Memory corruption vulnerabilities: Modern safeguards and their shortcomings

Charles Zumbaugh
Dept. of Computer Science
Kansas State University
Manhattan, Kansas
cazumbaugh@ksu.edu

***Abstract*—Memory corruption vulnerabilities are a class of software security vulnerabilities that occur when a system's memory is altered in an unintended way. In many cases, the result of memory corruption is a program crash or unintended behavior; however, many of these bugs can be exploited by malicious actors to gain unauthorized access to the system. Despite the advent of memory-safe languages and considerable effort identifying and fixing the sources of these vulnerabilities in memory-unsafe languages, this class of vulnerability remains one of the most common attack vectors in cyber-systems. Most modern computers include both hardware and operating system features to prevent an attacker from successfully exploiting these bugs, but nevertheless, attackers can bypass many of these protections. This paper discusses common memory corruption vulnerabilities, several modern safeguards to protect cyber-systems, and strategies that can be used to circumvent them.**

***Index Terms*—memory corruption, security, vulnerability, systems, malware**

I. Introduction

As computers have become more embedded in everyday life, the security of these systems has become a critical concern. Cyber attacks against government and non-government organizations has increased in recent years, particularly those targeting healthcare and large, multinational companies [1]. As personal data is increasingly being collected and stored on servers, the scale of these breaches is also increasing. For example, a recent breach of Change Healthcare in 2025 exposed the personal data of more than 192 million individuals in the United States [2]. The financial implications of security vulnerabilities are substantial, and IBM estimated the cost of a data breach in 2025 to be $4.4 million [3].

Despite being recognized as a critical vulnerability over 30 years ago, and numerous approaches being developed to mitigate them, memory corruption vulnerabilities are still found in all types of software [4]. Indeed, the 2024 list of the top 25 most dangerous software vulnerabilities included buffer overflows and use-after free in the second and eighth positions, respectively [5]. When exploited, these vulnerabilities can allow for arbitrary code execution, privilege escalation, and information leakage.

In practice, a program is considered memory safe as long as a set of memory errors can never occur [4], though rigorous, formal definitions have been presented in recent years [6]. Memory corruption is due to the lack of memory safety in low-level languages such as C and C++, and while memory and type safe languages such as Java and Rust provide memory safety, the use of memory-unsafe languages is ubiquitous in systems programming. Additionally, mitigation techniques are not always used, even among modern operating systems. These techniques may require non-existent hardware support on older systems, have large performance costs, or require recompilation of critical components [7].

II. Overview of memory corruption vulnerabilities

Memory corruption occurs when a system's memory is altered in an unintended way. These vulnerabilities are generally the result of subtle programming mistakes in low-level languages, where programmers must manually manage the memory throughout execution. Memory corruption will cause a crash or undefined behavior in many cases, but attackers can exploit memory errors to gain unauthorized access to the system by carefully modifying key regions of memory. While memory corruption vulnerabilities share some common characteristics as a class, their details differ and warrant discussion.

A. Buffer Overflow Vulnerabilities

Buffer overflows are one of the most common and dangerous memory corruption vulnerabilities. A buffer overflow can occur when a program writes data to a memory buffer without checking the bounds. If the data is larger than the buffer, the program will continue to write into adjacent regions of memory. These adjacent regions commonly contain important information, such as return addresses, that can be modified by an attacker to allow for arbitrary code execution. Figure 1 shows the general structure of a buffer overflow vulnerability and attack. While the precise details of the exploit depend on the the programs implementation details, the attack relies on the attacker gaining control of key data, such as the return address. In the simplest implementation of a buffer overflow exploit, the attacker overflows a buffer on the stack to overwrite the return address [8]. Thus, when the function returns execution will jump to the address given by the attacker providing arbitrary execution.

Buffer overflow vulnerabilities are commonly due to the inherent characteristics of low-level programming languages

such as C. Many data manipulation functions, such as *memcpy* do not check bounds. Additionally, data structures such as arrays are not first-class objects and are instead represented as pointers, making compile-time checks for overflow difficult [9]. Indeed, static analysis techniques were largely unsuccessful in detecting buffer overflow vulnerabilities in large software projects such as the Linux kernel [10].

