

Secret Management

- [Secret Management](#)

Secret Management

Hardware Overview & BIOS Setup (Intel PTT / TPM 2.0)

Enabling Intel PTT in BIOS (bmo95 / Motherboard Setup)

1. Reboot the machine and press `Del` or `F2` to enter UEFI BIOS.
 2. Press `F7` to switch to **Advanced Mode**.
 3. Go to the **Advanced** tab.
 4. Select **PCH-FW Configuration**.
 5. Set **TPM Device Selection** (or PTT) to **Enable Firmware TPM / PTT** (switch away from Discrete TPM / dTPM).
 6. Accept the Intel PTT confirmation prompt.
 7. *(Optional)* Under **Advanced -> Trusted Computing**, ensure **Security Device Support** is set to **Enable**.
 8. Press `F10` to save and restart.
-

Verification in Linux

Verify that the TPM 2.0 interface, character devices, and kernel drivers are active:

```
# Check device nodes
ls -l /dev/tpm* /dev/tpmrm*

# Check kernel driver initialization
dmesg | grep -i tpm

# Check systemd TPM 2.0 subsystem support
systemd-analyze has-tpm2
```

Expected output confirms `+firmware`, `+driver`, `+system`, `+subsystem`, and TSS2 user-space libraries are functional.

Capabilities: Everything You Can Do With the TPM 2.0 Chip

The TPM (Trusted Platform Module) is a secure crypto-processor embedded in hardware/firmware that provides tamper-resistant key storage, cryptographic operations, hardware entropy, and platform integrity measurement.

+-----+ +-----+ +-----+ +-----+											
TPM 2.0 Hardware / PTT											
+-----+ +-----+ +-----+ +-----+											
Hardware Cryptography			Data Sealing			Machine Identity					
- RSA / ECC keygen			- Lock secrets to			- Hardware Certs					
- Sign / Verify			firmware/boot PCRs			- mTLS & 802.1X					
- Hardware TRNG			- LUKS Auto-Unlock			- SSH Host Keys					
+-----+ +-----+ +-----+ +-----+											
+-----+ +-----+ +-----+ +-----+											
NVRAM Storage			Platform Attestation			Measured Boot					
- Tamper-proof indices			- Remote attestation			- PCR 0-23 state					
- Hardware monotonic			- Cryptographic state			- System integrity					
counters & auth			quotes & audit logs			verification					
+-----+ +-----+ +-----+ +-----+											
+-----+											

1. Encrypting & Decrypting Files & Secrets

The TPM can encrypt data using keys generated inside the chip. When data is sealed/encrypted with the TPM, it cannot be decrypted on any other physical machine.

A. Fast & Native File Encryption via `systemd-creds`

The modern, streamlined way to encrypt and decrypt arbitrary files or strings using the TPM:

```
# Encrypt text with the hardware TPM
echo -n "SuperSecretPassword123" | systemd-creds encrypt --with-key=tpm2 - /tmp/secret.enc

# Decrypt text directly from the TPM
systemd-creds decrypt /tmp/secret.enc -
```

B. Direct Key Generation & Asymmetric Encryption via tpm2-tools

Using raw TPM2 primitives to generate primary keys and encrypt files:

```
# 1. Create a Primary Key in the Owner Hierarchy (storage root)
tpm2_createprimary -C o -g sha256 -G rsa -c /tmp/primary.ctx

# 2. Generate a key pair inside the TPM
tpm2_create -C /tmp/primary.ctx -g sha256 -G rsa -u /tmp/key.pub -r /tmp/key.priv

# 3. Load the key into the TPM
tpm2_load -C /tmp/primary.ctx -u /tmp/key.pub -r /tmp/key.priv -c /tmp/key.ctx

# 4. Encrypt arbitrary data with the TPM key
echo "DatabaseAdminPassword" | tpm2_rsaencrypt -c /tmp/key.ctx -o /tmp/payload.enc

# 5. Decrypt using the hardware-protected key
tpm2_rsadecrypt -c /tmp/key.ctx -o /tmp/decrypted.txt /tmp/payload.enc
```

2. Platform Policy Sealing (Binding Secrets to Boot Integrity)

Sealing is the cryptographic process of locking a secret inside the TPM such that it can **only** be decrypted if specific Platform Configuration Registers (PCRs) match the exact authorized boot state (BIOS version, Secure Boot state, bootloader, kernel parameters).

