Logging in to the FRITZ!Box Web Interface

FRITZ!Box Interfaces

FRITZ!OS offers a number of options for accessing the configuration of the FRITZ! Device, accessing statistics, and live data:

- The TR-064 protocol
- The UPnP IGD protocol
- Das graphic web interface

For third-party applications accessing the FRITZ!Box's configuration or live data, we strongly recommended the use of the standardized TR-064 or UPnP IGD protocols. Extensive documentation on both is available at www.avm.de/schnittstellen.

Access to the configuration via the HTTP interface, used by the FRITZ!Box web interface itself, is not recommended. Its parameters and functions are not documented, and AVM reserves the right to introduce breaking changes at any time as part of updates. If you still wish to connect using this method, the following document describes how to request a valid session ID.

Login Types From the Home Network

Prior to FRITZ!OS 7.24, there were three ways to log in to the FRITZ!Box web interface from the home network:

- With user name and password
- Only with FRITZ!Box password (factory settings, FRITZ!Box password on sticker on bottom of device)
- Without a password (not recommended)

With FRITZ! OS 7.24, the available login types have been reduced. On a technical level, a login from the home network now always requires a user name and a password. Additional information and recommendations on user dialogs for the login process can be found at https://avm.de/schnittstellen in the document “Recommendations for User Guidance When Logging Into a FRITZ! Box”.

When logging in from the Internet, only login with a user name and password is supported. This was already the case in versions prior to 7.24.

Entry Page for Third-Party Applications

Starting with FRITZ!OS 5.50, information on the login procedure used and the assignment of the session ID is provided via HTTP GET on the entry page

```
https://fritz.box/login_sid.lua?version=2
```

Please note: If the page cannot be loaded, even though the FRITZ!Box is online, the firmware in use is too old to support session IDs.

Depending on the POST parameters transferred, the following actions can be performed:

1) Perform login.
   Parameters:
      "username" (name of user, or empty),
      "response" (a response generated from the password and challenge)

2) Check whether a certain session ID is valid.
   Parameters:
   "sid" (the session ID to be checked)

3) Logout, i.e., invalidate a session ID.
   Parameters:
   "logout", "sid" (a valid session ID)

The XML response always follows the same structure:

```
<SessionInfo>
      <SID>8b4994376ab804ca</SID>
      <Challenge>a0fa42bb</Challenge>
      <BlockTime>0</BlockTime>
      <Users>
            <User last=1>fritz1337</User>
      </Users>
      <Rights>
            <Name>NAS</Name>
            <Access>2</Access>
            <Name>App</Name>
            <Access>2</Access>
            <Name>HomeAuto</Name>
            <Access>2</Access>
            <Name>BoxAdmin</Name>
            <Access>2</Access>
            <Name>Phone</Name>
            <Access>2</Access>
      </Rights>
</SessionInfo>
```

Die number, and order, or values is variable. Only the rights actually assigned to the current session ID are returned.

Contents of the values:

- <SID>: If this value consists of only zeroes, no rights are granted. Login is required. Otherwise, you receive a valid session ID for further access to the FRITZ!Box. The current access rights can be viewed in the <Rights> area
- <Challenge>: Contains the challenge that can be used to log in using the challenge-response method.
- <BlockTime>: Time in seconds during which no further login attempt is allowed.
- <Users>: : A list of all users. Only available in the home network, if activated. The attribute “last = 1” marks the user who was last logged in, otherwise “last = 0”.
- <Rights>: The individual rights granted to the current session ID. Possible values are 1 (read only) and 2 (read and write access).

Obtaining a Session ID

As first part of a login process, a valid session ID must be generated. For security reasons the password is not sent in clear text during login, but via a challenge-response authentication.

Calculating the Response Value

PBKDF2 (FRITZ!OS 7.24 and later)

With FRITZ!OS 7.24 or later, a client can request the modern PBKDF2-based challenge-response procedure by appending `version=2` to the initial GET request. If the <Challenge> starts with "`2$`", PBKDF2 is supported by the FRITZ!OS version. Otherwise a fallback to MD5 is possible. Third-party applications are recommended to support PBKDF2, as this means users' passwords are better protected against attacks.

The `<challenge>` is retrieved from the entry page `login_sid.lua?version=2`.

The format of the challenge is defined as follows, separated by `$` signs:

```
2$<iter1>$<salt1>$<iter2>$<salt2>
```

The response is formed as follows:

```
<hash1> = pbdkf2_hmac_sha256(<password>, <salt1>, <iter1>)
<response> = <salt2>$ + pbdkf2_hmac_sha256(<hash1>, <salt2>, <iter2>)
```

The example challenge `"2$10000$5A1711$2000$5A1722"` and the password `"1example!"` (utf8-encoded) results in:

```
hash1 = pbdkf2_hmac_sha256("1example!", 5A1711, 10000)
=> 0x23428e9dec39d95ac7a40514062df0f9e94f996e17c398c79898d0403b332d3b (hex)

response = 5A1722$ + pbdkf2_hmac_sha256(hash1, 5A1722, 2000).hex()
=> 5A1722$1798a1672bca7c6463d6b245f82b53703b0f50813401b03e4045a5861e689adb
```

Note: the `hash1` is not stringified, but remains a raw bytes value.
The numbers `iter1` und `iter2` are the "iteration" parameters for PBKDF2 of the 1. und 2. round, respectively. These are kept variabel and may change in future releases of FRITZ!OS, in order to PBKDF2-based login future-proof.

