Formal specification and verification of the IETF MUD protocol

Simone Sampognaro s322918

Prof. Riccardo Sisto
PhD student Simone Bussa
02TYAUV - Security verification and testing

March 21, 2025

Contents

1 Introduction

The Manufacturer Usage Description (MUD) is an IETF standard designed to enhance the security and manageability of IoT devices with limited computational capabilities, such as smart home devices, sensors, and cameras. Due to their constrained nature, these devices often lack sophisticated security mechanisms, making them attractive targets for cyberattacks. To address this issue, MUD establishes a framework that allows IoT devices to declare their expected behavior and security requirements to the network infrastructure. Based on this information, the network enforces a whitelist-based policy, restricting device communication to only predefined and authorized interactions.
This report focuses on analyzing the security of the MUD protocol by developing a comprehensive threat model, identifying potential vulnerabilities, and assessing their impact on the protocol's security. Following this, the formal verification of the protocol will be conducted using Proverif, ensuring that it meets the specified security requirements and is resilient to potential attacks. The objective is to provide a structured security analysis of MUD and verify its effectiveness in protecting IoT networks.

2 Standard Description

The Manufacturer Usage Description (MUD) protocol gives manufacturers of network-connected devices (Things) a way to communicate intended network capabilities and restrictions to a MUD manager in the network. By doing so:

- **It reduces the threat surface** for IoT and other specialized devices by limiting their traffic to only the endpoints and protocols the manufacturer considers necessary.
- **It simplifies network policy configuration**, especially as new device types continuously appear. Instead of administrators crafting custom rules for each device, MUD can automatically generate suitable policies, using a file provided by (or on behalf of) the manufacturer.
- **It can accelerate response to vulnerabilities**: if a manufacturer discovers a problem, they can update the MUD file to change the device's recommended access patterns or impose new constraints. A MUD manager checking periodically for updates can then automatically adjust policies.
- **It keeps implementation costs low** by relying on simple, structured descriptions (based on YANG and JSON) to define device behavior.

MUD is therefore meant primarily to protect devices from unwanted incoming or outgoing connections, although it can also (in some deployments) reduce negative impacts a compromised device might have on the rest of the network.

2.1 Terminology

RFC 8520 defines several key terms relevant to MUD:

- **MUD File**: A JSON-encoded file containing (based on a YANG model) policy rules describing how a device should communicate. This file is hosted by or on behalf of the manufacturer.
- **MUD Manager**: A system or software component that (1) receives or collects MUD URLs from devices, (2) retrieves the corresponding MUD files, and (3) processes them, ultimately configuring the network elements (switches, routers, firewalls, or other enforcement points) so that the device's traffic is restricted as recommended.
- **MUD URL**: An HTTPS-based URL that uniquely identifies both the device type and where to retrieve the corresponding MUD file.

- **Thing**: A network-connected device that emits a MUD URL (e.g., a light bulb, thermostat, or other specialized IoT device).
- **Manufacturer**: The entity responsible for configuring the device to emit the MUD URL and for defining the corresponding network behavior in the MUD file. This role may be assumed by the device maker, a systems integrator, or a component provider.

2.2 MUD Architecture and Worflow

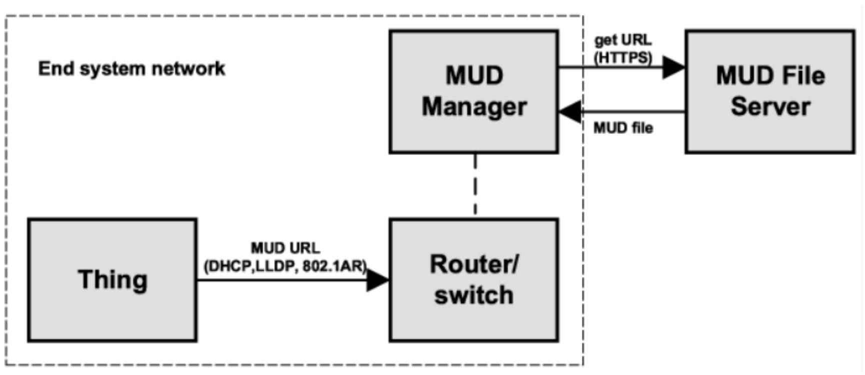


Figure 1: MUD architecture

While MUD may be integrated flexibly into different networks, a common sequence of operation is:

1. The device (Thing) emits a MUD URL.
2. The nearest network switch forwards the URL to a MUD manager. The specific forwarding mechanism depends on how the MUD URL is communicated (e.g., via DHCP, LLDP, or an X.509 certificate).
3. The MUD manager retrieves the MUD file and its signature from the designated MUD file server, provided it does not already have a cached copy. It then validates the signature and may check the URL and any referenced hosts against a reputation service.
4. The MUD manager might request approval from the administrator before adding the device (Thing) and applying its corresponding policy. However, if the device or its type is already recognized, this step can be skipped.
5. Once approved, the MUD manager applies network configurations based on the policy definitions provided in the MUD file.
6. The MUD manager then configures the nearest switch to enforce the policies. Additional network systems may also be updated as needed.
7. The information provided by the MUD file server remains valid for as long as the device stays connected or as defined in its description. When the device disconnects, any related configuration on the switch can be removed, ensuring that unused access rules do not persist unnecessarily. Additionally, the MUD file may be updated periodically to reflect changes in the device's functionality, communication patterns, or security requirements.

2.3 Finding a Policy: The MUD URL

As previously mentioned, each Thing in MUD is assigned a unique MUD URL that specifies its manufacturer and model. For example, a temperature sensor might use the URL:

`https://www.example.net/mudfiles/temperature_sensor/`

while a smart lightbulb could have a URL such as:

`https://example.com/lightbulbs/colour/v1`

For secure retrieval of the policy file, this URL must always use the HTTPS scheme to authenticate the MUD file server and ensure the integrity of the MUD file. There are several ways a device can provide its URL:

- **DHCP Option**: The device includes a custom DHCP option (in DHCPv4 or DHCPv6) with the MUD URL.
- **X.509 Certificate Extension**: For devices that use 802.1X or other certificate-based authentication, the MUD URL can be included as a non-critical certificate extension (such as in 802.1AR).
- **LLDP**: The MUD URL can also be transmitted within an LLDP packet as a vendor-specific Type-Length-Value (TLV) extension.

