

2/c Christian Johnson, 2/c Joseph Church, 1/c Ryan Younes

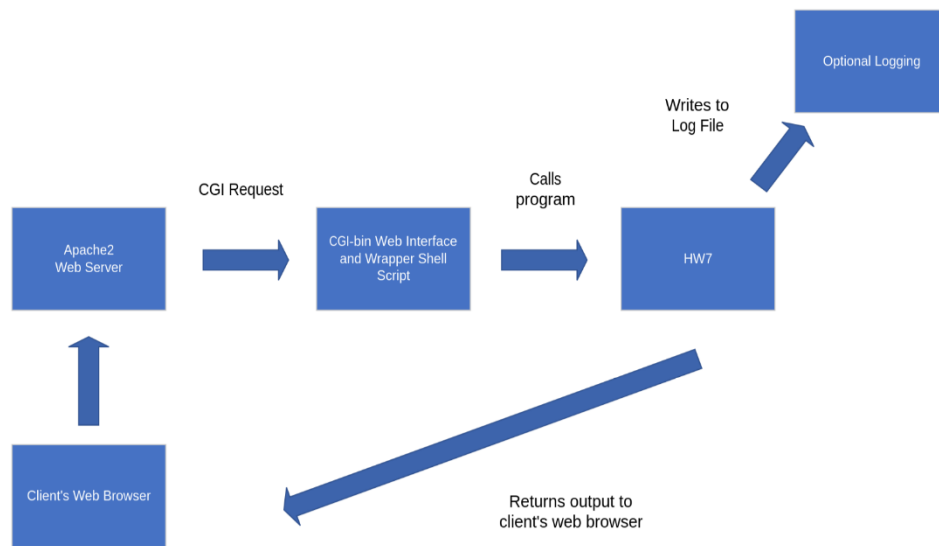
Operating Systems

Dr. Elwakil, Mr. Gold

29 Nov 2023

Vulnerability Assessment on Default Apache Webserver Configuration

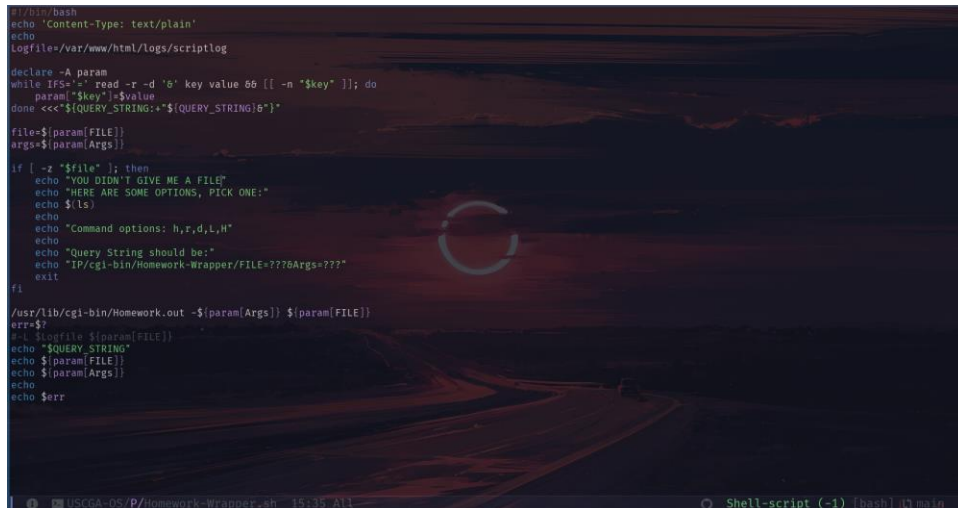
The Operating Systems Final Project builds upon all previous HW assignments and labs. In our system, the Apache2 web server is configured to receive HTTP client requests which are passed to the CGI-bin interface. The CGI-bin interface is configured to execute HW7 with options specified by the client (print, delay, log, reverse output, etc.). It then returns the output to the client's web browser. The CGI bin interface is necessary because the Apache2 web server is unable to execute code independently; CGI bin acts as a bridge between the web server and HW7. A diagram of the system is attached below:



This setup allows us to utilize shell commands in order to expand our webserver's capabilities. This is useful in many situations and allows the server host to present a much wider array of site functionalities, but it can also open the host machine to potential exploits and vulnerabilities. In most cases, the Apache server is designed to mitigate potential risk on its own, and the default configuration prevents many possible vulnerabilities. In our case however, we

have enabled CGI-bin functionality. This is inherently insecure, since it allows external users to run pre-defined scripts on the host machine. If the host does not make insecure decisions, and limits the capabilities of the scripts served through CGI-bin, then security should not be an issue. In our case however, the Homework script we have allowed our server to access has several security implications.

