact is introduced too early, students may spend most of their effort decoding surface syntax or debugger output rather than reasoning about the security consequences of unchecked writes. The paper therefore benefits from presenting LoginApp as one component in a broader teaching sequence rather than as a self-sufficient exercise that works equally well in every context.

Limitations of the Artifact

This artifact should be treated as a flexible teaching resource rather than as a one-size-fits-all module. Its usefulness will vary with course level, learner background, available tools, and the amount of time devoted to setup and debriefing. In classes with limited prior exposure to C, stack

memory, or debugger-based inspection, instructors may need additional preparation or simplification. In more advanced settings, the program may work best as a short introductory case before more complex memory-corruption examples. These constraints are not incidental. They affect how much of the exercise can be completed independently, how much instructor modeling is needed, and whether the main learning gain comes from exploit tracing, redesign, or a combination of both. They also shape how much explanation the paper itself must provide. A reader preparing to adopt the artifact needs enough detail to understand not only the bug, but also the classroom conditions under which the example remains manageable and instructionally useful.

Time is another practical constraint. A short session may support a code walkthrough and explanation of the overflow, but a longer format is better suited to memory inspection, exploit tracing, and redesign. Depending on course goals, instructors might spread the work across multiple meetings, pair it with a worksheet, or follow it with a separate refactoring exercise. Stating that limitation explicitly is useful because it reminds readers that the artifact is not only a piece of code but also a teaching sequence. The same program can look very different pedagogically when it is used in a 20-minute demonstration, a full lab, or a two-part assignment with follow-up redesign work.

Improvements and Future Directions

Future refinements can focus on instructor support rather than on changing the paper's core claim:

- Guided Lab Format: The exercise could be packaged as a semi-guided lab with a worksheet or structured handout. Students might first run the program with ordinary input, then try an oversized input, then map the buffer layout, and finally explain how the crafted input changes the comparison. That format would preserve the analytical progression while giving instructors a clearer way to support students who need more structure. It would also make the artifact easier to adopt in courses where instructors want students to work independently outside class but still need the activity to unfold in a controlled sequence.
- Use of Debugging Tools: A short debugger tutorial could be built into the module so that students pause execution immediately after the password read and inspect the relevant memory. Seeing the local buffers and hash values in place would make the mechanism more concrete and reduce the need for purely verbal explanation. This addition would be especially valuable in settings

where students already have basic command-line skills but have not yet used those skills to examine runtime state in a security context.

- Alternate Scenarios: Follow-up versions of LoginApp could introduce related flaws such as an off-by-one error or a different memory-corruption pattern. Those extensions would let instructors compare several failure modes while keeping the same emphasis on code reasoning, state corruption, and secure redesign. A family of closely related variants would also make it easier to show which lessons generalize across examples and which depend on the details of one particular buffer layout.

7. Conclusion and Future Work

LoginApp is a small teaching example that makes one memory-safety failure visible in a form students can inspect. Its main use is not to dramatize exploitation, but to connect unsafe input handling, adjacent-data corruption, and redesign within a single program. That focus makes the paper suitable for secure-coding instruction centered on explanation and prevention. The strongest claim the paper can make is therefore a bounded one: this artifact offers instructors a manageable way to discuss how a local implementation mistake can alter a security-critical decision without requiring a full unit on code injection or advanced control-flow attacks.
The most immediate extensions are practical ones: guided worksheets, debugger-oriented supplements, and redesign exercises that help instructors adapt the module to different course settings. Those additions would improve usability without changing the paper from an artifact description into a classroom outcomes study. They would also give future instructors more flexibility in deciding whether LoginApp should function as a lecture demonstration, a guided lab, or a short written code-analysis exercise. That kind of adaptability is valuable because the same example may need to serve different pedagogical purposes across software engineering, secure-coding, and introductory cybersecurity contexts.
If the authors later develop a broader sequence of related modules, that progression would be best presented as a separate curricular contribution. In the present paper, the stronger ending is that LoginApp offers a concrete, reusable example for teaching how routine implementation choices can undermine security-critical behavior. That narrower conclusion aligns better with the paper's actual evidence and leaves readers with a clear sense of what the artifact does well. It also keeps the conclusion focused on the strongest contribution already demonstrated in the manuscript: the ability

of one compact program to support explanation, memory reasoning, and secure redesign within a bounded instructional setting.

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. L. (2023). Giving Students a View of Buffer Overflow with Readily Available Tools. In Proceedings of the 2023 ASEE Annual Conference & Exposition. American Society for Engineering Education. asee.org

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).