2.4 Types of Policies

MUD files define access control rules that prioritize abstract definitions over static IP addresses, which are instantiated into actual IP addresses through local processing.
The file's structure follows standardized YANG modules, notably `ietf-access-control-list` and `ietf-acldns`, which define how to specify access rules. Additionally, MUD managers should support only a limited set of ACL features, such as `match-on-ipv4`, `match-on-ipv6`, `match-on-tcp`, `match-on-udp`, and `match-on-icmp`. The model also enforces simple access control actions, such as "accept" or "drop", avoiding complex policies that might require manual intervention by network administrators. Examples include:

- **Manufacturer-Based Policies**: Traffic only to or from devices whose MUD URL's domain name matches a specific manufacturer or the same authority as the requesting device.
- **Local Networks**: The device may be permitted to communicate only on a local subnet or within certain locally defined scopes.
- **Controller**: The device may need to talk to a dedicated "controller" or backend service. The MUD file can reference a "controller" class that the local administrator populates (e.g., IP addresses or domain names for that controller).
- **DNS or NTP**: MUD files often allow DNS and NTP by default (since these services are frequently required).

Two default behaviors are always applied alongside any explicitly defined policies. First, any action that is not explicitly allowed is automatically blocked. Second, communication with local DNS and NTP services is permitted by default, ensuring that the device can resolve domain names and synchronize time without additional configuration.

2.5 Creating and Verifying a MUD File Signature

To ensure the integrity and authenticity of MUD files, they must be signed using Cryptographic Message Syntax (CMS) with DER encoding, guaranteeing that any policy statement genuinely originates from the manufacturer or an authorized delegate.

1. **Signature Creation**: The MUD file is signed as a binary CMS object. To improve verification success rates, intermediate certificates should be included in the signature. The signature file is stored at the location specified within the MUD file and is distributed using the content type `application/pkcs7-signature`.

2. **Signature Verification**: Before processing a MUD file, the MUD manager must retrieve and validate its signature. The signing certificate must contain the Key Usage Extension with the `digitalSignature(0)` bit enabled. If the certificate includes the `id-pe-mudsigner` extension, the signature must originate from a certificate whose subject matches this extension. If these conditions are not met, or if the certificate chain cannot be validated against a trusted root, the MUD manager must halt processing until an administrator intervenes.

3 Threat Model

The next step focuses on developing a threat model, which is a structured approach used to identify, evaluate, and mitigate potential security risks in a system. It helps in understanding the attack surface, assessing vulnerabilities, and defining security measures to reduce risks. To systematically classify and analyze threats, I adopted the **STRIDE** methodology, which categorizes threats into six main types:

- **Spoofing**: Impersonating an entity to gain unauthorized access.
- **Tampering**: Modifying data or communications to alter behavior.
- **Repudiation**: Performing actions without the ability to trace accountability.
- **Information Disclosure**: Unauthorized access to sensitive data.
- **Denial of Service**: Disrupting system availability.
- **Elevation of Privilege**: Gaining unauthorized higher-level access.

To construct the threat model for MUD, I used a combination of approaches. I utilized the Microsoft Threat Modeling Tool and reviewed relevant research papers from the literature to gain insights into potential threats and mitigation strategies in IoT and network security. References to these sources will be provided at the end of this report.

3.1 Microsoft Threat Modeling Tool

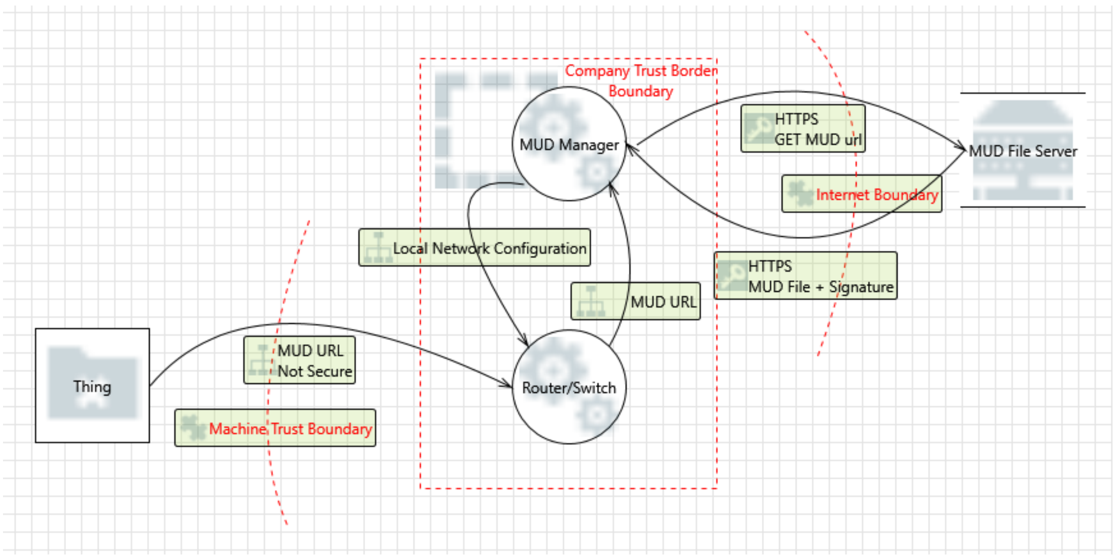


Figure 2: Threat Model Diagram

The Microsoft Threat Modeling Tool is designed to systematically identify threats in a system by mapping out its components, data flows, and trust boundaries. It leverages the STRIDE classification.

3.1.1 Components

Each component is configured with security-relevant attributes, which help the Threat Modeling Tool generate relevant threats to assess. Using the tool, the following elements were modeled to reflect the key components of the MUD architecture:

- **Thing**: Modeled as a **Generic External Interactor**, that represents an external entity interacting with the system, such as users or external devices, which may or may not have authentication mechanisms.
    - **Element Properties:**
        - Does not authenticate itself.
- **Router/Switch**: Modeled as a **Process**, that represents an active component that performs operations, processes data, and communicates with other elements. It can execute locally or as a network service.
    - **Element Properties:**
        - Running as network service.
        - Does not implement authentication mechanism.
        - Uses authorization mechanism: the switch applies the access control rules defined in the MUD profile.
        - Uses communication protocols: it indicates that the component is involved in network communication and could be a target for network-based attacks
- **MUD Manager**: Modeled as a **Process**.
    - **Element Properties:**
        - Running as local service.
        - Implements authentication mechanism: it authenticates the MUD File Server using HTTPS
        - Uses authorization mechanism: it ensures that only authorized devices can interact with specific parts of the network according to predefined rules.
        - Uses communication protocols.
- **MUD File Server**: Modeled as a **Generic Data Store**, that represents a storage element where data is persisted. It can be a database, file system, or any other structured storage medium.
    - **Element Properties:**
        - Store type: File system.
        - Signed: it means that the data stored within it is cryptographically signed to ensure its integrity and authenticity.

