s of Terminal Emulation Anomalies and Shell Syntax Pathologies
stion

Convergence of Display and Logic Failures

rminal emulators, and shell command interpreters constitutes one of the most fragile yet fundamental aspects of Unix-like operating systems. This report provides an exhaustive fore
or near unexpected token"—accompanied by the display of binary garbage and the failure of command substitutions. The investigation reveals that this is not merely a syntactic issu
l Line Discipline, ANSI escape sequence injection, and the shell's lexical analysis mechanisms.

ession degrades into unintelligible characters followed by inexplicable syntax errors—is a classic manifestation of the "in-band signaling" problem inherent in the design of standard
ta to the terminal's standard output (stdout), the terminal emulator does not passively display the data. Instead, it attempts to interpret the raw bytes as control instructions. This inte
age") and, critically, causes the terminal to transmit data back to the host system via Device Attribute (DA) responses.

errors are a direct consequence of this feedback loop. The shell, interpreting the terminal's auto-generated response as user input, fails to parse the sequence, resulting in the repor
y of the Bash parser under fuzzing conditions, the mechanics of command substitution failures, and the precise recovery methodologies required to restore system integrity. Through
f solutions for preventing recurrence.

f Terminal Emulation and In-Band Signaling

st first deconstruct the architecture of the Unix terminal subsystem. The error manifested in the user's log—bash: command substitution: syntax error near unexpected token—is a sy
or, and the shell.

etypes to Pseudo-Terminals (PTYs)

xterm, GNOME Terminal, or Alacritty) is rooted in the physical hardware of the mid–20th century, specifically Teletype (TTY) machines. These devices communicated with mainframe
th the text to be printed and the control signals to manage the hardware (e.g., ring the bell, move the carriage return, feed a new line).1

rdware is virtualized through Pseudo-Terminals (PTYs). A PTY acts as a bidirectional pipe with specific semantics. It consists of two ends:

emulator application. It receives characters from the application to display on the screen and sends keystrokes from the user to the slave side.
sh, Zsh) or other command-line interface (CLI) programs. To the shell, the slave side appears indistinguishable from a physical serial port or console.2

f in-band signaling. The standard output (stdout) of a command is treated as a continuous stream of bytes. Text files strictly adhere to character encodings like ASCII or UTF–8, whic
spectrum of 256 possible byte values (00_{16} to FF_{16}). When these bytes are sent to the terminal—for example, via an inadvertent cat of a binary file or an improper command substi
ndering engine.1

line (N_TTY)

el's TTY driver, which implements a "Line Discipline" (ldisc), typically N_TTY. This software layer is responsible for the standard processing of input and output. It handles features tha

yped.
a newline is received, allowing for backspace editing.
rol characters (like Ctrl+C or 0x03) into POSIX signals (SIGINT) sent to the foreground process.

e distinction between "data" and "control" blurs. The line discipline may interpret random binary bytes as control characters. For instance, a byte 0x03 inside a binary file, if echoed b
ne passes the raw binary stream to the terminal emulator, which maintains a complex state machine driven by these bytes.2

es and the State Machine

at includes cursor position, text color, active character sets, and input modes. This state is manipulated via escape sequences, primarily defined by the ECMA–48 standard and lega
^{6} or 0x1B), often followed by a bracket `

A)

rs observed. The sequence `ESC

if the script is currently executing a command that reads from stdin, it receives this sequence.

he Escape character begins a sequence, but if not handled by the Readline library correctly (or if the shell is in a specific mode), the subsequent characters are treated as literals or distinct
parator in Bash. The c is a literal. The 2 is a literal number. Crucially, depending on the specific response string (e.g., some terminals return strings containing parentheses or other metacha

the visual corruption (due to charset switching via 0x0E) and the syntax error (due to the shell trying to parse the terminal's auto-reply generated by ESC [c).4

y of Bash Syntax Errors

sage: bash: command substitution: syntax error near unexpected token. This section analyzes the linguistic processing of the Bourne Again Shell (Bash) to explain why this specific e

okenization

nce of stages: quoting, tokenization (lexical analysis), expansion, and parsing. The parser expects a stream of tokens adhering to a strict grammar defined (historically) by Yacc/Biso

g the syntax $(command) or the legacy backticks `command`—Bash spawns a subshell or creates a new execution context to run the enclosed command. It captures the command
argument list.5

ine Counting

presence—or total lack—of newline characters (0x0A). Snippet 1 raises the question of why cating a binary file causes "high line numbers" in error reports.

d, awk, and grep operate on a line-by-line basis. A "line" is defined strictly as a sequence of bytes terminated by 0x0A. In a binary file, 0x0A occurs purely by chance, statistically rou
ts (like compressed archives, JPEGs, or executables), 0x0A might not appear for megabytes of data.

