Trust Boundary Security Analysis Report: telnetd

Prepared by Brian Williams (brian@vartia.ai)

Executive Summary

We formally modeled the trust boundary between network-facing daemons (such as telnetd) and local authentication services (such as /usr/bin/login) using TLA+ and the TLC model checker. The model found that **without input sanitization, attacker-controlled data can trigger privileged authentication bypass.** With allowlist sanitization, the model verified no such path exists.

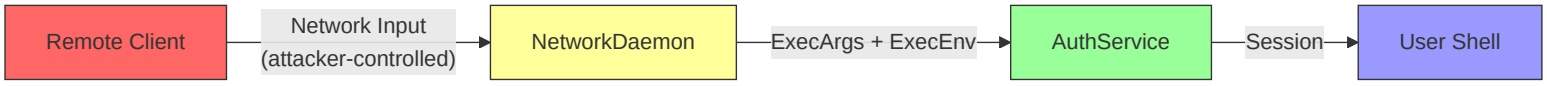
What Was Done

Scope

We created an abstract formal model of the data flow between:

- **NetworkDaemon**: A service that accepts untrusted input from remote clients
- **AuthService**: A local service that authenticates users and creates sessions

The analysis is informed by the behavior of telnetd as implemented in GNU Inetutils version 1.9.3.



The Model

Every piece of data is tagged with its **provenance** (origin):

Provenance	Trust Level	Example
`Network`	Untrusted	USER env var from telnet client
`Config`	Trusted	Values from /etc configuration
`Hardcoded`	Trusted	Compiled-in defaults

AuthService has two authentication modes:

- **NormalAuth**: Validates credentials (safe)
- **TrustedAuth**: Skips validation for pre-authenticated callers (dangerous if triggered by attacker)

What We Checked

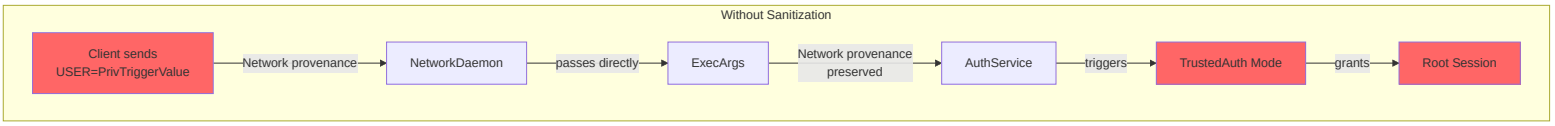
Three security invariants expressing general principles:

Invariant	Plain English
AuthenticationIntegrity	Skipping authentication requires trusted input
SessionIntegrity	Privileged sessions require proper authentication
ProvenanceIntegrity	Untrusted input cannot trigger privileged modes

---

What Was Found

The Vulnerability Pattern



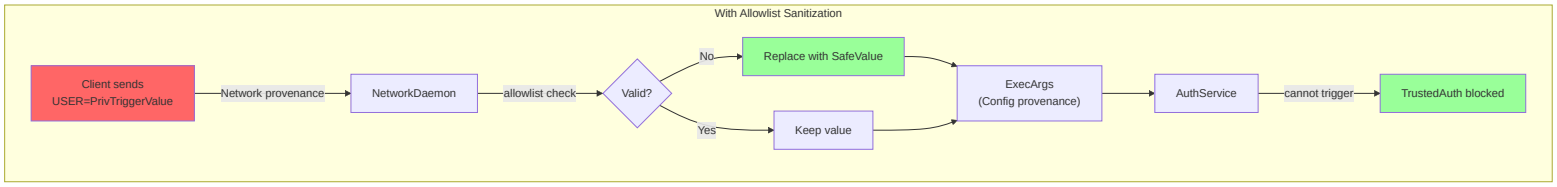
TLC found a counterexample in 4 states:

```
State 1: remoteEnvUser = PrivTriggerValue, provenance = Network
State 2: NetworkDaemon processes input
State 3: ExecArgs constructed, provenance still = Network
State 4: TrustedAuth triggered ← VIOLATION
```

The invariant `AuthenticationIntegrity` is violated because:

```
authModeUsed = "TrustedAuth" AND execArgsProvenance = Network
```

The Fix



TLC exhaustively checked 12 states with no violations.