3.1.2 Data Flows and Trust Boundaries

Data flows define how information moves between the components. As data moves across different trust boundaries, the tool helps identify potential threats that could arise from crossing these boundaries. Trust boundaries are central to threat modeling because they show where data crosses from one level of trust to another—potentially exposing the system to additional risk. The following data flows and boundaries were modeled:

- **Thing → Router/Switch**: Data flow crosses the **Machine Trust Boundary**.
    - This boundary represents communication from an external, unauthenticated device into the network infrastructure.

- **MUD Manager ↔ Router/Switch**: Data flow occurs inside the **Company Trust Boundary**.
  - Here, the communication is assumed to be within a controlled and more trusted environment, but still subject to insider threats and misconfigurations.
- **MUD Manager ↔ MUD File Server**: The HTTPS data flow crosses the **Internet Trust Boundary**.
  - This boundary separates internal systems (MUD Manager) from the external server (MUD File Server).
  - Data is protected by HTTPS, offering server authentication, confidentiality, and integrity.

3.2 Identified Threats and Risk Analysis

I identified several potential threats that could impact MUD's security, either by exploiting its policy enforcement mechanisms or by targeting the communication between MUD components. The identified threats are grouped according to the STRIDE framework:

- **Spoofing**
  - **Forged MUD URL Emission**: An attacker forges a MUD URL to falsely claim the identity or capabilities of a device, potentially gaining unauthorized access to network resources.
  - **Compromised MUD File Server**: Attackers can manipulate MUD policies by either hijacking domain ownership or compromising the MUD file server. If a legitimate MUD URL's domain is hijacked or sold, the new owner can serve malicious MUD files while maintaining network trust. Similarly, if the server hosting MUD files is compromised, attackers can modify or replace MUD files to enforce unauthorized network policies.
  - **MUD URL Replay Attack**: An attacker reuses a previously valid MUD URL to mislead the MUD Manager into applying outdated or inappropriate policies or the attacker could replay a valid MUD URL belonging to another device to impersonate it and illegitimately trigger the application of its associated policies.
  - **Arbitrary Policy Injection**: Exploitation of weaknesses and impersonification in the MUD manager allows attackers to inject unauthorized network policies into the system.
  - **Cross-Manufacturer Trust Exploitation**: A device claims a MUD URL associated with a different manufacturer to gain unintended access, exploiting assumptions of inter-manufacturer trust.
  - **Rogue Certification Authority**: A malicious or compromised certification authority signs a certificate matching the MUDsigner field, enabling unauthorized substitution of MUD files.
  - **Lack of Authentication in IoT Devices**: An attacker can spoof a device with high network privileges by forging its MAC address (L2) or IP address (L3), effectively bypassing access controls. Since MUD enforces policies based on IP addresses and domain names rather than verifying the actual identity of a device, this type of attack can be undetected at the network layer.
- **Tampering**:
  - **Certificate and Key Management Issues**: Weak management of cryptographic keys and certificates can result in unauthorized access or interception.
  - **Malicious Manufacturer Exploit**: A rogue manufacturer exploits vulnerabilities in the MUD file parser to execute arbitrary code or manipulate the MUD Manager.

- **Repudiation**
- **Information Disclosure**: Passive observation of MUD URL emissions reveals device types, allowing attackers to identify potential vulnerabilities in network-connected devices, facilitating tailored attacks on the network.
- **Denial of Service**: Attackers may attempt to overload MUD file servers or managers with excessive requests, preventing legitimate devices from retrieving their policies, potentially exposing them to unauthorized network access.
- **Elevation of Privilege**:
  - **Policy Misconfiguration**: Errors in MUD file creation can introduce vulnerabilities due to overly permissive or restrictive policies.
  - **Privilege Escalation via Mixed Authentication**: Mixing insecure and secure MUD URL emission methods in the same device class allows attackers to elevate privileges by exploiting weaker methods.

4 Formal Verification with Proverif

ProVerif is an advanced tool for the automatic analysis of cryptographic protocols, designed to verify security properties such as **secrecy, authentication, privacy, traceability, and verifiability**. It supports a wide range of cryptographic primitives, including symmetric and asymmetric encryption, digital signatures, hash functions, bit-commitment, and non-interactive zero-knowledge proofs. One of its key strengths is its ability to analyze protocols with an unbounded number of sessions and an unbounded message space, making it particularly valuable for real-world security evaluations. When a security property **cannot be proven**, ProVerif attempts to reconstruct an attack trace, demonstrating a possible execution that violates the desired property.
The input to ProVerif consists of a script that defines the protocol to be analyzed. This script is composed of:

- **Declarations of operations**: definitions of cryptographic primitives such as encryption, signatures, and hash functions.
- **Process macros**: reusable definitions of processes.
- **Main process**: the central definition that models the behavior and interactions of protocol participants
- **Queries**: security properties that ProVerif is required to verify, such as checking whether a secret remains confidential or if authentication holds.

ProVerif primarily operates on input files written in **the typed pi calculus**, a formal language for modeling concurrent processes communicating over public channels such as the Internet. This calculus is particularly useful for describing cryptographic protocols, where adversaries are modeled using the **Dolev-Yao threat model**. Under this assumption, attackers have complete control over communication channels and can intercept, modify, delete, and inject messages, but they are limited to performing cryptographic operations only if they possess the required keys. ProVerif assumes perfect cryptography, meaning that attackers cannot break encryption through computational means but can only apply the cryptographic primitives explicitly defined by the protocol.
The tool leverages **Horn clauses** to approximate protocol behavior based on the extended pi calculus. However, this approximation introduces certain limitations. Despite these approximations, the model has been **proven sound**, meaning that if ProVerif establishes that a secrecy property holds within the Horn clause model, it also holds in the original pi calculus model. However, due to **over-approximation**, false positives may occur, particularly in cases where the model fails to precisely capture message freshness or repetitions in protocol execution.

4.1 Declarations of Operations

The declaration of operations in this ProVerif script defines the necessary types, cryptographic functions, and protocol-related operations required for verifying the security properties of the Manufacturer Usage Description protocol. Below is a breakdown of the key types and functions used in the script.
Three public communication channels are defined, modeling the interactions between different entities:

- **c**: Used for communication between the Thing and the Switch.
- **c1**: Used for communication between the Switch and the Manager.
- **https**: Represents secure HTTPS communication between the Manager and the MUD Server.