Due to the double hashing, each with a different salt, a part of the answer can be statically precalculated in FRITZ!OS, making an attack more difficult, but ensuring the login for users can remain as fast as usual.

The login process now takes place via a POST request to login_sid.lua? Version = 2 with „`response`“ as parameter.

The response XML from the FRITZ!Box will now contain the valid SID.

Example code can be found below, see `calculate_pbkdf2_response`.

MD5

If the challenge does not start with `2$`, because it's an older FRITZ!OS version, or if the request is sent to login_sid.lua without the version=2 parameter, the MD5 digest method is used (FRITZ!OS 5.50 or later).

The response value is determined from the plain text password and a challenge as follows:

`<challenge>` is read from the entry page, login_sid.lua?version=2.

`<md5>` is the MD5 of `(<challenge>-<password>`) in 32 hexadecimal chars in lower case)

The MD5 hash is generated from the byte sequence of the UTF-16LE coding of this string (without BOM and without terminating 0 bytes).

For reasons of compatibility, the coding of the "." character (0x2e 0x00 in UTF-16LE) must be used for every character that has the unicode codepoint > 255. This therefore affects all characters that cannot be displayed in ISO-8859-1, e.g. the euro symbol.

Example with a German umlaut:

The challenge

```
<challenge> = "1234567z"
```

Combined with the password

```
<password> = "äbc"
```

yields the value

```
<response> = "1234567z-9e224a41eeefa284df7bb0f26c2913e2"
```

The login process now takes place via a POST request to login_sid.lua? Version = 2 with "response" as a parameter.

The FRITZ!Box response XML will now contain the valid SID.

Example code can be found below, in the function `calculate_md5_response`.

Using the Session ID

The session ID is a 64-bit number rendered by 16 hexadecimals. It is assigned at login and must be carried along for the duration of the session. A program should use only one session ID for each FRITZ!Box, since only a restricted number of sessions to each FRITZ!Box is permitted.

Once it has been assigned, a session ID is valid for 20 minutes. The validity is extended automatically whenever access to the FRITZ!Box is active.

The session ID 0 (`0000000000000000`) is always invalid.

The session ID is passed in the parameter "sid".

Access Without a Session ID

In principle, all dynamically generated pages can be opened only with a valid session ID. Even reading or writing web variables requires a session ID.

The following contents can be opened without a valid session ID:

- Entry pages (for example the login page)
- Static content (i.e. graphics)

Ending a Session

A session can be ended at any time by deleting the session ID, even before the automatic 10-minute timeout kicks in.

This is done by calling up the login_sid.lua?version=2 with the parameters "logout" and the current SID.