How We Know This Is True

Formal Verification Guarantees

TLC performs **exhaustive state-space exploration**. Within the model's bounds:

Claim	Basis
Vulnerable config has a flaw	TLC found an explicit counterexample trace
Secure config has no flaw	TLC checked ALL 12 reachable states, none violate invariants

This is not testing or fuzzing—it's mathematical proof that within the model, no sequence of transitions leads to an invariant violation.

Model Abstraction

The model uses **abstract values**:

- `PrivTriggerValue` = "some input that triggers TrustedAuth" (we don't specify what)
- `SafeValue` = "some input that doesn't trigger TrustedAuth"

This means the proof applies to **any** concrete exploit pattern, not just known ones.

Concrete Examples

The Vulnerable Code Path

From `telnetd.c:62`, the login command template:

```
char *login_invocation = PATH_LOGIN " -p -h %h %?u{-f %u}{%U}";
```

From `utility.c:1700`, the `%U` expansion:

```
case 'U':
    return getenv("USER") ? xstrdup(getenv("USER")) : xstrdup("");
```

From `pty.c:96-109`, the sanitization (insufficient):

```
static void scrub_env(void) {
    for (cpp2 = cpp = environ; *cpp; cpp++) {
        // Only removes: LD_*, _RLD_*, LIBPATH=, IFS=
        // Does NOT sanitize USER
        if (strncmp(*cpp, "LD_", 3) &&
            strncmp(*cpp, "_RLD_", 5) &&
            strncmp(*cpp, "LIBPATH=", 8) &&
            strncmp(*cpp, "IFS=", 4))
            *cpp2++ = *cpp;
    }
}
```

Example Attack

Step 1: Attacker connects and sets USER environment variable

```
$ telnet target.example.com
Trying 192.168.1.100...
Connected to target.example.com.

# During TELNET NEW-ENVIRON negotiation, client sends:
#   VAR USER VALUE -froot
```

Step 2: telnetd constructs login command

The template `%?u{-f %u}{%U}` means:

- If authenticated user (`%u`) exists → use `-f %u`
- Otherwise → use `%U` (the USER env var)

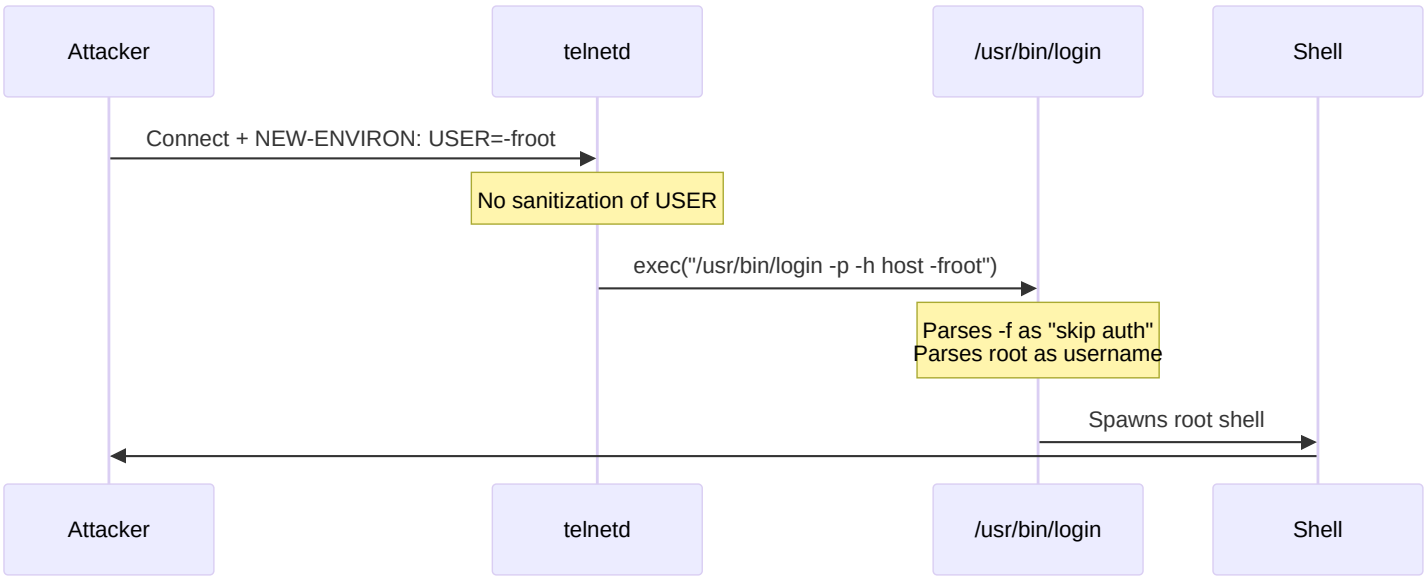
Without authentication, the command becomes:

```
/usr/bin/login -p -h client.example.com -froot
```