B. Use After Free

In recent years, use after free exploits have been growing, especially against web browsers [11]. Use after free vulnerabilities result from dangling pointers. A dangling pointer is a pointer that points to an invalid region of memory, commonly created when the object referenced by the pointer is deallocated without updating the pointer. A use after free occurs when the program attempts to access this freed memory, which in many cases will result in a crash or undefined behavior. Like buffer overflows, dangling pointers can be exploited by attackers to gain control of a system [12]. However, these vulnerabilities can be more difficult to detect as they involve multiple events that may be separated across time and the codebase. In fact, detecting dangling pointer errors statically is undecidable and runtime detection strategies are computationally expensive [13].

C. Double Free

D. Units

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive".
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter", not "webers/m^2". Spell out units when they appear in text: ". . . a few henries", not ". . . a few H".
- Use a zero before decimal points: "0.25", not ".25". Use "cm^3", not "cc".)

E. Equations

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)", not "Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

F. LaTeX-Specific Advice

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BibTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BibTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

G. Some Common Mistakes

- The word "data" is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter "o".
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".

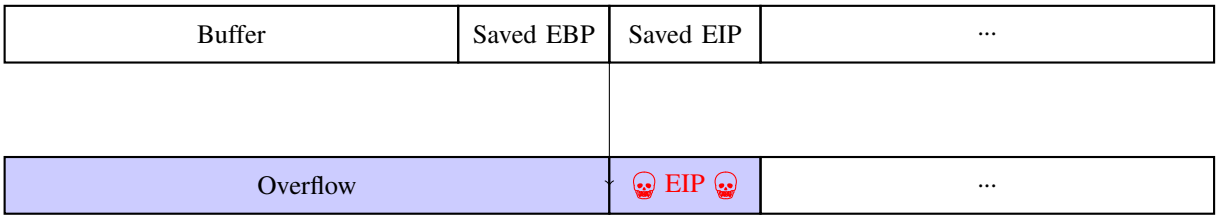

Fig. 1: Example of a basic stack buffer overflow attack. The attacker overflows the buffer, illustrated in blue, to overwrite the return address. When successful, this allows for arbitrary code execution.

- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is.

H. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

I. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

J. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 2", even at the beginning of a sentence.

TABLE I: Table Type Styles

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Fig. 2: Example of a figure caption.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

References

tnotes in the abstract or reference list. Use letters for table footnotes.

References

[1] H. Hammouchi, O. Cherqi, G. Mezzour, M. Ghogho, and M. El Koutbi, "Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time," *Procedia Computer Science*, vol. 151, pp. 1004–1009, 2019.

[2] P. R. Clearinghouse. (2025) Data breaches. [Online]. Available: https://privacyrights.org/data-breaches

[3] IBM. (2025) Cost of a data breach report 2025. [Online]. Available: https://www.ibm.com/reports/data-breach

[4] O. Llorente-Vazquez, I. Santos, I. Pastor-Lopez, and P. G. Bringas, "The neverending story: memory corruption 30 years later," in *Computational Intelligence in Security for Information Systems Conference*. Springer, 2021, pp. 136–145.

[5] CWE. (2024) 2024 cwe top 25 most dangerous software weaknesses. [Online]. Available: \url{https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html}

[6] A. Azevedo de Amorim, C. Hriţcu, and B. C. Pierce, "The meaning of memory safety," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 79–105.

[7] V. Van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2012, pp. 86–106.

[8] A. One. (1996) Smashing the stack for fun and profit. [Online]. Available: https://phrack.org/issues/49/14

[9] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: practice and experience*, vol. 33, no. 5, pp. 423–460, 2003.

[10] J. D. Pereira, N. Ivaki, and M. Vieira, "Characterizing buffer overflow vulnerabilities in large c/c++ projects," *IEEE Access*, vol. 9, pp. 142 879–142 892, 2021.

[11] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 133–143.

[12] J. Afek and A. Sharabani, "Dangling pointer: Smashing the pointer for fun and profit," 2007.

[13] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 269–280.