In addition to the basic features of the ProVerif language, which include standard data types for common cryptographic operations—such as public and private keys, symmetric keys, and results, I have introduced two additional types: **url**, representing a MUD URL, and **certype**, modeling public key certificates.
Beyond standard cryptographic operations, I have also defined custom functions and reduction rules to extend the model's capabilities:

4.1.1 Message Authentication Code (MAC)

To ensure message integrity and authentication, I introduced the **HMAC function**:

- `HMAC(bitstring, key): bitstring` – Computes a hash-based MAC using a given key.
- `verifyHMAC(HMAC(x, y), x, y) = ok()` – Verifies the authenticity of the message `x` by checking its MAC, which is computed using the key `y`.

4.1.2 Pseudo-Random Function (PRF) for Key Derivation

The **PRF function** is used for deriving cryptographic keys in the TLS handshake protocol:

- `PRF(bitstring, bitstring, bitstring): key` – Models key expansion as performed in TLS key derivation, where it is used to generate session keys securely.

4.1.3 URL Handling Functions

Since MUD relies on URLs to fetch security policies, I defined:

- `constructURL(bitstring, bitstring): url` – Constructs a MUD URL from a hostname and a path.
- `getHostnameFromURL(constructURL(hostname, path)) = hostname` – Extracts the hostname from a constructed URL.

4.1.4 Certificate Management

I defined custom certificate structures to enable the server process to authenticate itself and sign the MUD file, as well as to allow IoT devices to securely transmit the MUD URL:

- `cert(bitstring, pkey, skey): certype [private]` – Generates a MUD server certificate, binding a subject to a public key and signing it using the private key of a Certificate Authority.

- `certDevice(bitstring, url, skey): certype [private]` – Represents device certificates, where the first `bitstring` parameter corresponds to the `id-pe-mudsigner` extension, the `url` represents the MUD URL, and the `skey` is the private key of the same CA that issued the MUD server certificate for that specific URL. In accordance with IEEE 802.1AR certificates, the CA may be operated directly by the manufacturer or on behalf of the manufacturer. To avoid adding complexity to the model by representing certificate chains, I have assumed the same CA as the issuer for both types of certificates. This simplification is justified by the fact that, in certificate verification, this CA is inherently the root of trust in both cases.
- `checkcert(cert(subject, pubkey, sk), pk(sk)) = ok()` – Verifies that a certificate was issued by a valid CA by checking it against the CA's public key.
- `checkhostname(cert(subject, pubkey, sk), constructURL(subject, path)) = ok()` – Ensures that the server certificate's hostname matches the MUD URL hostname. This function is used because, as stated in RFC 8520, the controller **MUST** validate the certificate following the rules defined in **RFC2818, Section 3.1**.

4.1.5 MUD File Creation and Parsing

To model MUD file handling, I introduced:

- `makeMUDfile(url, bitstring): bitstring` – Creates a MUD file using a MUD URL and an ACL.
- `getMUDurl(makeMUDfile(MUDurl, acl)) = MUDurl` – Extracts the MUD URL from the file.
- `getAclList(makeMUDfile(MUDurl, acl)) = acl` – Extracts the Access Control List.

4.2 Process Macros

I have represented the main entities of the MUD protocol—namely, the Thing, the Switch, the MUD Manager, and the MUD File Server—as distinct processes. Each process is defined using the `let` construct and operates independently, communicating through defined channels.
To model different security levels in the transmission of the MUD URL, I have implemented two distinct models: one where the MUD URL is transmitted through an **insecure method** (e.g., DHCP or LLDP), and another where it is **securely** conveyed using an **X.509 certificate**. Throughout the text, I will provide a detailed breakdown of each process, specifying the cryptographic operations they perform and the arguments they take as input parameters.

4.2.1 Thing

This process takes as an argument either a certificate of type `certype`, representing the issuance of a MUD URL through an X.509 constraint, or a direct MUD URL of type `url`. The Thing process then transmits this argument over the communication channel `c` to the Switch and then the event `thingSentURL(url)` is triggered.

4.2.2 Switch

This process is responsible for receiving a MUD URL from the Thing in various forms, forwarding it to the MUD Manager on channel `c1`, and later applying the configuration received from the manager; then the event `switchReceivedConfiguration(url, configuration)` is triggered.

4.2.3 MUD Manager

The process differs depending on the transmission method used for the MUD URL. Regardless of whether the transmission method, this process always performs a **TLS-1.2 handshake** with the MUD File Server. This handshake is essential to ensure the security properties required by RFC 8520, which states:

> MUD URLs are required to use the `https` scheme, in order to establish the MUD file server's identity and assure integrity of the MUD file.

By enforcing this requirement, the MUD Manager ensures that the `https` communication channel with the MUD File Server provides **server and message authentication**.
Additionally, in both cases, the MUD Manager always takes as an input argument the **public key of the trusted CA**, which is used to verify the MUD File Server's certificate. This aligns with the Security Considerations of RFC 8520, ensuring that only trusted servers are allowed to provide MUD files.
Based on what has been stated, the following two validation checks are performed on the MUD File Server's certificate before proceeding with any further communication:

```
(* The hostname in the server's certificate must match the hostname in the MUD
    URL *)
if checkhostname(certServer, getURL(certThingX)) = ok() then

(* The server's certificate must be issued by the trusted CA *)
if checkcert(certServer, pkCA) = ok() then
```

After verifying the Finished message received from the server, the `serverAuthnSuccessful(certificate)` event is triggered and the handshake is successfully completed. Then, the actual MUD protocol exchange begins.
The MUD Manager sends the previously received MUD URL (after its reception, the event `managerReceivedURL(url)` is triggered) to the MUD File Server over the HTTPS channel. The security of the exchanged messages is further enhanced by HMAC authentication, using a key derived from the master secret established during the handshake.
Once the MUD Manager receives the MUD file from the server, it must validate that the file corresponds to the expected device and that it has not been tampered with during transmission.

```
(* x:MUD signature file, y: certificate used for signing, z:MAC of the message *)
in(https, (x:bitstring, y:certype, z:bitstring));
```

The first check performed is to verify that the MUD URL embedded in the received file matches the expected MUD URL initially transmitted by the switch. This ensures that the MUD Manager is processing the correct policy file associated with the device and prevents an attacker from injecting an unauthorized MUD file.

```
(* Verify that the URL in the received MUD file matches the expected one received
     from the switch *)
if(getMUDurl(getmess(x))) = MUDurlX then
```