While the CGI-bin functionality dramatically expands a web-server's capabilities, it also introduces significant vulnerabilities. Through our testing, we have identified several such vulnerabilities in our own code. Initially, following the basic setup for Apache; we weren't able to find many errors. Our server was configured to serve a single script, which ran our homework function with a predefined set of arguments and a predefined file. Essentially, our webpage would display the exact same thing every time, since the script ran the exact same command every time. As we sought to expand this functionality however, we began finding ways to pass arguments to the wrapper script using the webpage URL. We utilized the QUERY_STRING variable to save a specific set of key value pairs in the URL and pass those through the wrapper script to the Homework C program. This is shown in the image below.



```
#!/bin/bash
echo 'Content-Type: text/plain'
echo
logfile=/var/www/html/logs/scriptlog

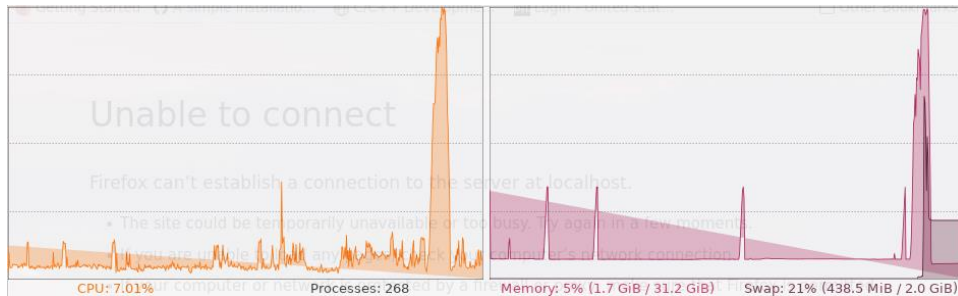
declare -A param
while IFS="=" read -r -d ' ' key value && [[ -n "$key" ]]; do
    param["$key"]="$value"
done <<< "${QUERY_STRING:+${QUERY_STRING#&}}"

file=${param[FILE]}
args=${param[Args]}

if [ -z "$file" ]; then
    echo "YOU DIDN'T GIVE ME A FILE!"
    echo "HERE ARE SOME OPTIONS, PICK ONE:"
    echo $(ls)
    echo
    echo "Command options: h,r,d,L,H"
    echo
    echo "Query String should be:"
    echo "IP/cgi-bin/Homework-Wrapper/FILE=???&Args=???"
    exit
fi

/usr/lib/cgi-bin/Homework.out -${param[Args]} ${param[FILE]}
err=$?
echo $(cat $logfile ${param[FILE]})
echo "QUERY_STRING"
echo ${param[FILE]}
echo ${param[Args]}
echo
echo $err
```

This setup allowed for a greater degree of flexibility, in that we had configured the script to take 2 arguments now, allowing the user to specify a file and a set of arguments to pass to the C program. What we failed to consider, however, was specifying a scope for the files available to the user. Typically, Apache limits the file that the user can access through the server, but since our script was set up to access the host filesystem, it was not doing so through Apache's file permissions (essentially the user could access any file that was accessible to root) - we demonstrated this vulnerability by using the webpage to access the contents of /etc/shadow, a file that contains sensitive information for the users of the host machine. Later we discovered that we were also able to pass the -m option to the C program. This option allows the user to specify an amount (in Mb) of memory to allocate to the running process. We were able to use this option to repeatedly create a process that was allocated a high amount of memory, overloading the resources allocated to the www-data user and crashing the apache2 process (thus temporarily shutting down the webserver). This is shown in the image below, the spike in memory and CPU usage indicating the instance when the server crashed.