.g., source binary_file or $(cat binary_file)), the shell sees one incredibly long line. When a syntax error inevitably occurs (because the binary data is not valid shell code), the error me
s interpreted as "Line 1," the error report often confusingly refers to the start of the file or a very high character offset disguised as a line number issue.1 Conversely, if the binary happ
"line 2453" of a file that the user believes is small or irrelevant.

n Failure Modes

plies that the parser encountered a grammatical token (like (,), |, &, fi, or done) in a position where the shell grammar forbids it.

bstitution

hell metacharacters, the result of the expansion undergoes **Word Splitting** and **Filename Expansion** (globbing).6

he expansion is used in a context like:

binary content happens to form a sequence that looks like a subshell start (or a function definition, the parser state machine may become desynchronized. For example, if the binar
fter a function name, the parser will flag it.

" Injection

4, if the binary stream triggers a terminal response `^ and.8

eturns (DOS Line Endings)

cted token," particularly involving tokens like do, done, fi, or then, is the presence of Carriage Return characters (\r or 0x0D) from Windows/DOS-formatted files.9

n DOS/Windows, it is Carriage Return + Line Feed (CRLF, \r\n). When Bash reads a script with CRLF endings:

e if statement. It does not find it (because then\r is distinct from then).

ders the fi to be unexpected and out of place.

token ‘fi’ or ‘done’.10

arbage, \r is a common byte in binary files. If a binary file is sourced or substituted, these \r bytes will corrupt token recognition just as they do in DOS-formatted text scripts. This cr ncoding mismatch caused by the binary data ingestion.

Shell Syntax Error Vectors

ctors by which binary data induces syntax errors in Bash.

ble of Binary Data
ontains `ESC, we can reconstruct the exact sequence of events that leads to the user’s screenshot (implied) and logs. This reconstruction assumes a standard Linux environment using Bash mpatible terminal emulator (like xterm, Konsole, or iTerm2).

ry data to stdout. Common culprits include:

resource without the -o flag to save to disk).
g binary context, causing grep to output the matching binary line).

ual Corruption)

ne passes them to the emulator.

(Bell) cause the terminal to beep or flash. Bytes 0x00–0x1F cause erratic cursor movements (0x08 Backspace, 0x09 Tab), overwriting text on the screen.
(Shift Out). The terminal switches to the G1 character set (Graphics). Subsequent output (even the shell prompt itself) is rendered using this set. The letter ‘a’ might appear as a checkerboa
sees a screen full of gibberish, and even when the command stops, typing commands yields only more gibberish.

ogical Corruption)

e 0x1B 0x5B 0x63 (`ESC

and Substitution Vulnerabilities

d Substitution Failures”. This warrants a dedicated analysis of how $(…) handles binary data compared to plain execution.

d Variable Handling

re null-terminated. However, Bash is capable of holding binary data in variables to an extent, but the read builtin and command substitution mechanisms have limitations with the Nu

re designed to process text streams. When command substitution runs var=$(cat binary), the shell creates a pipe. The cat process writes to the write-end of the pipe. The shell reads

utomatically strips trailing newlines.6 If the binary file ends with 0x0A, that byte is removed. This corruption alters the binary integrity, making the variable content essentially different from th
bytes better than older shells, passing variables containing null bytes as arguments to other commands (execve) often fails because the kernel treats the argument string as terminated at th
byte in input. This warning itself is a form of error that clutters the logs.

g

anding file lists. Similarly, if
$ARG_MAX(systemlimitonargumentsize), theexecutionwillfailwith“Argumentlisttoolong”. Thisisrelevantifthe“garbage”errorisfollowedbyacrash. Binaryfilescar$
$linevariableecho$
on Linux is typically around 2MB, meaning any binary larger than this will cause a hard failure if passed as an argument.

Risks

g: DIRNAME=“$(dirname$”FILE“)”. If the binary data is assigned to a variable without quotes:

ery space (0x20), tab (0x09), and newline (0x0A) in the binary file becomes a delimiter. The binary file is shattered into thousands of arguments.

