Continue

Python code injection cheat sheet

Command injection is a critical security flaw that can expose Python applications to unauthorized command execution, potentially leading to data breaches, system compromise, and other malicious activities. This vulnerability arises when an attacker injects arbitrary system commands into a vulnerable program by passing unsafe user data. Several scenarios can lead to command injection vulnerabilities, including: * Passing unsanitized user input to system commands * Dynamically constructing command strings based on parameters, such as user input * Misusing or misconfiguring system commands The consequences of command injection are severe, including unauthorized access, exfiltration of sensitive data, and corruption of the system. This can result in financial losses, reputational damage, and legal repercussions for organizations. To ensure the safety of Python applications from command injection, it's essential to understand these vulnerabilities and take proactive measures to counteract potential exploits. A Python example was examined, showing how directly using user-controlled input to execute a command can be vulnerable. The code prompts users for a command and then executes it without validation or checking, making it susceptible to attacks. Attackers could use this to cause permission escalation, data loss, and other issues. The insecure use of system commands and the subprocess module in Python was highlighted. This allows for robust interaction with the operating system but can introduce vulnerabilities when combined with user-controlled inputs. A vulnerability example using subprocess with shell=True and unsanitized user input showed how this exposes the script to command injection attacks. Additionally, dynamic command construction without proper security measures was discussed. A snippet was shown where a command string is constructed incorporating user input directly, making it vulnerable to command injection. An attacker could exploit this by entering a payload that includes command separators or control operators, leading to malicious actions such as deleting files or accessing sensitive data. Lastly, the insecure use of Python's eval() function was noted. This powerful tool evaluates strings as Python expressions but can be a double-edged sword if used incorrectly. A vulnerability example using eval() showed how user input can be executed as a Python expression without validation, making it susceptible to attacks and potential code execution. eval() is risky because it evaluates any Python code, not just mathematical expressions. This allows attackers to execute malicious commands by importing modules like os. For example, an attacker could input:__import__('os').popen('ls').read() This could lead to unauthorized access, data theft, or command execution. To mitigate this risk, implement proper input validation and sanitization. Use parameterized queries and prepared statements to prevent concatenating commands and data as strings. When using subprocess module, avoid shell=True argument with user inputs. Try running the application without user input. If that's not possible, sanitize everything and use context-aware encoding techniques to limit the potential damage. Always run with least privileges necessary, as this minimizes the attack surface. Implement strong access controls and consider isolating your application using tools like chroot or containers. Secure coding conventions in Python include validating and sanitizing inputs to prevent command injection vulnerabilities. For example, ensure that input lengths are reasonable and only contain alphanumeric characters before processing. Use the subprocess module securely, avoiding shell=True unless necessary. Always handle user input and external commands safely by using argument lists instead of concatenating or interpolating. Use static application security testing (SAST) and software composition analysis (SCA) tools like Snyk to find and fix security issues early in development. This helps prevent exposure to malicious packages, supply chain attacks, and outdated dependencies with unpatched vulnerabilities. By incorporating these tools into your process, you'll not only build functional software but also ensure its overall security. In the rapidly changing cyber threat landscape, it's crucial to implement secure Python development practices across teams. This involves regular code reviews and security audits, catching potential vulnerabilities early on through input validation, authentication mechanisms, and sensitive data handling. Conducting periodic security audits is vital using tools like OpenVAS, ZAP, and Metasploit. Keeping software and libraries up-to-date is also essential as outdated versions often contain known security vulnerabilities. Utilizing security tools and frameworks designed for Python can automate vulnerability detection, such as Snyk's ability to identify code-level issues and vulnerabilities in third-party packages. Fostering a culture of security awareness among developers through training, workshops, and resources empowers them to prioritize security from the start. This leads to improved overall software quality and reduced risk of vulnerabilities. Using IDE extensions for real-time vulnerability detection can help catch vulnerabilities early on, saving time and effort that would otherwise be spent later in the development cycle. fixes. It's like having a security expert looking over your shoulder as you code, ensuring that your code is safe and resilient from the start. This cheat sheet contains patterns of potential code injection vulnerabilities and provides recommendations for developers to prevent these issues in their applications. By following these guidelines, you can ensure that your code is free from code injection vulnerabilities. To check your project's security, use Semgrep. The following command runs an optimized set of rules for your project: semgrep --config p/default 1. Executing or evaluating code 1.A. Executing code with exec() * The exec() function allows the dynamic execution of Python code. * Be cautious when using exec() with non-literal values, as it can lead to a code injection vulnerability if user-inputted content is executed. Example: ```` user_input = "");import requests;requests.get('localhost:3000');print('" exec("foobar('{}')".format(user_input)) ``` References Mitigation * Avoid using exec() with non-literal values. * Ensure that executed content is not controllable by external sources. If it's not possible, strip everything except alphanumeric characters from the input. Semgrep rule: python.lang.security.audit.exec-detected.exec-detected 1.B. Evaluating code with eval() * The eval() function also supports the dynamic execution of Python code. * Like exec(), it can lead to a code injection vulnerability if user-inputted content is executed. Example: ```` user_input = "__import__('code').InteractiveInterpreter().runsource('import requests;requests.get(\'localhost:3000\')')" eval(user_input) ``` References Mitigation * Avoid using eval(). * If you need to use eval() with non-literal values, ensure that executed content is not controllable by external sources. If it's not possible, strip everything except alphanumeric characters from the input. Semgrep rule: python.lang.security.audit.eval-detected.eval-detected data_to_send = """[loggers]keys=root[handlers]keys=hand01[formatters]keys=form01[logger_root]level=NOTSEThandlers=hand01[handler_hand01]class=StreamHandlerlevel=NOTSETformatter=form01args=(print("pwned"),)[formatter_form01]format=F1 %(asctime)s %(levelname)s %(message)sdatefmt=class=logging.Formatter"""s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)s.connect(('localhost', 9999))s.send(struct.pack('>L', len(data_to_send)))s.send(data_to_send)s.close() References: logging.config.listen() documentation Mitigation: Verify what is sent across the socket. Alternatively, to avoid the risk, verify the argument to logging.config.listen() to prevent applying unrecognized configurations. This can be done by encrypting or signing what is sent across the socket, such that the verify callable can perform signature verification or decryption. The code module provides read-eval-print loops in Python, which are dangerous if external data reaches these function calls as it allows a malicious actor to run arbitrary Python code. For example, using InteractiveInterpreter.runcode or InteractiveConsole.push, a malicious actor can execute arbitrary Python code. Given article text here print('pwned')