After confirming the MUD URL, the integrity of the received message is validated using an HMAC. This step ensures that no potential tampering or message injection occurred.

```
(* Verify the integrity and authenticity of the received message using HMAC *)
if verifyHMAC(z,(x,y),keyMAC) = ok() then
```

Once the integrity of the message (containing the file and the certificate) is established, the MUD Manager verifies that the certificate used to sign the MUD file was issued by a trusted CA. The model accounts for the possibility that the server may use different certificates for establishing the TLS connection and for signing the MUD file, ensuring that the appropriate validation checks are applied to each certificate independently.

```
(* Check if the server certificate y is valid signed by the trusted CA (pkCA) *)
if checkcert(y,pkCA) = ok() then
```

Finally, the MUD Manager verifies the digital signature on the MUD file. This step confirms that the contents of the file have not been modified and that it was indeed signed by the owner of the certificate previously verified.

```
(* Verify the digital signature of the MUD file using the public key from server
    certificate y *)
if checksign(x, getPk(y))= ok() then
```

If any of these verification steps fail, the MUD file is not accepted, and no security policies are applied.
Once all verifications are successfully completed, the event `managerReceivedMUDfile(certificate, file)` is triggered, and the MUD Manager extracts the contents of the MUD file, which typically include access control rules. It then proceeds to instantiate the appropriate network configurations for the associated device and sends them to the Switch, only after triggering the event `managerSentConfiguration(url, configuration)`.

4.2.4 MUD File Server

This process is responsible for generating MUD files that define the appropriate access control policies for network devices. It receives the following input parameters from the main process:

- `certServer:certype`: The server's certificate, which is used to authenticate itself to the MUD Manager during the TLS handshake and to provide a trusted signing key for MUD files.
- `skServer:skey`: The private key of the server, corresponding to `certServer`, which is used to decrypt messages during the TLS handshake and to digitally sign MUD files before sending them to the MUD Manager.
- `MUDurl:url`: The MUD URL associated with the Thing, for which the manufacturer manages the file server.

It first establishes a secure communication channel by performing a TLS 1.2 handshake with the MUD Manager. The event `serverAuthnStarted(certificate)` is placed before the server's certificate is sent. With the secure TLS session established, the MUD protocol exchange begins. The server first derives a MAC key `keyMAC` from the master secret, which will be used to authenticate subsequent messages.

```
(* Derive a MAC key for MUD message authentication from the master secret *)
let keyMAC = PRF(key2bit(masterSecret), serverRandom, clientRandom) in
```

It then waits for the MUD URL, transmitted by the MUD Manager along with an HMAC over the `https` channel. Upon receiving this data, the server verifies its integrity. Additionally, a matching constraint is applied to the received MUD URL, ensuring that it exactly matches the expected value provided as an input parameter. This guarantees that the server only processes requests for the correct MUD URL.

```
(* Receive the MUD URL and its associated HMAC *)
in(https,(=MUDurl, MAC:bitstring));

(* Verify the integrity and authenticity of the received message using HMAC *)
if verifyHMAC(MAC, url2bit(MUDurl), keyMAC) = ok() then
```

After these two verifications, the event `serverReceivedURL(url)` is triggered. Next, the server generates a MUD file for the device corresponding to the received MUD URL.

```
(* Generate a new Access Control List and a MUD file for the device *)
new aclList: bitstring;
let MUDfile = makeMUDfile(MUDurl,aclList) in
```

Before sending the MUD file to the MUD Manager, the server ensures its authenticity by digitally signing it using its private key `skServer`. The event `serverSentMUDfile(certificate, file)` is triggered and then signed file is then transmitted along with the server's certificate `certServer` and an HMAC is computed over both the signed MUD file and the server's certificate.

```
(* Send the signed MUD file, the server's certificate, and an HMAC for integrity
    verification *)
out(https, (sign(MUDfile, skServer), certServer, HMAC( (sign(MUDfile, skServer),
    certServer), keyMAC) ));
```

4.2.5 Main Process

The main process in this ProVerif model establishes the overall structure of the protocol by creating cryptographic keys, certificates, MUD URLs, and instantiating the main entities of the MUD protocol. It ensures that both legitimate and adversarial conditions are modeled, allowing verification of the protocol's security properties.

1. **Creation of Certification Authorities (CAs)**
   The model first generates two Certificate Authorities (CAs):

   - A **trusted CA**, which is responsible for issuing valid certificates for the MUD File Servers and Things.
   - A **rogue CA**, which represents a potential attacker capable of issuing fraudulent certificates.

   For both CAs, a private key (`skCA`, `skRogueCA`) is generated, and the corresponding public key (`pkCA`, `pkRogueCA`) is derived and made publicly available. Additionally, the private key of the rogue CA (`skRogueCA`) is also exposed, allowing an adversary to issue forged certificates. This simulates an environment where an attacker has control over an unauthorized CA and can create fraudulent certificates

2. **Creation of MUD File Server Certificates**
   The model then creates multiple MUD File Servers, each with its own private key and public key. These keys are used to generate valid X.509 certificates (`certServer`, `certServer2`), which are signed by the trusted CA (`skCA`). Additionally, the model includes an attacker scenario where the rogue CA issues a fraudulent certificate (`certAttacker`) for a server using the same subject name (`subjectServer`) as a legitimate one. This models an attack scenario described in RFC 8520, Section 16, where an attacker may attempt to substitute a MUD file by forging a certificate with a misleading subject name.

3. **Generation of MUD URLs and Device Certificates**
   Each Thing has an associated MUD URL. The main process constructs two MUD URLs (`MUDurl`, `MUDurl2`), where:

   - The MUD URL is constructed using the hostname of the manufacturer's file server (`subjectServer`, `subjectServer2`) along with a predefined path.
   - Depending on the transmission method used in the model, the main process either generates only the MUD URL for the Things or additionally issues a certificate (`certThing`, `certThing2`) signed by the trusted CA, which includes the MUD URL and the `id_pe_mudsigner` extension.
   - In the case of MUD URL transmission via certificates, the main process also generates a fraudulent certificate for a malicious device, signed by a rogue CA. This simulates an attack scenario where an adversary attempts to use an illegitimate certificate to impersonate a legitimate device.