- If an attacker modifies the kernel, alters BIOS settings, boots a live USB, or moves the drive to another machine, the PCR hash changes and the TPM **refuses** to unseal the secret.

```
# Seal a secret to PCR 0 (Firmware), PCR 2 (Option ROMs), and PCR 7 (Secure Boot)
systemd-creds encrypt --with-key=tpm2 --tpm2-pcrs=0+2+7 /path/to/plaintext.key
/path/to/sealed.key

# Unsealing works only if the current boot measurements match:
systemd-creds decrypt /path/to/sealed.key /path/to/plaintext.key
```

Common PCR assignments for sealing policies:

- **PCR 0:** Core UEFI/BIOS firmware and platform configuration.
- **PCR 2:** Option ROM code (e.g., GPU/NIC firmware).
- **PCR 4:** Bootloader code and boot managers (e.g., systemd-boot/GRUB).
- **PCR 7:** Secure Boot state and platform certificates.
- **PCR 11:** Unified Kernel Image (UKI) measurement.

3. Machine Identity & Hardware-Backed X.509 Certificates

The TPM can act as a Hardware Security Module (HSM). The private key is generated and stored directly within the TPM silicon and can **never** be exported or extracted by malware, root compromises, or memory dumps. The system signs challenges and CSRs directly via the chip.

A. Generating an OpenSSL Key in TPM via `tpm2-openssl`

Using the modern OpenSSL 3.x TPM2 provider:

```
# Generate a hardware-backed RSA or ECC private key handle
openssl genpkey -provider tpm2 -algorithm EC -pkeyopt ec_paramgen_curve:prime256v1 -out
/etc/ssl/private/machine-tpm.key

# Generate a Certificate Signing Request (CSR) signed by the TPM
openssl req -provider tpm2 -provider default -new \
  -key /etc/ssl/private/machine-tpm.key \
  -out /etc/ssl/certs/machine.csr \
  -subj "/CN=bmo95.drheck.dev/O=DrHeck Infrastructure/"

# Issue or sign a certificate from your local CA against machine.csr
```

Use Cases for Hardware Machine Certs:

- **Mutual TLS (mTLS):** Machine-to-machine authentication where the client cert private key cannot be copied or exfiltrated.
- **802.1X Network Authentication:** RADIUS / EAP-TLS wired and Wi-Fi access tied to physical motherboard identity.
- **VPN Endpoints (WireGuard / IPsec / OpenVPN):** Authenticating servers and nodes with hardware-backed credentials.

4. Full Disk Encryption (LUKS Auto-Unlock)

Use `systemd-cryptenroll` to automatically unlock encrypted NVMe/SATA partitions at boot time via the TPM without entering a passphrase, while still requiring a PIN or measuring Secure Boot state:

```
# Enroll TPM 2.0 into an existing LUKS2 volume
sudo systemd-cryptenroll --tpm2-device=auto --tpm2-pcrs=0+2+7 /dev/nvme0n1p2

# Enroll with TPM 2.0 + PIN protection (requires both hardware TPM and a PIN)
sudo systemd-cryptenroll --tpm2-device=auto --tpm2-pcrs=0+2+7 --tpm2-with-pin=yes
/dev/nvme0n1p2

# List enrolled keys on the LUKS volume
sudo cryptsetup luksDump /dev/nvme0n1p2
```

Configure `/etc/crypttab` for automatic boot unlock:

```
# <name>          <device>                                <password> <options>
root_crypt        UUID=12345678-abcd-1234-abcd-123456789abc  none      tpm2-
device=auto,discard
```

5. Hardware-Backed SSH Keys

You can create SSH keys bound to the physical TPM chip so that SSH login credentials cannot be stolen from the filesystem.