Example Code to Retrieve a Session-ID

Example Code: Python3

```python
#!/usr/bin/env python3
# vim: expandtab sw=4 ts=4

"""
FRITZ!OS WebGUI Login

Get a sid (session ID) via PBKDF2 based challenge response algorithm.
Fallback to MD5 if FRITZ!OS has no PBKDF2 support.
AVM 2020-09-25
"""

import sys
import hashlib
import time
import urllib.request
import urllib.parse
import xml.etree.ElementTree as ET

LOGIN_SID_ROUTE = "/login_sid.lua?version=2"


class LoginState:
    def __init__(self, challenge: str, blocktime: int):
        self.challenge = challenge
        self.blocktime = blocktime
        self.is_pbkdf2 = challenge.startswith("2$")


def get_sid(box_url: str, username: str, password: str) -> str:
    """ Get a sid by solving the PBKDF2 (or MD5) challenge-response
process. """
    try:
        state = get_login_state(box_url)
    except Exception as ex:
        raise Exception("failed to get challenge") from ex

    if state.is_pbkdf2:
        print("PBKDF2 supported")
        challenge_response = calculate_pbkdf2_response(state.challenge,
password)
    else:
        print("Falling back to MD5")
        challenge_response = calculate_md5_response(state.challenge,
password)

    if state.blocktime > 0:
        print(f"Waiting for {state.blocktime} seconds...")
        time.sleep(state.blocktime)

    try:
        sid = send_response(box_url, username, challenge_response)
    except Exception as ex:
        raise Exception("failed to login") from ex
    if sid == "0000000000000000":
        raise Exception("wrong username or password")
    return sid


def get_login_state(box_url: str) -> LoginState:
    """ Get login state from FRITZ!Box using login_sid.lua?version=2 """
    url = box_url + LOGIN_SID_ROUTE
    http_response = urllib.request.urlopen(url)
    xml = ET.fromstring(http_response.read())
    # print(f"xml: {xml}")
    challenge = xml.find("Challenge").text
    blocktime = int(xml.find("BlockTime").text)
    return LoginState(challenge, blocktime)


def calculate_pbkdf2_response(challenge: str, password: str) -> str:
    """ Calculate the response for a given challenge via PBKDF2 """
    challenge_parts = challenge.split("$")
    # Extract all necessary values encoded into the challenge
    iter1 = int(challenge_parts[1])
    salt1 = bytes.fromhex(challenge_parts[2])
    iter2 = int(challenge_parts[3])
    salt2 = bytes.fromhex(challenge_parts[4])
    # Hash twice, once with static salt...
    hash1 = hashlib.pbkdf2_hmac("sha256", password.encode(), salt1, iter1)
    # Once with dynamic salt.
    hash2 = hashlib.pbkdf2_hmac("sha256", hash1, salt2, iter2)
    return f"{challenge_parts[4]}${hash2.hex()}"


def calculate_md5_response(challenge: str, password: str) -> str:
    """ Calculate the response for a challenge using legacy MD5 """
    response = challenge + "-" + password
    # the legacy response needs utf_16_le encoding
    response = response.encode("utf_16_le")
    md5_sum = hashlib.md5()
    md5_sum.update(response)
    response = challenge + "-" + md5_sum.hexdigest()
    return response


def send_response(box_url: str, username: str, challenge_response: str) ->
str:
    """ Send the response and return the parsed sid. raises an Exception on
error """
    # Build response params
    post_data_dict = {"username": username, "response": challenge_response}
    post_data = urllib.parse.urlencode(post_data_dict).encode()
    headers = {"Content-Type": "application/x-www-form-urlencoded"}
    url = box_url + LOGIN_SID_ROUTE
    # Send response
    http_request = urllib.request.Request(url, post_data, headers)
    http_response = urllib.request.urlopen(http_request)
    # Parse SID from resulting XML.
    xml = ET.fromstring(http_response.read())
    return xml.find("SID").text


def main():
    if len(sys.argv) < 4:
        print(
            f"Usage: {sys.argv[0]} http://fritz.box user pass"
        )
        exit(1)

    url = sys.argv[1]
    username = sys.argv[2]
    password = sys.argv[3]
```

```python
    sid = get_sid(url, username, password)
    print(f"Successful login for user: {username}")
    print(f"sid: {sid}")


if __name__ == "__main__":
    main()
```

Example Code: Java

```java
import java.nio.charset.StandardCharsets;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

public class Pbkdf2Login {

    /**
     * Calculate the secret key on Android.
     */
    public static String calculatePbkdf2Response(String challenge, String
password) {
        String[] challenge_parts = challenge.split("\\$");
        int iter1 = Integer.parseInt(challenge_parts[1]);
        byte[] salt1 = fromHex(challenge_parts[2]);
        int iter2 = Integer.parseInt(challenge_parts[3]);
        byte[] salt2 = fromHex(challenge_parts[4]);
        byte[] hash1 = pbkdf2HmacSha256(password.getBytes(Standard-
Charsets.UTF_8), salt1, iter1);
        byte[] hash2 = pbkdf2HmacSha256(hash1, salt2, iter2);
        return challenge_parts[4] + "$" + toHex(hash2);
    }

    /** Hex string to bytes */
    static byte[] fromHex(String hexString) {
        int len = hexString.length() / 2;
        byte[] ret = new byte[len];
        for (int i = 0; i < len; i++) {
            ret[i] = (byte) Short.parseShort(hexString.substring(i * 2, i *
2 + 2), 16);
        }
        return ret;
    }

    /** byte array to hex string */
    static String toHex(byte[] bytes) {
        StringBuilder s = new StringBuilder(bytes.length * 2);
        for (byte b : bytes) { s.append(String.format("%02x", b)); }
        return s.toString();
    }

    /**
     * Create a pbkdf2 HMAC by appling the Hmac iter times as specified.
     * We can't use the Android-internal PBKDF2 here, as it only accepts
char[] arrays, not bytes (for multi-stage hashing)
     */
    static byte[] pbkdf2HmacSha256(final byte[] password, final byte[]
salt, int iters) {
        try {
            String alg = "HmacSHA256";
            Mac sha256mac = Mac.getInstance(alg);
            sha256mac.init(new SecretKeySpec(password, alg));
            byte[] ret = new byte[sha256mac.getMacLength()];
            byte[] tmp = new byte[salt.length + 4];
            System.arraycopy(salt, 0, tmp, 0, salt.length);
            tmp[salt.length + 3] = 1;
            for (int i = 0; i < iters; i++) {
                tmp = sha256mac.doFinal(tmp);
                for (int k = 0; k < ret.length; k++) { ret[k] ^= tmp[k]; }
            }
            return ret;
        } catch (NoSuchAlgorithmException | InvalidKeyException e) {
            return null; // TODO: Handle this properly
        }
    }
}
```