4. **Instantiation of MUD Protocol Entities**
   The final part of the main process instantiates the entities of the protocol. Each entity runs in parallel (`|!()`), representing multiple simultaneous instances. The instantiated entities are:

   - **Two MUD File Servers** (`MUD_File_Server`)
   - **MUD Manager** (`MUD_Manager`)
   - **Two IoT Devices** (`Thing`)
   - **Switch** (`Switch`)

4.3 Queries

In ProVerif, security properties are expressed through various types of queries, with reachability properties and correspondence assertions playing a fundamental role. In this project, both types of queries were used to evaluate the security of the MUD protocol.

4.3.1 Reachability Properties

Reachability queries are used to determine whether a certain event can occur within the execution of the protocol. Typically, these tests are formulated to verify if a process can reach a particular state, such as an entity completing a specific operation. In this model, reachability queries take the following form:

$$\text{query event(endX()).}$$

where `endX()` represents the successful completion of an operation by an entity **X**. If the verification summary returns:

$$\text{Query not event(endX) is false.}$$

It indicates that the corresponding event must always occur, ensuring that the process terminates correctly.

4.3.2 Correspondence Assertions

Correspondence assertions are used to establish relationships between events in a protocol. They can be expressed in the form:

"If event e has been executed, then event e' must have been executed beforehand."

Additionally, these events can include arguments, allowing for an analysis of the relationships between their respective parameters. Injective correspondences enforce a **one-to-one** relationship between events. To verify an injective correspondence assertion, a specific query syntax is used:

$$\text{query } x_1 : t_1, \ldots, x_n : t_n; \quad \text{inj-event}(e(M_1, \ldots, M_j)) \Rightarrow \text{inj-event}(e'(N_1, \ldots, N_k))$$

Informally, this means that for every occurrence of event $e(M_1, \ldots, M_j)$, there must be a distinct **earlier** occurrence of event $e'(N_1, \ldots, N_k)$. Furthermore, the events must respect any relationships defined by their parameters. Specifically, the variables $x_1, \ldots, x_n$ must hold the same values in both $M_1, \ldots, M_j$ and $N_1, \ldots, N_k$, ensuring consistency in their correspondence.

4.3.3 Reachability Tests for Protocol Completion

The first set of queries verifies whether the key entities in the MUD protocol correctly complete their intended operations. The following events indicate successful completion of different stages:

- `endManager()` – The **MUD Manager** has processed the MUD file and sent configurations.
- `endFileServer()` – The **MUD File Server** has successfully sent the MUD file.
- `endThing()` – The **IoT device (Thing)** has sent the MUD URL.
- `endSwitch()` – The **Switch** has received and applied configurations.

The reachability queries check whether these events occur:

```
query event(endManager()).
query event(endFileServer()).
query event(endThing()).
query event(endSwitch()).
```

4.3.4 Verifying MUD File Integrity

If the MUD Manager has received a MUD file y signed with a valid certificate x, then it must have been sent by the legitimate MUD File Server:

```
query x:certype, y:bitstring;
    inj-event(managerReceivedMUDfile(x,y)) ==> inj-event(serverSentMUDfile(x,y)).
```

This property is formulated as an injective correspondence rather than a basic correspondence to accurately model the security guarantees provided by the protocol. The MUD File Server transmits the MUD file within a TLS-protected channel. Each TLS connection is uniquely associated with a specific server-manager interaction, meaning that its parameters, such as the session keys and random values exchanged during the handshake, are distinct for every connection between a single instance of the MUD File Server and a single instance of the MUD Manager. The query verifies whether each received MUD file corresponds to a single file transmission by the server within a TLS connection.

4.3.5 Verifying Server Authentication

The following query verifies that if a server is successfully authenticated using the certificate x, it must have correctly initiated the authentication process:

```
query x:certype;
      inj-event(serverAuthnSuccessful(x)) ==> inj-event(serverAuthnStarted(x)).
```

4.3.6 Verifying Switch Configuration Transmission

To ensure that network configurations sent by the MUD Manager are correctly received and applied by the Switch, the following query was tested:

```
query x:url, y:bitstring;
    inj-event(switchReceivedConfiguration(x,y))
                ==> inj-event(managerSentConfiguration(x,y)).
```

In this query, x represents the MUD URL that was forwarded from the Switch to the MUD Manager, while y corresponds to the network configurations derived from the MUD file associated with that specific MUD URL.
This property is formulated as an injective correspondence rather than a basic correspondence to prevent potential inconsistencies in network configurations. If a basic correspondence were used, multiple occurrences of `switchReceivedConfiguration(x,y)` could be associated with the same `managerSentConfiguration(x,y)`. This would allow for the possibility that a previously sent configuration associated with a URL is reapplied, potentially leading to outdated or incorrect network settings being enforced. By enforcing injectivity, the query verifies whether the configuration received at the Switch is uniquely linked to a single corresponding transmission from the MUD Manager.

4.3.7 Verifying MUD URL Transmission from IoT Device

This query ensures that if the MUD Manager receives a MUD URL, it must have been sent by the legitimate IoT device (Thing):

```
query x:url;
      inj-event(managerReceivedURL(x)) ==> inj-event(thingSentURL(x)).
```

The query is formulated as an injective correspondence to consider replay attacks. Using a basic correspondence would allow multiple occurrences of `managerReceivedURL(x)` to be linked to a single `thingSentURL(x)`, meaning an attacker could reuse a previously valid MUD URL to deceive the MUD Manager into applying outdated or inappropriate policies. More critically, the attacker could also send a valid MUD URL belonging to a different device, thereby falsely claiming the identity or capabilities of that device. By enforcing injectivity, this query verifies whether each received MUD URL originates from a distinct and legitimate transmission from the Thing.

4.3.8 Verifying MUD URL Transmission from MUD Manager to Server

This query ensures that if the MUD File Server receives a MUD URL, it must have been sent by the MUD Manager:

```
query x:url;
      inj-event(serverReceivedURL(x)) ==> inj-event(managerSentURL(x)).
```

As previously discussed regarding the TLS channel, each session is uniquely established between a single instance of the MUD Manager and a single instance of the MUD File Server and the MUD URL is transmitted within this secure channel.