Step 3: login interprets `-froot` as a flag

```
-f root  →  "skip authentication for user root"
```

Result: **Root shell without password**



What The Model Captures

In TLA+ terms:

Real System	Model Abstraction
`USER=-froot`	`PrivTriggerValue`
`USER=alice`	`SafeValue`
USER env var from client	`remoteEnvProvenance = Network`
`-f` triggers trusted login	`CouldTriggerPrivilegedMode(PrivTriggerValue) = TRUE`
No scrub_env for USER	`SanitizationPolicy = PolicyNone`

The model's counterexample:

```
State 3: execArgsUser = PrivTriggerValue, execArgsProvenance = Network
State 4: authModeUsed = "TrustedAuth"  ← VIOLATION
```

Maps directly to:

```
login receives "-froot" from network input → triggers -f flag → auth bypass
```

Example Fix

Option 1: Strip dangerous characters (Blocklist)

```
// In telnetd before constructing login command:
char *sanitize_username(const char *user) {
    if (user == NULL) return NULL;

    // Reject if starts with dash (prevents flag injection)
    if (user[0] == '-') {
        syslog(LOG_WARNING, "Rejected USER starting with dash: %s", user);
        return NULL;
    }

    // Reject control characters and shell metacharacters
    for (const char *p = user; *p; p++) {
        if (*p < 32 || *p > 126 || strchr(";|&$`\\\"'", *p)) {
            syslog(LOG_WARNING, "Rejected USER with bad char: %s", user);
            return NULL;
        }
    }

    return xstrdup(user);
}
```

Option 2: Allowlist only valid usernames (Allowlist) - RECOMMENDED

```
#include <regex.h>

char *sanitize_username(const char *user) {
    if (user == NULL) return NULL;

    // Only allow: lowercase letters, digits, underscore, dash (not at start)
    // Max 32 chars (reasonable username limit)
    regex_t regex;
    regcomp(&regex, "^[a-z_][a-z0-9_-]{0,31}$", REG_EXTENDED | REG_NOSUB);

    if (regexec(&regex, user, 0, NULL, 0) != 0) {
        syslog(LOG_WARNING, "Rejected invalid USER: %s", user);
        regfree(&regex);
        return NULL;
    }

    regfree(&regex);
    return xstrdup(user);
}
```

Option 3: Never use network input in login args (Drop Untrusted)

```
// Simply don't pass USER to login at all
// Let login prompt for username itself
char *login_invocation = PATH_LOGIN " -p -h %h";
// Remove %U entirely - login will ask for username interactively
```