If the binary contains *, ?, or `

Recovery Strategies

data requires specific knowledge of termios and escape sequences. The report identifies three tiers of recovery, ranging from simple resets to advanced state manipulation.

et

typed characters appear as gibberish. The user cannot see what they are typing. This psychological barrier often prevents effective recovery. The command reset 2 is the standard fi

zation string rs1/rs2 from the terminfo database).
).
).

acters, they must type reset<Enter> blindly. If the prompt is currently containing garbage text, it is recommended to press Ctrl+C first to clear the current line.15

or terminal replies which might be blocked), stty sane is the reliable fallback.2 stty manipulates the kernel Line Discipline directly. The sane argument resets the discipline to reasona

nabling backspace).
ne on input).
upt, Ctrl+Z for suspend).

ut might not fix the visual rendering (character sets) of the terminal emulator itself. It ensures that Enter works and commands are accepted.3

e Sequence

hift-Out issue) without a full reset, one can force the terminal back to the standard charset.

nand line. Snippet 3 suggests echo <ctrl-v><esc>c<enter>, which sends the “Full Reset” (RIS) escape sequence to the terminal. This is a hard reset for the emulator state, equivalen
all defaults.

t Cause Prevention)

bstitution” failures in the future, scripts must be hardened against binary data:

esigned for binary inspection. od (octal dump), hexdump, xxd, or cat -v.9 cat -v escapes non-printing characters (e.g., displaying `^ This is the single most effective defense against acciden
at might be binary, filter it.

76')

le ASCII and standard whitespace.
ts or input files to remove carriage returns \r that cause parser failures.10 This eliminates the “syntax error near done” class of bugs.

Recovery Commands

	Target Layer	Effect	Pros	Cons
ization strings.	Terminal Emulator	Full Re-initialization	Most complete fix.	Slow; can hang if I/O is blocked.
	Kernel Line Discipline	Restores Input Processing	Instant; fixes Enter/Backspace key mapping.	Does not fix “garbage” charset (G1/G0) display.
	Terminal Charset	Restores ASCII	Fixes “garbage” text immediately.	Does not fix broken input modes or tabs.
string.	Terminal Emulator	Re-initialization	Faster than reset command.	Requires ncurses installed.

nalysis of Provided Research Material

nted but consistent picture of the problem space. By synthesizing these disjointed data points, we can reinforce our analysis with specific evidence.

rror near unexpected token |”. This confirms that pipes or special characters inside substitution $(…) are sensitive to parsing errors if not quoted or if the content contains binary noise that
”. This specific token is highly indicative of the terminal injection theory. The response ^[[?1;2c contains a semicolon and numbers, but depending on the exact terminal (e.g., some report ^
cted token ‘done’” to DOS line endings. This is a crucial “false positive” to rule out. If the user’s binary garbage includes 0x0D, the shell acts exactly as it does with DOS scripts.

Evidence

for root cause analysis. They explicitly describe the scenario: file -m (binary output) -> “Tons of binary gibberish” -> “Warning:… invalid file” -> ^ asks: *“how does catting a binary file lead t
erminal) sequence. The terminal replies, and the reply is executed. This is the definitive explanation for the “Bash Error” component of the user’s query.

n Mechanics

yntax errors. This shows that even *Process Substitution* <(…) is susceptible to syntax errors if the underlying command produces output that confuses the parser or if the file descriptor han
paths with spaces (e.g., Program Files (x86)). While this is a text error, it parallels the binary error: unquoted special characters ((in x86) break the parser. In binary files, these characters ap

mb” utility. It performs no analysis; it just moves bytes from source to destination. The “high line numbers” mentioned in queries like 6 or 18 are due to the absence of newlines in binary stre
ne 1”. This explains why error logs from binary ingestion are often cryptic regarding location.

ical Implications

Escape Injection Attack

ecurity vulnerability known as **Terminal Escape Injection**. If an attacker can trick a user into cat-ing a file (e.g., “Check this log file for errors”), they can embed escape sequences th

ound color, effectively masking the user’s subsequent commands.
echanism or programmable function keys (on supported terminals) to inject arbitrary commands.
equences can remap the Enter key to execute rm -rf / before sending the carriage return.