5 Analysis of Results and Security Enhancements

5.1 Analysis of Results

The **reachability queries** produced this result in both models:

```
Query not event(endManager) is false.
Query not event(endFileServer) is false.
Query not event(endThing) is false.
Query not event(endSwitch) is false.
```

This indicates that the processes reach completion.
The **MUD File receival test** produced this result in both models:

```
Query inj-event(managerReceivedMUDfile(x_1,y_3)) ==>
        inj-event(serverSentMUDfile(x_1,y_3)) is true.
```

This confirms that every time the MUD Manager receives a MUD file, it must have originated from a legitimate MUD File Server, ensuring protection against MITM attacks and tampering. The server authentication test produced this result in both models:

```
Query inj-event(serverAuthnSuccessful(x_1))
        ==> inj-event(serverAuthnStarted(x_1)) is true.
```

This confirms that the MUD File Server's authentication process is properly enforced.
The **Switch configuration receival test** produced this result in both models:

```
Query inj-event(switchReceivedConfiguration(x_1,y_3))
        ==> inj-event(managerSentConfiguration(x_1,y_3)) is false.
```

This result suggests that the Switch may receive configurations that were not sent by the legitimate MUD Manager, possibly due to unauthorized configuration injection or lack of integrity verification. As shown by the attack graphs, in the case of the **insecure method** (Figure 4), the attacker is able to deceive the Switch by forging both the MUD URL (`a`) and the corresponding local configurations (`a_2`). In contrast, under the **secure method** (Figure 3), the attacker intercepts the device's certificate, forwards it to the Switch, and then impersonates the MUD Manager by generating and injecting forged configurations (`a_2`).

The **Manager receival of MUD URL test** produced this result in both models:

```
Query inj-event(managerReceivedURL(x_1))
        ==> inj-event(thingSentURL(x_1)) is false.
```

This indicates that an attacker may be able to impersonate a device and send a fake MUD URL to the MUD Manager. As illustrated by the attack graph, in the certificate-based transmission model (Figure 5), the attacker sends a **fraudulent device certificate** generated using a rogue CA. In the insecure method (Figure 6), the attacker directly forges the MUD URL without any certificate. In both cases, the attacker is able to transmit a MUD URL to the MUD Manager, highlighting a critical weakness: the lack of authentication in the MUD URL transmission process. Moreover, even in the absence of a fraudulent device certificate in the model, the result would still be **false**, as the attacker could simply forward a legitimate certificate belonging to

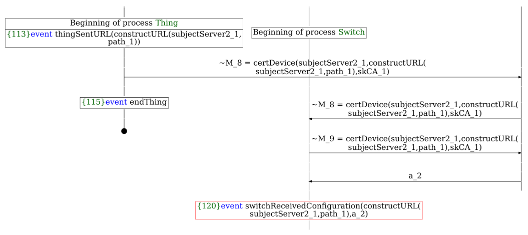

Beginning of process Thing
{113}event thingSentURL(constructURL(subjectServer2_1, path_1))
Beginning of process Switch
~M_8 = certDevice(subjectServer2_1,constructURL(subjectServer2_1,path_1),skCA_1)
{115}event endThing
~M_8 = certDevice(subjectServer2_1,constructURL(subjectServer2_1,path_1),skCA_1)
~M_9 = certDevice(subjectServer2_1,constructURL(subjectServer2_1,path_1),skCA_1)
a_2
{120}event switchReceivedConfiguration(constructURL(subjectServer2_1,path_1),a_2)

Figure 3: Secure Method

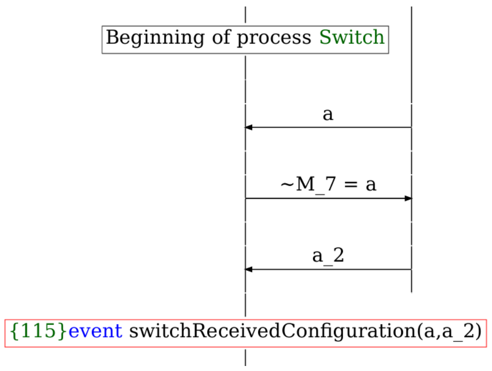

Beginning of process Switch
a
~M_7 = a
a_2
{115}event switchReceivedConfiguration(a,a_2)

Figure 4: Insecure Method

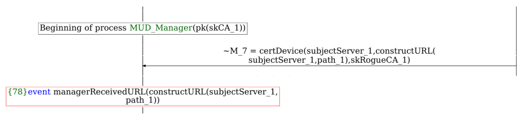

Beginning of process MUD_Manager(pk(skCA_1))
~M_7 = certDevice(subjectServer_1,constructURL(subjectServer_1,path_1),skRogueCA_1)
{78}event managerReceivedURL(constructURL(subjectServer_1, path_1))

Figure 5: Secure Method

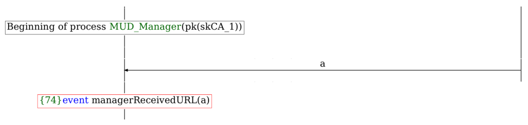

Beginning of process MUD_Manager(pk(skCA_1))
a
{74}event managerReceivedURL(a)