Brian Williams · Vartia.ai · brian@vartia.ai

Example Regression Test

```c
// test_sanitize_username.c

void test_rejects_flag_injection() {
    assert(sanitize_username("-froot") == NULL);
    assert(sanitize_username("-f root") == NULL);
    assert(sanitize_username("--help") == NULL);
    assert(sanitize_username("-") == NULL);
}

void test_rejects_shell_injection() {
    assert(sanitize_username("alice;id") == NULL);
    assert(sanitize_username("alice|cat /etc/passwd") == NULL);
    assert(sanitize_username("$(whoami)") == NULL);
    assert(sanitize_username("`id`") == NULL);
}

void test_rejects_special_chars() {
    assert(sanitize_username("alice bob") == NULL);  // space
    assert(sanitize_username("alice\nroot") == NULL); // newline
    assert(sanitize_username("alice\x00root") == NULL); // null
}

void test_accepts_valid_usernames() {
    assert(strcmp(sanitize_username("alice"), "alice") == 0);
    assert(strcmp(sanitize_username("bob123"), "bob123") == 0);
    assert(strcmp(sanitize_username("_system"), "_system") == 0);
    assert(strcmp(sanitize_username("web-app"), "web-app") == 0);
}

void test_length_limits() {
    char long_name[100];
    memset(long_name, 'a', 99);
    long_name[99] = '\0';
    assert(sanitize_username(long_name) == NULL);  // too long
}
```

Mapping Fix to Model

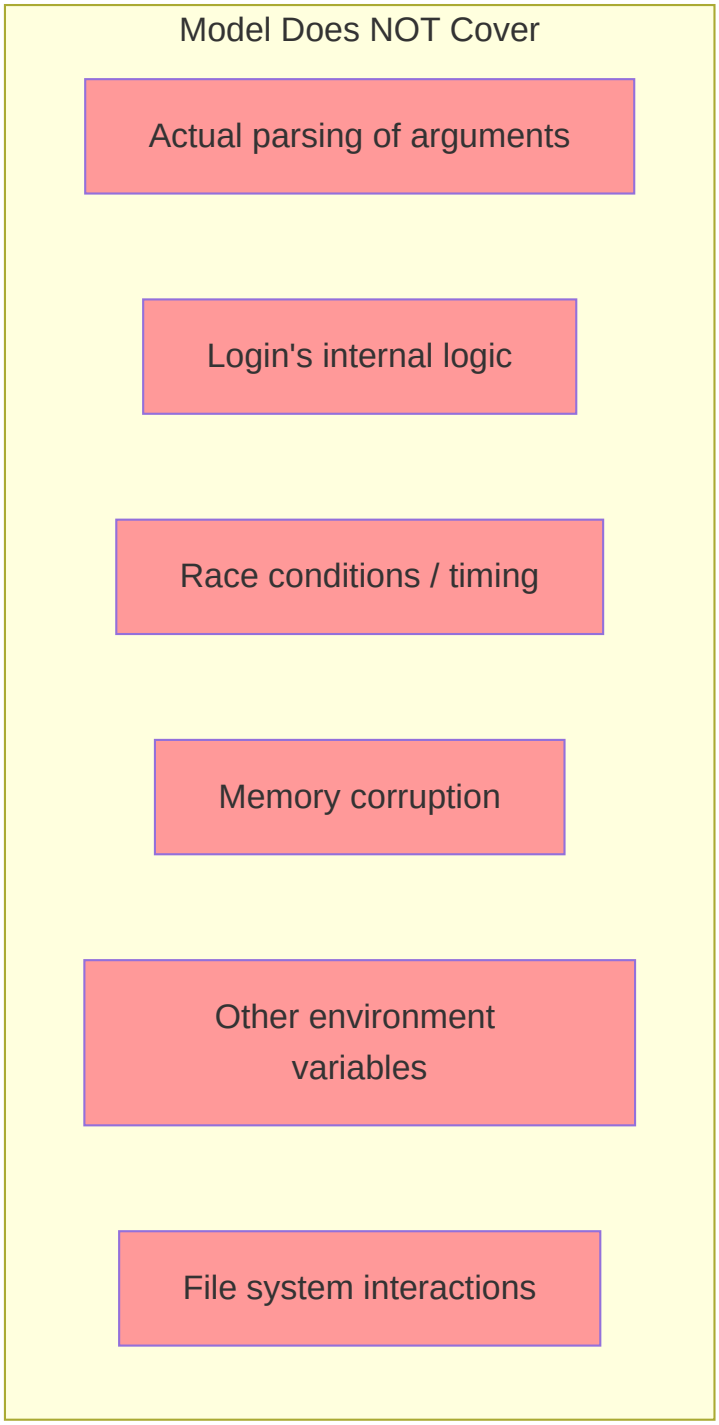
Fix Approach	Model Equivalent	Effect
Blocklist (strip `-`)	`SanitizationPolicy = PolicyBlocklist`	Removes known-bad patterns
Allowlist (regex)	`SanitizationPolicy = PolicyAllowlist`	Only permits known-good patterns
Drop untrusted	Don't include `%U` in template	Network data never reaches AuthService args