These issues are concerning on their own, since we do not want external users to be able to crash the server on demand, but Apache does provide us tools to help prevent these issues. There are a variety of built-in configuration options that allow the host to set limits on the server process and its children, as well as basic rules that allow for a rudimentary firewall. These options are part of what prevented the memory attack from slowing down the entire computer since (A.) the operating system limits the resources available to each user (www-data Apache also ships with a module system that allows it to expand its functionality, which can open the door for more advanced configuration as well. Below, we have outlined several categories that Apache allows the user to configure to limit server vulnerabilities.

a. Process limits and protections / Memory limits and protections

- Apache provides configuration options to limit processes and memory usage. By default, they are set to something logical with options like Timeout 300, which sets a maximum of 300 seconds to wait for a slow request before it is released, MaxKeepAliveRequests 100 which limits the number of persistent connections that can exist at a time, and KeepAliveTimeout 5, which limits the amount of time to wait for these persistent connections to wait for a new request.
- These appear to be all of the default settings, but there are other non-default options that may be helpful, such as MaxRequestsPerChild, which sets the number of requests that each child of an Apache process can handle at a time. This can help throttle network traffic if the process is further forked, and therefore keep this vector for attacks such as DoS at bay.

b. CPU limits ***

- The Apache 2 Webserver consumes many CPU resources under heavy load. Although it is not possible to limit CPU usage of Apache directly from its settings, many modules and apps exist to remedy this. The RLimitCPU directive can be used to limit the amount of CPU time that a process can use. The mod_cgi configuration can be used to set the maximum time that a CGI script is allowed to run. The mod_bw module allows you to limit the bandwidth usage and, indirectly, the CPU usage based on criteria including IP address, virtual host, or directory. The nice and renice commands can also be used to adjust the priority of Apache processes. Additionally, CPU usage can be monitored using a program such as Task Manager.

c. Network limits

- The Apache web server is vulnerable to network-based attacks including Denial-of-Service (DoS) attacks. Mod_cband is an Apache 2 module which provides bandwidth quotas and throttling, possibly preventing a DoS attack. Additionally, Apache2 hosted on the Linux machine can be configured with iptables or ufw (uncomplicated firewalls) to create rules to deny traffic on specific ports. This includes ports typically used by Apache including port 80 for HTTP and port 443 for HTTPS traffic.
- The mod_authz_host module could also be configured to provide access control based on hostnames, IP addresses, and other characteristics of client requests.

d. File security risks and limits/credential issues

- Apache seems to handle credentials with the assumption that whoever set up the server knew what they were doing. You can't use the URL to navigate outside of specified directories, and Apache can only access folders that it owns (set using chown www-data ...) but within directories for which these specifications are true, you seem to have full access.
- Credential settings aren't really applicable to our situation, since that information is all handled on the server side. If you do something that could affect security (like 'sudo chown www-data /') then there are going to be issues, since the server will have access to files that it shouldn't, but barring that there doesn't seem to be many options that are applicable to this project.

e. Possible risks due to programming errors

- In our original modification of the wrapper script from HW7 to accept client specified options (ex. -d), the query string was split with a "&" symbol. If the client entered a URL followed by a '&' and a string, the content preceding the '&' would execute as intended, but the string following the '&' would also run as a separate, independent Linux bash command. This presented a potential security vulnerability exploitable by an attacker, potentially allowing the client to run arbitrary bash commands on the host system. We fixed this by first removing the '&' in the wrapper script, and second, parsing the URL for arguments instead of parsing the URL as a command itself.

f. Other aspects of system fragility

- If the server fails or goes down unexpectedly, the absence of server backup and recovery options make the system potentially vulnerable to prolonged downtime. When we crashed the server manually, we were forced to manually reactivate the apache2 service with the command 'sudo systemctl start apache2'.
- The Apache2 Web Server must be up to date. Old versions of Apache are suspect to vulnerabilities including CVE-2017-9798 (Apache HTTP Server 2.4.17 to 2.4.23) that allows attackers to execute arbitrary code or cause a denial of server (DOS) via server-side request forgery in conjunction with a forwarding proxy OR CVE-2015-3183

(Apache HTTP Server 2.2.0 to 2.2.24, 2.4.0 to 2.4.10) that allowed attacks to execute arbitrary code with the privileges of the Apache HTTP Server.