quick-and-dirty PAM with LUA, mod_magnet and lighttpd -or- how to breach system security

Be warned: This example serves as an illustration on how to *NOT* do it.
It's just one of my examples I teach to apprentices at the office when it comes to learning scrips, and how important data input validation (or the absence of the same) is.
It's also a good illustration on how attackers may break into systems to steal data or make them part of a botnet.

The given situation depicts a lighttpd server, which exposes a directory which must be protected via LDAP-managed accounts, so there is an immediate need for PAM. However, lighttpd lacks a PAM implementation. Period. There's a very ugly and highly insecure way however ...

The procedure includes enabling mod_magnet in lighttpd to allow for some simple lua scripting, as documented on the lighttpd manual.

After enabling mod_magnet, a URL can be directed to a given lua script like this:

```
$HTTP["url"] =~ "/somedirectory(/.*)?" {
        magnet.attract-raw-url-to = (
                "/path/to/luapam.lua"
        )
}
```

Now, the actual script, which I took in parts from absLUAtion, looks as shown below.

```
require("mime")

--
-- send HTTP auth header
--
function sendAuthHeader()
 lighty.header["WWW-Authenticate"] = string.format('Basic realm="%s"', lighty.env["uri.authority"])
 return 401
end

--
-- authenticate against control panel
--
function checkAuthPAM(user,pass)
 cmdstr = string.format('/usr/bin/printf "%%sn%%sn" %s %s | /usr/local/bin/pwauth', user, pass)
 exitcode = os.execute(cmdstr)

 if exitcode == 0 then
  return true
 else
  return false
 end
end
```

```
-- MAIN
--
-- check for auth header
--
if (not lighty.request.Authorization) then
 --
 -- enforce authentication if no auth header found
 --
 return sendAuthHeader()
end

--
-- auth header found, try to extract base64 encoded username and password
--
_, _, http_basic_auth_data = string.find(lighty.request.Authorization, "Basic%s+(.+)$")
if (not http_basic_auth_data) then
 --
 -- enforce authentication if auth header invalid or incomplete
 --
 return sendAuthHeader()
end



--
-- decode basic auth data using base64
-- split playload by : (colon) to get single username and password fields
--
username_password = mime.unb64(http_basic_auth_data)
_, _, username, password = string.find(username_password, "(.+):(.+)$")



--
-- try to authenticate user against backend
--
if (not checkAuthPAM(username, password)) then
 --
 -- enforce authentication if auth header invalid or incomplete
 --
 return sendAuthHeader()
end



--
-- return to proceed normal operation
-- (end up here only upon successful authentication)
--
return
```

All it does is to decode the username and password from the auth data.

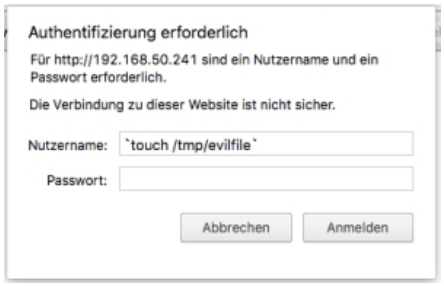
The key to mayhem is then to PIPE the decoded username and password into the **pwauth** utility, which in turn performs the actual PAM authentication.
Okay, while this surely does the job, it has some major issues and drawbacks:

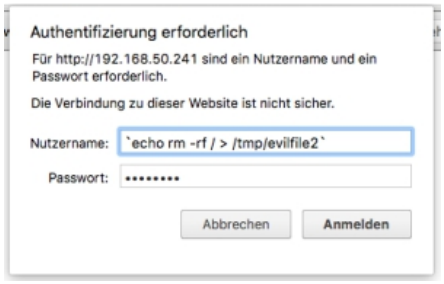
- The input is passed through the shell, and as such, allows for shell escaping in the worst of all nightmares
- Each request is sent to the authenticator, which means, you can easily shell bomb a host by just sending enough requests
- Besides that, it doesn't scale out
- Did I mention it's insecure as hell?

The reason why I chose this example was also to show, how a quick and dirty solution, as well-meant it may have been, may lead to unprectitable results and put a system to danger.

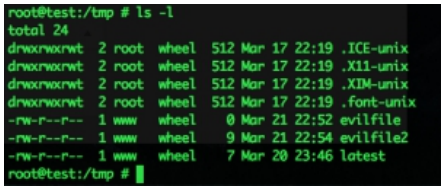
The danger in this case, the sole absence of input validation, opens a huge door to the system. As long as good and reasonable input is received, everything is fine. But what happens if someone provides a typical *nix command instead of a supposed username?

The nasty backtick operator allows command injection, so let's see two examples:





So, two files have been created in the systems /tmp directory. Surely not, what the author of the authenticator script intended:



The content of the second file created on the host:

So, this illustrates how easily remote-holes can be opened by omitting input validation. Think about it twice! What would have happened, if an attacker had actually ran this second file? The whole contents of the hard disk would be just erased!Assuming any malicous attacker could exploit this, it's only a matter of choice to delete or extract any data. Or the worse, install some malware, and turn an innocent server into a bot of a much larger botnet. This in turn could then serve as a legion to attack others.And this is exactly, how it works.
So, how to make the script a little more secure then? By just adding input validation. Since we want only usernames and passwords, a set of valid characters must be defined. Then, these can easily be validated using regurlar expressions as in the following example, where only alphanumeric characters are accepted for usernames, while the password accepts a wider variety of valid characters

```
rex = require("rex_pcre")

...

function checkAuthPAM(user,pass)
	if user == nil or not rex.match(user, '[a-z0-9]+$') or pass == nil or rex.match(pass, '[a-zA-Z0-9-_:.#!]$') then
		return false
	end

...
end
```

Still, this is not the ultimate solution yet, but way better than before.
Yet it's important to note that, when a system urges you to do a "baschtel" (swiss german word for quick and dirty, evil and ugly hackery), it may be better to replace it.
So, in this example, lighttpd is hindering the proper use of PAM. So, why not scrap it an use nginx or Apache instead? Maybe it's better to invest these extra 10 minutes to get a proper and well-working solution, instead of doing a hack, which endangers the whole system?

Just my 2 cents.