o escaping control characters when writing to a TTY, and why cat should be used with extreme caution on untrusted files. The cat -v flag acts as a sanitizer, stripping the executable

d Signaling

. The data channel (text to display) and the control channel (commands to the terminal) share the same stream (stdout). This dates back to teleprinters where distinct cables for cont
st system would separate display data from control metadata, preventing binary data from ever being interpreted as control instructions. However, the ubiquity of ANSI/VT100 compa

mbiguities

. When a terminal set to UTF–8 receives invalid byte sequences (e.g., 0xFF, 0xC0), it uses replacement characters () or drops bytes entirely. This unpredictable filtering can alter the e
ministic. A file might cause an error on xterm but not on iTerm2 due to different error-handling logic for invalid UTF–8 sequences. This variability makes debugging binary-induced sh

d substitution failures” reported by the user are a triad of symptoms resulting from a single root cause: **the unsafe transmission of raw binary data to a terminal emulator.**

erpreting random bytes as control sequences, specifically Shift-Out (0x0E) which corrupts the character set mapping.
interpreting random bytes as a “Send Device Attributes” query (ESC [c) and injecting the response (^[[?1;2c) into the shell’s standard input. The shell blindly attempts to parse this injectec

ed by the presence of null bytes, lack of newlines (causing buffer/argument limits), and the expansion of binary metacharacters into file globs.

tore the terminal state.
terminal. Use cat -v to inspect, base64 to transfer, or od/hexdump to analyze.
d sanitize inputs before processing to prevent the shell from choking on injected garbage.

nder of the Unix philosophy’s double-edged sword: the power of universal text streams comes with the peril of interpreting everything as a stream, even when it isn’t. The “everything
raction itself.

ness up the terminal?, accessed January 26, 2026, https://unix.stackexchange.com/questions/119480/why-and-how-did-using-cat-on-binary-files-mess-up-the-terminal
accessed January 26, 2026, http://www.lostpenguin.net/index.php/stty-tty-clear-reset/
een dumped inside it? [duplicate], accessed January 26, 2026, https://unix.stackexchange.com/questions/50752/how-to-fix-a-terminal-after-a-binary-file-has-been-dumped-inside-it
ing sent to stdin of the terminal I’m using?, accessed January 26, 2026, https://www.reddit.com/r/commandline/comments/1et3ed/how_does_catting_binary_data_lead_to_text_being/
ar unexpected token |’ - Stack Overflow, accessed January 26, 2026, https://stackoverflow.com/questions/43667509/bash-command-substitution-syntax-error-near-unexpected-token
ash - Unix & Linux Stack Exchange, accessed January 26, 2026, https://unix.stackexchange.com/questions/118433/quoting-within-command-substitution-in-bash
near unexpected token `(’ - Stack Overflow, accessed January 26, 2026, https://stackoverflow.com/questions/68035754/command-substitution-line–72-syntax-error-near-unexpected-toker
” error with process substitution, accessed January 26, 2026, https://unix.stackexchange.com/questions/669655/bash-syntax-error-near-unexpected-token-error-with-process-substitutior
e endings? - Stack Overflow, accessed January 26, 2026, https://stackoverflow.com/questions/39527571/are-shell-scripts-sensitive-to-encoding-and-line-endings
while read line - Unix & Linux Stack Exchange, accessed January 26, 2026, https://unix.stackexchange.com/questions/616504/syntax-error-near-unexpected-token-done-while-read-line
- bash - Stack Overflow, accessed January 26, 2026, https://stackoverflow.com/questions/20895946/syntax-error-near-unexpected-token-bash
h - Stack Overflow, accessed January 26, 2026, https://stackoverflow.com/questions/20586785/syntax-error-near-unexpected-token-fi
s) - Ask Ubuntu, accessed January 26, 2026, https://askubuntu.com/questions/1028197/file-list-command-line-hidden-and-subfolders
u, accessed January 26, 2026, https://askubuntu.com/questions/25077/how-to-really-clear-the-terminal
ix & Linux Stack Exchange, accessed January 26, 2026, https://unix.stackexchange.com/questions/79684/fix-terminal-after-displaying-a-binary-file
essed January 26, 2026, https://gist.github.com/JAAdrian/baae374cd24be58d4baa620d8dd6e473
near unexpected token `x86’” · Issue #1789 · JanDeDobbeleer/oh-my-posh - GitHub, accessed January 26, 2026, https://github.com/JanDeDobbeleer/oh-my-posh/issues/1789
ux Stack Exchange, accessed January 26, 2026, https://unix.stackexchange.com/questions/47407/cat-line-x-to-line-y-on-a-huge-file