Factors That Determine Whether a Programming Language Is Secure

1. **Memory Safety**: According to Cifuentes and Bierman (2019), one of the top security vulnerabilities in programming is buffer errors, which arise due to improper memory management. Secure languages typically offer managed memory (e.g., garbage collection in Python) or ownership-based memory management (e.g., Rust), which eliminates many memory-related vulnerabilities.
2. **Input Validation**: Injection errors, including SQL injection and XSS, are another significant source of vulnerabilities. Cifuentes and Bierman emphasize that secure languages must provide first-class abstractions or built-in tools to validate and sanitize input to mitigate these risks.
3. **Type Safety**: Languages that enforce strict typing help reduce vulnerabilities caused by unintended type misuse. Chapter 2 of *Software Architecture with Python* highlights how PEP-8 promotes clarity and type safety, which indirectly supports secure coding by reducing developer errors.
4. **Error Handling and Information Leakage**: Secure languages handle errors in ways that avoid revealing sensitive information. As discussed by Cifuentes and Bierman, improper handling of runtime errors and logging sensitive data are major causes of information leaks.
5. **Strong Cryptographic Libraries**: Chapter 6 of Pillai (2017) underscores the importance of providing secure cryptographic tools to hash sensitive data and prevent vulnerabilities like weak password storage.

Could Python Be Classed as a Secure Language?

Python exhibits several characteristics of a secure language:

- **Managed Memory**: Python uses garbage collection to prevent memory-related vulnerabilities like buffer overflows (Cifuentes & Bierman, 2019).
- **Input Validation**: Python frameworks like Flask provide tools to sanitize inputs and protect against injection attacks, though this requires explicit developer action (Pillai, 2017, Chapter 6).
- **Ease of Use**: Python's readability and high-level abstractions reduce cognitive load for developers, helping avoid accidental vulnerabilities (Pillai, 2017, Chapter 2).

However, Python lacks built-in taint tracking or abstractions for preventing injection and information leakage by default (Cifuentes & Bierman, 2019). Therefore, while Python is relatively secure for high-level applications, its security depends on developers' adherence to best practices.

Python vs. C for Creating Operating Systems

While Python excels in simplicity and developer productivity, it is not suitable for operating system development for the following reasons:

1. **Performance**: Python is an interpreted language with significant performance overhead compared to C, which is closer to machine code and ideal for low-level system tasks (Cifuentes & Bierman, 2019).
2. **Memory Control**: Operating systems require precise control over memory allocation and hardware interaction, which Python abstracts away with managed memory, making it less suitable (Pillai, 2017, Chapter 6).
3. **Threading and Concurrency**: Python's Global Interpreter Lock (GIL) limits its ability to perform multithreaded operations efficiently, which are critical in operating systems (Cifuentes & Bierman, 2019).

C remains the dominant choice for OS development due to its low-level control and minimal abstraction layers, even though it poses risks like buffer overflows. These risks are mitigated through disciplined coding and tools like static analyzers (Cifuentes & Bierman, 2019). Python, while secure for application development, is unsuitable for OS-level programming due to its abstraction layers and performance limitations.

References

Pillai, A.B. (2017) Software architecture with python: Design and architect highly scalable, robust, clean, and high performance applications in Python. Birmingham, UK: Packt Publishing.

Cifuentes, C.G., & Bierman, G.M. (2019). What is a Secure Programming Language? Summit on Advances in Programming Languages.