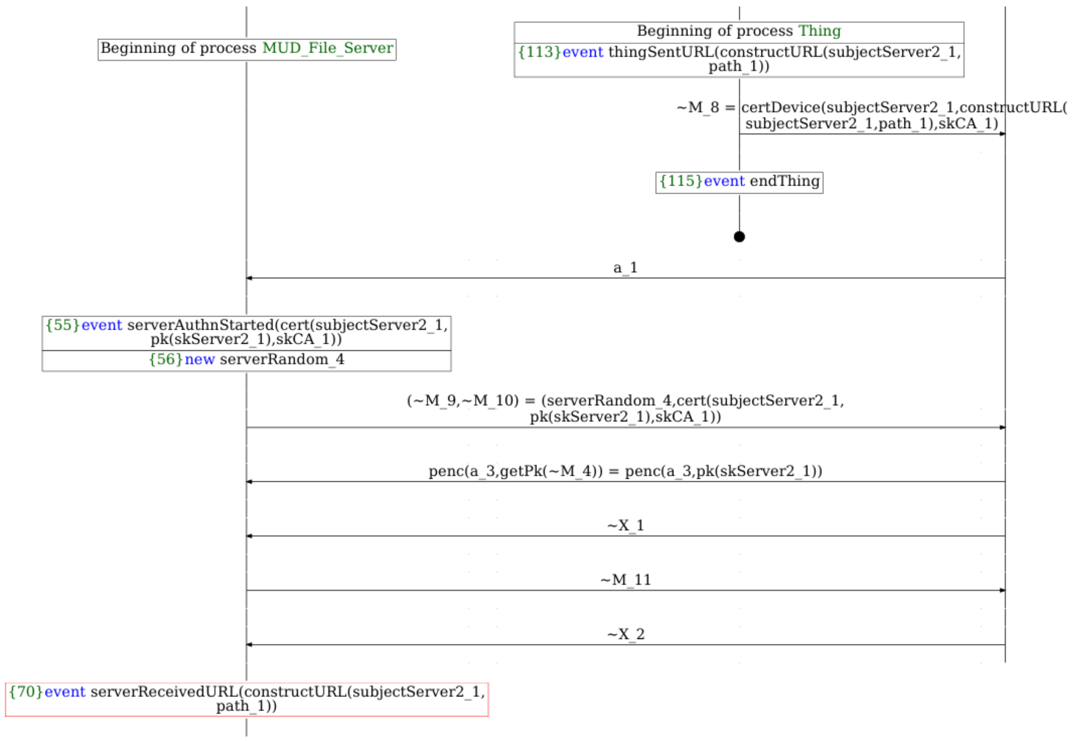
Figure 6: Insecure Method

another Thing, previously obtained from the communication channel, to a MUD Manager process.

The **Server receival of MUD URL test** produced this result in both models:

```
Query inj-event(serverReceivedURL(x_1))
    ==> inj-event(managerSentURL(x_1)) is false.
```

This suggests that the server may accept connection from unauthorized sources, likely due to the lack of MUD Manager authentication in the TLS handshake. As illustrated by the attack graph 7, the attacker is able to impersonate the MUD Manager during the TLS handshake after obtaining the MUD URL. The attacker initiates the handshake by sending a `ClientHello` message (`a_1`), followed by a `ClientKeyExchange` (`penc(a_3, ...)`) and a `Finished` (`X_1`) message, successfully establishing a TLS session with the server. Once the secure channel is in place, the attacker transmits the MUD URL, protected by an HMAC (`X_2`), as if it were a legitimate request from the MUD Manager.



Beginning of process MUD_File_Server
Beginning of process Thing
{113}event thingSentURL(constructURL(subjectServer2_1, path_1))
~M_8 = certDevice(subjectServer2_1,constructURL(subjectServer2_1,path_1),skCA_1)
{115}event endThing
a_1
{55}event serverAuthnStarted(cert(subjectServer2_1, pk(skServer2_1),skCA_1))
{56}new serverRandom_4
(~M_9,~M_10) = (serverRandom_4,cert(subjectServer2_1, pk(skServer2_1),skCA_1))
penc(a_3,getPk(~M_4)) = penc(a_3,pk(skServer2_1))
~X_1
~M_11
~X_2
{70}event serverReceivedURL(constructURL(subjectServer2_1, path_1))

Figure 7: Secure and Insecure Method

5.2 Security Enhancements

During the verification of the MUD protocol model, two major security weaknesses were identified:

- **Lack of device authentication in MUD URL transmission:** In both the insecure and certificate-based methods, there is no mechanism that ensures the MUD URL truly originates from the device. This enables attackers to replay previously valid URLs or impersonate devices by sending spoofed URLs.
- **Lack of Manager authentication and configuration integrity:** The Switch has no way to verify that the received network configuration truly originates from the MUD Manager. Without authentication or integrity protection, an attacker may inject unauthorized configurations, potentially misguiding the network behavior.

To address these issues, two protocol modifications were introduced to strengthen the authentication and integrity guarantees of the system, particularly on the local network side. These enhancements were applied to the certificate-based transmission model, assuming a corporate environment where network devices possess greater capabilities and where stronger security measures are essential.
To prevent **unauthorized entities from sending MUD URLs**, a certificate-based challenge-response mechanism was introduced between the Thing and the MUD Manager. The authentication process works as follows:

1. The Thing sends a certificate to the Manager. This certificate contains:
   - The MUD URL
   - The device's public key
2. The Manager generates a symmetric key and a nonce.
3. The symmetric key is encrypted using the device's public key and sent along with the nonce.

4. The Thing decrypts the symmetric key using its private key.
5. The Thing then encrypts the received nonce with the symmetric key and sends it back.
6. The Manager decrypts the response and verifies that the nonce matches the original one.
7. Upon successful verification, the Manager processes the MUD URL contained in the device's certificate.

This exchange ensures that only devices possessing the corresponding private key can respond correctly to the challenge. As a result, MUD URL transmission is tightly bound to the device's identity, preventing URL spoofing and impersonation attacks.
To secure **the configuration delivery from the MUD Manager** to the Switch, a digital signature mechanism was added. The MUD Manager is now equipped with an asymmetric key pair and signs the configuration message. The message also includes a nonce, originally sent by the Switch along with the MUD URL, to ensure freshness and prevent replay attacks. The Switch is assumed to have access to the Manager's public key (e.g., pre-installed or trusted within the same administrative domain, such as within a company network).
Upon receiving the configuration, the Switch verifies the digital signature and the validity of the nonce. Only if both checks succeed does the Switch apply the configuration, ensuring that it was sent by a legitimate MUD Manager.
The **verification summary** of this new model shows the same results as the previous ones, except for these two queries.

```
Query inj-event(switchReceivedConfiguration(x_4,y_4,z_2))
    ==> inj-event(managerSentConfiguration(x_4,y_4,z_2)) is true.

Query inj-event(managerReceivedURL(x_4))
    ==> inj-event(thingSentURL(x_4)) is true.
```

This demonstrates that the proposed changes are effective and successfully address the lack of entity authentication and message integrity identified during the formal analysis.

6 Conclusions

The security verification of the MUD model revealed both **successfully enforced properties** and **potential vulnerabilities**:

- **Verified Properties:**
  - The **MUD file transmission is correctly authenticated and protected from tampering**.
  - The **server authentication process is correctly enforced**.
- **Identified Vulnerabilities:**
  - The **Switch configuration application may be vulnerable to unauthorized modifications**.
  - The **MUD Manager can receive MUD URLs from unauthorized sources**.
  - The **MUD File Server may accept connections from unauthenticated entities** due to missing authentication in the TLS handshake, which could be exploited to exhaust server resources and potentially lead to a denial of service.

These findings provide critical insights into the security properties of the MUD protocol, helping to identify areas where additional integrity, authentication, and authorization mechanisms may be required.

7 References

- RFC 8520
- Defining the Behavior of IoT Devices Through the MUD Standard: Review, Challenges, and Research Directions
- MUDThread: Securing Constrained IoT Networks via Manufacturer Usage Descriptions
- The Impact of Manufacturer Usage Description (MUD) on IoT Security
- Role of Device Identification and Manufacturer Usage Description in IoT Security: A Survey