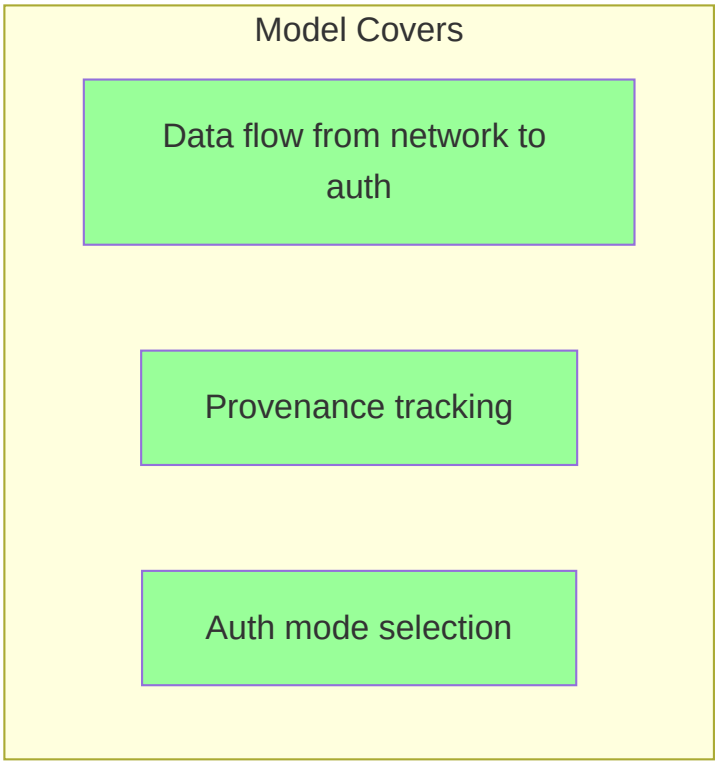
The model verified that **Allowlist** policy results in no invariant violations. This maps to: if `sanitize_username()` correctly rejects all flag-injection patterns, the auth bypass is prevented.