Method A: Using FIDO2 / TPM Resident Keys (`ssh-keygen`)

If using OpenSSH 8.2+ with FIDO2/TPM integration:

```
# Generate hardware-backed ECDSA key
ssh-keygen -t ecdsa-sk -O resident -O verify-required -f ~/.ssh/id_ecdsa_sk
```

Method B: Using `tpm2-pkcs11`

Expose the TPM as a standard PKCS#11 cryptographic token for SSH client operations:

```
# Initialize PKCS11 store
tpm2_ptool init
tpm2_ptool addtoken --pid 1 --label "ssh-token" --sopin "adminpin" --userpin "userpin"
tpm2_ptool addkey --label "ssh-token" --userpin "userpin" --key-label "ssh-key" --algorithm
ecc

# Export public key for authorized_keys
ssh-keygen -D /usr/lib/libtpm2_pkcs11.so

# Authenticate via SSH using TPM hardware
ssh -I /usr/lib/libtpm2_pkcs11.so user@remote-host.lan
```

6. Hardware NVRAM Storage (Tamper-Evident Non-Volatile Memory)

The TPM contains dedicated non-volatile RAM (NVRAM) where small arbitrary blobs (keys, tokens, counters) can be written and locked with hardware-enforced read/write permissions.

```
# 1. Define an NVRAM index (e.g., index 0x1500016, 32 bytes)
tpm2_nvdefine 0x1500016 -C o -s 32 -a "ownerread|ownerwrite"

# 2. Write a 32-byte master seed directly to hardware NVRAM
echo -n "0123456789abcdef0123456789abcdef" | tpm2_nvwrite 0x1500016 -C o -i -

# 3. Read the secret back from hardware NVRAM
tpm2_nvread 0x1500016 -C o

# 4. Lock NVRAM index against any further writes (Read-Only)
tpm2_nvwriteunlock 0x1500016 -C o
```

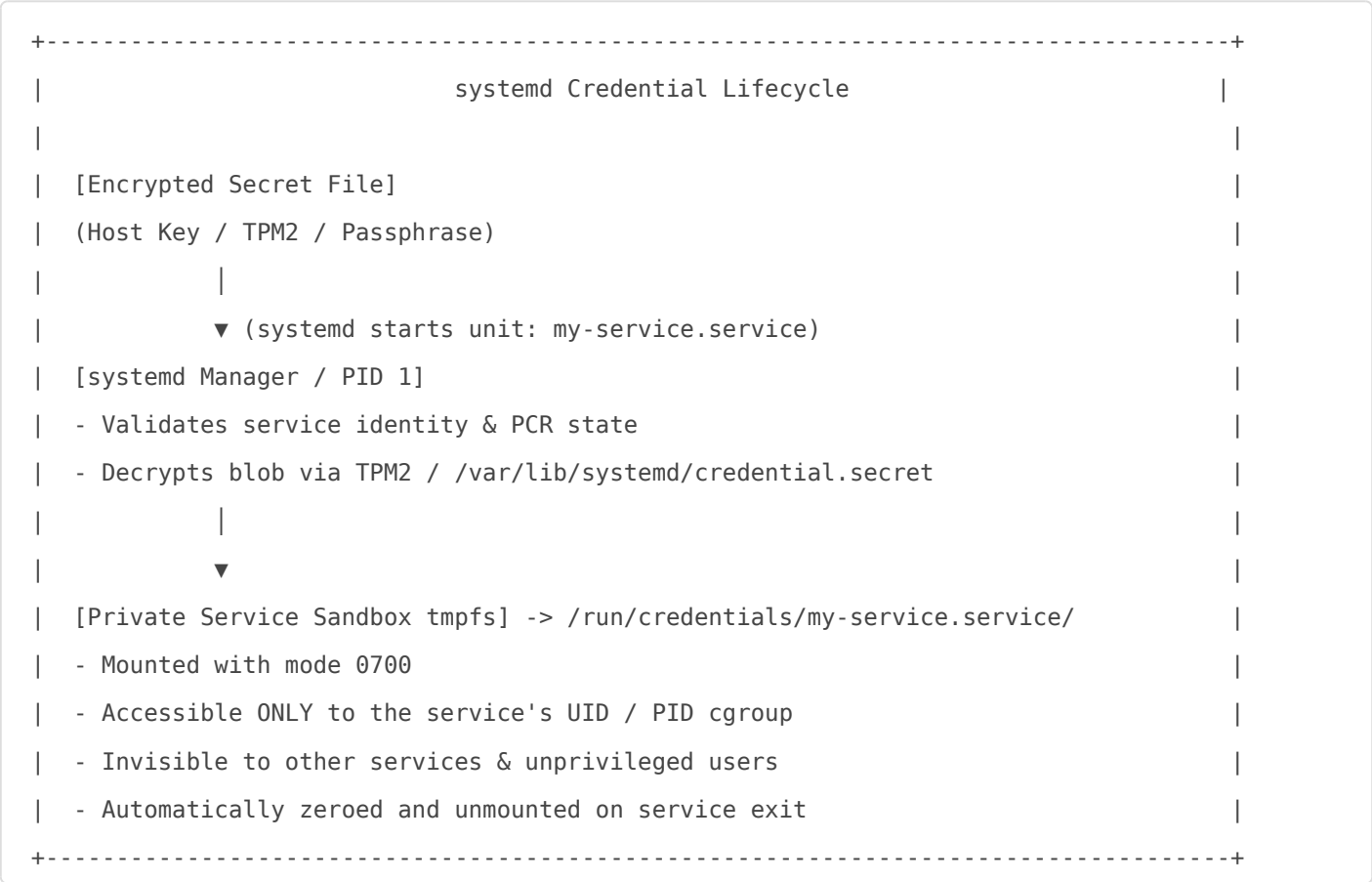
7. Hardware True Random Number Generation (TRNG)