Brian Williams · Vartia.ai · brian@vartia.ai

What Is NOT Proven

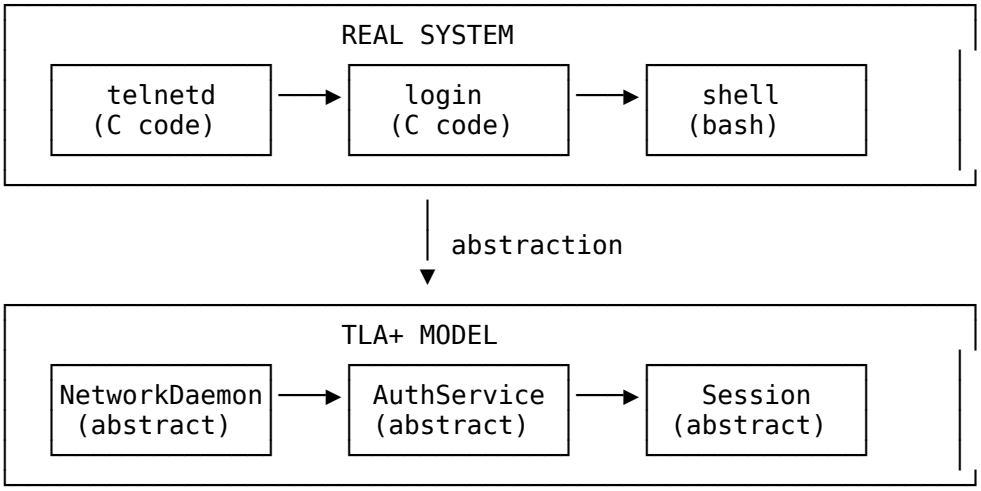
Limitations of the Model





Not Covered	Why It Matters
Actual argument parsing	Model assumes "PrivTriggerValue triggers TrustedAuth" abstractly; real login may have edge cases
Other input channels	Only modeled USER env var; other env vars or protocol fields may have similar issues
Implementation bugs	Sanitization logic itself could have bugs (buffer overflow, off-by-one, etc.)
Concurrency	Model is sequential; race conditions not considered
Completeness of allowlist	Model assumes allowlist correctly identifies all dangerous patterns

The Abstraction Gap



The model proves properties of the abstraction, not the implementation.

If the real system behaves differently than the model assumes, the guarantees don't transfer. Specifically:

1. **We assume** NetworkDaemon passes data to AuthService via ExecArgs/ExecEnv
2. **We assume** AuthService has exactly two auth modes (Normal/Trusted)
3. **We assume** sanitization correctly neutralizes all trigger patterns

If any assumption is wrong, the proof doesn't apply.

Conclusions

What We Can Claim

✅ **The architectural pattern is sound**: If a network daemon applies allowlist sanitization before passing data to an auth service, and the sanitization correctly identifies all privileged-mode triggers, then no network input can trigger authentication bypass.

✅ **The vulnerability class is real**: Without sanitization, the data flow pattern permits authentication bypass. This is a structural property, not dependent on specific exploits.

What We Cannot Claim

❌ **The real implementation is secure**: The model doesn't verify the actual C code.

❌ **The sanitization is complete**: The model assumes the allowlist catches all dangerous patterns; this must be verified separately.

❌ **No other vulnerabilities exist**: Only this specific data flow was modeled.

Recommendations

1. **Implement allowlist sanitization** for all data flowing from network to privileged services
2. **Audit the actual code** to verify it matches the model's assumptions
3. **Extend the model** to cover other input channels (other env vars, protocol fields)
4. **Test the sanitization** with fuzzing to find patterns the allowlist might miss

---

Brian Williams · Vartia.ai · brian@vartia.ai

Appendix: Model Sketch

```
CONSTANTS
    Network, Config, Hardcoded     \* Provenance levels
    SafeValue, PrivTriggerValue    \* Abstract input values
    SanitizationPolicy             \* None or Allowlist

VARIABLES
    phase                          \* Idle → Processing → AuthInvoked → Session/Denied
    remoteEnvUser                  \* Input value from network
    execArgsProvenance             \* Provenance of data sent to AuthService
    authModeUsed                   \* NormalAuth or TrustedAuth

INVARIANT AuthenticationIntegrity ==
    authModeUsed = "TrustedAuth" => execArgsProvenance ≠ Network
```

Full specification: `TrustBoundary.tla` (available upon request).

Contact

Brian Williams
Vartia.ai
brian@vartia.ai

If you have legacy network daemons or privileged authentication boundaries you’re concerned about, I offer short-trust boundary audits that produce explicit boundary contracts, checkable invariants, and concrete regression tests.

Book a call: https://vartia.ai/book.html