The TPM includes an onboard physical True Random Number Generator based on hardware thermal noise. The Linux kernel feeds `/dev/tpm0` directly into `/dev/hwrng` and the kernel entropy pool (`/dev/urandom`).

```
# Fetch 64 bytes of cryptographically secure hardware entropy directly from TPM
tpm2_getrandom 64 --hex
```

Comprehensive Guide: Secret Handling in systemd

Modern `systemd` (v250+) provides a built-in, secure credential management framework (`systemd-creds`). It completely replaces insecure methods like embedding passwords in environment variables, plaintext files, or command-line flags.



1. Key Tiers & Encryption Modes

Secrets can be encrypted under different hardware and system bindings:

Flag	Encryption Key Source	Protection / Portability
<code>--with-key=host</code>	<code>/var/lib/systemd/credential.secret</code>	Bound to this specific OS installation.
<code>--with-key=tpm2</code>	Hardware TPM 2.0 silicon	Bound to this physical motherboard; cannot be copied off-chip.
<code>--with-key=host+tpm2</code>	Both Host secret file + TPM 2.0	Highest security: requires both physical machine hardware AND OS filesystem key.
<code>--with-key=tpm2 --tpm2-pcrs=...</code>	TPM 2.0 + Boot Integrity	Unseals only if firmware, kernel, and secure boot states match.

2. Service Isolation: Limiting Secrets to Specific Services

systemd can strictly restrict a secret so that **only a designated systemd service** can decrypt and read it.

Service-Name Binding (`--name=`)

When encrypting a credential, supply `--name=<service-name>`. systemd mixes the service name into the cryptographic key derivation:

```
# Encrypt a database password specifically for 'forgejo.service' bound to Host + TPM
echo -n "DatabasePasswordSecret99" | systemd-creds encrypt \
  --with-key=host+tpm2 \
  --name=forgejo.service \
  - /etc/credstore.encrypted/forgejo.db_password
```

- If `vaultwarden.service` or any other unit tries to load this credential file, decryption **fails** with an authorization error.

3. Supplying Secrets to Services

systemd supports multiple directives inside unit files (`/etc/systemd/system/<unit>.service`):

A. `LoadCredentialEncrypted=` (Recommended for Production)

Loads an encrypted file from disk, decrypts it at runtime in memory, and passes it to the private sandbox:

```
[Unit]
Description=Forgejo Git Service
After=network.target

[Service]
Type=simple
User=git
Group=git

# Loads encrypted file; systemd decrypts into $CREDENTIALS_DIRECTORY/db_password
LoadCredentialEncrypted=db_password:/etc/credstore.encrypted/forgejo.db_password

# Restrict service access (hardened sandbox)
ProtectSystem=strict
ProtectHome=yes
PrivateTmp=yes

ExecStart=/usr/bin/forgejo web --config /etc/forgejo/app.ini
```

B. `SetCredentialEncrypted=` (Embedded Encrypted Blobs)

Stores the encrypted payload directly inside the `.service` file or drop-in override (useful for GitOps and infrastructure-as-code):

```
# Generate the exact unit file snippet
echo -n "SuperSecretApiKey" | systemd-creds encrypt \
  --with-key=host+tpm2 \
  --name=syncthing.service \
  --unit \
  -
```

Output:

```
SetCredentialEncrypted=api_key: \
  AAAAERkBAAAAEAAAAA... \
  ...base64-blob...
```

Paste that directive directly under `[Service]` in `/etc/systemd/system/syncthing.service.d/override.conf`.

C. `LoadCredential=` (Plaintext File to Sandboxed tmpfs)

Passes an existing unencrypted file into the service's private `$CREDENTIALS_DIRECTORY`:

```
LoadCredential=db_password:/etc/keys/plaintext_password.txt
```

D. `SetCredential=` (Plaintext Literal)

Passes a plaintext literal string directly into the private sandbox:

```
SetCredential=db_user:postgres_admin
```

4. How Applications Consume systemd Secrets

When systemd launches the service:

1. It creates an isolated `tmpfs` directory accessible only by that service's UID/PID.
2. It populates the decrypted secret as a file: `$CREDENTIALS_DIRECTORY/<credential_name>` (typically `/run/credentials/<unit-name>/<credential_name>`).
3. It sets the environment variable `$CREDENTIALS_DIRECTORY` inside the process.

Consuming in Bash / Scripts:

```
#!/usr/bin/env bash
if [ -n "$CREDENTIALS_DIRECTORY" ] && [ -f "$CREDENTIALS_DIRECTORY/db_password" ]; then
    DB_PASSWORD=$(cat "$CREDENTIALS_DIRECTORY/db_password")
else
    echo "Credential not available!" >&2
    exit 1
fi

echo "Connecting with password length: ${#DB_PASSWORD}"
```

Consuming in Python:

```
import os
from pathlib import Path

creds_dir = os.environ.get("CREDENTIALS_DIRECTORY")
if creds_dir:
    db_password = Path(creds_dir, "db_password").read_text().strip()
else:
    raise RuntimeError("Missing $CREDENTIALS_DIRECTORY")
```

Consuming in Node.js:

```
const fs = require('fs');
const path = require('path');

const credsDir = process.env.CREDENTIALS_DIRECTORY;
if (!credsDir) {
    throw new Error("CREDENTIALS_DIRECTORY is not set");
}

const dbPassword = fs.readFileSync(path.join(credsDir, 'db_password'), 'utf8').trim();
```

5. CLI Inspection & Validation Commands

```
# Verify host key and TPM readiness
systemd-creds has-tpm2
systemd-creds list

# Inspect metadata of an encrypted credential (shows cipher, key type, service name,
timestamp)
systemd-creds inspect /etc/credstore.encrypted/forgejo.db_password

# Decrypt and view credential contents manually (requires root or matching key permissions)
systemd-creds cat --name=forgejo.service /etc/credstore.encrypted/forgejo.db_password

# Clean up memory / auto-rotated host keys
systemd-creds setup
```

6. Standard Credential Storage Locations

systemd automatically searches the following directories when `LoadCredentialEncrypted=name` (without a path) is used:

- `/etc/credstore.encrypted/` — System-wide encrypted credentials.
- `/etc/credstore/` — System-wide plaintext credentials.
- `/run/credstore.encrypted/` — Runtime encrypted credentials.
- `/run/credstore/` — Runtime plaintext credentials.
- `~/.config/credstore.encrypted/` — User-service encrypted credentials.
- `~/.config/credstore/` — User-service plaintext credentials.